

Technische Universität München
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Integrierte Systeme

Dissertation

A Network Processor Architecture with Application-Optimized Reconfigurable Processing Paths (FlexPath NP)

Dipl.-Ing. Rainer Ohlendorf

Munich, September 28th, 2010

Technische Universität München
Lehrstuhl für Integrierte Systeme

A Network Processor Architecture with Application-Optimized Reconfigurable Processing Paths (FlexPath NP)

Rainer Ohlendorf

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr.-Ing. Eckehard Steinbach

Prüfer der Dissertation:

1. Univ.-Prof. Dr. sc.techn. Andreas Herkersdorf
2. Univ.-Prof. Dr.-Ing. Erik Maehle, Universität zu Lübeck

Die Dissertation wurde am 28. September 2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 1. Juli 2011 angenommen.

Erklärung

Ich erkläre an Eides statt, dass ich die der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Promotionsprüfung vorgelegte Arbeit mit dem Titel:

A Network Processor Architecture with Application-Optimized Reconfigurable Processing Paths (FlexPath NP)

am Lehrstuhl für Integrierte Systeme unter Anleitung und Betreuung durch Prof. Dr. sc.techn. Andreas Herkersdorf ohne sonstige Hilfe erstellt und bei der Abfassung nur die gemäß § 6 Abs. 5 angegebenen Hilfsmittel benutzt habe.

(✓) Ich habe die Dissertation in dieser oder ähnlicher Form in keinem anderen Prüfungsverfahren als Prüfungsleistung vorgelegt.

( ) Die vollständige Dissertation wurde in … veröffentlicht. Die Fakultät für Elektrotechnik und Informationstechnik hat der Vorveröffentlichung zugestimmt.

(✓) Ich habe den angestrebten Doktorgrad noch nicht erworben und bin nicht in einem früheren Promotionsverfahren für den angestrebten Doktorgrad endgültig gescheitert.

( ) Ich habe bereits am … bei der Fakultät für … der Hochschule … unter Vorlage einer Dissertation mit dem Thema … die Zulassung zur Promotion beantragt mit dem Ergebnis: …

Die Promotionsordnung der Technischen Universität München vom 1.8.2001 in der Fassung der 8. Änderungssatzung vom 22.12.2009 ist mir bekannt.

München, den 28. September 2010 _____

(Rainer Ohlendorf)

Acknowledgments

At first, I would like to thank Prof. Andreas Herkersdorf for having given me the opportunity to work on such an interesting research project under his guidance. The past six years at the Institute for Integrated Systems at TUM were a great chance for me to broaden my knowledge based on my initial course of studies and have provided me with valuable experiences through both the research work and the frequent interaction with students from all over the world. In addition, I would like to thank Prof. Erik Maehle from the University of Lübeck for his interest in this topic and for acting as my secondary examiner on the doctoral examination commission.

At the institute, my foremost thanks is devoted to Dr. Thomas Wild, who - as acting head of the networking research activities - was always a prime contact person and counselor for the discussion of viable conceptual alternatives and research direction. His detailed feedback concerning the structure of the dissertation and assistance in finding the right line of arguments greatly helped me to improve the presentation of the achieved contributions. For the more technical aspects of the work, I have to thank my officemate and FlexPath project partner Michael Meitinger for the frequent and fruitful discussions, in which we were able to solve many challenges to our mutual benefit. Also the co-operation with Daniel Llorente, who designed the SmartMem DMA engine next door, has to be appreciatively mentioned.

Further important help came from Johannes Zeppenfeld, who has contributed with his computer science background in developing sound simulation models, especially during the development and evaluation of the HDGA classification algorithm. I have learned quite a lot in our discussions and could significantly expand my C/C++ programming skills. Christopher Claus was of great help in resolving FPGA-related problems as the institute's expert in the Xilinx tool chain. Finally, I would like to mention the following current and previous colleagues at the institute, who have all contributed to a very pleasant working atmosphere: Felix Miller, Gregor Walla, Roman Plyaskin, Stefan Wallentowitz, Michael Feilen, Prof. Walter Stechele, Doris Zeller, Verena Draga, Holm Rauchfuss, Robert Hartl and Dr. Paul Zuber.

Finally, I would also like to thank my family for their support, which was also an important factor in being able to successfully complete this thesis.

Table of Contents

Summary of the Thesis

This thesis deals with improvements of switching nodes in Internet-based communication networks. During the past decade, increasing requirements for the networking infrastructure (i.e. routers, gateways, etc.) have led to the development of network processors (NPs). Network processors are highly integrated silicon components that achieve both high flexibility and performance. Contemporary networking infrastructure has to provide the flexibility of adapting to ever-changing new application needs, while the link speeds have increased to tens of gigabits per second with 100 Gbit/s already on the horizon. The current thesis proposes a new architectural approach to the network processing problem, in which dedicated hardware modules in the ingress and egress data path relieve a central processor cluster. The flexibility of the programmable processor cluster can be retained for those tasks that require this flexibility. More standardized tasks are solved by application-specific high performance hardware. Especially, a hardware unit for packet classification is proposed, which identifies the incoming traffic in real-time and dispatches the packets to the most suitable processing elements within the heterogeneous multi-processor cluster. Beyond a static processing path selection based on networking application characteristics, I have also investigated load balancing strategies that distribute the packets to paths supporting different quality-of-service levels within the NP. The presented hardware offload doubles the forwarding throughput of the NP in comparison to state of the art architectures with the same amount of processing resources.

Zusammenfassung der Arbeit

Die vorliegende Arbeit beschäftigt sich mit Verbesserungen von Netzknoten in Internet-basierten Kommunikationsnetzen. Steigende Anforderungen an die Netzwerkinfrastruktur (z.B. in Routern, Gateways, etc.) haben im letzten Jahrzehnt die Entwicklung von Netzwerkprozessoren (NPs) befördert. Netzwerkprozessoren sind hochintegrierte Siliziumbausteine, die gleichzeitig hohe Anforderungen an Flexibilität und Performance erfüllen. Die heutige Netzwerk-Infrastruktur muss flexibel genug sein, um sich an immer neu entstehende Anwendungsanforderungen anzupassen, während die Geschwindigkeiten auf den Übertragungsstrecken mittlerweile bei mehreren zig Gigabit pro Sekunde liegt und erste 100 Gbit/s Strecken in naher Zukunft folgen werden. Die vorliegende Arbeit schlägt einen neuartigen architekturalen Ansatz im Design von Netzwerkprozessoren vor, in dem dedizierte Hardware-Module im Ein- und Ausgangsdatenpfad den zentralen Netzwerkprozessorkomplex entlasten. Die Flexibilität der programmierbaren Ressourcen wird nur noch für die Aufgaben verwendet, die diese Flexibilität auch benötigen, während andere, besser standardisierte Aufgaben von spezialisierten Hardware-Modulen bearbeitet werden. Im Speziellen wird eine Hardware-Klassifikationseinheit vorgeschlagen, die den ankommenden Verkehrsfluss in Realzeit untersucht und die Pakete auf die für sie am Besten geeigneten Verarbeitungseinheiten innerhalb des heterogenen Multiprozessorclusters verteilt. Neben der statischen Verarbeitsungspfadwahl aufgrund von Applikationsanforderungen, habe ich in dieser Arbeit Lastbalancierungsstrategien untersucht, welche die Pakete auf Pfade mit unterschiedlichen Dienstgütemerkmalen (quality-of-service) innerhalb des NP-Systems verteilt. Die vorgestellte Entlastung des Prozessorclusters ermöglicht eine Verdoppelung des Paketdurchsatzes im Vergleich zu einem herkömmlichen NP mit gleich vielen Rechenressourcen.

1. Introduction

The work covered in this thesis is positioned in the context of Internet-based communication systems. I have proposed and investigated a new architecture for network processors (FlexPath NP) that optimizes the packet processing performance by providing different run-time reconfigurable processing paths and hardware-offload features that relieve the programmable processor resources. The following sections introduce the reader to the topic by describing the evolution of the Internet and give a high-level description of the packet processing infrastructure and networking application requirements. Based on these high-level observations, the fundamental ideas of the FlexPath NP architecture are mentioned and the introduction is concluded with the organization of the subsequent chapters of the dissertation.

Classical computer networks found in enterprises and universities, as home and office networks in residential areas and the server clusters operated by Internet service and content providers made up the Internet during the 1990s. Around the year 2000, an integration of the classical telephony networks (public switched telephone network, PSTN) and data networks took place. The introduction of Voice-over-IP (VoIP) protocols allowed transferring voice connections over asynchronous packet switched networks that were originally developed for data communication. Otherwise, backbones in the data networks started using Sonet/SDH technology with their high transmission bandwidth, which was originally developed to transmit high-order multiplexed voice signals in a strictly synchronous network. Around the same time, data services began to be offered by the mobile telephone providers (e.g. with GSM/GPRS and evolution towards 3G technologies found in UMTS), linking their networks into a unified, globally meshed communication network supporting both voice and data transmission.

The more widespread availability of the Internet to the general public and the increasing access speeds offered as customers were able to migrate from dialup connections (14 - 56 kbit/s) to DSL and cable modems (1 - 30 Mbit/s) also led to the introduction of new services, most importantly e-Commerce and multimedia. Those new applications, in addition to "plain" web traffic like http and email, required widespread use of cryptography for confidential data and a differentiation and prioritization of real-time from non-real time user applications. In a subsequent step, peer-to-peer applications, where individual users share content among each other (in contrast to the classical client-server model, where content is kept in a centralized place) caused an additional shift in communication patterns and increased the total traffic amounts in the Internet.

As a result of the above mentioned trends and developments, an exponential growth in Internet backbone transmission bandwidth with annual growth rates between

60% and 100% on average could be observed throughout the last decade [1]. This growth imposes significant pressure to improve the performance of the network architecture.

The networks making up the Internet are organized in a hierarchical fashion (see also Figure 1) with routers aggregating the traffic to and from smaller sub-networks and forwarding them towards the destination networks via peering or backbone links in the WAN core. Please note that the figure shows only the basic structure of the network aggregation and interconnection structure, the physical instances are not corresponding to an actual architecture, as such information is not publicly available from the actual ISPs.

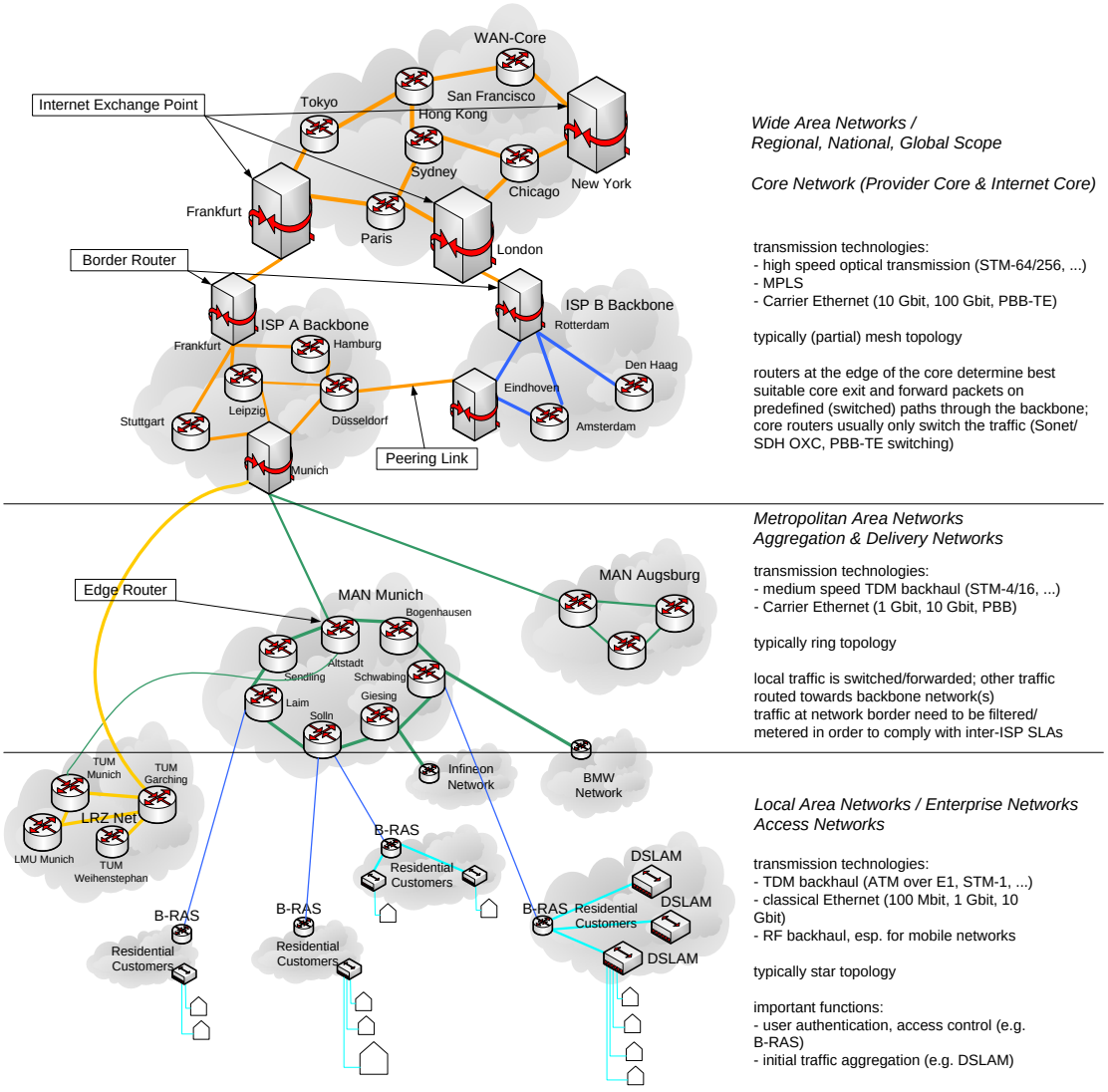


Figure 1: Hierarchical Structure of the Internet

Residential customers and companies/universities connect to the Internet through the ISPs' points-of-presence, usually entering an aggregation network that combines the traffic originating from the same geographical region. Of course, these networks allow switching traffic directly between locally close neighbors, while

networks lying further apart (this notion of distance applies also to networks attached to a different ISP operating in the same geographical region!) have to be reached through the wide area network. Although there exist some peering links between individual ISPs on a more regional level, the global connectivity is generally achieved through Internet Exchange Points [4], where so-called Tier 1 ISPs cross-connect their locally attached networks with each other.

Special gateway routers are found at the borders of the individual networks, so that the different providers have the ability to perform traffic monitoring and policing, and are able to translate traffic to a different protocol stack, which may be used in the adjacent network. The enforcement of inter-provider service level agreements (SLAs) at the network borders and eventual protocol conversions require a flexible router infrastructure with lots of general-purpose computing power. The same also holds true for the access equipment, where traffic first enters a provider's network and has to be inspected in order to achieve billing and accounting purposes as well as filtering functions, in order to block malicious behavior (B-RAS devices in Figure 1).

Traffic inside a provider's own network, which has to be forwarded between several internal switches or routers to reach the final endpoint of the network, is typically only forwarded without further packet inspection. In the Internet core, this is often achieved by using MPLS (multi-protocol label switching), which assigns pre-configured, locally unique labels for each of the predefined connections based on the initial routing information; and routers within the MPLS network perform a simple switching only on the MPLS labels rather than performing the traditional switching or routing function. As the MPLS labels are determined at the edges of an MPLS network by inspecting the IP destination addresses and QoS (quality of service) parameters of the packet, this kind of forwarding is often referred to as layer 2.5 forwarding [5]. In recent years, development of "carrier-grade" Ethernet technologies has started a trend among ISPs to simply switch the traffic based on L2 information rather than performing L3 routing based on the IP addresses in the packet.

As defined in RFC 1812 [2], routers are those functional entities in the network that perform the forwarding function with the help of routing protocols. As the variety of application layer protocols has increased dramatically since the early days of the Internet, and those applications come with greatly differing QoS requirements, DiffServ [3] has been introduced as a framework for differentiating traffic in the routers into different service classes and treating them in various different ways. Some of the necessary functions required to achieve QoS are packet filtering, metering and policing, and forwarding packets on different priority levels. Such QoS architectures are not only constrained to the IP protocol suite, but can also be found in recent Ethernet standards (e.g. VLAN IEEE 802.1Q), ATM and MPLS networks. In all places, where formerly unrelated networks are coupled together, gateways assure interoperability of the communication on both sides. In contrast to classical

routing, which is constrained to layer 3 of the OSI protocol stack, gateways may also work on higher layers up to the application layer (L7).

Figure 2 shows the modular architecture for routers and application gateways following standards such as AdvancedTCA [6]. The AdvancedTCA specification only provides standardization for the mechanical and electrical characteristics for rack-based communication systems (e.g. size of pluggable cards, electrical power supply, thermal power dissipation and a common backplane wiring scheme). Each vendor has the freedom to choose from a range of different backplane protocols and speeds and switch fabric parameters like line card connectivity and redundancy depending on the application requirements.

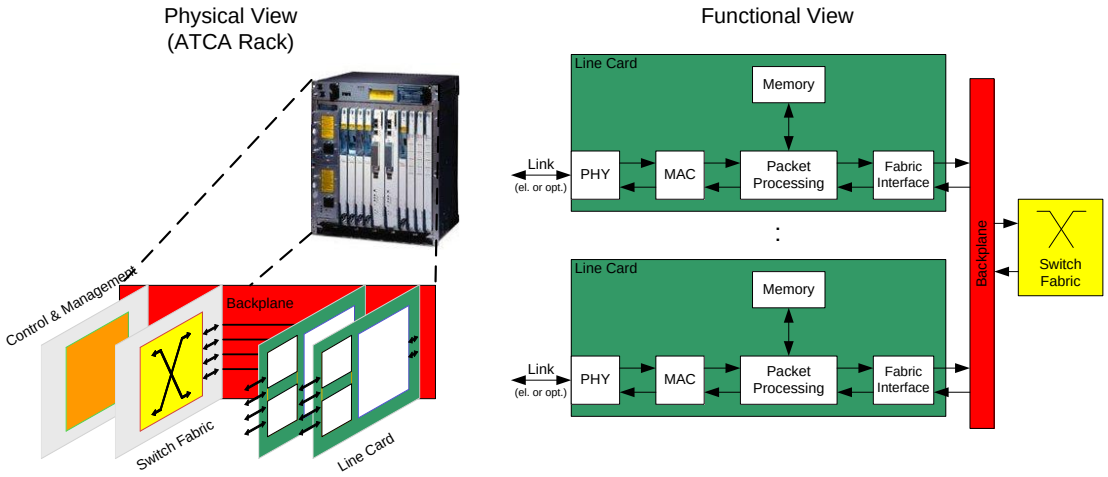


Figure 2: Typical Router Implementation with ATCA Standard

The actual packet processing takes place in one or several packet processing circuits on the individual line cards, while both line cards or the processing entities may be implemented in a half duplex or full duplex operation mode.

Traditionally, there existed high-performance ASIC (application-specific integrated circuit) solutions for high-speed switching in the telephony backbones. These solutions yielded a high performance, but as a hardwired function, they provided no flexibility / adaptability to newly emerging protocols and applications. A change in any of the transmission protocols comes at the cost of designing a new ASIC that can then be deployed in an improved line card. On the other hand, first routers in data networks were comprised of general-purpose PC systems with several network interface cards that implemented the forwarding functions and routing protocols in software. New protocols and functions could be easily deployed and tested by modifying the software, of course at a significantly lower performance level compared to the highly-optimized ASIC solutions.

With the integration of formerly separated networks and the advent of new applications and protocols around the year 2000, packet processing had to become

significantly faster while retaining the flexibility associated with the previous software implementations. Thus a migration of router technology from a general-purpose PC system to a custom-made ASIC (as they had been used in the telephony networks) would not serve the purpose. To address this performance / flexibility dilemma, network processors (NPs) were proposed to bridge the gap between slow but flexible general-purpose processors and high-performance ASIC implementations with their lack of flexibility and high development costs.

In general, NPs rely on a combination of application-specific instruction set processors (ASIP) and hardware accelerators. Hardware acceleration is used for networking-specific tasks that are common across many applications or where it is mandated by the computational complexity (e.g. cryptography). In addition, many NP architectures implement line / MAC / switch fabric interfaces on-chip and contain a set of memory controllers and connections to dedicated, off-chip hardware accelerators via standardized interfaces. This integration simplifies the board design of the router linecards and improves the overall system reliability, which is also an important aspect in the telecommunication industry. However, it is important to realize that no standard architecture has been found yet. Every NP vendor offers its specific solution, and designs from different suppliers look quite differently for NPs targeting different market segments and networking applications.

Based on an analysis of the first generation of network processors, the FlexPath NP architecture is proposed [7] that improves the performance of the NP by

- enhancing the software-programmable capabilities of the NP with hardware offload in order to relieve the processors from simple, recurring tasks faced across many networking applications and

- providing a variety of run-time reconfigurable processing paths (i.e. functional unit traversal sequences) in the data plane of the device that are optimized for the requirements of different networking applications.

The fundamental idea behind the FlexPath NP architecture is to dynamically adapt the processing paths for the arriving packets so that the requirements of the current traffic load can be best met by the available resources in the device. A special hardware unit called Path Dispatcher performs a real-time classification of the incoming traffic into a set of application classes, for which optimized processing paths are provisioned in the FlexPath NP architecture. These processing paths include traditional programmable processor resources, arbitrary combinations of hardware offload units and software processors and a dedicated hardware-only forwarding path ("AutoRoute") for simple switching / forwarding functions. The classification function can be achieved by the heterogeneous decision graph algorithm (HDGA), which is fine-tuned to the constraints of on-chip real-time packet

classification at multi-Gigabit/s packet rates. In contrast to most state-of-the-art classification techniques, which operate on five or less different header fields, HDGA scales for rules bases with up to 20 dimensions. The architectural support for assigning the packets to different processing paths can inherently be used to address the load balancing problem among several parallel processing instances. This thesis presents a combination of packet spraying and hash-based load balancing (S&H) as a novel load balancing strategy, which achieves a high processor utilization and system throughput by taking into account the different characteristics and requirements of various networking applications.

