Jonas Greutert

Abstraction and Implementation of predictable Packet Processing Systems

History

Date	Version	Remarks
2005-10-21	1.0	First complete Version, reviewed and corrected.

Abstract

Computers and other electronic devices are increasing connected with each other. Besides computers and servers, small and low-cost embedded systems, from simple sensor/actor devices to more complex control units, are added to networks.

While raw throughput is the main concern for most elements and applications in a network, there is a class of increasingly used devices with a different objective, namely predictable behavior. For these devices the absolute number of packets processed is less important than that specific packets are guaranteed to be processed within set limits.

Today, the most important issues when developing these devices are:

- The computational capacity of embedded systems has increased at a much lower pace than the bandwidth of the network: The computational capacity represents a bottleneck. It does not have the capacity to process all arriving packets at full line-speed. This problem will even increase in the future.
- It is difficult to model such systems with traditional real-time methods, as the input is unknown; we do not know when, how fast and in which order packets arrive at the system.
- As there is no analytical model for such systems, it is difficult to determine the required hardware performance. However to develop hardware with the required functionality at reasonable and competitive cost, it is necessary to determine the needed hardware performance with an analytical model. Without an analytical model and to stay on the safe side, we would be compelled to overbuild the hardware, which would lead to uncompetitive hardware costs.

In this thesis we present a method that allows the development and implementation of predictable packet processing systems on low-cost hardware. The main components are:

- A model for low-cost packet processing systems that can be adjusted according to the intended use. Using this model we can analyze and explore the system properties and determine what hardware performance is required.
- A software platform that allows to transform seamlessly the model to an implementation. The result is a predictable packet processing system. The practical and analytical results match closely.

Kurzfassung

Contents

1 Introduction 1-1

1.1 Problem Statement 1-2

1.1.1 Review of a Typical Embedded Low-Cost System 1-3

1.1.2 Summary of Problem Statement 1-6

1.2 Related Work 1-7

1.3 Target and Results of Thesis 1-10

1.4 Outline 1-11

2 Model 2-1

2.1 Demands on Model 2-1

2.2 Design of Model 2-5

2.2.1 Input Model 2-5

2.2.2 Application Model 2-15

2.2.3 Mapping Input to Application 2-22

2.2.4 Resource Model 2-25

2.2.5 Mapping Application to Resources 2-28

2.3 Summary 2-28

3 Analysis 3-1

3.1 Basic Calculations 3-2

3.2 Calculation Scheme 3-5

3.3 Exploration 3-9

3.3.1 Example System 3-10

3.4 Admission Control 3-16

3.5 Summary 3-16

4 Mapping to Implementation 4-1

4.1 From Application Model to Implementation Model 4-2

4.1.1 Task Instances 4-3

4.1.2 Annotation of Instance Information 4-13

4.2 Scheduler and Path-Threads 4-16

4.3 Source Flow 4-17

4.4 Summary 4-19

5 Software Platform RNOS 5-1

5.1 Elements of RNOS 5-3

5.1.1 Tasks and Task Graphs 5-4

5.1.2 Packets 5-13

5.1.3 Flows and Path-Threads 5-19

5.1.4 Source Flows and Source-Thread 5-22

5.1.5 Scheduler 5-24

5.1.6 Summary 5-27

5.2 RNOS Integration with the RTOS 5-30

5.3 RNOS: A higher Level Programming System 5-31

5.4 Advanced Features of RNOS 5-33

5.4.1 Virtual Tasks 5-33

5.4.2 Instrumentation 5-35

5.5 Schedulability Region of RNOS 5-36

5.5.1 Overhead 5-36
5.5.2 Throughput 5-39
5.5.3 Delay 5-42
5.5.4 Conclusion 5-56
5.6 Analysis with Network Calculus 5-57
5.6.1 Example Analysis: Worst-Case Packet Delay 5-59
5.7 Summary 5-64

6 Example: Implementation & Measurements 6-1
6.1 System Description 6-2
6.1.1 Hardware Platform 6-2
6.1.2 Real-Time Operating System 6-3
6.1.3 Service Curve 6-5
6.1.4 Application 6-6
6.1.5 RNOS Attributes 6-8
6.1.6 Scheduleability Region of Example Implementation 6-18
6.2 Scenarios 6-20
6.3 Summary 6-23

7 Conclusion 7-1

1 Introduction

As network infrastructure is becoming more widely available at reasonable cost, the number of applications using it is rapidly increasing. Packet processing takes place in all components of the network infrastructure and in all related applications. Packet processing can be defined as a set of tasks that are performed on a packet within a system from the time a packet is received or created, until it is transmitted or consumed.

While for most elements and applications in a network the raw number of packets that can be processed per second is the main concern, there is a new class of devices with different objectives. Small embedded devices, that have simple architectures and low performance, are deployed in large numbers. These are gateways of any type and small sensor/actuator devices connected to a network that provides a variety of services. Although these devices do not have a high packet processing capacity, they are often required to process specific "VIP" (Very Important Packets) packets in real-time. That processing deadlines for specific packets are kept is more important than total packet throughput. Just being faster is less efficient than giving certain predefined packets preferential treatment. Typically, these devices are low cost and are built around a standard communication controller, i.e. they do not contain a highly specialized network processor. Packet processing is usually only a part of their task, although critical with respect to predictability.

This thesis focuses on the analysis, exploration and implementation of the packet processing part in low cost embedded devices with the requirement of predictable behavior.

1.1 Problem Statement

A rule of thumb says that you need a 2 MHz CPU to process a 1 MBit/s packet stream. Today, the widely used physical line rate is 100 MBit/s and interfaces with a physical line rate of 1 GBit/s are currently being adopted by PCs. On the other hand, typical embedded systems with network attachment contain a communication controller running from 20 MHz to 100 MHz. Therefore, low cost embedded systems are not capable of processing the full line rates. Section 1.1.1 describes a typical embedded low cost system and provides some measurement results.

A typical requirement is that a system is capable of processing a certain packet rate within a timeframe. Other packets that arrive at an input are processed as available resources allow. However, as the arrival time of packets is unknown, it is very difficult to model and implement systems that can give hard real-time guarantees. Traditional methods of real-time computing are difficult to apply, as the arrival of input and the

availability of resources are not known in detail. Probabilistic methods do not give hard real-time guarantees and are best used to get results about average behavior. We need a model that is able to capture the unknown input and resources to calculate the processing requirements. This model must allow for the analysis and exploration of systems as well as provide a base for an efficient implementation.

1.1.1 Review of a Typical Embedded Low-Cost System

Typical embedded communication controllers consist of a core CPU, a bus interface unit and several physical interfaces, e.g. serial or Ethernet interfaces. A DMA controller takes care of transporting the data to and from the interfaces via bus interface unit to an external memory. The core CPU processes data packets in the external memory. Therefore, the core CPU and the DMA controller share the external memory. The bus interface unit has a bus arbitration protocol that controls the access to the external bus and memory. An example block diagram of such an embedded communication controller is shown in Figure 1-1.

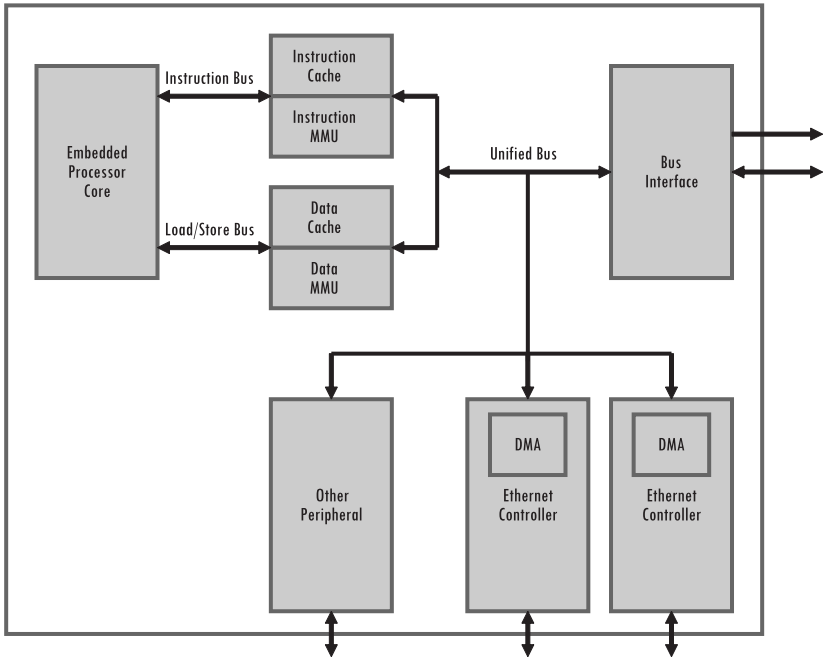

Figure 1-1: Block Diagram of a typical Embedded Communication Controller

A simple function of such an embedded communication controller is collecting information on one interface, processing this information and forwarding it to another interface. As the clock frequency of such a system typically is between 20 MHz and 100 MHz, the system is not capable of processing packets at full line rate (assuming that we have two Ethernet interfaces with a maximum line rate of 100 Mbit/s each). Figure 1-2 depicts the result of a throughput measurement of a simple IP forwarding application on such a system[1]. It is not capable of full speed forwarding for small packet sizes. The number of packets that can be processed is limited by the power of the

[1] 50MHz PowerPC CPU with commercial RTOS.

communication controller up to a certain packet size. From there on, the limiting factor is the line rate. The formula to estimate the achievable throughput p is given in (1.1).

$$p(s) = \min(r_{\max}, \frac{r_{line}}{s}), \qquad (1.1)$$

where s is the packet size in bytes, r_{max} is the maximum forwarding rate in packets per second and r_{line} is the maximum line rate in bytes per second.

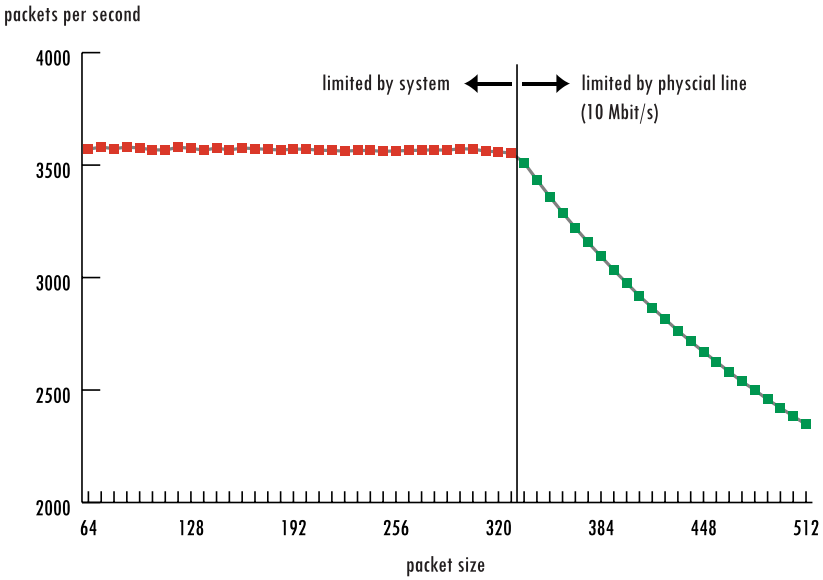

Figure 1-2: Measured throughput in packets per second for different packet sizes

In the interval in which the throughput is line rate limited, well-known algorithms (e.g. [1, 2]) and standards (e.g. [3-5]) can provide the requested quality of service. In the interval in which the throughput is limited by the system, it is required to schedule resources such that the requested quality of service or real-time behavior can be guaranteed. Not all packets can

be processed; some packets may be dropped without penalty, while other packets must be processed with minimal delay or within hard real-time constraints. As the arrival of packets is unknown, it is very difficult to create a scheduling algorithm that is not excessively conservative, i.e. always assumes a worst case arrival of packets. The following example shows that it is not feasible to assume a constant worst case arrival of packets: Assume that there are packets with hard real-time constraints. These packets are specified to arrive at an average rate of 50% of the forwarding capacity of the system. However, in worst case the packets can also arrive in bursts at line rate (the average rate will still be the same). The conservative scheduler would not allow processing any other packets, as a burst of real-time packets could arrive anytime and all resources must be ready for those packets.

1.1.2 Summary of Problem Statement

- Small, low-cost embedded systems do not have enough resources to process packets at line rate.
- It is difficult to use traditional scheduling and modeling methods for hard real-time systems as the input to the system (arrival of packets) is unknown.
- A system state based on the continual assumption of the worst-case scenario is not a viable base for a scheduling algorithm.
- Packet processing has to share the resources with other (real-time) applications that run on the system.

1.2 Related Work

A lot of focus has been put, and is being put on research for high-performance core and edge-network devices. The research objective is mainly processing power, that is, the number of packets per second that can be processed, and the extensibility of those systems. The issues addressed are optimal design, new architectures, better algorithms and implementation methods and tools (see e.g. [6-10]). Some of the results obtained there are also applicable to our target domain, the small, low-cost packet processing devices with predictable behavior.

Several operating systems, middleware components and frameworks that solve various issues in packet processing systems have been developed. Some focus on improving systems on a macro-level, e.g. the receive livelock problem [11-13], while others focus on extensibility [6, 14-16], traffic management [17-19], protocol implementation [20, 21] or resource scheduling [7, 22-24]. None of the above results can cover the requirements for low-cost and predictable packet processing systems. However, they contribute to the content of this thesis in one or the other way.

x-kernel [20] is an object-oriented framework for implementing and composing network protocols (stacks) on end-systems. It is one of the first frameworks that provided the idea of composing network protocol stacks based on small, self-contained objects. Scout OS [21] is derived from x-kernel and provides a communication oriented abstraction called “path”. A path defines the sequence of processing functions that are executed when data (packets) is moved through the system. Each path runs in its own thread. Therefore, explicit paths can be used to

improve resource allocation and scheduling [22]. In summary, Scout OS is a soft real-time system that provides admission control with respect to CPU load and memory, but does not provide mechanisms to calculate backlog and delay of individual flows. The router plugin system [16, 25] was designed to make IP routers extensible and provide a base for active networks [26]. A router plugin is a special software module that is executed based on the results of the classification of a packet. There are fixed points in the IP forwarder path at which such router plugins can be executed. Router plugins provide an efficient implementation for the extension of IP routers. However, they are highly specific to IP forwarding/active network nodes. Click [13, 14] provides a software architecture to build routers. In Click, routers are composed from (small) packet processing elements, which is a natural way to design networking applications. Click however lacks the concept of flows and does not provide any mechanism to schedule the available resources. Vera [7] is an extensible router architecture that uses a notion of paths similar to Scout OS [22] but with the focus on distributed resources. Resources are assigned and reserved to/for paths to satisfy quality of service reservations. However, Vera cannot provide hard real-time guarantees.

A concept, that provides QoS to certain flows in a software based router while optimizing the throughput for best effort traffic, has been described in [23]. Although the concept is based on scheduling the CPU resource, it does not provide any real-time guarantees. An Estimation-based Fair Queuing (EFQ) algorithm that is used to schedule processing resources has been described in [24]. It also contains a concept for online estimation of processing times and an admission control. A computation framework for extensible network routers has

been proposed by [27]. It concentrates on isolating the performance and integrity of the core router while providing extensible computation capabilities. It uses explicit flow contexts and an explicit resource reservation model for scheduling to provide soft real-time guarantees and fairness for using excess capacity.

However, all of these frameworks and architectures cannot provide real-time guarantees.

In summary, there are solutions available for various issues in software based routers. A unifying approach for small embedded devices that enables a formal analysis as well as an implementation that matches the predicted behavior is missing.

1.3 Target and Results of Thesis

The target of this thesis is to provide **a method to build predictable packet processing systems on small, low cost hardware**. In detail this thesis contributes:

1. A model that is well suited for the modeling of such systems and their environment.
2. An analysis system that allows exploring the properties of the modeled system and the application scenarios.
3. A method to map the model to an implementation
4. A software platform that supports the mapping to the implementation.

The concrete results of the thesis are:

- An easy to use model that allows to capture applications for packet processing, the (unknown) arrival of input, the real-time requirements and the resource of a single CPU low cost communication controller.
- A procedure to analyze, explore and test the system properties based on the model.
- The software platform RNOS (Real-time Network Operating System) that allows a seamless implementation of an application previously defined by the model.
- A sample implementation of a system based on RNOS, which proves that the implementation results match the analytical results.

In essence, it allows to build predictable packet processing systems on low cost hardware.

1.4 Outline

Thesis structure:

Chapter 2 introduces the model that is used throughout this thesis.

Chapter 3 presents the analysis of systems that are based on the model of this thesis.

Chapter 4 discusses the mapping of the model to an implementation.

Chapter 5 presents RNOS, which is a software platform to implement systems based on the model of this thesis.

Chapter 6 presents and discusses a sample implementation of RNOS, its properties and the measurement results.

Chapter 7 concludes the thesis with a summary and review of the results, and provides starting points for further research.

Part of the outline is shown (graphically) in Figure 1-3. It also depicts the overall design flow to implement predictable low cost packet processing systems.

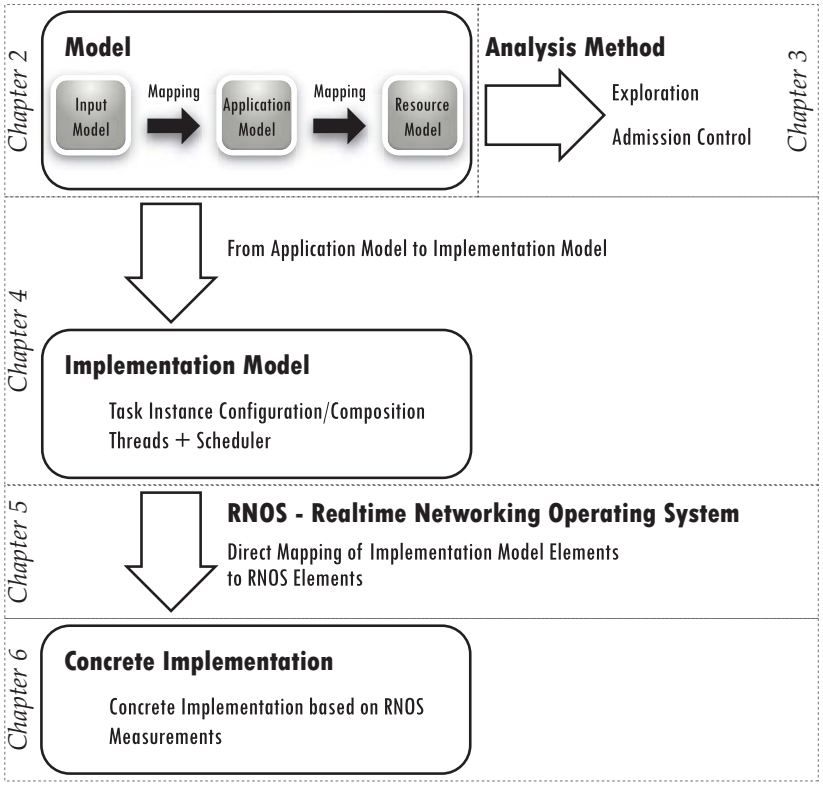

Figure 1-3: Outline of the thesis

2 Model

This chapter describes the model that later is used in the analysis and the implementation of low cost embedded packet processing systems. In the first part of this chapter, the demands on such a model are discussed and some specific terms are defined in the context of this thesis. The second part of this chapter presents the model itself.

2.1 Demands on Model

The model must be able to capture the domain specific characteristics that apply to a single network element. These are the input, the processing requirements of the input, the definition of the application and the available system resources.

Input

The input is defined as packets that arrive at the system. Packets belong to a flow (see Definition 1), for which there might be a service level agreement (SLA, see Definition 3) including quality of service parameters (see Definition 2) and a traffic profile (see Definition 4). The model must also be able to capture packets that belong to an unspecified flow or have no service level agreement or quality of service parameters.

By measuring a flow's actual traffic profile, it is possible to verify whether a flow is within the predefined limits. Such a verification is done by a policer. A traffic source might use a shaper to make sure that the traffic is within the specified bounds.

Processing Requirements

The processing requirements for packet flows can be extracted from end-to-end quality of service parameters and depend mainly on the application's tolerance to delay. Real-time applications need specific data by a certain point in time; if data arrives late, it is useless. Elastic applications will wait for packets for a certain amount of time.

Application

The application consists of functions that process packets. Each packet is required to be processed within a limited amount of time before it leaves the system or is consumed. Packets of the same flow do not necessarily pass the same processing functions. Typically, there is a common reception and classification function for all packets. Only after the execution of that function, it is clear to which flow a packet belongs.

It is important to understand that the packet processing may only represent a small part of the complete system. Interfaces connect the packet processing part with other applications of the system.

In contrast to many other application domains (e.g. signal processing), no recurrent or iterative computation takes place that manipulates a fixed input data set throughout the lifetime of the computation. Here, the packet itself and the current state of the system dynamically define what functions are to be computed.

System Resources

The main parts of a low cost embedded packet processing system are a CPU, memory and network interfaces. These elements put constraints on the actual packet processing capacity:

- The network interfaces define the maximum bandwidth for receiving or transmitting packets.
- The memory defines the maximum number packets that can be queued.
- The CPU sets the maximum processing speed.

The bus bandwidth between the memory, the CPU, and the DMA controllers represents an additional constraint to packet processing speed, as the memory is shared between the CPU and the DMA controllers. DMA controllers transfer the packets from and to the network interfaces. If execution of instructions is stalled due to DMA transfers, the available processing power of the CPU is reduced.

Definition 1

Flow	*A flow is a set of packets that display common properties within the data they contain. Typically, these are the incoming interface, ranges of source and destination IP addresses, transport protocol and ports or port ranges. Therefore, a flow may consist of an aggregation of packets from different applications or transport layer sessions.*

Definition 2

QoS	*The performance properties of a network service. May include such parameters as throughput, transit delay and priority.*

Definition 3

SLA	*A service level agreement (SLA) is a contract between a network service provider and a customer that specifies, usually in measurable terms, what services with which performance properties (QoS) the network service provider will provide. Besides the description of QoS parameters and assigned flows, an SLA may also include specifications of network availability, help desk, etc.*

Definition 4

Traffic Profile	*A traffic profile shows whether traffic of a flow is in compliance with the specified requirements or not.*

2.2 Design of Model

The model proposed in this thesis consists of several parts. There is an input model, an application model and a resource model. Then there are the mappings between these parts, namely a mapping that connects the input with the application and a mapping that connects the application with the resources (see Figure 2-1).

Figure 2-1: Model overview

2.2.1 Input Model

To provide any form of real-time behavior in the processing of packets on a per flow basis, packet arrivals from any flow need to be bounded in some way. There are various means of providing bounds on the incoming traffic profile. In this thesis, we concentrate on worst-case deterministic bounds. The following two models are the most commonly used to define traffic profiles [4, 30].

(σ, ρ) model: The (σ, ρ) model is defined by its two parameters, σ and ρ. σ describes the maximum burst size and ρ describes the long-term maximum rate. This is equivalent to the token bucket model as shown in Figure 2-2. The bucket is continuously filled at rate ρ using tokens, which represent units of bytes, up to the level σ, the size of the bucket. Initially, the bucket is filled up with tokens and traffic is allowed to pass the token bucket if there are a sufficient number of tokens in the bucket to match the passing tokens, i.e. that the number of tokens in the bucket has to be equal or higher than the length in bytes of the next packet. With each packet passing the token bucket, the number of tokens in the bucket is reduced by the length in bytes (=number of tokens) of the packet. If there are not enough tokens available, the packet has to "wait".

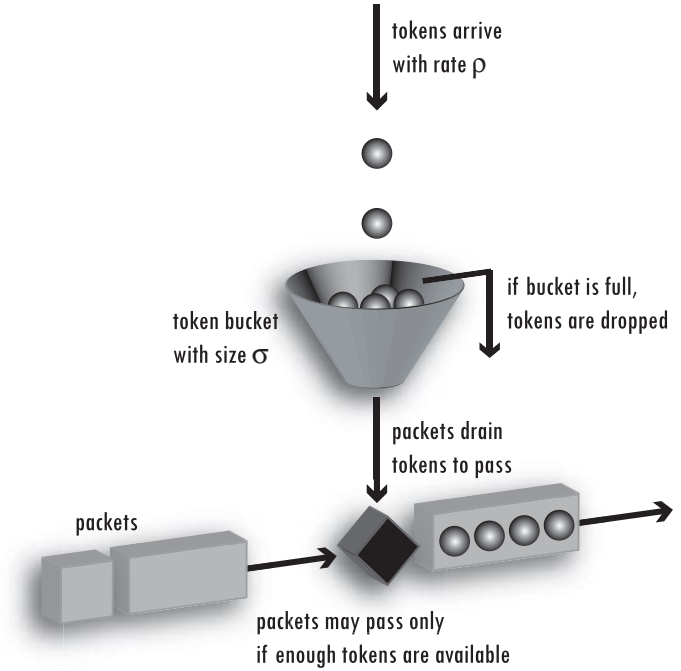

Figure 2-2: Single Token Bucket

The (σ, ρ) model is an input model in the sense that the output of a single token bucket is conformant to the input model.

Token buckets cannot only be used to shape traffic, they can also be used to check conformance of traffic (if a packet has to "wait" for more tokens the traffic is non-conformant) and to police it (drop packets that would have to "wait" for more tokens). The difference between shaper, policer and conformance checker is a possible queue in front of the token bucket (in case of a shaper) and the action that is performed when not enough tokens are available for the packet to pass (mark packet as non-conformant for conformance checker, drop packet for policer, wait in queue for shaper).

TSpec: The TSpec model was introduced in the context of QoS reservations on the Internet. Essentially, the TSpec model consists of two token buckets, as shown in Figure 2-3.

The parameters of the TSpec models are a token bucket with a token rate r and a bucket size b, and a second token bucket with rate p (peak rate) and a bucket size M (maximum packet size), and a minimum policed unit m.

Either the maximum throughput of a packet processing system is limited by the network interface bandwidth or by the number of packets per second it can process. For those systems, that are limited by the number of packets per second they can process, the minimum policed unit m helps in bounding the maximum number of packets per second, see (2.1). Without the term m, a system would have to assume that every flow is sending minimum-sized packets. With the term m, a packet that has a size equal or less than m is treated as a packet of size m. Obviously, m must be less or equal to M.

$$r_{packet}^{(\max)} = \frac{p}{m} \qquad (2.1)$$

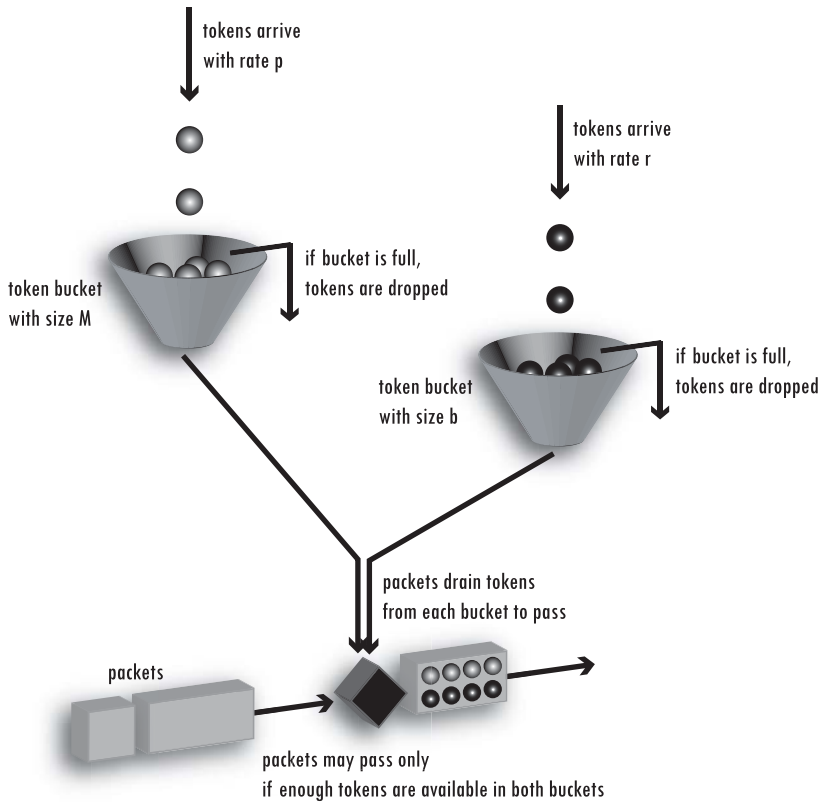

Figure 2-3: Dual Token Bucket for TSpec

The deterministic bounds on traffic flows considered in this thesis are based on network calculus [30, 31]. The two traffic profile models presented before can be easily translated to this algebra. Definition 5 and Definition 6 give the basic mechanism to capture traffic profiles in our model [32].

Definition 5

Arrival Function	*The arrival function $a_f(t)$ of a packet flow f is defined as the number of bytes or packets belonging to the flow that have arrived at a defined place in time interval [0,t]. Whether $a_f(t)$ refers to the number of bytes or the number of packets is either specified, or apparent within the context.*

Definition 6

Arrival Curve	*$\alpha^l(\Delta)$ is the minimum number of bytes or packets of the same flow that arrive in any given time interval Δ. Similar, $\alpha^u(\Delta)$ is the maximum number of bytes or packets of the same flow that arrive in any time given interval Δ. Whether $\alpha^l(\Delta)$ and $\alpha^u(\Delta)$ refer to the number of bytes or the number of packets is either specified, or apparent within the context.*

The upper and lower arrival curves specify the upper and lower bound for arrival of the number of bytes or packets in any time interval.

Depending on the actual function that is executed in the system, the load is dependent on the number of packets or the number of bytes (size of packets). Most packet processing functions have a per-packet resource demand and are more or less independent on the packet size. Typical functions that are dependent on the packet size are encryption, compression and the processes that receive and transmit packets[2].

[2] For systems that do not use a DMA controller to transfer packets from/to memory.

The (σ, ρ) model and the TSpec can be easily translated to arrival curves with packets as unit. (2.2) and (2.3) give the upper arrival curve for a (σ, ρ) model and a TSpec. Figure 2-4 and Figure 2-5 show their graphical representation. For the (σ, ρ) model, *m* is the minimum packet size. For the TSpec, it is the minimum policed unit, as defined by the TSpec.

$$\alpha^u(\Delta) = \frac{\sigma}{m} + \frac{\rho}{m} \cdot \Delta \tag{2.2}$$

$$\alpha^u(\Delta) = \min\left\{\frac{M}{m} + \frac{p}{m} \cdot \Delta, \frac{b}{m} + \frac{r}{m} \cdot \Delta\right\} \tag{2.3}$$

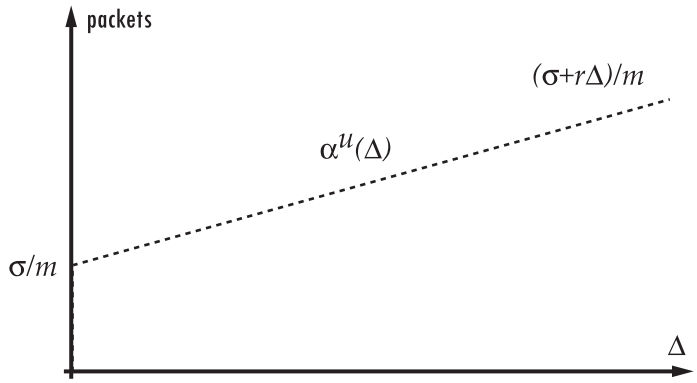

Figure 2-4: (σ, ρ) model as upper arrival curve

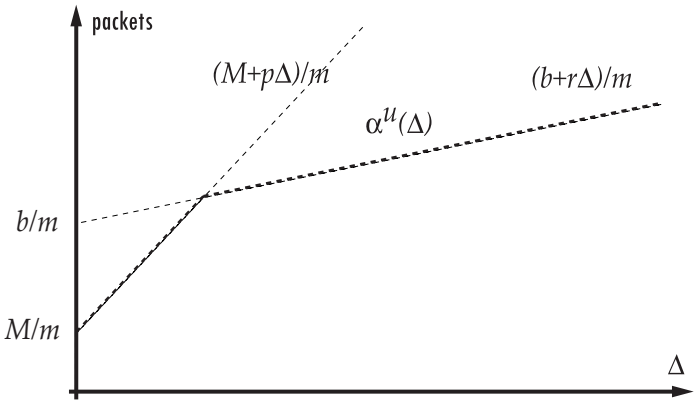

Figure 2-5: TSpec as upper arrival curve

In truth, the arrival curves would look like an integer (discrete) curve (a packet can be processed only if it has arrived completely), but to simplify calculations, we use the upper envelop of the steps.

