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

Abstract

In this paper, we present five case studies of advanced networking functions that detail how a network processor (NP) can provide high performance and also the necessary flexibility compared with Application-Specific Integrated Circuits (ASICs). We first review the basic NP system architectures, and describe 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 apply to a wider range of system architectures. Specifically, 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 (code interpretation and just-in-time compilation in active networks), and protocol stack termination (SCTP). Finally, we summarize the key features as revealed by each case study, and conclude with remarks on the future of NPs.

1 Introduction

The advent of network processors was driven by an increasing demand for high throughput combined with 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, simple network interface cards were replaced by intelligent ASICs which relieved the CP of most forwarding tasks. ASIC-based routers, however, turned out to be not as flexible as necessary in the fast and diverse network 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. In particular, a set of Application Programming Interfaces (APIs) and protocols are currently being standardized by bodies such as the Internet Engineering Task Force (IETF) ForCES working group and the Network Processor Forum.[1] This finally enables users to exploit the full range of NP capabilities: the powerful combination of performance and flexibility that allows an efficient development of advanced networking functions.

Today, a wide variety of NPs exists, each providing a different combination of flexibility, price, and performance [1]. 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 [2] network processor, it provides insight into a much wider selection of existing and forthcoming products, as the concepts discussed apply to most other NPs as well.

This paper provides insight into many of the

[1] Online at http://www.ietf.org/html.charters/forces-charters.html and http://www.npforum.org/, respectively.

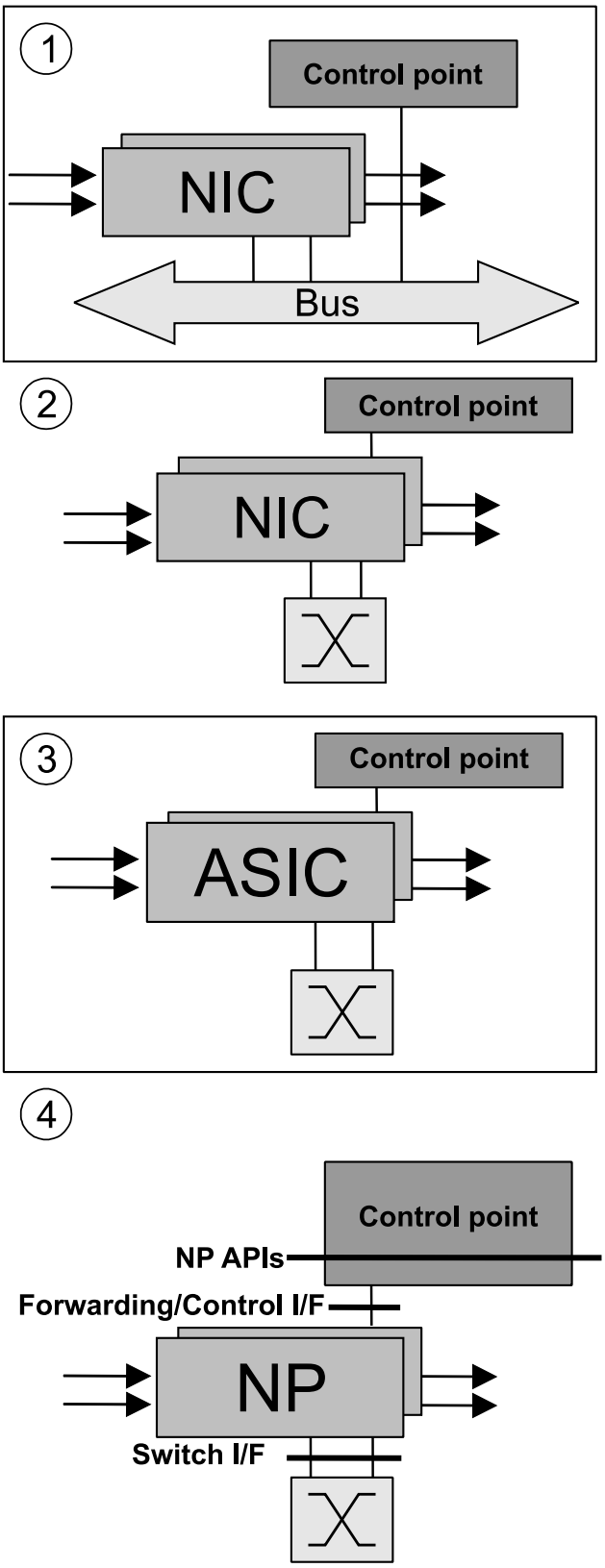

Figure 1: The advent of network processors.

