Towards High-Performance Active Networking*

Lukas Ruf[1], Roman Pletka[2], Pascal Erni[3], Patrick Droz[2], and Bernhard Plattner[1]

[1] Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH)
CH-8092 Zürich/Switzerland
{ruf,plattner}@tik.ee.ethz.ch
[2] IBM Zurich Research Laboratory
Säumerstrasse 4
CH-8803 Rüschlikon/Switzerland
{rap,dro}@zurich.ibm.com
[3] pascal@promethos.org

Abstract. Network processors have been developed to ease the implementation of new network protocols in high-speed routers. Being embedded in network interface cards, they enable extended packet processing at link speed as is required, for instance, for active network nodes. Active network nodes start using network processors for extended packet processing close to the link. The control and configuration of high-performance active network nodes with network processors such that new services can benefit from the additional processing capacity offered is nontrivial since the complexity to configure a node while providing sufficient level of abstraction is hard to master. In this paper, we present PromethOS NP which is a modular and flexible router architecture that provides a framework for dynamic service extension by plugins with integrated support of network processors, namely the IBM PowerNP 4GS3 network processor. We briefly introduce the PowerNP architecture in order to show how our active networking framework maps onto this network processor and provide results from performance measurements. Owing to architectural similarities of network processors, we believe that our considerations are also valid for other network processors.

1 Introduction and Motivation

Network processors (NPs) have been developed to ease the implementation of new networking functionalities and services in high-speed routers [14]. The programmable environments provided by processor manufacturers remove the burden of creating application-specific integrated circuits (ASICs) or other hardware components needed for extended or new services. Hence, NPs combine the high performance known from ASICs with the capability to adapt networking functionalities in software, while not requiring expensive modifications in hardware. Even though not designed for active networking in the first place, we are convinced NPs provide a perfect processing platform for dynamic service deployment and configuration.

* This work is partially sponsored by the Swiss Federal Institute of Technology (ETH) Zürich and the Swiss Federal Office for Education and Science (BBW Grant 99.0533). PromethOS v1 has been developed by ETH as a partner in IST Project FAIN (IST-1999-10561). We would like to acknowledge the great support received from the IBM Zurich Research Laboratory.

N. Wakamiya et al. (Eds.): IWAN 2003, LNCS 2982, pp. 32–43, 2004.

Modern high-performance active network nodes (hANNs) are built by a set of host CPUs and a set of network interface cards (NICs). NICs provide embedded NPs to process packets as close as possible to the network link. Host CPUs and NICs are interconnected by a switching fabric or an node internal bus. A common architecture of NPs is based on a legacy processor core and specialized processing engines on a single chip. Thus, the NPs with these engines and the processor core in conjunction with the host CPUs provide three different processing environments.

Conceptually, hANNs follow a three-level approach to separate management and control plane from packet forwarding [2]. A management plane is required to deploy service specifications and service components as used to setup and configure network-wide services and nodes [1]. In the control plane, service control information is exchanged. For example routing information is distributed. The transport plane provides the functionality to deal with the packets like for example forwarding, content encryption or packet filtering.

A framework that provides a flexible mapping of these three levels to a concrete implementation is essential for the management and control of a hANN. Code portability is important to ease the deployment of service components. At run-time, service components must be installed and interconnected such that the node-local, service-internal communication path can be established easily.

PromethOS NP provides a framework that copes with the complexity of such an hANN. It is based on an extended version of PromethOS [8], a Linux kernel-space-based NodeOS providing the PromethOS EE. The current implementation of PromethOS NP controls an hANN including an Application Reference Board (ARB) that is based on the IBM PowerNP 4GS3 network processor [5].

For the implementation of the PromethOS NP framework it is important how the three levels are mapped to the underlying platform such that the node performance is maximized. Thus, we present the architecture of PromethOS NP and the fundamental design considerations in Section 2. Subsequently, we give a brief introduction to the IBM PowerNP 4GS3 and the ARB in Section 3 and provide further implementation details. Our implementation is then evaluated by performance measurements and the results are presented in Section 4. In Section 5 we review related work, before we summarize and conclude our paper (Section 6).

