Creating Advanced Functions on Network Processors: Experience and Perspectives

Robert Haas*, Clark Jeffries[‡], Lukas Kencl, Andreas Kind, Bernard Metzler, Roman Pletka,
Marcel Waldvogel, Laurent Freléchoux, and Patrick Droz
IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland
[‡] IBM Corporation, Research Triangle Park, NC 27709, USA
* corresponding author: rha@zurich.ibm.com

November 4, 2002

Abstract

In this paper, we present five case studies of advanced networking functions and how a network processor (NP) can provide high-performance and flexible support for each of them. We first review the basic NP system architectures, and describe in more detail the IBM PowerNP architecture from a data plane as well as from a control plane point of view. We introduce models for the programmer's views of NPs that facilitate a global understanding of NP software programming. Then, for each case study, we present results from prototypes as well as general considerations that also apply to a wider range of system architectures. Namely, we investigate the suitability of NPs for quality-of-service (active queue management and traffic engineering), header processing (GPRS tunneling protocol), intelligent forwarding (load-balancing without flow disruption), payload processing (active networks code interpretation and just-in-time compilation), and protocol stack termination (SCTP). Finally, we summarize the key features required by each case study, and make concluding remarks regarding the future of NPs.

1 Introduction

The advent of network processors was driven by an increasing demand for high throughput and flexibility in packet routers. As a first step in the evolution from software-based routers to network processors (see Figure 1), the bus connecting network interface cards with the central control processor (Control Point, CP) was replaced by a switch fabric. As demand grew for even greater bandwidth, network interface cards were replaced by Application-Specific Integrated Circuits (ASICs), meaning that packets no longer had to be sent to the CP for forwarding. ASIC-based routers, however, turned out to be not as flexible as necessary in the fast and diverse net-

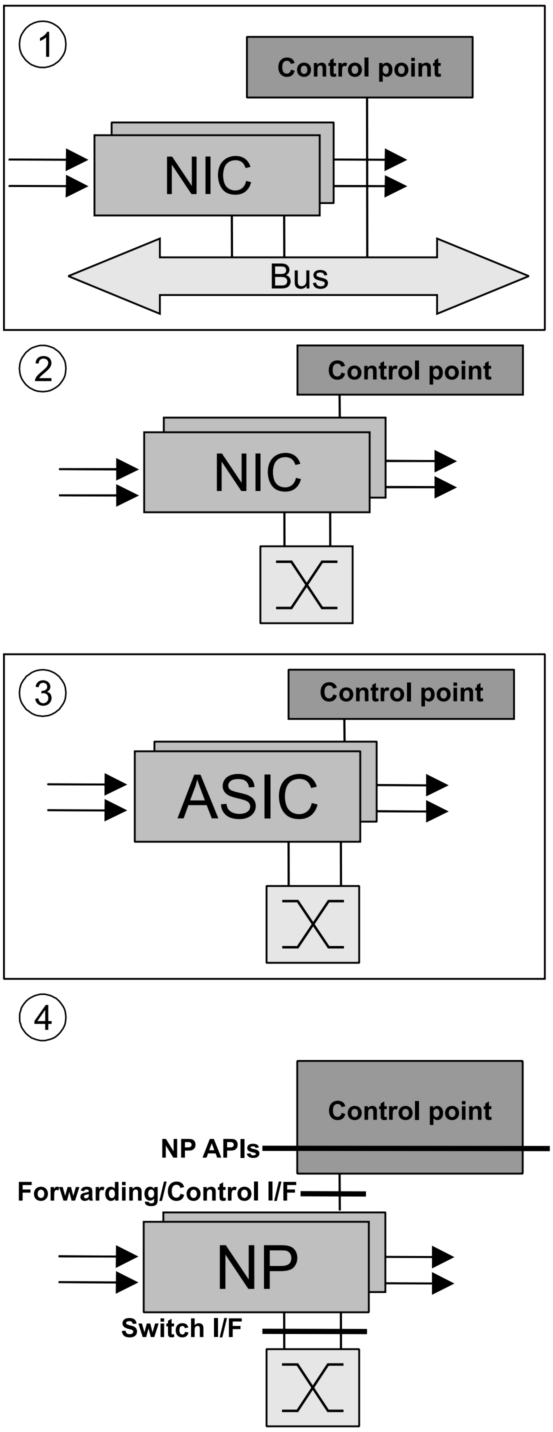


Figure 1: The advent of network processors.

work equipment market. Also, hardware development cycles tended to be too slow to accomodate the continuous demand for new features and support for additional protocols. The need for adaptability, differentiation, and short time-to-market brought about the idea of using Application Specific Instruction-set Processors (ASIPs). These so-called *Network Processors* (NPs) can be programmed easily and quickly according to specific products and deployment needs. Such a set of Application Programming Interfaces (APIs) and protocols are currently being standardized by bodies such as the Internet Engineering Task Force (IETF) ForCES [1] working group and the Network Processor Forum [2]. This finally enables users to easily take advantage of the full NP capabilities: the powerful combination of performance and flexibility that allows the efficient development of advanced networking functions.

Today, a wide variety of NPs exists, each providing a different combination of flexibility, price, and performance [3, 4]. Despite this variety, NPs share many features, such as the capability to simplify and accelerate the development of advanced networking functions. Even though this paper will focus on our experience with the IBM PowerNP [5] network processor, it provides insight into a much wider selection of existing and upcoming products, as the concepts discussed apply to most other NPs as well.

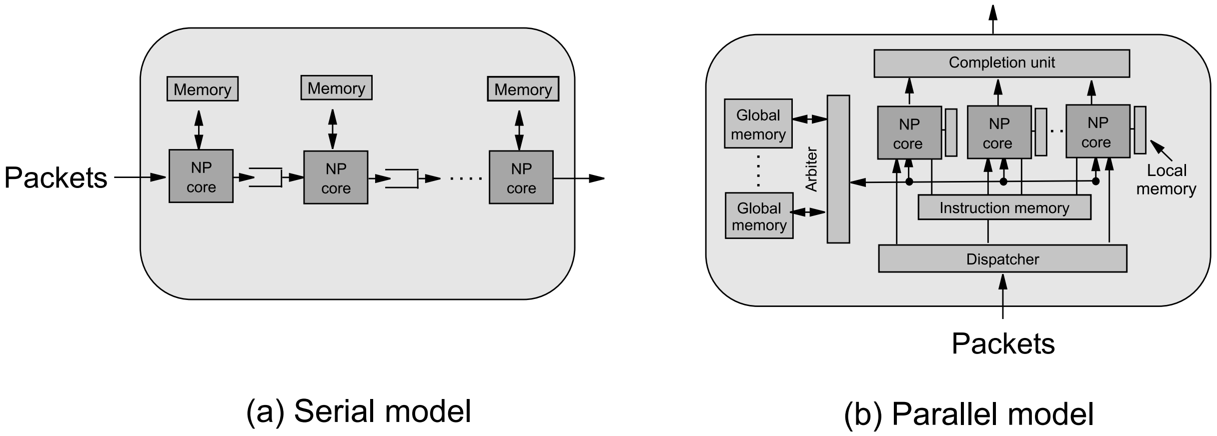
This paper describes our experience during the design and implementation of a wide selection of networking features. It provides insight into many of the enabling factors we found necessary during the implementation of functions ranging from currently widespread features such as header processing and Quality-of-Service (QoS) enforcement to traffic-engineered packet processing. Furthermore, we worked on functions that are less widely considered for use in NPs, such as code compilation, intelligent forwarding, and protocol termination. We are thus able to explain, based on our first-hand experience, how NPs simplify and help accelerate the development of these functions. This paper discusses our insight gained from this broad spectrum of designs and describes the lessons learned.