Examples

Assume that we have a constant rate packet source somewhere in the network. The service level agreement defines a maximum jitter *j* that is introduced by the network. From this information we can create the arrival curves as follows:

First we create the arrival function, see Figure 2-6. The maximum jitter *j* defines the duration at which packets might arrive back-to-back, resulting in a burst of packets at line rate.

Second, we create the lower and upper arrival curve thereof (see Figure 2-7). The lower and upper arrival curve satisfy the inequality given in (2.4).

$$\alpha^l(t-s) \le a_f(t) - a_f(s) \le \alpha^u(t-s), \quad \forall s,t, \text{ where } 0 \le s \le t \qquad (2.4)$$

For any $\Delta \ge 0$, $\alpha^l(\Delta) \ge 0$ and $\alpha^l(0) = 0$. Therefore, $\alpha^l(\Delta)$ gives the lower bound on the number of packets that can arrive within any time interval Δ. $\alpha^u(\Delta)$ gives the corresponding upper bound.

Given the arrival function, lower and upper arrival curve can be computed using (2.5) and (2.6).

$$\alpha^l(\Delta) = \inf_{t \ge 0} \left\{ a_f(\Delta + t) - a_f(t) \right\} \qquad (2.5)$$

$$\alpha^u(\Delta) = \sup_{t \ge 0} \left\{ a_f(\Delta + t) - a_f(t) \right\} \qquad (2.6)$$

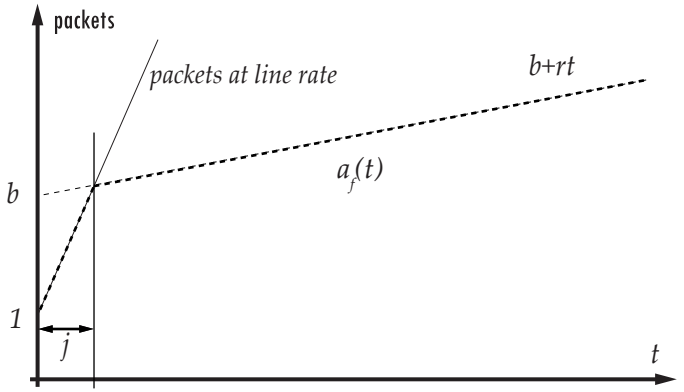

Figure 2-6: Arrival function for constant rate packet source with network jitter

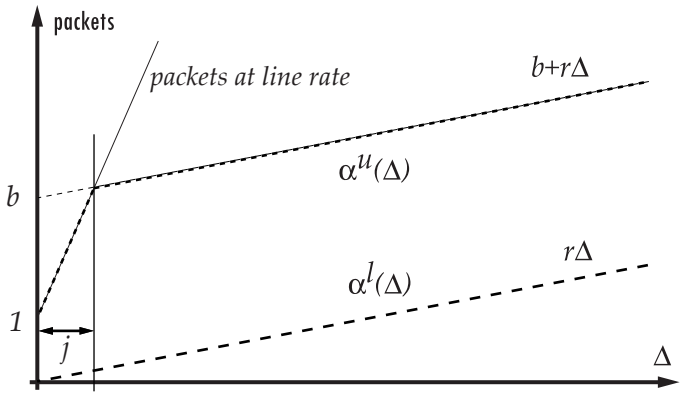

Figure 2-7: Arrival curves for constant rate packet source with network jitter

Internal "input", namely packets generated by an application or as part of a protocol, is modeled identically. If nothing is known about the incoming traffic on a physical port, we have to assume the worst case, namely packets arriving using the entire bandwidth and the packets being of smallest possible size. Figure 2-8 depicts the resulting arrival curves.

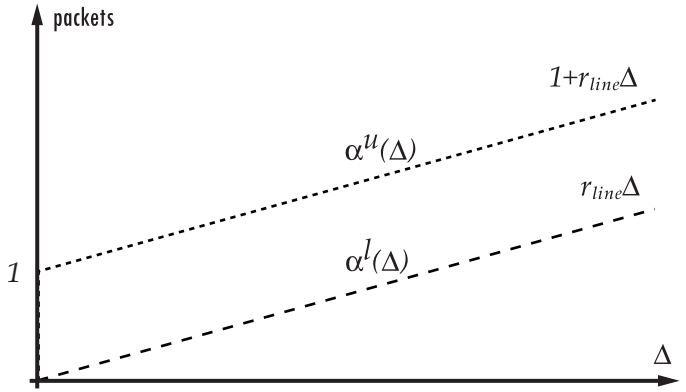

Figure 2-8: Arrival Curves for smallest packets back-to-back on line

Definition 7 defines the input of our model. It is based on flows which are specified by arrival curves and a processing requirement (quality of service parameter) in form of a deadline (Definition 8). For non real-time flows, the deadline can be infinite. The deadline of a flow is used to prioritize a flow over other flows. Alternatively, fixed priorities for flows can be used.

Definition 7

Input	*The input to the system is a set of flows $f \in F$. Each flow f has an associated lower arrival curve $\alpha^l(\Delta)$ and upper arrival curve $\alpha^u(\Delta)$. These curves are the lower and upper bound for the arrival of packets of that flow. Additionally, each flow has an associated processing requirement in form of a (possibly infinite) deadline.*

Definition 8

Deadline	*The deadline of a flow is a relative deadline. The absolute deadline can be calculated by adding the time when a packet of a flow enters the system and the relative deadline. The relative deadline is specified as an absolute value in e.g. milliseconds.*

In summary, the proposed input model consists of flows whose traffic profiles are specified by an upper and lower arrival curve. Further, the processing requirement in form of a (relative) deadline or a fixed priority specifies how packets of each flow shall be treated, i.e. if they are part of a real-time flow or not.

2.2.2 Application Model

The application model defines the required functionality of the packet processing part of the system. We have to be aware that other applications may also run on the system. Therefore, the model must also be able to interface with these applications.

To be useful, the application model has to be easy to use, i.e. it must be optimized to model packet processing applications. The application model proposed in this thesis is based on the fact that typical packet processing applications can be partitioned into small individual processing units which we will call tasks. In addition, the application model is based on the notion of events, e.g. a packet has been received or a timer elapsed. The combination of these concepts leads to a model where each event in the system has its own "program" that is triggered for execution when the event occurs. These "programs" consist of tasks.

Tasks

Tasks are non-preemptive execution blocks of code. Packets are received and delivered through at most one input and an arbitrary number of outputs, respectively. When a task executes, it takes the packet from the input, processes it and, depending on the content of the packet, puts the packet on exactly one of its outputs. There are three types of tasks: The tasks we just discussed (normal tasks), source tasks and sink tasks. Figure 2-9 shows the different types of task.

A source task has no inputs and will receive its packets from a driver, e.g. from an Ethernet driver, or create packets itself, e.g. based on an event that elapsed.

A sink task has no outputs and will either consume the packet, pass the packet to a driver for transmission, e.g. to the Ethernet driver, or pass it to another application mode, e.g. by the socket interface.

Source and sink tasks are also the interface to applications which implementations are not based on the model of this thesis.

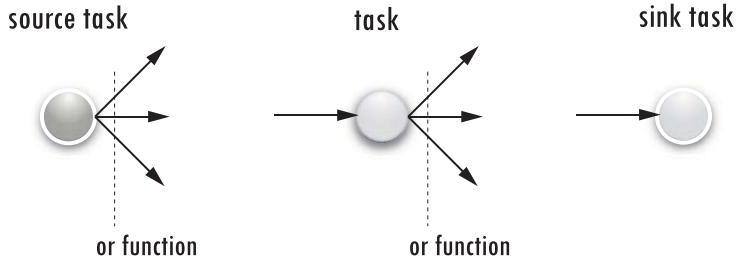

Figure 2-9: Tasks types

The different tasks are generally self-contained and independent of each other. However, they assume a packet of a given type as input. For example, a tasks that forwards IP packets based on a forwarding table expects an IP packet at its input. Therefore, only tasks that output IP packets may be connected to the input of the IP forwarder task. In order to connect the output of a task with the input of another task the packet type has to match, i.e. the packet type that is sent to the output has to be the same as the packet type the connected tasks expects at its input.

Any task can consume a packet and therefore become dynamically a sink. An example is the IP reassembly task, which consumes as many packets as are needed to reassemble a complete packet.

The worst-case and best-case execution times of each task need to be known, either by formal analysis [33], by simulation in case of soft real-time constraints [34] or by measurement (see Chapter 6).

The model does not say anything about the granularity of the tasks. Later we will see that the granularity of the tasks has an influence on the throughput and delay.

Definition 9

Task	*A task t has at most one input and n outputs. A task t with no input is called a source task. A task t with no output is called a sink task. A task takes a packet from its input, processes it, and puts it on one of its outputs. A source task creates the packet itself or gets the packet from a driver/other application. A sink task consumes the packet or forwards it to a driver/other application. Each task may become dynamically a sink task. Each task t has a worst-case (upper) execution time e^u and a best-case (lower) execution time e^l.*

Task Graph

A total application contains a set of task graphs, which consist of connected tasks. In particular, each output of a task is connected to one input of another task and for each packet source there is a task graph with a source task at its root. The task graph is a connected, directed and acyclic graph. An application consists of as many task graphs as there are packet sources.

Each output of a task is connected to exactly one input of another task. An input of a task may be the destination of several outputs of other tasks. A packet will traverse the task graph from the source to a sink. The source task will receive a packet, e.g. from the Ethernet, process it, and, depending on the content of the packet, pass it to one of its outputs where it is received and processed by the subsequent task. A packet will follow exactly one path when it traverses the task graph.

Definition 10

Task Graph	*A task graph T is a connected, directed and acyclic graph. It consists of a set of task nodes t_i with exactly one source task t^{src}, the node without in-edges, and at least one sink task t^{snk}, a node without out-edges, and a set of directed edges, which connect the outputs of the tasks with the input of other tasks. An input of a task may be the destination of several outputs of other tasks.*

Definition 11

Application	*A complete application consists of a set of task graphs. The number of task graphs is equal to the number of packet sources in the system. A packet source can be an internal source, an interface to an application part that is not part of this model or an external source, e.g. a network interface.*

Figure 2-10 shows a simplified task graph for packet reception in an IP router. The task graph is simplified as it only shows the paths for IP packets. The complete application consists of a number of such graphs, one for each packet source.

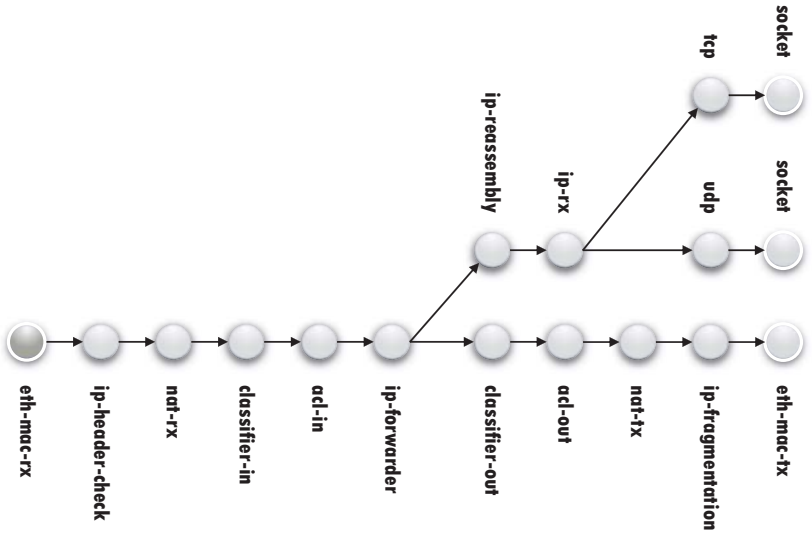

Figure 2-10: Simplified task graph for packet reception that contains three paths

Often, there are task graphs with no forks, i.e. simple sequences of tasks. Figure 2-11 shows a task graph that handles voice over IP data packets received from a DSP.

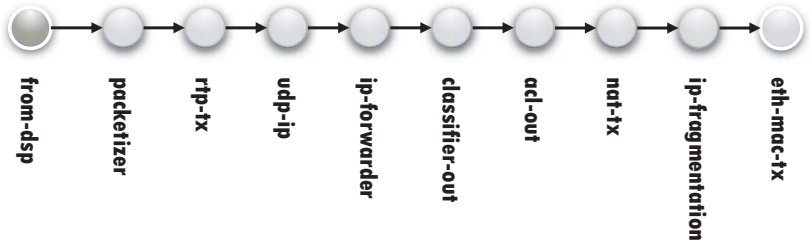

Figure 2-11: Task graph from DSP to Ethernet

Figure 2-12 shows a more complex task graph, which adds IP security support to the simple task graph of Figure 2-10. The tasks *esp-en*, *esp-de*, *ah-en*, and *ah-de* are collapsed into a single task. As an example, the non-collapsed version of the task *esp-de* is shown in the picture. As this task graph shows, it is possible to have the outputs of several tasks connected to the input of the same task. The task graph is still a directed acyclic graph. However, it is not allowed to have loops. This is why the task graph in Figure 2-12 contains the identical task *ip-forwarder* three times. There are several possibilities to draw a task graph. Equivalence between task graphs can be tested using Theorem 1 (see next section).

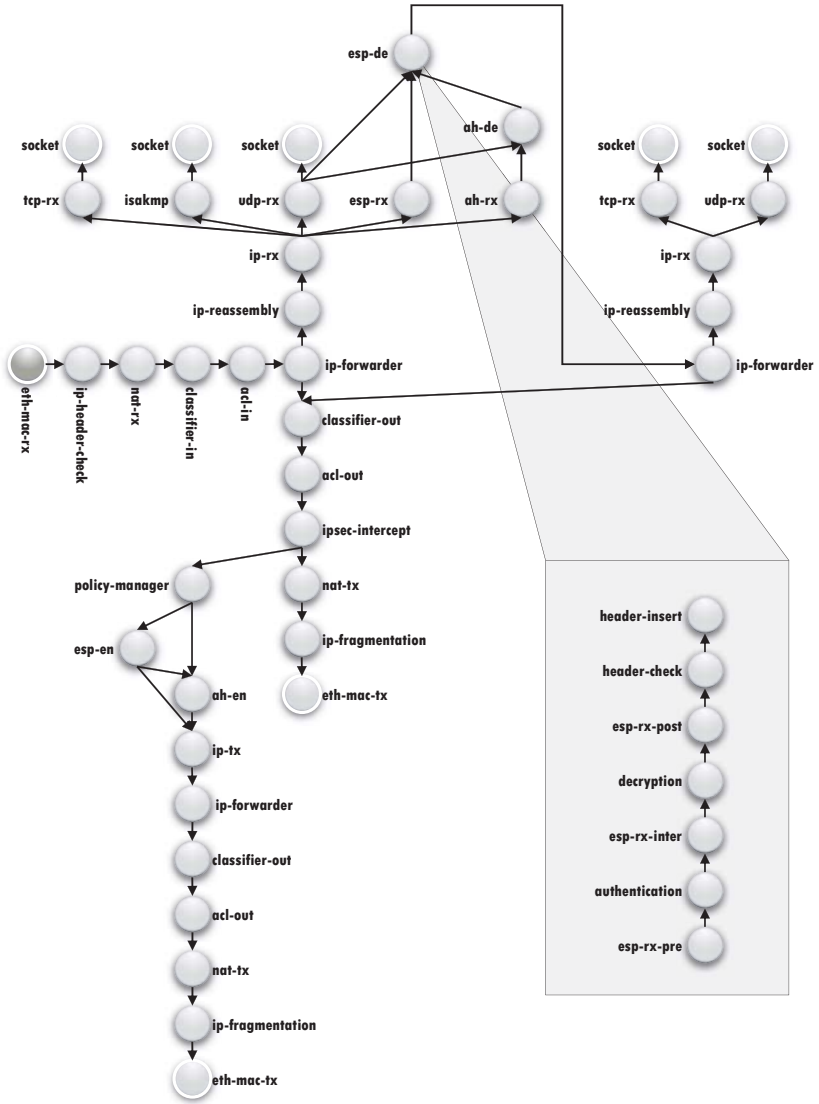

Figure 2-12: Simplified task graph for packet reception with IPSec support

In summary, the application model consists of multiple, directed and acyclic task graphs. The tasks of the task graph are non-preemptive execution blocks that have one input and can have several outputs. The input and output data of a task are packets.

2.2.3 Mapping Input to Application

We have a model for the input and a model for the applications. What is missing is the connection between the two.

Each packet in the system is associated with a flow. A flow has associated arrival curves and parameters from the service level agreement. Applications are made of task graphs. Each packet source in the system has an associated task graph. Therefore, packets that arrive at a specific source will traverse the same task graph.

Each flow is mapped to the task graph that contains the source task for this flow (where packets will arrive or will be created). For some flows we might be able to exclude some branches of the tasks graph. Therefore, the mapping is not to a complete task graph but to a set of paths through a task graph. A path is a sequence of tasks, starting at the source task and ending with a sink task. Figure 2-13 depicts a path p through a task graph, p = *(eth-max-rx, ip-header-check, nat-rx, classifier-in, acl-in, ip-forwarder, ip-reassembly, ip-rx, udp, socket)*.

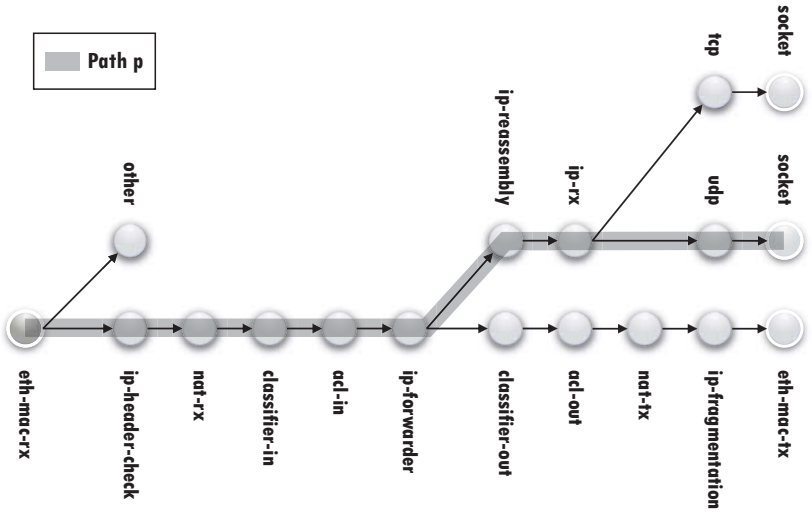

Figure 2-13: Path through a task graph

Definition 12

Path	*A path is a sequence of connected tasks of a task graph T. It starts with the source task and ends with a sink task of the task graph T.*

A task graph T contains a set P_T of paths $p_j \in P_T$. Each path starts with the same source task of T (there is exactly one source task in every task graph). The maximum number of task paths through a task graph can be computed using Algorithm 1.

With each task graph T, a set F_T of flows $f_k \in F_T$ is associated, i.e. those flows from which packets may enter the task graph via its source task.

With each flow $f_k \in F_T$ associated with task graph T, a set P_f of paths is associated, where $P_f \subseteq P_T$. The set P_f contains all the

paths through the task graph T a packet of flow f_k might traverse.

Algorithm 1: Counting paths through a task graph

```
01 path_counter = 0;
02 for each sink task t_s in task graph T
03   call traverse_and_count( t_s );
04
05 // path_counter now contains the number of paths
06 // through the task graph T
07
08
09 traverse_and_count( task t )
10 {
11   if ( t is a source task )
12     path_counter++;
13   else
14     for each previous task t_p of t
15       call traverse_and_count( t_p );
16   end if
17 }
```

Definition 13

Mapping of Flows
To each flow f a set of paths $p \in P_f$ is assigned. These sets are made up of the sum of all paths through the task graph T, which could potentially be traversed by a packet of that flow. If P_T is the set of all paths through the task graph T, $P_f \subseteq P_T$.

Theorem 1

Equivalence of Task Graphs
Two task graphs are equivalent if and only if they contain the same paths.

In summary, each flow is associated with a set of task paths. Each packet of the flow will traverse one of the paths in that set.

2.2.4 Resource Model

Analogous to arrival curves describing packet flows, the availability of computation capacity is characterized using service curves (see Definition 14 and Definition 15). The computation capacity can be defined as the availability of computation resource in microseconds in a given time interval [0,t]. The lower and upper service curve describing the computation capacity satisfy the inequality given in (2.7).

$$\beta^l(t-s) \le c(t) - c(s) \le \beta^u(t-s), \quad \forall s,t, \text{ where } 0 \le s \le t \tag{2.7}$$

For any $\Delta \ge 0$, $\beta^l(\Delta) \ge 0$ and $\beta^l(0) = 0$. Therefore, $\beta^l(\Delta)$ gives the lower bound on the computation capacity within any time interval Δ. $\beta^u(\Delta)$ gives the corresponding upper bound. Hence, the processing capacity over any time interval Δ is always greater than or equal to $\beta^l(\Delta)$ and less than or equal to $\beta^u(\Delta)$.

Given the service function, the lower and upper service curve can be computed using (2.8) and (2.9).

$$\beta^l(\Delta) = \inf_{t \ge 0} \{c(\Delta + t) - c(t)\} \tag{2.8}$$

$$\beta^u(\Delta) = \sup_{t \ge 0} \{c(\Delta + t) - c(t)\} \tag{2.9}$$

Definition 14

Service Function	*The service function c(t) of a CPU is defined as the available computation capacity at a given time in time interval [0,t].*

Definition 15

Service Curve	*$\beta^l(\Delta)$ is the minimum available computation capacity in any time interval Δ. Similar, $\beta^u(\Delta)$ is the maximum available computation capacity in any time interval Δ.*

The total computation capacity has to be shared with other applications running on the system. To determine the potential computation capacity, we need to know how much computation capacity is required by the other applications running on the system and what the pattern of use is. Typically we will have access to the resource in the first x% of a period, while the rest of the period is reserved for use by other applications (see Figure 2-14). Figure 2-15 shows the upper and lower service curves for such an access pattern.

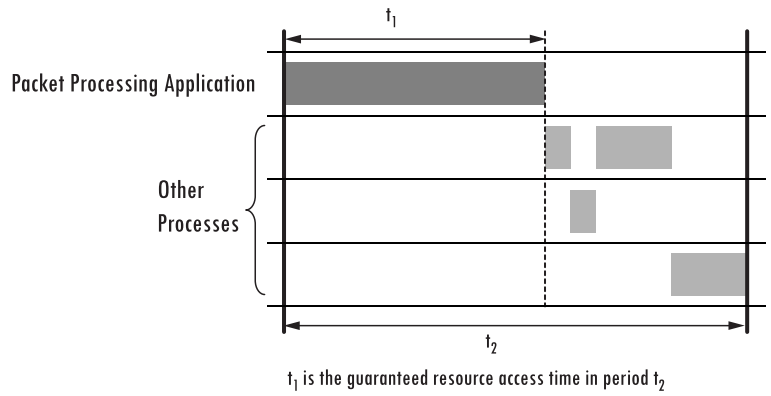

Figure 2-14: Resource Access Pattern

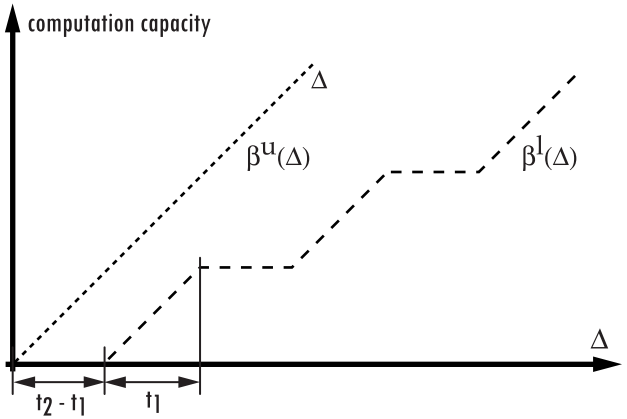

Figure 2-15: Upper and lower service curves for a CPU resource

Definition 16

Resource	*A resource C of a packet processing system is given by its computation capacity, which is represented by a lower service curve $\beta^l(\Delta)$ and an upper service curve $\beta^u(\Delta)$.*

2.2.5 Mapping Application to Resources

To complete our model, we have to map the application to the resources (compare Figure 2-1). As we have a single resource, the mapping of the application to the resource is trivial: All tasks are mapped to the same resource.

Definition 17

Mapping of Resource	*All tasks t of the entire application are mapped to the same single resource C.*

The model proposed in this thesis has a very simple resource model and mapping, which supports only one resource. We think that it should be possible to extend the model for multiple resources with interdependencies between them. However, this is out of scope of this thesis.

2.3 Summary

In this chapter we introduced a domain specific model for small low cost packet processing systems that will be used throughout the remainder of this thesis. It consists of an input model, an application model, a resource model and the mappings between these models.

The input and resource model are based on network calculus. The input model consists of flows. For each of these flows there are upper and lower arrival curves and associated qual-

ity of service parameters, which in essence indicate the tolerance to delay.

The application model is based on task graphs. Tasks are non-preemptive execution blocks. The task graphs are connected, directed and acyclic graphs of such tasks. An entire application consists of many tasks graphs; there is one for each packet source in the system. Packets arrive (or are created) at a source task and traverse the task graph up to a sink task, where they are either consumed or sent out of the system. This application model leads to the natural way of designing packet processing systems based on small (self-contained) processing elements.

The mapping between the input and application model is accomplished by assigning paths in a task graph to flows. Packets of the same flow traverse the same task graph, but not necessarily all paths in the task graph are taken. Therefore, the set of paths through the task graph that might be taken by any packet of the flow is assigned to that flow.

The resource model provides the available computation capacity of the system. As other applications, which are not designed by our model, run on the same system, the available computation capacity needs to be modeled.

The mapping between the application and resource model is simple, as the application domain has a single resource only. All tasks are assigned to the same and single resource.

3 Analysis

This chapter describes how to analyze a system based on the model presented in the previous chapter. The first part of this chapter gives some general background on using real-time calculus [35, 36]. The tools obtained there serve in the second part where the calculation scheme for the model described in Chapter 2 is introduced. The model and the analysis presented in this chapter will allow us to explore application scenarios and evaluate the properties of individual flows. In the third part of this chapter we provide an example that demonstrates the power of exploration provided by combination of model and analysis. We are now able to evaluate a system for its intended usage before it is built.

As our aim is to offer guarantees and real-time processing, we need an admission control that verifies whether a new flow with a specific requested quality of service should be admitted into the system or whether this flow should be rejected as it would hamper the processing of already admitted flows. The last part of this chapter shows how to perform the admission control for the model.

3.1 Basic Calculations

Based on the arrival curve of packets and a service curve for processing the packets, worst-case bounds for the delay and the backlog can be calculated. It is important to note that the arrival and service curve have to have the same units (e.g. packets). Given a flow with an arrival curve and a processing system with a service curve, the maximum delay and backlog experienced by packets of the flows in the system are given by the inequalities in (3.1) and (3.2) [30, 32, 37].

$$delay \le \sup\left\{\inf_{\Delta\ge 0}\left\{\tau : \tau \ge 0 \wedge \alpha^u(\Delta) \le \beta^l(\Delta-\tau)\right\}\right\} \tag{3.1}$$

$$backlog \le \sup_{\Delta\ge 0}\left\{\alpha^u(\Delta) - \beta^l(\Delta)\right\} \tag{3.2}$$

A geometrical interpretation of these inequalities is depicted in Figure 3-1. The maximum delay experienced by packets waiting to be served by the system can be bounded by the maximum horizontal distance between the upper arrival curve and the lower service curve. The maximum backlog is bounded by the maximum vertical distance between the same curves.

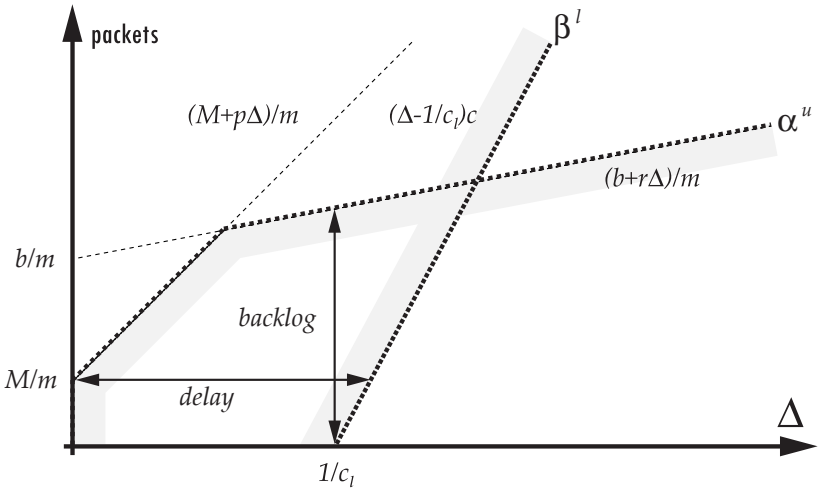

Figure 3-1: Bounds on delay and backlog

Before continuing, we define some operators that will be used throughout the remainder of this thesis.

$$v(\Delta) \wedge w(\Delta) = \min\{v(\Delta), w(\Delta)\}$$

$$v(\Delta) \underline{\oplus} w(\Delta) = \inf_{0 \le \lambda \le \Delta} \{v(\lambda) + w(\Delta - \lambda)\}$$

$$v(\Delta) \overline{\otimes} w(\Delta) = \sup_{0 \le \lambda} \{v(\Delta + \lambda) - w(\lambda)\} \qquad (3.3)$$

$$v(\Delta) \overline{\oplus} w(\Delta) = \sup_{0 \le \lambda \le \Delta} \{v(\lambda) + w(\Delta - \lambda)\}$$

$$v(\Delta) \underline{\otimes} w(\Delta) = \inf_{0 \le \lambda} \{v(\Delta + \lambda) - w(\lambda)\}$$

After the packets of a flow are processed, they have a different traffic profile. The physical model is that of a traffic shaper, i.e.

a single token bucket with a queue in front and a token bucket size of zero. The tokens represent the resource. The size zero of the token bucket means that resources cannot be accumulated or stored and that unused resources are thrown away immediately. Figure 3-2 depicts this physical model.

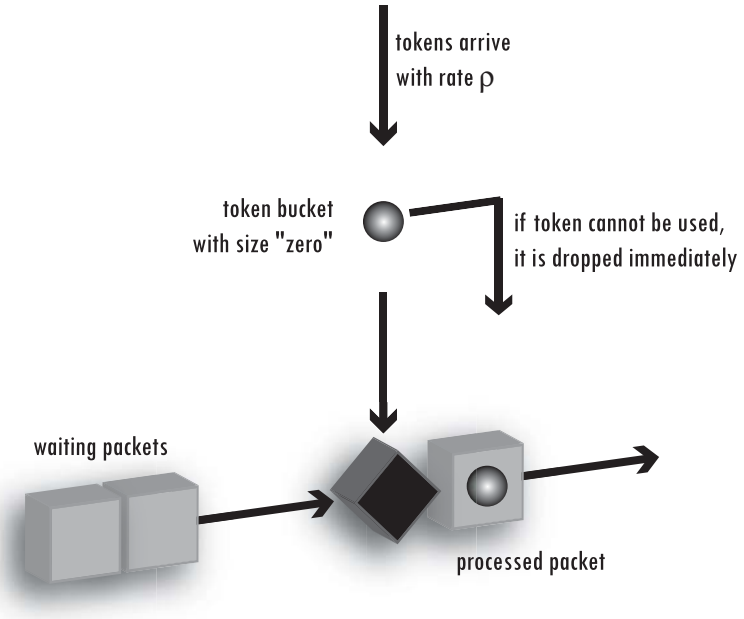

Figure 3-2: Physical processing model

Figure 3-3 shows the corresponding basic calculation layout for arrival and service curves. The packets of a flow arrive within the bounds specified by the lower and upper arrival curve α^l and α^u. They are processed by a service with the lower and upper service curve β^l and β^u. After processing, packets of the flow are within the bounds specified by the lower and upper arrival curve $\dot{\alpha}^l$ and $\dot{\alpha}^u$. Similar, after proc-

essing the packets, the bounds of the remaining available service are defined by $\dot{\beta}^l$ and $\dot{\beta}^u$ (see (3.4) to (3.7)) [35].

