# RNOS – A Middleware Platform for Low Cost Packet Processing Devices

Jonas Greutert
*NetModule AG, Switzerland*
*jonas.greutert@netmodule.com*

Lothar Thiele
*ETH Zurich, Switzerland*
*thiele@tik.ee.ethz.ch*

## Abstract

*There is an increasing demand in small networked embedded systems. As packet processing is only a part of the whole application, these devices typically do not contain a highly specialized network processor but a standard communication controller with limited resources. The number of packets that are processed by these systems is low compared to standard routers. On the other hand, they typically have stringent real-time requirements. In this paper, we describe a middleware platform, RNOS, which merges the system analysis with the system implementation to enforce predictable system behavior. Experimental results are provided that show a good agreement between the properties of an implementation using RNOS and the corresponding formal analysis.*

## 1. Introduction

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 the processing power, i.e. the number of packets per second that can be processed, and the extensibility of those systems. The questions that are addressed are optimal design, new architectures, better algorithms and implementation methods and tools, see e.g. [1-3].

There is another 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 with network attachment, providing a variety of services. Although these devices do not have a high capacity to process packets, they often have stringent demands to the real-time processing of certain flows. The keeping of deadlines is more important than the actual number of packets that can be processed. Just being faster would not help in most cases. 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 part of the complete application that is running on these devices, although a critical part with respect to predictability.

Significant work has been done in developing architectures for software based routers. In Click [4] applications are composed of 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. The Scout OS [5] has explicit paths to improve resource allocation and scheduling. It is a soft real-time system that provides admission control with respect to CPU load and memory, but it does not provide mechanisms to calculate backlog and delay of individual flows. 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 [6]. 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 [7]. It also contains a concept for online estimation of processing times and an admission control. Again, no real-time guarantees can be provided. Summarizing, there are solutions available for various issues in software based routers. A unifying approach for small embedded devices is missing that enables as well a formal analysis as an implementation that matches the predicted behavior.

This paper describes RNOS (Real-time Network Operating System), a middleware platform for low-cost packet processing devices with real-time requirements. The RNOS consists of an analysis model and an implementation model. The analysis model allows the exploration of packet processing applications for different input and resource scenarios with real-time requirements. The implementation model allows a seamless implementation of the analysis model on a single CPU system and guarantees the real-time behavior provided by the formal analysis of the model. As a result of the matching analysis and implementation model, we obtain a platform for the design of predictable small embedded devices.

The remainder of the paper is organized as follows: Section 2 describes the scenario we use throughout the following sections to illustrate the applicability of the approach. Section 3 presents the analysis model and Section 4 describes the implementation model of the software platform. In Section 5 we report measurement results and compare them with analysis results. Finally, we summarize and comment on future work in Section 6.

## 2. Scenario

A single scenario is used to illustrate the concepts presented in this paper. An existing home access router shall be extended with Voice over IP (VoIP) functionality. The hardware shall remain the same, with the only exception of an additional module that will be plugged to an existing extension port. That additional module, called VoIP-module, has the necessary DSP resources to code/decode the voice packets and detect/play DTMF tones and it has all the physical interfaces required to connect traditional phones or a PBX. Figure 1 shows the block-diagram of the hardware used in this scenario.

As an engineer we are faced with the following questions. How many voice channels are possible? Will the VoIP feature degrade existing functionality and by how much?

The hardware of the home access router is based on a commercial communication processor. It has two Ethernet interfaces and an extension port for the VoIP functionality. The extension port is a 16-Bit host port interface that connects seamlessly to the DSP. An external SDRAM provides the necessary memory for packets, data and the application. Independent DMA controllers and the CPU arbitrate for the memory. Dedicated hardware units are responsible for reception and transmission of Ethernet packets. The communication controller runs with a clock frequency of 50MHz. The CPU has 4kbyte instruction and data cache. The total bill of material is less than US$ 100 for low volumes, including the VoIP module.

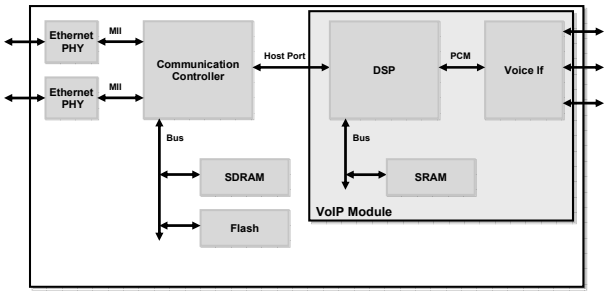


**Figure 1: Block-diagram of the low-cost Embedded System used in the Scenario**

## 3. Analysis Model of RNOS

The platform RNOS consists of two closely related parts, an analysis part and an implementation part. The purpose of the analysis part is to model the whole system in terms of application, input scenarios and resource usage and to provide a methodology that enables the analysis of relevant system properties. The implementation part which will be discussed in Section 4 provides a middleware infrastructure and a programming interface to implement those systems. It is constructed in a way that matches the formal models in the analysis part such that the predicted system behavior is obtained.