The remainder of the dissertation is organized in the following way:

- Chapter 2 covers the complete state-of-the-art relevant to the individual contributions of this thesis, starting with existing commercial network processors and academic approaches in the NP field (section 2.1). The NP state-of-the-art is complemented by a survey of currently important and evolving networking protocols (section 2.2). Section 2.3 summarizes previous work in the field of packet classification techniques as a base for the derivation of HDGA. Finally, existing load balancing strategies for NPs are presented in section 2.4.

- Chapter 3 presents the FlexPath NP concept with its specific architectural modules based on an analysis of existing networking applications and NP architectures. The claims made during the presentation of the architectural concept are further supported by system-level performance simulation results, which focus on the potential of performance improvements that the hardware-offload aspects in a FlexPath NP offer compared to a traditional processor-centric NP architecture.

- Chapter 4 focuses on the Path Dispatcher unit, which performs the real-time packet classification task in the FlexPath NP system. The elaboration comprises the concept of HDGA, functional simulation results and finding an optimized architecture for efficient hardware implementation. The chapter is concluded with synthesis results for the Path Dispatcher unit in an FPGA demonstrator platform.

- Chapter 5 introduces a combination of two different load balancing schemes (packet spraying and hash lookup, S&H) that exploit optimum performance of the processor resources in a given FlexPath NP architecture. Functional simulation results are provided that compare the individual components and the combined scheme to several techniques of the prior art. The achievable performance benefits are shown based on realistic Internet backbone traffic traces.

- Chapter 6 presents the implemented components and system setup of a combined FlexPath NP / SmartMem demonstrator on a Xilinx Virtex-4 FPGA development board. Selected measurement results are presented that illustrate

the performance of the FlexPath NP approach and prove the validity of the assumptions made during the concept development and simulations.

– Finally, chapter 7 summarizes the scientific contributions of this dissertation to the state of the art and presents an outlook to possible future research directions based on the lessons learned during the FlexPath NP project.

The work presented in this dissertation originates from the FlexPath NP project, which was associated with the German research foundation's priority program "Reconfigurable Computing" (SPP 1148) during the time frame 2005 - 2009. Two dissertations cover the entire work performed in the FlexPath project, with the current thesis focusing mainly on the ingress data path pipeline elements and load balancing strategies and the other dissertation by Michael Meitinger ([107]) discussing the egress data path pipeline elements. For the demonstration purposes in both theses, we implemented a common demonstrator of a FlexPath NP on an FPGA development platform that also includes the SmartMem DMA engine, which was developed in a parallel project by our colleague Daniel Llorente, and is covered in his dissertation ([108]).

2. State of the Art

The following chapter illustrates the state of the art for the work covered in this thesis, and is divided into three main topics:

- The first topic (section 2.1) focuses on the evolution of network processors during the past ten years and presents current implementation solutions and related academic approaches. In addition, the current application mix in the Internet is characterized (section 2.2) from which conclusions about the requirements for future networking compute architectures are drawn. The analysis of these two fields triggered the proposal of the FlexPath NP architecture, which is derived in detail in chapter 3.

- The second topic (section 2.3) focuses on existing approaches in packet classification. This field is relevant to the major contribution of this Dissertation, the HDGA packet classification scheme implemented in the Path Dispatcher, which is presented in chapter 4.

- The state of the art survey is concluded by discussing approaches to load balancing in network processors (section 2.4), which is an important function in multi-processor systems in general. Load balancing in the context of FlexPath NP is addressed in chapter 5.

2.1. Network Processors

2.1.1. Commercial Network Processor Architectures

2.1.1.1. Commercial NPs Prior to the FlexPath NP Proposal

Although the survey on NPs undertaken by Shah [9] dates back to the year 2001, this document provides an excellent starting point for understanding the evolution of the network processor field. Therefore, a selection of NPs from that period should be presented first in order to show the evolution of these devices and draw conclusions about the architectural trends that have taken place in the market ever since.

The Agere Payload Plus NP [10] is a multi-chip solution that consists of a Fast Pattern Processor (FPP) and Routing Switch Processor (RSP) in the data path and the Agere System Interface (ASI) for control plane functions and communications with a management host. The FPP receives the packets from the link, parses the packet and hands it over to the RSP chip. The FPP consists of a multi-threaded, pipelined processor and hardware assists for pattern matching and checksum / CRC calculations. The RSP chip receives the packets from the FPP along with certain classification information and performs traffic management, traffic shaping and queuing functions before performing final modifications on the packet and sending them out towards the switch fabric. The functions are implemented with three dedicated VLIW (very long instruction word) processors: traffic management compute engine, traffic shaper compute engine and stream editor compute engine. In addition to the VLIW processors, the chip provides interfaces to external SDRAM to store the packets while they are being queued.

The MSP5000 processor from Brecis Communications [11] addresses converged voice and data communications linking enterprise sites to the network edge. The task is achieved by two DSP (digital signal processor) processors for voice and packet processing while a MIPS RISC (reduced instruction set computer) core takes over control plane functions. The processors communicate with a special QoS-aware system interconnect (Multi-Service Bus Architecture) with a peak data rate of 3.2 Gbit/s. The processors are complemented on-chip with a set of hardware accelerators for cryptographic functions and CRC (cyclic redundancy check) calculation.

IBM's Power NP [12] is a representative of a massively parallel processor cluster. Apart from a single general-purpose PowerPC that is used for control plane processing, it features an embedded processor complex with 16 programmable protocol processors. In addition to the 16 cores, there are seven specialized hardware accelerators for DMA (direct memory access), checksum calculation, traffic shaping and policing and inter-processor communication.

The PXF NP from Cisco [13] features 16 processors arranged in eight parallel pipelines. The pipeline depth may be extended by chaining several PXF chips in a router system. By forcing the packet processing task into a pipeline structure, with every processor performing only a specific sub-task, a deterministic behavior of the NP with respect to packet throughput may be achieved.

Intel's IXP 1200 NP [14] also follows the processor cluster architecture found in the IBM Power NP. It provides six multi-threaded microengines that support a total of 24 tasks in the system. The instruction set of the microengines is specifically optimized for packet processing and they have to be programmed in their own assembly language in order to achieve maximum performance. The NP comes with integrated hardware support for hashing and queue management and features a StrongARM RISC processor for control and management purposes.

The X40 NP from Xelerated [15] targets the high speed end of the NP spectrum. It consists of a single pipeline with 10 stages; each stage consists of a classification and action stage. The action stage is made up of a packet instruction set computer (PISC), which is a processor with a specialized ISA (instruction set architecture) for packet processing. In addition, the chip allows accessing external memory and CAM (content addressable memory) from all pipeline stages. During operation, every stage in the pipeline works on a different packet and completes processing within a single clock cycle, thus achieving very high packet rates.

2.1.1.2. Evolution of the Commercial NP Field after the FlexPath NP Proposal

Following the burst of the New Economy bubble, a wave of consolidation started in the NP business. Some vendors went out of business, others were acquired by larger companies or product lines were spun off to new companies. Successful product lines, e.g. Intel's IXP product line, evolved over several generations. More processor cores were added to the system, interconnect structures upgraded, memory and I/O interfaces adapted for newer standards [16] and the devices were scaled down to new CMOS process generations. Starting with the IXP2400 series of NPs, the microengines were equipped with special "next-neighbor" interconnect registers. They allow very efficient passing of data among neighboring microengines, thus enabling a pipelined programming model in addition to the parallel processor cluster model of the IXP1200 series. Moreover, a further differentiation for the various targeted market segments could be observed. In 2005, processors from the second generation existed in a range of two to 16 microengines, clock frequencies between 600 MHz and 1.5 GHz and target line rates between 1 Gbit/s and 10 Gbit/s. The latest model (IXP2855, [17]) also features two hardware crypto cores that enable IPsec processing at up to 10 Gbit/s. However, in 2007 Intel sold its NP line to Netronome, which will further develop NP products evolving from the IXP28xx.

In January 2006, SafeNet announced the SafeXcel IP inline security engine [18] as a security application co-processor that works as a full-fledged offload for security application handling from general-purpose compute architectures in network processor SoC designs. The security engine can be used either as a traditional co-processor, relieving the general-purpose parts of the NP from the compute-intensive cryptographic algorithms, but it can also be integrated as an autonomously operating processor in "bump in the stack" (i.e. packets are en-/decrypted before reaching the processor, so that the processor sees only plaintext packets) or "bump in the wire" (i.e. packets are processed without even being touched by the processor) use cases. Especially the last mentioned "bump in the wire" use case is based on essentially the same idea as the proposed AutoRoute feature in a FlexPath NP, which will be described in detail in chapter 3.2. The architecture appears to be commercially successful, as the device is still actively marketed in 2009 [19]. Figure 3 shows the architecture of the SafeXcel-IP-196 block as of 2009.

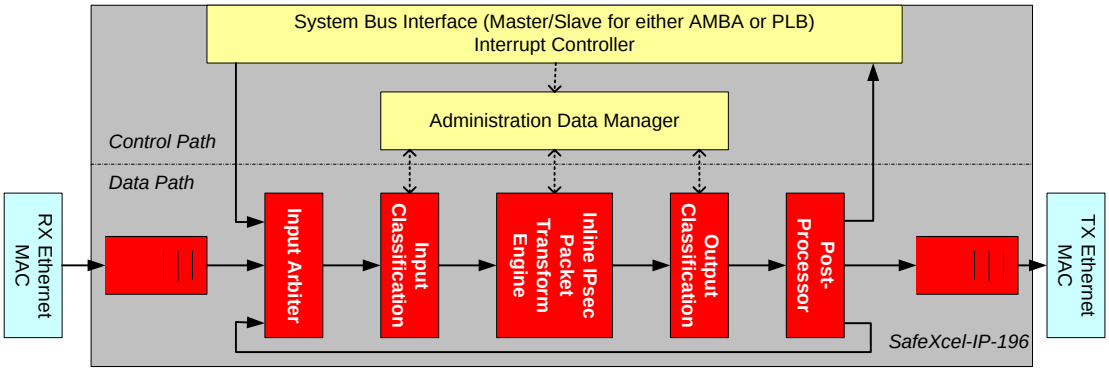


Figure 3: SafeXcel-IP-196 IP Flow-Through Packet Engine

The nP7300 from AMCC [20] follows the run-to-completion operation model (Figure 4a) with three nPcore processors, each of which supports 24 tasks. From the point of view of the NP programmer, the system performs like a 72 core processor, while the packet processing task for each packet is executed in a single thread. Consequently, there is no multi-processor overhead necessary during software development, i.e. the programmer does not need to consider splitting the application into several chunks, which might be distributed among the processors and organize data communication and synchronization between the cores. The data plane processor complex is enhanced with a Channel Service Module (CSM) that provides an autonomous DMA function to store and retrieve packets from the I/O interfaces without processor intervention. The chip also includes a dedicated traffic manager for traffic shaping, policing and queuing and a separate hashing unit. Memory and external co-processors (e.g. TCAM memories) can be accessed via standardized interfaces. The nP7300 has no dedicated control plane processor on chip, but can be connected to a host via 10/100/1000 Ethernet and the data plane is targeted for 10 Gbit/s half-duplex operation.

Netronome's NFP3200 NP [21], which is the first successor of Intel's IXP28xx product line, now features up to 40 microengines for data plane processing with local instruction stores optimized for run-to-completion or pool-of-threads programming models. In addition, an ARM11 embedded RISC core is used for IPsec key exchange algorithms, routing table updates and system management functions. The NP also comes with a hardware cryptography module that supports up to 10 Gbps, while the 40 microengines allow packet processing at 30 Mpps or 20 Gbps with 2,000 instructions per packet.

In 2008 Cisco released information about its own Quantum Flow Processor [24], which is initially a two-chip solution with one chip for the processors and a second chip for traffic management. The processing chip consists of 40 Tensilica RISC processor cores [25] that are C-language programmable and provide four threads per core at 900 MHz to 1.2 GHz. The packets arriving from either the line interfaces or the switching fabric are first handled by the traffic manager chip, which also provides access to a centralized memory and includes all system I/O interfaces. When the packets are ready for processing, they are dispatched to one of the 160 threads in the processor engine chip, which are connected with the rest of the system via a crossbar switch architecture. The initial two chip solution will be used in Cisco's ASR 1000 series aggregation switch routers with an internal packet processing capability of 5 to 100 Gbps. There are plans to integrate the system into a single chip design and increase the number of processor cores in the packet processor engine in future versions of the NP.

Another current design that adheres to the parallel processor cluster architecture is the Octeon II processor family from Cavium Networks [26], of which first processors are announced to ship in the fourth quarter of 2009. The NP family will feature a new generation of 64bit MIPS cores in the data plane. There will be devices with 1 to 32 cores, each of them running between 800 MHz and 1.5 GHz. There are also up to 75 hardware accelerators available in the system, which are connected to the cores via an eight Tbps Hyperconnect crossbar switch. The first NP generation targets the 40 Gbps market but is claimed to provide I/O capabilities for up to 100 Gbps.

Xelerated still pursues the strict pipeline approach (Figure 4b in section 2.1.3) with its X11 NP [22] released in 2008. In contrast to the X40 [15], the X11 features five blocks of 32 pipelined PISC processors, thus 160 processors in total. External memories and hardware accelerators may be accessed only from distinct Engine Access Points (EAP) at the beginning of each of the five pipeline blocks. The EAP includes packet buffers to cope with the latency associated with the individual accelerators or memory accesses. With a core frequency of 240 MHz the X11 is able to process packets at up to 24 Gbps. Xelerated has also announced a new generation of network processors (HX family) that addresses the evolving 100 Gbps Ethernet market. In comparison to the X11 NPs, the number of processors in the

programmable pipeline is increased to 512 and the devices feature an integrated traffic manager and switch fabric [23]. It is characterized by Xelerated with the term "linecard on a chip", due to its high level of integration that needs only external DRAM, TCAM (NSE) and PHYs as additional off-chip elements.

2.1.2. Academic Network Processor Projects

2.1.2.1. Academic NP Investigations Prior to the FlexPath NP Proposal

The Field-Programmable Port Extender (FPX) developed at Washington University in St. Louis [27] in 2001, provides an FPGA-based reconfigurable platform for network processing for ATM. The platform comprises an extension board with two FPGAs, which can be plugged in between the line card and switching backplane interfaces of an ATM-based Gigabit switch (WUGS). The first FPGA, which comprises a simple, reconfigurable switching fabric with a small control memory allows to route incoming traffic on a flow-level granularity (i.e. ATM VPI/VCI numbers) between the line card interfaces, switching backplane and two dynamically reconfigurable slots in the second FPGA. In addition, by sending special control cells to this FPGA, bitstreams for the second FPGA can be transmitted over the network, allowing a subsequent (partial) reconfiguration of the other FPGA. The second FPGA contains two reconfigurable slots for the actual user-defined packet processing functions and has interfaces to external SRAM and SDRAM. The FPX platform has been used to demonstrate IP packet routing, per-flow queuing and flow control and application-level content inspection and modification. By making use of reconfigurable FPGA resources in the network processing device, the benefits of run-time modification of the packet processing function can be combined with the hardware-like performance of the FPGA logic.

In 2002, Troxel et.al. from the University of Florida at Gainesville [28] propose a network processor architecture that allows to dynamically reconfigure the pipeline depth of microengines in an Intel IXP1200-like processor configuration during system runtime in order to improve the overall system performance given fluctuations in the arriving traffic pattern. The authors present only simulation results of the proposed system. Assuming that the networking application can be executed on microengine pipelines of various depths (i.e. the task can be partitioned to run on one, two or three engines with different resulting processing times per processor), they can exploit a performance gain by changing the pipeline depths assigned to different traffic types during the system runtime. They present an application scenario from a defense application with three different packet types, so that a generalization to Internet traffic is not straightforward.

The PRO3 network processor proposed by Papaefstathiou et.al. from Ellemedia and the Technical University of Crete [29] in 2004 introduces dedicated hardware support for DMA and queuing operations in the NP SoC and enhances two

programmable RISC cores with two dedicated hardware accelerators. The field extraction unit (FEX), which is a firmware-configurable hardware assist, parses the incoming packet and may write important header fields into the register file of the processor. The processor can then execute the actual high-level part of the networking application and the field modification unit (FMO) is available for writing back results from the RISC core registers into the packet, which may include bit- and byte-level operations that are hard to implement efficiently in the general-purpose RISC core. The authors show, that a PRO3 system with two FEX-RISC-FMO pipelines achieves a similar performance for benchmark TCP and UDP applications in comparison to the Intel IXP2400 with 6 microengines. In addition, they could demonstrate that the hardware-based queue management in the PRO3 is significantly more efficient than the standard software-based solution in the IXP, such that both systems could deliver roughly the same performance, while the PRO3 chip consumes only about one fifth of the IXP's die area.

In 2005, Ravindran et.al. from the University of California at Berkeley [30] investigated the forwarding performance of a network processor architecture based on parallel Xilinx Microblaze processor pipelines. After optimizing the partitioning of the IPv4 forwarding application onto a three-stage pipeline, a total system throughput of 1.8 Gbps can be achieved with a total of 12 Microblaze processors in four parallel pipelines. This value is compared to the forwarding performance of an Intel IXP2800, which achieves 10 Gbps with its 16 optimized microengines. After normalizing the results to chip area, the authors show that the FPGA-based solution performs only a factor of 2.6 worse than the commercial NP. The claimed benefit of the FPGA solution is that by using soft processor IP with the provided toolchain and standard off-the shelf FPGA products is an attractive choice for niche application domains, where the cost of starting a full ASIC design may be too high in comparison to the expected number of units to sell.

2.1.2.2. Academic NP Investigations after the FlexPath NP Proposal

DynaCORE ([31], chapter 16, pp. 335-354 and [90]), which was developed at the University of Lübeck in 2006, is a dynamically reconfigurable co-processor for compute-intensive payload manipulations in network processor systems. The FPGA-based architecture combines the near-hardware performance of an FPGA implementation with the dynamic partial reconfiguration capabilities offered by Xilinx FPGAs. The static part of the DynaCORE provides system I/O interfaces for communication with the off-chip NP (e.g. a commercial NP or our FlexPath NP demonstrator system (see [91])) and a controller for system monitoring and reconfiguration management. As requests arrive from the attached NP to execute cryptographic algorithms or pattern matching applications on the arriving packets, the reconfiguration controller insures that a sufficient amount of hardware accelerators is dynamically configured into the reconfigurable slices of the system

and forwards the incoming packets to the respective unit. Correct routing of the packets and glitch-free operation of the DynaCORE during partial reconfigurations is achieved by a special network-on-chip architecture called CoNoChi.

Another run-time reconfigurable NP architecture was presented by Kachris et.al. at the University of Delft in 2006 ([32]). They regard an NP architecture based on a Xilinx FPGA with either the Microblaze soft core or PowerPC hard core processors as central processing elements. These programmable resources may be assisted with hardware accelerators for Checksum calculations, DES encryption or IDCT transcoding as representative examples for plain IP forwarding, IPsec or voice/video application processing. The respective functionality may however also be achieved by the processors (at a lower performance level). Now, Kachris assumes different shares for the individual networking applications and computes an optimum combination of accelerators (type and quantity) in order to maximize throughput. During system runtime, the current load on the network interfaces is monitored and the hardware accelerators are dynamically reconfigured in order to yield maximum utilization of the available soft- and hardware instances.

The GigaNetIC architecture [33] developed by Niemann et.al. at the University of Paderborn in 2007 proposes a massively parallel multi-processor system for networking applications. Clusters consisting of four embedded RISC processors with local memories and hardware accelerators are interconnected using a 2D-mesh network-on-chip. Peripherals and hardware assists with system-wide relevance (e.g. Ethernet cores, IPsec accelerators, etc.) may be attached directly to the NoC and are thus universally reachable and can be accessed as a shared resource. In this way, the proposed architecture allows finding balanced solutions between locally shared and globally shared resources. The authors claim C-language programmability for the embedded processors and an architecture that may be programmed either as a functional pipeline or as run-to-completion cluster. Benchmarking results are presented for a simple TCP/UDP integrity check and forwarding application. Results from an FPGA-prototype implementation with only two processor clusters (total of eight cores) and two NoC switches are also extrapolated for an ASIC implementation with 20 clusters (total of 80 cores). The ASIC implementation would consume the same die area as current state-of-the-art desktop processors. The GigaNetIC is shown to have a forwarding performance which is roughly one order of magnitude greater than that of the general-purpose CPU, but it consumes about two orders of magnitude less energy. However, the authors give no comparisons to commercial NP implementations.

In 2006, researchers from Hitachi present investigations on a cache-based network processor architecture ([34]). A conventional NP cluster with parallel/pipelined processors is augmented with a hardware pipeline that provides pre- and post-processing capabilities and the cache system. The first packet of each packet

stream/burst is forwarded to the processor cluster, where the traditional forwarding function is implemented in software. After processing, all relevant information (i.e. flow identification and all types of packet manipulations / data) are recorded in the cache system. When a subsequent packet of the same flow arrives, this information is retrieved from the cache and may be applied on the packet in the post-processing stage. Measurement results performed on an FPGA prototype with real world Internet traffic revealed, that between 10% and 40% of the total processor cluster performance is sufficient to forward 100% of incoming traffic. In turn, the authors argue that a traditional 10 Gbps to 40 Gbps NP device augmented with their cache implementation would be able to process a 100 Gbps link in a lossless fashion. Such an implementation would in turn only consume about 45% of the power needed in comparison to a conventional NP that is scaled up to 100 Gbps performance.

