A Scalable High-performance Router Platform Supporting Dynamic Service Extensibility On Network and Host Processors

Lukas Ruf, Ralph Keller and Bernhard Plattner
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH)
CH-8092 Zürich/Switzerland
Email: {ruf,keller,plattner}@tik.ee.ethz.ch

***Abstract*— Emerging network services such as transcoding and encryption need application-specific handling of data streams within the network, thus requiring enormous computational capabilities on routers to process packets at link speed. Recently appeared Network Processors (NPs) are able to significantly increase the available processing capacities on a router by a chip-multi-processor architecture. Embedded within the network interface card, NPs provide several code-extensible processors with different capabilities located at various layers. However, the increase in processing capacity comes at the cost of a higher complexity to program and control various processor hierarchies provided on the router.**

In this paper, we introduce the model of active network nodes built of a processor hierarchy together with a component-based service model. We present the architecture of PromethOS NP which is a modular framework that manages and controls a network node with a multitude of processors in a scalable way. Specifically, we describe the mechanisms required to dynamically configure multiple processors organized in a hierarchical fashion such that they provide a new network service on behalf of applications. As a proof of concept, we have implemented a service framework for PromethOS NP. Our implementation is based on a network interface card with an embedded IBM PowerNP 4GS3 and an Intel Xeon processor, offering programmability on a three-tier hierarchy of processors.

I. Introduction and Motivation

For the network of the decades ahead, there is a strong demand for services beyond what is required for simple routing and data transport. Emerging services such as application-dependent firewalls, explicit congestion adaptation, media gateways, network address translation, intrusion detection, secure communication, and traffic engineering all require a network that not only forwards packets based on the destination address, but also performs packet processing on nodes interior to the network.

To foster this fundamental transformation, the research community has proposed active networking [1] that envisions a programmable network infrastructure that is extensible at run-time to accommodate the rapid evolution of protocols and services demanded by applications. In this context, services can be defined as application functions provided on network elements as opposed to traditional services implemented on end systems only. Thus routers are required to perform additional functions such as inspecting transiting packets, altering the forwarding and queuing behavior, or even modifying the packet content. Such additional functionality paves the way to connect the emerging field of pervasive computing with the infrastructure of the Internet where gateways provide the adaptation mechanisms that allow devices with low communication bandwidth and little processing capacity to access information originally designed for powerful computers.

To handle the growing transmission capabilities of links, router architectures are required to be highly scalable in order to support an increasing number of gigabit links. Recently, commercially available network processors (NPs) [2]–[4] have appeared on the market. Network processors are software programmable devices specialized for the networking application domain, with architectural features such as a fast I/O path and an instruction set specifically tailored for high speed packet processing (for example dedicated instructions for IP prefix matching). To overcome the performance limitations of traditional single processor systems, network processors often include multiple processor cores with integrated cache and memory on a single chip. Such processing clusters exploit the inherent independence among different traffic flows by processing packets concurrently. This parallel utilization of processing cores boosts computational capabilities and allows to arbitrarily scale processing if needed. Typically, NPs are built of a set of first-level packet processors and a smaller set of second-level control processors[1].

In an effort to support application-specific packet processing, router manufacturers have started to embed programmable elements into routers for providing network service functionality in a more flexible way. For example, current high-performance routers often include large sets of network interfaces with embedded network processors (so-called NP-

[1]NP vendors do not use a consistent naming scheme to refer to the code-extensible processors: the Intel IXP-architecture refers to the first-level processors as *microengines* while the IBM PowerNP identifies them as *picoprocessors* or *core language processors*. Second-level processors are named differently as well. For this reason, we refer to the first level of processing engines as *packet processors* and to those of the second level as *control processors*.

0-7803-8577-2/04/$20.00 ©2004 IEEE

blades). NP-blades are controlled and managed by a set of third-level *management processors* that are typically based on a legacy general purpose processor design such as the Intel ia32 [5].

While technology advances have demonstrated that network processors with high processing capabilities have become reality, the challenge remains in finding mechanisms that can coordinate and share the various processors offered on different levels in a way users can program an active node according to their specific service requirements. Clearly, the management and control of a programmable node are complex tasks since various architectural and performance constraints need to be considered when configuring a new service. Thus, extensible router platforms that allow for the dynamic installation, configuration and execution of new service functionality need a flexible node architecture as well as resource control mechanisms such that new services can benefit from the various capabilities offered.