The analysis part of RNOS consists of formal models for the application, the input scenarios in terms of packet streams, the available hardware and software resource, and of a calculus, which allows determining the throughput and delaying as experienced by the input streams. The basic model is taken from [8] and extended towards the networking domain and small embedded devices.

### 3.1. Application Model

The application model defines the required functionality of the system. To be useful, the application model has to be easy to use and has to be able to capture the domain specific functionality [9]. One way of modeling the whole application is based on a partitioning into small processing units that will be denoted as tasks. In addition, the application model will be based on the notion of events, e.g. a packet has been received or a timer elapsed. The combination of these concepts leads to the natural model that each event in the system has its own "program" that is triggered for execution when the event occurs and that consists 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 one of its outputs. A source task has no inputs and will receive its packets from a driver, e.g. from an Ethernet driver. A sink task has no outputs and will either consume the packet or it will pass the packet to a driver for transmit, e.g. to the Ethernet driver, or pass it to the user mode, e.g. to the socket interface. The worst case and best case execution times of each task need to be known, either by formal analysis or by simulation in case of soft real-time constraints.

**Task Trees**

A whole application contains a set of task trees 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 tree with a source task at its root. Therefore, an application consists of as many task trees as there are packet sources and each task tree has exactly one source.

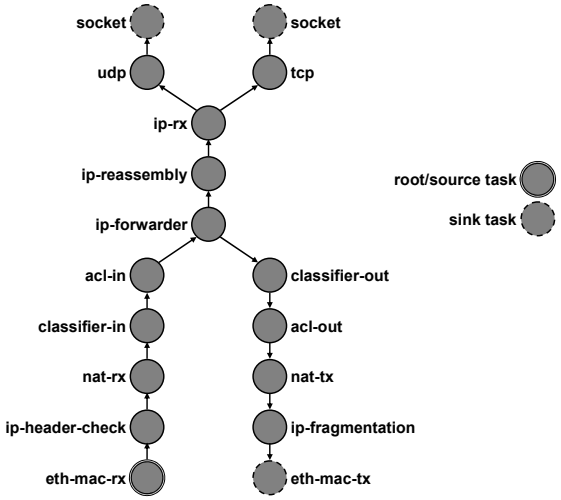


**Figure 2: Simplified task tree for packet reception that contains three paths**

A packet will traverse the task tree from the source to a sink. The source task will receive a packet, e.g. from Ethernet, process it and pass it to one of its outputs depending on the content of the packet where it is processed by the subsequent task. A packet will follow exactly one path when it traverses the task tree.

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

**Example 3.1** *In our scenario, we add the processing for VoIP traffic to the standard IP router functionality. Voice is especially sensitive to delay and delay jitter. It has been shown that a significant source of delay is the end-system itself [10]. Therefore, we add an optimized path for VoIP data packets to our task graph. To use the standard path for IP/UDP processing and pass the packet to the socket interface would be an unnecessary overhead. Figure 3 shows the extended task graph for packet reception with VoIP.*

*The voip-data-mux task has a filter that is optimized to filter VoIP data traffic. If the filter matches, it is a VoIP data packet. Then the packet is passed to a combined and highly optimized IP/UDP receive task. The RTP (Real-time Transport Protocol) receive task follows and finally the packet is passed to the DSP for decoding. The reverse direction is similar (not shown): DSP, RTP, UDP/IP and finally Ethernet transmit.*

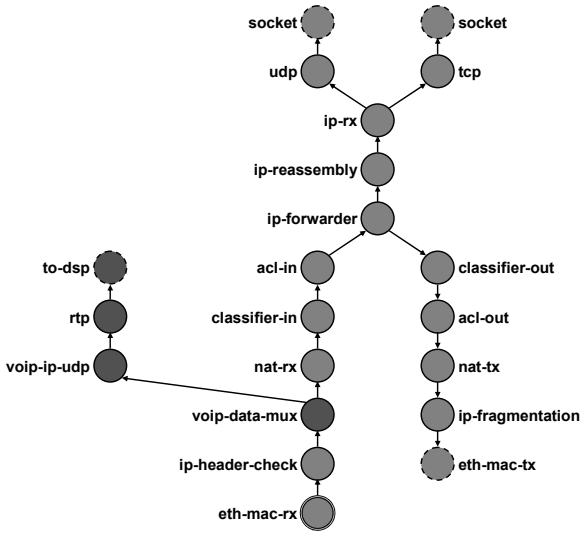


**Figure 3: Extended task tree for packet reception with VoIP**

### 3.2. Input Model – SLA, Flows and Microflows

The input model has to capture packet flows that are specified using service level agreements and packet flows, for which we do not know exactly (or know nothing) about how they will arrive at the system inputs.