The paper is structured as follows. Section 2 presents a small taxonomy of NP architectures and describes an example of an NP architecture in sufficient detail for the following five case studies. Section 3 states the seven goals of QoS and explains how they can be achieved by active queue management and traffic engineering. In Section 4, we present our experience implementing the GPRS Tunneling Protocol used for mobile Internet access. Section 5 introduces advanced server load balancing techniques and how they can be implemented efficiently by taking advantage of NP-specific functionality. Section 6 presents how just-in-time (JIT) compilation of active networking code can be implemented in NPs and how this helps performance. The last case study is presented in Section 7 and describes experiences gained from the implementation of the Stream Control Transport Protocol, which is expected to be widely adopted over the next few years for diverse applications. In Section 8, we conclude by summarizing and comparing the individual features which were major enabling factors for each of the case studies.

2 Network Processor Architecture

The challenge in NP design is to provide fast and flexible processing capabilities that enable a variety of functions on the data path yet keep the simplicity of programming similar to that of a General-Purpose Processor (GPP). The NP system architecture plays a significant role in this respect: such architectures are primarily designed according to a serial model (or pipeline) or a parallel model, as shown in Fig. 2 [6]. Emulating the other model is usually possible but can lead to degraded performance.

In the parallel model, each thread in an NP



(a) Serial model

(b) Parallel model

Figure 2: NP concurrency models

core receives a different packet and executes the entire data-path code for this packet. In the serial model, each NP core receives each packet and executes a different portion of the data-path code in a thread.

From a programming point of view, the serial model requires that the code be partioned such that the work is evenly distributed, as the performance of the NP equals that of its most heavily loaded NP core. In the parallel model, given that the same code can be performed by any NP Core and packets are assigned to the next available thread in any NP Core, the work is inherently evenly distributed. The Intel IXP [7] architecture is designed primarily on a serial model, whereas the IBM PowerNP [8] is designed on a parallel model. For realistic, changing traffic mixes, the serial model would require a dynamic repartitioning of the code to maintain performance, unlike the parallel model for which no partitioning takes place.

Figure 3 describes the NP core programmer's view of the PowerNP that can handle up to four 1 Gbps ports. It consists of 16 NP cores or picoprocessors that are scaled-down RISC processors running at 133 MHz, with a PowerPC-resembling instruction set. Each picoprocessor supports two threads, so there are up to 16 threads executing simultaneously. Multithreading keeps a picoprocessor busy, for instance when a thread is waiting for coprocessors results. A thread entirely processes a packet, i.e., threads are in run-to-completion mode.

To accelerate common tasks compared to executing them in picocode, eight coprocessors (shown with ellipses in Fig. 3) are integrated to perform asynchronous functions such as longest-prefix lookup, full-match lookup, packet classification, hashing (all done by the two Tree Search Engine (TSE) coprocessors), data copying, checksum computation, counter management, semaphores, and policing, and packet memory access. The TSEs also provide access to the control memory store, which is composed of internal and external memories of different widths and access times. A number of hardware assists are also available that accelerate tasks such as frame alteration or header parsing.

Packet processing is divided into two stages: ingress and egress. Ingress refers to the data-flow from the link towards the switch interface, and egress is the opposite. The same threads can perform processing at ingress or egress, thereby automatically balancing the processing power where needed. Along with the packet, additional context information can be transported from ingress

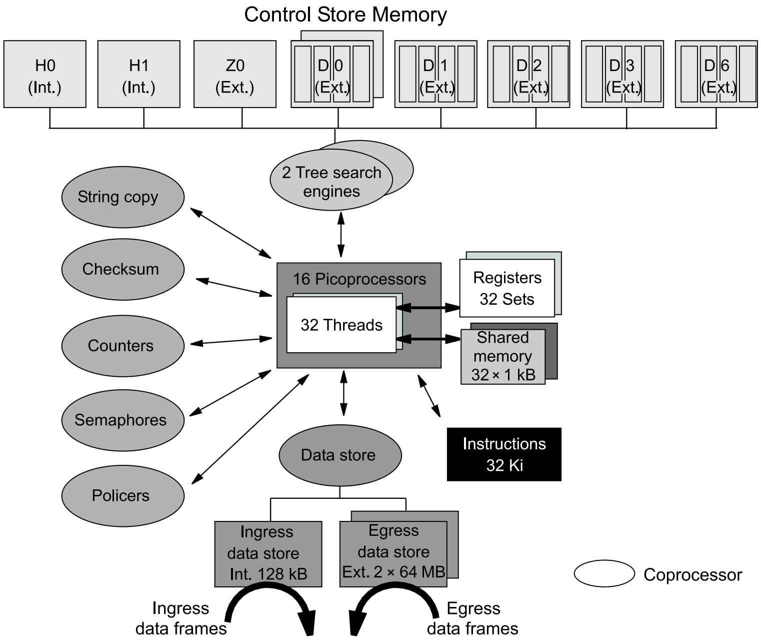


Figure 3: Programmer's view of the picoprocessors in PowerNP.

to egress, such as the output port identifier of the egress NP obtained by the IP forwarding lookup previously executed on the ingress NP. Note that the ingress and egress NP can be the same.

NPs and CPs do not map strictly to data plane and control plane. In fact, data-plane functions can be very well executed partly by the NP, partly by the CP: in order to optimize NP instruction memory usage, non-performance critical packet processing tasks can be deferred to the CP. Similarly, control path functions can be off-loaded from the CP to the NP if this results in performance gains, as illustrated in the following sections.

Available functions programmed in picocode in the NP as well as the rest of the NP hardware are driven by higher-level APIs from the CP. The communication between the two takes place using control messages processed by a special thread in the NP. These CP APIs control logical components that are represented by rounded rectangles in the ingress and egress data paths in Fig. 4.

3 QoS Provisioning

The drivers of the Internet are e-mail, web surfing, and bulk data transfer. All of these are based on a common IP forwarding service referred to as *Best Effort* (BE). With the convergence of telephony and data networks, increased Internet utility will next be engendered by QoS in IP networks. Needed is an optimal balance of possible approaches, roughly choosing between macromanagement (one-size-fits-all over-provisioned BE service) and micromanagement (circuit-switched telephony). We assume a QoS model in which traffic can either be BE or in some Premium class.

The general requirements placed on an IP router to enable an improved level of QoS over BE are the following.