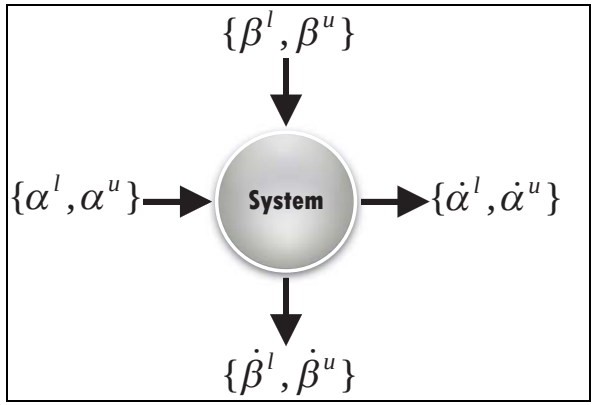

Figure 3-3: Basic calculation layout

$$\dot{\alpha}^u(\Delta) = \left(\left(\alpha^u(\Delta) \oplus \beta^u(\Delta)\right) \overline{\otimes}\, \beta^l(\Delta)\right) \wedge \beta^u(\Delta) \tag{3.4}$$

$$\dot{\alpha}^l(\Delta) = \left(\left(\alpha^l(\Delta) \overline{\otimes}\, \beta^u(\Delta)\right) \underline{\oplus}\, \beta^l(\Delta)\right) \wedge \beta^l(\Delta) \tag{3.5}$$

$$\dot{\beta}^u(\Delta) = \left(\beta^u(\Delta) - \alpha^l(\Delta)\right) \underline{\otimes}\, 0 \tag{3.6}$$

$$\dot{\beta}^l(\Delta) = \left(\beta^l(\Delta) - \alpha^u(\Delta)\right) \overline{\oplus}\, 0 \tag{3.7}$$

3.2 Calculation Scheme

It is not possible to apply the calculus of the previous section directly to our model, as the units of the arrival and service

curve do not match. The arrival curves define the arrival of packets in number of packets per second, while the service curves define the available processing capacity in number of instructions executed per second. Therefore, we must transform the number of packets per second into number of instructions executed per second. As we shall see, this transformation is given by the mapping of the flow to the flow's application.

Each task t_i has a lower and upper execution time e_i^l and e_i^u. Estimates of the lower and upper execution times can be measured while the system is under maximum load or idle, respectively. Nevertheless, this approach is restricted to soft quality of service constraints only. Another possibility is to formally analyze the tasks which yield bounds on the worst case and best case execution times [33, 38].

A task tree T contains a set P_T of task paths $p_j \in P_T$. Each task path p_j consists of a set of tasks $t_i \in p_j$. The lower and upper execution time $e_{p_j}^l$ and $e_{p_j}^u$ of a task path p_j is the sum of the execution times e_i^l and e_i^u of its tasks t_i (see (3.8) and (3.9)).

$$e_{p_j}^l = \sum_{i:t_i \in p_j} e_i^l \tag{3.8}$$

$$e_{p_j}^u = \sum_{i:t_i \in p_j} e_i^u \tag{3.9}$$

For each flow f_k there is an associated set P_{F_k} of task paths. The lower and upper execution time for the processing of a packet of that flow are given by (3.10) and (3.11) .

$$e^{l}_{f_k} = \min_{p_j \in P_{f_k}} \left(e^{l}_{p_j} \right) \tag{3.10}$$

$$e^{u}_{f_k} = \max_{p_j \in P_{f_k}} \left(e^{u}_{p_j} \right) \tag{3.11}$$

With this information we can transform the arrival curves given in arrival of packets of a certain flow (designated with a bar above the alpha) to arrival curves expressed in required processing capacity of that flow (without bar). The transformation is a simple scaling of the arrival curves, see (3.12) and (3.13).

$$\alpha^{u}_{f_k}(\Delta) = e^{u}_{f_k} \bar{\alpha}^{u}_{f_k}(\Delta) \tag{3.12}$$

$$\alpha^{l}_{f_k}(\Delta) = e^{l}_{f_k} \bar{\alpha}^{l}_{f_k}(\Delta) \tag{3.13}$$

(3.4) to (3.7), (3.1) and (3.2) are now ready to be used. To transform back the resulting arrival curves after the processing to their original form, (3.14) and (3.15) can be applied.

$$\dot{\bar{\alpha}}^{u}_{f_k}(\Delta) = \left\lceil \frac{1}{e^{u}_{f_k}} \dot{\alpha}^{u}_{f_k}(\Delta) \right\rceil \tag{3.14}$$

$$\dot{\bar{\alpha}}^{l}_{f_k}(\Delta) = \left\lfloor \frac{1}{e^{l}_{f_k}} \dot{\alpha}^{l}_{f_k}(\Delta) \right\rfloor \tag{3.15}$$

Figure 3-4 depicts the complete calculation scheme for one flow. In step 1, the lower and upper arrival curves are transformed such that they define the bounds for the required processing capacity. Step 2 computes the bounds of the remaining processing capacity and the lower and upper arrival curves of the flow after it has been processed. Finally, step 3 converts the lower and upper arrival curves back to units of packets per second.

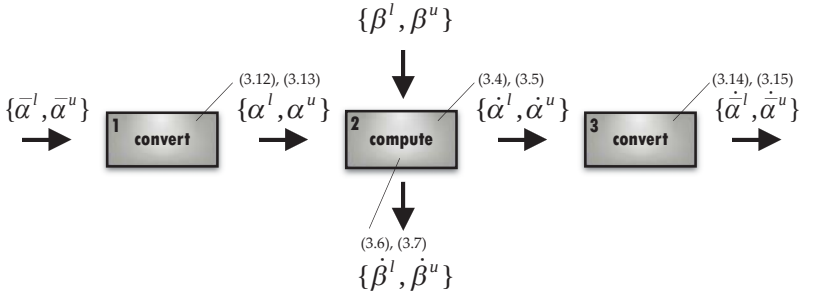

Figure 3-4: Calculation scheme for one flow

Figure 3-5 is an example for a fixed priority scheme for multiple flows. It is an extension of the calculation scheme for one flow by cascading the individual calculation schemes. The best effort flow has the lowest priority (infinite deadline). Accordingly, it is processed last with the remaining processing capacity. A possible way to assign the priorities is to order the flows according to their deadline. The lower the allowed delay (earlier deadline), the higher is the priority of the flow.

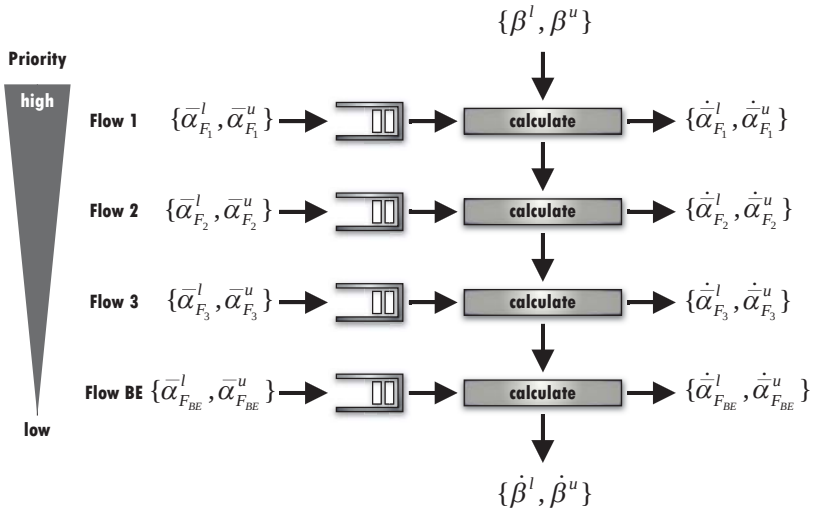

Figure 3-5: Calculation scheme for multiple flows

The calculations for the delay are correct for preemptive tasks only. In our model, which consists of non-preemptive tasks, the worst case delay is the calculated delay plus the processing time of the longest task (see (3.16)).

$$delay_{f_k}^{(\text{model})} = delay_{f_k} + \max_{\forall i} e_i^u \qquad (3.16)$$

In the following section we demonstrate how to use the model and its calculation scheme to explore various system scenarios.

3.3 Exploration

The model and analysis methods presented so far are well suited to explore various facets of a target system. The following sections offer an overview of possible explorations.

3.3.1 Example System

Our example consists of a so called intelligent access device (IAD), which provides several services for home users. Figure 3-6 depicts the IAD. The IAD provides data and voice services. The voice services are provided through Voice over IP technology (VoIP). There are ports to connect up to four phones and it has two Ethernet ports. One Ethernet port, called business Ethernet has precedence over the other Ethernet port, the kids and family Ethernet port. The business Ethernet is intended for VPN (Virtual Private Network) access to a company network to work from home, while the kids and family Ethernet is intended for Internet access in general. The xDSL (Digital Subscriber Loop) port provides access to the a service provider. In this example we assume that the access has a bidirectional access bandwidth of 2 Mbit/s.

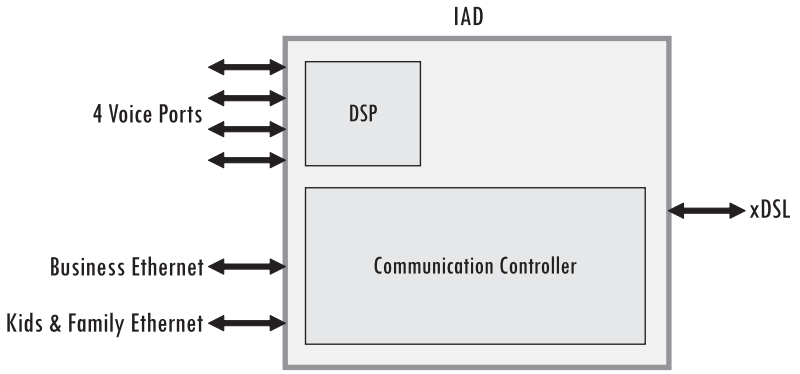

Figure 3-6: Example System

The flows are the following:

- Voice over IP data receive flow
- Voice over IP data transmit flow
- Voice over IP control receive flow

- Voice over IP control transmit flow
- Business data flow
- Kids and family data flow

The SLA (Service Level Agreement) for the VoIP data traffic has to be very stringent for a quality comparable to traditional PSTN quality should to be reached [39]. The SLA for VoIP data traffic can be specified as shown in Table 3-1.

Table 3-1: SLA parameters for the VoIP data flow

Parameter	Value
Minimum bandwidth	108kbit/s
Maximum delay	160ms
Loss probability	<1%
Maximum jitter	120ms

The minimum bandwidth can be calculated by adding the protocol headers to the payload and multiplying it with the number of packets per second. For G.711 [40] with a packet period of 10ms, this gives 150 bytes per packet, which is about 118kbit/s per direction (14 to 18 bytes Ethernet header, 6 bytes PPPoE header, 6 bytes PPP header, 20 bytes IP header, 8 bytes UDP header, 12 bytes RTP header, 80 bytes payload and 4 bytes Ethernet CRC).

From the given SLA we determine the arrival curves. The arrival curves are the lower and upper bounds for the number of packets to be processed in any time window. The bandwidth itself is not considered here. All the tasks in the application

have a per-packet execution time only. The SLA defines a maximum jitter of 120ms, which means that we could receive a maximum burst of 12 VoIP data packets at line rate. The line rate in our scenario is 2Mbit/s. A packet of the size of 138bytes requires about 530µs on the line. Figure 3-7 shows the arrival curves for the VoIP data receive traffic.

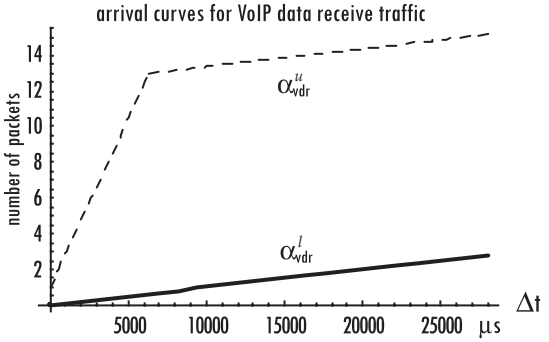

Figure 3-7: Arrival curves for VoIP data receive flow

The total delay a user will experience consists of the network delay plus the delay introduced by the end-systems. As we want to minimize the delay of VoIP data traffic introduced by our system, the VoIP data flows will have highest priority. The VoIP data transmit traffic is a constant packet rate flow. The data packets are generated at a constant rate by the internal DSP (Digital Signal Processor) of the IAD. Usually, this is a 10ms period for G.711 and longer periods for other coders. VoIP control traffic occurs mostly at call setup and teardown. During calls, not much information is exchanged between the endpoints or an endpoint and the gatekeeper. The amount of that information also depends on the actual protocol and which variant thereof is used, see also [41-45]. Here, we assume a simple constant rate of arrival for the control traffic of 10 packets per second. Business data traffic typically consists

of TCP connections for file transfers and data base lookups. These TCP connections have a good behavior in the sense that they do not produce large bursts [46, 47]. The same is true for the kids and family type of traffic, also based mainly on TCP. Here, the applications are web browsing and online games. While web browsing is not demanding in respect to throughput and delay, online games are very sensitive to delay. For both the business and kids and family traffic we allow a maximum burst arrival of 140 packets at Ethernet line rate. The sustainable rate we set to 2000 packets per second for the business traffic and 100 packets per second for the kids and family traffic. Table 3-2 summarizes the flow specification and adds the upper execution times. Note that the flows for the VoIP are flows per call and that our example system is able to handle up to four concurrent calls.

Table 3-2: Flow specification

Flow	Upper Execution Time	Relative Delay	Priority
VoIP Data Receive	90µs	5ms	1
VoIP Data Transmit	50µs	5ms	2
VoIP Control Receive	700µs	40ms	3
VoIP Control Transmit	800µs	40ms	4
Business Data	300µs	50ms	5
Kids and Family Data	300µs	50ms	6

For our scenario we assume that we know the execution times for all flows from a previous project, which used similar tasks on a similar hardware, or that we did a worst case execution

time analysis for the tasks [33]. We assume that we have access to the processing resource for 8ms in a period of 10ms (compare Chapter 2). Figure 3-8 shows the available processing resource as lower and upper service curves.

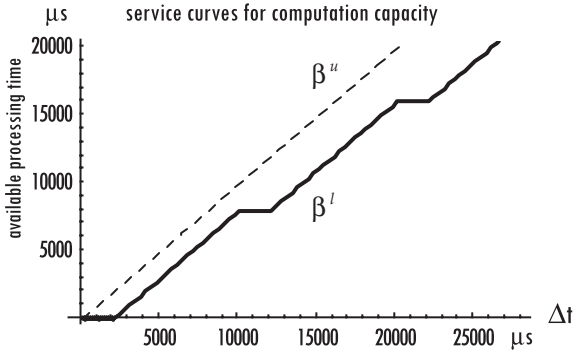

Figure 3-8: Service curves for computation capacity

Now, we will explore whether the chosen hardware system is sufficient to satisfy all requirements. We apply the calculation scheme from this chapter. Table 3-3 shows the results for backlog and delay for each individual flow.

Table 3-3: Initial results

Flow	Delay [ms]	Backlog [# packets]
VoIP Data Receive	2.56	10
VoIP Data Transmit	3.27	6
VoIP Control Receive	6.31	5
VoIP Control Transmit	9.99	5
Business Data	67.00	152
Kids and Family Data	1033.00	194

When we compare the results with the requirements for each flow, we see that the delay for business data is slightly above the requested maximum delay and that the delay for the kids and family data is way beyond the requested maximum delay. This means that with the given hardware we cannot satisfy the requirements. As the CPU to be used in our example system is available in different speed grades, we will explore which clock speed we will need to satisfy all requirements. Figure 3-9 shows the worst-case delay for the kids & family and the business flow for different CPU clock speeds. This shows that we can satisfy the delay requirements with a clock speed of 150MHz.

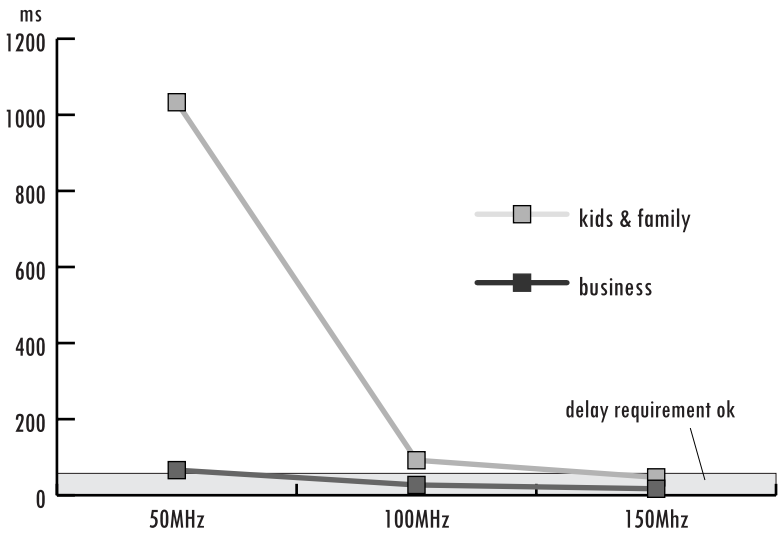

Figure 3-9: Delay of kids & family and business flow for different CPU clock speeds

In summary, the analysis presented in this chapter is well suited to explore systems defined by our model. Bottlenecks in

the system can easily be determined and appropriate design changes can be implemented.

3.4 Admission Control

The role of the admission control is to ensure that admittance of new flows to the system does not violate any quality of service commitments made to already admitted flows and that the quality of service requirements of new flows can be committed and fulfilled. Obviously, we can use the calculation scheme presented in Section 3.2 for the admission control. We calculate the delay and backlog for each flow (including the new flow) and admit the new flow only if the delay and backlog for each flow lies within the specified limits.

3.5 Summary

The analysis of the model is based on network calculus. It allows us to calculate the bounds for delay and backlog of individual flows. Thus, we can explore and design systems with adequate resources before they are actually built. The same calculation scheme is also used in the admission control of new flows.

All this only makes sense if we can implement the model to match the analysis results. The next chapter discusses how the model can be mapped to an implementation.

4 Mapping to Implementation

In the previous chapters we have introduced a model and an analysis system to capture and explore applications on low cost hardware. We have modeled the input, the applications and the processing resource, described the mapping between them and are now able to explore the properties of the modeled system and the application scenarios. The goal of this thesis is to achieve implementations of the model such that the behavior matches that of the analysis. In this chapter we describe what needs to be done to transform the model to an implementation; the existing model has to be enhanced with a few additional concepts. The application model as it was introduced in the previous chapters is a functional model. It tells us which paths a packet of a flow may take through a task graph, which is essentially the functionality that will be applied to the packet. It does not tell us anything about the physical representation of the tasks (task instances) and whether certain task instances are shared between different task graphs. So we need an implementation model that describes the physical representation of the tasks and a method to transform the application model to the implementation model. We need also an element that links an actual packet with its path through the system. Again, the model tells us which paths packets of a flow might take, but there is no notion of an individual packet. For an implementation we need to handle the individual packet. In addition, a scheduler is needed, that decides which task and therefore which packet should be processed next. Lastly, we have to enhance the model by a mechanism which accounts for the fact that the association of a packet to a flow is not known immediately when a packet enters the system.

4.1 From Application Model to Implementation Model

The application model has to be tranformed to an implementation model that can be implemented. In the implementation model we are not so much interested in the actual function a task provides, but in the actual physical representation of the tasks, i.e. the instances of the tasks. From the application model we know that each packet source in the system has its own task graph. However, it is not clear how this relates to the actual instances of tasks. If there are two identical physical ports, represented by two tasks graphs with different associated input specification, who can tell if the actual instances of the tasks are the same or not for these task graphs? Further, if there is any routing or switching element in the task graph (e.g. a forwarding task), it might not show that there are sev-

eral output ports as it is not necessary for the application model; the actual sequence of tasks that are executed is independent of the output port. However, to implement the model, it is mandatory not only to know the functional application model but also its actual physical presentation. While the application model describes the functionality, the implementation model tells us which tasks have to be instantiated in which configuration (physical representation).

In the following sections we will discuss when tasks should be separate instances and when and how they can be shared. Finally, we will annotate the model with instance information, such that it is also suitable for the implementation.

Implementations as click, router plugins and x-kernel [13, 14, 16, 20, 25] are based on connectable elements, which directly represent the implementation model. They do not have an application model and therefore do not require a transformation of the application model to the implementation model.

4.1.1 Task Instances

There are several pros and cons to having separate instances of tasks versus shared task instances. There is more involved than might be obvious at first glance. A simple example will give some idea about the decision process, and whether to have separate or shared instances of tasks.

Look at a two port Ethernet router. Its sole task is to forward packets from one port to the other or to deflect them such that they leave the system by the port by which they have been received. Figure 4-1 depicts the application model for the example. It consists of a total of six task graphs, three for each port. Remember that we have a task graph for each packet source in the system.

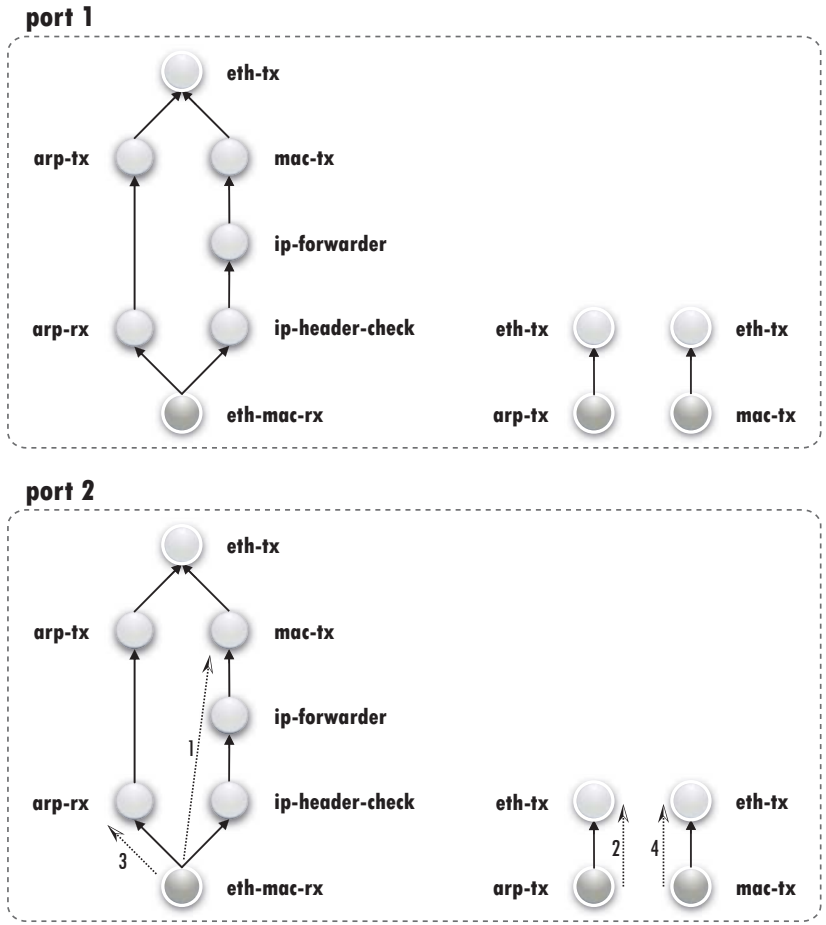

Figure 4-1: Model of two port router

The task graphs for both ports are identical. The tasks are described in Table 4-1. The task graph with *eth-mac-rx* as source task receives packets from an Ethernet port and depending on whether it is an IP or ARP packet (ARP = Address Resolution Protocol) the packet traverses a different path in the task graph. Received ARP packets are either ARP requests or re-

plies. ARP request packets will proceed to *arp-rx* and might result in a reply generated in *arp-tx* that leaves the system by *eth-tx*. ARP reply packets will proceed to *arp-rx* where the ARP information will be extracted and stored in the ARP data base. The task *arp-rx* is a sink for ARP reply packets. The entries in the ARP data base have a time-out on which the information has to be renewed. Therefore, new ARP requests are issued periodically, which is modeled by the task graph with *arp-tx* as source task. IP packets proceed from *eth-mac-rx* to the *ip-header-check*, *ip-forwarder*, *mac-tx* and leave the system by *eth-tx*. The *mac-tx* task adds the Ethernet header to the packet. To be able to do that it needs the Ethernet address of the next hop. If this information is not available, the IP packet has to be queued (arrow "1" in Figure 4-1). In this case the task *mac-tx* is a sink task. The queuing of a packet in *mac-tx* is an event for the task graph *arp-tx*, *eth-tx*: An ARP request will be issued (arrow "2" in Figure 4-1). Once an answer is received, the processing of the packet can continue (arrow "3" in Figure 4-1). The task *mac-tx*, *eth-tx* will be activated and the packet can leave the system (arrow "4" in Figure 4-1). Table 4-1 gives a description of the tasks in the two port Ethernet router. As this simple example shows, the complexity of packet processing systems is not to be underestimated.

Table 4-1: Tasks in the two port Etherent router

Task	Description
eth-mac-rx	Receives an Ethernet packet and processes the Ethernet header. Depending on the type of payload, the next task is either arp-rx or ip-header-check.

Task	Description
ip-header-check	Verifies that the IP header of the packet is correct.
ip-forwarder	Forwards packet based on the destination IP address. Has a forwarding information base.
mac-tx	Adds the Ethernet header to the IP packet. Looks up the ARP data base to for the next hop Ethernet address (MAC Address). If the lookup fails, it queues the packet and generates an event that will trigger the arp-request task graph.
arp-rx	Receives an ARP packet and verifies its correctness. If it is an ARP reply, the ARP information is stored in the ARP data base and the packet is consumed. If it is an ARP request packet and the system is the target of the request, the packet is forwarded to the arp-tx task.
arp-tx	If the arp-tx task is triggered by a packet, an ARP reply for that packet will be generated. Otherwise, it will generate an ARP request.
eth-tx	Transmits a packet.

Now we would like to transform that application model to an implementation. To do this, we need the instances of the tasks. We have to map the functional model (our task graphs) to an implementation model that consists of instantiated tasks. Figure 4-2 depicts a possible, natural implementation model of the application. To distinguish an instance of a task from its functional representation, a task instance is depicted with a square instead of a circle. In Figure 4-2 there are separate task

instances for all the tasks but the *ip-forwarder* task. The *ip-forwarder* task instance is shared, as it has a switching function. However, this is not mandatory: One might think of separate *ip-forwarder* task instances that contain a distributed forwarding data base. The ARP data base is a data base per Ethernet port. These data bases need to be accessed by the *arp-tx* and *mac-tx* task. Usually, the *mac-tx* task contains a mirrored, passive data base of the original data base, which is located in the *arp-tx* task.

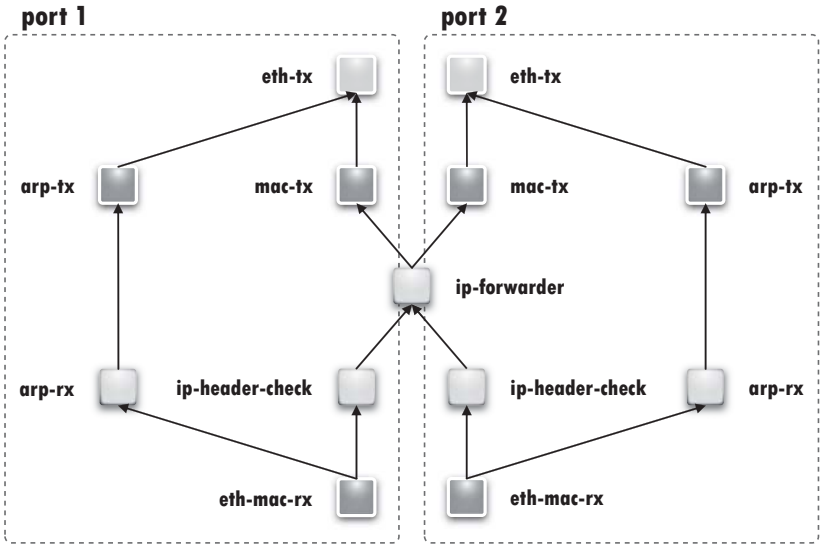

Figure 4-2: Possible implementation model of the two port router

Another possibility of an implementation model is shown by Figure 4-3. Here, all task instances are shared but the Ethernet port receive and transmit task instances (*eth-mac-rx* and *eth-tx*). To make this implementation model work, the incoming port information has to be annotated into the packet that traverses the instantiated tasks. Further, the forwarding task instance

has to annotate the outgoing port into the packet such that the *mac-tx* task instance can multiplex the packet to the correct sink task instance. Similar, the *arp-tx* task instance has to multiplex packets to the correct *eth-tx* task instance.

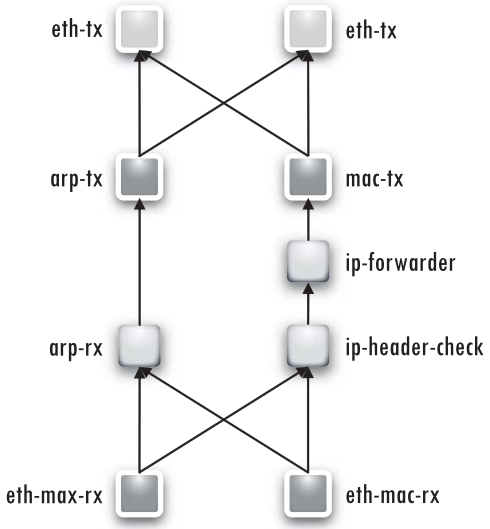

Figure 4-3: Another implementation model of the two port router

As this example shows, there are several possibilities how the implementation model of an application can look like. Before we give an overview of the pros and cons for sharing tasks[3] (instances), we discuss the ability of tasks to be sharable. Basically, there are three types of tasks with regard to their ability to be shareable:

1. Tasks that have no context information.

[3] For better readability we omit the word "instance" in the context of task instances from now on.

2. Tasks that have local context information, e.g. local statistic values.
3. Tasks that have global context information, e.g. a forwarding information base.

Tasks that have no context information

It is obvious that these tasks can be shared any time, as there is no information that is stored in its context. However, by sharing such tasks we loose information about previous tasks; each task has only one input, therefore we do not know which task preceded the current task. If we need this information, the previous task can annotate the packet with that information.

Examples for tasks with no context information are pure data processing tasks that do not have statistic values, e.g. tasks that do packet compression or encryption.

Tasks that have local context information

These are the majority of tasks. Most of the context information is in form of statistic values that are part of the MIB (Management Information Base). To make such tasks shareable, some sort of instance information has to be carried in the packet, either part of the data of the packet or as annotated information by a previous task. With this instance information it is possible for a task to have separate local contexts for each of the required instances. If this is feasible from a software engineering point of view depends on the actual functionality of the task and on the other tasks in the system. There is more effort involved to manage the different contexts manually, rather than letting the compiler handle it (assuming that an object oriented language like C++ is used for the implementation). Additionally, as we will see later, the available space for annotation of information in a packet is limited. Also, the precondition that certain information has to be annotated in the packet limits the independency of the tasks from each other. Considering all this, it is preferable to have separate instances of this type of tasks, rather than sharing a single instance.