A service level agreement (SLA) specifies end-to-end quality of service properties for a flow. A flow is identified by a set of common properties derived from data in the packet. Typically these are the incoming interface, ranges of source and destination IP addresses, transport protocol and ports or port ranges. A service level agreement usually contains parameters as minimum bandwidth, maximum delay, loss probability and maximum jitter. Another form of an SLA is the T-Spec model of IETF [11].

Each packet that is received by our application belongs to a flow. There is a service agreement for each flow, which specifies the end-to-end properties of that flow and therefore specifies how we should treat the packet inside our application, e.g. with what priority the packet should traverse the task graph. All the packets that do not match a flow are associated with the best effort flow. The best effort flow has no end-to-end quality of service properties and therefore has lowest priority.

Packets of the same flow might not traverse the same path in the task tree. A flow specification might cover a larger set of connections, a connection being 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 our model such a connection is called microflow. Packets that belong to the same microflow will traverse exactly the same path through the task tree. Figure 4 shows a flow that consists of two microflows; each of those traverses a different task path.

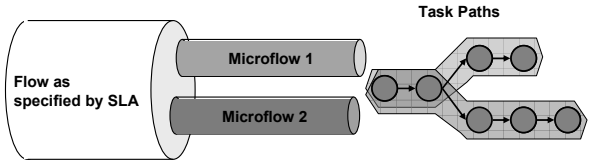


**Figure 4: Flow and Microflow**

Service level agreements can easily be formalized as arrival curves and deadlines, see e.g. [8], [12].

**Definition 3.1** *$\alpha^l(\Delta)$ is the minimum number of packets that arrive in any time interval of length $\Delta$. Similar, $\alpha^u(\Delta)$ is the maximum number of packets that arrive in any time interval of length $\Delta$.*

The upper and lower arrival curves specify the bounds for arrival of traffic. Therefore, we can model the uncertainties of the arrival of packets. The deadline is an internal deadline for the execution of packets of that flow. It is specified depending on the priority of a flow.

Summarizing, a flow consists of one or more microflows and its quality of service requirements are specified using arrival curves and a deadline. All packets of a microflow will pass the same task path and in the order of arrival.

**Example 3.2** *In our scenario, we have a flow for the VoIP data receive and transmit traffic, a flow for the VoIP control receive and transmit traffic and a best effort flow for all other input packets.*

*The SLA for the VoIP data receive traffic is very stringent if a quality comparable to traditional PSTN quality should be reached [13]. The SLA for VoIP-data (uni-directional) could be specified as follows:*

| | |
|---|---|
| *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 multiply it with the number of packets per second. For G.711 [19] with a packet period of 10ms this gives 138 bytes per packet which is about 108kbit/s per direction (14 to 18 bytes Ethernet 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 an arrival curve and a deadline. The arrival curve gives 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 burst of 12 VoIP data packets at line rate. The line rate in our scenario is 10Mbit/s. A packet of the size of 138bytes requires about 106$\mu$s on the line. Figure 5 shows the arrival function for the VoIP data receive traffic. 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, we set the deadline to the execution time for the task path on the target system. Therefore, the flow will have maximum priority.*

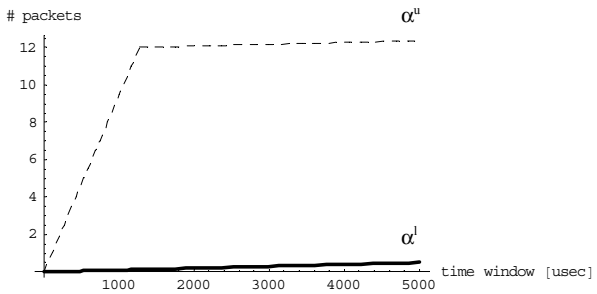


**Figure 5: Upper and lower arrival curve for a VoIP data traffic receive flow**

*The VoIP data transmit traffic is a constant packet rate flow. The data packets are generated at a constant rate by the DSP. Usually, it is a 10ms period for G.711 and longer periods for other coders. VoIP control traffic occurs mostly at call setup and teardown. During the calls, not much information is exchanged between the endpoints or an endpoint and gatekeeper. The amount of also depends on the actual protocol and which variant thereof is used, see also [14-18].*

*The best-effort flow has an upper arrival curve that is equal to the line rate. The deadline of the best effort flow is infinite and therefore it has lowest priority.*

## 3.3. Resource Model

The resource model for the target systems, i.e. small, low-cost, single CPU systems, is simple. We model the single CPU resource using a service curve as proposed in [8].

**Definition 3.2** *$\beta^l(\Delta)$ is the minimum available CPU time in any time interval of length $\Delta$. Similar, $\beta^u(\Delta)$ is the*

*maximum available CPU time in any time interval of length Δ.*

Therefore, upper and lower service curves specify the bounds for the availability of the CPU resource. They allow us to work with uncertain availability of the resource. We will denote the available CPU time as CPU capacity in the following.