1. Despite any physically possible level of data congestion, realtime Premium traffic such as IP Voice that arrives at a rate under its contractual bandwidth limit must not be

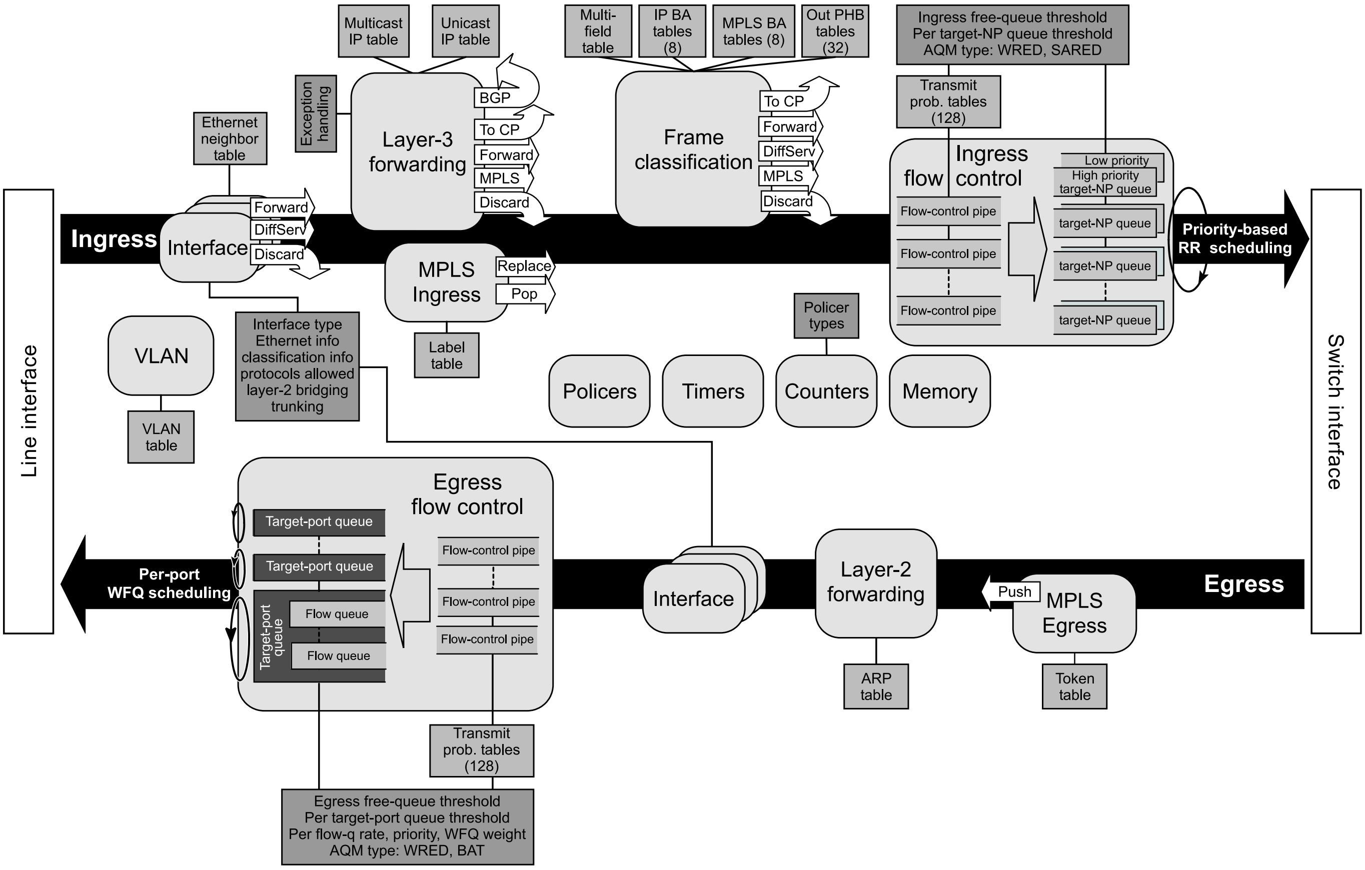


Figure 4: Programmer's view of the PowerNP from the CP APIs.

dropped or unduly delayed.

2. Non-realtime Premium traffic, that is, Premium data, that conforms to its bandwidth guarantees should not be dropped and its queuing delay must be short regardless of congestion by BE traffic.

3. During steady congestion conditions, all queues should be low, ensuring low queuing latency for all traffic. Exceptional bursts should fill the buffer, however. There is no excuse for high queuing latency during prolonged episodes of relatively steady congestion.

4. Utilization should be high, that is, if excess bandwidth remains after Premium traffic has been served, then it should all or almost all go to BE.

5. Bandwidth allocation should be fair and predictable.

6. As congesting conditions change, the response of the NP should be fast convergence to a new equilibrium that includes the above five requirements.

7. Last but of highest importance, all of the above should be automatic and easy to understand and administer.

Typically, up to a certain offered load, latency and loss are minimal and nearly constant, and then increase abruptly, i.e., the onset of congestion. Subtleties include finite packet life, finite storage, traffic value and precedence, and unpredictable duration of bursts. An *active queue management system* addresses these issues in routers by actively dropping packets before queues overflow. The next subsection describes the implementation of an active queue management system for enforcing QoS on an NP. It shows that the programmability of NPs allows one to implement active queue management systems without sacrificing performance.

3.1 Active Queue Management

An implementation of an active queue management system for enforcing QoS guarantees can make use of the following NP hardware support:

- Hardware-supported flow control is invoked when a packet is enqueued in ingress and egress. The transmit probabilities used for flow control are taken from a table stored in fast access memory, i.e., the *Transmit Probability table.*

- The key to index into the transmit probability table is composed of packet header bits (e.g. IETF DiffServ code point). The access to these header bits is supported by a *header parser hardware-assist.*

- Furthermore, detailed flow accounting information about individual offered loads is provided via *hardware counters.* Such counters as well as *queue level indicators* and *queue depletion indicators* at various locations in a NP can be used to update the transmit probability memory (e.g. ingress general queue, ingress target NP-queue, egress flow queue, egress port queue).

Traditional active queue management systems use general queue levels to determine transmit probabilities. However, such an approach is not feasible for high-speed routers with significantly varying offered loads [9].

The *BAT* active queue management system developed as a standard feature for the PowerNP allows Premium traffic to be organized at each bottleneck in the NP into flow control pipes (see Fig. 4) of different kinds [10]. A pipe is a local (per bottleneck) aggregation of traffic in one class. In our view, the simplest way to provide useful QoS is for each pipe to have an aggregate minimum bandwidth guarantee *min* and an aggregate bandwidth maximum *max.* Hence $0 \leq min \leq max$. If multiple minimum bandwidth guarantees are represented by constituent flows in a pipe, then these must be summed to achieve a pipe minimum. As network end-to-end paths with guarantees are added or deleted, ensuring the guarantees of the minimums at each NP in the network is the complex yet hidden work of network control, as described in the next subsection.