enabling factors we found to be necessary during the implementation of a wide selection of functions, ranging from features of increasing significance 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. Based on our first-hand experience, we are thus able to explain how NPs simplify and help accelerate the development of this broad spectrum of functions, and describe the lessons learned. The paper is structured as follows. Section 2 introduces a small taxonomy of NP architectures and describes a sample an NP architecture in sufficient detail for the following five case studies. Section 3 states the seven goals of QoS and explains how to achieve them using active queue management and traffic engineering. Section 4 presents our experience implementing the GPRS Tunneling Protocol used for mobile Internet access. Section 5 introduces advanced server load-balancing techniques and their efficient implementation supported by NP-specific functionality. Section 6 outlines the high flexibility of NPs by discussing just-in-time (JIT) compilation of active networking code and its performance gains. Section 7 describes experiences gained from the implementation of the Stream Control Transport Protocol, which over the next few years is expected to be widely adopted for diverse applications. In Section 8, we conclude by summarizing and comparing the individual features that 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 retain a simplicity of programming similar to that of a General-Purpose Processor. NP systems are composed of multiple processors ("cores") whose organization can be classified into a serial (or pipeline) model and a parallel model (Figure 2).

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

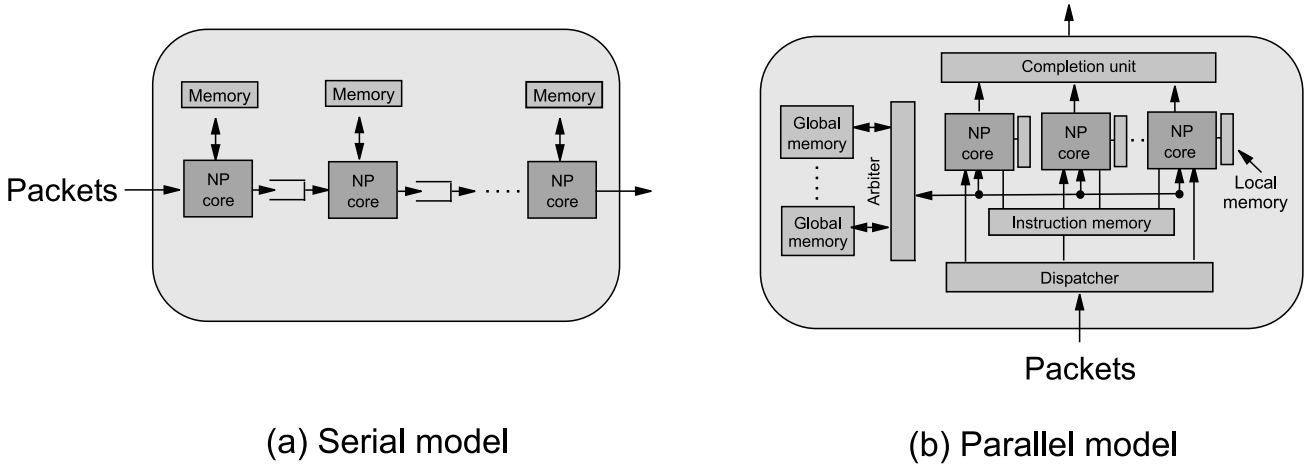

Figure 2: NP concurrency models

the packet is pipelined through a subset of NP cores, in each of which a different portion of the data-path code is applied.

From a programming point of view, the serial model requires that the code be partitioned 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 inherently is evenly distributed. To maintain comparable performance under all conditions of realistic, fluctuating traffic mixes, the serial model would require a dynamic repartitioning of the code, whereas in the parallel model no partitioning is necessary.

While most NPs are able to operate in both parallel and serial models, architectural decisions often favor one over the other and thus need to be considered in software design and benchmarking, to avoid unnecessary bottlenecks and processor stalls waiting for coprocessors, to name just a few. For example, larger applications on the Intel IXP [3] may exceed the per-core instruction memory and require serialization, distributing the code among more cores. The IBM PowerNP [2] is designed based on the parallel model. Preferences for either model may be given by state interdependence between packets (e.g., in stateful protocol termination, as shown for SCTP in Section 7) or the use of powerful semi-autonomous coprocessors such as in the Motorola C-5 or the IBM PowerNP.

The PowerNP consists of 16 concurrent NP cores that are scaled-down RISC processors running at 133 MHz, with a PowerPC-resembling instruction set, each supporting two independent threads. Multithreading keeps an NP core busy, even when one thread waits for coprocessors results. A thread entirely processes a packet, i.e., threads are in *run-to-completion* mode unless the thread explicitly interrupts processing (Section 7).

To accelerate common tasks compared with their execution in picocode (i.e., a program in a NP core), each pair of NP cores is assisted by eight dedicated coprocessors. They perform asynchronous functions such as longest-prefix lookup, full-match lookup, packet classification, hashing (all done by the two Tree Search Engines (TSE)), data copying, checksum computation, counter management, semaphores, policing, and packet memory access. In addition, the TSEs provide access to the control store consisting of several memories with different characteristics. Some of the coprocessors are shown in the lower right corner of Figure 3 Furthermore, a number of hardware assists accelerate tasks such as frame alteration[2] and header parsing.

Packet processing is divided into two stages: Ingress processing directs packets from the link to the switch, egress processing vice versa. The threads can perform either task and dynamically balance the load. Along with the packet, additional context information (e.g., output-port identifier obtained by the IP forwarding lookup) can be

[2]Hardware-assisted frame alteration can, for example, update the TTL field in an IP header, generate frame CRC, or overlay an existing Layer 2 wrapper with a new one.

