Design Methodology for a Modular Service-Driven Network Processor Architecture

Maria Gabrani, Gero Dittmann, Andreas Döring,
Andreas Herkersdorf, Patricia Sagmeister, Jan van Lunteren

IBM Zurich Research Laboratory
{mga, ged, ado, anh, psa, jvl}@zurich.ibm.com

Abstract

We present a design methodology for a modular network processor architecture that leads to a balanced, service-defined mix between programmable processor cores, configurable hardware assists, and specialized coprocessors. Whereas the processor cores address the flexibility and extendibility needs of the networking market, the hardware components offload the processors, or even allow them to be bypassed for certain network processor-typical tasks to optimize chip area, performance, and power efficiency. We describe the rationale behind the selected functional partitioning in hardware and software components and discuss the challenges of designing the hardware components, and of organizing and integrating the programmable cores. We quantify our approach with a performance evaluation of the overall system.

Key words: Network processors, modular and scalable architectures, systems on a chip, open systems, performance evaluation.
PACS: 84.40.U, 85.40, 07.05.B, 07.05.T

1 Introduction

The networking market demands ever-higher data rates and at the same time support for the continuously changing networking standards and applications. Networking equipment vendors seek to reduce time-to-market and to decrease development costs and power consumption. Networking equipment customers desire increased time-in-market, higher performance and low power consumption. Support of new standards and applications, increased time-in-market and reduced development costs imply programmable solutions and have led to

the introduction of network processor units (NPU). Higher data rates, better performance and reduced power consumption imply hardware solutions that correspond to traditional switch and router designs. To satisfy the diverse, and in some cases conflicting, requirements we pursue a hybrid solution. Whereas flexibility and programmability needs are met with the use of processor cores, hereafter called CPUs, performance requirements are fulfilled by incorporating a number of deterministic performance (e.g. hardware) components. Aiming for a system-on-a-chip (SoC) modular architecture, the question we address is how to incorporate the above components in a most efficient and effective way.

We suggest a design methodology for a NPU architecture that leads to a modular, balanced, service-driven mix between CPUs and components with deterministic performance, which satisfy the next two attributes. First, they should perform certain functions, the results of which can be used by the CPUs. Second, they should provide the necessary functionality so that certain types of protocol data units (PDUs, e.g. packets, frames, cells) need not be processed by the CPUs. The second attribute implies that the components perform data plane functions. The data plane path formed by a set of deterministic performance components that satisfy the above two attributes is called *service-driven data flow* (SDF).

Hereafter we discuss the rationale behind the introduction of the SDF concept, as well as the challenges encountered and solutions developed for the design and integration of certain system components. We start in Section 2 by describing the requirements of an NPU and how they give rise to the service-driven mapping into the system's components. In Section 3 we discuss the requirements and challenges of designing two representative types of SDF components. The implications in integrating a cluster of CPUs in the proposed architecture are presented in Section 4. The methodology designed to evaluate the system's performance is the focus of Section 5. We conclude the paper in Section 6 with a summary and directions for further study.

2 Service-driven NPU architecture

Network processors perform central functions in network elements (NEs) such as traditional IP routers or LAN/WAN switches, as well as in emerging systems such as firewalls and DSLAMs (digital subscriber line access multiplexers). Therefore, before designing a specific NPU architecture, it is important to understand the functional and performance requirements of such systems.

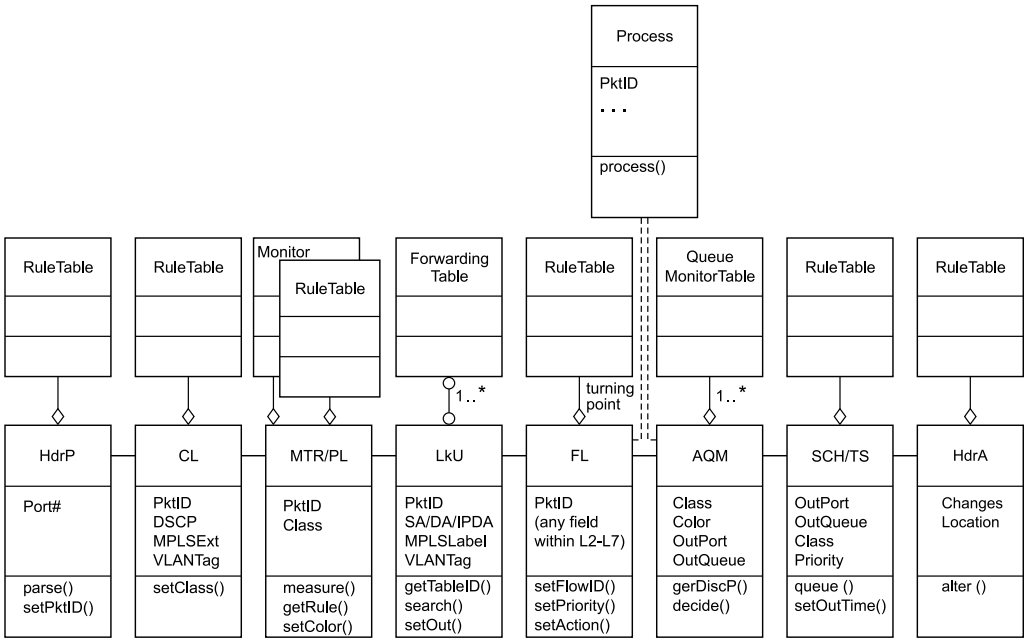

Figure 1. Example of a DiffServ-compliant data-plane functionality of an enterprise or edge NE. Access to a rule table is used here to indicate the flexibility of a function to handle various cases of traffic in a user-defined way.

2.1 Functional requirements

The requirements of an NE —both in functionality and performance— are dependent on the target market and application area. The basic functionality of edge and core NEs is Layer 2 switching and IP routing. This basic functionality is extended with quality-of-service (QoS) traffic management in order to handle delay and jitter-sensitive traffic [6,26,28,29]. Congestion is avoided with an active queue management (AQM) function [7,14,27]. The functionality of enterprise NEs may be further enhanced with security, network address translation (NAT), or other application-specific support.

An example of a differentiated services (DiffServ, [11,28]) compliant functionality of an enterprise, edge or core NE is illustrated with a UML-like graph in Figure 1. In such an NE, upon receipt of a PDU, the following actions occur. The header of the PDU is parsed (HdrP) and the type of the PDU is identified (PktID) to decide which forwarding table to search (e.g. virtual local area network (VLAN), multiprotocol label switching (MPLS), L2/L3). The PDU is classified (CL) to establish the flow and class of the PDU and possibly its priority. Next, the metering (MTR) function checks whether the PDU's flow conforms to its agreements. The policing (PL) function "colors" the PDU according to its flow's behavior; the color may change the priority of the PDU ((re)marking, not shown in the figure). The output coordinates of the PDU are looked up (LkU). A filtering (FL) function determines whether

the PDU requests illegal or unauthorized access. Typically at this point it is decided whether the PDU is merely forwarded or needs further processing (e.g. load balancing or TCP splicing). We call this point of the functional pipeline a *turning point*. Further processing may alter the result of the first LkU. If no further treatment is required then the AQM function decides whether to forward or discard the PDU according to the output queues' fill levels and the PDU's class and/or priority and possibly its color. The scheduler (SCH) ensures that the PDU's transmission is prioritized according to its relative class/priority. The traffic shaper (TS) ensures that the PDU is transmitted according to its QoS profile and/or agreements. The header of the PDU is changed (HdrA) appropriately, and the PDU is transmitted. Let us call the above functions that provide QoS-based forwarding *standard functions*.