The implemented active queue management system uses a modest amount of control theory to adapt transmit probabilities to current offered loads. A signal is declared that defines the existence of excess bandwidth. This signal can use flow measurements, queue occupancy, rate of change of queue occupancy, or other factors. If there is excess bandwidth, then flows that are not already at their *max*s are allowed to increase linearly. Otherwise, the rates allocated to flows not already below their *min*s must decrease exponentially.

For example, let us suppose four strict-priority flows in egress are all destined to the same 100 Mbps target port as shown in Table 1. All bandwidth units are in Mbps. If the four flows offer rates of 15, 15, 50, and 100 (so sum = 180), then the correct allocation should be: all Flow0, Flow1, and Flow2 traffic should be transmitted, and of Flow3 one fifth of the packets should be transmitted (packets selected randomly).

Furthermore, the above allocation should be reached quickly and with low queue occupancy.

Table 1: Example of flows for bandwidth allocation.

Flow Label	Type	Minimum	Maximum	Priority (strict)
Flow0	realtime	10	30	0 (highest)
Flow1	non-realtime	20	40	1
Flow2	non-realtime	0	100	2
Flow3	non-realtime	0	100	3 (lowest)

We emphasize the above requirements 1–3: queuing latency during steady congestion is low. This is the latency packets endure while awaiting processing. For example, suppose 101 Mbps of traffic is forwarded to one 100 Mbps target port and the egress data store is 128 Mb. Suppose the flow control were conventional taildrop, that is, drop arriving packets only if the queue occupancy becomes full. It follows that steady congestion of 101 Mbps would fill the egress data store to the top and cause unacceptable queuing latency of 1.28 s. At the same time, setting a low taildrop threshold could cause unacceptable shaving of bursts.

The control loop based on control theory addresses also the remaining requirements 4–7. The feedback signal can be tuned to achieve high utilization, fast convergence, fairness, and administrative simplicity [10]. The latter advantage is by virtue of the fact that the active queue management system is configured with minimum and maximum flow rates rather than queue levels.

There are virtually no costs for implementing the active queue management system in the data plane of the PowerNP because the header parser hardware-assist and hardware flow control support can be configured according to the new scheme. The implementation costs in the control plane are about 300 cycles to update a single transmit probability value. This update is triggered in fixed time intervals for each pipe at each output port. These costs cover line-speed forwarding and normal control operation. It was possible to make this advanced networking function swiftly operational as a standard feature on the PowerNP which demonstrates the flexibility of NPs.

3.2 Traffic Engineering Reference Platform (TERP)

In this section, we highlight how the NP is used in the context of Traffic Engineering (TE), both from the data plane and control plane points of view. Our implementation of TE relies on RSVP-TE to set up MPLS paths (or LSP, for Label-Switched Path) through the network and on Diff-Serv to provide QoS. An OSPF-routing mechanism with specific TE extensions is used to collect QoS-usage information throughout the network. Our own Route Server component computes paths on request from RSVP. TE permits ISPs (Internet Service Providers) to start offering value-added commercial-grade services. Each MPLS node is composed of one or more NPs, interconnected with a swtiching fabric and attached to a CP: these NPs and the CP together act as a single MPLS node.

In the data plane, TE nodes perform traffic classification, policing, shaping, marking, dropping, and forwarding. The logical components described in Fig. 4 are configured from the CP by a Label-and-Resource Manager process to perform accordingly: Ingress MPLS nodes perform traffic classification, policing, and DiffServ marking, whereas transit nodes perform MPLS forwarding (MPLS label replacement) and egress MPLS nodes perform IP forwarding. All nodes can be configured to use WRED or BAT AQM to perform bandwidth allocation.

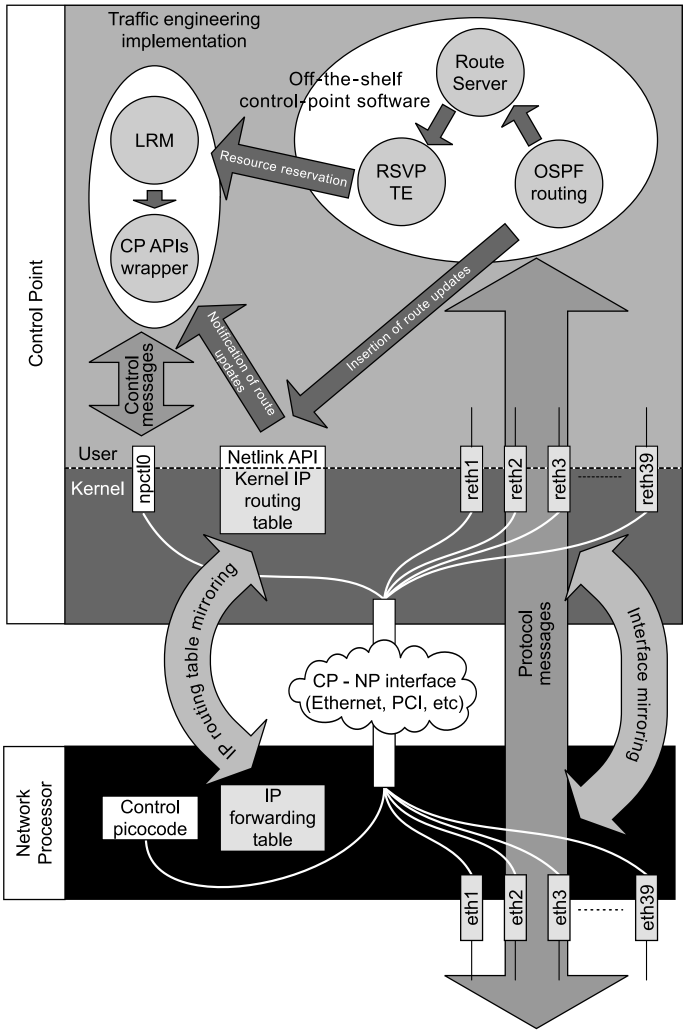


Figure 5: TERP control-point architecture.

In the control plane, the CP runs the RSVP signaling, OSPF routing, and possibly the Router Server processes. As shown in Fig. 5, each NP Ethernet port (eth1 through eth39) is mirrored in the CP as a normal interface (reth1 through reth39): CP processes therefore can send and receive packets (protocol messages) as if the MPLS node were built as a centralized software-based router. In addition, the CP kernel IP routing table is mirrored automatically into the NP(s): OSPF routing therefore operates completely transparently from the underlying NP architecture: it uses the netlink API to insert routes into the table. These route updates are notified to the CP-APIs-wrapper process that creates the appropriate control messages to automatically insert these routes into the NP. RSVP interfaces with LRM to perform resource reservation, i.e., to reserve, commit, and release resources as necessary in each node. The LRM performs CP APIs calls that are then translated into control messages destined to the NP.

All these features allow a seamless integration of off-the-shelf control-plane software into the CP.