Examples for tasks with local context information are ip-header-check, ip-fragmentation, access-control etc.

Tasks that have global context information

These are typically tasks which have a switching functionality[4]. The switching decision is based on an information store which is global to the system. Usually, these tasks are shared and require that some information about the previous tasks is annotated in the packet, e.g. the information about the incoming port. It is possible to separate these tasks. However in that case they need a distributed implementation of the information store, or they have to access a “global” resource outside of the task.

Examples for tasks with global context information are ip-forwarder with its forwarding information base or parts of the routing protocols which usually have a distributed implementation of the routing information base.

Table 4-2 gives an overview of typical tasks in packet processing systems and the type of task in respect to context information.

[4] Remember that tasks can not have more than one input; a task resembles a de-multiplexer (at most one input, zero or more outputs).

Table 4-2: Context information in packet processing tasks

Task	Context	Description of Context
eth-mac-rx	Local	Statistic values.
ip-header-check	Local	Statistic values.
ip-forwarder	Global	Statistic values and forwarding information base.
ip-fragmentation	Local	Statistic values and packet queue for fragmentation.
eth-tx	Local	Statistic values.
mac-tx	Local	Statistic values and ARP database.
decryption / encryption	None	-
acl-rx / acl-tx	Local	Statistic values and filter database.

Overview pro/cons for sharing of tasks

As a basic rule, it makes no sense to share tasks (with a few exceptions). The reasons are that the effort spent on making tasks sharable is better spent somewhere else and that there are no cache effects that we can profit from when sharing tasks. On the contrary, the context administration in sharable tasks adds additional instructions that have to be loaded and executed. The following are exceptions for which it makes sense to share a task:

1) "Switching" tasks should be shared as it better represents the natural packet flow. This makes it easier to understand, develop and maintain the software.
2) Tasks accessing a physical resource are usually shared as they represent a single logical abstraction of a single physical resource. Nobody would instantiate multiple drivers to access the same hardware resource.
3) Tasks that logically belong together as they share the same local context information should be shared, otherwise the context information becomes global and is accessed by all these tasks. An example is the *arp-tx* task from the two port router model (see Figure 4-1). For each port the *arp-tx* task is present in two task graphs, however, they work on a common arp database. Therefore it would not make sense to have separate instances for these tasks. Another example is the *mac-tx* task, which may queue a packet and sometime later process the queued packets. The packet-queue is the local context information that has to be shared between the two *mac-tx* tasks for each port. Again, it would not make sense to instantiate two tasks.

In summary, we propose to create the implementation model based on the following rules.

Tasks are shared if they

1) are a representation of a physical resource, e.g. network interfaces
2) share local context information, i.e. logically belong together (usually this happens if there are source tasks which create packets that will travel along paths that are identical to paths in an other task graphs)
3) have global context information and a "switching" function

4.1.2 Annotation of Instance Information

We will back-annotate the instance information into the task graphs. To uniquely identify it, each task graph receives an identification number (Definition 18). In addition, each task receives a unique identifier, which consists of the task graph identifier of its task graph and separated by a dot, a second identifier, which numbers the task inside the task graph over the same task type (e.g. *arp-tx*). The first task of a given task type in a task graph is numbered "1", the second task of the same type will receive a "2". Pre-pending the task name to the task identifier allows to uniquely identifying any task in the application (e.g. *arp-tx.2.1*). The complete identifier therefore follows the format <*task-type*>.<*task-graph-id*>.<*number*> (see Definition 19). Each task also receives a set of task identifiers, which at the minimum contains its own task identifier. If a task instance is shared, the set of task identifiers must contain the task identifiers of those tasks with which the task instance is shared (see Definition 20). As an example, Figure 4-4 depicts the annotated task graph based on the implementation model given in Figure 4-2. As only identical tasks can be shared, the task name in the task identifier can be omitted in the task identifier set.

Definition 18

Task Graph Identifier

Each task graph has a unique task graph identifier. It consists of the number from the numeration of the task graph.

Definition 19

Task Identifier *Each task has a unique task identifier. It consists of the task name, the task graph identifier and the number of numeration of identical tasks in the same task graph.*

Definition 20

Task Identifier Set

Each task has a set of task identifiers. It contains at the least its own task identifier. If the instance of the task is shared with other tasks, it contains the task identifiers of those tasks with which it is shared. All tasks that have the same instance have an identical set of task identifiers.

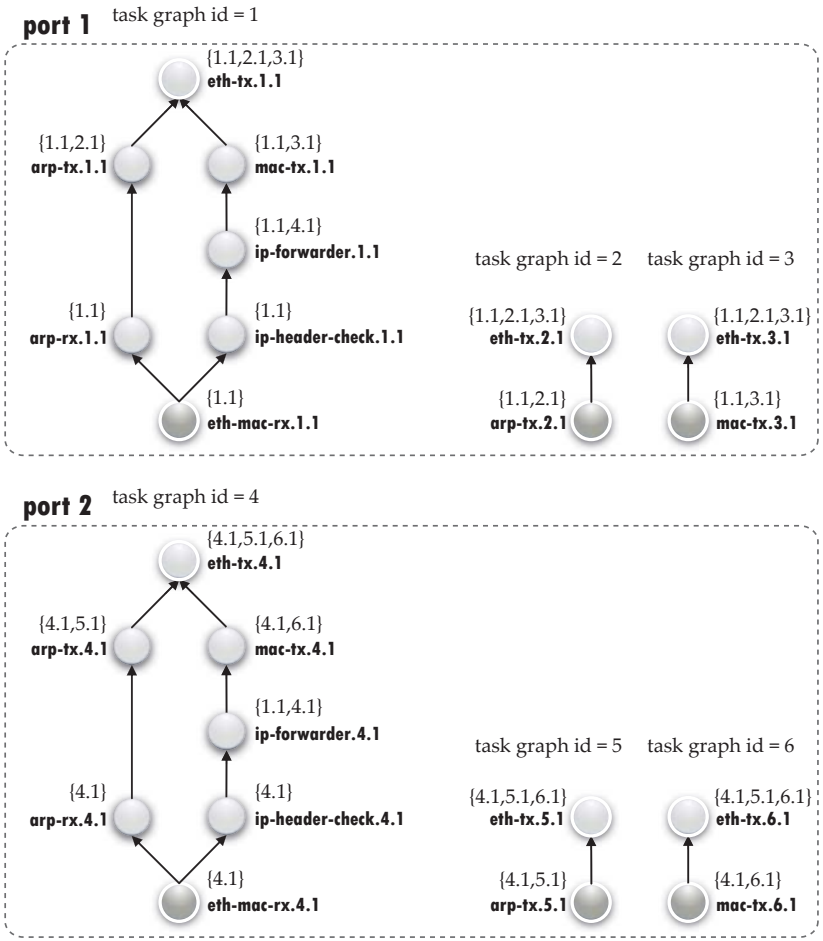

Figure 4-4: Task graph of a two port router annotated with instance information

4.2 Scheduler and Path-Threads

The available processing resource must be assigned to the processing of tasks such that the implementation adheres to the agreed quality of service parameters (delay). Some packet processing architectures do not require a scheduler as they follow a simple first come first served packet processing scheme [13, 14, 16, 25]. Others use a light-weight thread model to schedule and control the processing of packets [7, 22, 23]. We use a model that is similar to the light-weight thread model. To closely control the processing of a packet, we need a thread for each packet in the system. The thread, called path-thread in our model, is a very light-weight implementation; it does not have its own stack or memory space and is only preemptible at task boundaries (see Chapter 2). When a path-thread is scheduled, it processes one task before surrendering the control back to the scheduler. Besides the processing of packets, the path-thread polices the packet behavior. A packet is only allowed to travel paths as specified in the flow (see Chapter 2).

When a packet arrives in the system, it is classified to a flow. Then, a path-thread for that flow is created and the packet is bound to that path-thread. The path-thread is then ready to be scheduled (see Figure 4-5).

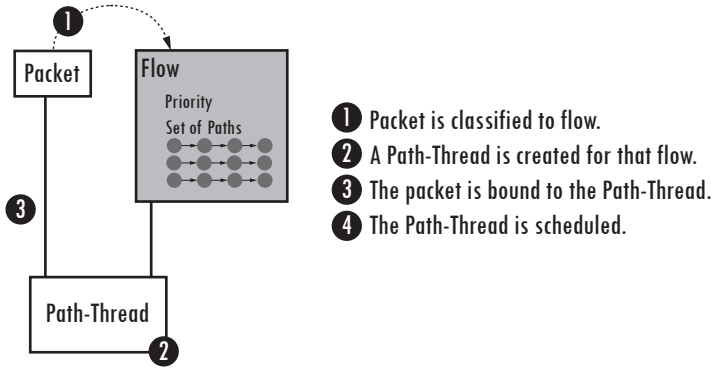

Figure 4-5: Packet and Path-Thread

The scheduler's decision which path-thread to run next depends on the quality of service information of the flows (each path-thread belongs to a flow) and the scheduler's algorithm. A simple implementation can use a fixed priority algorithm; in that case each flow must have an assigned priority (compare Chapter 3). A more sophisticated scheduler would use EDF (earliest deadline first); each flow must have a relative deadline (see Chapter 2) from which the absolute deadline is calculated on creation of the path-thread. As the tasks are non-preemptive (the path-threads are preemtible at task boundaries only), the granularity of the tasks provides the preemption points for the EDF scheduler. Non-real-time flows have an infinite deadline and therefore only get executed when there is no pending packet of a real-time flow.

4.3 Source Flow

When a packet arrives at the system, it is not known to which flow it belongs. Only later, when the packet has been analyzed, typically by a classifier, the association of the packet to its flow becomes clear. However, without classification of the packet to a flow we cannot create a path-thread and the packet is not processed. Therefore we classify packets that arrive at the system to a source flow. There is a source flow for each packet source that is the origin of packets of more than one flow. A source flow has exactly one path. This path contains all tasks until the packet is fully classified to a flow. The paths as specified by the model are split at the flow classification point. The parts before the classification point are assigned to the source flow. Figure 4-6 depicts an example for the splitting of the paths. The path from the source to the classifier is assigned to a source flow. The other paths are the existing but shortened paths; they start at the task following the classifier.

A packet that arrives at the system is assigned to the source flow. Then a path-thread is created and the packet is bound to that path-thread. Later, when the path-thread processes the classifier task, a new path-thread is created, based on the classification result. The packet is bound to the new path-thread, while the old path-thread is destroyed, as it executed the last task in the packet's path as member of the source flow.

A source flow inherits the maximum priority of all flows a packet in the source flow later might be classified to. This is because we have to assume that every packet that arrives could be a high priority packet. Note that the arrival curve for the source flow is typically defined by the physical interface.

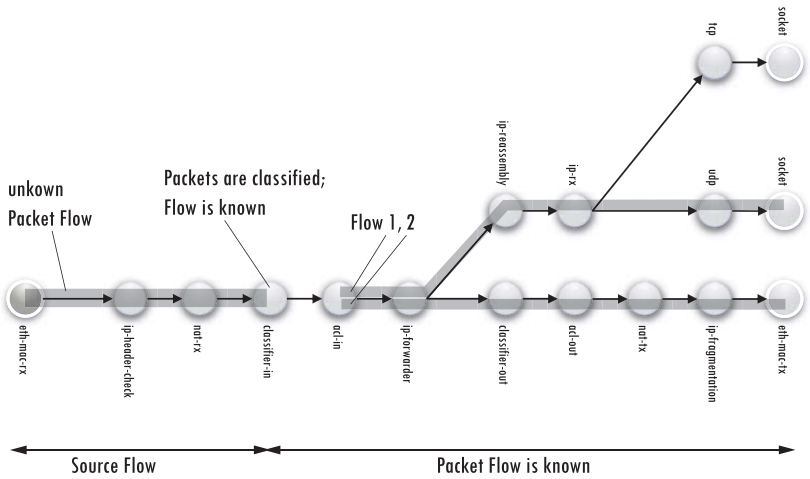

Figure 4-6: Source Flow from Splitting of Path

4.4 Summary

For an implementation, the original application model has to be transformed to an implementation model that represents the actual physical representation of the tasks. Here we need to know which tasks have to be instantiated and in what configuration. This process is not straight forward; different instance configurations/compositions with similar results are possible. The final implementation also depends on the design of the tasks. Tasks can be implemented such that they can handle multiple contexts or not. Both variants have their advantages and disadvantages. We have given some guidelines for the instantiation of tasks; finally it is also a designer's choice.

A scheduler controls the assignment of the processing resource to the packets in the system. It uses path-threads, a very light-weight thread model, to process the packets; for each packet there is a path-thread.

The fact that the flow of a packet is not always known when it enters the system was omitted in the original model. For the implementation, the mechanism of the source flow is introduced: Until the flow of a packet is known, it is assigned to a source flow, which inherits the priority of the maximum priority a packet might later be classified to.

5 Software Platform RNOS

In the previous chapter we have discussed how our model can be transformed to an implementation and what additional elements are necessary to do this. In this chapter we present the software platform Real-time Networking Operating System, in short RNOS, which supports the transformation of the model to an implementation. It provides the necessary infrastructure and an application programming interface to implement applications based on the model presented in the previous chapters. The approach of RNOS is characterized by two properties:

1. The software platform is constructed in a way that matches the analysis model. Therefore, the performance properties of the resulting implementation are within the calculated bounds.
2. The programming interface reflects the abstraction provided by the application structure, i.e. task graphs and task paths, and the input model, i.e. flows.

This chapter is split into six parts. In the first part, we look at the elements of RNOS. These elements represent the programming interface, most of them having a direct representation in the model, and the core of RNOS. They allow a direct transformation of the model to an efficient implementation. The means by which this is reached and how even complex applications can be modeled and implemented are described in this part.

This thesis targets small low cost packet processing systems and this is also where we will use RNOS. As we have stated before, packet processing is only a (possibly small) part of the complete application of the system. Such systems usually do contain a standard real-time operating system. The second part of this chapter describes how RNOS can be integrated into such standard real-time operating systems.

A user that designs and implements applications based on our model and RNOS works with tasks, tasks graphs, flows and other packet processing and model related elements. We could say that RNOS provides a higher level of programming abstraction than a standard operating system. In the third part of this chapter we will discuss these abstractions and compare them to the elements of a standard real-time operating system.

In the fourth part of this chapter we present some of the advanced features of RNOS. They include basic instrumentation of RNOS and a mechanism that allows to influence the preemption points and, as we shall see later, also the system performance properties.

Software platforms including RNOS are not free of overhead. In the fifth part of this chapter we look at the schedulablity region of RNOS. We investigate where overhead occurs in RNOS and how it influences throughput and delay. Further we analyze the worst-case delay of a highest-priority packet in a RNOS based system.

In the last part of this chapter we show how to use the analysis presented in Chapter 3 in combination with RNOS. The result is a continuous design process from model, to analysis and to the implementation that allows building predictable packet processing systems.

5.1 Elements of RNOS

RNOS is a software platform. As such, it provides a construction kit to build applications based on the model presented in the previous chapters. A target of this thesis is the analysis and the seamless design/implementation of packet processing applications. Therefore, the elements of RNOS reflect the abstractions provided by the model, such that there is a direct, predefined path from the model to the implementation. The elements are

- tasks and task graphs,
- packets,
- flows and path-threads,
- source flows and source-threads,
- and the scheduler.

These are exactly the elements of the model (see Chapter 2) and the additional elements required for implementation (see

Chapter 4). Here, we present their concrete implementation as elements of the software platform RNOS.

RNOS is based on Embedded C++ [49], which is a standardized subset of the C++ language that is optimized for embedded systems. Language constructs which result in high memory usage or unpredictable processing times are excluded in Embedded C++.

5.1.1 Tasks and Task Graphs

The tasks and the task graphs represent the processing elements and the application structure. The design targets for the tasks and the task graphs are

- ease of use and
- efficiency.

As the application structure can become quite complex (see Chapter 2), the ease of use is an important factor for the usability of the software platform. Complex structures can be simplified by introducing hierarchies; complexity is hidden by encapsulating application-parts in hierarchical entities. However, it must not lead to a less efficient software platform: efficiency is our primary concern. If applications based on the software platform RNOS were not efficient, the usefulness of RNOS would be limited. The two design targets seem to conflict; we will see how RNOS is able to solve the conflict in the next sections.

Task Object

The basic element of the model is the task. In RNOS tasks are represented by task objects. A task object has at most one input connector and zero or more output connectors. The con-

nectors reflect the connection hooks as defined by model. Remember that an input connector can be the sink of multiple output connectors. Figure 5-1 shows four task objects with their input and output connectors.

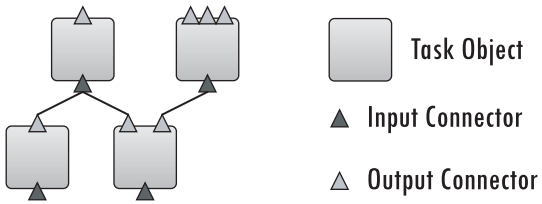

Figure 5-1: Task objects with connectors

It is obvious that not all tasks (task objects) can be connected in an arbitrary configuration. Tasks have some preconditions to what kind of input they can process. One of the preconditions is the content or payload type of the packet, e.g. an IP forwarder task requires IP packets as input. We associate a payload type to each input and output connector. A packet that leaves a task object through an output connector must be of the payload type associated to that connector. Likewise, a packet that enters a task object through the input connector must be of the associated payload type. Therefore, only connectors with equal associated payload types can be connected. As a consequence, a task object can only process packets of one payload type.

Some task objects require information about packets that are calculated in another task object. To that purpose, the model allots the annotation space that is available in each packet. Information can be carried and transported by packets along their path. In RNOS, each input connector has an associated payload type and an associated list of required information that is calculated elsewhere. Each task that calculates information that might be used later along the path has an associated list of that information on its output connectors.

In order to build a valid task graph the following conditions apply:

1. For each connection between two task objects, the output and input connector must have equal associated payload types.
2. A task object that requires information about packets that is calculated in another task object can only be connected to the task graph if the task object which provides that information lies along the path to that task object. Concretely, all items on the associated list of information on the input connector have to be found on the associated lists of the output connectors on previous task objects.

Figure 5-2 depicts a simple example of correct task graph.

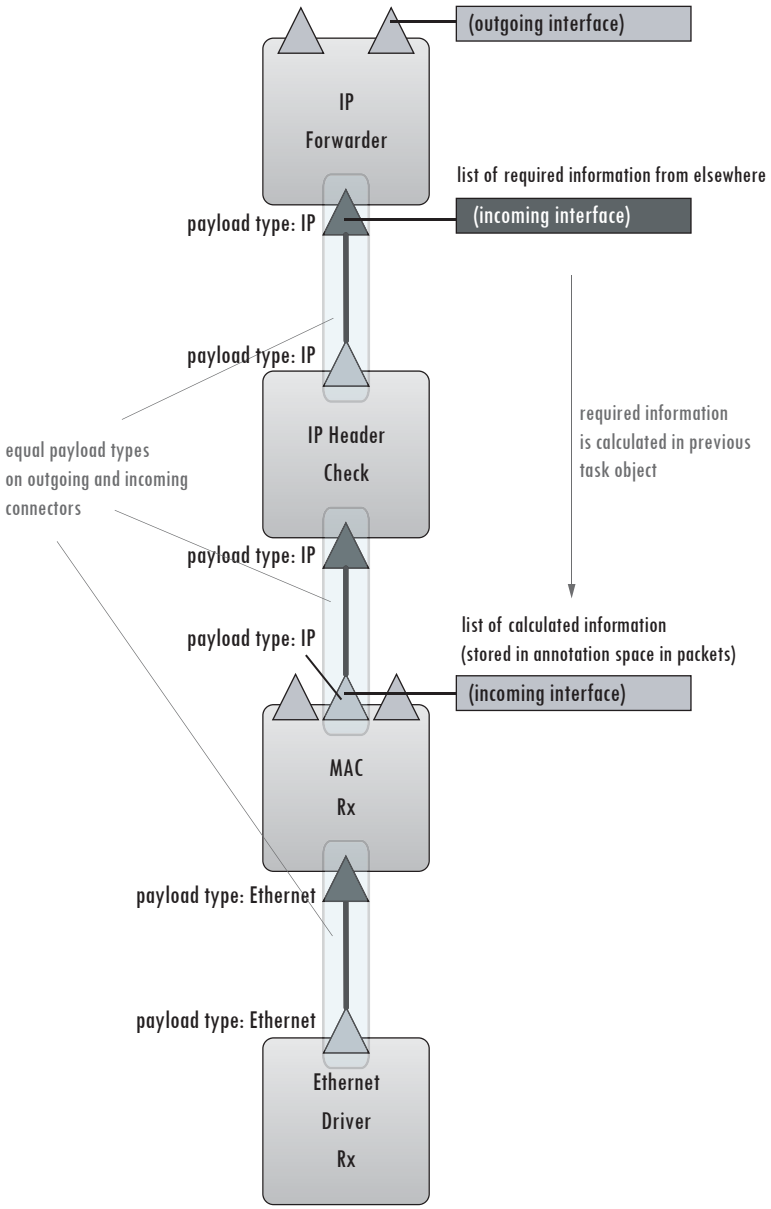

Figure 5-2: Connectors must match in payload type and required annotations

Task Frames

To be able to handle complex application structures, RNOS provides an element called task frame that allows grouping of task objects. Complex applications are divided into sub-applications and an arbitrary number of hierarchical levels encapsulate and hide details. Figure 5-3 shows the basic concept of task frames. A task frame has zero or more input connectors (in contrary to the task object which has at most one input connector) and zero or more output connectors. It contains zero or more task objects or task frames. Therefore, task frames can be nested and the functionality of an application can be implemented in a top down approach.

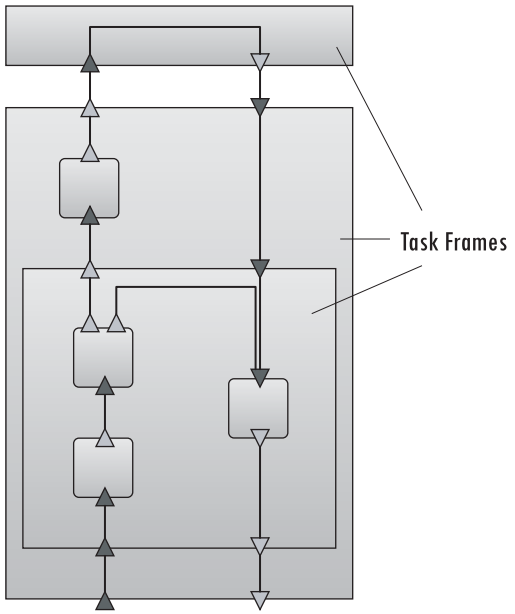

Figure 5-3: Basic Concept of Task Frames

Figure 5-4 shows an example of a link layer, consisting of Ethernet, ARP and PPPoE processing. The Ethernet frame con-

sists of two other frames. One handles the MAC protocol, while the other takes care of the hardware interface. The Ethernet frame has three output connecters: IP, PPPoE and ARP. It has two input connectors: MAC and Ethernet. The task frames allow the programmer to hide the (sometimes complex) functionality, and the connectors provide the interface to the task frames.

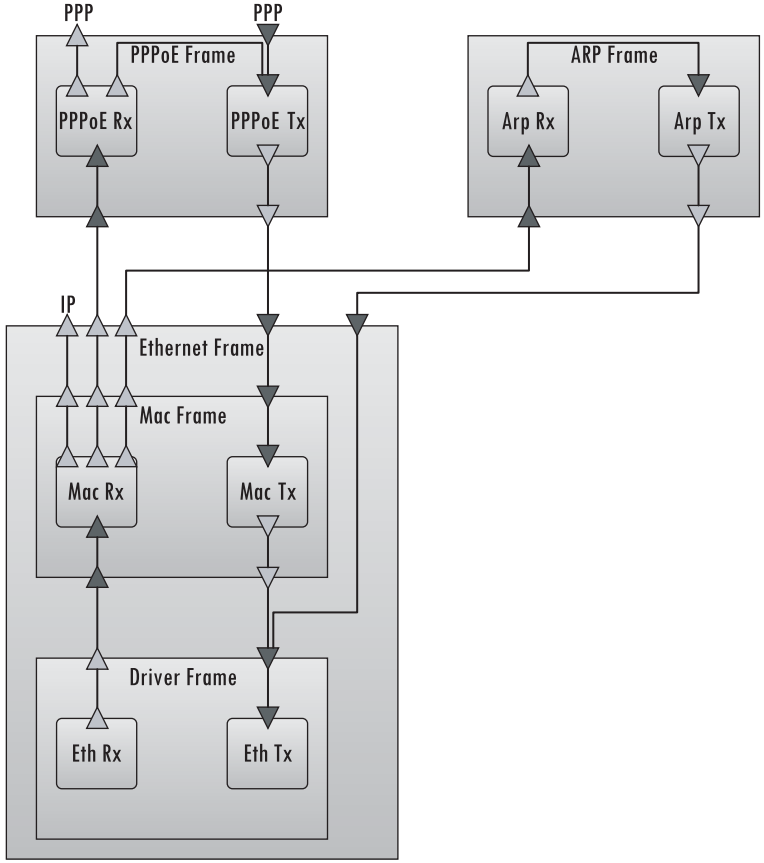

Figure 5-4: Link Layer with Task Frames

Other packet processing frameworks as [13, 14] do not provide hierarchical elements on a programming level. [14] provides a

tool for search-and-replace of parts of a task graph. The idea behind it is to replace a collection of slow elements with a single, more specialized, faster element. They use patterns, which essentially are single task frames, to group a collection of elements. However, these patterns are only used as an off-line tool and not on a programming level.

It is obvious that the connectors and task frames would impose a heavy penalty on the efficiency of RNOS. This is why RNOS follows the approach of separating the administration of the application structure from the actual packet processing part. What has been presented in the previous paragraphs is the administrative part of tasks and task graphs, the packet processing part is presented in the following sections. We will see how the two parts are connected and how they yield to high efficiency while preserving the design goal of "ease of use".

Task Object and Packet Processor

The first step to optimize for efficiency is concerned with the task object. As we have learned, the task object is responsible for the administrative part of the application structure. Aggregated into the task object is the packet processor. The packet processor is the actual processing engine of the task. Figure 5-5 figure shows the packet processor and how it is aggregated in the task object. The separation of the processing engine from the administrative part (the task object) has several advantages. The separation makes it easier to optimize the processing code, as it is isolated. Further, the processing code can run in a different memory region (faster memory or locked cache), a different CPU or even use a hardware accelerator. While the

task object is generic, the packet processor is tailored and optimized for its function.

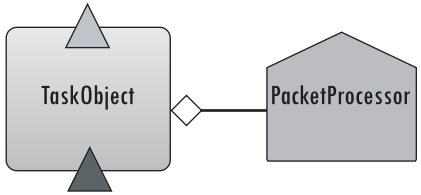

Figure 5-5: Task Object and Packet Processor

The second step to optimize for efficiency is concerned with the flow of packets trough the application structure. The path of a packet through a task object graph is not from connector to connector, but directly from packet processor to packet processor. Thus the path through the hierarchy of task frames and task objects is optimized, such that packets take the shortest possible path, i.e. there is no overhead associated with task frames (hierarchies) and connectors. Figure 5-6 shows an example for the actual path a packet will take in a task object graph. As we can see, the administrative part does not impose an overhead to the packet processing, i.e. a task frame with no task objects and therefore containing no packet processors is completely removed from the packet path.

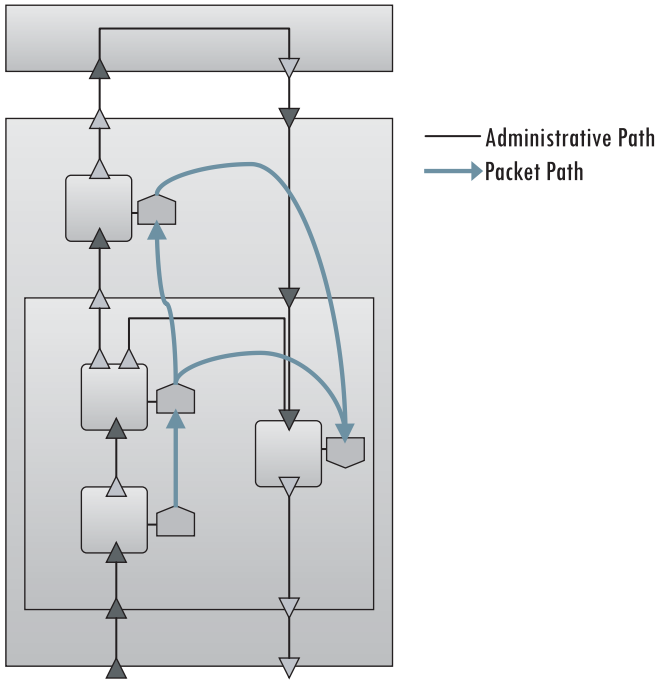

Figure 5-6: Packet path optimization

RNOS automatically creates the direct connections between the packet processors at runtime when task objects are connected.

In summary, although RNOS supports hierarchical structures, it does not impose a runtime penalty. The possibility to hide complex protocols inside task frames makes RNOS more usable.

Note that RNOS is built such that task objects can easily be exchanged at runtime. This is useful when the current configuration of the system allows replacing a task object for reasons of processing time and resource usage, e.g. with one that has reduced but sufficient functionality or the other way round.

5.1.2 Packets

Packet processing is the target application of this thesis. Therefore much thought has been put into how packets are stored and managed inside the system. The related objectives are

- support for different hardware architecture and buffer schemes,
- transparent access to packet data and
- a possibility to annotate information to the packet.

First let us review how packet data is accessed during its lifetime in the system. Each task of a task graph will only access (read and/or write) a part of the packet. For example, the task that processes the Ethernet header of an incoming packet will look only at the Ethernet header part of the packet. Similarly, the IP forwarder task will consider only the IP header of the packet. More generally formulated, each task has a certain scope on the packet content. In a previous section we have spoken of the payload type that a task object expects at its input. The way how this is accomplished is as follows: A pointer defines the beginning of the current scope, e.g. it points to the beginning of the Ethernet header. Each task that completes the processing of the current scope moves the pointer to the next scope. Fortunately, packet processing is a sequential process; the next scope starts at the end of the current scope. Packets are processed from the outside to the inside and reverse. This is shown in Figure 5-7, where the left two illustrations show a packet that is processed from the outside to the inside and the right illustration shows a packet that is processed from the inside to the outside.

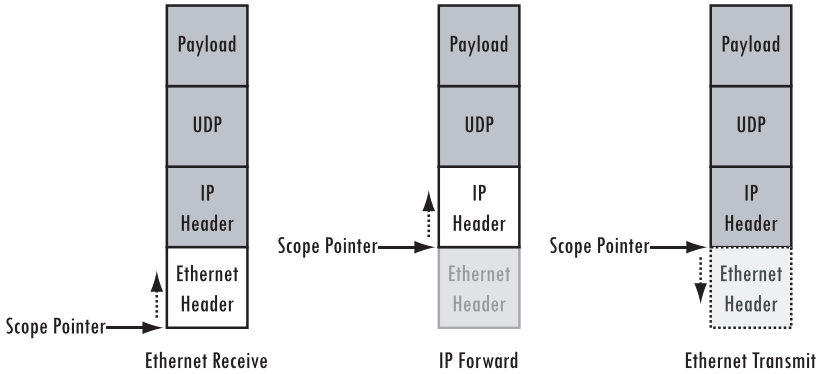

Figure 5-7: Scope on packet data

The payload type of a packet is defined by the current scope on the packet. It is obvious that the pointer which defines the current scope has to be stored somewhere. In RNOS, this pointer is stored in an element called packet descriptor.