The services of access market NEs can be quite diverse because they are highly dependent on their application area, such as cable, storage and digital subscriber line (DSL) systems, and radio access networks. For example, in radio access networks, PDU sizes are usually very small and bandwidth is scarce, therefore NEs (e.g. base stations) may perform header compression (HC) to enhance bandwidth efficiency [1]. Hence, when a PDU is identified as header-compressed, it is first decompressed (HD) and then QoS-based forwarded (Figure 2). Before retransmission, depending on the output link's speed, the PDU header may be compressed. Note that the new service (HC) can be built on top of the standard functions. This observation indicates that the sequence of the standard functions is not "frozen" and that both the function modules and their interfaces have to be well defined. Note also that not all the PDUs traversing an NE need to be processed by the service specific functions, i.e. compressed/decompressed in our example. This note indicates a mixed traffic pattern.

The set of standard functions may vary in order and type from system to system, depending on the implementation, the NE's position in the network and the application area. Nevertheless, in an NE a set of standard functions comprises the *minimum service* required (here QoS-based forwarding) and any additional service can be built on top of this by appropriately locating and interconnecting the new *service-specific functions*, typically introducing the deep-packet processing at the turning point (Figure 3a).

2.2 Performance requirements

The basic performance requirements of NEs are scalability, throughput and low power consumption. Typically, scalability is ensured via a distributed forwarding architecture. Such an architecture offloads QoS requirements from

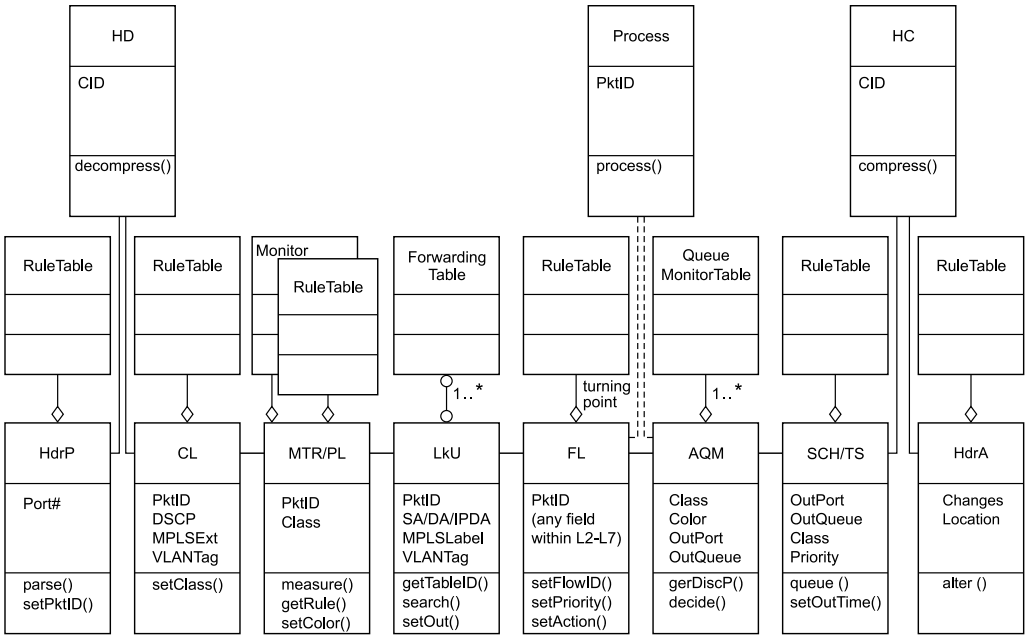

Figure 2. Example of a data plane functionality of a radio access network NE.

the switch fabric by distributing it to the line cards, to which the NPUs are targeted. Scalability also calls for modular design.

The throughput of a system, that is the number of PDUs per second that can be forwarded by the system without loss, can range between multiple 100 K packets per second (Kpps) and 25 Mpps, depending on the NE application area. The throughput can be boosted in a work-conservative system by increasing the number of buffers and the buffer bandwidth, and decreasing the access time to the buffers. The provision of enough buffers allows the PDU to be stored after it has been policed, filtered and its output determined. It is preferable to use off-chip memory for the bulk storage requirements ($N_1 \times 10$ MB at $N_2 \times 10$ ns) and on-chip memory for fast-path PDUs ($N_3 \times 100$ KB at $N_4 \times$ns), where $N_1, \ldots, N_4$ are single-digit integers. On-chip memory, moreover, impacts throughput in the case of bursts of small PDUs and multicast flows. The required aggregate memory bandwidth, as a general rule, ranges between 4 to 10 times the aggregate link rate.

PDU latency and jitter can be kept tightly bounded if on-chip memories are used and functions in the data path are performed with deterministic and low latencies, e.g. if they are hardware assisted. The utilization of application-specific integrated circuits (ASICs), moreover, ensures lower power consumption and occupies less area.

The above requirements along with the programmability needs of the networking market constitute the main drivers in any NPU design.

2.3 Current solutions

In the recent history of NPUs, a variety of solutions has been introduced [2,3,13,16,20]. From a system architecture perspective, the commonality among these solutions is restricted to the use of multiple processor cores for data plane processing. Beyond that, the approaches show a great degree of architectural diversity.

Single-chip multiprocessor pipeline architectures with minimal hardware assists (e.g. hash function) [16] provide a high degree of flexibility, but require that the entire networking function be implemented as pure software process. Thus, the attainable wire-speed performance of the NPU is tightly linked to the code path length of the application. Multi-chip solutions with service-specialized processors enhance performance and reduce power consumption at the expense of flexibility [3,13]. In addition, there are single-chip hybrid solutions that blend programmable processor resources with functionally mapped hardware components to balance performance, flexibility, power and area [2,20]. In conclusion, it can be stated that no industry standard NPU architecture has been established yet. This is not surprising considering the youth of the discipline.

Taking a more abstract view of the current solutions one may surmise that modularity is achieved only on the chip and the software level. Our aim is an intra-chip (SoC) modular solution. Looking beyond the processing elements and multiprocessor clusters, we argue that the value proposition of next-generation NPUs is not focused solely on the processing engines of an NPU, but consists equally of the ability to compose an NPU out of modular, standard and service-specific components that can be mapped on a variety of implementation alternatives ranging from hardwired and configurable logic, to ASIP, to software processes running on a CPU.

2.4 Service-driven mapping into system's components

To derive a modular, hybrid, SoC NPU architecture we started with the end application in mind, performed a functional decomposition and made the following observations. First, an NPU is targeted for line cards, the functionality requirements of which include QoS-based forwarding and support of functional extensions. Among the functions that a PDU traverses in an NE ([9,11], Figures 1, 2) several are standards-defined, e.g. CL, MTR, PL, SCH, and HC, which implies that their operation and interaction with other functions are well defined. These functions span both standard and service-specific functions (Figure 3(a)). Note also that certain well-known cases of traffic can be