In 2007, Li et.al. from the National University of Defense Technology in China proposed the DynaNP architecture ([35]). A DynaNP consists of a set of processors that are connected over a central interconnect to shared memory and ingress / egress management engines (IME, EME), which perform a DMA function to and from the shared buffer and initial packet pre-classification. The networking application is partitioned into tasks, which are subsequently assigned to run on distinct processors. Depending on the type of arriving packet, processing may be achieved by executing a variable number of tasks. The initial packet classification in the IME determines the first task to be executed for the incoming packet and subsequently assigns it to the queue of the respective processor. After processing each task, the processor decides whether further steps are necessary (sending it on to another processor) or back to the EME for retransmission over the link or fabric interfaces. As an initial partitioning and mapping of the total task set among the available processors will not be optimally balanced, and the utilization of individual processors is also expected to change with variable traffic loads, the authors propose a dynamic task migration algorithm, with which they essentially perform load balancing of the tasks among the processor cluster. The publication presents system-level simulation results of the proposed DynaNP architecture. However, the elaboration lacks a prototypic implementation and the authors do not comment about the overhead associated with frequent task migrations and how to insure packet ordering when reconfiguring the processing paths.

2.1.3. Conclusions

The characteristics of the first and second generation of NPs form the basis, from which the FlexPath NP architecture [7] was initially defined:

– A number of different companies have developed NPs for different market segments with quite different architectural approaches and processing resources.

– All regarded NPs combine programmable resources with hardware accelerators for compute-intensive and networking-specific tasks. Most NPs also include line and fabric interfaces and memory controllers for off-chip packet storage. Control Plane functions, which are not performance-critical are usually mapped to a general-purpose RISC processor. For the packet processing task, RISC processors, some of them with application specific instruction set extensions, DSPs, or traditional ASIP designs are used.

– While there are also some multi-chip solutions, most designs favor integration into a single chip design.

– Due to the performance requirements of network processing, multi-threading and parallel processing are widely used. Multi-threading allows hiding long memory or hardware accelerator access latencies, as the programmable core can continue working on other packets being processed in different threads. As far as operation models are concerned, processors may be used in a symmetric multi-processor cluster (run-to-completion architecture), a dedicated processor pipeline, or a combination of both (parallel pipelines). Figure 4 illustrates these three architectural approaches in an abstracted form.

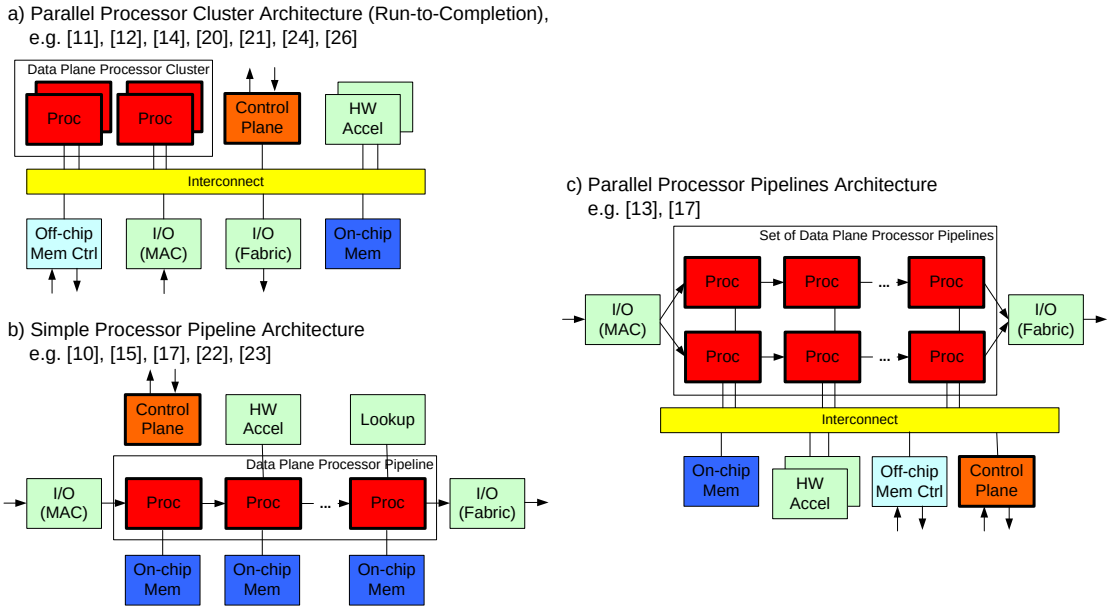


Figure 4: Fundamental NP Architectures: run-to-completion parallel processor cluster (a), simple processor pipeline (b), and parallel processor pipelines (c)

More recent developments in the commercial NP field can later be used to analyze the industrial relevance of the proposals made in the FlexPath NP approach. The following conclusions about commercial network processor architectures may be drawn with respect to the subsequent features:

- **Programmability:** Programming network processors is still an important challenge for some devices. As far as the data plane processors consist of cores with packet processing specific instruction sets, no standard compiler tool chain may be available. Consequently, those cores have to be programmed in their own assembly language, which can become quite awkward. Big vendors, such as Intel have therefore provided software libraries that include optimized code for a big variety of commonly needed protocols. However, the possibility of simply and efficiently upgrading an NP-based system in the field with new software patches for new protocols is somewhat limited. In contrast, some vendors resort to standard embedded cores that are programmable in C, like Cisco's QFP or the MIPS core used in the Cavium Octeon II. As integration density has improved largely, it is now possible to trade off the comfort of C programming on a few more standard embedded RISC cores versus fewer processors featuring an application-optimized instruction set. Another programmability aspect can be found when comparing the run-to-completion solutions with the pipelined architectures. Pipelined architectures have the inherent advantage of a deterministic behavior, and thus a fixed maximum packet rate. The fixed packet rate is helpful when a manufacturer guarantees operation of his device for a given speed rate, as a worst case scenario with a continuous stream of shortest size packets leads to a fixed packet rate for any given line speed. On the other hand, this guarantee comes at the price of having to partition the application into chunks that can be executed within the individual pipeline stages in the mandated time. Also, when there are only limited access points to external accelerators and memories, this restricts the freedom of software programmable solutions.

- **Interconnect:** As NP manufacturers have scaled up the number of processor cores in run-to-completion architectures, traditional on-chip buses have become a system bottleneck. Therefore, a migration to more sophisticated structures such as crossbar switches were necessary in order to fully exploit the processing performance of the larger processor clusters. In contrast, pipelined architectures may be implemented easily, as they require only simple point-to-point connections between neighboring processing elements. In addition to the guaranteed throughput that pipelined architectures can offer, this explains, why NPs targeting the highest speed market segment still adhere to the pipeline model.

- **Processors:** Current packet processors are predominantly RISC processors, some of them with a customized instruction set. Other processor types such as

the MSP5000 from Brecis Communications [11] with DSPs or VLIW processors (Agere, [10]) are no longer found in current designs. Specific high-performance tasks are still solved by means of hardware support, and not mapped to software programmable units. However, there is a strong trend towards multi- and even manycore processors, and multi-threading is used extensively in order to hide accelerator and memory access latencies. As packet processing usually treats the packets as independent units, packet processing can be far easier parallelized than traditional general-purpose compute applications. The advances made in modern CMOS process technologies helped increase the clock frequencies of the processors from a few hundred MHz in the early NP designs to well above the GHz margin.

- **Hardware Acceleration:** For compute-intensive tasks in packet processing, such as CRC checksum calculations or IPsec cryptographic algorithms, only hardwired logic is able to deliver real-time performance for current link rates. But also other fixed and standard tasks such as queuing and DMA that have to be performed for every packet are often offloaded to dedicated hardware units. In total, one can observe that both the variety and number of instantiated accelerators has been increased in parallel with the number of processor cores and the cumulated line rates on current router blades.

- **Integration:** The shrinking process technologies not only allow scaling chips towards containing even more processor cores and dedicated hardware units. Integrating as much functions as possible into a single chip design also helps to significantly reduce design complexity and cost and it increases reliability. Complex and expensive interconnects across printed circuit boards can be saved, if it is possible to integrate the entire processing chain from the MAC interfaces and the actual processor complex towards the switch fabric interface and memory controllers into a single chip. These single chip NPs are currently standard, except for the most processing intensive solutions for the highest possible speed grades (e.g. Cisco's ASR 1000 router, which provides deep packet processing performance in the multi-Gigabit domain).

- **Specialization:** While initial NP design proposals tried to address the problem of network processing with a full breadth approach, recent developments show a strong differentiation of the devices that target individual market segments. Devices for high-speed switching and routing in backbone networks are typically addressed with high-performance pipelined processors and hardware support for lookups, CRC calculations and traffic management. The processor architecture can be optimized to efficiently execute the functions on Layers 2 to 3 of the OSI stack, and don't have to provide as much general purpose processing power as for application layer or deep packet processing. In contrast, the parallel processor architecture NPs are more ideally suited for edge and access network

deployments, where the individual line rates may be slower than in the aggregated network core, but access control, intrusion detection, QoS policing, etc. have to be performed on the incoming packets. These deep packet processing applications, which may work on the higher protocol layers or even parts of the packet payload in addition to the pure L2/L3 forwarding can be better achieved with a more general-purpose processor and a single-threaded, run-to-completion processing model. Traditional router deployments in central office environment are typically implemented using rack-mounted systems with the possibility of scaling the performance by adding additional line cards or switching fabrics as needed. In contrast, smaller form factors with the NP as SoC solution and only few peripherals on a single PCB are available for mobile network base stations or customer premises equipment. The employed NPs need less processing performance and come with less cores and lower operating frequency to provide more power efficient systems.

Regarding academic network processor concepts, ideas from the following research areas have been investigated by the research community:

- **Reconfigurable computing:** several projects ([27], [31], [32]) have used the reconfigurability of FPGA devices in order to adapt an NP during runtime to changing conditions in the incoming traffic. In addition, by making use of reconfiguration, the functions implemented in the device may be almost as easily changed as in a conventional software system, but the performance of FPGA hardware accelerators is more similar to that of ASICs.

- **Hardware offload:** The PRO3 project [29] demonstrated the benefits of assisting general-purpose processors with networking-specific configurable hardware. A more radical kind of offload is proposed by Hitachi [34], where a full packet forwarding path is implemented in hardware that is controlled by the contents of the packet processing cache.

- **Interconnect:** The GigaNetIC project [33] pointed out an architecture that is well suited for scaling to much larger numbers of programmable resources. As commercial manycore NP designs moved away from shared bus architectures towards crossbar switches and processor pipelines, the GigaNetIC proposes a network-on-chip (NoC) based design.

2.2. Networking Applications

2.2.1. IP Forwarding

The traditional task of routers is forwarding of IP packets towards their final destination. The associated tasks are defined in RFC 1812 [2] for IP version 4, which is still the dominant IP version today. After packet reception, the link layer information of the packet is discarded. Next, the router has to validate the IP header, which includes checking the IP checksum and the time-to-live field in the packet header. If the packet is valid, the IP destination address is used together with the routing table information to determine the output interface onto which the packet has to be forwarded. The CIDR addressing scheme [36], which is currently used for IPv4 mandates a longest prefix match of the destination address versus the prefixes stored in the routing table. Finally, the time-to-live field has to be decremented by at least one and the IP checksum must be re-calculated. After that a new link layer header may be appended to the packet and the packet can be placed into the output queue associated with the determined physical output port. In this best effort scenario, all IP packets are treated with equal priority, such that no QoS guarantees will be given by the network.

2.2.2. QoS Mechanisms

With the introduction of multimedia applications over the Internet, the traditional best effort forwarding model of the Internet has proven to be insufficient. Two alternative architectures have been proposed to allow service differentiation in the Internet and give priority to certain packets over others.

In the IntServ model [37] proposed in 1994, hosts or routers can establish virtual connections with certain associated performance guarantees. If the routers along the connection have sufficient resources available, the virtual connection is accepted and packets of this connection are treated separately from the other traffic. This separation requires some kind of input filtering or access control and metering whether the traffic does not exceed the predefined service parameters such as a bandwidth limit. In addition, the router must provide different queues and a scheduling mechanism that insures proper multiplexing before the output interfaces. Due to the requirement of establishing virtual connections and having to classify each incoming packet against the full set of connections, this approach is not scalable to a large number of users and is therefore only rarely used.

In contrast, the DiffServ architecture [3] proposed in 1998 uses the old type-of-service field in the IP header as DiffServ codepoint (DSCP) to indicate that a packet belongs to a certain predefined traffic class. The network operator associates a certain per-hop-behavior with each DSCP, which may include parameters such as maximum allowable bandwidth, forwarding and queuing priority, etc. The individual

packets have to be marked with valid DSCP values either by the end hosts (if they know about the network operators' traffic classes), or by the border routers sitting at the edge of the DiffServ network. For the routers within the network there is the big advantage that no state information has to be maintained. The forwarding function is simply inspecting the DSCP field when determining the processing priorities or queuing priorities. Thus, a full classification and policing of the individual packets only happens once at the network edge. Within the DiffServ network the DSCP value determines the forwarding behavior, which is typically limited to around ten different classes [38]. Therefore DiffServ scales far better and may be easier implemented compared to IntServ. The concept of marking individual packets with short QoS identifiers has been considered successful enough, so that the same concept is now also implemented in the most recent carrier grade Ethernet standards (see chapter 2.2.6).

2.2.3. Security Applications

As the Internet evolved from a pure academic research network towards a ubiquitous communication network, transmission of sensitive information (like trade secrets, financial information, etc.) caused serious security and privacy concerns. In order to tackle these challenges, authentication and encryption technologies had to be provided. The IPsec framework [39], which became initially standardized along with IPv6 in 1998, provides those services for both IPv6 and IPv4. The IPsec framework consists of the two data plane protocols encapsulating security payload (ESP) and authentication header (AH).

The AH protocol only assures that a packet comes from the claimed sender, and that the packet did not get modified en route to the receiver. This is achieved by applying cryptographic operations on the header and by calculating a cryptographic checksum over the payload. However, the payload itself is not encrypted, and can therefore be read by anyone tapping into the communication path.

The ESP protocol encrypts the payload, i.e. the original content of the packet is no longer legible for others between the two IPsec endpoints. Standards-conforming implementations (current RFC from 2005) have to support AES and 3DES algorithms for encryption of the payloads and HMAC-SHA1 as cryptographic checksum.

IPsec implementations make use of two databases:

- Security Policy Database (SPD): The SPD contains entries of connection endpoints and the action, which should be applied to packets between those endpoints. Possible actions are Discard, Bypass (IPsec processing) and Protect (en- / decrypt). The router effectively has to perform firewall filtering for the entire traffic, when it matches the incoming packets to the connections listed in the SPD.

– Security Association Database (SAD): The SAD contains the negotiated security associations (i.e. cryptographic keys, algorithm, etc.) for each (simplex) connection between two endpoints. It needs to be consulted when the SPD query results in "Protect" and an actual IPsec operation has to be performed on the packet.

IPsec can be deployed both in the network devices as well as at hosts (i.e. computers). If two hosts protect their communication with IPsec protocols, the routers in the network simply forward those packets, so there are no extra requirements for the NPs in those systems. The more interesting case for the network infrastructure happens, when IPsec protocols are used to establish a secure connection between two sites that are connected over the public Internet (Figure 5). Here, the hosts within the corporate networks (NW1 and NW2) can trust each other and don't have to encrypt their messages. However, as people from one site need to communicate with people from the other site, packets are encrypted by a virtual private network (VPN) gateway router before being released into the public network. Routers in the Internet can only read the outer packet headers going from VPN GW NW1 to VPN GW NW2, but cannot gain any information about the actual communication partners or the contents of the communication.

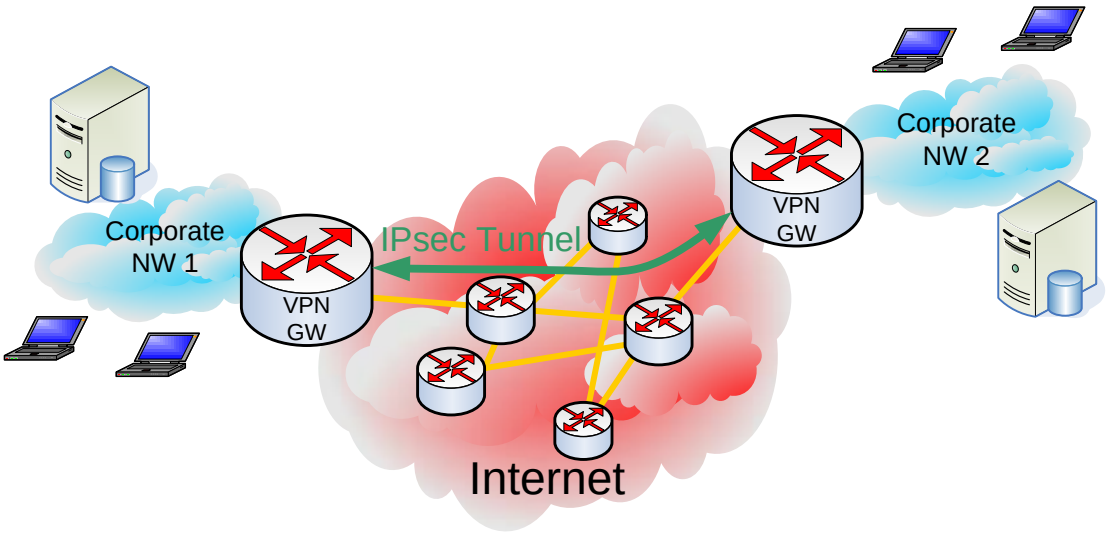


Figure 5: Confidential Data Transmission with IPsec Tunnel

Depending on the communication bandwidth between the two sites, en- and decryption of the aggregate traffic between the two sites may represent a significant processing burden for the gateway routers, which may not be handled by software processing alone, but is often handled by hardware accelerators (see also the Netronome NFP3200 [21] or SafeNet EIP-196 [19]).

Another application that uses cryptography is the domain of wireless LANs (WLAN). Due to the open nature of the wireless radio link in contrast to wireline links, all communications between the end user device and the WLAN hot spot can be

overheard by anyone without the use of cryptographic methods. Therefore the IEEE WLAN standards have published schemes like wired equivalent privacy (WEP) or Wi-Fi Protected Access (WPA) to better protect wireless networks from attacks. For the same security concerns, the digital radio channels of both GSM and UMTS mobile communication systems feature encryption technology in order to insure confidentiality of the transported information.

2.2.4. Multimedia Applications

With the increasing availability of high-bandwidth packet data networks, transmission of voice and video data over Internet networks became feasible. The fact, that transmission via packet switched networks is offered for a lower price together with potential cost savings by consolidating voice and data traffic into a single network infrastructure posed another incentive for companies to push for a converged network. In 2003, the IETF released two standards that describe the RTP/RTCP [40] protocols and mappings for voice and video data into RTP streams [41] to allow for transmission of voice and video streams over classical IP networks. RTP is typically used on top of UDP to provide sequence numbers and time stamps for the otherwise unprotected datagram delivery protocol. Transmission of real-time data using the TCP protocol that already insures correct packet sequence at Layer 4 is not advisable, as the delays caused by the TCP protocol e.g. in case of packet loss or reordering is not acceptable for interactive communication. However, the RTP/RTCP protocols alone are not sufficient to implement a voice-over-IP (VoIP) system [42], as it contains no signaling protocol. For this purpose, protocols like session initiation protocol (SIP) [43], [44] or H.323 have to be used. These protocols negotiate the call parameters between two or more endpoints (e.g. used codec, port numbers for RTP and RTCP connection for both directions, bandwidth reservations, etc.) before the actual RTP connection can be established to transport the digitized voice samples in an appropriate format (see Figure 6). Apart from the communication via SIP/RTP protocols, commercial VoIP providers like, for example, Skype have also developed their own, proprietary protocols to achieve IP-based telephony services.

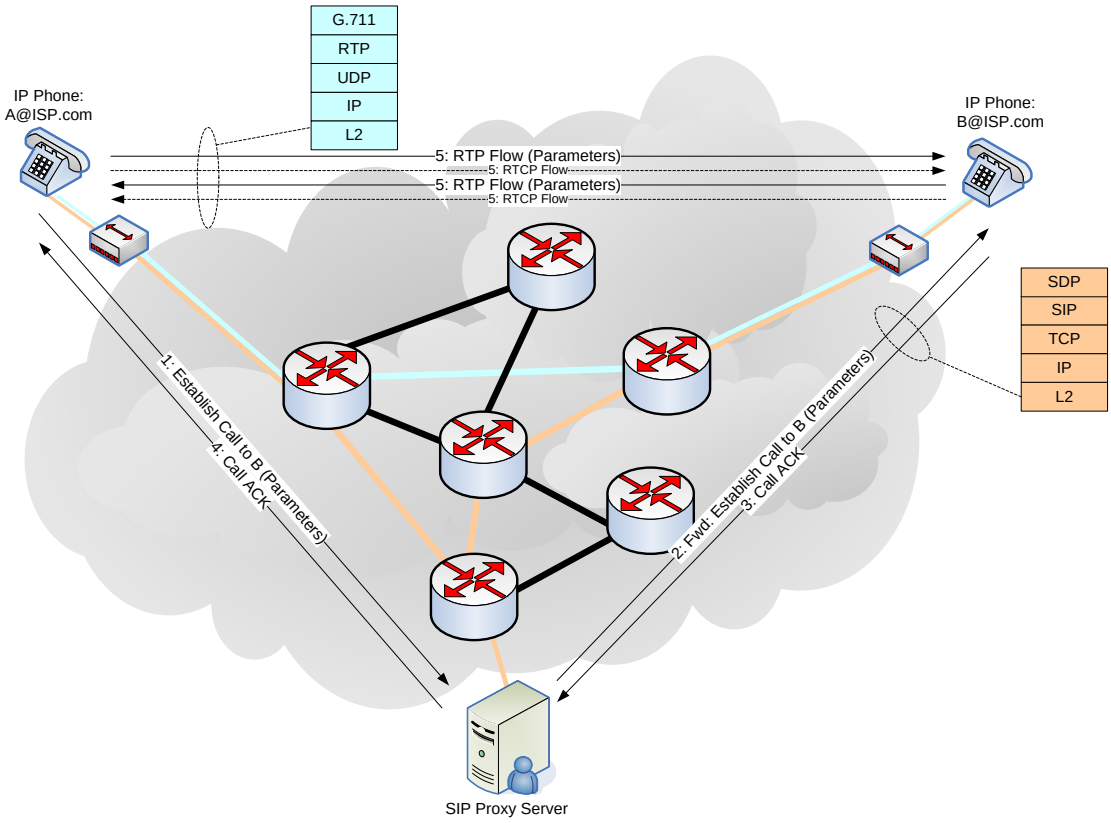


Figure 6: Simplified Connection Setup and Protocol Stack for VoIP

When transmitting voice calls over the Internet, the digitized voice samples are coded using one of the traditional telephone standards, like G.711 (ISDN), G.726 (ADPCM) or the GSM voice codec. Depending on certain connection parameters, a packetization interval is chosen, from which all coded samples are assembled into a single RTP packet payload. At the receiving end, the voice samples are retrieved and stored for a predefined period to compensate for possible packet reorderings or transmission jitter. However, a maximum transmission delay of more than 150 ms may already cause a significant deterioration of the user's quality of experience. The RTCP connection that is established along with the RTP flow constantly monitors the connection quality, and may trigger a change of important parameters like the used codec or the packetization interval to minimize negative effects for the users.