Packet Descriptor

To each packet a packet descriptor is associated. The packet descriptor is an important element in RNOS. The packet descriptor has the tasks to

- hide the underlying buffer scheme and hardware architecture,
- provide a transparent access to packet data,
- provide space for annotations and
- provide the possibility for caching often used values.

Essentially, the packet descriptor is an abstraction layer for the access of the packet data and its annotations. This abstraction layer provides the possibility to use buffer schemes as found in UNIX derivates (e.g. [50]) or simpler buffer schemes like fixed sized buffers. Also, as low cost embedded systems do not always support bus snooping, the actual packet data might

have to reside in a non-cacheable memory region. In the latter case, the caching of often used values in the packet descriptor (which can reside in a cached memory region) brings a significant performance advantage. In RNOS, annotations are the mechanism to cache values in the packet descriptor.

[14] has its own optimized packet buffering scheme that is not compatible with standard buffer schemes that is typically found in operating systems (e.g. [50], [53]). For pure forwarding application the optimized packet buffering scheme is probably slightly more efficient than RNOS' portable packet descriptor. On the other hand, as RNOS integrates into existing packet buffering schemes, no complex packet transformation is required for packets leaving or entering the RNOS controlled part of the software and the porting of existing code to RNOS requires less effort.

Annotations

The space for the annotations is part of the packet descriptor. For an efficient implementation, the packet descriptors are of a fixed size. Therefore, the space for annotations is limited. In a previous section we have learned that for each input and output connector there are associated lists of information that are calculated (in case of the output connector) or required (in case of the input connector). Information that is calculated is stored in the annotation space of the packet. The task object (later along the path the packet is traveling) that requires that information reads it from the annotation space. The last task along the path of the packet that requires the information removes it from the annotation space in order to free up space. To have an efficient implementation, the location of information inside the annotation space is static per compile time. With that we

can create simple access functions to the information in the annotation space. An offline annotation compiler helps us in optimizing the use of the limited annotation space. It assigns the annotation space optimally to information items.

The annotation compiler requires a textual specification of the task graph including the calculated information and required information list on the output and input connectors. The format is simple and has the following syntax:

a) declaration of task objects

```
name[number of output connectors]{comma separated
list of required information },{comma separated
list of calculated information on output connector
1}…{comma separated list of calculated information
on output connector n}
```

b) task graph: task object port -> task object

```
name1[output connector]->name2
```

The declaration of a task object includes its name "*name*", the number of output connectors "[*number of output connectors*]", the required information on the input connector "{*comma separated list of required information*}," and the calculated information for each output connector "{*comma separated list of calculated information on output connector 1*}...{}". Support for task frames is not yet included, but could easily be added in the future. The connection statement "*name1[output connector]-> name2*" creates a connection from the output connector of task object "*name1*" to the input connector of task object "*name2*". The task objects must be declared before they are used in connections. Example 5-1 shows the definition of the task graph shown in Figure 5-2.

Example 5-1: Simple Input File for Annotation Compiler

```
01 // declaration of task objects
02 EthernetDriverRx[1]{},{}
03 MacRx[3]{},{}{incoming interface}{}
04 IPHeaderCheck[1]{},{}
05 IPForwarder[2]{incoming interface},{}{outgoing
     interface}
06
07 // task graph
08 EthernetDriverRx[1] -> MacRx
09 MacRx[2] -> IPHeaderCheck
10 IPHeaderCheck[1] -> IPForwarder
```

The functionality of the annotation compiler can be compared to register coloring in standard compilers [51]. The differences are

- we do not want to move an annotated value from one place to another place in the annotation space and
- we want to have a global assignment of the annotated values to places in the annotation space.

This allows us to generate static access functions to the annotation space. Algorithm 2 is the core algorithm of the annotation compiler. It optimally assigns annotation values to places in the annotation space. It relies on the notion that an annotated value is valid (active) for certain path fragments only. While an annotated value is valid at a certain task, it is called active, else it is called passive (see Definition 21 and Definition 22).

Definition 21

Active	*An annotated value is called active at a given task in the task graph if the annotated value will be used after this task.*

Definition 22

Passive	*An annotated value is called passive at a given task in the task graph if the annotated value is not used in this task and after that task.*

Algorithm 2: Coloring of Annotation Values

```
01 L = List of annotation values
02 e = head element of list L
03 remove head element in list L
04 put e in list A
05 n = 0
06 assign e place n in annotation space
07 While list L is not empty Do
08   e = head element of list L
09   remove head element in list L
10   For each element s in list A Do
11     failure = false
12     For each path through each task graph
                 and not failure Do
13       For each task in the path
                  and not failure Do
14         If e and s are active at task Then
15           failure = true;
16         End If
17       End For
18     End For
19     If not failure Then
20       assign e place of s in annotation space
21       break
22     End If
23   End For
24   put e in list A
25   If failure Then
26     assign e place n in annotation space
27     n = n + 1
28   End If
29 End While
```

5.1.3 Flows and Path-Threads

For a packet of a specific flow there are one or more allowed paths through the task graph. The path-thread, to which the packet is bound, observes the path of the packet and will drop the packet if it leaves the allowed sets of paths. Otherwise a misbehaving packet stream could compromise the quality of service of other packet streams. A trivial implementation that compares the current path against all allowed paths (as specified by the packet's flow) is not very efficient. Two paths only match, if they have the same sequence of tasks, from source task to the current task. If the set of allowed paths contains more than one path, the overhead becomes substantial. The solution of RNOS is to introduce a concept called microflows.

Microflow

The microflow concept is based on the fact that packets of the same connection traverse exactly the same path through the task graph. A connection, as we recall, is defined as having the same incoming and outgoing interface, same source and destination IP address, the same transport protocol and the same source and destination port. In RNOS such connections are called microflows. Packets that belong to the same microflow will traverse exactly the same path through the task graph. The result of this concept is that to each path-thread there is an associated microflow. Thereby the path-thread can verify that a packet follows the path defined by the microflow. The advantage of having a defined path for a packet is that verifying against one path is much easier and uses up less computation capacity than verifying against a set of paths. Verification is necessary as we do not want misbehaving sources to compromise the real-time guarantees.

Figure 5-8 depicts the object relations of RNOS. Note that there are one to one relations between packet and path-thread and microflow and path. The microflow identifies a path for the flow and links it to the path-thread. Path-threads are short lived. They exist during the live-time of the associated packet. Microflows exist during the live-time of the connection (which usually consists of at least some hundreds of packets) or are static. Note that a flow usually contains many microflows. Flows on the other hand are static elements that are created manually or by a resource reservation protocol [52]. They carry the flow information, which essentially is the relative deadline or priority (depending on the scheduler, as we later will see) for packets of that flow.

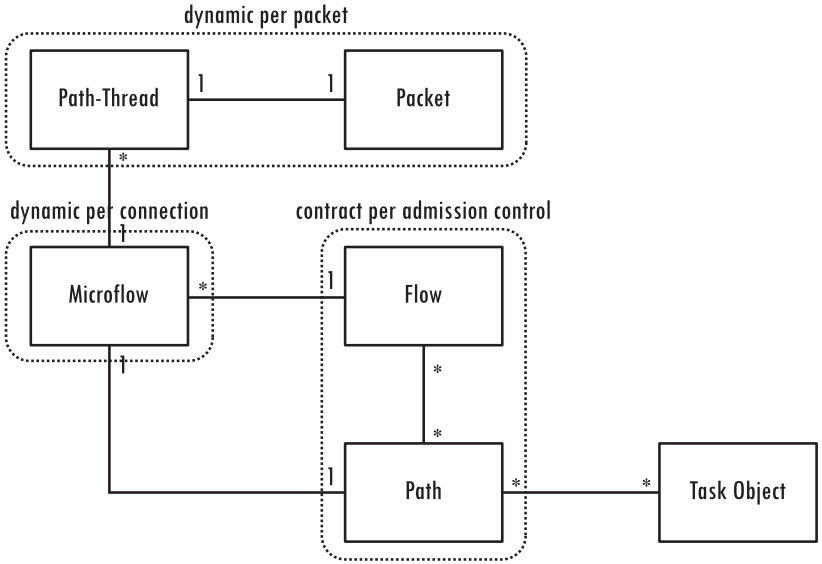

Figure 5-8: Object Relation Diagram with Microflow Object

RNOS provides two mechanisms to create microflows. One is to statically create a microflow and assign a path to it. The sec-

ond is based on online learning the path of a microflow. The basic learning sequence is as follows:

1. If the path for a microflow is unknown, a learn-thread is scheduled for this packet.
2. The learn-thread records the path the packet traverses through the task graph and compares it to the allowed paths for this flow. Should it leave the allowed paths, the packet is dropped.
3. Once the processing of the packet is completed, i.e. the packet has been consumed or has been sent out of the system, the state of the microflow is set to ready. From that moment on the path of the microflow is known.

What will happen when a packet leaves the path pre-defined by the microflow? The path of a microflow can change over time. As long as the new path is still in the path set of the flow the packet does not need to be dropped. The solution here is that when the path-thread observes that the packet leaves the microflow path, the microflow is put back in learning-mode and the new path taken is recorded.

As a flow might cover thousands of microflows, there might be a storage limit on how many microflows the system can handle at the same time. Depending on the application, microflows are short-lived. RNOS does a least recently used replacement of microflows. This means that there is another quality of service parameter that has to be considered, namely the number of maximum concurrent microflows or "connections" that the system must be able to handle. RNOS provides the option that best effort packets do not use the microflow feature. This has the disadvantage that path verification is

turned off for best effort packets. However, as best effort packets have the lowest priority, this is acceptable[5].

Learning threads do not influence the real-time behavior of existing flows; the learning threads have lower priority than standard path-threads. Misbehaving packet sources either never go beyond the learn thread, are easily policed within the microflow, or are rate limited by a policing object early in the task path. However, the setup of a microflow causes the first packet of a microflow to potentially violate the agreed quality of service. There are two approaches to overcome this possibility. First, the contract could specifically not include the first packet of a connection (microflow). Second, if the first approach is not acceptable for a given connection, we must use statically created microflows.

5.1.4 Source Flows and Source-Thread

In Chapter 4 source flows have been introduced. Source flows cover all packets that have an unknown flow association. These are those packets that arrive at the system and are not classified to a flow yet. As long as a packet is not classified, we have to assume that it belongs to the highest-priority flow. That means that the path-thread to which the packet is bound must run with highest priority. For simplification we call this path-thread "source-thread", due to its association to the source flow. Obviously, for a good system performance, the runtime of the source-thread should be minimized. Therefore, packets have to be classified to their flows as early as possible. Most packet processing implementations do not care about

[5] Buffer usage is also not a problem, as the buffer space for best-effort packets is limited.

this; they assume that there is enough processing power to classify packets at the worst-case packet rate [16, 24, 25, 27]. However, in our target domain of small, low-cost embedded devices, we cannot adopt this assumption. A common solution to shorten the path until the classification is to add a hash-based classifier as a first task (immediately after the reception of the packet). RNOS extends this concept in that it does not classify to flows but to microflows. In RNOS, this task is called filter-task.

Filter-Task

The filter-task works closely together with the learn-thread infrastructure. The filter-task matches a packet directly to its microflow. If no microflow exists yet, a learn-thread is scheduled. The filter-task is necessary for those packet sources that are input to more than one microflow. Typically, these are external interfaces, while internal sources usually "create" packets of a predefined microflow.

Figure 5-9 depicts a task graph with a filter task. The packet reception and the filter task run in a thread called source thread. This source thread inherits the highest priority any thread of that task graph might have. Based on the matching results, the filter task creates either a thread for the packet's actual microflow (in case a match was found), or a learn thread to learn the packet's path and create a new microflow. If path verification is turned off for best effort flows, it also can be a best effort flow thread.

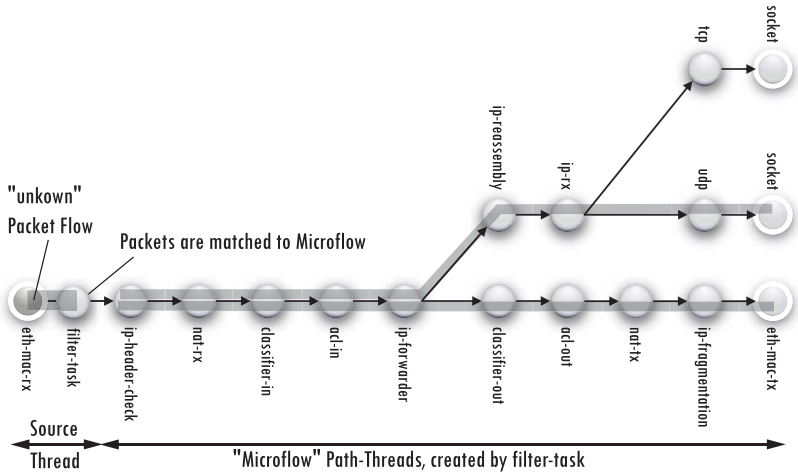

Figure 5-9: Source Thread creates Path-Threads

In contrary to standard path-threads, the source-thread is never destroyed; it is always present, although it might be idle. The source-thread is idle if there are no packets pending in interface input queues. When a packet arrives, the source-thread is woken and the packet is scheduled for execution with a minimal deadline. The received packet on executing the receive task is bound to the source-thread. Once the packet is classified to a microflow (by the filter-task) a new path-thread is created and the packet is bound to that new path-thread. If there are no pending packets in interface input queues, the source-thread becomes idle again.

5.1.5 Scheduler

To complete the picture of the model elements in RNOS, the scheduler is presented here. The scheduler is the core of RNOS as it controls the execution of its threads. In RNOS, for each flow there is only one path-thread in the set of schedulable threads. With that it is ensured that there is no reordering of

packets inside a flow. Note that threads[6] are not threads in the sense of operating system threads. They are very light-weight (as they must be; for each packet a thread is used) and do not impose much overhead (basically, the costs of a virtual function call[7]).

For each flow there is a waiting queue (first-in first-out) for path-threads. The scheduler will take another path-thread out of its waiting queue into its set of schedulable threads when a path-thread of that flow has terminated. Figure 5-10 depicts the RNOS scheduler architecture with an EDF scheduler using a priority queue. The depth of the priority queue is identical to the sum of the number of flows plus three (for the learn-thread queue, the source-thread queue and the best effort queue).

[6] The term "thread" is used here, and shall from now on be used, for any type of thread of RNOS (path-threads, source-threads, learn-threads, best-effort-threads).

[7] Virtual function call: The call of a virtual method in C++. Using single inheritance (multiple inheritances are forbidden in Embedded C++), it basically results into dereferencing a pointer and a jump.

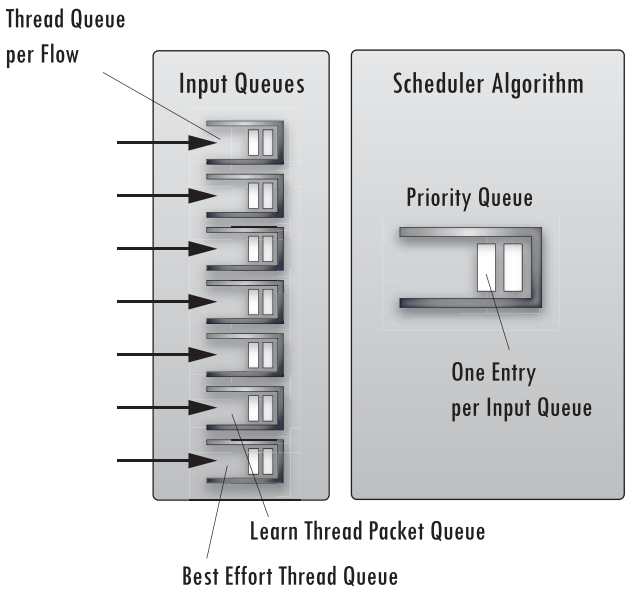

Figure 5-10: Scheduler Architecture with EDF Algorithm

Algorithm 3 gives the core loop of the RNOS scheduler. It gets the thread with the smallest deadline from the priority queue and lets the thread execute. The thread will surrender control back to the scheduler at the next preemption point[8]. If the thread still has tasks to execute, the thread is scheduled again (added into the priority queue). If a thread terminates, i.e. it has executed the last task of a path, the next pending thread of the same flow is added to the priority queue. If there are no pending threads in the priority queue, the scheduler is idle. It wakes up as soon as a thread is created. Note that the scheduler runs at each preemption point, regardless of whether a thread switch is necessary or not.

[8] Remember that tasks cannot be preempted; the preemption points lie between tasks.

The deadlines for the threads are calculated when the threads are created (the deadlines are based on the relative deadline defined by the flow, see also Chapter 2).

Algorithm 3: Scheduler

```
30 Forever {
31   pThread = priorityQueue.head();
32   if (0 != pThread) {
33     continue = pThread->execute();
34     if (true == continue) {
35       priorityQueue.add(pThread);
36     }
37     else { // thread has terminated
38       flowId = pThread->flowId();
39       pThread->terminate();
40       pThread= nextPendingThreadOfFlow(flowId);
41       if (0 != pThread) {
42         priorityQueue.add(pThread);
43       }
44     }
45   }
46   else { // idle
47     pendForThread();
48   }
49 } // forever
```

5.1.6 Summary

In summary, RNOS provides all the necessary elements to build applications based on the model described in the previous chapters. Most of the elements are a direct representation of the elements of that model. The objectives for the software platform RNOS are efficiency and ease of use. These objectives are achieved by a carefully designing of the RNOS architecture. First of all, efficiency is reached by separating the administrative part of the application structure from the actual data path through the use of packet processors. Elements such as the source thread with its filter task, the microflow infrastructure and the use of a packet descriptor contribute to the efficiency of RNOS. The threads in RNOS are very light-weight and do not impose a large overhead. Second, ease of use is reached by the introduction of hierarchical structure elements, called task frames. They allow grouping and hiding fragments of task graphs to aid in the building of complex applications. Last, the packet descriptor allows a transparent handling of packet buffers, regardless of the underlying hardware or operating system architecture.

Figure 5-11 gives an overview of some elements of RNOS. It shows the scheduler as central element with the path-threads that are scheduled. Packets, consisting of a packet descriptor and a buffering scheme, are bound to path-threads. Each path-thread belongs to a microflow and each microflow belongs to a flow. The three levels (path-thread with packet, microflow and flow) are necessary as they have different life-times. A path-thread exists only for as long as a packet is bound to it (with the exception of the source-thread). A microflow represents a connection between two endpoints and exists as long as this connection is active. The flow is a more or less static element; it is created by the admission control, either by configuration (manually) or by a resource reservation protocol (e.g. [52])

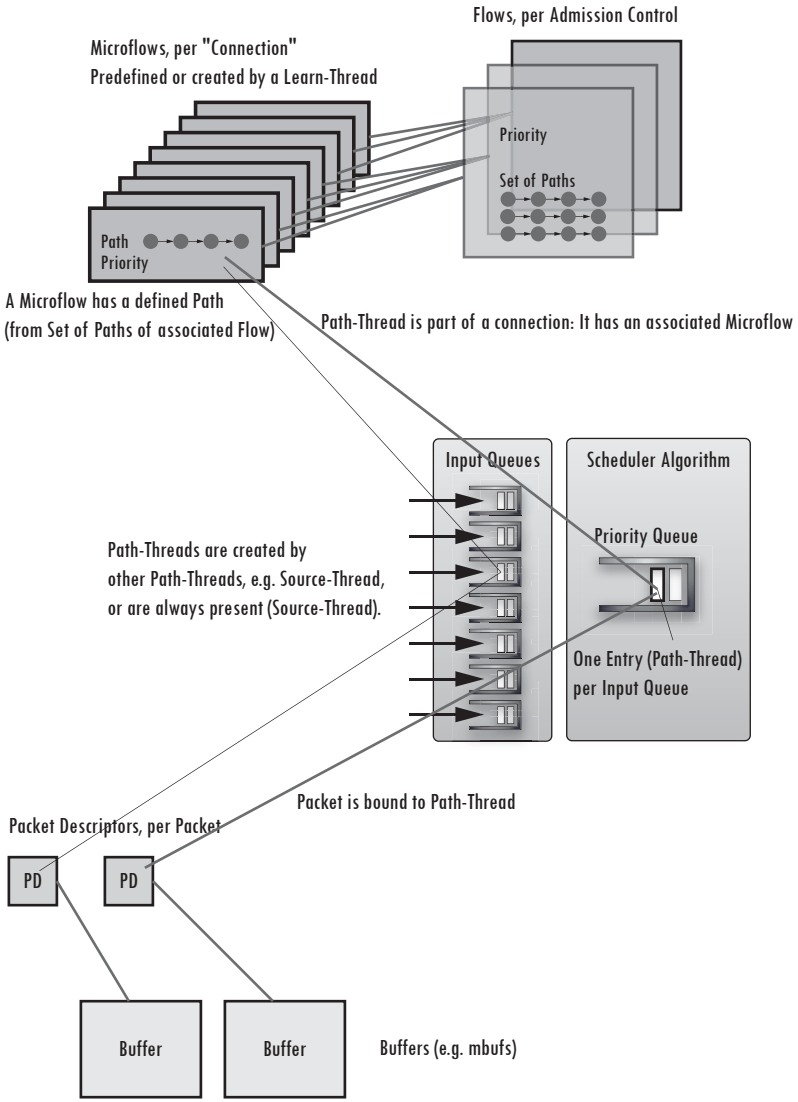

Figure 5-11: Overview of some Elements of RNOS

5.2 RNOS Integration with the RTOS

In the target domain of RNOS, packet processing is only a small part of the complete application. Therefore, RNOS has to be integrated with a standard real-time operating system. The resources of the system have to be shared between RNOS and other applications. The underlying real-time operating system provides RNOS with the computation resource, while RNOS redistributes that resource to its own threads. Therefore, RNOS runs in one (RTOS-) process (see Figure 5-12). Note that the threads in the RNOS process are not threads of the RTOS but RNOS threads as discussed in the previous subchapter.

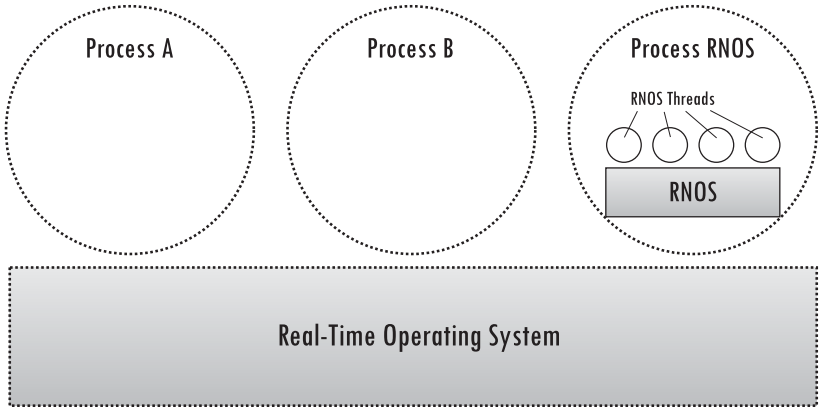

Figure 5-12: RNOS running in an RTOS process

As defined by the resource model, RNOS requests having a defined and guaranteed minimal access to computation resource. Typically, this is a minimum number of CPU time in a given time period. Figure 5-13 shows an example in with RNOS receives a CPU time t_1 in every period t_2.

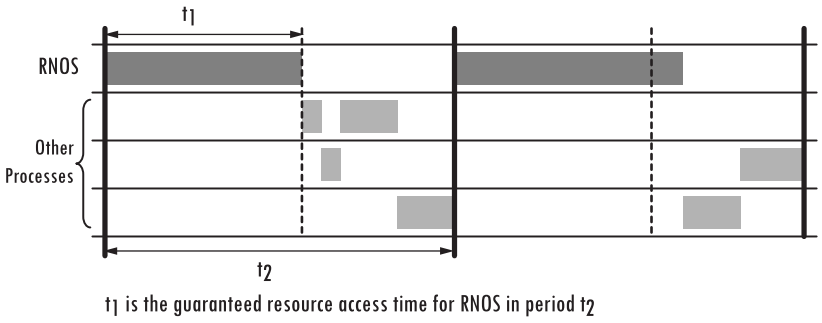

Figure 5-13: RNOS running on RTOS as a process

As the example shows, RNOS is allowed to run for longer than the minimum requested time t_1, but must never receive less CPU time than specified by t_1. Interrupts also have to be taken into account. An interrupt that occurs while RNOS is running will consume CPU time assigned to the RNOS process. Either interrupts are disabled during the runtime of the RNOS process, or they have to be included in the analysis of the system (see subchapter 5.5).

In summary, there is a single requirement to the underlying RTOS: to provide a minimum and guaranteed processing time to the RNOS process. This makes it possible to port RNOS to almost any available real-time operating system. In Chapter 6 we will see how the guaranteed access to the computation resource is implemented on top of a commercial RTOS.

5.3 RNOS: A higher Level Programming System

A real-time operating system provides an API (application programming interface) with processes, threads, semaphores, and for memory management, interrupt processing etc. RNOS

provides an API that is very domain specific and reflects the model presented in the previous chapters. Using RNOS, the programmer will construct his applications based on tasks by connecting them to task graphs and define flows and their quality of service parameter. Thus, RNOS provides a higher level of programming abstraction for the target domain of packet processing (see Figure 5-14).

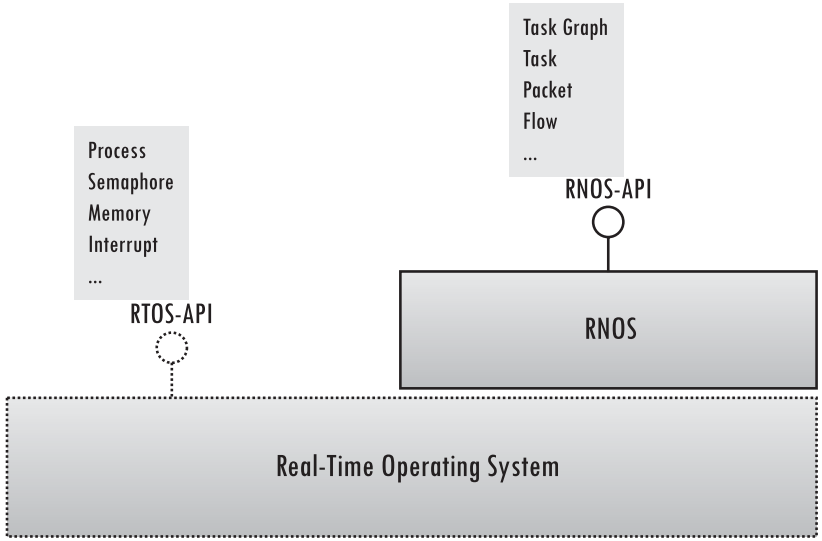

Figure 5-14: RNOS' higher level of abstraction API

Comparing an RTOS with RNOS we can see that RNOS has a similar structure to that of an RTOS. The difference is that the elements of RNOS are on a higher level of abstraction. Instead of instructions, RNOS executes tasks. Instead of a program, RNOS has task paths. The equivalent to a process in an RTOS is the RNOS thread. The program counter becomes a task pointer (each path-thread has a pointer to the task that will be executed next) and the registers and memory are replaced by packets and their annotations. Table 5-1 relates the elements of a standard real-time operating system with the elements of

RNOS. Other frameworks to build packet processing systems [6, 7, 13-16, 18, 20, 21-25] do not provide a full set of equivalent elements to an RTOS on a higher level of abstraction. They provide some elements which still need to be integrated and combined with standard RTOS elements. RNOS on the other hand provides a complete framework for packet processing that is similar in its form to frameworks seen in signal processing [59].

Table 5-1: Analogy between RTOS and RNOS

RTOS	RNOS
Instruction	Task
Program	Task Path
Thread	Path-Thread
Program Counter	Task Pointer
Thread Priority	Deadline
Registers and Memory	Packet and Packet Annotations

5.4 Advanced Features of RNOS

RNOS has several configuration parameters and built-in tools to optimize and explore system behavior. Some of those features are presented here, as they are important in the context of this thesis.

5.4.1 Virtual Tasks

As we have seen in a previous subchapter, the scheduler executes at each preemption point. It is obvious that this execu-

tion is not free and requires some processing resources. In the model, the tasks are non-preemptive. The task boundaries represent the preemption points. That means that a thread will execute one task and once the task is finished, the control is surrendered back to the scheduler. The granularity of the tasks becomes an important factor in the system. The fewer tasks there are in a task path (every task takes longer to execute as it covers more functionality), the less preemption points there are and, therefore, the less overhead is incurred by the scheduler. On the other hand, as we have seen in Chapter 3, the worst case delay is influenced by the task with the longest processing time. There is a tradeoff between efficiency (negatively influenced by overhead) and delay (negatively influenced by the longest task). We will look at this more closely later.

RNOS allows for the grouping of tasks into so called virtual tasks. As a consequence, the preemption points are defined by the boundaries of the virtual tasks, giving the programmer the freedom to choose the task sizes and group them together into virtual tasks. To simplify this, RNOS allows an automatic creation of virtual tasks by specifying a minimum worst case processing time of virtual tasks. RNOS then creates at runtime the virtual tasks by traversing all task paths and grouping tasks together to virtual tasks, such that their worst case processing time is at least the specified time limit. This mechanism allows having small tasks in respect to processing time while it reduces the disadvantage of too many preemption points and the overhead associated with these. Figure 5-15 depicts an example of virtual tasks in a task graph. It also shows that a task may be part of more than one virtual task for different paths through the task graph.

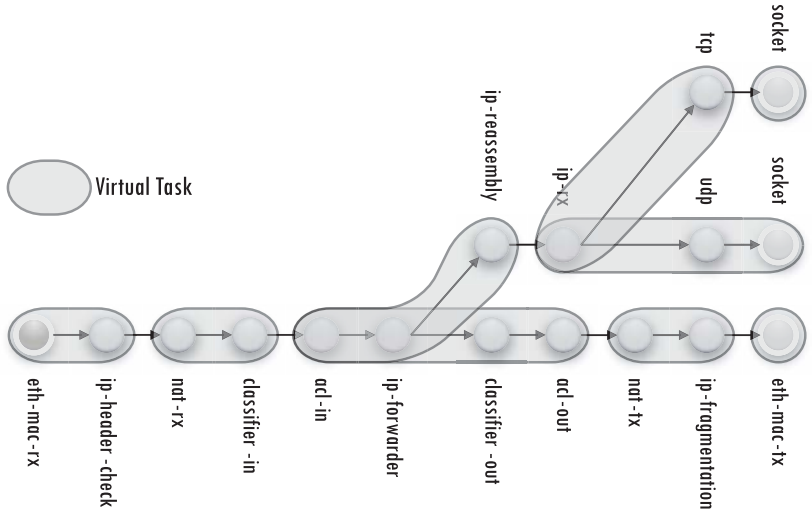

Figure 5-15: Virtual Tasks

5.4.2 Instrumentation

The RNOS provides various statistics for gathering data about the behavior of the system. As the gathering of statistic data incurs additional overhead, it can be removed or added at compile time. The following data is available:

- Worst-case, best-case and average processing time of each task.
- Worst-case, best-case and average system presence time of packets per microflow. The system presence time of a packet is from the time it enters the system until it leaves it again. This includes the processing time and the time the packet has been waiting for processing.

This built-in statistic gathering allows a verification against external measurement equipment. More about measurement is available in Chapter 6.

5.5 Schedulability Region of RNOS

It is obvious that RNOS is not free of overhead and that this overhead has an impact on the performance of the system. Before looking at measurements (see Chapter 6), we will do a formal overhead analysis of RNOS and find its schedulability region. What are the conditions such that RNOS makes sense? What is the influence of the overhead and other parameters on throughput and delay?

5.5.1 Overhead

Overhead consists of everything except the actual processing of packets. Applied to the model the overhead consists of everything except the execution of the tasks (or packet processors in RNOS). The overhead in an RNOS based system can be classified into two categories:

1. Overhead that occurs in any implementation of a packet processing system, i.e. that is not specific to RNOS.
2. Overhead that is specific to the way RNOS works.