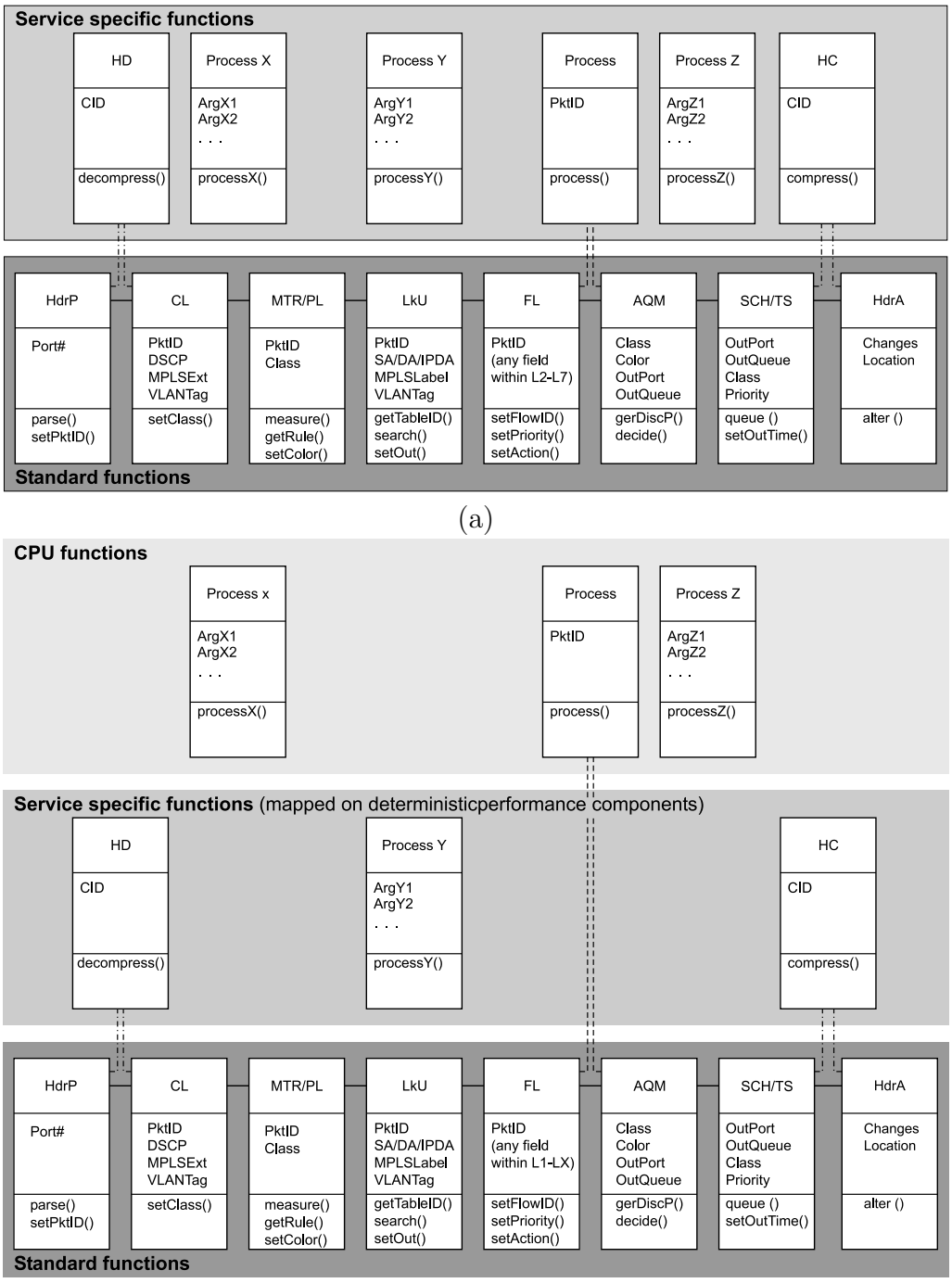

Figure 3. Layering of functions in an NE: (a) standard and service-specific functions, (b) further distinguishing the service-specific into CPU and deterministic performance functions. The functions belonging to the bottom two layers of (b) are candidates for SDF implementation.

handled fully via standard functions (e.g. switching, QoS based forwarding), which cover some of the traffic in any NE. Moreover, certain functions may be used for different purposes in a user-defined way. For example a filter can be used for security, for selecting different processing paths within the system, or for applying different rules to other functions, e.g. classification. Finally, network functions of all OSI layers frequently require common processes, such as LkU, checksum calculation and HdrA.

From the above observations one might deduce that standard functions could also be called by the CPU(s). Moreover, standard functions and standards-defined, service-specific functions are the best candidates for implementation on deterministic performance (latency, jitter, power) components. Such components may be ASICs, specialized coprocessors or ASIPs. The rest of the functionality is mapped on CPUs. This association further splits the service-specific functions into CPU and deterministic performance functions (Figure 3(b)). Finally, the appropriate interconnection and interfacing of deterministic performance components allow them to accommodate fully a sizable percentage of traffic.

Based on the above resolutions we infer an architecture that is a service-defined mix between CPUs and deterministic performance components, with the latter forming a service-driven data flow (SDF). We define as *CPU bypass* the sizable amount of traffic that can be fully processed by the SDF, and as *CPU offload* the attribute that the results of the SDF components can be used by the CPU. We call such an NPU architecture a *service-driven architecture*.

A generic implementation of such an architecture is depicted in the form of a block diagram in Figure 4. The main parts of the architecture consist of the SDF, the processor complex, the control point, memory and interconnect. The SDF supports the minimum service of QoS-based forwarding. If the PDU requires further processing it is sent to a CPU. The SDF functionality can be expanded via on- or off-chip coprocessors to accommodate different application area requirements (service-specific functions). This expansion is accomplished via the interconnect and an extension interface. Finally, the system provides a number of both on- and off-chip buffers to increase throughput and reduce latency. The buffers are managed by the buffer manager (BM). Latency and jitter are kept bounded for bypass PDUs, and minimal for non-bypass PDUs.

The main contribution of the proposed design methodology is the methodical and coherent way in which the functional partitioning is carried out, which defines the rules for the service decomposition and mapping as well as the relationship between the components. The outcome of this is an NPU architecture which is a balanced mix between CPUs and deterministic performance components. Whereas CPU-offload has been used in [2,20] by functionally mapping

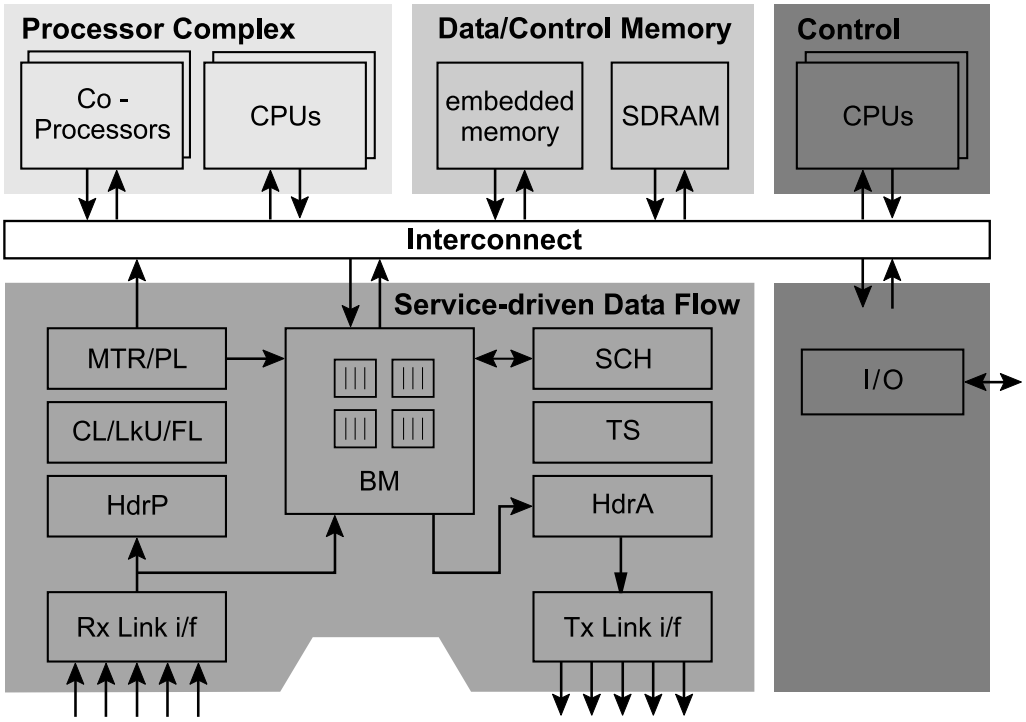

Figure 4. Generic block diagram of a service-driven NPU architecture.

certain system components, it is the functionally complete and sequentially correct PDU traversal that enables CPU bypass.