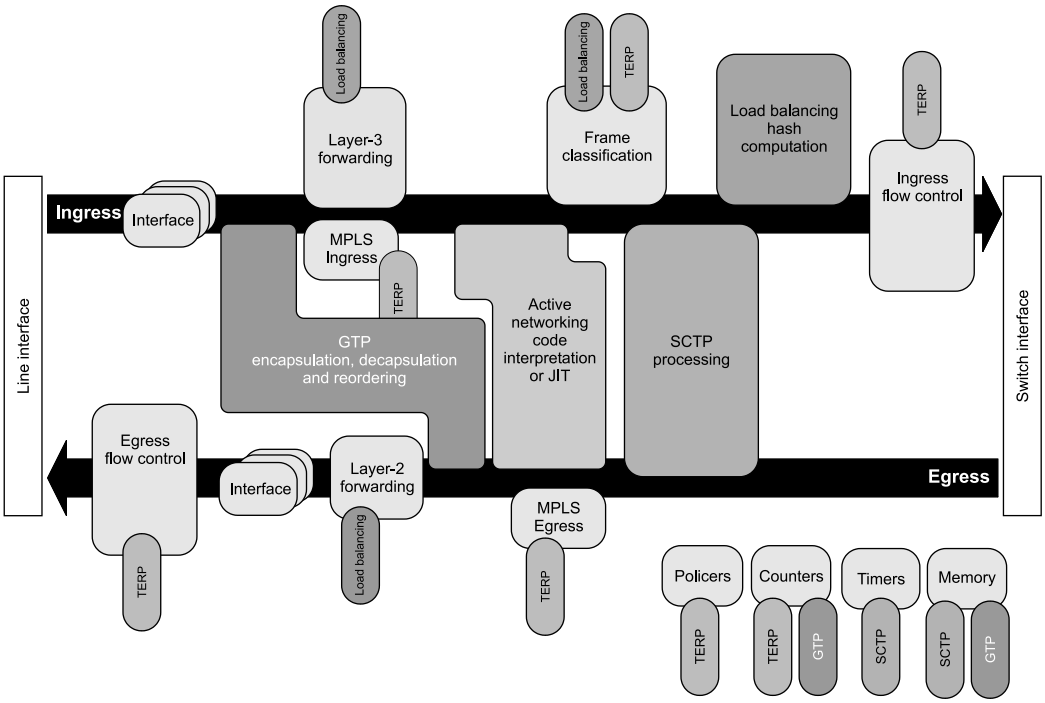

Figure 3: PowerNP programmer's view of the five case studies.

transported from ingress to egress. Note that no switch is required and that the ingress and egress NP can be the same.

NPs and CPs do not strictly map to data plane and control plane. In fact, complex data-plane operations can be shared between NP and CP. Several of our case studies also showed the converse, i.e., that NPs perform control-plane functions.

Available functions programmed in picocode and 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, a subset of which are represented by rounded rectangles along the ingress and egress data paths in Figure 3.

3 QoS Provisioning

Good QoS design differentiates both realtime and nonrealtime traffic into classes of service, and meets the following criteria.

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

2. Premium data that conforms to its bandwidth guarantees should not be dropped, and its queuing delay must be short regardless of congestion due to *Best-Effort* (BE) data traffic.

3. During steady congestion conditions, all queues should be low, ensuring a low queuing latency of the backlog. Exceptional bursts should fill the buffer, however.

4. Utilization should be high. If excess bandwidth remains after Premium traffic has been served, it should be used by BE.

5. Bandwidth allocation should be fair and predictable.

6. As congestion conditions change, the response of the NP should be fast convergence to a new equilibrium.

7. Last but not least, all of the above should be autonomic and easy to administer.

Typically, latency and loss are minimal and nearly constant, up to a certain offered load, and

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)

increase abruptly on the onset of congestion. Further considerations include finite packet life, finite storage, priority, and unpredictable duration of bursts. *Active Queue Management* (AQM) should address these issues in routers by actively dropping packets before queues overflow. The next subsection describes the implementation of an AQM system called *Bandwidth Allocation Technology* (BAT).

3.1 Active Queue Management

BAT makes use of the following NP hardware support:

- BAT 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, the *Transmit Probability table*, relieving the NP cores from individual drop decisions.
- The key indexing the transmit probability table is composed of packet header bits (e.g. DiffServ code point). The access to these header bits is supported by a *header parser hardware-assist*, which is capable of recognizing packet attributes such as priority or TCP SYN flags.
- Furthermore, detailed flow accounting information about individual offered loads is provided via *hardware counters*. Such counters as well as *queue occupancy* and *queue depletion indicators* at various locations in an NP can be used by a control theory algorithm to update transmit probabilities.

Traditional AQM schemes such as *Random Early Detection* (RED) and its variants use average queue occupancy to determine transmit probabilities. However, RED itself utilizes hand-tuned thresholds and therefore does not easily meet the above QoS criteria, especially the seventh.