2 PromethOS NP

The PromethOS NP framework controls an hANN with NICs that provide NPs for extended packet processing. It is composed of management applications and the PromethOS NodeOS as well as the PromethOS EE. Figure 1 provides an overview of the main components of a PromethOS NP node:

- **Management applications:** The management applications control the NodeOS. Further, they initiate component installation and service configuration. They are implemented by the NP Control Daemon (NP CtrlD), the NP Control Client (NP Ctrl).
- **NodeOS:** The PromethOS NodeOS functionality is provided mainly by the PromethOS plugin manager, which is responsible for the creation, configuration and

control of the PromethOS EE. It attaches to the legacy hooks of the IP stack and to the fast-path of the proxy device driver.

– **EE:** The PromethOS EE follows the plugin paradigm, in which plugins are organized as a directed graph of modules.
– **Plugins:** Code components installed in the PromethOS EE are called PromethOS plugins. They provide the service functionality. Every PromethOS plugin is identified by a node unique plugin ID. We make a difference between PromethOS plugins installed on NP cores (specialized processor engines of a NP) and those installed on a general-purpose processor (GPP). PromethOS plugins running on the NP cores are called picoPlugins. In their current implementation, picoPlugins provide packet classification only.

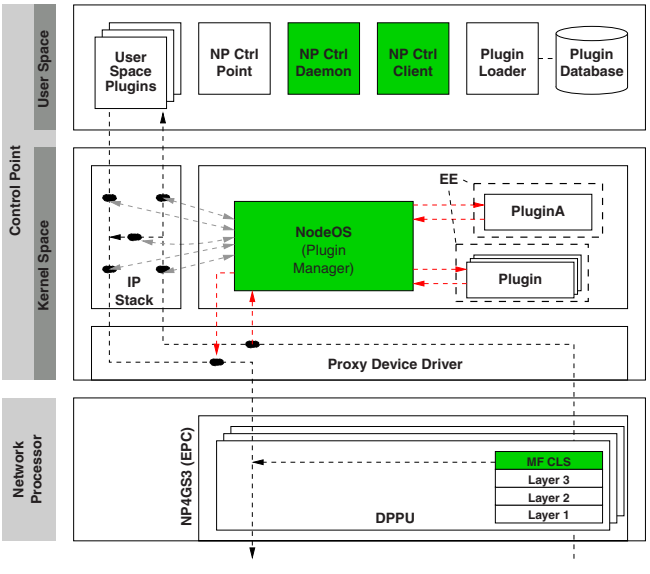


Fig. 1. PromethOS NP: Architectural Overview

A PromethOS NP node spans all processing environments: by design, PromethOS EEs are located on all three levels, thus providing environments for active service components.

PromethOS NP has been extended from traditional PromethOS by the management applications required to control the NP. Further, the fast-path has been introduced that makes benefit of early packet classification: if the picoPlugin is able to demultiplex the packet to the correct service components, packets do not traverse the legacy IP stack of Linux. Like traditional PromethOS, PromethOS NP is registered at the hooks provided by Netfilter [11], too. Thus, it allows for packet reception from the IP stack as well.

The PromethOS EE provides a unified interface to service components irrespective whether PromethOS runs with NP support or not. By the PromethOS NodeOS com-

ponent, the EE and plugin management is decoupled from the underlying hardware. Thus, source code compatibility is provided for different hardware platforms. With NP support, even program code compatibility is provided for the GPP on the NP and the host CPU if they are based on the same processor architecture.

The PromethOS NodeOS component runs in kernel space while the management applications are located in user space. Conceptually of minor concern, it is important to implement components as close as possible to the network link if they are used frequently: The PromethOS NodeOS component dispatches packets to service components. Thus, it is of major importance to avoid overhead where possible. Management processes are carried out infrequently. So, it is perfectly valid to install the management applications in user space where code development is easier due to extended library and debugging support.