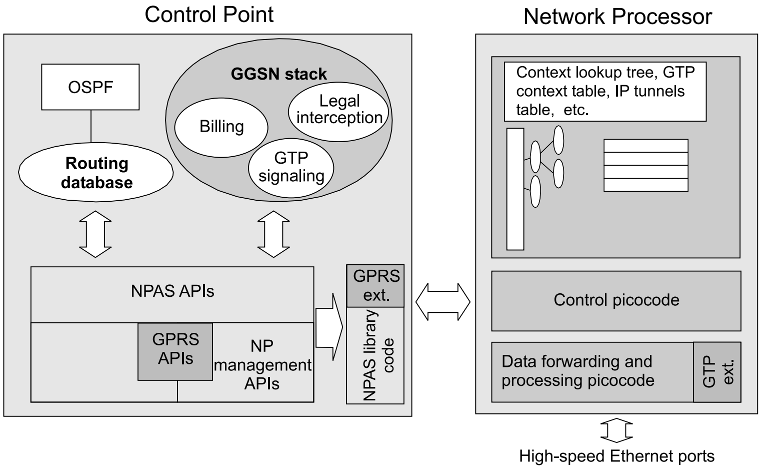


Figure 6: GTP extensions to the NP software.

Table 2: Performance of GTP tunneling

	Instruction cycles	Coprocessors stall cycles	Other stall cycles	Total cycles
encapsulation	402	170	190	762
decapsulation	455	240	207	902

4 Header Processing: GTP

General packet radio service (GPRS) is a set of protocols for converging mobile data with IP packet data. GPRS requires a new infrastructure in the form of GPRS Support Nodes (GSNs) to process packets at a very high rate, yet maintain flexibility as GPRS deployment is still emerging.

Aside from common functions performed by any IP router, such as routing table lookup and packet forwarding, a GSN has to encapsulate or decapsulate IP packets according to the GPRS tunneling protocol (GTP) that associates a specific GTP tunnel with each mobile terminal and performs traffic-volume recording for billing and flow-mirroring for legal interception.

The design of the early prototype allows up to one million GTP tunnels. The increase in processing complexity required by GTP encapsulation and decapsulation results in a processing capability of roughly 2.2 million packets per second (Mpps) per NP.

The encapsulation process requires the retrieval of a GTP context (i.e., a mapping to a GTP tunnel) based on the IP address of the packet being encapsulated and the construction of the GTP header using information contained in the context. A header chain composed of the GTP, UDP, and IP headers is then prepended to the packet. The decapsulation process requires the retrieval of the GTP context from the IP address of the inner IP header. The outer header chain composed of the IP, UDP, and GTP headers is stripped, and normal IP forwarding is applied to the decapsulated packet. In both encapsulation and decapsulation cases, traffic counters associated with the context are asynchronously incremented to account for the data transmission.

As shown in Fig. 6, the implementation of the GPRS extensions to the network processor consists of the following:

- Design of the GPRS service APIs
- Design of the data structures for storing the GTP contexts, counters, and tree lookup
- Extension of the control-code library on the CP to provide new GPRS CP-APIs
- Extension of the control picocode on the NP to implement the CP APIs
- Extension of the picocode on the data path to perform the GTP tunneling function.

Generic CP APIs for tree, table, memory block, and counter management can be used that greatly simplify the management of the GTP lookup-tree, the GTP context table, and the GTP counter tables. The design makes extensive use of the NP coprocessors: GTP contexts are organized as a tree, and the TSE coprocessor is used to retrieve the GTP context associated with an incoming packet. Counters for traffic accounting are incremented using the counter-management coprocessor. Header prepending and stripping use the flexible frame-alteration hardware assists of the NP.

Table 2 shows the number of cycles spent for one frame both in ingress and egress processing for the tunneling tasks, including stall cycles spent waiting for coprocessor results or memory and instructions accesses. The processing includes layer-2, layer-3, and GTP encapsulation or decapsulation.

5 Intelligent Forwarding: Adaptive Load Balancing

Many networking applications, such as Web server farms, benefit from spreading the data processing among multiple servers or processing units. The task of adaptively balancing the load among multiple servers is nontrivial due to the high volume and unknown characteristics of the traffic and the need to maintain connectivity of active packet flows between hosts. To balance simultaneously a large number of flows, it is necessary to minimize the amount of state information stored.

The adaptive load balancing method (Fig. 7) uses a hardware-based hash function to determine the destination server. The hash function is based on the robust hash routing algorithm [11], which supports arbitrary processing capacities of the balanced servers, and on its adaptive extension [12], which minimizes the disruption of the flow-to-processor mapping. The hash is performed on a portion of the packet that is constant for the duration of a flow, such as the source address. The method alternates between a state where only one hash function is computed, and a transient state, where two hash functions are computed.

At the initial phase, one hash function is configured, based on the resource capacities of the servers. Once the method has been put into operation, statistics are accumulated on the resources utilization. If some servers become over-utilized relative to other servers, a new hash computation is determined. The new hash function H_{new} optimally distributes network traffic based on the statistics gathered.

During the transient period, both the old (H_{old}) and the new hash functions (H_{new}) are computed on each packet simultaneously. Packets in the intersection of the two hash functions ($H_{new} = H_{old}$) continue to be routed to the resulting server. Packets that do not fall in the intersection of the two hash functions ($H_{new} \neq H_{old}$) are redirected to the CP for routing. The CP keeps state for each flow it sees and routes flows in progress to the result of the old hash function and new flows to the result of the new hash function. The state for flows that are finished or do time-out is deleted. After a configured period of time, Multi-Field Classification rules, which spec-