In this paper, we introduce an extended model of a high-performance active network node, and briefly present the model of a service that is composed of service components. We come up with a flexible and scalable architecture of a code-extensible router platform that provides management and control of a heterogeneous active network node. The proposed PromethOS NP [6] framework is based on an extended version of PromethOS [7], which is a modular and extensible router architecture that is based on Linux and supports the concept of kernel-loadable router plugins [8]. Unlike the initial single processor-based model, PromethOS NP extends the processing model to a multitude of processors which are organized in a hierarchical fashion. By design, our node architecture is much more scalable and can sustain even very high packet processing rates as demonstrated in its evaluation.

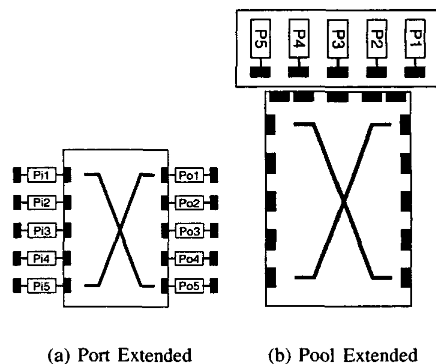
The remainder of this paper is structured as follows: In section II we introduce the node, service and component model used to design and implement the architecture of the PromethOS NP framework. We illustrate the modular architecture of the PromethOS NP framework in section III, and present evaluation results of our proof of concept implementation that is running on a legacy computer with an installed Application Reference Board (ARB) embedding one IBM PowerNP 4GS3 network processor [4]. In section IV, we discuss our approach related to previous work and conclude this paper by a brief summary and outlook in section V.

II. Models

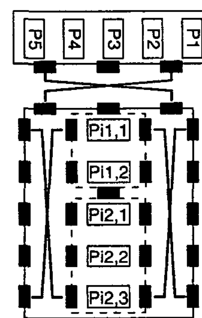
Our work is based on three different models: first, the *node model* that describes a hierarchical high-performance active network node, second, the *service component model* that describes a single function, and third, a *service model* that is used to describe a full service. These models provide the basis to create the different specifications required such as to be able to dynamically install and execute service components on the available processing elements.

A. Node Model

Generally, a router or a network switch is modelled as a set of input-ports linked with a multistage interconnection network or a crossbar with a set of output-ports. Incoming packets are first classified before they are forwarded to the output ports[2]. At the egress side, packets are dispatched to the outbound ports and transmitted on the link.



(a) Port Extended (b) Pool Extended



(c) Hierarchy Extended

Fig. 1. Active Router Models

As visualized in Fig. 1, active networking extends this basic router model by additional processing capabilities on which services can carry out extended functions. Two fundamentally different approaches have existed so far for the provisioning of additional processing capacity: One approach is to add dedicated processing elements to both the input and output ports (Fig. 1(a)). Every link interface has embedded programmable processors with memory to store code and application-specific flow state. There is a clear one-to-one mapping between ports and processing elements and the route through the switch fabric determines the processing element where a specific function needs to be executed. An example of this architecture is the Smart Port Card [9]. An alternative approach is to connect a pool of dedicated computing modules to the router and use them as a pool for processing capabilities

[2]In the case of an IP-router, the time-to-live counter must be decremented and the checksum re-calculated if the packet is not dropped

(Fig. 1(b)). Packets requiring active processing are routed to a processor from the pool, and from there to the appropriate output port. Thus computation of active flows can be evenly distributed over the processing engines that are available. Scalability of this approach is guaranteed through the ability of configuring any number of processing elements. An example of this architecture is the Multi-Gigabit Router [10]. This approach requires a distributed load sharing algorithm which dynamically distributes active flows over the processing elements. We refer to the models sketched in Fig. 1(a)) as the *port extended*, and to the one of Fig. 1(b)) as the *pool extended* model.

Hierarchical router architectures as envisioned for scalable high-performance active network nodes, cannot be modelled satisfactorily by either model due to the hierarchy of processing elements and their dynamic assignability to network ports. While the hierarchy of processing elements can be described by a composition of both, this superposition does not totally satisfy our requirements, since the expressiveness of dynamic assignment of hierarchical clusters of processing elements to ports is limited. Therefore, we propose the *hierarchy extended* model as visualized in Fig. 1(c)) that provides, first, the expressiveness to model hierarchies of processing elements, and second, to assign processing elements dynamically to ports.[3] The port extended as well as the pool extended models can be described by this model by statically assigning the respective processing elements.