2.1 Design Considerations

There are three different approaches to how PromethOS plugins can be implemented on NPs. First, PromethOS plugins can be added in the embedded processor complex (EPC) and run directly on a NP core. This has the advantage that no additional copying of the packet is required. As actions are taken directly in the data plane, the overhead of sending the packet to a control point processor is avoided. On the other hand, the instruction memory can hold 32k picocode instructions shared among all NP cores, which suffices for traditional packet-forwarding tasks and advanced networking functions [3] but limits the size, and therefore the functionality of PromethOS plugins. Although theoretically feasible, picocode or parts of it cannot be dynamically reloaded with the current version of the network processor application services (NPAS). This would require all plugins to be downloaded during the initialization phase, thereby losing the benefit of dynamic code loading of the plugin approach. Running plugins on NP cores eliminates bottlenecks due to external interfaces but might add new ones on the code-execution level: Additional limits can arise owing to the scaled-down RISC architecture of the NP cores (e.g., there is no floating-point support). Even though a C-compiler for the NP cores exists, efficient code is closely linked to the hardware and therefore often written directly in picocode, which lacks code portability. A just-in-time compiler which translates an architecturally neutral programming language into picocode [9] would then be required. A general question is *where* the code will be executed, i.e., on ingress, egress, or both. Active code placed in the data path and executed on NP cores has been evaluated in [9] for a simple active networking language.

Second, the ePPC (embedded PowerPC) in the EPC can be used to run PromethOS plugins. After classification, PromethOS relevant packets are redirected to the CP residing on the ePPC; all other packets in the data path are handled by the NP cores. The former is done by the general PowerPC handler (GPH), an NP core capable of writing the packet into the ePPC's memory and indicate its arrival to the ePPC by means of an interrupt. The packet then traverses the Linux IP stack before being handed to the plugin manager. The plugin identifier found during classification on the NP allows the plugin manager to select the appropriate plugin. Here the advantages are that only PromethOS-relevant packets will be redirected to the ePPC, while the flexibility of the Linux kernel (e.g., Netfilter support) is retained. No additional processor is needed and

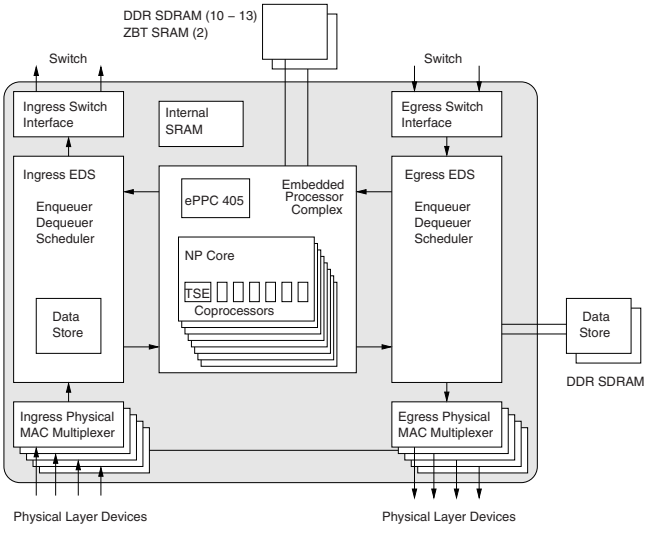


Fig. 2. Main functional blocks of an IBM PowerNP 4GS3.

the approach behaves much like a system-on-a-chip. The approach will eventually encounter performance limitation due to the interface between NP cores and the ePPC. Moreover, the ePPC is clocked at 133 MHz, which might not be enough for extensive plugin processing.

As a third option, the PromethOS plugin manager can run on an Ethernet-attached external CP, usually a GPP. This approach is similar to the previous one, but uses a physical interface and the GMII Gigabit Ethernet-to-PCI-X bridge to copy packet data into the CP memory. Redirection is done by a guided frame handler (GFH) NP core. Processing of plugins is limited by the clock speed of the attached external GPP CP.

Compared with an approach without NPs the benefits are that packet classification is done by the NP, hence reducing packet handling in the Linux IP stack, while normal data packets are directly forwarded by the NP. In this paper we analyze the latter two approaches, where the plugin manager resides on the ePPC or an external CP. Given its limited functionality, the approach with dynamically (re-)loadable picocode plugins is left for future work.