In general, it is important for VoIP applications that there is little or no packet loss and packet reordering, and a low end-to-end latency. This can generally not be guaranteed by the traditional best effort forwarding of UDP packets in the Internet. Consequently, network providers willing to promote use of VoIP services over the Internet have to undertake certain measures to prioritize such traffic over other flows. The DiffServ architecture referenced in section 2.2.2, in combination with call acceptance policies (RSVP, SIP), can be an adequate means to insure a timely and reliable delivery of VoIP traffic over a packet switched network. However, there are

two main challenges to effectively implement the required QoS on an end-to-end communication path:

- The RTP connections use dynamically assigned (i.e. random) UDP port numbers that are negotiated during the connection setup phase by one of the established signaling protocols. If the ISP is not able to wiretap the call setup traffic, it will later not be able to differentiate between the corresponding RTP packets and any other (possibly low priority) UDP traffic.

- Even if the provider knows about the negotiated connection parameters and assigns the traffic to a high-priority DiffServ traffic class, the set DSCP value might not be regarded by other ISPs, when the packets are delivered outside the original service provider's network.

For video applications transmitting over the Internet the situation is similar to that of VoIP. If it is only a unidirectional connection (like e.g. viewing a video on YouTube or watching a TV program online), it is possible to provide larger intermediate buffers on the receiver side in order to compensate for reordered packets and packet jitter. If the video belongs to an interactive videoconferencing session, the same delay considerations as for VoIP hold. Of course, the required bandwidth to deliver video in an acceptable quality for the end user is significantly higher than that of pure voice transmission.

2.2.5. Mobile Networks

The following section, which summarizes some of the findings in [45], gives a short introduction to the network backbone in mobile data networks. The following parts will focus on the data plane network topology and protocol stacks, as this is most relevant for FlexPath applicability in chapter 3. Further details about GSM and UMTS network architecture and network elements are found in [45].

2.2.5.1. UMTS-PS Network Topology

Although explicit data about real network topologies is not publicly available, some conclusions may be drawn from the physical layer dimensioning of the individual links in the system and analyses in [46] and [47].

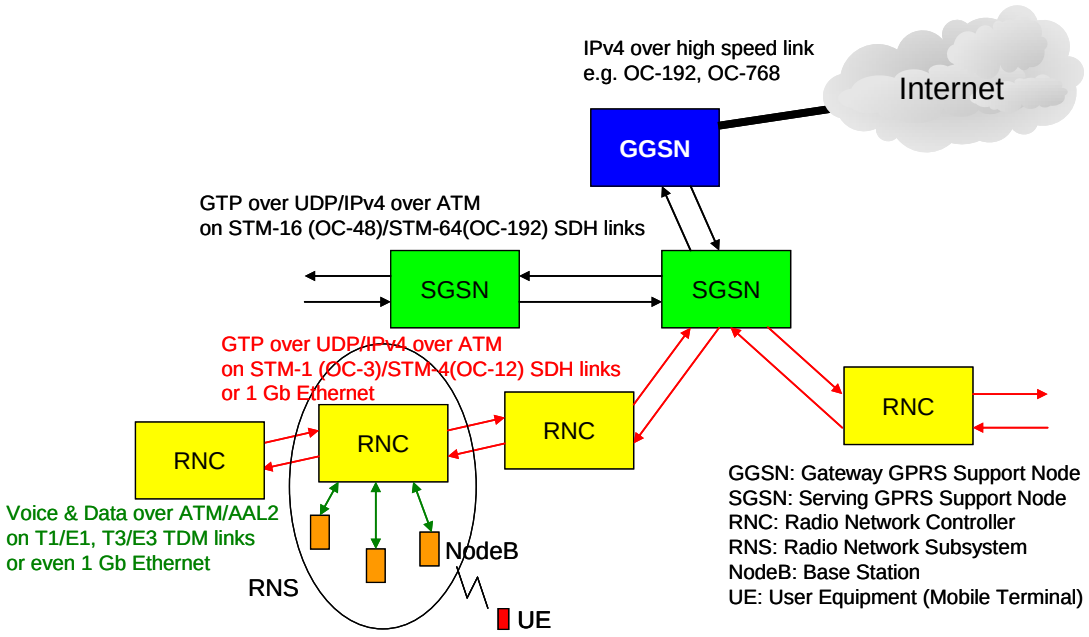


Figure 7: Exemplary Network Topology of a UMTS Packet Domain Network

Figure 7 illustrates an extrapolated snapshot from the PS (packet switched) domain of UMTS. As the link speeds of the node interconnects increase on each level of hierarchy from the base stations towards the core network elements, aggregation factors for each hierarchy level can be estimated. Assuming a peak data rate of 2 Mbit/s per radio cell and a base station serving three cells (120° sector antennas), an individual NodeB would carry up to 6 Mbit/s of traffic. Newer modulation schemes as found in HSDPA and HSUPA would raise the figures into the order of 3×20 Mbit/s=60 Mbit/s.

Table 1: UMTS Backbone Aggregation Factors

Link	*Interconnect Standard*	*Aggregation (cells)*	*Aggregation (from lower hierarchy)*
Radio Cell	2 Mbps radio	1	n/a
NodeB-RNC	E1/STM-0	1/25	1 to 25
RNC-SGSN	STM-1/STM-4	75/300	3 to 12
SGSN-GGSN	STM-16/STM-64	1,200/5,000	4 to 64
GGSN-External	STM-64/STM-256	5,000/20,000	1 to 16

Table 1 lists typical interconnect technologies on the different links and extracts the resulting aggregation factors for each hierarchy level based on 2 Mbit/s traffic per cell.

The RNC is responsible for many control plane tasks for the attached NodeBs. These functions are typically mapped to software and are hardly suitable for hardware acceleration. While the traffic to and from the locally attached NodeBs is

less than 51 Mbit/s (STM-0 rate) and the RNCs are connected with peering links among each other and towards the SGSN with STM-1 or STM-4 links, an aggregation factor of 3 to 12 may be derived, i.e. up to 12 RNC devices can be chained towards a single SGSN interface. Using the same method, it can be assumed that up to 64 SGSNs are linked towards a GGSN, where an additional traffic aggregation can be observed towards the external network links.

2.2.5.2. UMTS-PS Data Plane Protocol Stacks

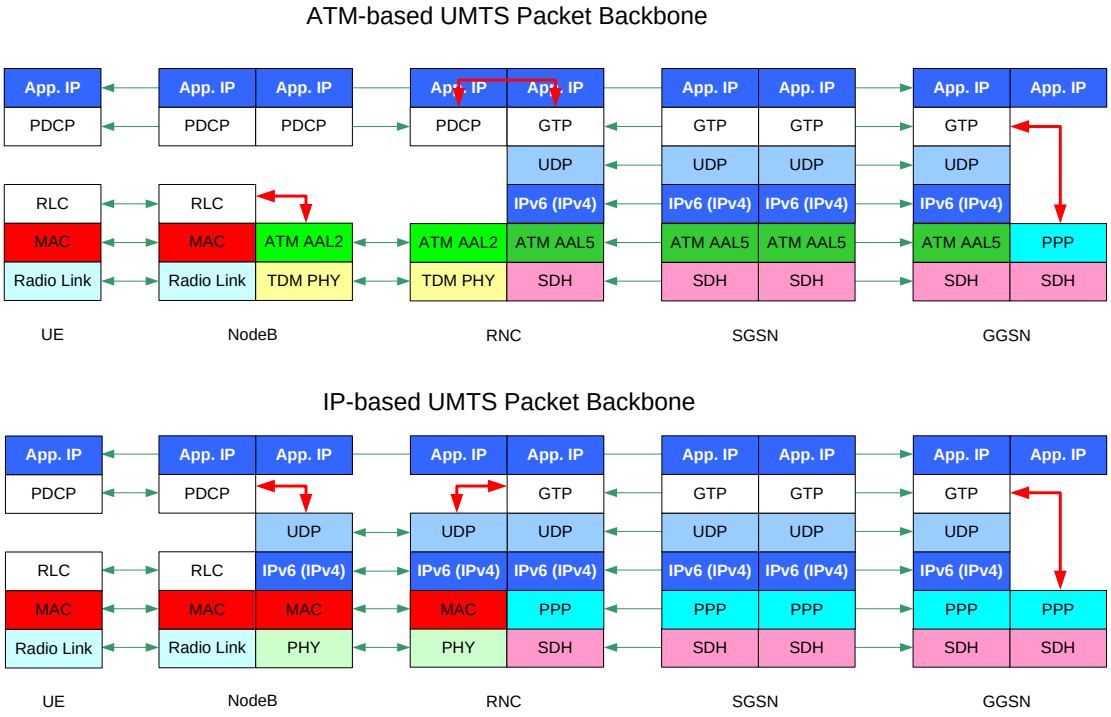


Figure 8: Data Plane Protocol Stacks of UMTS/GPRS with ATM and All-IP Backbone

Figure 8 shows the data plane protocol stacks for UMTS/GPRS with both ATM and IP as networking protocols. The original version of the UMTS standard is based on ATM, with AAL2 and AAL5 used to support both voice and data traffic in a unified network architecture. The QoS behavior of ATM with its virtual circuit connections makes this solution also interesting from a network management point of view. In the all-IP network, which is currently proposed, additional measures beyond IP are necessary to avoid interference between real-time and non-real-time traffic classes (see also 2.2.2, 2.2.4). It can be seen that the data plane functionality of the SGSN is quite simple as no protocol conversions are performed at this unit. All other network elements have to perform protocol conversions (gateway functions, cf. red arrows in Figure 8) for all traffic that is forwarded towards another hierarchy element.

2.2.6. Carrier-grade Ethernet and Internet Backbone Evolution

Originally, Ethernet was developed during the 1970s and first standardized by the IEEE in the 802.3 standard in 1983. It had been developed as a local area networking technology, used to connect computers and servers within the same building. Over the course of the years, Ethernet became the dominant LAN technology and continuous development efforts increased the transmission bandwidth from the original 10 Mbit/s to 10 Gbit/s. As of 2009, standardization work has begun defining a 100 Gbit/s standard.

Initially evolving from traditional long-distance telephony networks as only wide area networks, data network standards such as ATM, MPLS and Sonet/SDH were developed as transport architectures for packet traffic based on optical fiber technology with data rates between 155 Mbit/s (STM-1) and 39.8 Gbit/s (STM-256). These technologies were designed for supporting both digital voice and packet-based data communication over a shared infrastructure and included very efficient QoS methods and fault-tolerant redundant transmission necessary for high-availability. Due to the relatively few systems needed for the backbone infrastructure, these systems are significantly more expensive than Ethernet technology deployed in the LAN field. While the transmission speed of the classical backbone technologies was initially much larger than in the LAN technology in use at the same time, this is no longer true considering the most recent advances in the Ethernet standardization bodies.

As virtually all traffic in the Internet somehow emerges from and is destined for local area networks (both for residential customers, who typically maintain a small LAN behind their DSL-Router or cable modems, as well as content providers with their private enterprise networks) and taking into account the effort necessary on the ISP side to translate between the different transmission standards, it is increasingly attractive to transform the Internet backbone into an Ethernet-based network. In addition, operators can hope to make use of the better economies of scale, when migrating towards the higher-volume Ethernet infrastructure ([5], [48]).

However, traditional Ethernet as it was intended for LAN use, has serious scaling issues and does not implement the QoS and fault-tolerance mechanisms found in current backbone networks. Recent efforts in the IEEE 802 standards committee have added VLAN tagging with QoS marking (IEEE 802.1Q, IEEE 802.1p) in a similar fashion as the DSCP codepoints found in DiffServ-enabled IP networks (see 2.2.2). The concept of VLANs allows to provision logically separate networks over the same physical medium. In order to address the scaling limitations of traditional Ethernet (it is practically infeasible to maintain lookup tables with millions of 48-bit MAC addresses, which are globally unique, but not in any form structured in accordance with the network topology), traffic between a pair of networks or network access

points may be aggregated into distinct VLANs. As the VLAN tags only support just over four thousand such VLANs, provider backbone bridging (PBB, IEEE 802.1ad) allows to build a stack of VLAN tags at the start of the Ethernet frame that allows constructing a hierarchy of VLANs in order to transport aggregated flows from the access network through the aggregation network towards the packet core (see also [48], [49] and Figure 1 in chapter 1).

Although it could be argued that these latest developments may ultimately lead to a network architecture that would be fully implemented using Ethernet, carrier Ethernet and IP technologies; the large installed base of non-Ethernet ("legacy") networks enforces a gradual transition, where newly constructed or recently upgraded networks may be using the latest technology, but the installed base with its variety of technologies ranging from ATM over Sonet/SDH to MPLS will remain in use for the remainder of that equipment's lifetime. Consequently, there will be a continued need for gateway devices that are able to translate between these different protocols at the edges of the individual networks, in order to insure full connectivity and interoperability.

2.2.7. Conclusions

The various examples for networking applications discussed before can be categorized into two different groups with respect to the interdependence between individual packets:

- **Stateless Networking Applications**: simple IP forwarding and layer 2 switching are representatives of the stateless networking applications. In these applications, the packets can be processed individually, i.e. processing of a later packet can be achieved independently of the processing of earlier packets. This independence can be exploited very well by NP architectures with many parallel processing units, as the task of processing multiple packets can be parallelized in a straightforward fashion. In addition to plain forwarding and switching, DiffServ forwarding with different QoS priorities for various traffic classes can be regarded as a stateless networking application.

- **Stateful Networking Applications**: In contrast, when the forwarding function is appended with flow-specific information like (1) the traffic parameters in an IntServ environment, (2) connection-specific sequence numbers used for IPsec or (3) forwarding is based on higher layer connection information found in gateway functions, the networking application relies on some kind of state information. An important aspect is that the state information must be updated after processing a packet and the processing of the subsequent packet relies on the results triggered by previous packets from the same connection. When mapping such stateful applications on parallel processing units, caution has to be exercised in

order to insure both the consistency of the state information and the correct processing sequence of the individual packets from each different flow.

But the regarded networking applications may also be classified according to the individual processing requirements within the network-internal nodes and with respect to their feasibility for hardware support:

- IP forwarding can be accomplished with a few relatively simple operations. These can be accomplished in an optimized way either by using processors with customized instruction sets or dedicated hardware accelerators with limited configurability. Cryptographic algorithms also have a regular structure, but are very computationally expensive. For this reason, the en- or decryption is usually transferred to hardware accelerators, if a significant share of the traffic has to be protected.

- However, control and management of the state information (SPD and SAD databases) require general-purpose calculations, which are usually not moved to dedicated hardware. Changes in the operational details of the protocols, e.g. improved key exchange protocols, also require an architecture that can be easily adapted in the field. The same is also true for the interworking functions in gateway devices or deep packet processing applications like virus scanning or intrusion detection, where entire protocol stacks including layers 4 and higher have to be processed.

The latter classification of networking application characteristics is already reflected in the offered mix of programmable units and customized hardware in current commercial NP designs (see chapter 2.1). The FlexPath NP architecture, which will be presented in chapter 3, optimizes the performance by providing different processing paths (i.e. software / hardware unit traversal sequences) that are best suited for the different traffic types. Application classes of the arriving packets have to be identified, and then the packets can be dispatched to the best fitting processing path. In addition, by differentiating stateful and stateless applications, it is possible to apply a combination of load balancing techniques, which are well suited for either case.

2.3. Packet Classification

Packet classification is a necessary task in all current network processing devices and has to be executed in various fashions depending on the application requirements. Packet classification algorithms can be classified into:

- **Single-field classification:** a decision on the further processing of a packet depends only on a single header field. This is for example the case for simple routing lookups, which rely only on the IP destination address field. QoS-aware forwarding within a DiffServ domain, where the DSCP field in the IP header determines the service class of the packet is also a single-field classification problem. State-of-the-art techniques for single field classification, many of which are also used as components in multi-field classification algorithms, are presented in section 2.3.1.

- **Multi-field classification:** More complex applications like access control / firewalls, flow specific processing, etc. base the action on multiple header fields. The most important example is the Internet five-tuple, which consists of the IP source and destination addresses, layer four protocol and layer four source and destination port numbers. The Internet five-tuple is generally conceived to unambiguously describe an individual flow (i.e. connection) between any two parties. State-of-the-art techniques for multi-field classification algorithms are later described in section 2.3.2.

As I will show later in chapters 3.2 and 4, the Path Dispatcher, which determines the best suitable processing path of the arriving packets, contains a reconfigurable rule base that effectively performs a multi-field packet classification. The heterogeneous decision graph algorithm (HDGA) proposed later in chapter 4, is based on some ideas of existing classification schemes and optimizes them for the specific environment faced in the FlexPath Path Dispatcher.

2.3.1. Single-Field Classification

The simplest form of classification is based on only one field. A practical example of such a single-field classification is the routing lookup, where packets at the router have to be classified according to their IP destination address. The extracted address has to be matched to the entries in the routing table, and the packet is forwarded to the interface stored next to the matching address entry. In case of the IP next-hop lookup, the entries of the routing table can be either fully specified IP addresses, or address prefixes. The packet then has to be forwarded to the interface associated with the longest matching prefix, i.e. the prefix with the highest number of corresponding bits. The following subchapters present some basic search techniques that are used to find entries corresponding to a given search key.

2.3.1.1. Linear Search

The most basic search mechanism is linear or sequential search ([50], chapter 6.1, pp. 396 ff.). At first, the keys of the database are stored in a list. When the search commences, the list is checked from the beginning until the key corresponding to the searched item is found. If the searched string is not in the list, the search is unsuccessful. The average search time for a list with N entries is $\frac{N+1}{2}$, if all entries are sought with equal probability. Thus the search complexity is O(N). Considering the example database of Table 2, we observe an average search time of 5.5 cycles.

Table 2: Linear Search Table for Example Database

Entry	*Value (Key)*	*Key in binary format*
1	67	100 0011
2	27	001 1011
3	56	011 1000
4	32	010 0000
5	75	100 1011
6	29	001 1101
7	50	011 0010
8	39	010 0111
9	10	000 1010
10	84	101 0100

Due to the search complexity of O(N), it is apparent that linear search does not scale well for larger tables. However, the basic scheme can be used with any list of elements that do not have to be sorted in any kind. Therefore, linear search can be beneficial especially in cases, where it is hard to establish an ordered list, e.g. by frequently updating the entries in the table.

The search performance can be improved, if linear search can be applied to an ordered list, where there are basically two possibilities. If the list is ordered by the numerical order of its elements, the search can be stopped when the first value greater (or less) than the requested key is found. Therefore, searching for an element that is not contained in the list is accelerated. The second optimization would be to sort the elements in descending order with respect to the search frequencies. In this way, keys that are searched more often are situated at the beginning of the list and are in consequence found earlier. In case the frequency is not known beforehand, there are also adaptive schemes proposed, which update the sequence of the list during the search operations, so that the list is self-adapting towards the current operational environment.

In the networking domain, linear search can often be found as search technique for collision resolution in hashing-based searches (see section 2.3.1.4).

2.3.1.2. Binary Tree Search

Binary tree search (see [50], chapter 6.2.2, pp. 426 ff.) requires that the elements of the database are sortable into order. Starting from a selected root element, a tree structure is generated with two children per node. Elements that are smaller than the value stored at the inspected node are stored in the left sub-tree, larger elements in the right sub-tree. When a search is initiated, elements are compared starting at the root node and if the search string is not found either the right or left sub-tree is searched recursively. As the number of elements stored in each level of the tree increase by a factor of two for a balanced tree, the search complexity is reduced to $O(\log_2 N)$. This reduction in complexity makes binary search a very attractive method also for large databases.