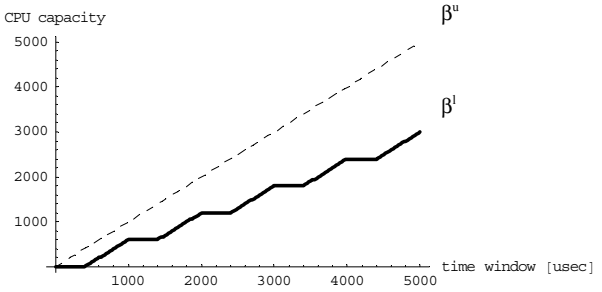


**Figure 6: Upper and lower service curves for a CPU resource**

***Example 3.3*** *In the scenario, the CPU of the communication controller runs a standard real-time operating system. The real-time operating system will provide 6ms of a 10ms interval to the execution of the tasks. The other 4ms will be used to run other (user-level) applications. However, if no other applications require the CPU, it is available to the packet processing. Figure 6 shows the upper and lower service curves for the CPU resource as it is available for the execution of tasks. The CPU capacity divided by the time interval is the fraction of the total CPU resource that is available.*

### 3.4. Calculus

The methodology to analyze important system properties is based on the calculus that has been introduced in [8]. It allows determining the throughput and delay as it is experienced by the various input flows. The inputs to the calculus are the arrival curve, the deadline, the worst case and best case execution times for each flow and the service curves of the resources.

Each task $t_i$ has an upper and lower execution time $e_i^u$ and $e_i^l$. Estimates of the upper and lower 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.

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 upper and lower execution time $e_{p_j}^u$ and $e_{p_j}^l$ of a task path $p_j$ is the sum of the execution times $e_i^u$ and $e_i^l$ of its tasks $t_i$, i.e.

$$e_{p_j}^u = \sum_{i \in p_j} e_i^u \qquad e_{p_j}^l = \sum_{i \in p_j} e_i^l$$

$$e_T^u = \max_{p_j \in P} e_{p_j}^u \qquad e_T^l = \max_{p_j \in P} e_{p_j}^l$$

where $e_T^u$ and $e_T^l$ denote the maximum upper and lower execution time of a task tree $T$.

A flow $F$ contains a set $M_F$ of microflows $m_k \in M_F$. The upper and lower execution time $e_{m_k}^u$ and $e_{m_k}^l$ of a microflow $m_k$ is equal to the upper and lower execution time $e_{p_j}^u$ and $e_{p_j}^l$ of its associated task path $p_j$ , and $e_F^u$ and $e_F^l$ denote the maximum upper and lower execution time of a flow $F$, i.e.

$$e_{m_k}^u = e_{p_j}^u \qquad e_{m_k}^l = e_{p_j}^l$$

$$e_F^u = \max_{m_k \in F} e_{m_k}^u \qquad e_F^l = \max_{m_k \in F} e_{m_k}^l$$

Now, we can determine the upper and lower arrival curves $\alpha^{u'}$ and $\alpha^{l'}$ that describe the processed packet stream and the upper and lower service curves $\beta^{u'}$ and $\beta^{l'}$ that describe the remaining CPU capacity. The equations are derived from the real-time calculus that has been used in [8] to estimate the worst-case performance of complex network processor architectures. The equations are slightly different from those in [8] and yield closer bounds. At first, we define some operators that will be used:

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

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

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

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

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

Using these definitions, the required relations can be expressed as follows:

$$\alpha^u = e_F^u \overline{\alpha}^u \qquad \alpha^l = e_F^l \overline{\alpha}^l \qquad (1)$$

$$\overline{\alpha}^{u'} = \lceil \alpha^{u'} / e_F^u \rceil \qquad \overline{\alpha}^{l'} = \lfloor \alpha^{l'} / e_F^l \rfloor \qquad (2)$$

$$\alpha^{u'} = \left(\left(\alpha^u \underline{\oplus} \beta^u\right) \overline{\otimes} \beta^l\right) \wedge \beta^u \quad (3)$$

$$\alpha^{l'} = \left(\left(\alpha^l \overline{\otimes} \beta^u\right) \underline{\oplus} \beta^l\right) \wedge \beta^l \quad (4)$$

$$\beta^{u'} = \left(\beta^u - \alpha^l\right) \underline{\otimes} 0 \quad (5)$$

$$\beta^{l'} = \left(\beta^l - \alpha^u\right) \overline{\oplus} 0 \quad (6)$$

Equations (1) and (2) are used to scale the arrival of packets with the execution times and back. The resulting worst case delay of packets can be calculated to be the maximal horizontal distance between the upper arrival and lower service curves, i.e.

$$delay \le \sup_{u \ge 0}\left(\inf\left(\tau \ge 0 : \alpha^u(u) \le \beta^l(u+\tau)\right)\right)$$

The calculations have been derived for preemptive tasks. As the tasks in the application model of RNOS are non-preemptive execution blocks, the actual worst case delay is the calculated delay plus the execution time of the longest task, i.e. $delay + \max_i(e_i)$. For best effort we use the worst case and best case execution times $e_T^u$ and $e_T^l$ instead of $e_F^u$ and $e_F^l$. Figure 7 represents graphically the calculation scheme that is applied.