BAT allows organization of traffic into pipes at each processing bottleneck (see Figure 3) [4]. A pipe is a local (per bottleneck) aggregation of traffic of one traffic class. As network end-to-end paths with guarantees are being added or deleted, network control is responsible for ensuring the guarantees of the minima at each NP in the network and fairly distributing any available excess bandwidth, which is a complex task of network control.

BAT uses control theory to adapt transmit probabilities to current offered loads. A binary signal defines the existence of excess bandwidth. This signal can incorporate flow measurements, queue occupancy, rate of change of queue occupancy, or other factors such as network congestion signals. If there is excess bandwidth, then flows that are not already at their *max*s are allowed to increase their bandwidth share linearly. Otherwise, the rates allocated to flows not already below their *min*s must decrease exponentially.

For example, let us suppose that four 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, 50, 15, and 100 (sum is 180), then (noting the max of Flow1) the correct allocation should be: 15 of Flow0, 40 of Flow1, 15 of Flow2, and 30 of Flow3. The correct transmit probabilities are therefore 1, 0.8, 1, 0.3. Moreover, the above allocation should be reached quickly and with low queue occupancy. We emphasize the QoS criteria 1–3: queuing latency during steady congestion must remain low.

By contrast, suppose the flow control were conventional taildrop (i.e., packet drop only occurs when the queue occupancy becomes full) and the egress data store were 128 Mb. BE packets forwarded to a 100-Mbps link arriving at 101 Mbps would fill up the egress data store, causing an unac-

ceptable queuing latency of 1.28 s. However, setting a low taildrop threshold would result in an unacceptable shaving of bursts. Thus QoS with taildrop may fail criterion 7.

BAT meets criteria 1–7, achieving high utilization, fast convergence, fairness, and administrative simplicity [4]. The latter advantage is by virtue of the fact that the AQM system is configured with minimum and maximum flow rates and priorities rather than queue levels.

The implementation of BAT in the data plane of the PowerNP is straightforward because the header parser hardware-assist and hardware flow-control support can be configured according to the new scheme. No additional processing is needed in the data plane. The execution cost in the control plane is less than 2 μs ($<$ 267 cycles) to update a single transmit probability value. This update is triggered every 80 μs in ingress and every 8 ms in egress. Therefore even running 32 ingress pipes and 2K egress pipes would occupy only about 8% of the computational power of all NP cores.

3.2 Traffic Engineering Reference Platform (TERP)

We now highlight how the NP is used in the context of Traffic Engineering (TE), from both the data plane and the control plane point 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 DiffServ 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 switching 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 Figure 3 are configured by the CP using a Label and Resource Manager (LRM) process: Ingress MPLS nodes perform traffic classification, policing, and DiffServ marking, whereas transit nodes perform MPLS forwarding (MPLS label swapping),

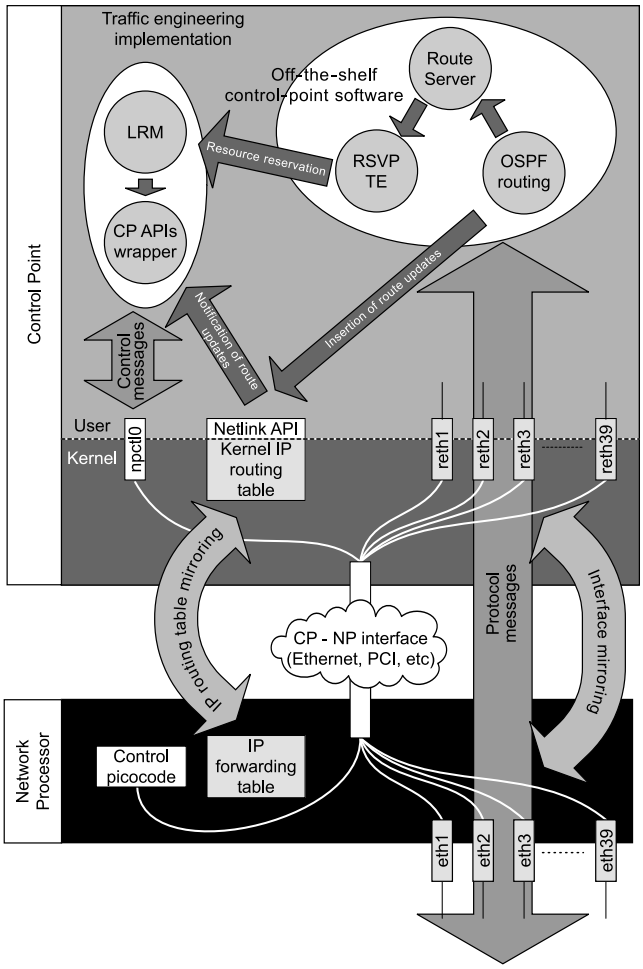

Figure 4: TERP control-point architecture.

and egress MPLS nodes perform IP forwarding. All nodes can be configured to use WRED or BAT AQM to perform bandwidth allocation.