The communication mechanisms of the hierarchy extended model allow for paths where paths lead from ports directly to the different hierarchy tiers, or where paths follow strictly the order of the hierarchy, or a mixture of both. Our model provides furthermore the expressiveness to describe the direct communication among peers.

B. Service Component Model

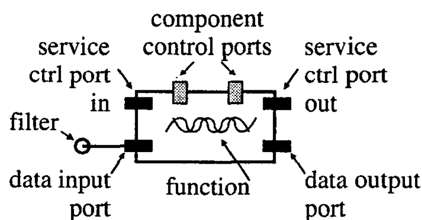


Fig. 2. Service Component Model

Fig. 2 visualizes the model of a single service component. A service component describes a single function according to the plugin model [8]. In addition to the data ports, it provides one service control input port and one service control output port that are used to hook the component to a service control bus on which exceptions can be signalled (see below). To the data input port, a filter is specified to describe the data traffic that requires processing by the component (cf. section III-A). Optionally, it may export one component control input and one component control output port that are used to export an interface by which control information can be exchanged with components that do not make part of the data forwarding path; for example statistical data on a flow processed by the component could be gathered via component control ports, or keys to encrypt/decrypt could be specified if the plugin provides the appropriate functionality.

[3]For clarity reason, we visualize only two tiers of the processor hierarchy.

C. Service Model

Services are modelled as a graph of individual service components where the edges represent service components and the vertices provide the connection between them. Hence, a service is composed of the service components which provide a single function each. At vertices, fork and join semantics are supported. A service control bus between vertices provides the signalling capability required among vertices and service components. For example, such signals could provide the information from a service component to the subsequent vertex that it has discarded the packet, or from the terminating vertex to the initiating vertex that it has received the packet and thus the service component has finished processing. While the complete service is referred to as a *service graph*, we name the service component with its adjacent vertices a *service chain*.

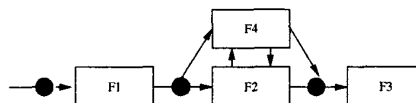


Fig. 3. Service Model

Fig. 3 visualizes a service where four different service components *F1, F2, F3, F4* are inter-connected with one fork and one join operation. The vertices represent the filters specified by the service components. Thus, the graph starts always with a vertex. We do not present the service control bus for clarity reasons. Although service component architecture specific, we mention that the service model supports control and data plane [11] semantic per component. Hence it is possible to have a control relationship between components as visualized by the vertical connections between components *F2* and *F4*.

D. Programmable Distributor Model

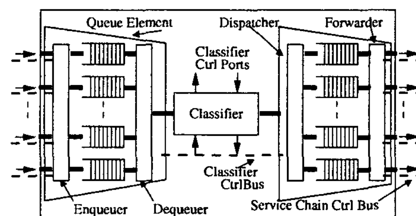


Fig. 4. Programmable Distributor Model

Fig. 4 visualizes the model of our programmable demultiplexer which we name *programmable distributor* due to the

fact that it not only demultiplexes packets but distributes (i.e. dispatches) them to selected egress ports. A programmable distributor follows the same ideas of a service chain whereas vertices provide the two queueing elements. A service chain control bus is terminated at the ingress side and initiated at the egress side of these vertices thus limiting the propagation of signals to single service chains. Our programmable distributor model consists of the following components of which the description is grouped according to the functional elements.

- An ingress queueing element that consists of an enqueuer, multiple queues and a dequeuer. The enqueuer provides the functionality required to receive packets and to handle the service chain control bus signals properly as visualized by the respective dashed lines. The dequeuer serves the packet classifier. Various dequeuing strategies can be implemented if needed.
- A classifier component that is modelled by the service component model as introduced above. Except that no filter is evaluated at its ingress data port, it sticks to the description of a common service component but must adhere to the imposed demultiplexing semantic. By the classifier control bus, the classifier component signals demultiplexing decisions to the dispatcher. The classifier control ports are used to configure and change classification rules.
- An egress queueing element that consists of a dispatcher, one queue per output port and a forwarder. The dispatcher inserts packets into the queues according to the demultiplexing decision by the classifier, and signals the reception of the packet back to the dequeuer. The forwarder element delivers packets to the ports that are ready to process a next packet.

By this model, we describe the functionality required for the join and fork semantic that our service graph model supports.