3 The IBM NP4GS3 Network Processor

3.1 The Power NP4GS3 Architecture

The IBM PowerNP 4GS3 is composed of an embedded processor complex (EPC), the enqueuer dequeuer scheduler (EDS) blocks, the switch interfaces, the physical MAC multiplexers, embedded SRAM memory, and additional memory interfaces for external memories. The EDS is responsible for hardware flow control and scheduling while

the MAC multiplexers transfer packets from/to the physical-layer devices. The main functional blocks of a PowerNP are shown in Figure 2.

The EPC consists of 16 packet processor engines called NP cores each supporting two independent threads, a set of eight specialized coprocessors for each NP core, and an embedded PowerPC 405 microprocessor, all running at 133 MHz. The coprocessors perform asynchronous functions such as longest-prefix lookup, full-match lookup, packet classification, hashing (all performed by two tree search engines (TSE) per NP core), data copying, checksum computation, counter management, semaphores, policing, and access to the packet memory. The NP cores are scaled-down RISC processors which execute the so-called picocode. The picocode instruction set is specifically designed for packet processing and forwarding.

Packet processing is divided into two *stages*: Ingress processing directs packets from the physical interface to the switch, egress processing does the reverse. Every NP core can handle both stages, but usually one is associated virtually at dispatch time for convenience. Threads are dispatched upon packet arrival from the physical interface or the switch, or by an interrupt. Each thread has its own independent set of registers, so there is no overhead in switching threads. When a thread stalls (e.g., when waiting for a coprocessor), multi-threading will switch to the other thread if this one is ready for execution. This dynamic thread execution helps to balance the processor load. A thread entirely processes a stage of a packet, which is called run-to-completion mode. Additional context information (e.g., output interface identifier gained from the IP forwarding lookup) can be transferred from ingress to egress along with the packet.

We based our implementation on the Application Reference Board (ARB) from Silicon Software System This board provides a BroadCom PCI-X Ethernet controller (BCM5700) for bridging between the application reference board and the host.

3.2 PromethOS NP Implementation

Figure 3 gives an overview of the architecture for the external CP. In case the CP is internal (running on the ePPC), incoming packets are redirected from the E-EDS to the ePPC directly and outgoing ones vice versa. Administration and configuration of classifier rules are handled by the NP CtrlD and the NP Ctrl. The client allows a user to manage classification rules and plugin IDs similar to `tc` of the Traffic Control [4] package in Linux. The daemon provides an interface to the client process and talks to the NP using the proxy interface to the NPAS from the NP control point. For this, the daemon performs the necessary translation process and maintains counters of rule hits at the same time. The NP CP uses the Proxy Device Driver to encapsulate control traffic from the CP to the NP.

The implementation of PromethOS on the PowerNP is based on the multi-field classifier from the NPAS which provides a CP API and its corresponding picocode part. Depending on the memory size, up to 5192 multi-field classification rules can be stored. The classifier picocode has been enhanced in order to return the plugin ID (later being used by the plugin manager) if a rule matches. A rule match redirects an incoming packet, including the plugin ID found, to the CP for further processing.

While the redirection *decision* is taken on the ingress (i.e. during packet classification), the redirection *action* occurs at the egress. In the case of an attached external CP,

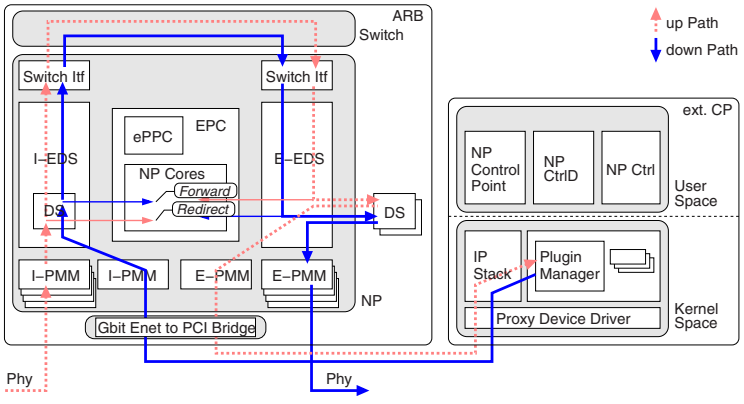


Fig. 3. Data path of packets handled by the external CP.