In the control plane, the CP runs the RSVP signaling, OSPF routing, and possibly the Router Server processes. As shown in Figure 4, each NP Ethernet port (*eth1* to *eth39*) is mirrored in the CP as a normal interface (*reth1* to *reth39*): CP processes can therefore send and receive packets (protocol messages) as if the MPLS node were built as a centralized software-based router. In addition, the CP kernel's IP routing table is mirrored automatically into the NPs. OSPF routing therefore operates completely transparently from the underlying NP architecture; it uses the Linux Netlink API to insert routes into the table. These route updates are reported to the CP-APIs-wrapper process that creates the appropriate control messages to automatically insert these routes into the NP. RSVP interfaces with the LRM to perform resource reserva-

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

tion, i.e., to reserve, commit, and release resources as needed in each node. The LRM performs CP APIs calls that are then translated into control messages destined for the NP.

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

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 because GPRS deployment is still in process.

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 our early prototype supports 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 is stripped at the destination GSN, and normal IP forwarding is applied to the decapsulated packet. In encapsulation and decapsulation, traffic counters associated with the context are asynchronously incremented to account for the data transmission.

As shown in Figure 5, the implementation of the GPRS extensions to the NP 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.

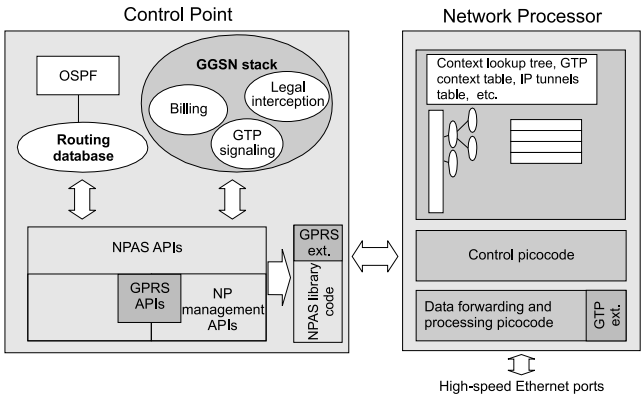

Figure 5: GTP extensions to the NP software.

For tree, table, memory block, and counter management, generic CP APIs can be used that 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

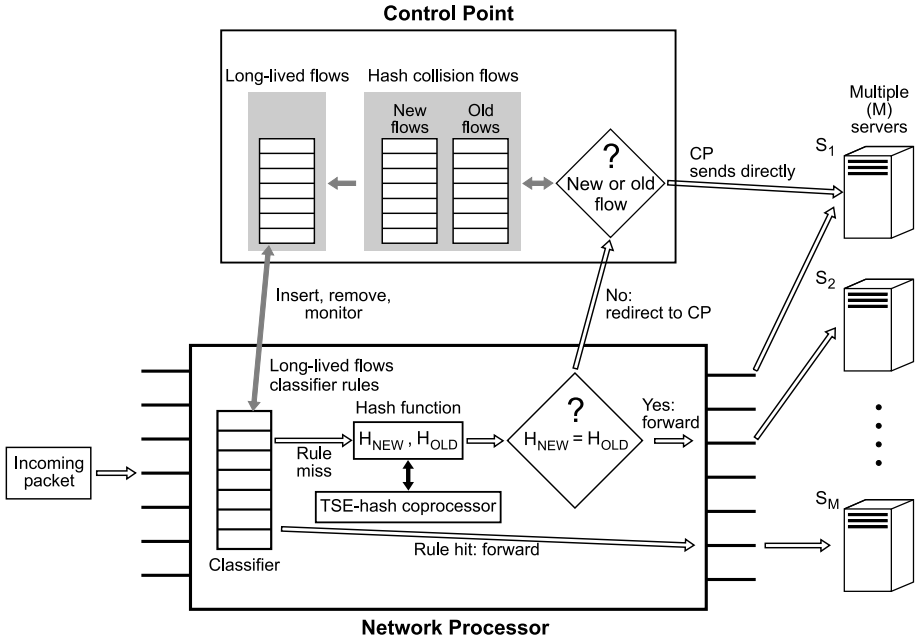

Figure 6: Server load balancer on the NP: data path diagram.

context associated with an incoming packet. Counters for traffic accounting are incremented asynchronously 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 a frame spends in ingress and egress processing for the tunneling tasks, including stall cycles spent waiting for coprocessor results or memory and instruction 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 distributing the data-processing tasks among multiple servers or processing units. The task of adaptively balancing the load among multiple servers is nontrivial because of the high volume and unknown characteristics of the traffic and the need to maintain connectivity of active packet flows between hosts. To balance a large number of flows simultaneously, it is necessary to minimize the amount of state information stored.

The adaptive load-balancing method (Figure 6) uses a hardware-based hash function to determine the destination server. The hash function is based on the robust hash routing algorithm [5], which supports arbitrary processing capacities of the balanced servers, and on its adaptive extension [6] that 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 the stable state, in which only one hash function is computed, and a transient state with two hash calculations.