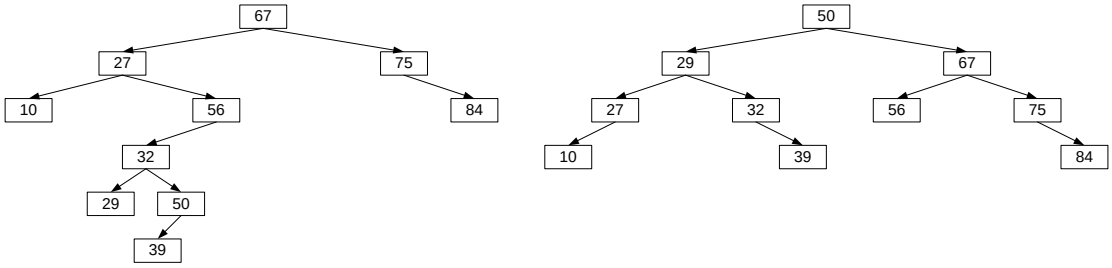


Figure 9: Binary Search Trees for Example Database

However, if the keys are already in a sorted order when inserting them into the tree, a degenerated tree may result that approaches linear search as a worst case. Therefore, tree balancing schemes are used in situations, where frequent updates of the database occur, in order to obtain a near-optimum performance (see also [50], chapter 6.2.3, pp. 458 ff.).

This problem is illustrated in Figure 9, where the entries of Table 2 are inserted into the tree in an unmodified order (left tree). This tree has a maximum depth of 6 nodes and requires on average 3.4 cycles to find the searched value. In the right tree, the insertion order has been changed such that the maximum tree depth is reduced to four levels ($\lceil \log_2 10 \rceil = 4$). The average search time for equally likely values is reduced to 2.9 cycles. As N=10 is not a power of two, some nodes at the fourth level are unoccupied.

2.3.1.3. Binary Tries

A binary trie (derived from the word reTRIEval and pronounced "try"; [50], chapter 6.3, pp. 492 ff.) looks similar to a binary tree at first. However, instead of storing the keys within the nodes along with the two child pointers and comparing the key to the searched element, the position in the trie already determines the actual

contents. The binary trie treats a number as a string of a certain length. The root node contains the empty string. From here, a branching is made recursively with the left child appending a '0' to the string and the right child appending a '1'.

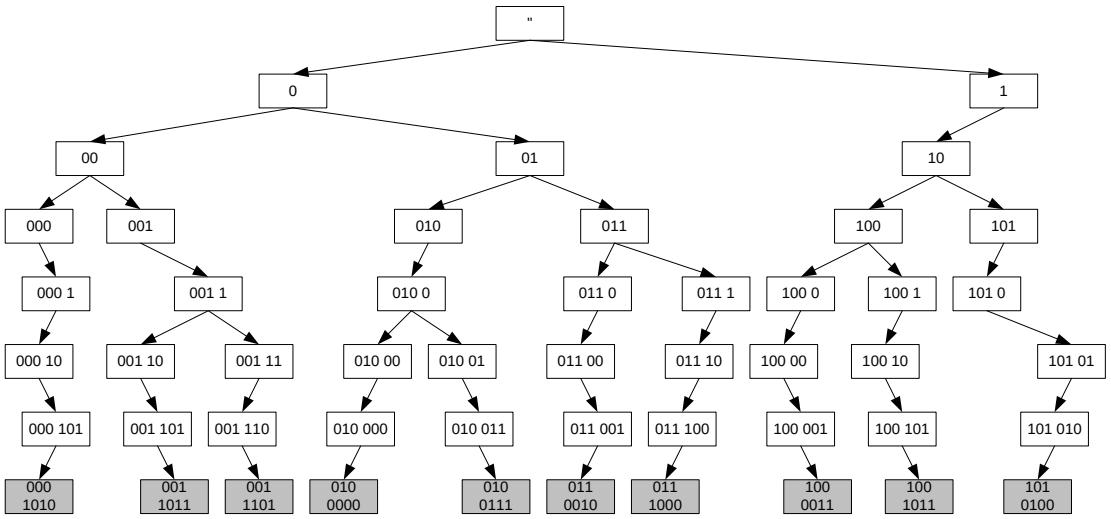


Figure 10: Binary Trie for Example Database

Figure 10 shows the binary trie for the same database as presented in Table 2 and Figure 9. In order to get the binary trie, we have to convert the keys to their binary representation first. The labels within the individual nodes are featured for clarity only. The actual trie would only store markers (grayed fields) for the nodes representing actual values from the database. White nodes or nodes that are not linked within the trie lead to nodes that do not exist in the original database. Searching the trie takes seven cycles, as all database entries can be symbolized as 7 bit numbers. In general, the search time can be expressed as O(w) for any database where the largest member can be represented with w bits.

Binary tries can also be used very efficiently, when the longest matching prefix for a given key is sought as this is the case in CIDR routing table lookups (see chapter 2.2.1).

An effective variant to reduce both the size and search time complexity for tries has been proposed by Morrison [52] with the PATRICIA (PATRICIA is an acronym for "Practical Algorithm to Retrieve Information Coded in Alphanumeric") tries. PATRICIA is based on a binary trie, but nodes that have only a single child are skipped. Instead, the nodes are containing information, at which bit position the next test has to be performed. Only the final node contains a copy of the original string that has to be compared to the search word in order to verify an actual match. Figure 11 shows the PATRICIA trie corresponding to our example. The PATRICIA trie has N-1 internal nodes plus N leaves pointing to the keys of the actually stored strings.

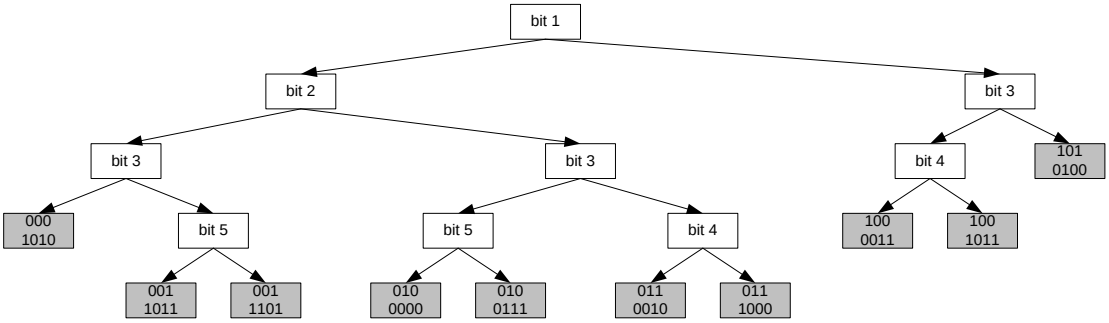


Figure 11: PATRICIA Trie of Example Database

If a PATRICIA trie is constructed for variable length prefixes of IP addresses, the longest matching prefix may be found very effectively as all intermediate prefixes are passed on internal nodes during the search operation from the root towards the leaf nodes. In addition, new entries may be inserted or deleted from trie data structures with little effort, as the structure of the trie is directly related to the contents and no complex re-balancing operations have to be performed.

2.3.1.4. Hash Table Search

The final search algorithm of the classical single-field searches presented in this work is hash table search (see [50], chapter 6.4, pp. 513 ff.). In contrast to the previously discussed search techniques, the search is not performed on the keys itself, but a hash value h(k) is computed from the key k using the hash function h. An important property of hash functions is that the hash value h(k) has fewer bits than the original search key k. Basically, any function may be used for hashing, but in order to obtain a good search performance, functions with certain mathematical properties have to be chosen.

Hash table searching is intended in areas, where there are far fewer entries in the database than the theoretically possible number of entries given the width of the search keys. Instead of working on tables or lists with the original length, the hash function is used to compress the search space to a significantly smaller space. Consider the example database of Table 2 and focus on the binary representation of the stored values. We need seven bits to encode the numbers in range smaller than 100, but we have only 10 entries. We could choose a hash function h(x)=x mod 16, effectively regarding only the four least significant bits of every key. Table 3 shows the resulting hash table for the example database of Table 2.

When searching for a specific key, e.g. 39, we first compute the hash function h(39) = 39 mod 16 = 7. Now the hash table is inspected at position 7, and the matching entry is immediately found. Thus, the search can be completed with a single lookup operation (plus the computation of the hash function), which yields an optimal search complexity of O(1). However, when searching for the key 75, a different behavior can be observed. The index h(75)=11 lists two entries, namely 27 and 75.

This effect is called collision and happens, when two different keys from the initial database evaluate to the same hash value. As the hash space (in our example 2^4=16) is smaller than the number of possible values (100), collisions are unavoidable, if the number of entries in the table exceeds the size of the hash space. But collisions can also not be excluded if this requirement is fulfilled. The amount of collisions happening when the table size is smaller than the hash space depends on both the used hash function and the entries of the database. From a theoretical point of view there exist "perfect hash functions" that produce collision-free distributions of the keys. In practice, different classes of hash functions are used for general-purpose applications that try to find a suitable compromise between computational complexity, balanced distribution of the input values to the hash space and applicability for key sets that are unknown during design time. In the networking field, cyclic redundancy checks are a popular choice for constructing hash tables [51].

Table 3: Hash Table for Example Database

Index	*Contents*
0000	010 0000; EOL
0001	EOL
0010	011 0010; EOL
0011	100 0011; EOL
0100	101 0100; EOL
0101	EOL
0110	EOL
0111	010 0111; EOL
1000	011 1000; EOL
1001	EOL
1010	000 1010; EOL
1011	001 1011; 100 1011; EOL
1100	EOL
1101	001 1101; EOL
1110	EOL
1111	EOL

When using imperfect hash functions, the hash table lookup algorithm has to provide means to tackle collision resolution. One popular and simple method is implemented in Table 3: chaining. Colliding entries are stored in a linked list (see also 2.3.1.1) under the corresponding index. This list has to be searched sequentially during collision resolution and increases the total search time from O(1). A worst case upper bound of O(n) would be achieved, if a degenerate database contains only colliding entries that all map to the same hash value. However, for real world problems with good hash functions, the collision probability is small, so that

the actual overhead remains well limited and hash table searches achieve attractive performance results. Alternative collision resolution schemes like subsequent hash searches, tree structures, etc. exist, but shall not be further discussed within the focus of this thesis.

2.3.1.5. Content Addressable Memories

In contrast to the algorithmic search techniques presented in the previous subchapters, which may be implemented either in software or hardware, content addressable memories (CAM) are hardware solutions that are specially designed for search problems. They are currently widely employed as standard co-processors in commercial solutions for both routing lookup (single-field packet classification) and firewall filtering (multi-field packet classification) applications in network search engines (NSE) [53], [54].

The underlying working principle of CAMs may be understood as that of an inverted SRAM memory. In the classical SRAM, memory cells are grouped into words of distinct width (e.g. 32/64/256 bits) holding the actual information. The address width is derived from the number of words contained in the memory device, e.g. a 4 MB memory with 32 bit word width would be 1M words large and is addressed by a $\log_2(1M) = 20$ bit address. When the user puts a certain address on the device's address bus, the data stored in the corresponding memory cell is retrieved and delivered on the device's data bus.

In a CAM device, the contents of the database are stored in the memory cells during the initialization. When the user searches for a specific key in the database, he will put the data on the data bus of the device and the CAM performs a parallel comparison of the data word to all contents in the memory. If the data is stored in the CAM, the address of the corresponding cell is delivered on the address bus and may be used to retrieve further associated data that is stored in a traditional memory.

In contrast to simple CAMs, which are based on a single SRAM cell per CAM cell and store only the distinct values 0 and 1; ternary CAMs (TCAM) are also available that make use of a second SRAM within each CAM cell and allow storing "don't care" values. When a search is requested, matches are reported if the '0' and '1' bit positions in the database entries and on the data bus correspond, skipping the contents in the "don't care" positions. In this fashion, it is easily possible to perform searches with wildcard parameters, prefix matches and certain types of range matches. The big advantage is that for such entries a single TCAM word is able to represent a multitude of exact matching values, which is very space efficient.

As the matching logic within the CAM/TCAM solutions adds additional overhead compared to plain SRAMs, the access times are not that fast and are typically in the

range of several tens of ns. A major drawback of TCAMs is the high power consumption as all memory cells of a database are activated during a search operation.

2.3.2. Multi-Field Classification

While searching large databases of only a single packet header field under hard real-time conditions on multi-gigabit/s links may already be quite a challenging task, the problem gets even harder when several (independent) fields from the packet header have to be inspected. All applications that rely on flow identification (e.g. firewall filtering, IPsec, etc.) involve identifying the associated information from either the Internet five-tuple or a combination of further fields from the packet header.

Mathematically, the problem of multi-field packet classification on n fields can be interpreted as finding the highest-priority rectangle in n-dimensional space, which contains the point defined by the packet's header fields [55]. Gupta et.al. show that for realistic rule base sizes, the performance bounds of algorithms known from computational geometry are infeasible in the networking environment. For rule bases with N rules over a d-dimensional space (i.e. d different header fields) the bounded complexities are either O(log N) search time with $O(N^d)$ storage or $O((\log N)^{d-1})$ search time with O(N) storage space. Therefore, heuristic algorithms that try to exploit characteristic features from the application domain have been developed.

Multi-field packet classification has been established as an industry standard taking the structure of Cisco access control lists (ACLs) as a common template [61], [67]. In addition to specifying exact match address values or prefixes, the Cisco ACL may contain wildcards and range specifications for the port numbers. In realistic rule bases, it is also possible that several rules overlap. This effect can also be seen in the example rule base shown in Figure 13 in chapter 2.3.2.2, where rules 0 and 5, 2 and 5, and 3 and 5 partially overlap each other. Therefore, in addition to the rule specifications, a priority is assigned with each rule, such that the highest-priority rule can be chosen in such cases. This prioritization and the resulting danger of finding several matching rules within a given region of packet values further complicate the classification problem.

The packet classification problem can be formalized in the following way ([55], [58], [60], [61]):

Given are d header fields for each packet P that are relevant in the classification problem

$$P[0], P[1], ... P[d-1] \tag{2-1}$$

The rule base or filter set B of size N is a prioritized list of rules R_p, i.e. the rule index p, $p \in \{0,1,..., N-1\}$ is also the priority of the rule and each rule consists of d expressions E[i] on all possible header fields P[i]

$$B = \{R_0, R_1, ..., R_{N-1}\} \tag{2-2}$$

$$R_p = \{E[0], E[1], ..., E[d-1]\} \tag{2-3}$$

The following types of expressions are found in most practical rule bases, although further expressions with a Boolean result are conceivable:

– Exact Match:

$$E[i] : P[i] = value \tag{2-4}$$

$$E[i] : P[i] \neq value \tag{2-5}$$

– Wildcard Match:

$$E[i] : (P[i] \wedge mask) = value \tag{2-6}$$

– Range Match:

$$E[i] : value1 \leq P[i] \leq value2 \tag{2-7}$$

– Prefix Match:

$$E[i] : \left\lfloor \frac{P[i]}{2^m} \right\rfloor = prefix \tag{2-8}$$

An incoming packet matches the rules in M with

$$B \supseteq M = \{R_m | R_m : \forall i \in \{0,..., d-1\} : E[i] \text{ is true, given } P[i]\} \tag{2-9}$$

and

$$R_h ; h = \min\{i | R_i \in M\} \tag{2-10}$$

is the highest priority matching rule.

Table 4: Example Rule Bases B with d=2 and N=7 (a, left) and d=3 and N=7 (b, right)

$R_0 = \{0 \le P[0] \le 31, 24 \le P[1] \le 31\}$	$R_0 = \{0 \le P[0] \le 31, 24 \le P[1] \le 31, P[2] = 0\}$
$R_1 = \{0 \le P[0] \le 3, 32 \le P[1] \le 63\}$	$R_1 = \{0 \le P[0] \le 3, 32 \le P[1] \le 63, P[2] = 0\}$
$R_2 = \{4 \le P[0] \le 19, 32 \le P[1] \le 47\}$	$R_2 = \{4 \le P[0] \le 19, 32 \le P[1] \le 47, P[2] = 1\}$
$R_3 = \{0 \le P[0] \le 63, 0 \le P[1] \le 7\}$	$R_3 = \{0 \le P[0] \le 63, 0 \le P[1] \le 7, P[2] = *\}$ [1]
$R_4 = \{48 \le P[0] \le 63, 16 \le P[1] \le 31\}$	$R_4 = \{48 \le P[0] \le 63, 16 \le P[1] \le 31, P[2] = *\}$
$R_5 = \{4 \le P[0] \le 15, 0 \le P[1] \le 39\}$	$R_5 = \{4 \le P[0] \le 15, 0 \le P[1] \le 39, P[2] = *\}$
$R_6 = \{24 \le P[0] \le 47, 32 \le P[1] \le 63\}$	$R_6 = \{24 \le P[0] \le 47, 32 \le P[1] \le 63, P[2] = 0\}$

Table 4 introduces two example rule bases with range matches in two dimensions and exact/wildcard matches in a third dimension that will be used later to illustrate some of the discussed classification algorithms.

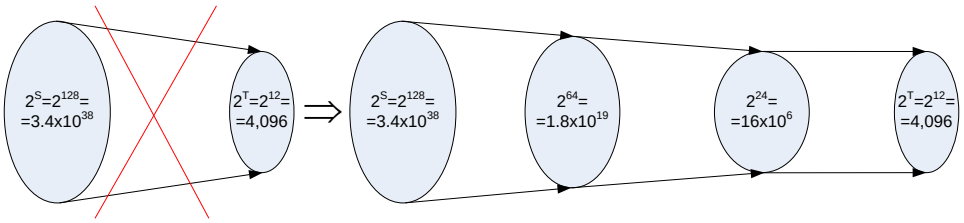
The following chapters present the wide range of state-of-the-art multi-field packet classification techniques.

2.3.2.1. Recursive Flow Classification (RFC)

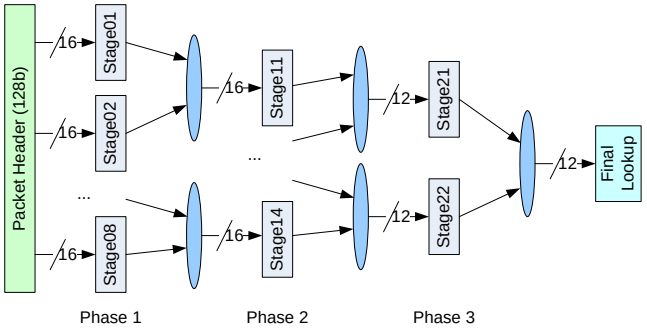
Recursive Flow Classification (RFC) has been proposed by Gupta and McKeown in 1999 [57]. The basic idea behind RFC is to find an efficient mapping from a long key (actually the concatenation of all relevant header fields from the current packet) towards a short index, which describes the appropriate action. It is practically impossible to pre-compute the action for each key and then look up the result in a single step, since this would require a memory with 2^S entries, where S is the concatenated length of all relevant header fields. Therefore, a multi-stage approach is chosen. At first, the concatenated header fields are split into several shorter sub-keys. Each of these keys is used as an address for a memory. The obtained values from the first stage are combined and yield the addresses for the memories of the subsequent step. In the final step, the action can be calculated by combining the lookup results from the last stage. Figure 12 illustrates the working principle of RFC. The S=128 concatenated bits from the packet header are reduced to a T=12 bit classification result in three phases using a total of 14 memory blocks.

[1] Wildcard Match P[2]=* may also be expressed as P[2] ^ 0 = 0 in line with the formal definition above. The asterisk is a common shorthand notation for wildcards.

[2] These calculations assume transmission of 40-byte and 1500-byte IP datagrams over PoS and Ethernet media. Protocol overhead calculations for PoS include SDH overhead, 9 bytes



a) Reduction from concatenated header fields (e.g. 128 bit) to classification result (e.g. 4k actions)



b) Three-phase RFC classification: 128 bit input with 16 bit chunks in first stage; 12 bit classification result

Figure 12: Working Principle of RFC

In [57], Gupta and McKeown state that they had investigated real-world classifiers with 1700 rules in four dimensions and the RFC algorithm supports classification at up to 10 Gbit/s line rates. However, both the storage requirements for the rule table and pre-processing time (essential for dynamic updates of the memories when the rule base changes) grow rapidly for classifiers with more than 6000 rules. In addition, the RFC algorithm has no incremental update scheme, i.e. changes in the classification rule base may lead to a complete recalculation and reconfiguration of the stage memory contents.

2.3.2.2. HiCuts/HyperCuts

HiCuts [59] is a decision-tree based classification scheme also proposed by Gupta and McKeown in 1999. The HiCuts classification may be explained by approaching the packet classification problem from its geometric interpretation. Figure 13 shows the two-dimensional rule base of Table 4a in its graphical representation.

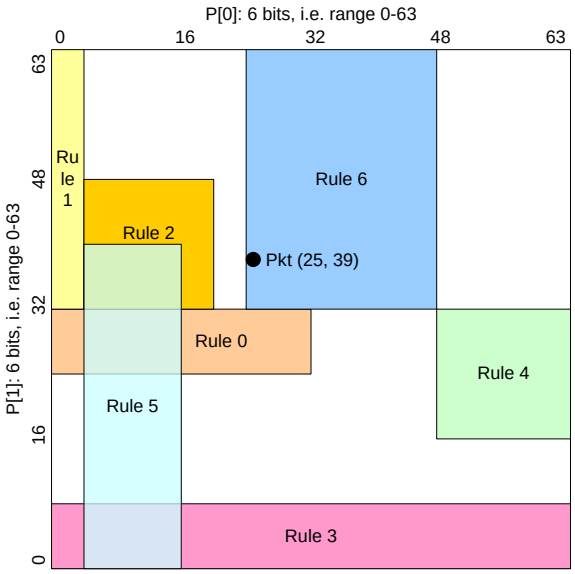


Figure 13: Graphical Representation of B from Table 4a

At the root node of the decision tree, the entire d-dimensional space is represented. In each node, the d-dimensional space is split up into n equally sized sub-spaces by cutting one selected dimension into n equally sized intervals. This process is continued iteratively until a pre-defined number of classification rules are remaining within the reached sub-space that may be resolved by a final linear search step. Figure 14 shows the operation of HiCuts for an arriving packet with values (P[0]=25, P[1]=39).