Such an architecture generates a number of requirements and implications for the design of the system's components. SDF components cover basic or computationally expensive functionality, however their design should allow flexibility and extendability. In Section 3.1 we present the challenges and requirements of such a design using the example of the LkU (or search) function.

No matter how configurable or programmable the standard function components of the SDF, new functions might be introduced at any time that may not be fully covered, and are computationally expensive to run in a CPU. We address future proof by incorporating in the SDF specialized processors that are optimized to handle certain clusters of functions. We develop a design methodology for such components and present the challenges and requirements thereof in Section 3.2. We provide further programmability and extendability via the CPUs. However, the CPUs must be extended and organized such that they can make optimal use of the SDF capabilities of the system, as we discuss in Section 4.

Finally, we strive for a modular, scalable, SoC NPU architecture. In order to verify the concept and get feedback regarding bottleneck identification, we developed the design evaluation methodology described in Section 5.

3 SDF components

The aim of this section is to convey the unique features and requirements for the design of an SDF component. To that end, we discuss two representative topics: the design of a standard function component, and an ASIP design methodology.

3.1 Generic search engine

A search engine (or LkU) is one of the standard functions encountered in any NPU that can also be called by the CPU. Therefore, it must be able to perform all searches corresponding to the applications that the NPU is intended to support, and it must do this for a variety of link speeds. These searches can involve a single field in the packet header (e.g. a routing table lookup), multiple fields (e.g. firewall and QoS applications), and even the packet payload (e.g. content-aware applications). Important performance parameters are the search rate and latency, the storage requirements and the update performance.

The rapid increase of link speeds, which has been much faster than the improvement of SDRAM performance and that of other memory technologies over the past several years, has posed two major challenges to new search engine designs:

(1) Available *memory bandwidth* has to be used more efficiently: search and update operations must perform fewer memory accesses and exploit memory system characteristics (e.g. burst modes) to improve bandwidth utilization.
(2) Available *memory capacity* has to be used more efficiently. Only in this way can faster but more expensive memory technologies such as SRAM and on-chip DRAM be used to realize cost-efficient search engines that provide wire-speed performance for higher link speeds such as 10 and 40 Gbps.

The only way to meet these challenges appears to be by means of dedicated yet generic hardware support; a purely software solution is not able to achieve the required levels of performance for the higher link speeds. For example, for a 10 Gbps link a total of approximately 125 M searches per second will be needed, assuming a minimum packet size of about 40 bytes and a total of 5 searches per packet. For this reason, there has been an increased focus in recent years on the development of hardware-oriented search algorithms. One of these schemes is the BART (balanced routing table) search scheme, which was created to search large routing tables at a speed of 10 Gbps and beyond using state-of-the-art CMOS technology, but can also be used for any

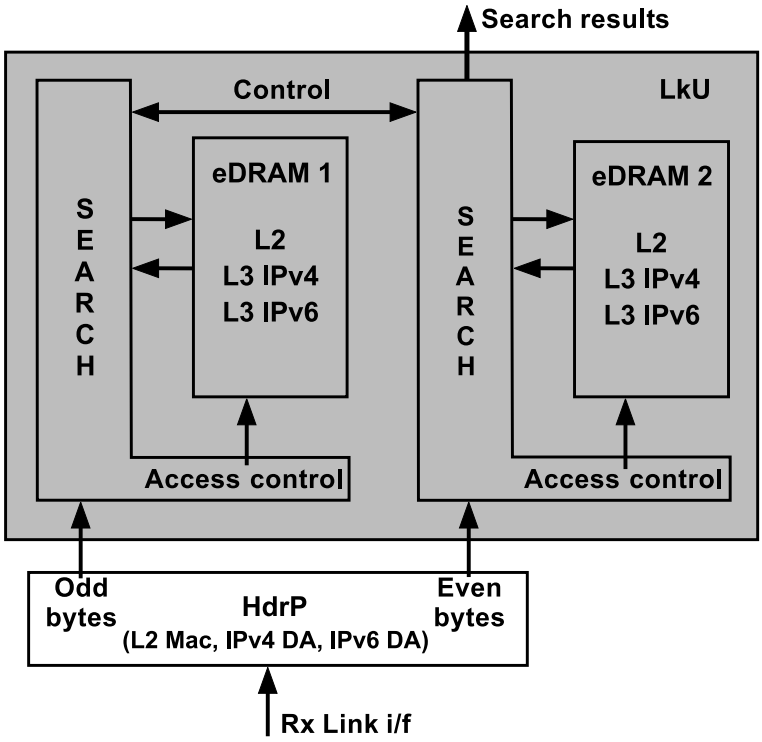

Figure 5. BART implementation based on two eDRAM banks.

type of exact- and prefix-match searches [22]. BART meets the performance requirements by means of a novel compression technique that achieves one of the most storage-efficient data structures in the industry (e.g. BART fits a 72 K-entry routing table in less than 500 KB) in combination with wire-speed search performance and high incremental update rates. The P^2C (Parallel Packet Classification) scheme extends BART for efficient multifield searches [23]. Figure 5 shows an example of a BART implementation based on two embedded DRAM (eDRAM) banks, which searches MAC addresses, IPv4 and IPv6 destination addresses (DAs) using a partitioning of 8-bit segments (see [22]). The available memory bandwidth is used efficiently by alternatingly accessing the two eDRAM banks to "process" the even and odd bytes of the MAC and IP addresses.

3.1.1 Implementation options

Depending on the search and update performance requirements of an application, the BART algorithm can be implemented entirely in either software or hardware. In both cases, specific characteristics of the memory system (e.g., cache line sizes, wide data buses available with on-chip memory) are exploited to improve the performance. A hybrid approach is also feasible in which the update part of the algorithm is implemented in software and the actual search function is implemented in hardware.

Having both a hardware (SDF) and a software (CPU) implementation of BART allows the following interesting scenarios:

(1) The hardware handles the more frequent and simple searches (CPU bypass), whereas the software handles the less frequent but more complex searches (CPU path).
(2) The hardware handles the more frequently used part of the rule set, which is cached on-chip (CPU bypass), whereas the software is only invoked to handle cache misses (CPU path).
(3) The hardware is used by the software as a coprocessor (CPU offload).

3.2 ASIP design methodology

As Figure 4 shows, the SDF comprises a number of functionally mapped components. For some, e.g. the HdrP, the HdrA, or some combination of other functions in Figure 3b, programmability would be desirable in order to support post-deployment implementation of future services. Programmability can be provided by processor cores with a specialized architecture and instruction set, known as application-specific instruction-set processors (ASIPs). They represent a tradeoff between the flexibility of a general-purpose processor (GPP) and the high performance at low area and power cost of hardwired logic.

For the design of ASIPs in our environment, we are working towards an integrated approach for computer-aided derivation of instruction sets and architecture features, starting from specifications of benchmark applications in a high-level language, such as C.

3.2.1 Generic ASIP design flow

We have devised an ASIP design flow, shown in Figure 6, that plugs a number of methods together to proceed from the application code to a new processor description and custom processor tools. The application code is used as input to a compiler front-end, which leads to an intermediate representation (IR) of the application. The IR can usually be visualized as a graph, e.g. a control data flow graph (CDFG), with nodes of very basic instructions, such as add, subtract, shift, multiply, divide, etc.