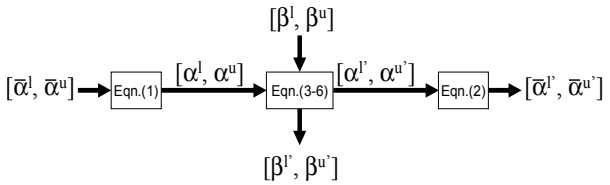


**Figure 7: Calculation scheme**

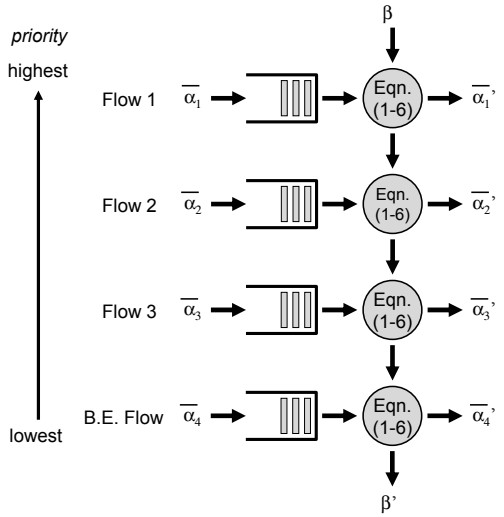


**Figure 8: Fix-priority calculation scheme**

To get the priority of the flow *F*, we divide the deadline $d_F$ by the upper execution time $e_F^u$ of the flow. A lower number denotes a higher priority. Now we can repeatedly apply the calculation scheme shown in Figure 7 to the fixed priority scenario as shown in Figure 8.

**Example 3.4** *In our scenario we would like to know how many voice channels can be processed and what the impact is on the best effort forwarding service. As packets can arrive at line rate and the system is not capable of processing all the packets at line rate, there must be a flow filter early in the task tree. Up to the flow filter, the system has to process all the packets that arrive at the system. Only after the flow filter, the system will know which packets to prioritize. Figure 9 shows the added flow-filter task in the task tree for packet reception.*

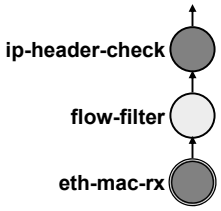


**Figure 9: Added flow-filter task in task tree for packet reception**

*Table 1 shows the upper execution times and the deadlines for the flows, including the special flow-filter flow, which includes the tasks eth-mac-rx and flow-filter. The flow-filter flow contains all packets. The arrival curve for the flow-filter flow is the maximum packet rate (smallest packet size divided by line rate). The arrival curves of the other flows have been discussed previously.*

**Table 1: Execution Times and Deadline of Flows**

| *Flow* | *Upper Execution Time* | *Priority/ internal Deadline* |
|---|---|---|
| Flow Filter | 22µs | 22µs |
| VoIP Data Receive | 68µs | 1000µs |
| VoIP Data Transmit | 48µs | 2000µs |
| VoIP Control Receive | 678µs | 100'000µs |
| VoIP Control Transmit | 778µs | 100'000µs |
| Best Effort Packet Forwarding | 258µs | Infinite |

*Figure 10 depicts the best effort traffic forwarding rate depending on the number of active voice channels.*

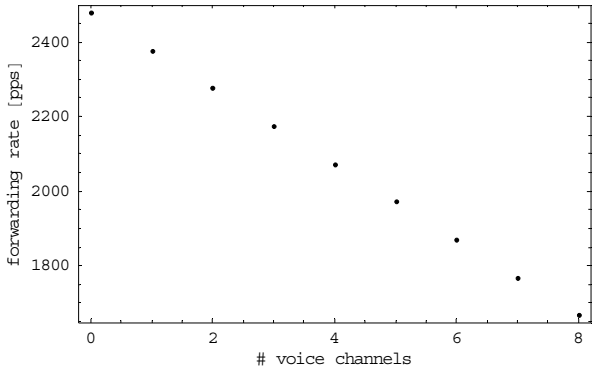


**Figure 10: Best effort traffic forwarding rate as a function of active voice channels**

*The above described calculus allows us to calculate the backlog and delay of each individual flow.*

*As an example, we can determine the influence of the worst-case arrival jitter of voice data traffic on the delay of the receive control traffic. Figure 11 depicts the results for a worst-case jitter of 0, 60, 120 and 180ms.*

With the calculus and the model it is easy to determine weak points in the system and where optimizations are worth to consider. It is obvious, that the implementation of the flow-filter should be very efficient; otherwise, most of the available resources will be spent filtering input traffic.