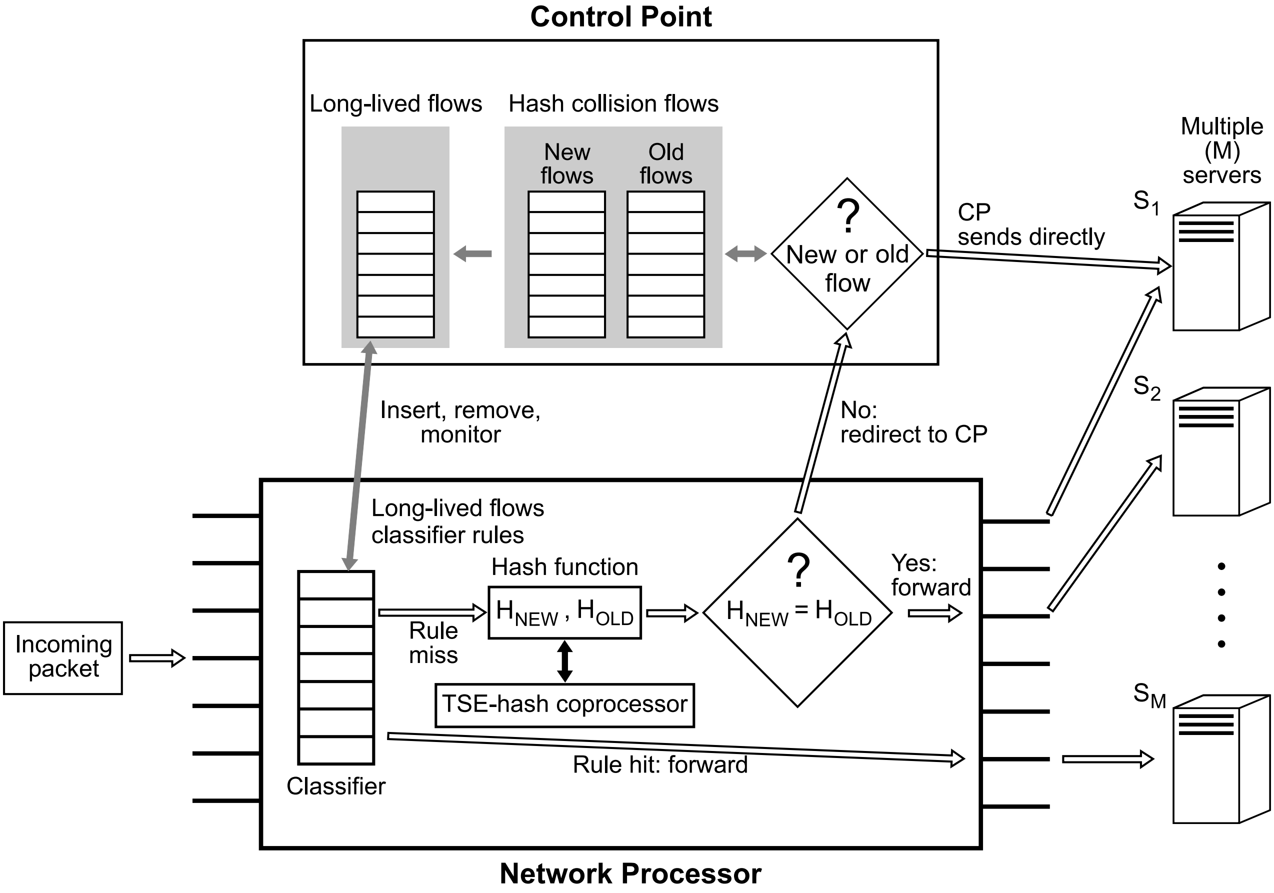


Figure 7: Server Load Balancer on the NP: diagram of the data path.

ify to which server the flow is routed, are installed for the remaining (”long-lived”) flows in the state table. The old hash function is then discarded and the method returns to a state of a single hash function. The Multi-Field Classification rules are being further monitored and removed upon flow termination or time-out.

The method continually alternates between the one-hash and two-hash states, thus adapting to the current traffic conditions.

Advantages of this approach include:

- No flows in progress are ever moved between servers, ensuring uninterrupted flow connectivity;
- State information is only maintained for flows not in the intersection of the two hash functions, minimizing hardware costs;
- The intersection of the two hash functions is mathematically maximized, thus further minimizing the state kept;
- Routing is performed partially in hardware, using hashes performed by the TSE coprocessor in the NP, thus exploiting the high data rate of the device.

Several software and hardware components of the PowerNP have been used in prototyping the load balancer application: standard layer-

2, layer-3 and layer-4 (Multi-Field Classification) forwarding elements, as well as the hash function of the TSE coprocessor. The availability of these ready-made components made the datapath programming on the NP shrink to the relatively modest work of implementing the hash routing method, its managing API and the CP redirection. The speed and good spreading properties of the hash function in the TSE coprocessor enable us to consider the hash computation a black-box, eliminating the necessity to implement one's own hash function.

The number of processor instructions required to execute the hash routing method on each packet is dependent on the number of balanced servers M. The instructions are primarily dedicated to reading and carrying out operations on the per-server weights, while calling the TSE coprocessor in parallel. Up to $M = 8$ the prototype implementation requires executing $75 + M * 20$ instructions. For $M > 8$, the number of instructions executed grows logarithmically with M, as the weights' table is then organized into a tree structure. Further details about the implementation can be found in [13].

6 Payload Processing: Active Networking

The following two major approaches have been discussed by the Active Networking community: The capsule approach [14–16] embed into data packets active code that is executed on each node along the path. The programmable switch approach maintains the existing packet format and provides programmability by a discrete mechanism that supports the dynamic downloading of programs [17, 18]. Custom code has to be loaded into routers prior to data path packet handling so that it can be executed upon packet arrival.

The combination of the capsule approach with a byte-code language that possesses intrinsic safety properties [16] leads to code compactness and architectural neutrality, and fits into the run-to-completion thread model or the pipeline model of NPs. The execution environment that emulates the corresponding virtual machine can be implemented in the data plane of NPs. In the pipeline model, a certain number of NP cores could be dedicated to active code processing.

Such a framework provides the necessary flexibility and safety needed in active networks [19]. Although the approach is general enough to be used for other applications, our work on Active Networks is motivated by the fact that end-to-end QoS guarantees cannot be given in IP networks today. Active Networks shift the traditional view of networking where programmability is given by the definition of protocols, and hence limited to their functionalities, towards a world where packets can carry active code that is being executed on-the-fly in networking nodes. Inter-operation between different protocols and translation mechanisms between existing QoS frameworks, as can be encountered in heterogeneous environments, is not feasible using protocols, but can be solved with active networks.

Additional speed up compared to interpretation in the execution environment can be achieved by just-in-time (JIT) compilation of active code [20]. For a typical active network framework the number of processor cycles spent for compiling an individual instruction is only slightly greater than the number of cycles needed for interpretation. We measured that native execution of an individual machine instruction is, on average, more than ten times faster than interpreting an equivalent byte-code from the virtual instruction set. From these results, it is clear that the performance benefit of JIT compilation increases the more often the processor executes a part of the program, or even the entire program, without recompilation. Reuse depends on the amount of recursion as well as looping in the programs and can be supported by caching compiled code either at routers (e.g., reuse for packets of the same flow) or inside the packet. In the latter case, compiled code can be reused at every intermediate router that supports the same native instruction set.

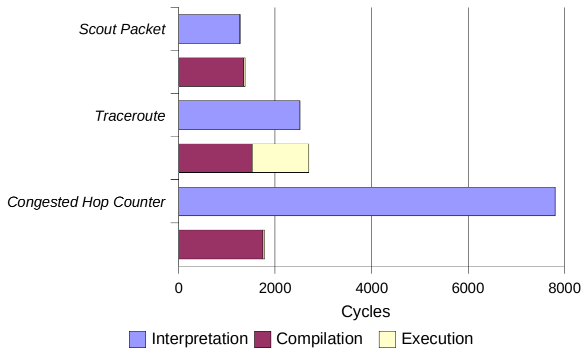


Figure 8: Execution cycles for active code on a NP.