This graph can be optimized with respect to a basic processor-architecture template using techniques from compilers for GPPs [4] and domain-specific optimizations. An instruction scheduler assigns time steps to graph nodes, and a pattern finder searches for recurring combinations of nodes that might be worth implementing as special instructions [5,18]. It is worth noting that

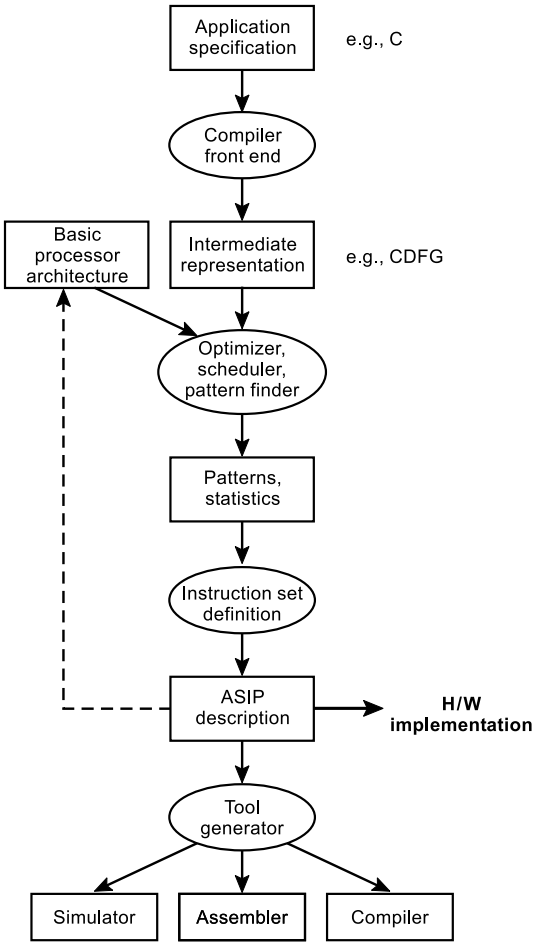

Figure 6. ASIP design flow.

optimizer, scheduler, and pattern finder influence each other's work and may be called repeatedly and in different orders.

The result of the pattern finding is a library of patterns, annotated with metrics about the benefit of each pattern. Based on the metrics, a decision is made regarding which of the patterns to implement as new instructions and how to modify the processor architecture. This step usually still involves interaction with the processor designer, because the set of metrics in current approaches is quite rudimentary and, for example, implementation complexity must be estimated by the designer.

When a new ASIP has been defined in this way, its description can be fed back into the process by replacing the basic architecture template. The process can then be restarted, leading to a further improved architecture. The process is iterated until the performance requirements for the ASIP have been fulfilled. At this point, the ASIP description can be implemented in silicon and it can also be used with a tool generator [12,25] to generate a simulator, an assembler, and possibly a compiler.

3.2.2 Extensions for NPU ASIP design

Most of the ASIP design approaches found in the literature are targeted towards the digital signal processing (DSP) domain. Some of the open questions in this domain remain relevant when applying the methods to protocol-header processing, including how to:

- express more of the programmer's expertise in the application code for better optimization,
- find and compute better metrics for the value of a pattern in order to improve automation, and
- find patterns for special addressing and control instructions.

Furthermore, it is desirable to extend the process with additional optimizations and to adjust it for the particular application domain.

A crucial point for the design methodology is the IR. Restrictions of the IR inadvertently result in deficiencies of the entire process. An IR for our target domain must carry the following information:

- control flow as well as data flow,
- concurrency *and* sequentiality,
- timing constraints, and
- as much of the programmer's expertise as possible.

We use a CDFG-based IR, which fulfills the first two requirements and enables the invocation of standard compiler optimizations as well as the pattern search algorithm in [5], which works on directed acyclic graphs. The CDFG is annotated with minimum and maximum time between nodes, as specified by the application programmer, as well as with the scheduled time step of nodes, as seen in *output transition graphs* for controller synthesis [24].

Furthermore, the application programmer can annotate control edges with minimum and maximum number of visits to support execution-time optimization. Guided by this information, operations can be scheduled for non-critical parts of the program to fulfill timing constraints.

In the instruction-set definition part of the design flow we consider the tradeoff between operand encoding and instruction flexibility. In embedded systems, memory size is critical (e.g., for Program Memory 1 and 2 in Figure 7) and each bit saved in instruction-word length is valuable. Hence, it makes a significant difference whether, for example, a register is encoded explicitly in the instruction word or implicitly with equal source and target registers.

All these methods together yield a consistent flow from the specification of benchmark applications to the description of a new ASIP. An example of a

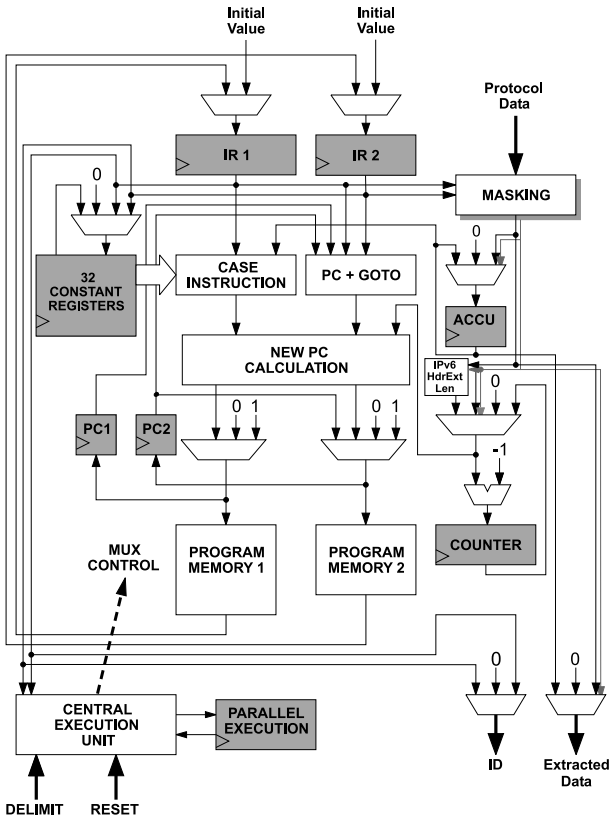

Figure 7. Block diagram of a HdrP.

resulting component would be a HdrP as shown in Figure 4. The design of such a HdrP has been described in [10]. Starting with pseudo code for IPv4 and IPv6, the special instructions given in Table 1 were derived. The architecture, depicted in Figure 7, furthermore comprises a specialized register set, a secondary memory with area-efficient organization for parallel instructions, and additional parallel branching.

Table 1
Specialized instructions for a HdrP.

Instruction	Effect
SEND	Extract bit range from header word and send it to next SDF component.
SEND_REG	Send register content to next SDF component.
WRITE_REG	Extract bit range from header word and write it to register.
IP6_COUNTER	Load counter register with length of IPv6 extension header.
INIT_CASE	Branch depending on which of *num_cases* values matches a register.
LD_CONST	Load condition registers for INIT_CASE.

As a result, this highly specialized processor allows one to completely offload the task of header parsing, which is costly if implemented in GPP software. The size of the parser, including a small instruction memory, is of the order of 0.45 mm^2 in a 0.18-μm process when synthesized for 10 Gbps (311 MHz system clock at 32 bit data path). This demonstrates the area efficiency of the ASIP approach.

3.2.3 Iterative SDF enhancement

In the context of a service-driven NPU the designer would start with a number of services, e.g. HC, NAT, etc., that are to be mapped to the SDF. The services are specified in a high-level language and partitioned for individual SDF components. Several services might share components, such as a HdrP, or one type of service might be mapped to a separate component, e.g. an HC unit. The software fragments for one SDF component represent the benchmark applications at the start of the ASIP design flow out of which the component is generated.