the packet is sent to the physical interface and then traverses the Ethernet-to-PCI bridge to reach the CP. As the plugin ID is already known, the packet will not traverse the full Linux IP stack, but is handed directly to the plugin manager by the proxy device driver (fast-path). After processing, the plugin manager sends the packet back to the NP. It will again traverse the ingress side of the NP, but this time the forwarding decision is taken. Next it traverses the switch and the egress side of the NP as a normal IP packet does. In the case of an internal CP, the GPH sends the packet directly to the ePPC, where it will be handled, and receives it back afterwards for forwarding on the egress.

3.3 Performance Characteristics

The following list mentions the performance characteristics of the PowerNP that play a major role for all PromethOS NP configurations, as discussed in Section 2.1.

- **Data Mover Units:** The PowerNP provides five data mover units (DMU). Each DMU moves data at 1 gigabit per second (Gbps) in both ingress and egress directions. Four of them can be configured independently (e.g., as an Ethernet medium access control (MAC)). The fifth pair is directly inter-connected to move data from the egress to the ingress side of the NP4GS3.
- **Ethernet:** Three DMUs are configured as 1000Base-T GMII Ethernet ports. The fourth establishes the connection to the attached external GPP by means of a GMII gigabit Ethernet-to-PCI-X bridge.
- **Switch interface:** The switch interface consists of two data-aligned synchronous link (DASL) interfaces in each direction. Each of them provides a transfer rate between 3.25 and 4 Gbps surpassing the accumulated bandwidth of the four gigabit Ethernet interfaces [5]. These interfaces can either connect an NP to a switch fabric, to another network processor, or directly transfer the data from the ingress to the egress interface. Thus, this interface will not cause any performance degradation.

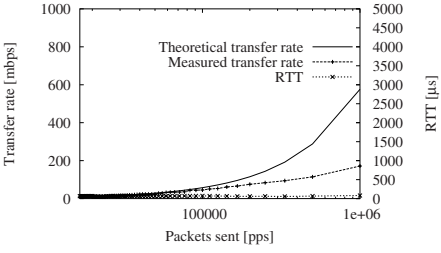
- **Data store coprocessor:** Data are copied into or from the EPC by the data store coprocessor of the NP. The packet throughput depends linearly on the number of bytes copied per packet: Usually only 64 bytes are copied, as this is sufficient for header inspection. The PowerNP achieves 4.80 Gbps of aggregated throughput of Internet-like traffic when doing layer 3 packet forwarding [6]. Depending on the PromethOS configuration, data packets traverse each stage up to two times. Because PromethOS requires additional layer 4 classification we except that the PowerNP can provide up to 1.5 Gbps throughput.
- **PCI bus:** The ARB can be integrated into an hANN using its Ethernet-to-PCI bridge. The BroadCom PCI-X Ethernet Controller BCM5700 permits bridging at 1 Gbps full duplex. The PCI standard v2.3 defines the following bus transfer rates: 1.1 Gbps for an interface with 32 bits width running at 33 MHz (32b/33MHz), 4.3 Gbps (64b/66MHz), and 8.5 Gbps (64b/133MHz PCI-X 1.0). However, the PCI bus does not provide full duplex. So, if the ARB were placed in a 32b/33MHz PCI system, we could expect a throughput of at most 0.55 Gbps (provided the bus is not used by other devices). Thus, at least 2 Gbps are required from the PCI bus bandwidth to satisfy the ARB.
- **General PowerPC Handler:** The ePPC is connected to the general PowerPC handler (GPH), a NP core with extended capabilities, via shared memory for data transmission. The GPH copies data packets into the external DRAM, and signals this to the ePPC by an interrupt. Thus, it provides functionality comparable to a programmable DMA controller. The reverse process is carried out if the ePPC sends a packet. Passing packets to the ePPC has been designed for the control path, hence we cannot expect high throughput for data-path applications. However, it can be extremely valuable to offload complex data-path processing as encountered in active networks in order to prevent packet redirection to an external CP, as long as the rate is bounded to an acceptable value. As it is difficult to estimate the performance of this interface, we provide empirical results in section 4.1.