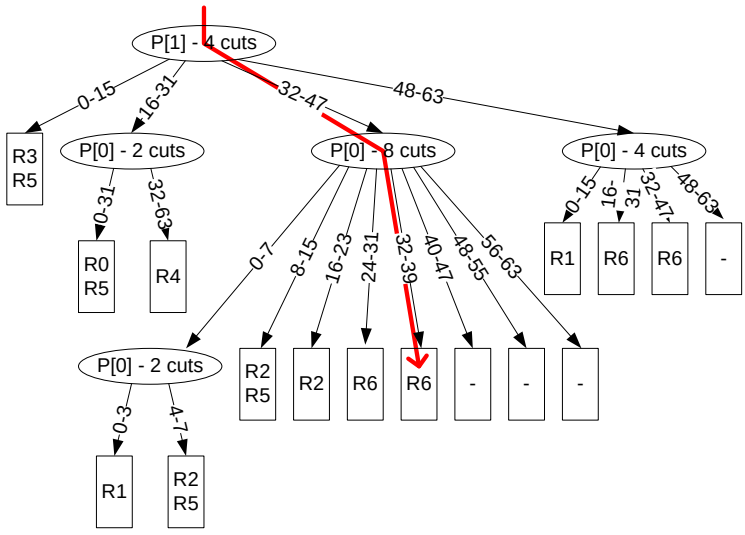


Figure 14: HiCuts Tree; at most 2 Rules for Linear Search and 8 Cuts per Tree Node

Singh et.al. extended the HiCuts algorithm in 2003 by allowing cuts in several dimensions to happen within a single node and supply an appropriately updated heuristic to generate the resulting tree (Figure 15). Allowing to cut along several dimensions instead of sticking with a single dimension in each tree node, reduces the resulting decision tree for realistic classification rule bases to fewer levels. In

[58], the authors quote that they require 2 to 10 times less storage space for a HyperCuts tree compared to HiCuts and the worst case search time is ranging between 50% and 500% better than HiCuts.

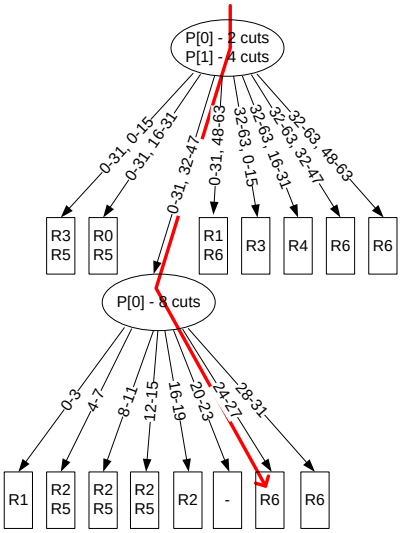


Figure 15: HyperCuts Tree; at most 2 Rules for Linear Search and 8 Cuts per Tree Node

As both schemes cut the interval of each dimension into equally sized sub-intervals, it may take several successive cuts (i.e. tree levels and correspondingly memory accesses / search time) in order to resolve exact matches on a specific field. This behavior would deteriorate the performance most, if the rule base contained exact matches for long header fields, like e.g. the 32 bit IPv4 addresses. Both algorithms use the maximum number of cuts allowed to be performed at each node (i.e. number of children) and a bucket size for the remaining rules in a specific region as tunable parameters during tree construction. By appropriately choosing these parameters, search time, search complexity and tree size can be traded against each other. HyperCuts additionally allows choosing the maximum number of dimensions along which the cuts may be taken at each tree node.

2.3.2.3. Grid-of-Tries

The Grid-of-Tries classification algorithm was first described by Srinivasan et.al. in 1998 [60] and may be applied for two-dimensional classification, especially source-destination IP address pairs. A first trie is constructed for the destination addresses found in the classification rule base. Each of this trie's leaves contains a pointer that leads to a source address trie containing all source addresses that share the same destination address in the rule base. They state in their publication that although an extension of the scheme towards higher dimensions (protocol field and port numbers) would be theoretically possible; but the algorithm would not perform well. The performance degrades significantly when the rule base includes range specifications for the L4 port fields. Therefore, Cross-Producting (see 2.3.2.4) is proposed for five-tuple classification.

In 2003, Baboescu et.al. [61] revisited the grid-of-tries algorithm and proposed some modifications to support realistic five-tuple classification. The adapted scheme is called EGT-PC (extended grid-of-tries with path compression). The authors investigated several rule bases obtained from Internet Service Providers (ISPs) and found out that across all these classifiers not more than 20 rules would share the same source / destination address pair. In most cases, there would be only between 3 and 5 rules per address pair. Therefore, they propose a two-stage classification algorithm that would find all potentially matching source / destination address pairs for an incoming packet and then linearly search the list with remaining rules. As the address specifications often include prefixes in addition to full addresses, a modification in the grid-of-tries structure is necessary in order to cover all prefixes without backtracking in the trie. Additionally, they propose to compress the source and destination address tries in a similar fashion as in PATRICIA (see 2.3.1.3).

In 2006, Pao and Liu [62] presented a further refinement to the EGT-PC scheme that makes it better scale for larger classification rule bases and IPv6 addresses.

2.3.2.4. Bitmap-Intersection and Crossproducting

Bitmap-intersection and crossproducting are very similar classification schemes that have been independently developed and presented at SIGCOMM 1998.

In bitmap-intersection, proposed by Lakshman and Stiliadis [63], the multi-dimensional search problem is at first broken up into a set of one-dimensional range searches. In a pre-processing step, a N-bit (for N rules) bit-map is calculated in which for each interval in the given dimension the n^{th} bit is set if rule n is contained within the respective interval. During the search operation, the packet's fields are used to determine the d intervals, in which the header fields lie. The d bitmaps may then be combined by a logical AND-operation in order to find all rules matching the packet fields in all dimensions. The searches in d dimensions may be parallelized in a hardware implementation in order to save search time and increase the performance of the algorithm. One drawback of bitmap-intersection is its storage complexity of $O(dN^2)$ [55], i.e. a classification rule base in d dimensions with N rules scales quadratically in the number of rules. Table 5 shows the bitmap-intersection scheme for the two-dimensional rule base presented in Table 4a.

Table 5: Bitmap Intersection - Intervals and Bitmaps

Dimension	*Interval #*	*Interval*	*Associated Bitmap*
0	1	0 ≤ P[0] ≤ 3	1101000
0	2	4 ≤ P[0] ≤ 15	1011010
0	3	16 ≤ P[0] ≤ 19	1011000
0	4	20 ≤ P[0] ≤ 23	1001000
0	5	24 ≤ P[0] ≤ 31	1001001
0	6	32 ≤ P[0] ≤ 47	0001001
0	7	48 ≤ P[0] ≤ 63	0001100
1	1	0 ≤ P[1] ≤ 7	0001010
1	2	8 ≤ P[1] ≤ 15	0000010
1	3	16 ≤ P[1] ≤ 23	0000110
1	4	24 ≤ P[1] ≤ 31	1000110
1	5	32 ≤ P[1] ≤ 39	0110011
1	6	40 ≤ P[1] ≤ 47	0110001
1	7	48 ≤ P[1] ≤ 63	0100001

Assume an arriving packet with P[0]=25 and P[1]=39. The one-dimensional searches will figure out that 25 belongs to interval 5 in the first dimension and 39 belongs to interval 5 in the second dimension. The algorithm now combines the two concerning bitmaps (1001001 AND 0110011)=000001 and the resulting bitmap indicates that rule 6 is the only matching result. If we assume a packet with P[0]=10 and P[1]=30 the combination of the bitmaps (indices 2 and 4) would yield (1011010 AND 1000110)=1000010 and rule zero (the leftmost bit) would have to be chosen as the highest priority match.

Similar as in bitmap-intersection, also crossproducting [60] constructs the d-dimensional classification out of d one-dimensional range lookups. In contrast to bitmap-intersection Srinivasan et.al. precompute a crossproduct table that contains the best matching rule for all possible combinations (i.e. crossproducts) of the ranges in each dimension. The worst case storage complexity of the crossproduct table $O(N^d)$ [55] is even worse than that of bitmap intersection, which makes the algorithm practical only for relatively small rule bases. In order to address the principle storage space problem, the authors of [60] propose an on-demand calculation of the table that behaves like a cache. For every search operation, the matching ranges are identified and the crossproduct of the indices is formed. If this is already contained in the (incomplete) crossproduct table, the classification result may be obtained in a single lookup. Otherwise the rule base has to be inspected (by linear search!) and the corresponding entry will be appended to the crossproduct table cache. By this optimization the authors promise to achieve a good average-case classification performance with a viable storage requirement. In addition, the

authors of both bitmap-intersection and crossproducting assume smaller rule bases with a few thousand rules at most.

2.3.2.5. Distributed Crossproducting of Field Labels (DCFL)

In 2004, Taylor introduced DCFL [64], [65], which is essentially an optimization of the crossproducting scheme with a strong focus for hardware implementation. The classification process is split into several steps, which may be implemented in a hardware pipeline as shown in Figure 16.

At first, relevant packet fields are extracted from the packet and the one-dimensional field searches are initiated, which compare the field value of the current packet against the superset of all possible values present in the given classification rule base. The result of these searches is a set of labels, representing one or more matches per header field.

Second, the labels are fed into a so-called aggregation network. Here, at least two label sets from the first search stage are combined (i.e. the crossproduct of the label sets is calculated) and the outcome is compared against the set of valid crossproducts that exist in the given rule base. In order to keep the complexity of the crossproduct evaluation and set membership query low, combining only two one-dimensional search results at a time appears to be a reasonable choice. The remaining set of crossproducts is assigned new labels that can be fed into subsequent aggregation stages in a recursive fashion.

At the final step of the aggregation network, a label for one matching rule or several labels in case of overlapping rules will be found. In the latter case, a priority resolution scheme is used to find the highest-priority matching rule from the given set of labels.

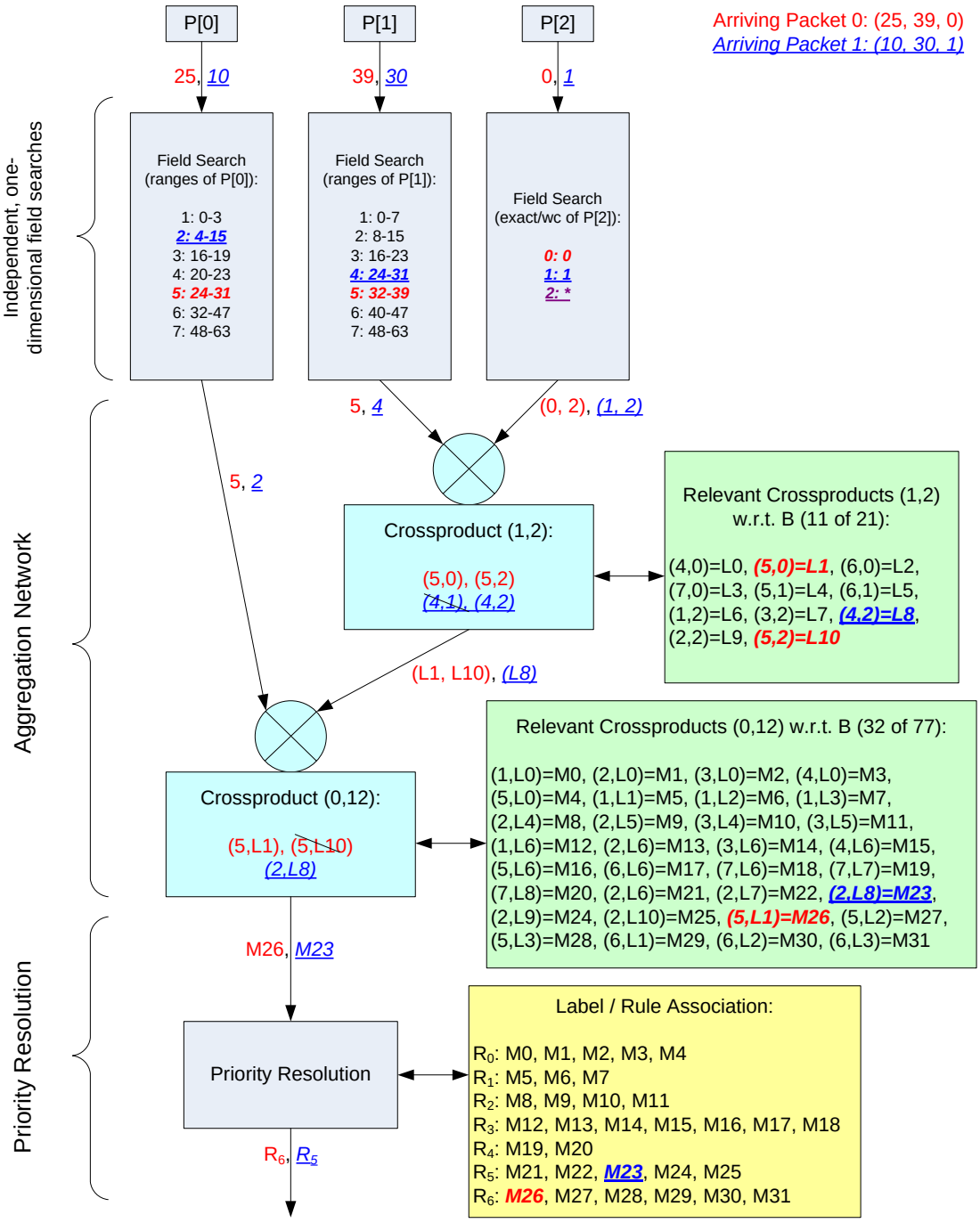


Figure 16: DCFL Classification with Three-dimensional Rule Base from Table 4b

A distinct feature of DCFL is its update procedure, which is accomplished by sending special "update packets" through the stages of the aggregation network that may update the local crossproduct tables and label assignments. This update feature allows for consistent rule base changes while maintaining a high search performance.

In contrast to the previously presented crossproducting scheme (see 2.3.2.4), the storage space issue is addressed by aggregating the results not in a single step, but distributing the decision over several pipeline stages. As in each stage some

crossproducts, which would not lead to matching results in the final rule base, are filtered out; the total storage complexity can be reduced. This effect can be observed in Figure 16, where for packet 1 the crossproduct (5,L10) is filtered out in the second stage; the same holds for crossproduct (4,1) for packet 1 in the first aggregation stage. As the chosen example rule base has wildcards - and therefore overlapping specifications for many rules - only in its third dimension, the number of matches from the individual field searches is limited to one or two. The number of matches per field may however increase to an order of up to 5 for realistic rule bases with overlapping range definitions and more wildcard specifications [64]. In consequence, the size of the crossproduct at each aggregation note may multiply to a range of 5×5=25 crossproducts that have to be searched against the list of relevant crossproducts. While the size of the crossproduct could therefore grow in subsequent steps (e.g. 5×25=125), the filtering of the obtained crossproducts against the set of relevant crossproducts keeps the number of labels propagating to downstream aggregation nodes limited.

Taylor states himself that the choice of the correct aggregation sequence, i.e. which header field combinations should be combined first, has a significant impact on the size of the intermediate crossproducts; and in turn onto the performance of the set membership query. For static rule bases, an optimized aggregation sequence may be found in advance, but the performance may deteriorate by subsequent incremental updates to the rule base. He proposes using a dynamically reconfigurable interconnection network between the individual pipeline stages, but has not further elaborated this concept.

2.3.2.6. Multi-Field Classification using Binary Decision Diagrams (BDDs)

In their 2003 paper, Prakash et.al. [66] regard both the IP lookup function and the more general packet classification task as a logical synthesis problem. Lookup and classification problems are formulated as a Boolean function that takes the bits of the packet header as inputs and computes the index of the desired action or the next-hop destination as output.

In an initial attempt, they tried to feed the Boolean equations obtained from disassembling a backbone routing table to the Xilinx synthesis tools for implementation. This approach failed, but they were later able to route a manually generated BDD structure onto the FPGA fabric. However, this solution disappointed performance-wise with a combinatorial path delay of 85 ns.

Finally, a solution for the IP lookup problem is presented, in which the BDD of the routing table is calculated offline and its nodes are encoded in an array of SRAM memories that are used for searching through the BDD with the address bits from the incoming packet. Some optimizations are presented to save on resources needed for implementing the IP lookup.

The authors prove that a straightforward extension of the BDD-based routing scheme is not feasible for full five-tuple classification, as the memory requirements for the BDD scales exponentially. This is also a known problem for existing trie-based techniques. In order to obtain a viable solution, they present a partitioning technique for the classification rule base in order to obtain a set of small BDDs that may be evaluated in parallel. The resulting individual actions are then fed into a priority resolution stage.

An important advantage of using BDDs versus binary decision trees (see 2.3.1.2) is their node sharing property. Isomorphic sub-trees of a binary decision tree are merged into a single instance, which allows saving memory proportional to the amount of common structure in the rule base. In the outlook section of the paper [66], the authors mention the possibility to move the approach from reduced ordered BDDs (ROBDD) to free BDDs, where the variable ordering between different branches in the diagram is not necessarily uniform. Although free BDDs offer a theoretical benefit in storage space, they state that a synthesis methodology for such free BDDs is not yet known.

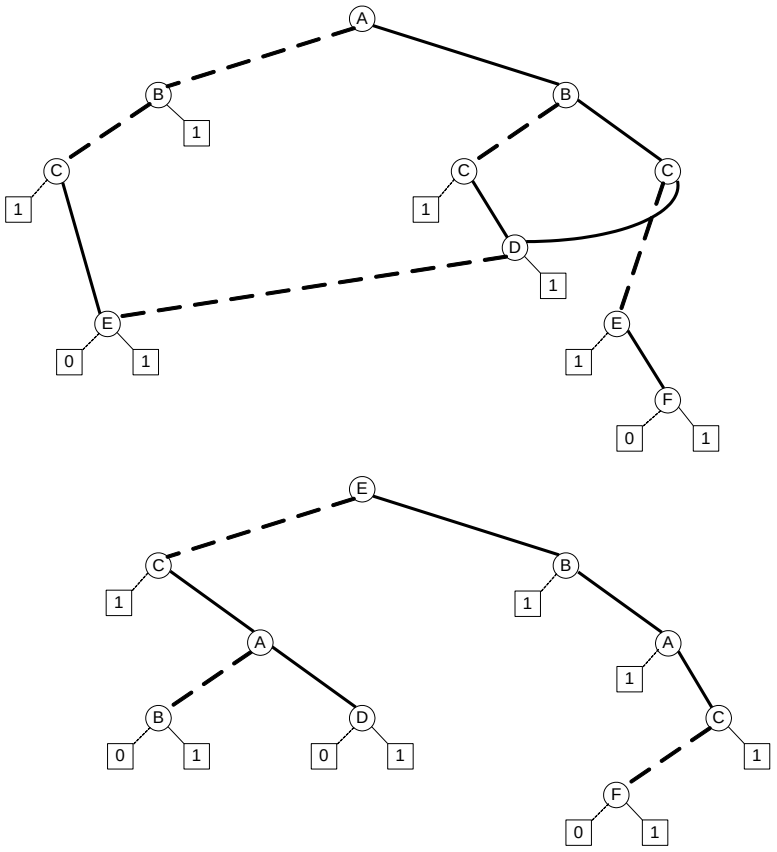


Figure 17: Reduced Ordered BDD (ROBDD, top) and Free BDD (bottom) for Boolean Function $f = AD\overline{E} + \overline{B}E + EF + \overline{A}B + \overline{C}\overline{E} + CE$

Figure 17 shows both a ROBDD for the variable ordering A, B, C, D, E, F and a free BDD for an example Boolean function $f = AD\overline{E} + \overline{B}E + EF + \overline{A}B + \overline{C}\overline{E} + CE$. While a

binary decision tree for the given function would result in 63 internal nodes, 64 leaves and a uniform depth of 6, the ROBDD represents the same function with only 10 nodes and two leaves (0 and 1; shown several times in Figure 17 for clarity) and an average depth of 4.14. A free BDD, which has been constructed using the heuristic described in chapter 4.2.3, has 9 nodes and an average depth of 3.7.

The gained insights on the theory of BDDs and their properties have partially inspired the development of the HDGA classification algorithm implemented in the FlexPath Path Dispatcher (see 4.2).

2.3.2.7. Traffic-Aware Decision Tree Classifiers

In 2005, Cohen and Lund [67] presented a design method of a traffic-aware decision tree classifier for software implementation of a standard ACL deployment in ISP edge routers. The method exploits structure found in real-world firewall applications to obtain a decision tree with good average-case search times and reasonable memory consumption.

Regarding commercial deployments, the authors state that TCAMs (see 2.3.1.5) are the most widely used form of classification engines, but they hint to insufficient support of range matches and high power consumption as incentives to look for alternative solutions. Tree-based classifiers are identified as most effective candidates with respect to the memory versus search time tradeoff.

One specialty of the proposed tree, that is in addition to the classical decision tree scheme as discussed in 2.3.1.2, is the "common branching" technique. When splitting the rule base at an internal node of the tree rules, which would replicate to both children, are handled separately in a list structure. This technique addresses the memory blowup problem otherwise associated with tree-based approaches, but on the other hand increases the search time needed within the tree node. In addition, a tree node is not further split, if the remaining set of rules could also be resolved by linear search. This further reduces the storage requirements for the tree structure.

Another optimization is based on the observation that realistic firewalls contain a few "allow" rules and many "deny" rules. However, most of the traffic will be admitted to the network: either it has been filtered before in other parts of the network already, or it belongs to "legal traffic". Thus, it seems attractive to construct the tree in a manner such that "allow" packets are evaluated before the "deny" packets, i.e. they can be evaluated in deeper branches of the tree. The optimized tree would then yield a good average-case performance with realistic traffic.