III. Building Blocks

For the near future, we assume that there will be *source-code portability* among all code-extensible processing elements of one tier, i.e. there will be abstractions that allow the implementation of a service component once and have it compiled for any processing element of the same tier. Further, we assume that there will be processing environments that can be instantiated on groups of tiers. For example, the PromethOS NP processing environment [6] can be instantiated on both control and management processors. We assume also that the processing resources of each tier increase continuously such that it is worth installing code components dynamically in appropriate processing environments on every tier; the enhancements provided by the Intel IXP2xxx series of NPs [3] compared to their predecessors, IXP12xx NPs [2], give evidence to these assumptions.

Our hierarchy extended active router model (cf. section II) describes a node that supports code-extensibility on all tiers and allows the dynamic allocation of processing elements on sets of output ports. To manage and control such a node, we propose the PromethOS NP framework that provides the mechanisms to install code components on all processing elements dynamically if the processing elements support updating code at run-time. The proof of concept implementation has been mapped on an instance of our router model that represents a pyramid of processors consisting of three-tiers, i.e. the node is built of a management (aka. host) processor and NP-blades that consist of one control processor and a set of packet processors each. Before presenting the node architecture that consist of *node management* components and *processing environments* with their controlling components, we introduce the service parts that are installed, instantiated and configured within a processing environment.

A. Service Component

A service component is described by a *service component specification* (CompSpec). The CompSpec specifies the processing logic either as a reference to source code or as a reference to a binary component. In case it is specified as source code, our framework automatically detects for which type of processing environment it has been implemented, and whether the component exports control ports or not. Based on this information, our framework creates the appropriate binary components. In the other case, meta-information is required that identifies the processing environment and provides the information on control ports. The CompSpec specifies further the resource limits to be provided minimally and maximally, and the filter specification which represents either a classification using a seven tuple or a reference to a classifier component that provides service-specific knowledge to demultiplex packages. The seven tuple consists of the input and output network ports, source and destination addresses and ports as well as protocol identification; wildcards are supported, thus creating the semantic of a cut-through channel between two adjacent service components. The output port must be specified only for service components that must be placed onto processing engines that are statically bound to the appropriate egress port of the node. Network input ports can be used for classification purposes at every ingress port of a component but are mandatory only if the respective service component must be placed on specific ingress processing engines.

B. Programmable Distributor

Programmable distributors provide the demultiplexing functionality as introduced above. For code-extensibility they make use of the service chain control bus and a selector functionality per output port. Packets are dispatched only to service chains that are idle, i.e. that do not process a packet currently. Since the initiating programmable distributor is not aware when the terminating distributor receives a packet, the initiator expects a signal from the terminator upon reception of either the packet or a discard message from the service component. To avoid inconsistencies service chains should be replaced only when the respective service chain is idle which implies a request-reply protocol between the controlling component and the programmable distributor to safely replace service

chains. Based on this signaling mechanism, the programmable distributors implement profiling and calibrating procedures to adjust service chain scheduling to the specified resource limits.

C. Service

A service is described by a *service specification* (ServSpec). The ServSpec specifies a graph of service components by a declarative language in which the service components are described by the respective CompSpecs. Further, the ServSpec contains the specification of the resource limits a service requires minimally and must be limited to consume maximally. If control relationships are required, they are specified in the ServSpec, too. Since we follow a unified service model to specify services that can span all tiers, the concept of service specific classifiers paves the way to smoothly introduce, for example, completely new network layer protocols that run in parallel to legacy IP [12], [13].

D. Node Architecture Overview

We present our node architecture in Fig. 5 as it has been implemented to prove our concepts on a three-tier hierarchical router that is built of a host processor (Intel Xeon) and an NP-blade (ARB) that consists of a control processor and a set of packet processors.

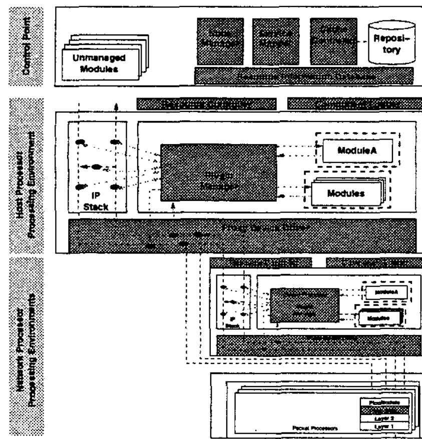


Fig. 5. Node Architecture

Fundamentally, the PromethOS NP node is made up of two different classes of building blocks: those that provide the node management components and those that provide the run-time environments. While the node management components may reside on any computer that is connected to the active network node, the run-time control components are strictly bound to the processing environments. The instances of processing environments are strictly bound to specific processing elements. We introduce the node management components next and focus then on the processing environments where we present the implementation specific aspects required to implement the service model sketched above.