Initially, a single hash function is configured based on the resources available on the servers. During operation, if some servers become overloaded while others still have capacities, hashing is adapted to distribute network traffic optimally based on the statistics gathered.

During this transition, 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 ($H_{\text{new}} = H_{\text{old}}$) continue to be routed to the resulting server. Packets that do not fall in the intersection of the two hash functions ($H_{\text{new}} \neq H_{\text{old}}$) are redirected to the CP for routing. The CP determines and keeps track whether the flow is old or new, and routes ac-

cording to the appropriate hash function. After a configured period of time, multi-field classification rules are installed for the remaining ("long-lived") flows in the state table until the flows terminate or time out. The rules avoid breaking the flow in future transitions and allow the return to a single hash function.

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

Advantages of this approach include the following:

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

A number of software and hardware components of the PowerNP have been used in prototyping the load-balancer application; among them are 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 reduced the data-path programming on the NP 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 built into the TSE coprocessor eliminated the necessity to implement a hash function.

The number of processor instructions required to execute the hash-routing method on each packet depends on the number of balanced servers, m. The instructions are primarily dedicated to reading and executing operations on the per-server weights, while calling the TSE coprocessor in parallel. For $m \leq 8$, the prototype implementation executes $20m + 75$ instructions. For $m > 8$, the number of instructions executed grows logarithmically with m, as the weights table is then organized into a tree structure.

6 Payload Processing: Active Networking

Two main approaches dominate Active Networking (AN) research: The *capsule approach* embeds active code into data packets that is executed on each node along the path. The *programmable switch* approach maintains the existing packet format. Programmability is provided through dynamic downloading of programs, which are then executed upon arrival of packets matching a filter.

Combining the capsule approach with a byte-code language possessing intrinsic safety properties [7] leads to an architecture-neutral compact code suited for both parallel and serial NP models. The execution environment provides a virtual machine that can be implemented in the data plane of NPs.

The resulting framework provides the flexibility and safety required in active networks. Although the approach is application-neutral, our AN work is motivated by the fact that no end-to-end QoS guarantees can be given in IP networks today. AN shifts 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. Protocols, unlike active code, are not powerful enough to provide translation between existing QoS frameworks.

The bottleneck of byte-code interpretation can be overcome by just-in-time (JIT) compilation of active code [8]. Our work on the IBM PowerNP showed that native execution is more than an order of magnitude more efficient than interpretation. We also observed that the compilation speed almost matched interpretation. Accordingly, already small amounts of reuse quickly result in performance gains. Reuse comes as loops or function calls, but also by caching compiled code either at routers or inside the packet, as a service to subsequent NPs.

We implemented and tested a general active net-

work setup based on a dialect of the SNAP active networking language [7] on the PowerNP. Results show that JIT compilation is not only feasible in an active router because the compiler is sufficiently small and fast to run in the data plane's NP cores, but also leads to significant performance improvements. Figure 7 compares three different types of active packets.

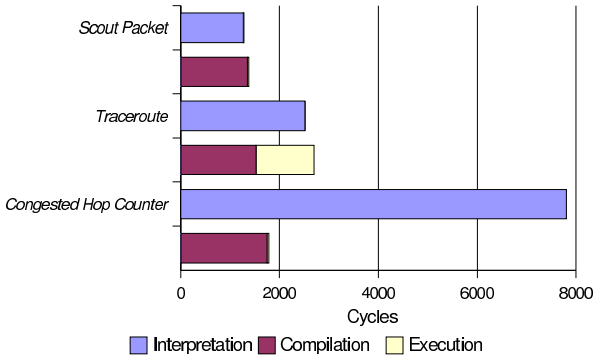

Figure 7: Execution cycles for active code on a NP.

The *scout* active packet discovers the list of active routers between the source and destination and is 22 byte-code instructions long, including bounds checks. Although compilation and execution do not outperform interpretation, the remarkable cycle cost of execution readily 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. The traceroute packet consisting of 26 byte-code instructions; no bounds checking is needed, as information is sent back immediately. The execution time becomes visible as a result of the high cost of packet creation.

The third active packet is a *congested hop counter* in 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 programs described above, this one uses a loop to accumulate the data, resulting in a fourfold speedup.

Critical for the generic applicability of JIT compilation in active networks are unrestrained write access to the instruction memory and a sufficiently large instruction memory to hold the compiler and the JIT-compiled active code; some NPs fail these requirements.

7 Protocol Termination: SCTP

Offloading of transport-level protocol processing from the end-system host CPU is a technique that is receiving increasing attention. On the one hand, this off-CPU "protocol termination" helps to relieve overloaded end systems such as servers from processor-cycle-consuming protocol processing, especially at emerging multigigabit line rates. A typical example of this technique is the development of TCP offloading engines (TOEs) located on intelligent network interface cards that perform TCP processing on dedicated hardware.