For the next generation of a service-driven NPU, several services will exist that are running on a CPU and should now be moved to an improved SDF. Hence, the software is already specified and only needs to be partitioned for individual SDF components. The basic architecture template in Figure 6 can be replaced by a description of the component in the current NPU. This data starts the process that improves the SDF component.

This demonstrates an easy-to-follow way to exploit field experience with an NPU model to design a next-generation SDF by means of the introduced design flow.

4 Programmable multiprocessor complex

The term network processor includes the notion of a flexible device that can be programmed for a wide area of networking applications. This universality creates momentum for a broad application of the device and support by tools, companion devices and skills. The components introduced in the previous sections are dedicated to classes of such problems as LkU or SCH. Although they boost the performance of functions found in most applications, their flexibility can be restricted. In addition, high-performance processor resources have to be provided to implement functions that have to meet stringent latency constraints.

In typical NPU applications a very high degree of thread-level parallelism is present with low impact on the programming effort. Even if a considerable amount of traffic bypasses the CPUs, a high number of PDUs can be processed in parallel. Of course, synchronization is required to handle data dependencies.

4.1 Processor architecture

In the past, GPPs have been improved to provide very high performance for a single thread. The main methods for this are wider basic word size, RISC design, pipelining, instruction and data caches, superscalar instruction issue, branch prediction, register renaming, out-of-order execution, and dedicated instructions for selected topics (floating point, multimedia etc.). This high performance has its price in terms of area because most methods provide a diminished return of performance per invested chip area. Nevertheless, the mentioned methods are well invested for general-purpose computing applications.

Although word size and RISC design apply to networking applications as well, the other methods do so to a lesser degree or not at all. They rely on such properties of the application as data locality or control predictability. These properties are not or only weakly present in current NPU applications. By exploiting the high thread-level parallelism, overall high performance per invested hardware resources can be achieved by applying a number of processors with only modest effort to improve the performance of a single thread: pipelined RISC design, small per-CPU data and instruction caches, low degree of superscalarity restricted to different operations. For a sample set of applications an optimized architecture with respect to performance per area is presented in [33].

4.2 NPUs as CMPs

By integrating several CPUs into one device the processor complex of an NPU represents a chip-level multiprocessor (CMP). An overview of CMP design issues is given in [19]. It is instructive to compare the architecture of NPUs with other CMP architectures. For example consider the digital signal processor (DSP) TMS320C80 from Texas Instruments Incorporated [31] and the Hydra architecture from the University of Berkeley [17]. Both are designed to have very high performance. However the applications are quite different: signal processing and multimedia-rich, general-purpose computing. Both designs rely on a high thread-level parallelism, created by the programmer (TMS320C80) or the compiler (Hydra). The Hydra concept allows a variable number of processors. In one implementation, four cores have been used.

Whereas the processors in the Hydra are equivalent to each other, the architecture of the TMS320C80 is heterogeneous: one floating-point-capable master processor controls four integer-only slave processors. As in the proposed NPU architecture certain critical subtasks such as lookup and scheduling are mapped to dedicated hardware, the tasks that remain for the CPUs are more uniform than in NPUs without SDF. Therefore, an implementation with homogeneous processors will cover a wide range of applications. An NPU dedicated to applications with special requirements affecting only a fraction of the PDUs can also combine different types of processors (e.g. cryptographic) to form a heterogeneous system.

To allow the processors to work at full clock speed, memory is included on the chip in the two example CMPs. As on-chip memory is expensive, its utilization has to be maximized by adapting it to the needs of the intended applications. Hydra uses caches extended with a mechanism to handle speculative execution. This includes checking for dependencies on the fly and undoing the result of thread execution if a violation is detected. Although this method was developed originally for parallelizing loops it can be used for advanced NPUs as well when certain applications only occasionally write back to a data object (flow context). In contrast the DSP heavily relies on directly addressed memory (8 KByte per processor) in addition to caches combined with a DMA engine. Such an arrangement can be used in an NPU as well, especially when targeting deep packet processing. In this case, the PDU data can be transferred before and after processing independently on the CPU into/out of the on-chip memory. The shared memory pool (4 KByte per processor) of the IBM NP4GS3 implements this approach [20]. Providing further features for the memory access can be beneficial for NPUs depending on the intended application spectrum such as scheduling memory write-backs or insert/delete operations on frames.

The traditional instruction and data cache architecture can be modified to better fit the NPU requirements. As typical performance requirements enforce applications to fit entirely into the instruction cache a dynamic instruction cache with larger than typical line size combines the advantages of low area cost and large possible program size while guaranteeing low latency when the instruction footprint is small. A data cache example of an NPU-specific optimization is the following: Every application needs to access the PDU header. This can be exploited by triggering the data cache of a standard processor core to prefetch the beginning of a PDU before starting execution.

The current workload, such as incoming PDUs or expired timers, needs to be distributed in some way. This can either be done in a centralized fashion by software on a dedicated processor, in a distributed manner by software on all processors or with hardware support. The latter option has the advantage of easier use of knowledge that is deep in the SDF, e.g. in the CL, and better coupling of enqueueing and work delivery to conserve the order of PDUs.

A natural way to adapt a multiprocessor architecture to an application area is the instruction set. In fact, several NPUs provide special instructions to accelerate the intended application space and reduce program size. In an NPU that provides such basic functions as HdrP, LkU and MTR in hardware, there is a reduced profit for instructions that support the same functions in the processor. However, functions on a higher level such as encryption can be provided by coprocessors, which improves the overall performance per area and power efficiency.

Obviously, the more standard tasks on lower protocol layers can be delegated to units in the SDF, the higher the fraction of work on higher layers for the processor. If problems with very short program paths such as IP forwarding are performed in hardware, the average available program path length per PDU on the CPU increases. These higher layer tasks are more diverse, more complex and therefore represent a greater investment in software. Hence, the way the software is created converges to conventional system software including modularization, use of high-level languages including the acceptance of overhead for maintainability and ease of development. In the course of this process, standard processors offer significant advantages over specifically designed processors.

5 Performance evaluation

One of the main goals of the proposed approach is to complement and augment service functions with NPU-specific cores to offload the CPU from instruction-cycle-intensive operations. To investigate such new NPU structures as well as to analyze and quantify whether the overall performance requirements for a specific solution are fulfilled, performance evaluation in an early design stage is necessary. Moreover we wish to exploit the inherent advantages of a core-based SoC design approach, which emphasizes the reuse of already existing components such as bus structures, memory controllers, interface controllers and processor cores.

As design space exploration is an important step in the design of today's SoCs, it is essential to integrate this process into the overall design methodology as well as into the existing design flow tool chain. The challenge for such an environment is to combine architecture validation with performance evaluation aspects, disciplines which traditionally have been dealt with separately. Moreover reusability of models, which can be further refined during the design flow, is a key issue.

5.1 Current approaches

There are currently only a few approaches to design space exploration. All of these environments attempt to map specific functionality onto predefined architectural components and to find optimal architectural system scenarios that fulfill specific performance requirements [8,32].

As mentioned above, most of the existing approaches separate the design phase from the performance evaluation process of a system. The general system design flow is based on the reuse of library components on different levels of abstraction and their successive refinement [15]. In the performance evaluation process, however, this is not the case. Here there are many different issues concerning the performance of a system, and a separate abstract model exists for each one. This results in a large set of models, each of which deals with a specific analysis, and hence is not reusable for any other performance issue. Mainly analytical models based on queuing theory or stochastic models are used here. The result is a large gap between the common design flow and the performance evaluation process.

5.2 Component-based design space exploration

Integrating performance evaluation into the common SoC design flow while considering the flexibility of our hybrid SDF concept is the overall goal of the proposed methodology, called *component-based design space exploration*. Our approach is based on a library of models, which combine abstract functional behavior with performance evaluation aspects. To achieve high reusability, there is only one model for each component.