E. Node Management Components

The PromethOS NP node is primarily managed by three node management components, the *Node Manager*, the *Cache Controller* and the *Service Mapper* that share a *Resource Information Database*:

- The Resource Information Database (RID) contains the following elements: the node specification, the service specifications and the specifications of the service components. Along with these specifications, the RID disposes of the description of the average workload per each of these elements.
- The Cache Controller is responsible for managing the repository of binary components. If the CompSpec specifies a component as source code, the Cache Controller creates the various binary components as required for each of the appropriate processing environment instances. We assume the components (either source or binary) are provided to the Cache Controller in a magic way. Hence, we do not deal with the problems involved with network-wide distribution of components although we support a basic fetch mechanism that provides the instruments to retrieve components from a remote repository.
- The Service Mapper is responsible for the creation of a *map specification* (MapSpec) that describes the exact mapping of service chains to processing environments. To create such a MapSpec, the service mapper processes the different specifications and workload descriptors as stored within the RID, and contacts the Cache Controller for every service component specified within the ServSpec. The ServSpec may contain several fork and join operations. It is mandatory for the Service Mapper to insert appropriate programmable distributors at these vertices. Wherever cut-through filters are specified, the Service Mapper disposes of the liberty to re-arrange the vertices such as to improve the distribution of the overall router workload among the processing elements and communication paths available. This re-arranging of vertices provides the freedom to concatenate service components as well as to insert the programmable distributors required to allocate resources for two adjacent service chains on different processing elements.
- The Node Manager is responsible for the whole node. It accepts ServSpecs, hands them over to the Service Mapper, and expects a MapSpec in return. It distributes then the different service chains to the identified processing environments together with the appropriate binary service and classifier components, as well as the configuration information required to inter-connect service chains within one or across various processing environments. Hence the node manager administrates the node unique identifiers for the instances of service chains (ChainID) as well as for the instances of service components (ModuleID). The Node Manger is responsible for keeping the RID up-to-date. To get hold of the required workload information, it queries the resource controllers of the different processing

environments which deliver the aggregated information regarding the actual workload of the processing element as well as of each service chain instantiated within the processing environment. Simple policies are implemented within the Node Manager to decide on how to handle exceptions if service chains violate the resource limits specified.

F. Processing Environments

The PromethOS NP node supports two different types of service extensible processing environments (PEs), where service chains can be installed and instantiated. One of the processing environments provides an extended PromethOS EE for general purpose processor cores. The other one establishes a run-time environment on the packet processors. While the former provides the mechanisms to dynamically extend service functionality in the Linux Kernel, and to instantiate a service component several times, the latter is sticked to the capabilities provided by the packet processors which provide usually no virtualization of the address space, i.e. they support a one-to-one mapping of installed and executed code only. To relieve this limitations, we provide the concept of programmable distributors on the packet processors as well such as to pave the way for dynamic code-updates.

Each processing environment consists of the following elements to provide the run-time environment for our services: a *module loader*, a *resource controller*, and a *plugin manager* that provides the implementation of the programmable distributors in the case of the extended PromethOS EE. The resource controller and the module loader provide the following function:

- The Component Loader is responsible for the loading of binary components into the processing environment. The loading process is triggered by the reception of the service chain specification from the Node Manager. So, the Component Loader controls the instantiating and removal of service components within the respective processing environment. We refer to instantiated service components by the term *module*.
- The Resource Controller is responsible for configuring and controlling the programmable distributors. In the configuration phase, the resource controller binds service chains to the appropriate ports of the programmable distributors and configures the respective classifier and dispatcher components. Configuration of new service chains is carried out according to the procedure sketched above (see section III-B). Further, it provides the controlling competency to specify the resource constraints to the respective programmable distributor per service chain, and to query the corresponding profiling results. These results are then propagated to the node manager upon request.

Although not visualized in Fig. 5 for clarity reason, the resource controller and component loader that are responsible for the processing environment on the packet processors are co-located with those of the processing environment on the respective control processor.