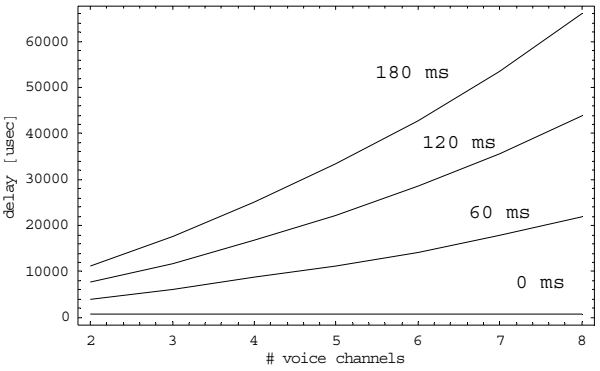


**Figure 11: Delay of voice receive control traffic for different worst-case jitter of the voice data traffic**

## 4. Implementation Model of RNOS

The implementation part of the middleware platform provides the necessary infrastructure and application programming interface to implement applications that have been modeled using the analysis model. The approach is characterized by two properties:

1. The middleware 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 trees and task paths, and the input model, i.e. flows and microflows.

In addition to the application, input and resource models described in section 3, two further concepts are provided to build the implementation model. One is the path-thread, which links packets (which are part of a flow and belong to a microflow) with a task path. The task path is defined by the microflow the packet belongs to. The second element is the scheduler, which decides which path-thread and therefore which task is executed next.

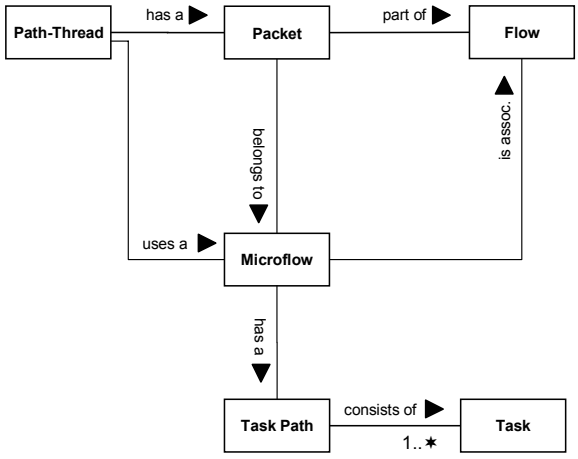


**Figure 12: Object-diagram of implementation model**

### 4.1. Path-Threads

For each packet that is to be processed, a path-thread is created. A path-thread has a task counter, a packet, an associated flow and microflow description and therefore uniquely identifies a task path. The task counter points to the next task that is to be executed. Figure 12 gives the complete picture of the elements of our implementation model.

***Example 4.1*** *In our scenario we have a source path-thread that matches the incoming packet to a microflow. If the microflow is known, the source path-thread will create a new path-thread for the packet of that microflow. Otherwise, it will create a path-thread for the default non-real-time flow. Figure 13 shows the source path-*

*thread and three path-threads that might be created by the source path-thread.*

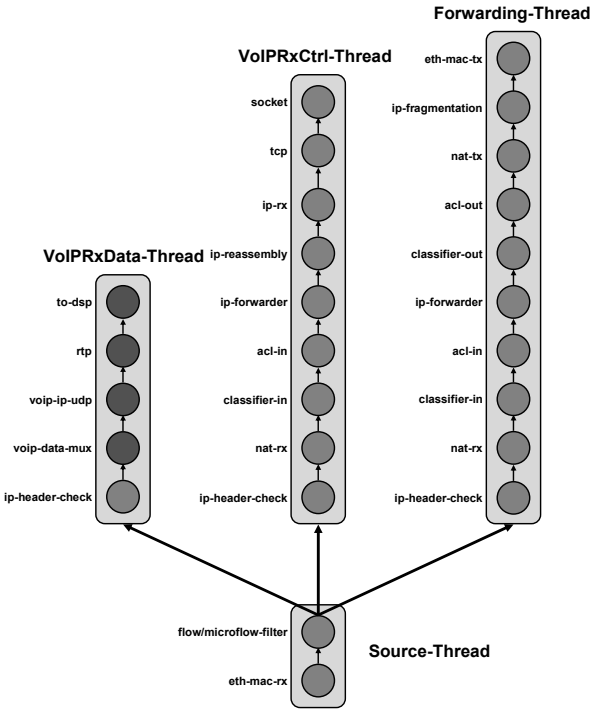


**Figure 13: Source Path-Thread and three other Path-Threads**

## 4.2. Scheduler

The scheduler decides which path-thread runs next. The selected path-thread will execute one task and surrender control back to the scheduler. We use an EDF scheduler to schedule the path-threads. As the tasks are non-preemptive, the granularity of the tasks provides the preemption points for the EDF scheduler. Non-real-time flows have an infinite deadline and therefore get only executed when there is no pending packet of a real-time flow.