On the other hand, moving protocol processing further into the network 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 off-the-shelf end-host networking stacks, 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. The termination of a typical stateful, reliable transport protocol such as TCP poses the following challenges:

- A per-connection state context must be maintained efficiently.
- A timer service for protocol timers must be offered.
- Segmentation and reassembly (SAR) and retransmission services need per-context intermediate packet buffering.
- The often limited instruction memory size conflicts with the extensive functionality of some protocols.
- The protocol-termination environment should provide a clean API to allow seamless and efficient integration of applications such as a firewall or a load balancer.

A challenging example is the termination of the SCTP protocol on the PowerNP. The SCTP protocol was chosen because it combines advanced protocol features such as multihoming, multistreaming, partial message ordering, and cookie-based association establishment with traditional reliability and robustness requirements.

A first prototype on the PowerNP implements full SCTP termination and already includes multi-streaming and multihoming. It offers a socket-like API to link to the envisioned applications at the picocode level. The implementation provides sufficient 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. Given the model of several parallel, run-to-completion threads, each operating on a specific SCTP context at a time, the use of a semaphore-based context-locking mechanism is required. The run-to-completion model, on the other hand, allows the implementation of a natural, event-based code path. Possible events are incoming packets, timer expiry, and application downcalls.

Except incoming packet checksum verification, the entire SCTP code operates on packets in the egress data store of the NP. This provides the proper amount of packet memory required to store data for send and receive windows, SAR, and packet retransmission. Preservation of data sequence within a stream requires packets to be stored until all prerequisite packets have been received. Even though the PowerNP favors the run-to-completion model, our implementations of SCTP and GTP show that waiting for missing data can easily and efficiently be achieved without blocking any threads.

Although the NP-based SCTP stack was designed to extend the functionality of an intermediate system, it can also be employed within an intelligent network adapter to provide host-CPU protocol offloading.

Table 3: Feature requirements

Case	Key features and accelerators
BAT	Probability operations Many per-flow queues Timers
TERP	Policing, flow-control, scheduling, and Layer-3/4 classification Packet forwarding to CP State synchronization NP $\leftrightarrow$ CP
GTP	Millions of exact-match classifier rules Concurrent counters Frame size alteration (pre-/appending) Large packet storage for reordering
Load balancer	Collaboration NP $\leftrightarrow$ CP Fast hash of disjoint header fields Uniformly spreading hash function Multi-Field Classification (exceptions) Fast update of MF Classification rules
Active networks	Direct write to instruction memory Access to forwarding information Programmable per-packet forwarding Program-controlled multicast Payload processing
SCTP	Collaboration NP $\leftrightarrow$ CP Fast CRC Frame-size alteration (pre-/appending) Large packet storage for reassembly and reordering Mutual exclusion Scalable per-flow timer support Payload processing

8 Conclusion

The case studies presented here help us visualize and understand several driving factors behind NPs. Figure 3 summarizes the configurations of existing logical blocks and/or new logical blocks with their own CP APIs for each case study.

The case studies first 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 current NPs cover such a wide range of the design space. Depending on the applications envisioned by the designers, different decisions and compromises had to be implemented.

One of the open issues in the NP space remains software durability, which NPs 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 whenever new family members appear.

Fortunately, this situation is changing for the better. For code running in the data plane, the use of a smart optimizing compiler permits relatively architecture-independent code to be written. With appropriate, evolving libraries, key subfunctions can be transferred progressively and seamlessly from software implementation to hardware, maintaining backward compatibility. In the control plane, standard interfaces are being developed and joined with capabilities for dynamic NP feature discovery to allow NP-independent code. Working groups such as IETF ForCES and NP Forum are developing the relevant protocols and semantics, allowing key performance functions to be easily offloaded from the CP onto the NP.

The implementation of only a subset of the examples presented in this paper on an ASIC 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, NPs speed up the development of both control and data path thanks to higher-level interfaces and pre-existing, reusable building blocks.

Taking a step back to see the big picture, we can conclude that versatility is probably the strongest NP characteristic. With comparably little effort, it is possible to implement new features to dramatically increase the value offered by a network device, and to offer new and powerful functions, often combined from 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 can adapt themselves and provide new, fast routing concepts [9] and which in turn will call for new methods to achieve efficient deployment of services in such networks [10]. The unique combination of flexibility and power of NPs already now can help pave the way towards the versatile and fast evolving networks of the future.

References

[1] Linley Gwennap and Bob Wheeler, *A Guide to Network Processors*, The Linley Group, Mountain View, CA, USA, 2001.

[2] 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*, vol. 47, no. 2-3, pp. 177–194, 2003.

[3] Muthu Vekatachalam, Prashant Chandra, and Raj Yavatkar, "A highly flexible, distributed multiprocessor architecture for network processing," *Computer Networks*, vol. 41, no. 5, pp. 563–586, 2003.

[4] Ed Bowen, Clark Jeffries, Lukas Kencl, Andreas Kind, and Roman Pletka, "Bandwidth allocation for non-responsive flows with active queue management," in *IEEE International Zurich Seminar on Broadband Communications, IZS 2002*, Feb. 2002.