We conclude from this analysis first that the PowerNP should be powerful enough to carry out packet classification for PromethOS plugins on the one hand, and, on the other hand, to forward packets of other streams at link speed (1 Gbps) simultaneously. Second, the PowerNP has no performance bottlenecks if PromethOS NP is run on an external CP. However, in the case where PromethOS NP and the PromethOS plugins are run on the ePPC directly, we presume performance limitations since the PowerNP was originally not designed for this configuration of transport plane packet handling. This has to be taken into account by the configuration process of PromethOS NP.

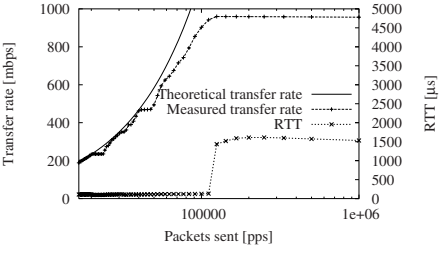
4 Evaluation

4.1 Performance Measurements

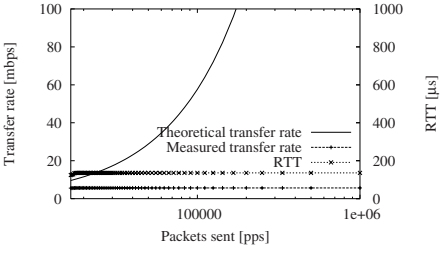
Following the analysis of all interfaces involved (cf. section 3), we base our evaluation on an hANN with an Intel Xeon 2.4 GHz processor running Linux 2.4.18 in which the ARB is installed. The ARB operates at 64b/66MHz PCI speed.



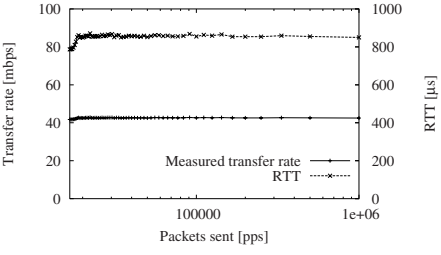
(a) 72B/packet, PromethOS/host CPU



(b) 1460B/packet, PromethOS/host CPU



(c) 72B/packet, PromethOS/ePPC



(d) 1460B/packet, PromethOS/ePPC

Fig. 4. PromethOS NP on the host CPU (a,b) and on the ePPC (c,d) – Transfer Rate and Round Trip Time: (a,c) 72 Bytes per packet; (b,d) 1460 Bytes per packet.

We measured the performance of the hANN without real service functionality of the plugins because otherwise throughput would additionally depend on the service complexity rather than on the efficiency of the framework. Packets were sent by a traffic generator (source) to the plugin manager (sink), whereby the plugin manager acts as source and sink at the same time for convenience. Packets were sent out by one Ethernet interface and received on another via crossed cables. The up and down paths taken by packets traversing a hANN are visualized in Figure 3.

Latency, throughput and packet loss have been measured in two configurations: In the first case PromethOS NP was running on the Ethernet-attached external CP, in the second case it was placed on the CP running on the ePPC. The results are for different packets sizes, namely, 72 and 1460 Bytes. We chose these packet sizes since we assume the former to be the size of signalling control packets approximately while the latter corresponds to usual data packets. In Figure 4 (a) and (b), we plot the results of the first configuration in which the NP cores are only used for packet classification. The measurement results achieved for the second configuration are shown in Figure 4 (c) and (d). The x-axis (number of packets per second) is plotted with a logarithmic scale. The packet size corresponds to the number of bytes sent at the Ethernet interface, omitting the internal header (36 Bytes) added by the Linux proxy device driver for signaling. In Figure 4, the transfer rate (TR) is shown in megabits per second (Mbps), the round trip

time (RTT) in units of microseconds (μs), and the packet transfer rate in units of packets per second (pps). For comparison, we also plot the ideal transfer rate, where the number of packets attempted to send corresponds to the number of packets received, assuming all transmission attempts are successful.