Source Code Portability: Packet processors provide fast classification capabilities. These capabilities open up the room for the creation of a fast path besides the usual slow path of the Linux network stack. For the fast path, packets to be dispatched to service chains on general purpose processors are classified already by programmable distributors on the packet processors. We gave favour to flexibility instead of maximum performance. Hence we based our implementation on a layered model with a proxy device driver that provides the hardware abstraction layer to access NP-blades. To benefit as much as possible from the fast path while keeping the flexibility, we introduced hooks in the proxy device driver that allow for fast packet dispatching to the previously identified service chains. If the respective service chain could not be identified on a packet processor, packets are passed along the slow path of the Linux network stack, and classified by the mechanisms of the Linux Netfilter architecture [14]. Our plugin manager registers therefore not only at the hooks provided by the Linux Netfilter, but also at the newly introduced hooks of the proxy device driver if available. Thus, our framework provides an environment that can be instantiated on any type of processor where Linux is available. Provided an appropriate proxy device driver exists, the environment can provide the benefits of a NP transparently to the service chains.

Our hierarchical router platform that provides service component extensibility at node run-time has been evaluated in [6]. We evaluated our proof of concept implementation on a three-tier hierarchical router in which an Intel Xeon 2.4GHz was installed as a host processor together with an application reference board for the IBM PowerNP 4GS3. For a configuration without real service functionality, in which packets were sent from the host processor directly to the egress packet processors of one network port, received at the ingress side at another port and then sent back to the host processor, we measured a maximal packet transfer rate of approx. 298kpps (kilo-packets-per-second) with packets of 72Bytes each, and a maximal data transfer rate of approx. 956Mbps (megabit-per-second) with packets of 1460Bytes each.

IV. Related Work

VERA [15] introduces the hierarchy of classifiers as a chain of classifiers that is mapped on a hierarchical router modelled as a spanning tree composed of switches and processors. The model is focused on building a three-level standards compliant, modularized, extensible router of commercial off-the-shelf personal computer components. It defines extensibility as the ability to provide resources for additional services. However, the core components of VERA do not provide at run-time extensibility, and VERA does not deal with the complexity of instantiating services that span all tiers.

NetBind [16] proposes an approach to construct data paths dynamically on a network processor-based router. Low latency on dynamic binding is one of the outstanding features of NetBind. The low latency is achieved due to the patches that

are applied to the component's machine code. Therefore no overhead takes place at the execution time, except for the machine code changes. Nevertheless, NetBind is not a generic framework for adding new services on network processor-based routers, i.e., NetBind does not deploy services on all tiers of the processor hierarchy. Instead, NetBind aims to tackle solely the construction of a dynamic data path based on packet processor components.

Click [17] is an extensible component-based router for commodity operating systems. The Click router is constructed through the selection of a certain number of components that carry out small tasks (called elements), which in turn are aggregated into a graph structure. Click poses as an extensible architecture by providing a low cost insertion of new configurations and is still able to implement complex behaviours like packet scheduling. Originally Click was not developed for network processor based routers. NP-Click [18] is an extension of Click for such environments. However, Click defines an architecture that does not allow for dynamic extension at run-time, but only at compile-time.

Chameleon [19] facilitates the installation of a service on a single node. A service is expressed using a node-independent description, allowing one to send the same service specification to different types of active nodes. Each node that needs to install a service, resolves the service specification depending on its locally available processing capabilities, thus allowing a service to exploit the particular functionality of an active node. This way, the service installation scheme can support heterogeneous active nodes and still take advantage of node-specific performance features. While our work can benefit from the specific descriptors provided by Chameleon, Chameleon models a node at the level of execution environments and, hence, does not deal with the complexity of managing and controlling the resources of hierarchical, heterogeneous nodes.

Our active router model generalizes the hierarchy of processors by removing the limitations incurred with the spanning tree, and provides the expressiveness to describe the dynamic assignment of processing elements to link interfaces. Based on this generalized model, we have designed and implemented the architecture of PromethOS NP, a framework that provides scalability by the concept of distributed processing environments instantiated on general purpose processors like embedded in legacy computers as well as in programmable network interfaces. The processing environments provide the mechanisms to install, instantiate, configure and execute code components on all tiers at run-time of the node. Our framework provides the resource management and control mechanisms that are required to instantiate and run services that span various processing elements. 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 programmable distributor concept, 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.