2.3.2.8. Modular Packet Classification with Parallel Search Trees and Linear Search

The final classification technique to be mentioned in this section is a modular approach proposed by Woo [68] in 2000. The classification scheme fits to an arbitrary-dimension problem, although only two- and five-dimensional classification have been explicitly addressed in the paper. The basic idea is to reduce the number of eligible rules that may match a given packet in three basic steps:

- In the first step some bits from the incoming packet header are used to determine one of several binary or 2^m-ary search trees for the second step.
- As the selected search tree is traversed, the number of rules still fitting the packet header is further reduced. The partitioning continues until a specified limit of rules (the paper proposes values between 8 and 128 rules) is reached to avoid the memory size explosion associated with exact rule matching in tree structures.
- The final step of the algorithm searches through the remaining set of rules either by linear search, binary search or using a TCAM.

For the scope of the paper, a software implementation is presented, that has been especially optimized to work well with page-oriented memory hierarchies and achieves a maximum classification throughput of 100k packets per second. The author addresses the issue of covering and overlapping filter definitions and shows that a separation of similar filters using linear search may be more efficient than trying to distinguish them in a search tree structure.

2.3.3. Packet Classification and Logic Minimization

An interesting aspect about both single-field and multi-field packet classification is addressed by Lysecky et.al. in [69]. In their publication, they look at the task of implementing a firewall ACL in a TCAM device. The problem is that the cost for the implementation is growing with the total amount of memory consumed by the rule base. They state that logic minimization has been used before on IP routing tables in order to reduce the number of entries by exploiting overlapping specifications in the original routing table and replacing them with a merged entry that contains additional wildcard entries. The authors show that they can apply this technique also to the field of firewall filter rule bases and present a logic minimization tool that is specially tailored for embedded deployment. While reaching a similar optimization performance as the state-of-the-art Espresso technique ([70], [71]), they can claim a factor of 20 improvement in processor runtime on an embedded ARM7 CPU. The investigated rule bases have been reduced by between 17% and 40% using their logic minimization algorithm.

In contrast to the algorithms presented in sections 2.3.1 and 2.3.2, the logic minimization is not used to calculate the best matching rule of the rule base given the header fields of an arriving packet, but it performs a pre-processing of the rule base. The size (and therefore the search complexity) of the rule base may be reduced, in order to keep the required storage space for the rule base smaller and aid the classification algorithms in finding the classification result in a faster way.

2.3.4. Conclusions

Packet classification is a crucial part of packet processing that exhibits a high degree of complexity and is very performance critical. Therefore, packet classification has attracted lots of attention in the academic environment for decades. In the early stages, many researchers focused on single-field classification problems in order to improve the processing performance for simple Internet routers. Later, the focus changed more towards multi-field packet classification with the increasing importance of more advanced networking applications that introduce QoS and security features into the network infrastructure and require flow-specific treatment of the network packets.

The state-of-the-art multi-field classification techniques are all optimized for evaluating firewall rule bases, which typically involve specifications of the Internet five-tuple along with the associated actions. This flow-based granularity of the rules leads to a very regular structure of the rule bases and a size between several hundred and a few ten thousand entries. Range and prefix matches, which are commonly found in rule specifications, can be most effectively addressed with tree- or trie-based search structures. However, these techniques suffer from an exponential memory consumption, if exact matches have to be determined. Therefore, some more advanced tree classification algorithms try to combine the tree traversal with linear search, binary search or TCAM lookups for selecting the matching rule from a smaller set of candidate rules determined by the decision tree. Another group of classification algorithms addresses the problem with a multi-stage approach that constructs the final decision out of the results of individual field searches (e.g. Crossproducting, Grid-of-Tries and DCFL) or by cascading several searches (e.g. RFC, HiCuts, HyperCuts). Except for DCFL and simple tree-based schemes, incremental updates of the rule base are not easily supported, which means that a change in the rule base requires a complete recomputation of the search data structure.

Another interesting aspect found in this prior art survey is the fact that the complexity of the classification problem may be reduced in a rule base pre-processing step by means of logic minimization. The logic minimization allows compressing the rule base by dropping redundant or contradicting rule base

specifications, which are typically introduced by the network operators during manual firewall specification.

As I will discuss later in chapter 4.1, the multi-field classification problem associated with determining the correct processing path for the arriving packets differs in several parameters from the firewall-centric investigations of previously published algorithms. The processing path in a FlexPath NP is mainly determined by the application characteristics of the underlying networking application (see chapter 2.2), and not by the individual flow ID associated with the arriving packet. Therefore, the classification can be limited to fewer rules, but the rule specifications have to be extended to contain more fields from the packet header than just the traditional Internet five-tuple.

2.4. Multi-Processor Load Balancing

The prior art survey on current network processor architectures in section 2.1 has shown that all relevant NP architectures are multi-core devices with a multitude of processing elements. Although there exist some strictly pipelined NPs, most devices today feature a parallel processor cluster and adhere to the run-to-completion programming model. In these architectures, arriving packets will be assigned to a specific processor and the networking application can be written as a simple, sequential program from the point of view of the programmer. In order to exploit the parallelism of the processing resources, a load balancing strategy is needed, which decides about an appropriate CPU, to which an arriving packet will be assigned. If the load can not be (almost) equally distributed over the available pool of processing resources, some processors may run idle, while others become overloaded and some of the arriving packets are lost. In such a case, the available resources of the device would not be utilized, resulting in an inferior system performance.

This load balancing problem also extends to system setups, where the processing is achieved by a sequence of processors (e.g. parallel pipeline processing, where one out of several pipelines has to be selected) or a combination of parallel processors and shared hardware accelerators.

2.4.1. Hashing-based Load Balancing Schemes

At INFOCOM 2000, Cao et.al. [51] presented a comprehensive performance comparison of direct hashing-based Internet load balancing techniques for different hash functions and for realistic Internet traffic traces. In addition, the direct hashing-based methods were compared to table-based schemes with threshold mapping and index-based load assignment (see Figure 18) that allow run-time adaptation of the flow-bundle to processor mapping. However, the authors did not elaborate potential adaptation strategies in detail but restrict their observations to a rather abstract description and a few simulation results.

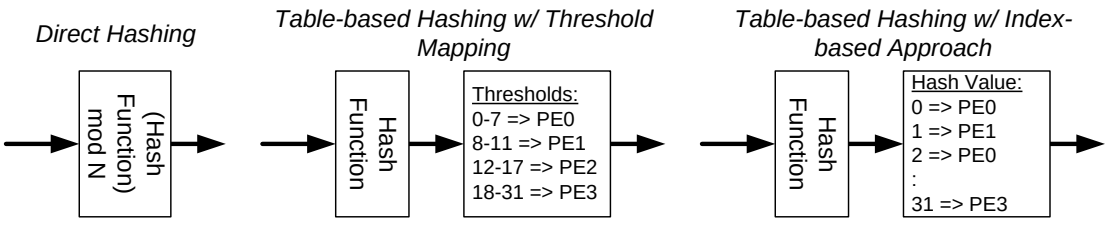


Figure 18: Classification of Hashing-based Load Balancing Schemes

The CRC-16 (16 bit cyclic redundancy check) of the Internet five-tuple is identified to provide the best load balancing results applied in a static direct hashing-based assignment. The table-based adaptive algorithms are based on an XOR of the IP source and destination addresses and achieve a similar performance as the CRC-based direct hashing approach.

2.4.2. Hash-based Load Balancing with Overload Spraying

In 2002, Dittmann and Herkersdorf proposed a hashing-based load balancing system for parallel processing element (PE) network processors [73]. For each incoming packet a hash value is computed out of the Internet five-tuple header fields. A load balancing table maintains a list of hash values, their associated PE and a timestamp, when the last packet with the given five-tuple hash had entered the system, i.e. it can be categorized in the table-based hashing techniques with an index-based approach as shown in Figure 18. If the incoming packet's flow has not expired, the packet is forwarded towards the PE queue specified in the load balancing table. If the timestamp is older than a predefined timeout value, the entry in the load balancing table is updated to direct the packet towards the least loaded PE queue (see also Figure 19).

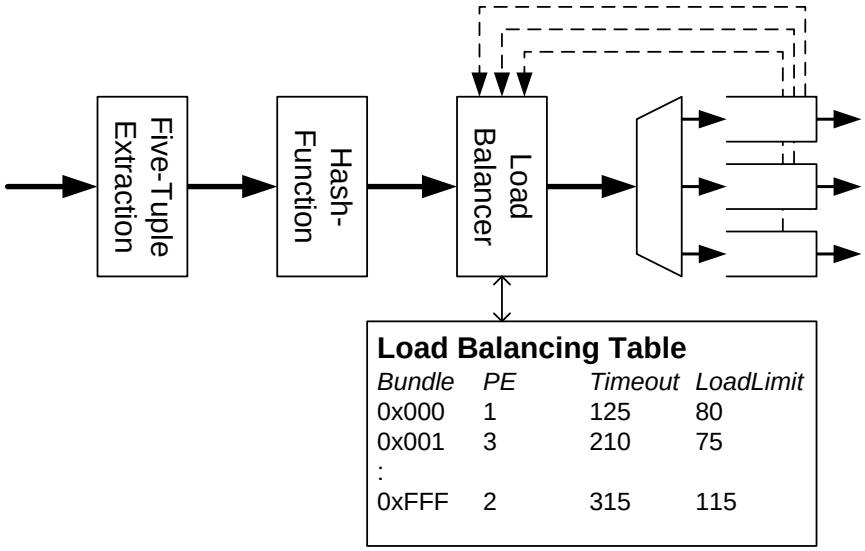


Figure 19: Hash-based Load Balancing with Overload Spraying

Two exceptions from this basic scheme are presented: An existing flow entry may be changed and re-mapped to another PE before the timeout has occurred, if the initial PE queue is already overloaded. In this fashion, an unnecessary packet loss in the NP is avoided. Second, if the set of flows mapped to a single hash value (i.e. hash collisions!) would exceed the processing capabilities of a single PE, these packets may be distributed over several PEs, which is called packet spraying.

It is important to realize that packets may be reordered, when a re-mapping takes place during a flow bundle's lifetime, and - of course - when packets of an excessive flow bundle get sprayed. Packet reordering has been identified to cause problems with the congestion avoidance mechanism of TCP, and should be avoided as far as possible [78].

2.4.3. Adaptive HRW Hashing (AHH)

Kencl [74] refines the basic idea of hashing-based load distribution in network processors in his dissertation of 2003. He introduces an adaptive control loop in combination with a robust highest random weight (HRW) hashing to the load assignment process, called adaptive HRW hashing (AHH).

The highest random weight algorithm finds the target PE index j out of m PEs for a given packet with the header field vector v using PE-specific weights x_k, $k \in \{0, \ldots, m-1\}$ in the following way:

$$x_j \times h(v, j) = \max_{k \in \{0,\ldots,m-1\}} (x_k \times h(v, k)) \quad (2\text{-}11)$$

This means that the hash function has to be calculated for each combination of the extracted packet header fields (typically the Internet five-tuple) and each possible processor index k. The hash values, which are assumed to yield random values, are weighted with factors x_k that can be adjusted during system runtime in order to reflect the relative utilization of the associated processor. The packet is assigned to the PE, for which the product of the weighting factor and the hash function computed over the packet ID and processor ID is maximized.

An adaptive control loop assures that the weight vector $x=[x_0 \ldots x_{m-1}]$ used in the HRW hashing is modified during runtime, such that the assignment of flow bundles to the PEs is more evenly balanced for the biased hash bundles found in real Internet traffic. Packet reordering may occur, when the weight adaptation triggers a re-balancing of flow bundles from one PE to another.

An important characteristic of the AHH load balancing scheme is the minimum disruption property, which means that when gradual changes are necessary in rebalancing the load, only a few distinct packet flows are re-mapped and not the entire hash space is reassigned with new values. This could be well observed when comparing AHH to an interval-based adaptive hashing scheme (see Figure 18 in 2.4.1), where up to 50% of all flows could be re-mapped if a single PE fails and the load has to be re-distributed over the remaining PEs in the processor complex. With the minimal disruption property, only the flows assigned to the failing PE are shifted.

2.4.4. Adaptive Burst Shifting (ABS)

The load balancing approach of Shi et.al. [75] from 2005 is based on the observation that Internet traffic usually consists of many flows with relatively low activity and only a few flows with high activity, referred to as aggressive flows. Such a classification has also been described before by Brownlee and Claffy [79] in 2002, where they classify Internet traffic flows into dragonflies and tortoises or elephants and mice. Dragonfly and tortoise traffic refers to the lifetime of traffic flows, while the elephant and mouse analogy refers to the flow size or intensity, i.e. the amount of

data exchanged between two communication partners. Shi et.al. introduce an adaptive burst shifter (ABS) that complements known hash-based load assignment schemes. It remaps only aggressive flows from one PE to another if the hashing-based assignment leads to a temporarily unbalanced load situation. The non-aggressive flows are still mapped solely by the result of the implemented hashing scheme (see Figure 20).

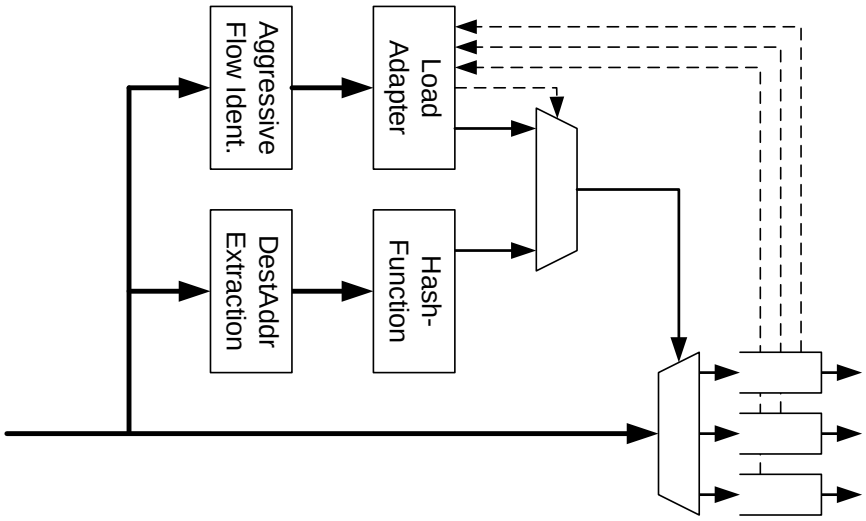


Figure 20: Adaptive Burst Shifting (ABS)

Since the shift of a few aggressive flows already has an appreciable effect on PE load, the number of hash flow shifts can be reduced and thus also packet reordering within the processor cluster of the NP. ABS uses a two stage approach with a Flow Classifier to identify the aggressive flows in the traffic, and a Load Adapter that remaps the aggressive flows to the least-loaded PE when needed.

2.4.5. Hashing Adapted by Burst Shifting (HABS)

In 2006, Kencl and Shi proposed to combine their previously implemented load balancing schemes to achieve even better performance [77]. In the combined method called HABS the result of an AHH load assignment is fed into an active Flow Classifier and Load Adapter structure of the ABS scheme (see Figure 21). In contrast to the original ABS scheme presented in [75], all flows may be considered by the Burst Shifter now [76] and not just the aggressive ones. If the system is in a well balanced state, the assignment from the hashing-based stage is used. Whenever an imbalance is detected, the ABS part comes into play and moves loads directly away from the heaviest-loaded PE to the least-loaded PE, even before the adaptation routine inherent in the AHH load balancer might react. In addition, the algorithm insures that flows may only be re-mapped at the beginning of a burst, i.e. when no other packet from the same flow is already in the system.

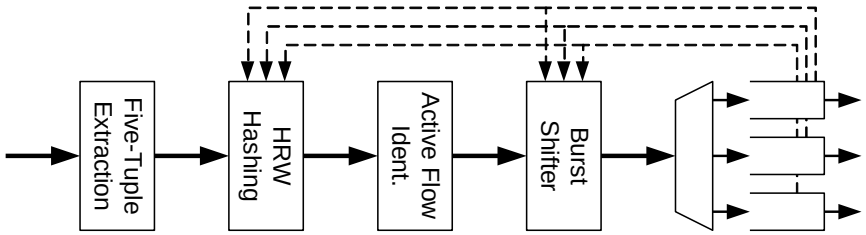


Figure 21: Hashing Adapted by Burst Shifting (HABS)

The combined HABS scheme delivers the best performance of the current schemes proposed in the prior art with respect to the number of active flow re-mappings and packet reordering rates. However, the burst shifting algorithm requires maintenance of a lot of state information, which makes the algorithm somewhat complex for implementation.

2.4.6. Conclusions

While load balancing schemes have been discussed in the scientific and high-performance computing area for a long time, they gained attention in the network processing field with the introduction of network processors that are implemented as multi-processor system-on-chip architectures. In contrast to other fields, load balancing for networking applications comes with domain-specific constraints, which are all reflected in the previously presented load balancing schemes.

The most prominent requirement for network processor load balancing is that packets, which belong to the same logical connection, should be forwarded in the same sequence as they arrived. This means, that packet reordering caused by assigning packets of the same flow to different PEs should be avoided as far as possible:

- Static hashing-based load balancing insures this sequence by assigning the packets based on a hashing of the Internet five-tuple, which serves as a unique flow ID. The drawback of such static schemes is that due to an uneven distribution of the hash bundles in the total hash space, some processors are assigned more traffic than others. As a consequence, the available processing resources of the NP are not utilized to their full potential.

- Adaptive load balancing schemes address this problem by allowing a reassignment of flows during the system runtime. Obvious benefits are a more evenly balanced load distribution, at the price of risking a few out-of-order packets during the load adaptations.

Another important property of stateless network processing applications (which constitute the majority of network traffic) is the independence of the individual packets. This independence allows deploying the packet spraying technique that permits a very uniform distribution of the arriving traffic over the available processor

resources. However, packet spraying is only proposed as a remedy for overload situations due to the before mentioned packet reordering issue.

3. FlexPath NP Architecture

In the following sections, I will introduce the FlexPath NP architecture that provides an NP with different on-chip processing paths. These paths comprise processing with varying degrees of hardware offload - optimized for different networking applications. After deriving the architectural concept based on an analysis of networking applications and state-of-the-art NPs, the benefits are shown by means of analytical analysis and SystemC simulations of a basic FlexPath NP model.

The concept for the packet classification algorithm that is accomplished in the Path Dispatcher is a crucial element of a FlexPath NP. Because of its importance for the proposed NP architecture, I have devoted a separate chapter for its detailed discussion (chapter 4).

3.1. Motivation and Problem Formulation

Figure 22 shows the typical functional unit traversal of state-of-the-art NP architectures as presented in chapter 2.1.

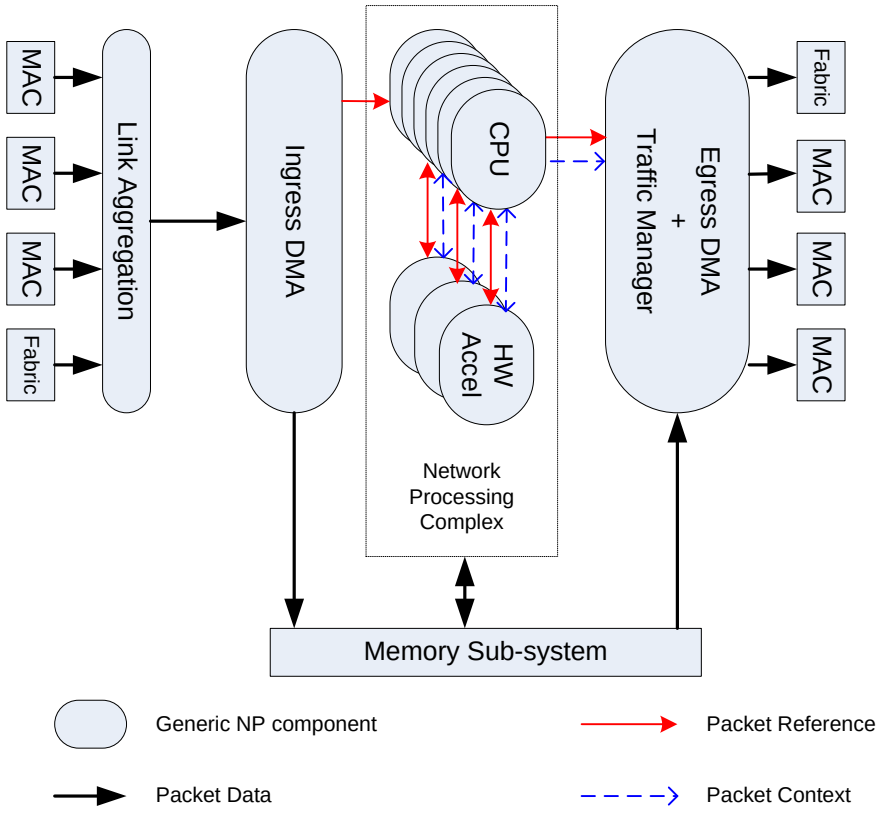


Figure 22: Functional Unit Traversal in a Generic Network Processor

Depending on the individual design, one or several MAC interfaces provide a connection to either the physical link attachment(s) or the switching fabric (see also Figure 2 in chapter 1). The traffic may be aggregated before it will need to be stored (typically this means implementation of some kind of DMA), in order to make the

packets accessible to the packet processing units. Then, the packets are transferred to the central processing complex, which usually consists of a multitude of programmable elements (PE) - also referred to as CPUs in the following - and networking-specific hardware accelerators. As discussed in chapter 2.1.3, the predominant arrangements for the CPUs are parallel processing clusters and processor pipelines (Figure 4). Hardware accelerators are accessed under control of the programmable processors. In both processor cluster and processor pipeline architectures, every packet will traverse a CPU at least once and the exact traversal pattern of the various functional units (e.g. frequency and type of hardware accelerator calls, one or several software threads) is determined by software. After the correct destination interface and scheduling priority has been selected and any possible additional protocol processing is finished, the packets are forwarded to the Traffic Manager. Here, the packets are queued to resolve output port contention. Depending on operational requirements, queuing may be achieved on a coarse traffic class granularity with several prioritized queues per port or on a fine grained flow basis. In addition to queuing, the Traffic Manager may also perform traffic shaping before releasing the packets onto the outgoing links. Virtually all of the commercial network processors presented in chapter 2.1 feature such a Traffic Manager - either as an integrated building block in the NP chip, or as a separate chip - which is tightly coupled to the actual processor chip.