Table 1. Comparison of transfer rates and round trip times

PromethOS NP on the host CPU:

72 Bytes per packet			1460 Bytes per packet		
TR (pps)	TR (Mbps)	RTT (μs)	TR (pps)	TR (Mbps)	RTT (μs)
297985	171.639	81.2	81846	955.966	1531.4
20134	11.597	48.7	20110	234.879	96.2

PromethOS NP on the ePPC:

72 Bytes per packet			1460 Bytes per packet		
TR (pps)	TR (Mbps)	RTT (μs)	TR (pps)	TR (Mbps)	RTT (μs)
9807	5.649	135.9	3638	42.497	849.8
9640	5.553	124.3	3574	41.471	786.6

In Table 1, we compare the maximum throughput, the maximum transfer rate, and the minimum round trip time for both configurations. The increase in latency found in Figure 4 (b) corresponds to the default queue-threshold configuration of the PowerNP. We note the difference in performance between the two configurations. We further investigated the second configuration: First, we measured the performance of Linux with regard to its capacity of creating, sending and receiving socket buffers without real transmission, i.e. the socket buffers are not flattened and then sent out at the physical interface, but the receive-function is called directly. We achieved a transfer rate of 697.39 Mbps. Second, we measured the performance of the interface between the NP cores and the ePPC by transferring full-sized packets (1460 Bytes) via the shared memory and interrupt signaling back and forth. We were able to measure a transfer rate of 298.04 Mbps[1].

From the measurement results, we conclude: The first configuration provides sufficient performance to handle at least one gigabit link. Measurements of the PowerNP proved that the PowerNP is still capable of carrying out packet forwarding for an additional, non-active 1 Gbps flow. Measurement results in the second configuration lead to the conclusion that the Linux/PromethOS on the ePPC should not be used for transport plane packet handling. However the extensible platform provides a very useful environment for control plane functionality where less packet processing is expected.

5 Related Work

VERA [7] provides a three-level router architecture to provide a modularized, standards compliant router. It is implemented by a device driver that interfaces to the IXP1200

[1] Note that we did not vary the internal socket-buffer limits imposed by Linux which can further improve our results.

and, thus, provides the hardware abstraction. In [13] resource allocation and scheduling issues are analyzed on a three-level processor hierarchy, and evaluates the performance of the Intel IXP 1200 for vanilla IP packets. In [10], an IXP1200-based network interface card offering four 100T ports was evaluated. On the IXP1200 StrongARM core, Linux is run, but used for initialization and debugging purposes only; processing is carried out in the so-called kernels run on the microEngines of the IXP, while the host CPU is used for extended processing. A very interesting approach to datapath packet processing is provided in [12] where the performance of a Click-based NP software architecture is evaluated. The Active Packet Editing (APE) approach [15] is a two-level active networking architecture that consists of an active packet processor in software running on a GPP and a packet editor based on an FPGA with content-addressable memory (CAM) for efficiency. The packet processor configures the packet editor, which performs packet classification and simple packet-modification tasks through active packets. Their packet editor prototype achieves slightly less than 1 Gbps of throughput for simple IP header modifications and the packet processor is capable of handling 10 Mbps of small-sized packets. With the PromethOS NP framework, we focus on run-time extensibility and mapping flexibility of active service components. The unified interface provided by the PromethOS EE allows for the portability of service components. By the PromethOS NodeOS component, the required abstraction is provided such that the service can benefit most from the underlying hardware irrespective whether NP-based or just legacy NICs are available on an hANN. With PromethOS NP running on the ePPC, an active platform for control plane functionality is provided thus allowing for greater scalability of the node since not all control plane traffic must be forwarded to the host CPU.