V. Summary, Conclusion and Outlook

In this paper, we introduced the model of a hierarchical high-performance router that consists of a multitude of processor hierarchy levels, along with a service model composed of individual service components. We presented the architecture of the PromethOS NP framework that supports component portability by the extended PromethOS EE across different node configurations. The scalable architecture of PromethOS NP is based on the concept of code-extensible, distributed processing environments that can be instantiated on general purpose processors, and benefit from NPs if available by that processing environments that follow the same model are instantiated on packet processors.

The performance measurements carried out with our proof of concept implementation prove the efficiency of our architecture. Our dynamically, code-extensible software router architecture, PromethOS NP supported by the IBM PowerNP 4GS3, 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 host processor, the PowerNP provided ample capacity for additional flow-processing. Hence we are convinced that PromethOS NP in conjunction with network processors provides a flexible and efficient architecture and platform for active services that need to process packets at link-speed.

Currently, we are working on scalable algorithms to alleviate the mapping process that is required to allocate services efficiently on all of the available processing elements thus improving the overall workload distribution among heterogeneous processors.

Acknowledgment

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, and the donation of an IXP1200 Evaluation Board from Intel Corp.

References

[1] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden, "A Survey of Active Network Research," *IEEE Communications*, vol. Jan., 1997.

[2] Intel Corp., "Intel IXP1200 network processor – datasheet."

[3] ——, "Intel IXP2400/IXP2800 Network Processor: Programmer's Reference Manual," http://www.intel.com, January 2004.

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

[5] Intel Corp., *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 2002.

[6] L. Ruf, R. Pletka, P. Erni, P. Droz, and B. Plattner, "Towards High-performance Active Networking," in *Proc. of the 5th Annual Int. Working Conf. on Active Networks IWAN*, ser. Lecture Notes in Computer Science, no. 2982. Kyoto, Japan: Springer Verlag, Heidelberg, December 2003.

[7] R. Keller, L. Ruf, A. Guindehi, and B. Plattner, "PromethOS: A dynamically extensible router architecture supporting explicit routing," in *Proc. of the 4th Annual Int. Working Conf. on Active Networks IWAN*, December 2002.
[8] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, "Router plugins: A software architecture for next-generation routers," 2000.
[9] S. Choi, D. Decasper, J. DeHart, R. Keller, J. Lockwood, J. Turner, and T. Wolf, "Design of a Flexible Open Platform for High Performance Active Networks," in *Proc. of the Allerton Conf. on Communication, Control, and Computing*, Sep. 1999.
[10] F. Kuhns, J. DeHart, A. Kantawala, R. Keller, J. Lockwood, P. Pappu, D. Richards, D. Taylor, J. Parwatikar, E. Spitznagel, J. Turner, and K. Wong, "Design of a High Performance Dynamically Extensible Router," in *Proceedings of the Fourth Annual International Working Conference on Active Networks IWAN*, ser. DARPA Active Networks Conference and Exposition (DANCE), San Francisco, May 2002.
[11] The FAIN Consortium, *D7: Final Active Network Architecture and Design*, 2003.
[12] J. Postel, "Internet Protocol Specification," ISI, RFC 791, September 1981.
[13] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) – Specification," ISI, RFC 2460, December 1998.
[14] P. R. Russell, "The NetFilter Project," http://www.netfilter.org, 2004.
[15] S. Karlin and L. Peterson, "VERA: An extensible router architecture," in *Proc. of the 4th Int. Conf. on Open Architectures and Network Programming (OPENARCH)*, April 2001, pp. 3–14.
[16] A. Campbell, M. Kounavis, D. Villela, J. Vicente, H. de Meer, K. Miki, and K. Kalaichelvan, "NetBind: A Binding Tool for Constructing Data Paths in Network Processor-based Routers," in *Proc. of the 5th Int. Conf. on Open Architectures and Network Programming (OPENARCH)*, June 2002.
[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, M. Kaashoek, and C. Modular, "The click modular router." *ACM Transactions on Computer Systems*, vol. 18(3), pp. 263–297, August 2000.
[18] N. Shah, W. Plishker, and K. Keutzer, "NP-Click: A programming model for the Intel IXP1200," in *Proc. of 9th Int. Symp. on High Performance Computer Architectures (HPCA), 2nd Workshop on Network Processors*, Feb. 2003.
[19] 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.