In an architecture composition step, see Figure 8, some of these models are adopted to build a specific architectural scenario. The evaluation is done by simulation. The performance data collected during the model execution is then analyzed and the evaluation results constitute the feedback to the architecture composition phase for architectural improvements.

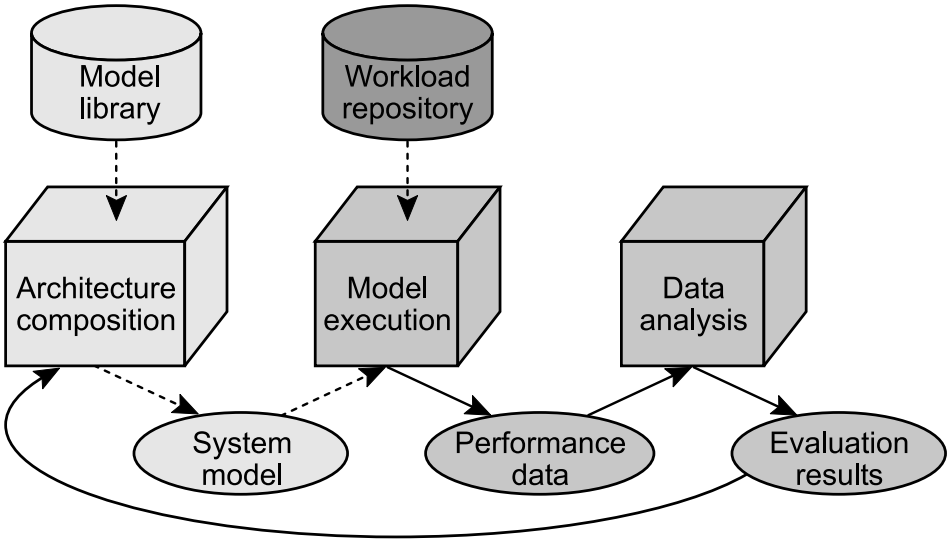

Figure 8. Performance evaluation framework.

Instead of customizing the performance aspects of a model to one specific metric, the metric part is extendable. To meet this challenge, every model is composed of two layers, a functional and a data collection layer. The functional layer implements the abstract behavior of the component with clearly defined interfaces. The data collection layer deals exclusively with the performance metrics. This separation allows for functional refinements without changing the data collection mechanisms, but also enhances the flexibility of the data collection.

5.3 Performance evaluation of a service-driven NPU architecture

To evaluate performance, we use a simplification of the service-driven NPU architecture shown in Figure 4 with only one CPU core. All models are implemented in SystemC [30]. The SDF model consists of two parameterizable receive and transmit links, a simple LkU, a CL and a BM. The links receive PDUs at a parameterized input rate, the CL tags the PDU for the CPU bypass or the CPU. Based on this tag the BM organizes the transfer to a transmit link or over the bus interconnect to memory. PDU processing is done through an abstract and flexible model of a CPU core, which mimics the workload of software-implemented service functions without having to have the code of specific applications. For example, this model can be customized by a set of system and application-specific parameters, such as number of instructions per packet, I-cache/D-cache size, probability of cache misses or number of load and stores from and to memory.

For our evaluation experiments we assume an aggregate input load of 1 Gbps. The workload for every scenario consists of same-sized PDUs. The CPU model

runs at 200 MHz. In our evaluation we consider two different scenarios, one with a heavy packet processing effort and another one with simple packet processing. According to [21] we assume in the first scenario 3000 instructions per PDU if the processor core is not supported by an SDF, whereas for the second scenario only 500 instructions per PDU are taken. The effects on the throughput are illustrated in Figure 9(a) and (b). In both charts the y-axis represents the number of PDUs in thousands processed in one second and the x-axis distinguishes between different PDU sizes. As the aggregate input load is kept constant at 1 Gbps, the number of offered PDUs decreases with PDU size. So the theoretical bound on both figures resembles the ideal situation, where all offered PDUs can be processed by the network processor.

Offloading the CPU reduces the number of instructions per PDU. A good example of this case is the introduction of a CL component as part of the SDF. Depending on the application considered, the CL function can consume up to 80% of all necessary CPU cycles [21], which in our first scenario results in a reduction by 2400 instructions per PDU. When looking at the graph where no CPU offload and no bypassing of PDUs takes place, the ideal throughput according to the theoretical bound is not reached for any of the packet sizes. By CPU offloading the overall performance can be increased, resulting in a 1-Gbps throughput for large PDU sizes. This effect can be enhanced by adding bypass capability to our SDF. We assumed a traffic mix, where 40% of all received PDUs can be handled by the DiffServ IP forwarding SDF components without having to be passed to the CPU. Consequently the processing headroom for PDUs traversing through the CPU is almost doubled, which has the effect that now PDUs with a size of 512 bytes can be processed at wire speed.

The results obtained for our simple packet processing scenario are shown in Figure 9(b). In this case the maximum throughput of 1 Gbps can be achieved for large PDU sizes without any SDF support. But even for PDUs with a size of 512 byte, only a throughput of 554 Mbps is available. When assisting the CPU with a CL component in the SDF the performance can be increased to almost line speed. This effect can be achieved also for smaller PDUs, such as 256 byte, by bypassing 40% of the offered traffic mix, which results here in a throughput of 903 Mbps. Considering the fact that the average PDU size of an Internet traffic mix is 512 byte, our example NPU with SDF support could be used for wire-speed processing of 1 Gbps.

5.4 Outlook

The above-described concept of component-based design space exploration complements traditional design flow with performance evaluation. Its flexibility and extendability both on the functional and the metric level ensure model

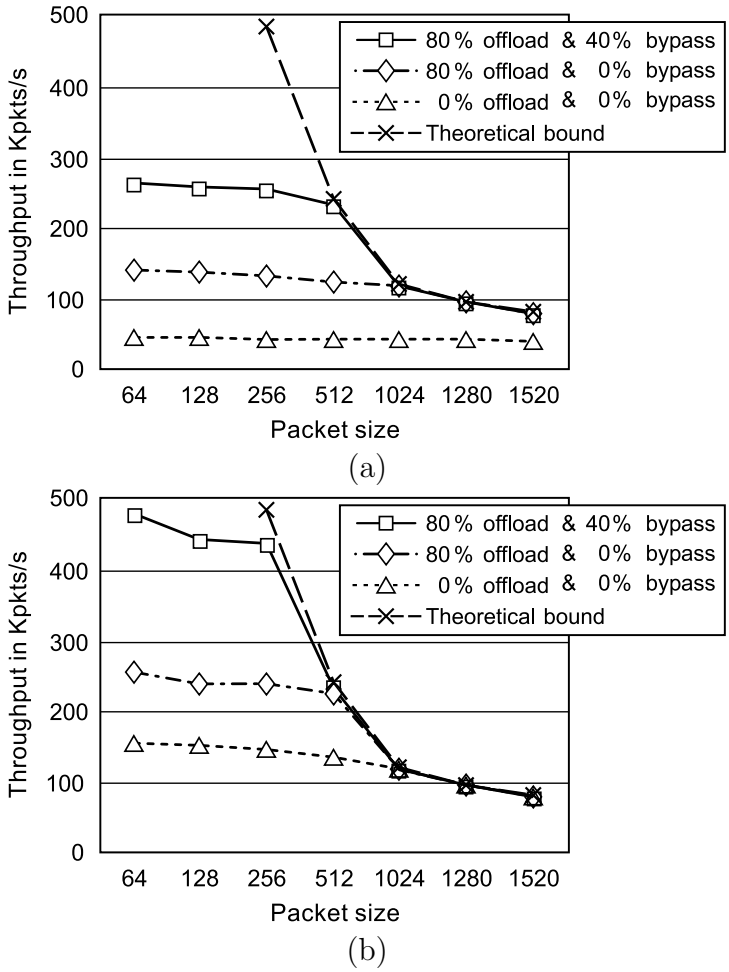

Figure 9. Packet processing scenarios with (a) 3000 and (b) 500 instructions per PDU.

reuse. Even with the currently existing basic library it is possible to evaluate different design scenarios, such as the SDF concept. Using this evaluation methodology we are able to verify the soundness of the proposed architecture and at the same time to identify the different requirements and implications for each component.