6 Summary, Conclusion, and Outlook

In this paper, we introduced PromethOS NP, a framework that eases the use of network processors for high-performance active network nodes The framework provides extended NodeOS functionality that supports plugin portability by the PromethOS EE across different node configurations. The presented implementation is based on an hANN supported by network interface cards with an embedded IBM PowerNP 4GS3 network processor. It is run either on the host CPU (Ethernet-attached external control point) or on the embedded general-purpose processor of the network processor. In both configurations, the NP cores provide packet classification for the fast-path to circumvent legacy packet classification by the network stack of the operating system.

Our performance measurements prove the efficiency of our architecture. PromethOS NP supported by the PowerNP was able to handle gigabit link speed ($\sim$956 Mbps); 297,985 packets per second could be processed without any optimization of legacy Linux. In addition, when PromethOS NP was run on the Ethernet-attached external control point (host CPU), the PowerNP provided ample capacity for additional flow-processing. In the configuration with PromethOS NP run on the embedded PowerPC (ePPC), measurement results favour the use of the extensible environment for control plane functionality but not for transport plane packet processing. The PowerNP, whose ePPC is designed for control plane functionalities and exceptional data-plane packet

processing, supports, thus, node scalability with regard to control of multiple, concurrent transport plane services.

We are convinced that PromethOS NP in conjunction with the IBM PowerNP 4GS3 provides a flexible and efficient architecture and platform for active services that need to process packets at link-speed. Currently, we are investigating the extended use of the NP cores as well as optimizations of a NodeOS running on the host processor as well as on the PowerNP creating a multiprocessor high-performance active node.

References

[1] M. Bossardt, L. Ruf, R. Stadler, and B. Plattner. A service deployment architecture for heterogeneous active network nodes. In *IFIP International Conference on Intelligence in Networks (SmartNet)*, April 2002.

[2] The FAIN Consortium. *D7: Final Active Network Architecture and Design*, 2003.

[3] R. Haas, C. Jeffries, L. Kencl, A. Kind, B. Metzler, R. Pletka, M. Waldvogel, L. Freléchoux, and P. Droz. Creating advanced functions on network processors: Experience and perspectives. *IEEE Network*, 17(4), July 2003.

[4] B. Hubert et al. Linux Advanced Routing & Traffic Control. `http://lartc.org`, 2003.

[5] IBM Corp. IBM PowerNP NP4GS3 databook. `http://www.ibm.com`, 2002.

[6] IBM Corp. LinleyBench 2002 test results, IBM PowerNP NP4GS3. `http://www.chips.ibm.com/techlib`, 2002.

[7] S. Karlin and L. Peterson. VERA: An extensible router architecture. In *Proceedings of the 4th International Conference on Open Architectures and Network Programming (OPENARCH)*, pages 3–14, April 2001.

[8] R. Keller, L. Ruf, A. Guindehi, and B. Plattner. PromethOS: A dynamically extensible router architecture supporting explicit routing. In *Proceedings of the Fourth Annual International Working Conference on Active Networks IWAN*, December 2002.

[9] A. Kind, R. Pletka, and B. Stiller. The potential of just-in-time compilation in active networks based on network processors. In *Proceedings of IEEE OPENARCH*, pages 79–90, June 2002.

[10] K. Mackenzie, W. Shi, A. McDonald, and I Ganev. An Intel IXP1200-based network interface. In *Proceedings of the Workshop on Novel Uses of System Area Networks at HPCA (SAN-2 2003)*, 2003.

[11] P. R. Russell. The NetFilter Project. http://www.netfilter.org, 2003.

[12] N. Shah, W. Plishker, and K. Keutzer. NP-Click: A programming model for the Intel IXP1200. In *Proceedings of 9th International Symposium on High Performance Computer Architectures (HPCA), 2nd Workshop on Network Processors*, February 2003.

[13] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a robust software-based router using network processors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 216–229, October 2001.

[14] J.P.G. Sterbenz. Intelligence in Future Broadband Networks: Challenges and Opportunities in High-Speed Active Networking. In *Proceedings of IEEE International Zürich Seminar on Broadband Communications (IZS 2002)*, Feb. 2002.

[15] N. Takahashi, T. Miyazaki, and T. Murooka. APE: Fast and secure active networking architecture for active packet editing. In *Proceedings of IEEE OPENARCH '02*, pages 104–113, June 2002.