We implemented and tested a general active network setup based on a dialect of the SNAP active networking language [16] on the PowerNP. Results show that JIT compilation is not only feasible in an active router because the compiler is small enough and fast enough to run in the data plane's picoprocessors, but also leads to significant performance improvements. Figure 8 compares three different types of active packets. The *scout* active packet discovers the list of active routers between the source and destination and is 22 byte-code instructions long. It ensures that the list does not grow beyond its allocated memory boundary. Although compilation and execution do not outperform interpretation, the remarkable execution cycle cost shows the potential of native code caching techniques. The *traceroute* active packet sends a new active packet containing the IP address of the current traversed router back to the source. As opposed to the scout packet, the traceroute packet consisting of 26 byte-code instructions does not perform memory boundary checks because information is immediately sent back. The performance of JIT compilation is slightly reduced as not all byte-code instructions are executed on all hops, therefore the JIT compiler unnecessarily compiles parts of the code not needed at the current hop. The third active packet is a *congested hop counter* using 28 byte-code instructions. The program collects information on the number of congested queues in active routers along a path through the network. In contrast to the previous packets, this program uses a loop to accumulate the data and performs tree lookups in the loop, hence leading to accelerating JIT compilation.

Not all of the assumptions apply to currently existing NPs: critical for the generic applicability of JIT compilation in active networks are write access to the instruction memory and sufficient instruction memory size to hold the compiler and the JIT-compiled active code.

7 Protocol Termination: SCTP

Protocol termination within the network is a technique that allows the realization of applications such as stateful firewalls, application-level server load balancers or even transport-layer protocol gateways. Available solutions are often based on a given host-based transport stack, which typically leads to significant performance limitations. To overcome these limitations, the integration of a wire-speed protocol termination into an NP-based intermediate system was envisioned. In this context, the termination of a typical stateful, reliable transport protocol such as TCP poses the following challenges:

- A per-microflow state context must be maintained efficiently.
- A timer service for protocol timers must be offered.
- Segmentation and reassembly (SAR), and retransmission services will employ per-context intermediate packet buffering.
- The typical NP limitations on the available instruction memory size are in conflict with

the extensive functionality of some protocols.

- The protocol termination environment should provide a clean API to allow the seamless and efficient integration of applications such as a firewall or a load balancer.

A challenging example is the termination of the SCTP protocol [21] on the PowerNP. The SCTP protocol was chosen because it combines protocol complexity such as multihoming, multistreaming, partial message ordering and cookie-based association establishment with such typical demands as reliability and robustness.

A first prototype on the PowerNP implements full SCTP termination and already includes multistreaming and multihoming. It offers a socket–like API to easily link together with the envisioned applications at the picocode level. The implementation gives enough headroom in available code space to integrate with such applications.

Although it is still too early to give performance numbers it can be stated that the termination of several hundred thousands of SCTP flows on this type of NP is possible. The model of several parallel, run-to-completion threads, each operating on a given SCTP context at a time, on the one hand, forces to use a semaphore-based context locking mechanism. On the other hand, the run-to-completion model allows the implementation of a natural, event-based code path. Possible events are incoming packets, timer events and application downcalls.

Except incoming packet checksum verification, the entire SCTP code operates on packets in the egress data store of the NP. This gives the necessary amount of packet memory to store data to implement send and receive windows, for SAR and for packet retransmission.

8 Conclusion

The case studies presented here help us visualize and understand several driving factors related to NPs. Figure 9 summarizes how each case study is composed of configurations of existing logical blocks (cf. Fig. 4) and/or new logical blocks with their own CP APIs.

First, they provide insight into the functions that can be performed on NPs, how they can be implemented and how they fit into applications.

Second, we examined the NP features required by different applications. Some of these requirements have been summarized in Table 3.

Third, these different requirements also help explain why the current NPs cover such a wide variety of the design space. Depending on the applications envisioned by the designers, different decisions and compromises had to be implemented.

Some of the open issues in the NP space include software durability, which they share with many other specialized, embedded systems. The processor families offer various programming paradigms, abstraction layers, and coprocessors and/or hardware assists. Therefore, it is currently nearly impossible to write code that would easily port to a different family. But it is also difficult to foresee what improvements and new features future members of a family will support, thus making it advisable to revisit and reoptimize the code when new family members appear.

Fortunately, this is changing for the better. For code running in the data plane, use of a smart optimizing compiler permits to write relatively architecture-independent code. With appropriate, evolving libraries, key subfunctions can be transferred progressively and seamlessly from software implementation to hardware, maintaining backwards compatibility. In the control plane, standard interfaces are being developed and joined with capabilities for dynamic NP feature discovery to allow NP-independent code. Currently, working groups such IETF ForCES [1]

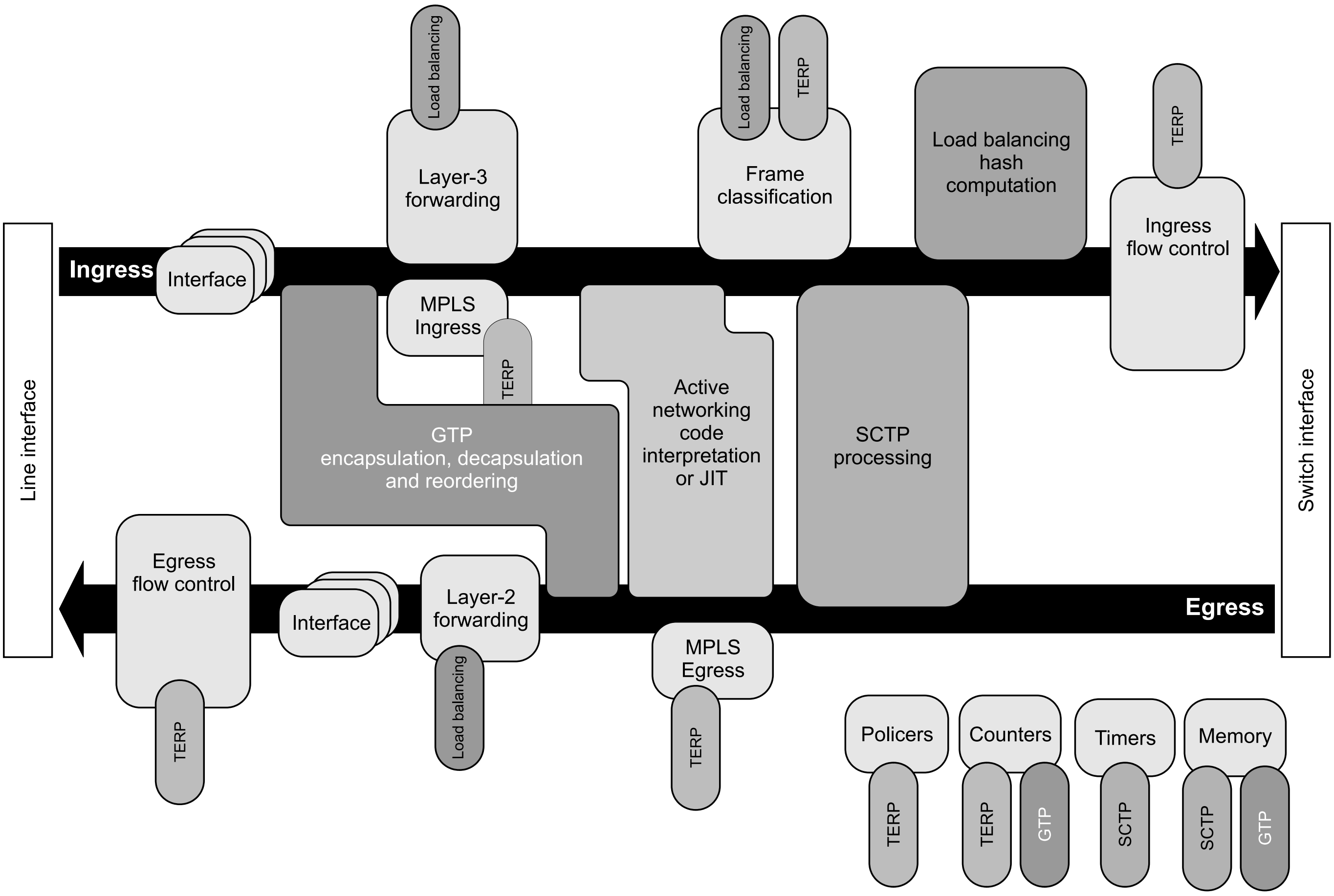


Figure 9: Programmer's view of the five case studies.