6 Summary and conclusions

Our message is that the value proposition of NPUs must come not only from the processing elements or multiprocessor clusters, but from a balanced mix of processor cores and service-mapped components. This is crucial as NPU designers strive for more flexibility and simultaneously enhanced performance. We propose a design methodology for a modular, hybrid, SoC NPU architecture. It starts with the end application in mind, and ceases by mapping

the service required into the system's components, which include processor cores and deterministic performance components, such as configurable logic and ASIPs. The flexibility requirements are addressed via the processor cores and the ASIPs. The performance requirements are fulfilled with the service-mapped deterministic performance components. The latter are placed and interconnected such that they form a service-driven data flow (SDF). The SDF provides a minimum service that allows a sizable amount of traffic to bypass the CPUs.

We describe challenges and methods to design certain SDF components, integrate and organize CPUs, and evaluate the system. The topics covered here certainly do not exhaust the issues and challenges that this architecture design methodology poses. Issues for further study include open on-chip interfaces and data structures. The proposed methodology provides an open framework for NPU design that can fit into various markets and applications areas. The SDF addresses the performance requirements of a market, and advanced services can be provided both via the CPUs and by incorporating appropriate coprocessors. Thus, different types of traffic follow along different paths in the system, allowing the system to cover both the functional and performance needs of the flows, the network and the user.

7 Acknowledgments

The authors thank Kerry Imming, John Irish and the entire IBM Rochester development team for their constructive and sagacious technical feedback, and Lars Annel, Vick Chandra and John Fakiris for their stimulating marketing insight and valuable customer validation support.

References

[1] 3GPP TS 23.060, General Packet Radio Service (GPRS); Service Description, v4.2.0, October 2000.

[2] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich and H. Wilkinson, The Next Generation of Intel IXP Network Processors, Intel Technology Journal, vol. 06, no. 03, August 15, 2002.

[3] Agere Systems, Inc., The Challenge for Next Generation Network Processors, White Paper, April 2001.

[4] A. V. Aho, R. Sethi, and J. D. Ullman, Compilers, Addison-Wesley, 1986.

[5] M. Arnold and H. Corporaal, Designing domain-specific processors, in: Proc. of 9th Int. Symposium on Hardware/Software Codesign (CODES'01), April 2001, pp. 61–66.

[6] AF-TM-0056.000, ATM Traffic Specification Version 4.0, ATM Forum, April 1996.

[7] E. Bowen, C. Jeffries, L. Kencl, A. Kind, and R. Pletka, Bandwidth Allocation for Non-Responsive Flows with Active Queue Management, in: Proc. of Int. Zurich Seminar on Broadband Communications, Zurich, 2002.

[8] S. Chakraborty, S. Künzli, L. Thiele, A. Herkersdorf, and P. Sagmeister, Fast and Accurate Performance Evaluation of Network Processor Architectures: Combining Simulation with Analytical Estimation, submitted to: Computer Networks, Special Issue on Network Processors.

[9] P. Crowley and J.-L. Baer, A Modeling Framework for Network Processor Systems, in: Proc. of 8th Int. Symposium on High-Performance Computer Architectures; Workshop on Network Processors, Cambridge, MA, February 3, 2002.

[10] G. Dittmann, Programmable Finite State Machines for High-speed Communication Components, Master's Thesis, Darmstadt University of Technology, 2000.
`http://www.zurich.ibm.com/~ged/HeaderParser_Dittmann.pdf`.

[11] draft-ietf-diffserv-model-06.txt, An Informal Management Model for Diffserv Routers, Y. Bernet, S. Blake, and A. Smith, February, 2001.

[12] F. Engel, J. Nuhrenberg, and G. P. Fettweis, A generic tool set for application specific processor architectures, in: Proc. of 8th Int. Workshop on Hardware/Software Codesign (CODES 2000), May 2000, pp. 126–130.

[13] EZchip Technologies Ltd, Network Processor Designs for Next-Generation Networking Equipment, White Paper, 1999.

[14] S. Floyd and V. Jacobson, Random Early Detection Gateways for Congestion Avoidance, ACM Transactions on Networking, vol. 1, no. 4, August 1993, pp. 397–413.

[15] D. D. Gajski, F. Vahid, S. Narayan and J. Gong, Specification and Design of Embedded Systems, PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[16] P. N. Glaskowsky, Intel Beefs Up Networking Line: New Chips Help IXP Family Reach New Markets, Microdesign Resources, Cahners Microprocessor Report, March 18, 2002.

[17] L. Hammond, B. A. Nayfeh and K. Olukotun, A Single-Chip Multiprocessor, IEEE Computer, vol. 30, no. 9, September 1997,pp. 79–85.

[18] I.-J. Huang and A. M. Despain, Generating instruction sets and microarchitectures from applications, in: Proc. of Int. Conference on Computer Aided Design (ICCAD-94), November 1994, pp. 391–396.

[19] J. Huh, S. W. Keckler and D. Burger, Exploring the Design Space of Future CMPs, in: Proc. of Int. Conference on Parallel Architectures and Compilation Techniques (PACT 2001), pp. 199–210.

[20] IBM Corporation, Network Processor 4GS3 Overview, Application Note, October 1999.

[21] C. Jenkins, NPU Co-Processors, The Power to Process, Presentation at Network Processor Conference, August 2000.

[22] J. van Lunteren, Searching very large routing tables in wide embedded memory, in: Proc. IEEE Globecom, vol. 3, November 2001, pp. 1615–1619.

[23] J. van Lunteren and A. P. J. Engbersen, Dynamic multi-field packet classification, in: Proc. IEEE Globecom, November 2002.

[24] J. A. Nestor and V. Tamas, Exploiting scheduling freedom in controller synthesis, in: Proc. of 6th International Workshop on High-Level Synthesis, November 1992, pp. 74–86.

[25] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, LISA: Machine description language for cycle-accurate models of programmable dsp architectures, in: Proc. of 35th Design Automation Conference (DAC'99), June 1999, pp. 933–938.

[26] RFC 1633, Integrated Services in the Internet Architecture: An Overview, R. Braden, D. Clark and S. Shenker, July 1994.

[27] RFC 2309, Recommendations on Queue Management and Congestion Avoidance in the Internet, B. Braden, et. al, April 1998.

[28] RFC 2475, An Architecture for Differentiated Services, S. Blake, et. al, December 1998.

[29] RFC 3031, Multiprotocol Label Switching, E. Rosen, et. al, January 2001.

[30] SystemC, Version 2.0 Beta-1, User's Guide, 2001.

[31] Texas Instruments Incorporated, TMS320C80 Digital Signal Processor Data Sheet 2000.

[32] L. Thiele, S. Chakraborty, M. Gries, S. Künzli, Design Space Exploration of Network Processor Architectures, in: Proc. of 8th Int. Symposium on High-Performance Computer Architecture; Workshop on Network Processors, Cambridge, MA, February 3, 2002.

[33] T. Wolf and M. A. Franklin, Design Tradeoffs for Embedded Network Processors, in: ARCS 2002, Karlsruhe, Germany 2002, pp. 149-164.