Non-specific Overhead

Overhead incurred by the operating system due to process scheduling does not have to be considered, as RNOS requests a predefined minimum amount of processing resources (see section 5.2). On the other hand, overhead incurred by interrupt service routines has to be considered.

Interrupts are a common method for hardware components to signal an event to the software. In case of packet processing systems, interrupts are usually only used to wakeup an idle

system and not for each packet that arrives. However, there are still driver models that are completely interrupt based, i.e. that use interrupts for every packet [53, 54]. The interrupt overhead related with packet reception and, depending on the actual system, also with packet transmission (transmission complete interrupt) has to be accounted for. All other non packet processing related interrupts must be either disabled during the time the RNOS process is running, or, if this is not possible, we have to take into account that less processing time might be available for the RNOS process. We will use arrival curves to model interrupts (see section 5.6).

RNOS specific Overhead

RNOS specific overhead originates in its scheduler[9] and in the creation and termination of path-threads. Other components such as source-threads or learn-threads[10] are either always present, and as such do not impose a direct overhead, or are not present in a stationary state.

The overhead of the scheduler is dependent on the number of active flows and occurs for each preemption point, while the overhead for the creation and termination of path-threads is per packet. The overhead of the scheduler is dependent on the number of active flows because of the implementation of the priority queue for the EDF algorithm of the scheduler as a heap [55]. (5.1) shows the overhead of the scheduler per preemption point. The overhead is given as required processing

[9] The overhead of thread switches is included in the scheduler overhead.

[10] Learn-threads are only active for the first packet of a connection and only if there is not a predefined microflow for that connection.

time to process the overhead. Therefore, the unit of the overhead is a time unit (e.g. microseconds). Note that the overhead is independent of whether a thread switch occurs or not. The scheduler runs at each preemption point and a re-scheduling of the current thread takes the same time as scheduling a new thread[11]. Also note that as the priority queue for the EDF is implemented as a fixed-sized heap. The maximum number of possible flows is known[12]. The overhead incurred by the creation and termination of a path-thread is shown in (5.2).

$$o_{preemption-point} = o_{scheduler} + o_{prio-queue} \log_2 N, \quad \text{where } N \text{ is the maximum number of flows.} \tag{5.1}$$

$$o_{path-thread} = o_{path-thread-creation} + o_{path-thread-termination} \tag{5.2}$$

Similar to the lower and upper execution time of packets in Chapter 3, we will use lower and upper overhead in the following explanations, the lower overhead being the minimum overhead and the upper overhead being the maximum (worst-case) overhead.

With the knowledge about the overhead we can now look at its influence on throughput and delay of an RNOS based system.

[11] Valid for worst case only, cache effects could make a re-scheduling of the current thread faster.

[12] The maximum number of flows is also known due to the admission control, which must take place to admit a new flow (flows not admitted are aggregated in the best effort flow)

5.5.2 Throughput

The throughput of a system is a common benchmark value for packet processing systems. Here, we will look at the impact of the RNOS specific overhead on throughput. Note that throughput is defined (see Definition 23) without the notion of flows or priorities; all packets have to be processed completely, regardless of their priority. Buffering is not possible as the packet rate is constant and an infinite period would require infinite buffer space.

Definition 23

Throughput	*The maximum packet rate (number of packets per second) that can be processed over an infinite period of time without loosing or dropping any packet.*

Maximum throughput is achieved when packets are processed back-to-back, the system never being idle. (5.3) gives the general equation for the lower bound of the maximum throughput. It is defined by the upper bound for the packet processing time e^u_{packet} (which is equal to $e^u_{f_k}$, the upper bound for the packet processing time of a packet of flow f_k, see Chapter 3) and the upper bound for the associated overhead o^u_{packet}.

$$p^l_{max} = \frac{1}{e^u_{packet} + o^u_{packet}} \tag{5.3}$$

(5.4) shows the overhead for an RNOS based system. It consists of the overhead due to the RTOS (interrupt service routines and process scheduling) and the overhead specific to RNOS.

$$o^u_{packet} = o^u_{RTOS/packet} + o^u_{RNOS/packet} \tag{5.4}$$

In the previous section we have learned that the overhead of RNOS occurs on path-thread creation, termination and at each preemption point. The worst-case scenario for the maximum throughput is having a thread switch at each preemption point. (5.5) shows the total RNOS specific overhead per packet. Here we assume that non packet processing related interrupts are disabled during execution of the RNOS process and that the packet processing related interrupts take place in the RTOS overhead part. Note that the maximum number of preemption points can be influenced by using the virtual task mechanism of RNOS (compare previous subchapter about virtual tasks).

$$o_{RNOS/packet} = o_{path-thread} + n \cdot o_{preemption-point}$$
where n is the maximum number of preemption points. (5.5)

By defining the upper bound for the RNOS specific overhead as a factor $c_{RNOS-specific}$ of the RTOS overhead, the lower bound for the maximum throughput of RNOS can be specified as in (5.6).

$$p^l_{max} = \frac{1}{e^u_{packet} + (1 + c_{RNOS-specific}) o^u_{RTOS/packet}} \tag{5.6}$$

Using $o^u_{RTOS/packet} = c_{RTOS/packet} e^u_{packet}$ and by dividing the lower bounds of the maximum throughputs of RNOS and a system

without RNOS, we get the relative lower bound for the maximum throughput of RNOS, see (5.7).

$$p^{l}_{max\text{-}relative} = \frac{1 + c_{RTOS\,/\,packet}}{1 + (1 + c_{RNOS-specific})c_{RTOS\,/\,packet}} \qquad (5.7)$$

Figure 5-16 shows the impact of the additional overhead (specific overhead) of RNOS on throughput compared to a system without RNOS. The RTOS overhead is assumed to be 5% of the total packet processing time ($c_{RTOS\,/\,packet} = 0.05$). The factor $c_{RNOS-specific}$ for the RNOS specific overhead in the plot range is from 0 to 10, which means from zero to 10 times the overhead of a system without RNOS. It also means that the RNOS specific overhead is from 0% to 50% of the total packet processing time, which is shown in the X-axis of the figure.

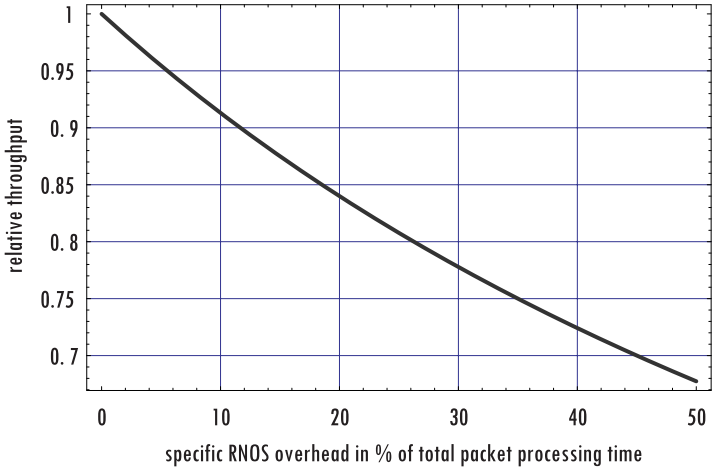

Figure 5-16: Impact of Overhead on Throughput

The figure shows that as long as the additional overhead incurred by RNOS is small compared to the total packet processing time, the impact on maximum throughput is small. Having an additional overhead of 10% of the total packet processing time results in a lower bound of the maximum throughput of 92% of what a system without RNOS would achieve.

5.5.3 Delay

After having seen the influence of overhead on throughput, we will now look at its influence on the delay of a packet. First of all, we will give some definitions used throughout the remainder of this chapter. Looking at a packet lifetime inside a system, we distinguish between different phases:

- Waiting Time
- Overhead Time
- Processing Time

See Definition 24 to Definition 26.

Definition 24

Waiting Time	*The waiting time is the total time the packet resides inside the system and is idle due to higher priority packets, i.e. processing is applied to other packet. The waiting time only includes the time the packet is waiting while other packets are being processed (incl. the overhead associated with those packets) and not any operating system overhead associated with the packet itself.*

Definition 25

Overhead Time	*The overhead time is the total time the packet resides inside the system and the operating system is executing overhead functions associated with this packet.*

Definition 26

Processing Time	*The processing time is the total time the packet resides inside the system and is being processed, i.e. processing is applied to the packet. It does not include any operating system overhead.*

Definition 27

Delay	*The delay of a packet is the time the packet resides inside the system, e.g. from the time it enters the system (or is created) until the time it leaves the system (or is consumed/deleted).*

The total delay of a packet comprises waiting time, overhead time and processing time (see Definition 27). The overhead associated with the termination of a path-thread is not part of the overhead time, as the packet has already left the system (or has been consumed) at that time. Figure 5-17 depicts an example for the alignment of phases in a packet lifetime.

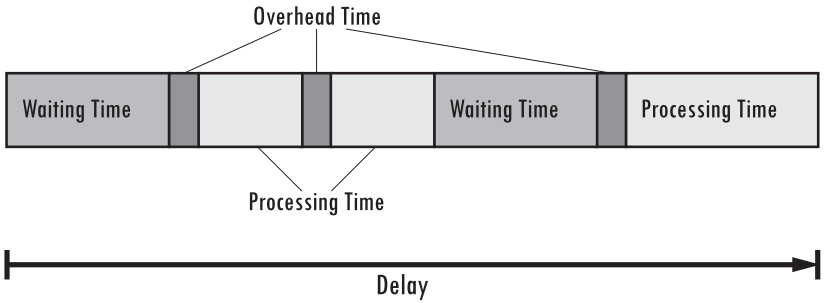

Figure 5-17: Example alignment of phases

We will now analyze the different parts of the delay by using a worst-case scenario. The aim is to analytically determine the worst-case delay of a highest-priority packet. We assume that non packet processing related interrupts are disabled during the runtime of the RNOS process and that RNOS gets a fixed amount of minimum processing time as described previously. In the following paragraphs we will discuss only the time inside the RNOS process, we mask out the time running outside RNOS. At the end of this section, we will then calculate the actual delay by also taking into account the time that elapsed outside the RNOS process.

Waiting Time

Let us consider a highest-priority packet. There is no other packet in the system with higher priority. However, the source thread with its reception and filter task (compare previous subchapter) takes always precedence. Without that, we would risk loosing packets due to input queue overflows.

The worst-case waiting time can be elicited as follows:

0. The highest-priority packet arrives at the system. Waiting time measurement starts.

1. The longest task in the system has started processing just before the arrival of the highest-priority packet. While this task is being processed, other packets arrive at the system. They arrive with the maximum packet rate (defined by the physical line rate and the smallest packet size).
2. Once this task has finished execution, the source thread starts running. It will receive and filter all packets that have in the meantime been received. While the source thread is running, additional packets arrive with the maximum packet rate. These packets also have to be received and filtered.
3. As the packet can be processed now, we stop the measurement of the waiting time. However, every time that there is a preemption point, the source thread will run again and process the packets that have arrived at the system. During all the time, packets arrive with maximum packet rate. These periods of source thread processing are added to the waiting time.

Figure 5-18 depicts the different phases in the lifetime of the packet. The numbers in the figure relate to the above explanations.

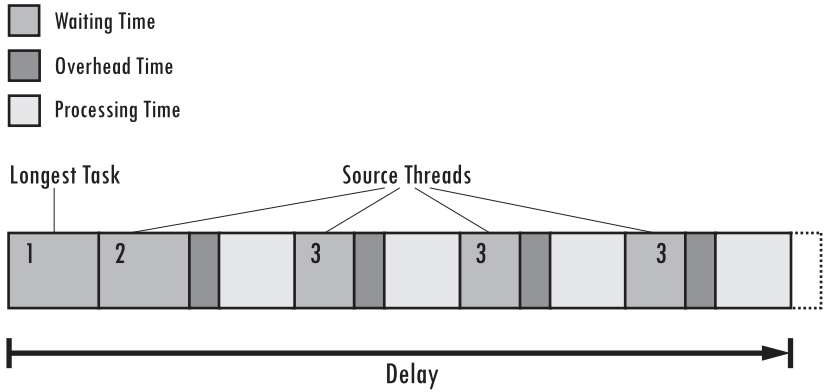

Figure 5-18: Worst-case waiting time for a high-priority packet

(5.8) gives the equation for the worst-case waiting time of a highest-priority packet. The equation is grouped into three parts which relate to the above explanations and Figure 5-18.

$$w^u_{packet} = e^u_{longest-task} + o^u_{preemption-point} + \left(e^u_{longest-task} + o^u_{preemption-point}\right)\left(\frac{r_{line}e^u_{rx\&filter}}{1-r_{line}e^u_{rx\&filter}}\right) + (n-1)o^u_{preemption-point} + \left((2n-1)\cdot o^u_{preemption-point} + e^u_{packet} - e^u_{last-task}\right)\left(\frac{r_{line}e^u_{rx\&filter}}{1-r_{line}e^u_{rx\&filter}}\right)$$

(5.8)

The first part of the equation consists of the upper execution time for the longest task $e^u_{longest-task}$ in the system and the overhead of RNOS due to the immediate thread switch to the source thread $o^u_{preemption-point}$.

The term $\left(e^u_{longest-task} + o^u_{preemption-point}\right)\left(\frac{r_{line} e^u_{rx\&\,filter}}{1 - r_{line} e^u_{rx\&\,filter}}\right)$ in the second part of the equation is the result of the source thread. The packets that have arrived during the execution of the longest task and the following thread switch have to be processed by the source thread. The packets arrived with a packet rate of r_{line}, requiring an upper execution time $e^u_{rx\&\,filter}$ per packet, resulting in $\left(e^u_{longest-task}\right) r_{max-packet} e^u_{rx\&\,filter}$. While the source thread runs, new packets arrive at the system. Also these packets have to be processed. And while these packets are processed, new packets arrive and so on. We get

$$\left(e^u_{longest-task}\right) r_{max-packet} e^u_{rx\&\,filter} + \left(e^u_{longest-task} r_{max-packet} e^u_{rx\&\,filter}\right) r_{max-packet} e^u_{rx\&\,filter} + \ldots$$

which is a geometric series and results in the said term.

In the following we will use $s = \frac{r_{line} e^u_{rx\&\,filter}}{1 - r_{line} e^u_{rx\&\,filter}}$.

Also note that $r_{line} e^u_{rx\&\,filter}$ can be interpreted as the fraction of processing time that is required to receive and filter packets from system interfaces and that $r_{line} e^u_{rx\&\,filter} < 1$ is required for a valid system (otherwise the system is over-occupied with receiving and filtering packets and will not do anything useful). Also note that $e^u_{rx\&\,filter}$ includes the creation of path-threads (see Section 5.1.4).

The third part of the equation is similar to the second part. It covers the running of the source thread between each preemption point. The source thread runs as long as the input queues are not empty. Only then, the processing of the highest priority packet is resumed. In the equation, *n* denotes the maximum number of preemption points for the highest-priority packet.

The term $(n-1) o^u_{preemption-point}$ represents the thread switches to the source thread. The term $\left((2n-1) \cdot o^u_{preemption-point} + e^u_{packet} - e^u_{last-task}\right)$ is the time during which new packets have arrived and the source thread was not running. There are 2*n*-1 thread switches, from the highest-priority packet processing to the source thread and back. The upper execution time of the last task of the highest-priority packet has to be subtracted from the total upper execution time of the packet as the packets that arrive during the execution of the last task do not add to the waiting time of the highest-priority packet.

(5.9) gives a simplified form of (5.8) that assumes that the application of the highest-priority packet has n preemption points and that all the tasks in the system have an identical processing time.

$$w^u_{packet} = \frac{e^u_{packet}}{n} + o^u_{preemption-point} + \left(\frac{e^u_{packet}}{n} + o^u_{preemption-point}\right) s + (n-1) o^u_{preemption-point} + \left((2n-1) o^u_{preemption-point} + \frac{e^u_{packet}(n-1)}{n}\right) s \quad (5.9)$$

(5.10) is identical to (5.9), but the terms are reordered such that the influence of the number of tasks *n* is easier to understand. With a growing *n* the first part of the equation will result in a smaller value, while the second part of the equation will result in a larger value. We can interpret this as follows: The more tasks (or preemption points) there are, the less the influence of

the "longest" task will be, as it becomes "shorter". On the other hand, more tasks means more thread switches (as the source thread has always higher priority), which results in more overhead.

$$w^u_{packet} = \left(s + \frac{1}{n}\right) e^u_{packet} + n(1+2s)\, o^u_{preemption-point} \quad (5.10)$$

Figure 5-19 shows the upper bound for the waiting time of a highest priority packet for a different number of tasks and packet processing times. The figures are plotted with $r_{line} e^u_{rx\&\ filter}$=0.1, which is a reasonable load for the reception and filter tasks. The upper bound for the packet processing times e^u_{packet} are 100µs, 300µs, 500µs, 700µs and 900µs and the upper bound of the preemption point overhead $o^u_{preemption-point}$ is 6µs.

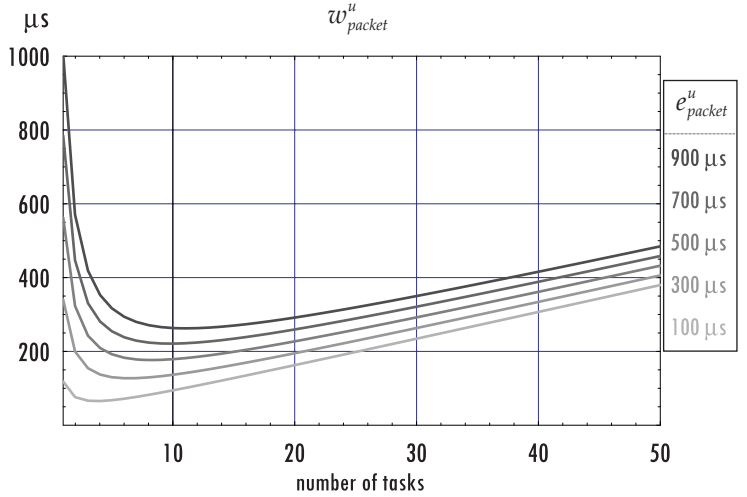

Figure 5-19: Upper bounds for waiting times for a highest priority packet

Overhead Time

The overhead time in the worst-case scenario is simple to determine. There are as many thread switches as there are preemption points. Figure 5-20 depicts the worst-case overhead times for a highest-priority packet with four preemption points in the path-thread. Note that the overhead (e.g. thread switches) associated with other threads are part of the waiting time. Overhead time does only include the overhead for one specific packet (see Definition 25). Again, the overhead associated with the termination of a path-thread is not part of the overhead time, as the packet already has left the system (or has been consumed) at that time and the overhead associated with the creation of a path-thread is included in the waiting time (in $e^u_{rx\&\ filter}$).

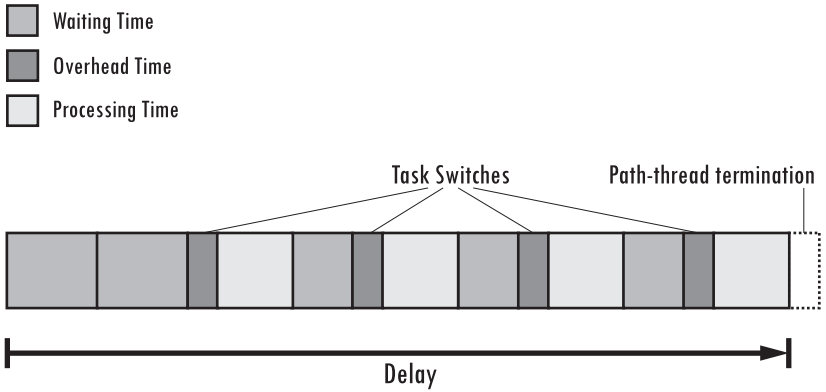

Figure 5-20: Worst-case overhead time for a highest-priority packet

(5.11) shows the upper overhead time for a highest-priority packet.

$$o^u_{packet} = no^u_{preemption-point},$$ (5.11)

where n is the maximum number of preemption points.

Processing Time

The worst-case processing time for a highest-priority packet is the upper processing time e^u_{packet} (which is equal to $e^u_{f_k}$, the upper bound for the packet processing time of a packet of flow f_k, compare also Chapter 3).

Delay

As we know from the definition of delay (see Definition 27), the upper bound for the delay d^u_{packet} is the sum of the processing time, overhead time and waiting time ($d^u_{packet} = e^u_{packet} + o^u_{packet} + w^u_{packet}$). (5.12) shows the resulting equation.

$$d^u_{packet} = e^u_{packet} + no^u_{preemption-point} + \left(s + \frac{1}{n}\right)e^u_{packet} + n(1+2s)o^u_{preemption-point}$$ (5.12)

Reordering the equation for the influence of the number of preemption points (or (virtual) tasks) n yields (5.13).

$$d^u_{packet} = (1+s)e^u_{packet} + o^u_{preemption-point} + \frac{1}{n}e^u_{packet} + 2n(1+s)o^u_{preemption-point}$$ (5.13)

Before discussing the delay we have to include the time outside the RNOS process. (5.13) does not give the actual delay, as it does consider the time inside the RNOS process only. To get the actual delay d'^u_{packet} we have to

- add the worst-case initial waiting time before the RNOS process gets scheduled by the RTOS and
- consider also all the packets that have arrived during that initial waiting time, i.e. also those packets that have to be received and filtered by the source thread.

Here we assume that the RNOS process will run at least as long as the time needed to process the highest-priority packet completely. This is always the case when packet processing is

an important part of the complete application.[13] According to the model, the RNOS process receives a minimum amount of processing time in a defined period of time (compare previous subchapter). (5.14) shows the actual delay d'^{u}_{packet}, with $w^{u}_{RNOS\text{-}Process}$ as the upper waiting time before the RNOS process gets access to the processing resource. By Figure 5-13 $w^{u}_{RNOS\text{-}Process}$ can be defined as $w^{u}_{RNOS-process} = t_2 - t_1$.

$$d'^{u}_{packet} = d^{u}_{packet} + (1+s)w^{u}_{RNOS\text{-}Process} \tag{5.14}$$

We recognize that the delay d'^{u}_{packet} consists of a constant term, a term that shrinks with larger n (the longest-task in the system) and a term that grows with larger n. The effect of the number of preemption points n can easily been seen in Figure 5-21, which shows the upper bound of delay for highest priority packets plotted versus the number of tasks or preemption points n. The upper bound for the waiting time before the RNOS process gets scheduled $w^{u}_{RNOS\text{-}Process}$ is assumed to be 2000µs and other values are identical to those used for Figure 5-19.

[13] Otherwise, the packet throughput of the system is very low.

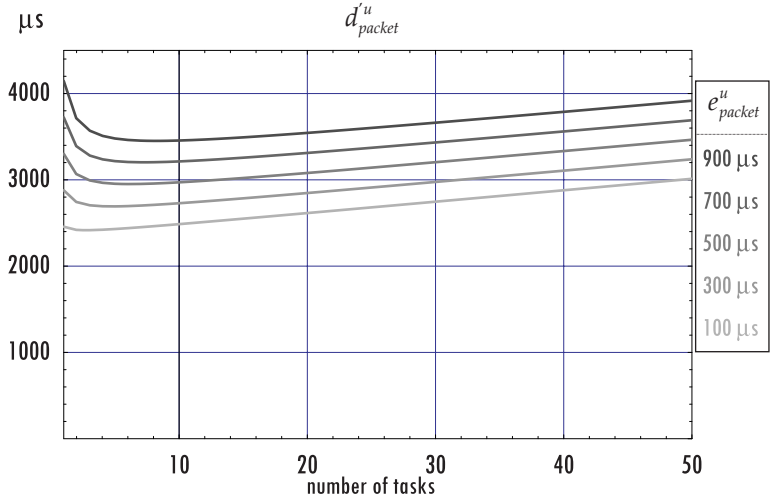

Figure 5-21: Upper bound of delay for highest priority packet

For each upper packet processing time, there is a number of tasks $n_{delay\text{-}min}$ for which the delay is minimized. If $n > n_{delay\text{-}min}$, the delay increases and the overall packet throughput decreases (as more thread switches incur more overhead and therefore the processing of each packet requires more system resources). If $n < n_{delay\text{-}min}$ the delay also increases, but the overall system performance is higher. Therefore, n should be chosen with $n \le n_{delay\text{-}min}$. To get $n_{delay\text{-}min}$ we derivate d^{u}_{packet} with subject n (5.15) and find the solutions for the derivation being equal to zero (5.16).

$$\frac{\delta d^{u}_{packet}}{\delta n} = -\frac{1}{n^2} e^{u}_{packet} + 2(1+s) o^{u}_{preemption-point} \tag{5.15}$$

$$-\frac{1}{n^2} e^{u}_{packet} + 2(1+s) o^{u}_{preemption-point} = 0 \tag{5.16}$$

(5.17) gives the solution for (5.16), which is the number of tasks or preemption points $n_{delay\text{-}min}$ that result in a minimum worst-case delay for a highest priority packet. This is also the maximum number of preemption points that are feasible. Of course the value must be a cardinal, therefore the actual n is either $\lfloor n_{delay\text{-}min} \rfloor$ or $\lceil n_{delay\text{-}min} \rceil$, depending on which results in a lower delay value.

$$n_{delay\text{-}min} = \sqrt{\frac{e^u_{packet}\left(1 - r_{line} e^u_{rx\&\,filter}\right)}{2o^u_{preemption-point}}} \qquad (5.17)$$

Figure 5-22 is a plot of (5.17) versus the total packet processing time e^u_{packet}, other values being identical to those used in Figure 5-19.

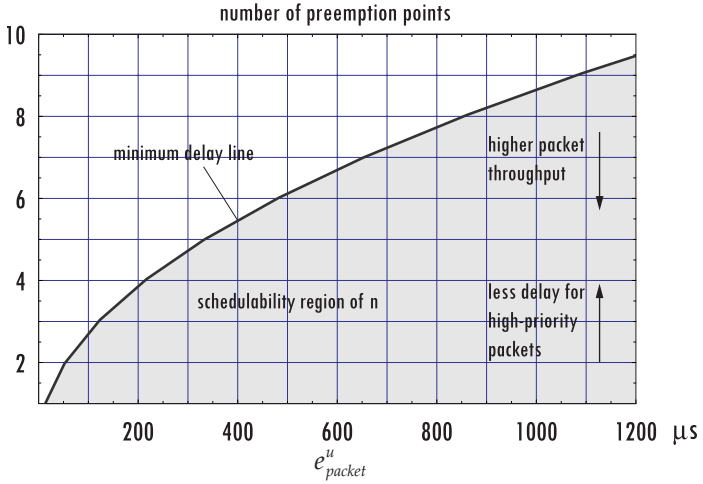

Figure 5-22: Feasible number of preemption points, depending on e^u_{packet}

The above figure gives the schedulability region for n. Inside this region the following applies:

- The less preemption points the higher the packet throughput and the larger the delay for a highest-priority packet
- The more preemption points the smaller the delay for a highest-priority packet and the less the packet throughput.

Remember that the above discussions are valid under the assumption that the preemption points are equidistant. In general, the optimal number of preemption points depends on more than one criterion (here: worst-case delay for highest priority flow). We call the analysis method we have used in this section "runtime analysis", as it models the exact behavior for a given scenario. In Chapter 3 we have introduced an analysis method for our model based on network calculus. In Section 5.6 we show how network calculus can be applied to RNOS based systems.

5.5.4 Conclusion

The schedulability region of RNOS is bounded by various system parameters. First of all, the processing time required to receive and filter incoming packets multiplied by the maximum packet rate of all incoming interfaces must be less than one, preferably as small as possible. Otherwise, all or most of the processing capacity is consumed for receiving and filtering packets and not enough capacity is left to process packets. Second, there is a tradeoff between maximum throughput and the worst-case delay that is tolerable for high priority packets. The parameters of this tradeoff are the maximum number of preemption points and the task with the longest execution

time. The execution time of the longest task directly adds to the worst-case delay. Third, the overhead incurred by RNOS should be small compared to the total packet processing time. Otherwise, the maximum throughput becomes small (too small). An RNOS specific overhead of 10% of the total packet processing time results in a 92% throughput compared to a system without RNOS.

5.6 Analysis with Network Calculus

In Chapter 3 we have introduced how to do system analysis with network calculus for systems based on our model. In this section, we show how to build system models that are to be implemented based on RNOS and how to analyze such systems with network calculus.

Using RNOS as a base for the implementation requires that we include the following properties in the system model and calculus:

- Source thread with its receive and classification task
- Overhead

The source thread can be modeled by a receive and classification flow which receives packets at all input interfaces at maximum rate (defined by physical line parameters and minimum packet size) (see also Figure 5-9). Figure 5-23 shows the calculation scheme for RNOS. The source flow has higher priority than all other flows. Packet flows have lower priority. The figure also shows an ISR (Interrupt Service Routine) flow, which can be used to model the interrupts.

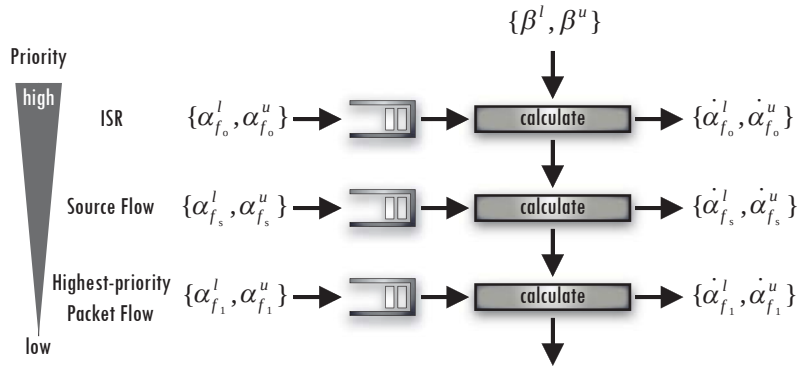

Figure 5-23: Calculation scheme for RNOS

Overhead occurs at creation and termination of path-threads and at each preemption point (see section 5.5). We add the overhead to the execution times:

- The overhead for the creation of a path-thread is already included in $e^u_{rx\&filter}$. The overhead for the termination of a path-thread is modeled by adding an additional task after the sink task, which represents the termination of the path-thread. Note that if we do a delay analysis, we have to omit the termination overhead for the flow we are looking at[14].
- The overhead at the preemption points is modeled by adding two times the preemption-point overhead for each (virtual) task for each path except the one of the source thread.

Consider the following: All input interfaces receive packets at maximum rate and the source thread has higher priority than any other thread. Therefore we have thread-switches at each

[14] Path-thread termination occurs after the packet has left the system or has been consumed.

preemption point. Figure 5-24 shows an example for such a scheduling pattern with two path-threads and a source thread.

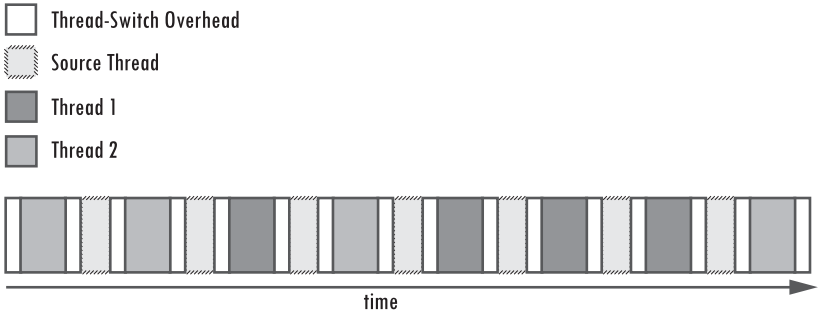

Figure 5-24: Thread-switch pattern

With the above guidelines on how to do the analysis for a RNOS based system, we have continuous design flow from the model to the analysis and to the implementation. In the following section we show how exactly to do the calculations. We will repeat the worst-case delay analysis for a highest-priority packet in RNOS. This time we use the calculus of our model instead of the runtime analysis method of Section 5.5.

5.6.1 Example Analysis: Worst-Case Packet Delay

Step 1 – Define Flows

We have two flows to model, the source flow f_s and the highest priority packet flow f_{hp}.