and NP-Forum [2] are developing the relevant protocols and semantics, allowing key performance functions to be easily offloaded onto the NP from the CP.

Being forced to implement on an ASIC only a subset of the examples presented in this paper would critically delay time-to-market as well as significantly reduce the ability to adapt to future changes in network patterns or protocols. Offloading them to an embedded general-purpose processor or even the CP would radically reduce the performance achievable. Thanks to their high speed combined with extensibility and modularity, they speed up the development of both control and data path thanks to higher-level interfaces and pre-existing, reusable building blocks.

Stepping back to see the big picture, we can conclude that versatility is one of NPs strongest features. With comparably little effort, it is possible to implement new features to dramatically increase the value offered by a network device, and to to offer new and powerful functions, often combined from amongst a wide variety of existing building blocks. We believe that in the networking world, this makes NPs a strong competitor to ASICs for all but the highest-performance network devices and thus expect their use to grow dramatically in the near future. This will open the door to ever more versatile networks,which in turn will call for new methods to achieve efficient

Table 3: Feature requirements

Case	Requirements
BAT	Probability operations Many per-flow queues Timers
TERP	Policing, Flow-Control, Scheduling, and BA Classification Packet forwarding to CP State synchronization NP ↔ CP
GTP	Millions of exact-match classifier rules Concurrent counters Frame size alteration (pre-/appending) Large packet storage for reordering
Load Bal.	Collaboration NP ↔ CP Fast hash of disjoint header fields Variable hash function Multi-Field Classification for exceptions Fast update of MF Classification rules
Active Nets	Direct write to instruction memory Access to forwarding information Programmable per-packet forwarding Program-controlled multicast Payload processing
SCTP	Collaboration NP ↔ CP Fast CRC Frame size alteration (pre-/appending) Large packet storage for reassembly and reordering Mutual exclusion Scalable per-flow timer support Payload processing

deployment of services in such networks [22].

References

[1] IETF forwarding and control element separation (ForCES). http://www.ietf.org/html.charters/forces-charter.html.

[2] Network Processing Forum. http://www.npforum.org/.

[3] Linley Gwennap and Bob Wheeler. A guide to network processors. Technical report, The Linley Group, 2001.

[4] Peter Glaskowsky. Network processors mature in 2001. Technical report, Cahners, 2002.

[5] IBM PowerNP NP4GS3 network processor datasheet. http://www.ibm.com/chips/techlib/techlib.nsf/products/PowerNP_NP4GS3.

[6] Mohammad Peyravian and Jean Calvignac. Fundamental architectural considerations for network processors. *Computer Networks*, (to appear), 2003.

[7] Matthew Adiletta, Mark Rosenbluth, Debra Bernstein, Gilbert Wolrich, and Hugh Wilkinson. The next generation of Intel IXP processors. *Intel Technology Journal*, 6(3):6–18, August 2002.

[8] James Allen, Brian Bass, Claude Basso, Rick Boivie, Jean Calvignac, Gordon Davis, Laurent Frelechoux, Marco Heddes, Andreas Herkersdorf, Andreas Kind, Joe Logan, Mohammad Peyravian, Mark Rinaldi, Ravi Sabhikhi, Michael Siegel, and Marcel Waldvogel. PowerNP network processor: Hardware, software and applications. *IBM Journal of Research and Development*, (to appear), 2003.

[9] Sally Floyd, Ramakrishna Gummadi, and Scott Shenker. Adaptive RED: An Algorithm for Increasing the Robustness of RED. Technical Report, 2001.

[10] Ed Bowen, Clark Jeffries, Lukas Kencl, Andreas Kind, and Roman Pletka. Bandwidth

allocation for non-responsive flows with active queue management. In *Int. Zurich Seminar on Broadband Communications, IZS 2002*, February 2002.

[11] Keith W. Ross. Hash routing for collections of shared web caches. *IEEE Network*, 11(6), November/December 1997.

[12] Lukas Kencl and Jean-Yves Le Boudec. Adaptive load sharing for network processors. In *Proceedings of the INFOCOM'02 Conference*, June 2002.

[13] Ricardo Russo, Lukas Kencl, Bernard Metzler, and Patrick Droz. Scalable and adaptive load balancing on IBM PowerNP. Research Report RZ-3431, IBM Zurich Research Laboratory, July 2002.

[14] Beverly Schwartz, Wenyi Zhou, Alden W. Jackson, W. Timothy Strayer, Dennis Rockwell, and Craig Partridge. Smart packets for active networks. *ACM Computer Communication Review*, January 1998.

[15] Michael W. Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *International Conference on Functional Programming*, pages 86–93, 1998.

[16] Jonathan T. Moore, Michael W. Hicks, and Scott Nettles. Practical programmable packets. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '01)*, April 2001.

[17] Jonathan M. Smith, Dave J. Farber, David C. Feldmeier, Carl A. Gunter, Scott M. Nettles, William D. Sincoskie, and Scott Alexander. SwitchWare: Accelerating network evolution. Technical Report MS-CIS-96-38, UPenn, CIS, 1996.

[18] Dan Decasper, Guru Parulkar, Sumi Choi, John DeHart, Tilman Wolf, and Bernhard Plattner. A scalable, high performance active network node. *IEEE Network*, January 1999.

[19] Roman Pletka and Burkhard Stiller. Adaptive end-to-end quality-of-service guarantees in ip networks using an active networking approach. In *OpenSig 2002*, October 2002.

[20] Andreas Kind, Roman Pletka, and Burkhard Stiller. The potential of just-in-time compilation in active networks based on network processors. In *Proceedings of IEEE OpenArch 2002*, June 2002.

[21] Randall R. Stewart, Qiaobing Xie, Ken Morneault, Chip Sharp, Hanns Jürgen Schwarzbauer, Tom Taylor, Ian Rytina, Malleswar Kalla, Lixia Zhang, and Vern Paxson. Stream control transmission protocol. RFC 2960, Internet Engineering Task Force, October 2000.

[22] Robert Haas, Patrick Droz, and Burkhard Stiller. Autonomic service deployment in networks. *IBM Systems Journal*, 42(1), 2003.