[5] Keith W. Ross, "Hash routing for collections of shared web caches," *IEEE Network*, vol. 11, no. 6, pp. 37–44, Nov./Dec. 1997.

[6] Lukas Kencl and Jean-Yves Le Boudec, "Adaptive load sharing for network processors," in *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '02)*, June 2002, pp. 545–554.

[7] 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)*, Apr. 2001, pp. 41–50.

[8] 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, pp. 79–90.

[9] Marcel Waldvogel and Roberto Rinaldi, "Efficient topology-aware overlay network," in *Proceedings of ACM HotNets-I*, Oct. 2002.

[10] Robert Haas, Patrick Droz, and Burkhard Stiller, "Autonomic service deployment in networks," *IBM Systems Journal*, vol. 42, no. 1, pp. 150–164, 2003.

Robert Haas is a Research Staff Member at the Communication Systems Department of the IBM Zurich Research Laboratory. His current research interests focus on programmable networks and techniques to automate service deployment in such networks. He received the degree in communication systems engineering from the Swiss Federal Institute of Technology (EPFL) in Lausanne, Switzerland, together with the Eurécom Institute, Sophia-Antipolis, France, in 1996, and then joined the IBM T.J. Watson Research Center. He is currently completing his Ph.D. work with the Swiss Federal Institute of Technology (ETH), Zurich.

Clark Jeffries earned a Ph.D. from the University of Toronto, Canada, in mathematics. He joined Clemson University in Sout Carolina, USA, as lecturer in 1987 and rose to professor in 1994. Following a pleasant sabbatical with IBM, he joined that company to help design NPs in 1998. Current interests include security features efficiently enabled in NPs.

Lukas Kencl was working toward his Ph.D. degree while at IBM Research, Zurich Research Laboratory. His research interests include load-sharing algorithms, router architecture, network processors, and computer networking in general. Kencl received a Master's degree in computer science from the Charles University, Prague, Czech Republic, in 1995, and a Ph.D. degree in communication networks from the Swiss Federal Institute of Technology (EPFL) in Lausanne, Switzerland, in 2003. He is now a senior researcher at the Intel Research Laboratory in Cambridge, UK.

Andreas Kind joined the IBM Research, Zurich Research Laboratory in 2000. Before, he worked at the C&C Research Laboratories of NEC Europe and received his Ph.D. from the University of Bath, UK. His interests include active queue management, traffic profiling, programmable networks, distributed computing, and programming languages.

Bernard Metzler is a Research Staff Member at IBM Research, Zurich Research Laboratory. His research interests include the design and implementation of efficient and flexible communications stacks and high-speed internetworking. Metzler received a diploma degree in electrical engineering form the Humboldt-University Berlin and a Ph.D. from the Technical University of Braunschweig, Germany.

Roman Pletka graduated in electrical engineering at the Swiss Federal Institute of Technology (EPFL) in Lausanne, Switzerland, in 1996 and continued his studies in communication systems there. He spent his final year at Eurécom Institute in Sophia Antipolis, France, where he majored in corporate communications and received his diploma degree in communication systems engineering in 1999. He is currently pursuing his Ph.D. degree at the Swiss Federal Institute of Technology (ETH) Zurich on adaptive end-to-end QoS guarantees in IP networks. Since 1999, he is with IBM Research, at the Zurich Research Laboratory in Rüschlikon, Switzerland.

Marcel Waldvogel is a Research Staff Member at IBM Research, Zurich Research Laboratory. His research interests include high-performance advanced routers, IP lookup and classification algorithms, multimedia data distribution, distributed data retrieval, network security, and denial of service. Waldvogel received a diploma degree in computer science and a Ph.D. in electrical engineering, both from Swiss Federal Institute of Technology (ETH), Zurich. He also enjoyed serving on the computer science faculty at Washington University in St. Louis, Missouri, USA, and is a senior member of the IEEE.

Laurent Freléchoux was a Research Staff Member at IBM Research, Zurich Research Lab-

oratory. His research interests include networking and network processors with a focus on mobile and wireless networking. He received a degree in computer science from the Swiss Federal Institute of Technology (EPFL) in Lausanne, Switzerland in 1996 and is currently undertaking an MBA at IMD, Lausanne.

Patrick Droz is a Research Staff Member at the IBM Research Laboratory in Zurich, Switzerland. His current focus is on high-speed communication protocols in the context of network processors, and he actively participates in relevant standards activities such as the IETF. He holds a Ph.D. in computer science from Swiss Federal Institute of Technology (ETH) Zurich. His previous experience include ATM/PNNI control point design and implementations as well as ATM/IP integration with many contributions to the ATM Forum and the IETF.

Direct questions and comments about this article to Robert Haas, IBM Research, Zurich Research Laboratory, Säumerstrasse 4 / Postfach, CH–8803 Rüschlikon, Switzerland; rha@zurich.ibm.com. For additional information, please visit http://www.zurich.ibm.com/cs/networking_sw/.