In the previous runtime analysis we have used $r_{line}e^u_{rx\&\,filter}$=0.1 for the arrival of packets. Therefore, the arrival curves $\alpha^l_{f_s}$, $\alpha^u_{f_s}$ of the source flow look like shown in Figure 5-25.

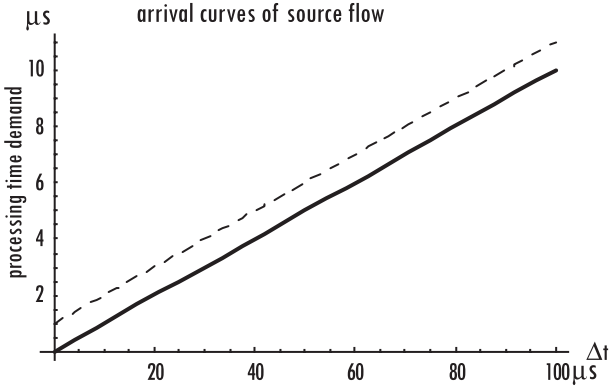

Figure 5-25: Arrival curves of source flow

The upper execution time[15] $e^u_{f_{hp}}$ of the highest-priority packet flow f_{hp} has to be modified as explained in the previous section. (5.18) shows the modified upper execution time $e'^u_{f_{hp}}$, where n is the number of preemption points.

$$e'^u_{f_{hp}} = e^u_{f_{hp}} + (2n-1)o^u_{preemption-point} \qquad (5.18)$$

Note that we do not add the overheads of the last preemption point and the path-thread termination, as in case of the delay analysis the packet is already sent out of the system or has been consumed at this point.

For $e^u_{f_{hp}}$ we use the same values as in the previous analysis, 100µs, 300µs, 500µs, 700µs and 900µs. The arrival curves $\bar{\alpha}^l_{f_{hp}}$, $\bar{\alpha}^u_{f_{hp}}$ could look like shown in Figure 5-26 (with $e^u_{f_{hp}}$ = 300µs). The actual slope does not matter (here) for the delay analysis.

[15] For the worst-case delay analysis we only need the upper execution time.

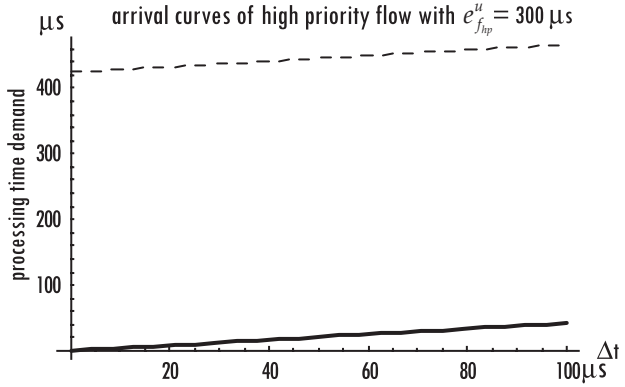

Figure 5-26: Example arrival curves of high-priority flow

Step 2 – Define Resource

The RNOS process periodically receives at least 8ms of processing time within a period of 10ms. Therefore, the service curves β^l, β^u can be defined as shown in Figure 5-27.

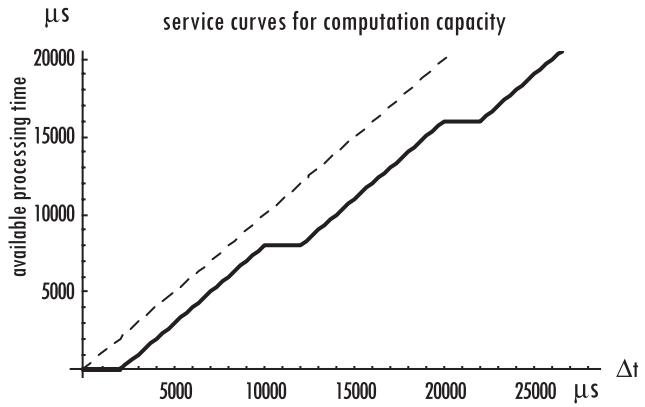

Figure 5-27: Service curves of processing resource

Step 3 – Execute Calculations

Using the operators as defined in Chapter 3, we can now execute the analysis by applying the calculation scheme for RNOS as shown in Figure 5-23. (5.19), (5.20) and (5.21) are used to calculate the upper bound for the delay. Note, that we have to add the execution time of the longest task to the calculated delay as defined in Chapter 3. In RNOS, we also have to add the overhead of one additional preemption point (see (5.21)). Both are necessary to model the initial waiting time. Again, n denotes the number of preemption points.

$$\beta_s^l(\Delta) = \left(\beta^l(\Delta) - \alpha_{f_s}^u(\Delta)\right) \bar{\oplus}\, 0 \tag{5.19}$$

$$d_{hp}^u = \sup\left\{\inf_{\Delta \geq 0}\left\{\tau : \tau \geq 0 \wedge \alpha_{f_{hp}}^u(\Delta) \leq \beta_s^l(\Delta - \tau)\right\}\right\} \tag{5.20}$$

$$d_{hp}'^u = d_{hp}^u + \frac{e_{hp}^u}{n} + o_{preemption-point}^u \tag{5.21}$$

Figure 5-28 shows the resulting worst-case delay for packets of the highest-priority flow f_{hp}. The results are identical to those we got by using the runtime analysis method (compare Figure 5-21). The results differ by 1.556µs, which is less than half of a thousandth part of the delay and therefore is negligible.

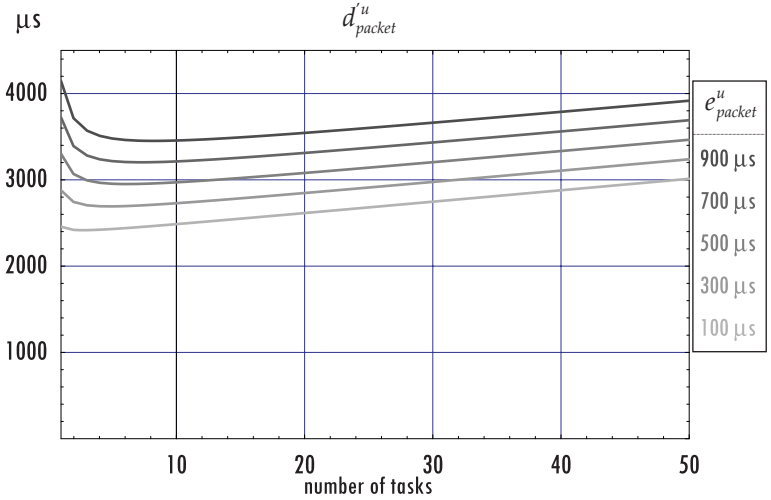

Figure 5-28: Upper bound for delay for packets of highest priority flow

We have presented two analysis methods for our system. While the method based on network calculus works on the model introduced in Chapter 2, the runtime analysis method requires a thorough understanding of the mechanisms of RNOS. While the network calculus only enables us to calculate upper and lower bounds, the runtime analysis allows us to understand what happens in the system on a finer grained base. This allows us to also calculate average (statistical) values. The biggest advantage of the analysis method based on network calculus is its much easier application, especially if we have many flows. However, both methods yield the same results, at least for the scenarios we have calculated.

5.7 Summary

In this chapter we have presented the software platform RNOS. It provides all the functionality and elements to implement packet processing applications defined by our model. This allows for a seamless process from design to the implementation.

RNOS is at the same time efficient and easy to use. Complex applications can be built through the use of hierarchical structures and efficiency is provided by separating the packet processing part from the application structure part. The specialized packet processing elements have a flat view of the system, bypassing any structural elements. To protect the system from misbehaving packet sources, RNOS uses an element called microflow to observe packets and acertain that they follow their predefined path.

As packet processing is only a (possibly small) part of the complete application in our target domain, RNOS has to be integrated on standard real-time operating systems. The integration is such that RNOS runs in a process of the underlying RTOS. The RTOS has to provide a minimum guaranteed processing time to RNOS. As these are all the requirements to the RTOS, RNOS can be ported to almost any available RTOS (commercial, open source or proprietary).

RNOS can be looked at as a higher level programming system. Based on tasks; the programmer puts together applications by connecting tasks to task graphs. Flows are used to define the input to these task graphs and the flow's quality of service parameter delay defines how packets of that flow will be treated inside the system.

Obviously any software platform adds some overhead to execution. RNOS, also, is not free of overhead and the schedulability analysis has shown the influences of the overhead on delay and throughput. As the overhead of RNOS is dependent on the number of preemption points, RNOS has a mechanism that allows to determine the distance of the preemption points (with the granularity of the underlying task sizes). This allows optimizing a system for delay or throughput (the two optimization targets are contrary), while leaving the tasks themselves untouched.

Two analysis methods are available to explore the properties of a RNOS based systems. Both yield the same results.

In conclusion, RNOS provides the means to have a seamless engineering path from model to implementation.

6 Example: Implementation & Measurements

In this chapter we present as an example the implementation of an application based on RNOS on a specific system. The first part of this chapter describes the example system and specific implementation issues. We then describe the application, and measure and calculate the attributes of the system (overhead, execution time of individual tasks, minimum delay).

In the second part of this chapter we compare for specific scenarios theoretical calculation results with practical measurement results.

6.1 System Description

The example consists of the hardware platform, the real-time operating system, the implementation of RNOS itself and the example application on top of RNOS.

6.1.1 Hardware Platform

For the example, a hardware architecture was chosen that is widely used in the target domains of industrial automation, life science and small communication devices. It is based around a PowerPC communication controller from Freescale (formerly Motorola) that runs with 50 MHz [56]. The architecture of the communication controller consists of the core CPU and various support units that offload the CPU. Independent DMA controllers allow transporting data to and from memory and to and from the physical interfaces without interaction of the core CPU. The core CPU, a PowerPC derivate, has 4 Kbyte instruction cache and 2 Kbyte data cache. The caches are small, but together with external SDRAM, burst-access to/from memory is possible. The external bus runs with 50 MHz and the size of the SDRAM is 16 MByte. A specialized microcontroller executes the low-layer protocol for serial lines, Ethernet and USB (other embedded controllers have these functions hardwired). Although the specialized microcontroller is not programmable by the user, the manufacturer (Freescale) is able to modify/upgrade the functionality by supplying microcode patches. Figure 6-1 show the block diagram of the embedded communication controller.

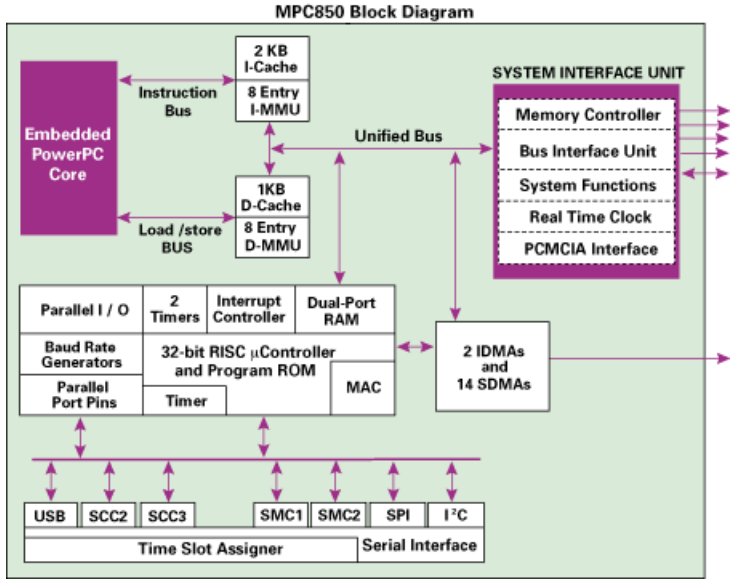

Figure 6-1: Block Diagram of Embedded Controller

The advantage of having the DMA controllers is that for reception and transmission of packets the influence of the packet size to the load of the CPU can be neglected, i.e. this results in having the same packet throughput per second independent of the packet size[16].

6.1.2 Real-Time Operating System

As underlying "real-time" operating system we use VxWorks from WindRiver [53]. VxWorks provides a preemptive process scheduling with priorities. Processes of the same priority will do a round-robin with a predefined time-slice. The request of RNOS for receiving a predefined minimum amount of processing cycles in a defined time interval could not be satisfied

[16] As long as the system is not line rate limited, compare Chapter 1.

directly. Therefore, we had to modify the RTOS such that it satisfies the requirements of RNOS:

The basic idea is to let RNOS run with highest priority until the minimum amount of processing cycles is consumed. Then, RNOS waits on a semaphore to be signaled. The semaphore will be signaled by the idle process or by the end of the time interval, whichever is first. Figure 6-2 depicts the basic idea.

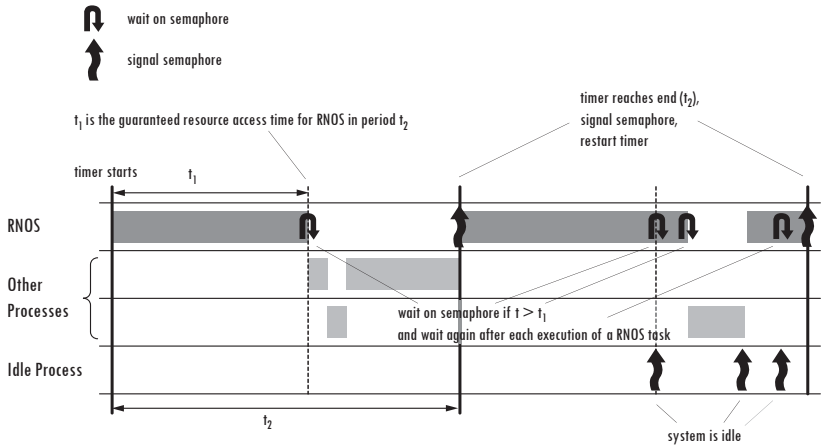

Figure 6-2: RNOS on VxWorks

The minimum processing time is measured with a timer. As a timer, a special hardware decrementer is used. The decrementer is a register that counts back with system frequency. For each time interval, the decrementer is reset to the start value (length of time interval). Between the executions of tasks, RNOS reads the current value of the decrementer and compares it to a predefined value. The predefined value is cycles of the time interval minus minimum cycles for RNOS. As soon as the decrementer is less than this value, RNOS waits on the semaphore. Once this happens, the RNOS process waits for the semaphore to be signaled and the RTOS will schedule

other processes. As soon as the system is idle, the idle process will run and signal the semaphore. As a consequence, the RNOS process wakes up (the semaphore has been signaled), goes into ready state and is scheduled by the RTOS. The semaphore will also be signaled whenever the decrementer reaches zero and the time interval restarts.

Beside the kernel functionality of the RTOS (processes, semaphores, memory), no services are used.

6.1.3 Service Curve

As we have now modified the RTOS such that it can satisfy the demand of RNOS, it is simple to provide the service curve. Remember that the service curve provides the lower and upper bound of the computation capacity of the system for the execution of RNOS tasks. The time intervals t_1 and t_2 define the lower and upper bound of the computation capacity for RNOS (see Chapter 5). Figure 6-1 depicts the lower and upper service curve for RNOS on the target system. We use 7ms for t_1 and 10ms for t_2.

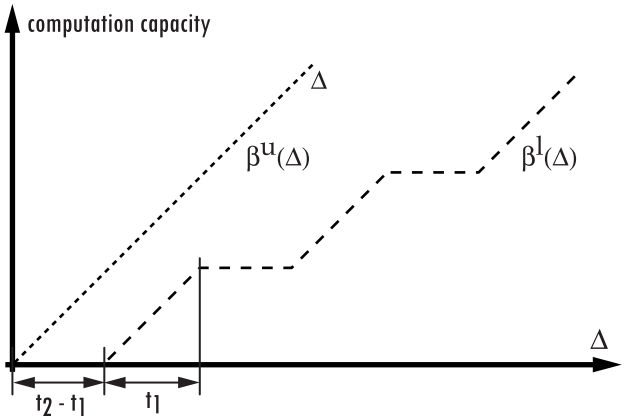

Figure 6-3: Lower and upper service curve for RNOS on target system

6.1.4 Application

We have implemented a standard IP router application. This allows us to investigate the behavior of the system by injecting traffic to the router and recording the output traffic. The recorded traffic can be analyzed offline to get results about the behavior of the system. The main task path is the IP forwarding functionality. It consists of 11 tasks (without the filter and receive task) as shown in Table 6-1. We have determined the worst-case execution time of these tasks by using the internal instrumentation of RNOS. For the measurement we turned off all caches, which resembles a worst-case situation. The worst-case execution time of the complete IP forwarding path is 314 µs.

Table 6-1: Tasks in the forwarding path

Task	Short Description	Worst-case Execution Time [µs]
Eth-mac-rx	Process Ethernet header	18
Ip-hdr-chk	Validate IP header	48
Rtp-interceptor	Intercepts incoming RTP packets for processing (special path)	15
Acl-in	Input access control list	17
Ip-forwarder	IP forwarder	38
Acl-out	Output access control list	11
Ipsec-interceptor	Intercepts outgoing IP packets for IPSec processing (special path)	13
Ip-fragm	IP packet fragmentation	9
Ip-hdr-compl	Completes IP header	14
Eth-mac-ip-tx	Adds Ethernet header	52
Driver-tx	Transmits packet	79
e_f^u	*Total forwarding time*	*314*

We have implemented additional tasks for the RNOS software platform, which are not used here. Table 6-2 gives an overview of the available protocols/functionality. Be aware that the implementation of these protocols consists of multiple tasks.

Table 6-2: Additional functionality that is available as tasks for RNOS

Protocol	Short Description
RTP	Realtime Transport Protocol
IPSec	IP Security, tunnel and transport mode
PPTP	Point to Point Tunneling Protocol
PPP	Point to Point Protocol
PPPoE	PPP over Ethernet
LinkScheduler	Link scheduling with WFQ (SCFQ) and Priority
Dejitter Buffer	Dejittering of RTP packets
IP	Complete IP stack, incl. Options etc.

6.1.5 RNOS Attributes

To use our analysis method and explore system properties under different scenarios, we have to determine the attributes of RNOS as described in the previous chapter. The required attributes are the upper execution time of the receive and filter task $e^u_{rx\&filter}$ and the overhead incurred by a preemption point $o^u_{preemption-point}$. With those we can apply our analysis method to the model.

Determine Preemption-Point Overhead

Through the use of the virtual task mechanism of RNOS (see Chapter 5), the preemption point overhead can easily be determined:

1) Measure the delay d_n of a packet on a system configured for n preemption points.
2) Measure the delay d_m of a packet on a system configured for m preemption points.
3) With $d=|d_n-d_m|$ we get the preemption point overhead for $|n-m|$ preemption points. Therefore, the preemption point overhead $o^u_{preemption-point}=d/|n-m|$.

We measure the delay d_n with a special measurement equipment from Spirent [57]. Figure 6-4 shows the measurement setup: The measurement equipment sends packets; they enter our system and are forwarded; the measurement equipment receives those packets and determines the delay for each packet. We execute all measurements with caches turned-off, which resembles a worst-case situation (cache misses).

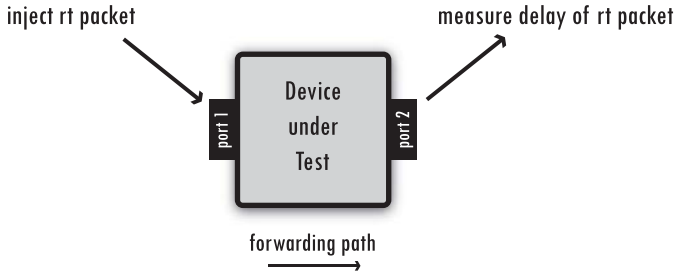

Figure 6-4: Measurement setup

For the determination of the preemption point overhead, we inject single packets on the idle[17] RNOS process and measure the delay. By doing this continually (and for different number of preemption points), we get results as shown in Figure 6-5.

[17] Idle here means that the RNOS process is idle; the other processes do not surrender any computation time to the RNOS process, they are NOT idle.

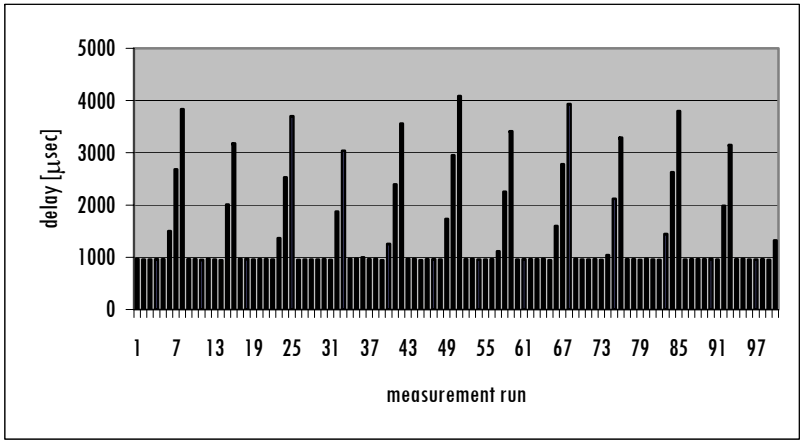

Figure 6-5: Delay measurement result (11 preemption points)

The result can be explained as follows. We know that RNOS gets access to the computation resource with a regular periodicity (see Section 6.1.3). Depending on where the measurement run hits the period, RNOS has no access to the computation resource and has to wait. Therefore the delay varies. Statistically we should hit every region inside the computation access period. Therefore, by sorting the measurement runs for ascending delay, we get the distribution of delay inside the period (see Figure 6-6). It correlates perfectly with RNOS' service curve (see Section 6.1.3). The first 70% of graph (Figure 6-6) shows a constant delay while after that the delay increases linearly. We can map this to the delay inside the 10 ms computation access period as shown in Figure 6-7. The first 7 ms the delay of a packet in a system in which the RNOS process is idle is constant; RNOS is ready to process the packet immediately. After that the packets have to wait for RNOS to get access to the computation resource again.

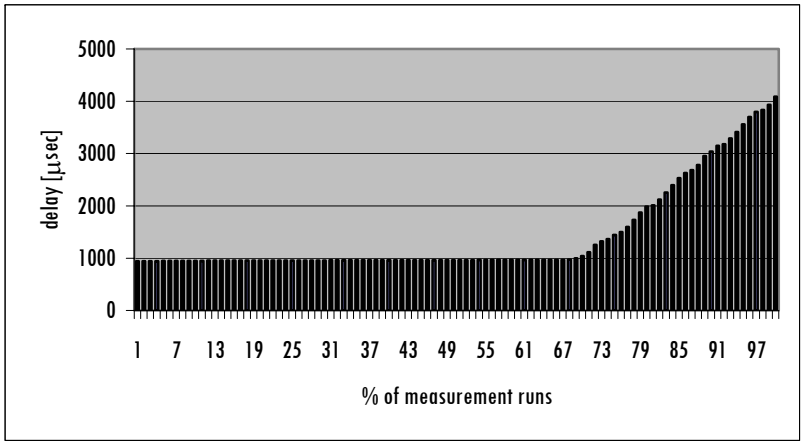

Figure 6-6: Sorted measurement results (ascending delay)

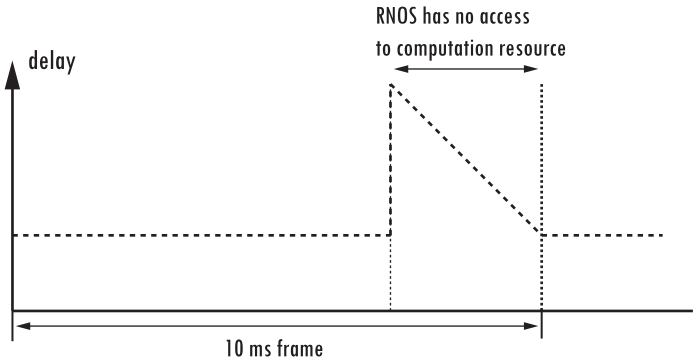

Figure 6-7: Delay of single packet

To determine the preemption point overhead, we will use the minimum delay that we receive for 70% of the delay measurement runs. Figure 6-8 shows measurement results for minimum delay. Note that the individual results vary by less than +/-1%. For the determination of the preemption point overhead, we use the average of those results.

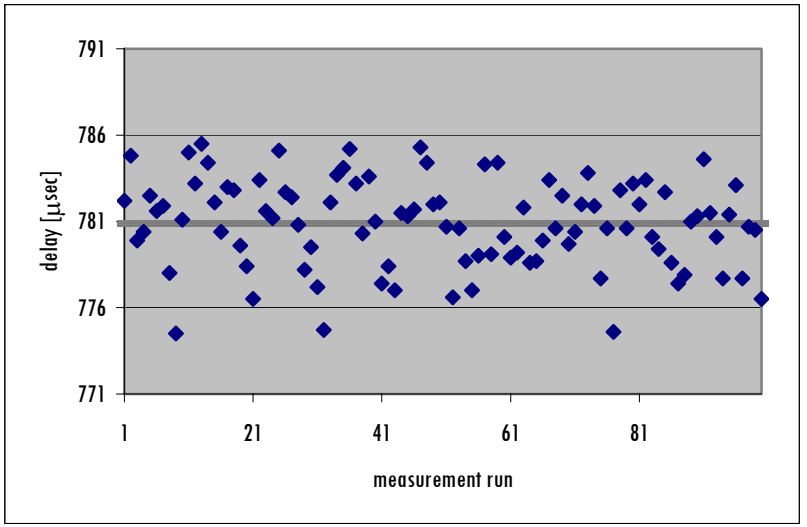

Figure 6-8: Minimum delay measurement (2 preemption points)

By determining the minimum delay for each number of preemption points we can calculate the preemption point overhead $o^u_{preemption-point}$. Figure 6-9 shows the minimum delay versus the number of preemption points. We can also determine the preemption point overhead from this figure; it is the (average) slope. The upper preemption point overhead $o^u_{preemption-point}$ is 19µs.

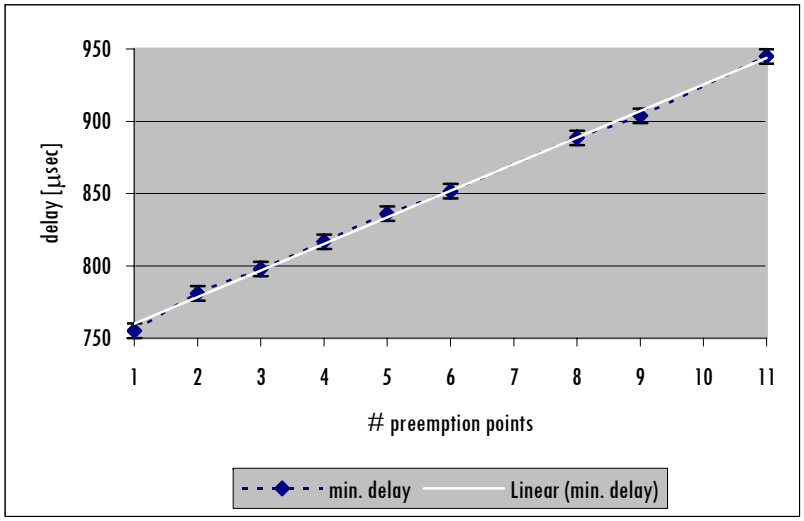

Figure 6-9: Minimum delay vs. number of preemption points

We can use the same method to determine the preemption point overhead by using the maximum delay instead of the minimum delay. To capture the maximum delay in a measurement run is much harder than capture the minimum delay. The reason for this becomes clear when looking at Figure 6-6 and Figure 6-7: For each 10ms time period, there is exactly one moment for which a packet will experience the maximum delay. We tried to find the maximum delay by performing very long measurement runs (each run 10'000 seconds). Figure 6-10 shows measurement results for maximum delay. Note that the variation here is larger than for the minimum delay. It is still less than +/-2%. Figure 6-11 shows the maximum delay versus number of preemption points. The resulting slope is identical to the slope for the minimum delay versus number of preemption points!

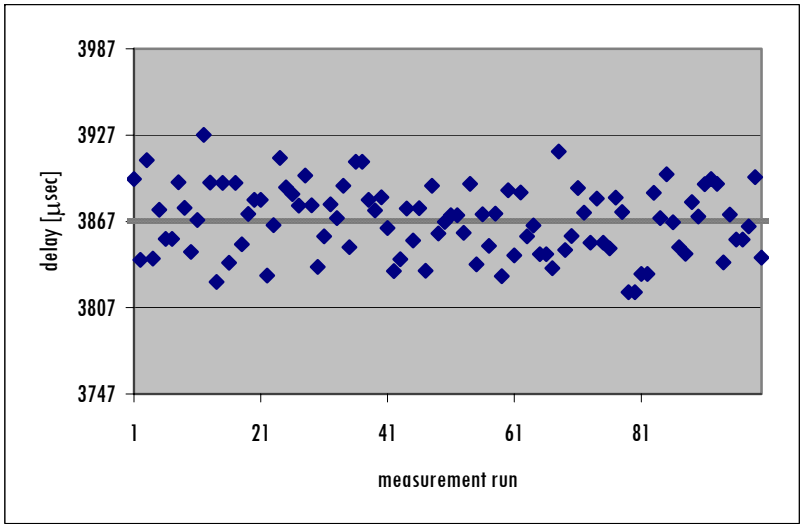

Figure 6-10: Maximum delay (2 preemption points)

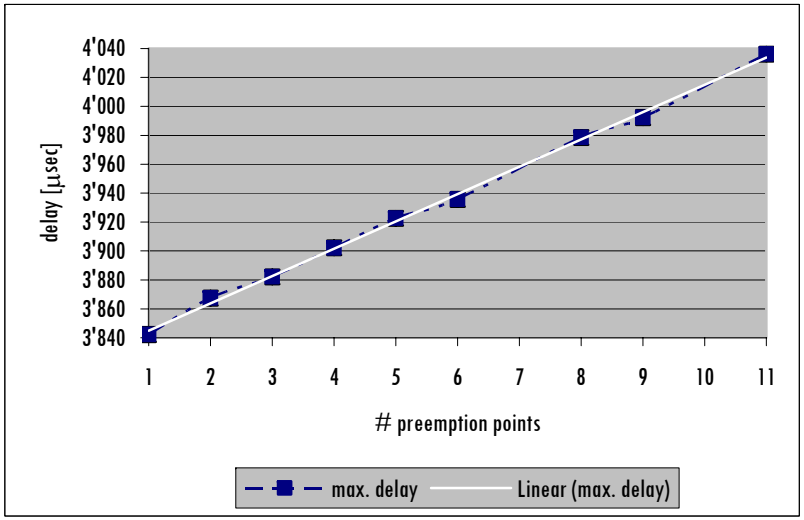

Figure 6-11: Maximum delay vs. number of preemption points

In summary, the *additional overhead per preemption point* is about 2.5% of the minimum delay for a single preemption point or about 0.5% of the maximum delay for a single preemption point.

Other results obtained are the minimum and maximum forwarding delay of a packet in case the RNOS process is idle but does not receive any surplus computation time form other processes (see Figure 6-12).