## 4.3. Implementation

The RNOS is implemented on top of an existing real-time operating system. It runs in one thread of the RTOS. The only precondition to the RTOS is that it must be capable of providing a predefined amount of processing power to the RNOS. In our implementation, the RNOS gets 6ms of processing power in each interval of 10ms, see Example 3.3. RNOS is implemented in Embedded C++. A complete set of APIs and base classes allow an efficient implementation of networking applications. Figure 14 depicts the elements of the RNOS and that it runs in a thread of the underlying RTOS.

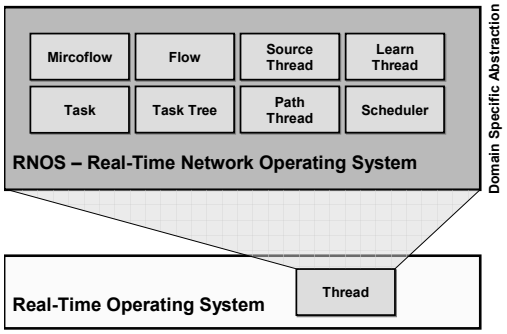


**Figure 14: RNOS runs in a thread of the underlying RTOS**

Programming with RNOS is different to programming with a standard RTOS. Applications are built from a set of existing and new tasks. Since the analysis cannot be completed without having the worst case and best case execution times of all the tasks, the tasks itself have to be programmed first. The RNOS supports the measurement of worst case and best case execution times. The best case execution time is measured while executing the task on an idle system while all the interrupts are disabled and the task has been preloaded into the cache. The worst case execution time is more difficult to determine. It is measured having all caches and interrupts disabled. To that, the worst case execution times of the interrupt service routines are added. Another possibility to get the execution times is to formally analyze the compiled task code, see e.g. [20]. The more tasks that are available, the more RNOS becomes a construction kit to build networking applications. With RNOS, we have path-threads instead of standard threads. The life-time of a path-thread is short, compared to standard threads. It exists only as long as its associated packet has not been processed completely. For each packet there is a path thread. As we are concerned with packet processing only, the path threads appear to be a very natural way of programming. Table 2 gives some analogies of the elements of a traditional RTOS and RNOS.

**Table 2: Analogy of RTOS and RNOS**

| *Real-time Operating System* | *Real-time Network Operating System* |
|---|---|
| Instruction | Task |
| Program | Task Path |
| Thread | Path-Thread |
| Program Counter | Task Counter |
| Thread Priority | Deadline |
| Registers and Memory | Packet and Packet Annotations |

The RNOS is not free of overhead. The creation and termination of path-threads and switching between path-threads reduce the CPU capacity available to the

processing of tasks. The overhead of RNOS has been determined by comparing the throughput of the system dependent on the number of thread-switches. Compared to a hand-coded system without path-threads, an overhead of 1% of the overall execution time for the creation and termination of path-threads and 3% for each switching between path-threads is introduced on our example system. If we allow less preemption points (by combining several tasks to a virtual task), the delay incurred to real-time flows increases and the overhead decreases. The granularity of tasks influences the trade-off between packet delay and available throughput. The investigation of this relationship is subject of further work.

## 5. Measurements & Comparison

We have measured the best effort forwarding rate under worst-case conditions. Figure 15 shows the measurement setup. The SmartBits is a professional network performance analysis system. It produces Ethernet traffic at line rate with 64 byte packets on line A. System 1 forwards this traffic to System 2 which forwards it back to the SmartBits (by line B). This way it is possible to measure throughput and delay. Additionally, we inject voice calls from System 1 to System 2. The scenario represents a worst case system load for our example system. The voice channels do not get distorted (measured with Abacus, a professional telephony test system), i.e. the voice data packets are processed with highest priority, while the traffic generated by the SmartBits represents the worst case load for the example system. Figure 16 shows the measured and calculated best effort traffic that is forwarded depending on the number of active voice channels.

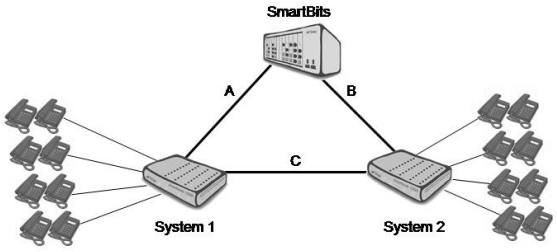


**Figure 15: Measurement Setup**

The measured numbers are 1-5% higher than the calculated numbers. For zero voice channels, the numbers lie within the inaccuracy of the measurement (1%). The larger the number of voice channels, the larger becomes the difference. The cache efficiency is higher with larger number of voice channels, as the voice packets arrive in bursts (voice packets of all channels come immediately behind each other) and therefore a better cache hit ratio occurs.