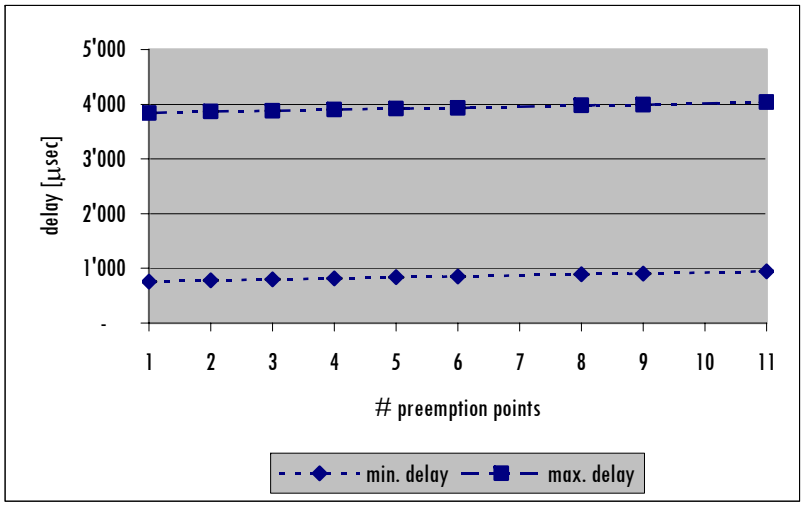

Figure 6-12: Minimum and maximum delay

Determining execution time of receive and filter task

As we know the minimum delay, we can easily calculate the execution time of the receive and filter task. The minimum delay is made up of the execution time for the forwarding path, the preemption point overhead and the execution time of the

receive and filter task. Therefore, (6.1) gives us what we are looking for.

$$e^u_{rx\&filter} = d_{min}(n) - e^u_f - no^u_{preemption-point} \quad (6.1)$$

Evaluating (6.1) for all possible number of preemption points results in $e^u_{rx\&filter}$=424µs +/-1%. This seems to be very high compared with the execution time of the forwarding path e^u_f=314µs. However, this time also includes the delay from the interrupt to the actual reception of the packet, which is incurred by the underlying operating system.

As we have now all the required attributes of RNOS on the target system, we can calculate the upper bound for the delay. Figure 6-13 shows the measured maximum and the calculated upper bound of the delay. The calculated upper bound is never surpassed by any measurement result. The minimal difference between the bound and the maximum measured is 8µs.

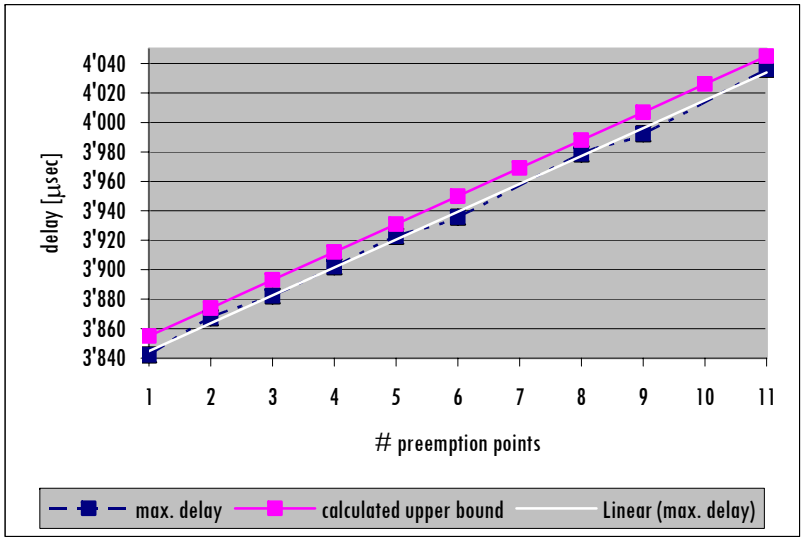

Figure 6-13: Measured max. and calculated upper bound of delay

Table 6-3 summarizes the attributes of RNOS on our target system.

Table 6-3: RNOS attributes and their values on target system

Attribute	Upper Value [µs]
Upper preemption point overhead $o^u_{preemption-point}$	19
Upper execution time of receive and filter task $e^u_{rx\&\,filter}$	424

6.1.6 Scheduleability Region of Example Implementation

In Chapter 5 we have proposed (6.2) to get the maximum number of preemption points that yield the minimum delay for highest-priority packets under a worst-case scenario.

$$n_{delay-min} = \sqrt{\frac{e^u_f\left(1 - r_{line} e^u_{rx\&\,filter}\right)}{2o^u_{preemption-point}}} \qquad (6.2)$$

Figure 6-14 shows (6.2) plotted for our example implementation. Here, more than two preemption points do not make sense and if the maximum line or burst rate that has to be supported is higher than 1200 packets/s, one preemption point is the right choice. However, this is only true for the worst-case scenario as modeled in Chapter 5. We will see in the next section that for other scenarios higher number of preemption points can make sense. Also, Figure 6-15 shows (6.2) with e^u_f=3140µs. For such an application, more preemption points are feasible.

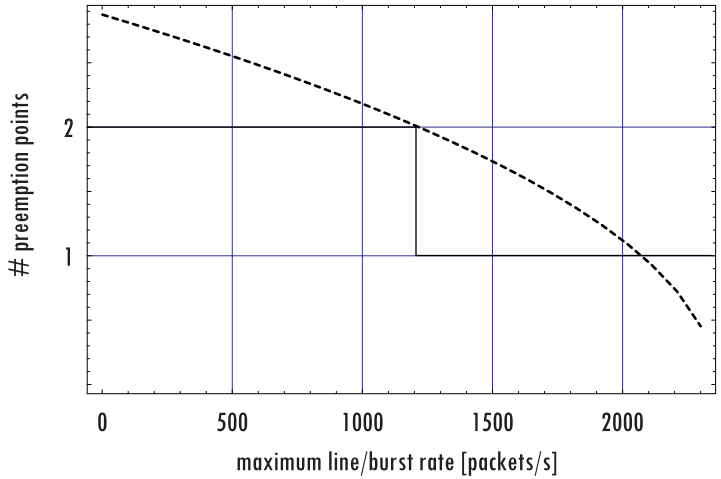

Figure 6-14: Maximum number of feasible preemption points for forwarding

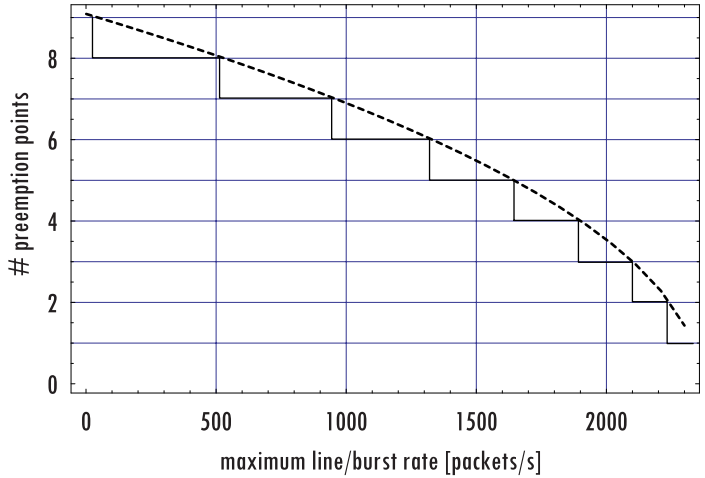

Figure 6-15: Maximum number of feasible preemption points for a long path

6.2 Scenarios

In this section we create scenarios, for which we will a) execute the calculation scheme of our model and b) execute measurements on the example implementation. We will then compare the two results.

We choose a simple scenario that consists of forwarding one real-time flow and three non real-time flows (best effort flows) (see Figure 6-16). The scenario is modified by altering the packet rate of the flows. We start with very low packet rates and then increase the rates for all flows. For the low rates, the system will be capable to process all packets, whereas for the highest rates, the system will have to drop most of the non real-time packets (but is still capable of forwarding all real-time packets).

Again, we will turn off all caches for the measurements in order to create a worst-case situation.

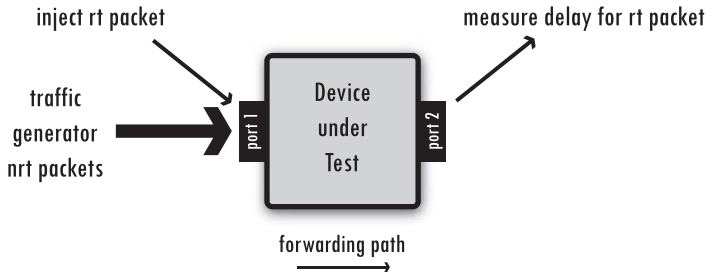

Figure 6-16: Measurement setup with one rt-flow and multiple nrt-flows

Table 6-4: Scenarios

Scenario #	Rate of teal-time flow [packets/s]	Rate of non real-time flows (all three flows together) [packets/s]
1	45	136
2	68	203
3	90	271
4	113	340
5	136	407
6	158	475
7	181	543
8	204	611
9	226	679
10	249	747
11	272	815
12	294	883
13	317	951
14	340	1'019
15	362	1'087
16	385	1'154
17	408	1'223
18	430	1'291

The calculations show that depending on the number of preemption points the worst-case delay is infinite, i.e. that the scenario is not feasible in the sense that an upper bound for the delay can be specified. Figure 6-17 shows the calculated upper bound for the delay for scenarios 1 to 14 and from 1 to 11 preemption points. The numbers in the legend are µs and the region above 14'000µs means "infinite", i.e. no upper bound can be specified. From the figure we also can see that

the upper bound for the delay is lowest for two and three preemption points from scenario 1 to 12, and that one preemption point provides the lowest upper bound for scenarios 13 and 14 (only with one preemption point we get an upper bound for scenario 14).

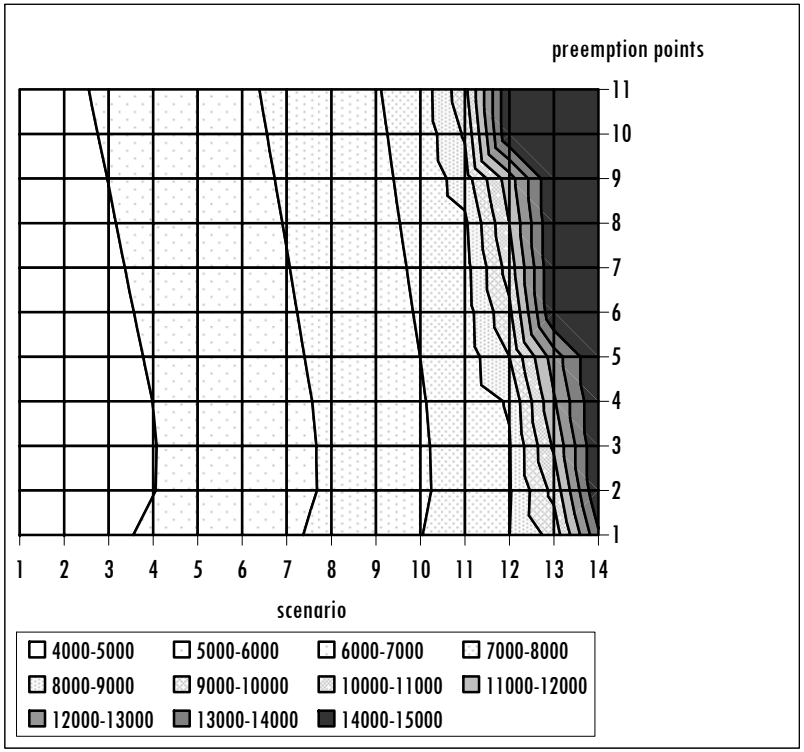

Figure 6-17: Calculation results for upper bound of delay

Figure 6-18 shows the measurement results for two preemption points and scenarios 1 to 14. The curve in the figure is the calculated upper bound for the delay, while the single dots are measurement results. We see that all measurement results lie below the calculated upper bound. For scenarios 14 and above,

the system also looses real-time packets and therefore the scenarios cannot be satisfied on the actual hardware.

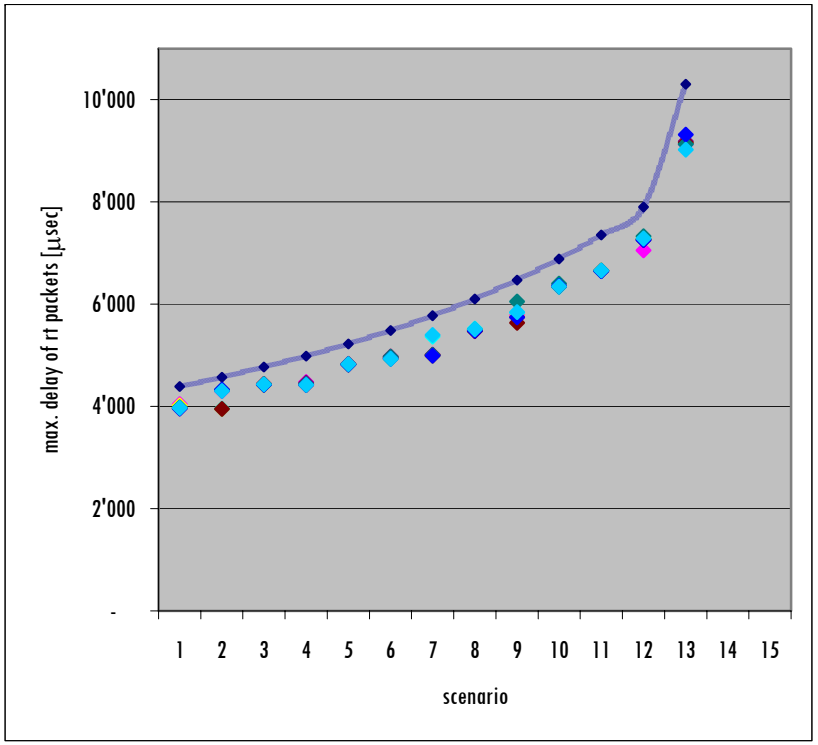

Figure 6-18: Maximum delay measurement (2 preemption points, real-time flow)

6.3 Summary

In this chapter we have presented our example implementation of a system based on RNOS. We have shown how RNOS can be integrated on top of a commercial real-time operating system on a specific CPU and how to achieve a guaranteed access to the computation resource. We have implemented a complete IP router based on RNOS. The division of the functionality into tasks made it easy to start with a minimal subset and enhance it task by task. For the determination of the RNOS attributes we used the IP forwarder path. The preemption point overhead of 19µs is small enough such that it can make sense for certain scenarios to have more than one preemption point. For the IP forwarding path it means 6% additional execution time per preemption point. Here, it might not make sense to have more than 2 preemption points. However, if we have longer executing paths, e.g. with encryption or compression functionality, it might well make sense to have more preemption points.

We have measured and calculated various scenarios; all of them show similar results:

- The calculated upper bounds are never surpassed by any measurement.
- The calculated upper bounds are “near” the actual measured worst-cast delay, meaning that our model closely represents reality.

The model stays true not only in theory, but also when implemented with RNOS.

7 Conclusion

In this thesis we have presented a method to build predictable packet processing systems on small, low cost hardware. The method is based on a domain specific model that is well suited for the modeling of such systems. It consists of an input sub-model, an application sub-model, a resource sub-model and the mappings between them. This domain specific model is used for the analysis and as a basis for the implementation.

The analysis method used in our model is based on network calculus. It allows us to explore the worst-case bounds of individual flows for delay and backlog, which is a pre-requisite for tailoring the hardware with adequate but not superfluous resources at competitive costs.

For a seamless transformation of the model to an implementation we have built the software platform RNOS. It provides a complete set of elements and services that are inherent in our model. Systems built with RNOS perform within the calculated bounds.

As with every software platform, RNOS introduces an overhead to a system. The overhead is decisively influenced by the scheduler; each preemption point adds to the overhead. For each application and scenario there is therefore an optimal number of preemption points. By using our analysis method, the influence of the number of preemption points on the delay and throughput of individual flows can be determined and the optimal number found, before the system is built.

The main results of this thesis are:

- A model that allows capturing packet processing systems and their scenarios.
- A software platform that allows a seamless implementation of the systems captured by the model.
- An analysis method for systems that are based on RNOS that allows calculating worst-case bounds for different application scenarios. The implementation will perform inside the calculated bounds.

The results of this thesis enable building predictable packet processing systems while

- reducing the development risks by allowing to explore a system before it is built and while
- reducing the hardware cost by allowing to tailor the hardware such that it exactly meets the requirements (no over-provisioning is necessary).

Our work gives rise to new questions and also provides various possibilities for improvements. A few of these are listed below:

- Currently, the calculations (for system exploration) are executed using a simple Mathematica [58] library. It re-

quires a lot of programming to get to results. A tool should be developed that allows the graphical modeling of packet processing systems and simplifies system explorations. One part that is currently particularly cumbersome is finding the sensitivity of parameters of the model regarding an exploration result. Having a tool would allow to include an automatic sensitivity analysis for all parameters, thereby making it easier to find critical parts of the system.

- In this thesis we only did look at the processing (CPU) and (marginally) at the memory resources. However, even small low cost embedded systems have resources that are shared between different processing engines, e.g. a DMA controller uses the same bus and memory as the CPU does. The resource model could be extended such that it can cope with multiple and possibly shared resources.
- The system performance can be improved by a more efficient use of the instruction caches. This leads to the idea called batch processing: Instead of processing only one packet of a microflow we might be able to collect a few packets and then process them as a batch; each packet processor is executed for each packet in the batch, thereby benefiting from better instruction cache usage. How this is to be included into our model is for further study
- Improve event handling by using polling mode where appropriate. The idea is to switch dynamically between interrupt based event reception and a polling mode. The interrupt is used to wake-up the idle system. It will then switch to the polling mode, which is active until all events are received (all input queues are empty). Then the system switches back to interrupt mode. A first prototype implementation for the Ethernet interfaces shows that we can

reduce $e^u_{rx\&\,filter}$. However, $e^u_{rx\&\,filter}$ does not change for the first packet on an idle system. For most application cases it makes no sense to include the behavior of the polling mode, as we are interested in the worst-case behavior to be able to give absolute guarantees. One idea is to include the polling mode into our model by analyzing the system for both modes, the interrupt mode and polling mode. In the polling mode we have a different value for $e^u_{rx\&\,filter}$. The polling mode is applicable when the system is under a defined and continuous load, while the interrupt mode is applicable for all other situations. However, whether this is feasible is for further study.

Bibliography

[1] Abhay K. Parekh and Robert G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: singlenode case," *IEEE/ACM Transactions on Networking*, vol. 1, pp. 344-357, 1993.

[2] Charles R. Kalmanek, Hemant Kanakia, and Srinivason Keshav, "Rate controlled servers for very high-speed networks," *GLOBECOM'90*, New York, NY, USA, 1990.

[3] K. Nichols, S. Blake, F. Backer, and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers," Internet Engineering Task Force (IETF) December 1998.

[4] Scott Shenker and John Wroclawski, "General Characterization Parameters for Integrated Service Network Elements," Request for Comments 2215, Internet Engineering Task Force (IETF), September 1997.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," Request for Comment 2475, Internet Engineering Task Force, December 1998.

[6] R. Keller, L. Ruf, A. Guindehi, and B. Plattner, "PromethOS: A dynamically extensible router architecture for active networks," *Proceedings of IWAN 2002*, Zurich, Switzerland, 2002.

[7] Scott Karlin and Larry Peterson, "VERA: An Extensible Router Architecture," *Computer Networks*, vol. 38, pp. 277-293, 2002.

[8] Lukas Kencl and J. Y. Le Boudec, "Adaptive load sharing for network processors," *Proceedings of Infocom*, 2002.

[9] M. Gries, "Algorithm-Architecture Trade-offs in Network Processor Design," in *Computer Engineering and Networks Laboratory*: Swiss Federal Institute of Technology Zurich, 2001.

[10] P. Crowley, M. Fiuczynski, and J.-L. Bear, "On the performance of multithreaded architectures for network processors," Department of Computer Science, University of Washington 2000 2000.

[11] P. Druschel, L. Peterson, and B. Davie, "Experiences with a high-speed network adatpor: A software perspective," *Proceedings of ACM SIGCOMM '94*, 1994.

[12] Jeffery C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-drivern kernel," *ACM Transactions on Computer Systems*, vol. 15, pp. 217-252, 1997.

[13] E. Kohler, "The Click Modular Router," in *Department of Electrical Engineering and Computer Science*: Massachusetts Institute of Technology, 2000.

[14] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek, "The Click Modular Router," *ACM Transactions on Computer Systems*, vol. 18, pp. 263-297, 2000.

[15] Julian Elischer and Archie Cobbs, "The Netgraph networking system," Whistle Communication January 1998.

[16] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner, "Router plugins: A software architecture for next generation routers," *Proceedings of ACM SIGCOMM '98*, 1998.

[17] Kenjiro Cho, "A framework for alternate queueing: towards traffic management by PC-UNIX based routers," *Proceedings of USENIX 1998 Annual Technical Conference*, 1998.

[18] B. Stiller, "The Design and Implementation of a Flexible Middleware for Multimedia Communication Comprising Usage Experience," July 1998.

[19] J. Nieh and M.S. Lam, "The design, implementation and evaluation of SMART: A scheduler for multimedia applications," *Proceedings of 16th ACM Symposium on Operating System Priniciples*, 1997.

[20] C. Hutchinson and P. Peterson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, 1991.

[21] Larry L. Peterson, Scott C. Karlin, and Kai Li, "OS Support for general purpose routers.," *Proceedings of 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, 1999.

[22] David Mosberger and Larry L. Peterson, "Making Paths Explicit in the Scout Operating System," *Operating Systems Design and Implementation*, 1996.

[23] X. Qie, A. Bavier, L. Peterson, and S. Karlin, "Scheduling Computations on a Programmable Router," *ACM SIGMETRICS 2001 Conference*, 2001.

[24] P. Pappu and T. Wolf, "Scheduling Processing Resources in Programmable Routers," *Twenty-First IEEE Conference on Computer Communications (INFOCOM)*, New-York, NY, USA, 2001.

[25] Dan S. Decasper, "A Software Architecture for Next Generation Routers." Zurich: Swiss Federal Institute of Technology Zurich, 1999.

[26] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden, "A Survey of Active Network Research," *IEEE Communications Magazine*, vol. 35, pp. 80-86.

[27] Prashant Pradhan and Tzi-Cker Chiueh, "Computation framework for an extensible network router: Design, implementation and evaluation," 2000.

[28] Jonas Greutert and Lothar Thiele, "RNOS: A Middleware Platform for Low Cost Packet Processing Devices," *Third Workshop on Network Processors & Applications (NP-3) at the 10th International Symposium on High-Performance Computer Architecture (HPCA-10)*, Madrid, Spain, 2004.

[29] Jonas Greutert and Lothar Thiele, "RNOS - A Middleware Platform for Low-Cost Packet-Processing Devices," in *Network Processor Design, Issues and Practices Volume 3*, P. Crowley, M. A. Franklin, H. Hadimioglu, and P. Z. Onufryk, Eds.: Morgan-Kaufmann, 2005, pp. 173-195.

[30] Rene L. Cruz, "A calculus of network delay. Part I. Network elements and isolation," *IEEE Transactions on Information Theory*, vol. 37, pp. 114-131, 1991.

[31] Rene L. Cruz, "A calculus for network delay, Part II: Network analysis," *IEEE Transactions on Information Theory*, vol. 37, pp. 132-141, 1991.

[32] Jean-Yves Le Boudec and Patrick Thiran, *Network calculus : a theory of deterministic queuing systems for the Internet*. New York: Springer, 2001.

[33] J. Engblom, A. Ermedahl, M. Sjoedin, J. Gustafsson, and H. Hansson, "Worst-case execution-time analysis for embedded real-time systems," *Journal of Software Tool and Transfer Technology (STTT)*, vol. 4, pp. 437-455, 2003.

[34] Patrick Crowley and Jean-Loup Bear, "A Modeling Framework for Network Processor Systems," *First Workshop on Network Processors (NP1) at the 8th International Symposium on High Performance Computer Architecture (HPCA8)*, Cambridge, MA, USA.

[35] Lothar Thiele, Samarjit Chakraborty, Matthias Gries, Alexander Maxiaguine, and Jonas Greutert, "Embedded Software in Network Processors - Models and Algorithms," *Lecture Notes in Computer Science*, pp. 416-434, 2001.

[36] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele, "Real-time calculus for scheduling hard real-time systems," *IEEE International Symposium on Circuit and Systems (ISCAS)*, 2000.

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

[38] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm, "The Influence of Processor Architecture on the Design and the Results of WCET Tools," *Proceedings of IEEE on Real-Time Systems*, vol. 91, pp. 1038-1054, 2003.

[39] J. Janssen, Danny De Vleeschauwer, and Guido H. Petit, "Delay and distortion bounds for packetized voice calls of traditional PSTN quality," *Proceedings of the 1st IP-Telephony Workshop (IPTel 2000)*, pp. 105-110, 2000.

[40] International Telecommunication Union, "Pulse code modulation (PCM) of voice frequencies," Recommendation G.711, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, November 1998.

[41] International Telecommunication Union, "Packet based multimedia communication systems," Recommendation H.323, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, February 1998.

[42] F. Andreasen and B. Foster, "Media Gateway Control Protocol (MGCP) Version 1.0," Request for Comment (Informational) 3435, Internet Engineering Task Force, January 2003.

[43] F. Cuervo, N. Greene, A. Rayhan, C. Huitema, B. Rosen, and J. Segers, "Megaco Protocol Version 1.0," Request for Comment (Proposed Standard) 3015, Internet Engineering Task Force, November 2000.

[44] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, "SIP: Session Initiation Protocol," Request for Comment (Proposed Standard) 2543, Internet Engineering Task Force, March 1999.

[45] M. Handley and V. Jacobson, "SDP: Session Description Protocol," Request for Comment (Proposed Standard) 2327, Internet Engineering Task Force, April 1998.

[46] Van Jacobson, "Congestion Avoidance and Control," *In Proceedings SIGCOMM '99*, pp. 314-329, 1988.

[47] J. Padhye, V. Firoiu, D. Towsley, and J. Krusoe, "Modeling TCP Throughput: A Simple Model and its Empirical Validation," *Proceedings of the ACM SIGCOMM '98*, pp. 303-314, 1998.

[48] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Hard Real-Time Scheduling: The Deadline Monotonic Approach," *in Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, pp. 127-132, 1991.

[49] JTC1 / SC22, "Programming languages -- C++," ISO/IEC 14882:2003, International Organization for Standardization, November 2003.

[50] "NetBSD," www.netbsd.org.

[51] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein, "Register allocation via coloring," in *Computer Languages*, vol. 6, pp. 47-57.

[52] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource Reservation Protocol (RSVP) -- Version 1 Functional Specification," Request for Comment 2205, Internet Engineering Task Force, September 1997.

[53] WindRiver, "VxWorks," www.windriver.com.

[54] Microsoft, "Windows CE," www.microsoft.com/windows/embedded.

[55] John Morris, "Data Structures and Algorithms," University of Western Australia 1998.

[56] Freescale, "MPC850," www.freescale.com.

[57] Spirent, "SmartBits," www.spirent.com.

[58] Wolfram Research, "Mathematica," www.wolfram.com.

[59] The MathWorks, "Simulink", www.mathworks.com

List of Tables & Figures

Table 3-1: SLA parameters for the VoIP data flow 3-11
Table 3-2: Flow specification 3-13
Table 3-3: Initial results 3-14
Table 4-1: Tasks in the two port Etherent router 4-5
Table 4-2: Context information in packet processing tasks 4-11
Table 5-1: Analogy between RTOS and RNOS 5-33
Table 6-1: Tasks in the forwarding path 6-7
Table 6-2: Additional functionality that is available as tasks for RNOS 6-8
Table 6-3: RNOS attributes and their values on target system 6-17
Table 6-4: Scenarios 6-21

Figure 1-1: Block Diagram of a typical Embedded Communication Controller 1-4
Figure 1-2: Measured throughput in packets per second for different packet sizes 1-5
Figure 1-3: Outline of the thesis 1-12
Figure 2-1: Model overview 2-5
Figure 2-2: Single Token Bucket 2-6
Figure 2-3: Dual Token Bucket for TSpec 2-8
Figure 2-4: (σ, ρ) model as upper arrival curve 2-10
Figure 2-5: TSpec as upper arrival curve 2-11
Figure 2-6: Arrival function for constant rate packet source with network jitter 2-12
Figure 2-7: Arrival curves for constant rate packet source with network jitter 2-13
Figure 2-8: Arrival Curves for smallest packets back-to-back on line 2-13
Figure 2-9: Tasks types 2-16
Figure 2-10: Simplified task graph for packet reception that contains three paths 2-19
Figure 2-11: Task graph from DSP to Ethernet 2-19
Figure 2-12: Simplified task graph for packet reception with IPSec support 2-21
Figure 2-13: Path through a task graph 2-23
Figure 2-14: Resource Access Pattern 2-27
Figure 2-15: Upper and lower service curves for a CPU resource 2-27
Figure 3-1: Bounds on delay and backlog 3-3
Figure 3-2: Physical processing model 3-4
Figure 3-3: Basic calculation layout 3-5
Figure 3-4: Calculation scheme for one flow 3-8
Figure 3-5: Calculation scheme for multiple flows 3-9
Figure 3-6: Example System 3-10
Figure 3-7: Arrival curves for VoIP data receive flow 3-12
Figure 3-8: Service curves for computation capacity 3-14
Figure 3-9: Delay of kids & family and business flow for different CPU clock speeds 3-15
Figure 4-1: Model of two port router 4-4
Figure 4-2: Possible implementation model of the two port router 4-7
Figure 4-3: Another implementation model of the two port router 4-8
Figure 4-4: Task graph of a two port router annotated with instance information 4-15
Figure 4-5: Packet and Path-Thread 4-17
Figure 4-6: Source Flow from Splitting of Path 4-19
Figure 5-1: Task objects with connectors 5-5
Figure 5-2: Connectors must match in payload type and required annotations 5-7

Figure 5-3: Basic Concept of Task Frames 5-8
Figure 5-4: Link Layer with Task Frames 5-9
Figure 5-5: Task Object and Packet Processor 5-11
Figure 5-6: Packet path optimization 5-12
Figure 5-7: Scope on packet data 5-14
Figure 5-8: Object Relation Diagram with Microflow Object 5-20
Figure 5-9: Source Thread creates Path-Threads 5-24
Figure 5-10: Scheduler Architecture with EDF Algorithm 5-26
Figure 5-11: Overview of some Elements of RNOS 5-29
Figure 5-12: RNOS running in an RTOS process 5-30
Figure 5-13: RNOS running on RTOS as a process 5-31
Figure 5-14: RNOS' higher level of abstraction API 5-32
Figure 5-15: Virtual Tasks 5-35
Figure 5-16: Impact of Overhead on Throughput 5-41
Figure 5-17: Example alignment of phases 5-44
Figure 5-18: Worst-case waiting time for a high-priority packet 5-46
Figure 5-19: Upper bounds for waiting times for a highest priority packet 5-50
Figure 5-20: Worst-case overhead time for a highest-priority packet 5-51
Figure 5-21: Upper bound of delay for highest priority packet 5-54
Figure 5-22: Feasible number of preemption points, depending on e^{u}_{packet} 5-55
Figure 5-23: Calculation scheme for RNOS 5-58
Figure 5-24: Thread-switch pattern 5-59
Figure 5-25: Arrival curves of source flow 5-60
Figure 5-26: Example arrival curves of high-priority flow 5-61
Figure 5-27: Service curves of processing resource 5-61
Figure 5-28: Upper bound for delay for packets of highest priority flow 5-63
Figure 6-1: Block Diagram of Embedded Controller 6-3
Figure 6-2: RNOS on VxWorks 6-4
Figure 6-3: Lower and upper service curve for RNOS on target system 6-6

Figure 6-4: Measurement setup 6-9
Figure 6-5: Delay measurement result (11 preemption points) 6-10
Figure 6-6: Sorted measurement results (ascending delay) 6-11
Figure 6-7: Delay of single packet 6-11
Figure 6-8: Minimum delay measurement (2 preemption points) 6-12
Figure 6-9: Minimum delay vs. number of preemption points 6-13
Figure 6-10: Maximum delay (2 preemption points) 6-14
Figure 6-11: Maximum delay vs. number of preemption points 6-14
Figure 6-12: Minimum and maximum delay 6-15
Figure 6-13: Measured max. and calculated upper bound of delay 6-17
Figure 6-14: Maximum number of feasible preemption points for forwarding 6-19
Figure 6-15: Maximum number of feasible preemption points for a long path 6-19
Figure 6-16: Measurement setup with one rt-flow and multiple nrt-flows 6-20
Figure 6-17: Calculation results for upper bound of delay 6-22
Figure 6-18: Maximum delay measurement (2 preemption points, real-time flow) 6-23