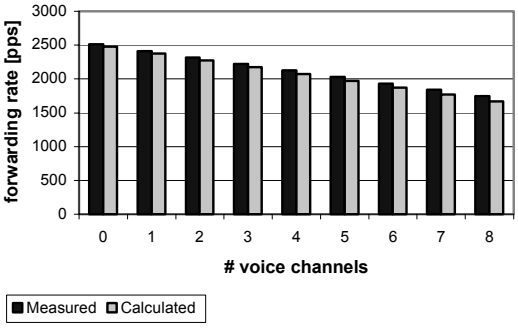


**Figure 16: Measured and Calculated Best-Effort Forwarding-Rate**

## 6. Conclusion and Outlook

The software platform RNOS allows a natural domain specific modeling of applications for single CPU systems. Using the real-time calculus, the system parameters can be explored and weak points of the system can be exposed early in the design cycle. In a second step, the modeled application can be implemented seamlessly on top of RNOS. The results that are obtained with the real system are comparable with the results obtained in the analysis, which was one of the major goals of this work.

There are several open questions that will be investigated further. One issue is related to the granularity of tasks and the corresponding impact on delay, throughput and overhead. A next question is related to the assignment of deadlines. The current implementation adds a static deadline to each packet. At least for multimedia flows we could add deadlines depending on the current state, e.g. a buffer level. The provided calculus needs to be extended in order to cope with this extension. A final issue is the analysis of trade-offs related to the batch-processing in terms of a better cache usage, a smaller overhead and an increased packet delay.

## 7. References

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

[2] Scott Karlin and Larry Peterson, "VERA: An extensible router architecture," *IEEE OPENARCH 01*, Anchorage, AK, USA, April 2001.

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

[4] E. Kohler et al., "The Click Modular Router," *ASM Transactions on Computer Systems*, 18(3), pg. 263-297, August 2000.

[5] David Mosberger and Larry L. Peterson, "Making paths explicit in the Scout operating system," in *Proceedings of the*

*second symposium on Operating systems design and implementation*, October 1996.
[6] X. Qie, A. Bavier, L. Peterson, and S. Karlin, "Scheduling Computations on a Programmable Router," in *Proceedings of the ACM SIGMETRICS 2001 Conference*, June 2001.
[7] Prashanth Pappu and Tilman Wolf, "Scheduling processing resources in programmable routers," in *Proceedings of the Twenty-First IEEE Conference on Computer Communications (INFOCOM)*, New York, NY, USA, June 2002.
[8] L. Thiele, S. Chakraborty, M. Gries, S. Künzli, "Design Space Exploration of Network Processor Architectures," *First Workshop on Network Processors (NP1)* at the 8th International Symposium on High Performance Computer Architecture (HPCA8), Cambridge, MA, USA, February 2002.
[9] Niraj Shah, William Plishker, Kurt Keutzer, "NP-Click: A Programming Model for the Intel IXP1200," *Second Workshop on Network Processors (NP2)* at the 9th International Symposium on High Performance Computer Architecture (HPCA9), Anaheim, CA, USA, February 2003.
[10] B. Goodman, "Internet Telephony and Modem Delay," *IEEE Network*, May 1999.
[11] S. Shenker and J. Wroclawski, "General characterization parameters for integrated service network elements," Request for Comment 2215, Internet Engineering Task Force, September 1997.
[12] Matthias Gries, "Algorithm-Architecture Trade-offs in Network Processor Design," Ph.D. dissertation, ETH Zurich, Switzerland, July 2001.
[13] J. Janssen, Danny De Vleeschauwer and Guido H. Petit,"Delay and Distortion Bounds for Packetized Voice Calls of traditional PSTN Quality," in *Proceedings of the 1st IP-Telephony Workshop (IPTel 2000)*, April 2000.
[14] International Telecommunication Union, "Packet based multimedia communication systems," Recommendation H.323, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, February 1998.
[15] F. Andreasen, B. Foster, "Media Gateway Control Protocol (MGCP) Version 1.0," Request for Comment (Informational) 3435, Internet Engineering Task Force, January 2003.
[16] F. Cuervo, N. Greene, A. Rayhan, C. Huitema, B. Rosen, J. Segers, "Megaco Protocol Version 1.0," Request for Comment (Proposed Standard) 3015, Internet Engineering Task Force, November 2000.
[17] 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.
[18] M. Handley and V. Jacobson, "SDP: session description protocol," Request for Comment (Proposed Standard) 2327, Internet Engineering Task Force, April 1998.
[19] International Telecommunication Union, "Pulse code modulation (PCM) of voice frequencies," Recommendation G.711, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, November 1988.
[20] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm, "The Influence of Processor Architecture on the Design and the Results of WCET Tools," in *Proceedings of IEEE on Real-Time Systems*, 91(7), July 2003.