In order to illustrate the operational constraints for an NP system, consider the following case study for a 10 Gbit/s full duplex device, which is quite representative for the current mid-range equipment. The 10 Gbit/s may be achieved for example by connecting the NP to either four OC-48/STM-16 Sonet/SDH, ten 1 Gbit/s Ethernet or a single 10 Gbit/s Ethernet link. Due to the different protocol overheads on the physical layers, the following packet rates and inter-arrival times can be derived assuming that on average the same 10 Gbit/s received from the lines are transferred over the backplane (i.e. switching fabric) interface.

As we can see from the figures in Table 6, the packet rate is in the tens of Mpps range for the worst case of shortest size packets. The packet rate is directly coupled to the event or interrupt rate seen by the central processor cluster. Consequently, the interarrival time of two consecutive packets seen by the processor complex may be as low as 20 ns. The Packet-over-Sonet/SDH (PoS) protocol is more efficient for shortest size IP payloads, thus the requirements for an NP targeting PoS are higher than for comparable Ethernet deployments.

Table 6: Processing Constraints for 4x STM-16 Packet-over-Sonet/SDH or 10 Gbit/s Ethernet Links[2]

Transmission Standard	*Packet Size*	*IP Data Rate (duplex)*	*Packet Rate*	*Interarrival time*
Sonet/SDH, PoS	40 byte IP	15.648 Gbps	48.9 Mpps	20.5 ns
Sonet/SDH, PoS	1500 byte IP	18.904 Gbps	1.576 Mpps	635 ns
Ethernet	40 byte IP	9.524 Gbps	32.06 Mpps	31.2 ns
Ethernet	1500 byte IP	19.506 Gbps	1.626 Mpps	615 ns

According to benchmark results ([85], [86]), simple IP packet forwarding consumes around 350 instructions per packet, while more complex deep packet processing applications (e.g. intrusion detection, virus scanning, software-based cryptography) may require up to 3,000 instructions normalized to the shortest packet size. Applying these figures to the packet rates in Table 6, up to 17,115 MIPS processing performance are required for the simple forwarding task and up to 146,700 MIPS for the most complex deep packet processing applications, if this kind of processing would have to be applied onto the entire traffic from the PoS links. Assuming a state-of-the-art embedded processor core with 1.5 GHz clock frequency and an optimistic CPI of 1.0, this processing power would translate to 12 cores for forwarding and 98 cores for the deep packet processing task.

Of course, by making use of application-specific hardware accelerators, the latter figure can be reduced significantly, as the computational density for dedicated hardware modules is between two and four orders of magnitude better than that for programmable CPUs [87]. The extensive use of hardware acceleration for deep packet processing can also be observed in virtually all commercial NP architectures (see section 2.1.1). Still, the processor cluster has to deal with the high event rates caused by the incoming packet stream, and the event rate would already scale by a factor of two, when a packet is once handed over between the CPU and a hardware accelerator.

The above described case study reveals the following challenges for current NP architectures:

[2] These calculations assume transmission of 40-byte and 1500-byte IP datagrams over PoS and Ethernet media. Protocol overhead calculations for PoS include SDH overhead, 9 bytes PPP overhead and 1/128 times payload size for byte stuffing (see also [80]). For Ethernet, padding for minimum frame size of 64 bytes, preamble and inter-frame gaps are included.

– Even simple networking applications (e.g. IP forwarding) require many parallel processor resources in order to cope with the high packet rates observed on links with bursts of shortest size packets.

– Hardware accelerators help to reduce the number of required processors for compute-intensive applications (e.g. IPsec, virus scanning) because of their higher computational density. However, software-controlled accelerator calls increase the event rate for the controlling CPU by at least a factor of two.

– The processing path of the packets through the NP system is determined by software. In certain situations, it can happen that the processor receives a packet and relies on the results of hardware accelerators (e.g. classification or look-up using a network search engine, decryption of a protected packet, etc.) before the software can continue with meaningful processing. The overhead associated with inspecting the packet and immediately dispatching it to another unit in the NP deteriorates the overall system performance.

3.2. FlexPath NP Concept

Based on the challenges described in the previous section, ways to improve the performance of network processors on an architectural level are sought.

The basic idea behind FlexPath NP is to improve the performance of the system by most effectively utilizing the available processing resources. Current NP architectures have already found efficient software and hardware means for the data path processing of various applications. But the control path, which is currently still implemented in software, might be improved with the help of specialized hardware units in several positions of the architecture that help direct packets to the most suitable processing element (PE). The FlexPath NP architecture [7], proposed in 2005, achieves performance benefits in contrast to conventional NP architectures through the following measures:

- Introduction of hardware-offload units near the ingress and egress side interfaces of the NP, which are able to relieve the central processor complex from simple, recurring tasks such that the intelligence inherent in the programmable resources is not "wasted" for "routine" tasks.

- The hardware-offload units should be able to handle basic forwarding traffic, such that the central processor complex can be completely bypassed for those kinds of packets. This feature is in the following referred to as "AutoRoute" path in contrast to a CPU path.

- The FlexPath NP provides a classification capability near the ingress interfaces in order to differentiate between packets of various networking applications. The classified packets are then directed on a processing path (i.e. traversal sequence of processing units), that is especially optimized for the application. The simplest example would be the choice between a path through the central processor cluster and the AutoRoute path.

- The architecture should feature a packet distribution system, with which hardware accelerators may be accessed directly, i.e. without involving a CPU. In case of arriving IPsec packets, the event rate for the CPUs can be decreased, if the decryption core may be directly accessed by the ingress side hardware. After the hardware accelerator has finished its work on the packet, a mechanism is needed to efficiently pass it on to a CPU to finish the packet processing functions. Thus, packet paths through the NP system may comprise several chained entities (multi-hop paths).

- In addition, the classification function should be run-time reconfigurable, such that the system can be adapted for newly developing applications in the field and also to short-time changes in the traffic mix during system runtime. The classification

may also be exploited to support advanced QoS features and load balancing algorithms in the NP that further improve the performance of the system.

Figure 23 shows the traversal of the different functional units in a FlexPath NP with the before mentioned extensions marked in orange and yellow. While the entire range of hardware extensions will be presented in detail in the following paragraphs, I will later focus only on the functional modules in the ingress part of the NP, namely Pre-Processor, Path Dispatcher and load balancing techniques (orange blocks). The remaining functions (yellow blocks) are fully elaborated in Michael Meitinger's dissertation [107]. In our final demonstrator implementation of a FlexPath NP (see chapter 6), we have included the SmartMem buffer manager developed by Daniel Llorente [108] as DMA engine (green blocks). The following paragraphs introduce the most important characteristics of the entire FlexPath architecture.

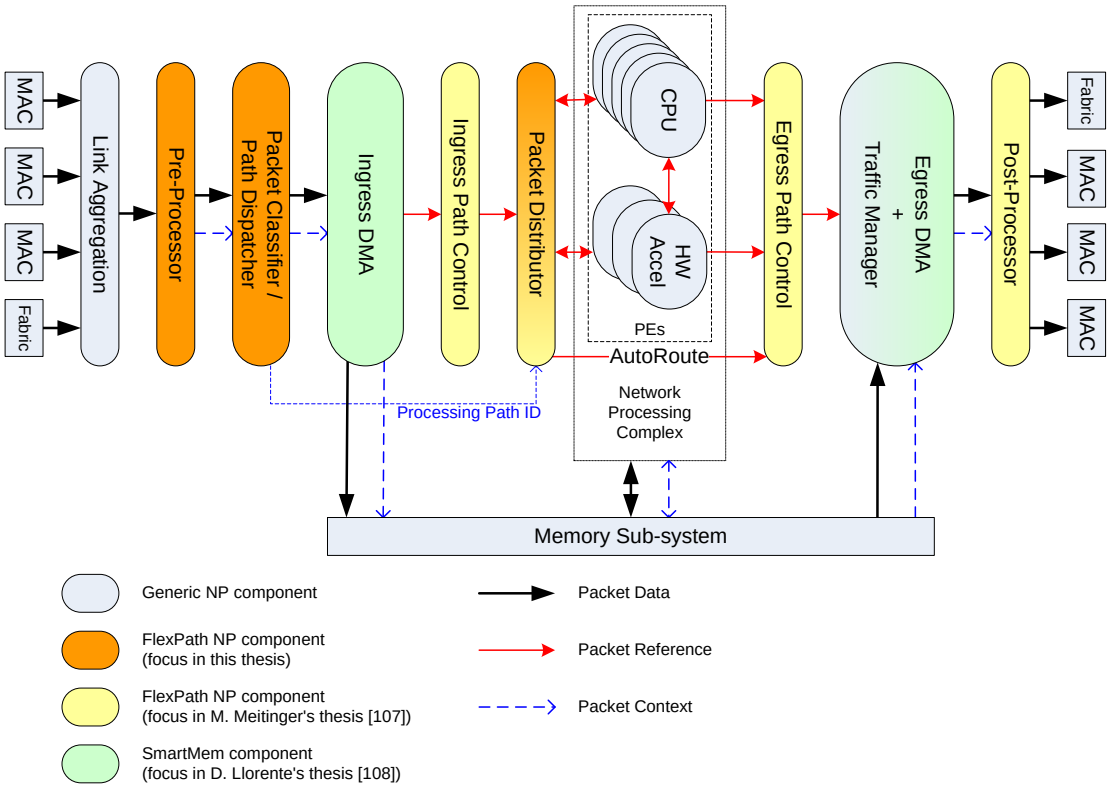


Figure 23: Functional Unit Traversal in a FlexPath NP

Just like in the generic NP case (Figure 22), it may be assumed that the aggregate of the line interfaces and the switching fabric interface is combined onto a single high-capacity data path.

The **Pre-Processor** is the first FlexPath functional unit after the initial traffic aggregation (see Figure 23). As the decision about the further processing path in the system is dependent on the networking application required for each individual packet, the packets have to be analyzed first. A first step comprises parsing the header fields and extracting information like the Internet five-tuple (i.e. IP source and

destination addresses, layer four protocol number and layer four port numbers). In addition further differentiating fields like the IP DiffServ codepoint (DSCP), which contains information about the forwarding priority of the packet, IPsec SPI numbers or control plane protocols will be recognized. The parsed header fields may also be used directly to initiate lookups (e.g. IP routing lookup) in an attached hardware lookup engine (e.g. TCAM, NSE). The collected information about each packet is assembled into an additional data structure called packet context (dashed blue arrows in Figure 23) that is accessible by all further NP-internal processing modules. In addition, it is also possible to perform basic packet integrity checks at this early stage. If corrupt frames can be detected early, they might either be handed over directly to the NP control point for further error handling (e.g. generating ICMP messages) or could be silently discarded without requiring CPU intervention or wasting memory bandwidth during the ingress DMA. Furthermore, the CPUs in the processing cluster don't have to perform these checks in software anymore, which reduces the processing performance requirements per packet.

The **Path Dispatcher** uses information in the packet context for the classification of the arriving packets into the different applications and traffic classes, which are currently supported by the NP system. A software process that is running on the control plane CPU of the NP keeps track of the current system state and computes an appropriate classification rule base, which is then applied to the incoming traffic. Based on the classification result, the packets are dispatched to the pre-configured processing path in the NP system, by adding the type and sequence of further processing modules to the packet context. Apart from more or less static rules, which direct certain traffic types like control packets to the control plane CPU, discard corrupt frames, etc., dynamic rule table updates can be used to perform load balancing within the processing complex or dynamically enable AutoRoute for certain established and well understood traffic types. A default rule should always be configured, which directs "unknown" packets to the data plane CPU cluster for further determination of an appropriate action.

After the choice of the further PE traversal in the system, packets and packet contexts have to be transferred into an appropriate memory for processing. In our FlexPath demonstrator system we have made use of the SmartMem DMA architecture ([88], [89], [108]) that supports different storage locations depending on the further destination of the packet, i.e. for example that CPU-bound packets may be stored in a local SRAM near the processors, while AutoRoute traffic may be transferred into external SDRAM, where packets may sit until they are scheduled for retransmission by the Traffic Manager. The SmartMem generates a reference to the stored packet data and context called Packet Descriptor (red arrows in Figure 23), which is further passed through the individual functional units in the NP.

The different processing paths in a FlexPath NP result in varying processing latencies. As there is a possibility to change the path for certain traffic types during system runtime, packets of the same connection may become out-of-order. This packet reordering has a negative effect on the efficiency of the most dominant TCP transmission protocol [78]. The reordered packets should therefore be re-sequenced by the architecture, which is achieved in FlexPath by the **Path Control**. After the DMA has happened, the arrival sequence of the incoming packet descriptors is recorded by the **Ingress Path Control**. This information is later used by the Egress Path Control in order to detect reordered packets before retransmission from the NP.

The packet descriptors are now assigned to the respective functional units (e.g. CPUs, hardware accelerators or the AutoRoute path) by the **Packet Distributor**. Depending on the type of processing element, this might require some amount of queuing and implementation of a suitable interrupt scheme for the respective types of processing elements. In case of a multi-hop processing path, the Packet Descriptor will be sent back to the Packet Distributor after each processing stage has completed its processing to reach the subsequent processing element.

The network processing complex will usually be implemented by a combination of generic programmable processors (e.g. embedded RISC cores, ASIPs or microengines) and a set of hardware accelerators for specific high-performance operations. As laid out in the prior art discussion in chapter 2.1.3, the programmable resources may be arranged either in a run-to-completion architecture or also as a processing pipeline (see Figure 4). With respect to the concept of various reconfigurable processing paths, the actual arrangement does not play an important role. In a symmetrical multi-processor cluster, the FlexPath functional modules may be used as means to perform load balancing or distributing traffic to CPUs, which are reserved for specific parts of the traffic (e.g. for QoS-sensitive applications or stateful processing applications). In addition, it is possible to invoke co-processor engines without prior CPU intervention, if the hardware accelerator is able to determine the necessary processing steps from the packet context alone. When regarding the pipelined processor approach, there may be different parallel pipelines or various entry points into a single pipeline for different application types. Here, the classification result can be used to choose a suitable pipeline or pipeline stage in advance.

After having traversed the network processing complex, the packet descriptors reach the **Egress Path Control**. Packet sequence is determined on a flow-bundle basis and out-of-sequence packet descriptors are queued before passing them onwards to the Traffic Manager.

As in every conventional NP, the Traffic Manager performs per flow and/or port queuing, possibly with several different priority levels to resolve output port contention and it may implement egress side traffic shaping. After this, the packet descriptors are handed over to the DMA engine, which fetches the packet data along with an optional packet context for the Post-Processor.

In the **Post-Processor** certain basic packet manipulations like MAC address replacement, TTL decrement, checksum calculation, etc. can be performed. The functionality has to be implemented thus far, that at least simple forwarding operations may be completed in order to enable the AutoRoute path. The Post-Processor operations are encoded in a set of low-level instructions, which are stored in the packet context that travels along with the packet data. The Post-Processor releases the completely processed packets towards the transmit side interfaces of the NP.

3.3. Concept Evaluation

Based on the overview of the functional units of a FlexPath NP, I will first focus on the expected benefits of the AutoRoute feature for the proposed architecture by means of an analytical calculation. Subsequently, a trace-based SystemC performance simulation model of a FlexPath NP architecture is developed and used to evaluate the system behavior with respect to overall system throughput and highlighting scalability issues. Reference simulation results illustrate the baseline performance of a conventional processor-centric NP. These results are then compared to simulations with partial hardware-offload provided by the Pre- and Post-Processor units and the AutoRoute feature.

3.3.1. Analytical Evaluation of AutoRoute in FlexPath NP

The AutoRoute functionality, where the entire processing burden is shifted away from the programmable resources to a pure hardware path, provides the most significant relief for the processor cluster. In turn, the saved instructions otherwise to be spent on the AutoRoute packets can be dedicated to other traffic types that are present in the application mix at the same time. In order to compare the processing performance of a FlexPath NP with AutoRoute versus a conventional NP architecture, consider the following case study. Let's assume a conventional NP architecture with a parallel processor cluster as found in current Cavium devices [26] with 32 dual-issue superscalar RISC cores operating at 1.5 GHz. This is compared to a FlexPath NP architecture featuring only 16 or 24 of these cores in the processor complex, but they are complemented with the AutoRoute functionality (see Table 7).

Table 7: Network Processing Complex Performance Comparison

	Conventional NP	*FlexPath NP*	*FlexPath NP*
CPU clock [f]	1.5 GHz	1.5 GHz	1.5 GHz
Packet Rate [r]	49 Mpps	49 Mpps	49 Mpps
CPU count [N]	32	16	24
CPI	0.5	0.5	0.5
Nominal Performance	96,000 MIPS	48,000 MIPS	72,000 MIPS
Avg. Instr. per packet (no AutoRoute) [IPP]	1,959	980	1,469

If we assume that a fraction b of the traffic may be forwarded using the AutoRoute path, the available number of instructions per packet for the remaining traffic share can be calculated with the following formula, given the parameters from Table 7:

$$IPP = \frac{N \times f}{CPI \times r}\left(\frac{1}{1-b}\right) \tag{3-1}$$

The resulting instruction budget IPP is plotted versus increasing AutoRoute shares b in Figure 24.

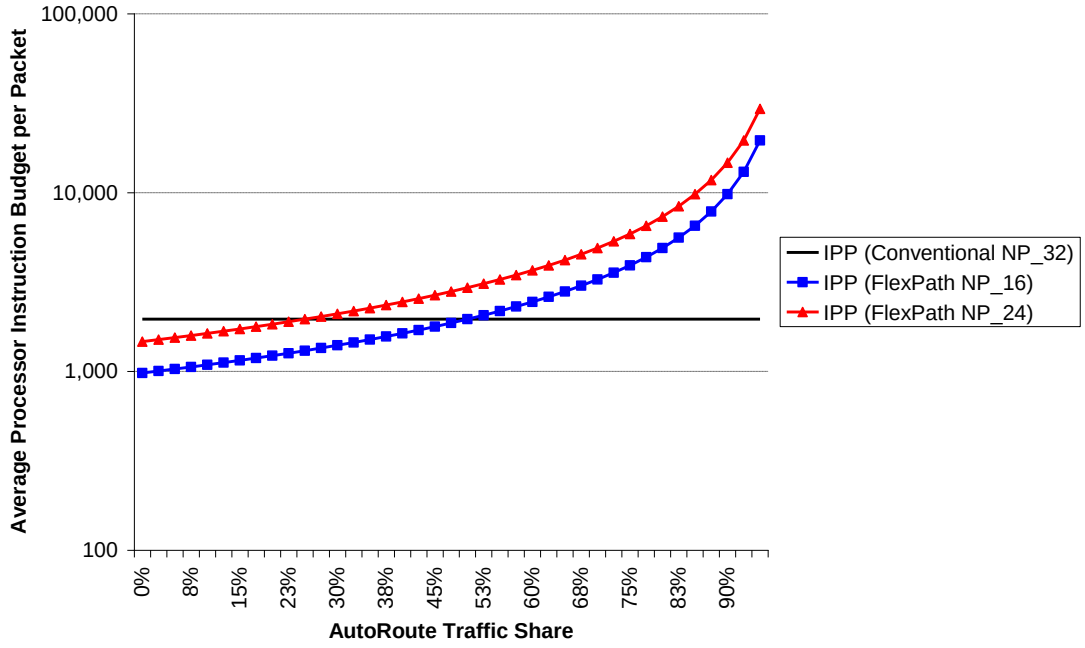


Figure 24: NP Processing Performance Comparison Conventional vs. FlexPath NP

As the figures in Table 7 show, the two FlexPath alternatives possess only 50% or 75% of the nominal processing performance compared to the processor-centric solution. However, by offloading parts of the traffic to the hardware-based AutoRoute forwarding path, CPU resources are freed and the resulting instruction budget for the remainder of the traffic increases. The break-even points for the investigated cases are reached at 25% offload for the FlexPath_24 and 50% for the FlexPath_16 scenario. Beyond these points the available processing budget is increasing dramatically; the 3,000 instruction limit for deep packet processing applications (see section 3.1) is matched at 53% and 68% offload with FlexPath_24 and FlexPath_16 respectively - this is already 1.53 times the average performance offered by the 32 core reference approach.

The critical question, which remains to be answered now, is: How much traffic can actually be offloaded to the AutoRoute path in realistic NP deployment scenarios?

There is not a general answer to this question, because the suitability of traffic for hardware offload depends on the precise application mix and processing requirements which differ greatly in various locations in the network. As I have argued before, AutoRoute is only possible for well-understood and stable protocol stacks, where the processing capabilities remain limited.

Plain IP forwarding is a typical representative of an AutoRoute-friendly application. As the next-hop lookup operation is usually implemented in NSEs (see [53], [54]), it is easily conceivable that such a lookup may already be invoked by the Pre-Processor, once the destination address has been retrieved from the packet header. The other necessary operations such as TTL decrement and checksum recalculation are covered by the Post-Processor. However, the offload cannot be used, if for certain reasons the processing of some or all of the traffic requires further inspection and recording of contents from upper protocol layers (e.g. application-layer information for application filtering, URL-based forwarding / load balancing, virus scanning, intrusion detection, etc.). However, these higher layer operations can by definition be applied only to packets carrying payload above the TCP header. As such, TCP acknowledgment packets without further payload provide a lower bound for the AutoRoute traffic share in an IP forwarding scenario. Internet traffic statistics made available by Sprint ([92]) show that there exist significant numbers of packets carrying only headers without further payload data. Statistics for 35 traces recorded in 2004 and 2005 are summarized in Figure 25 below.

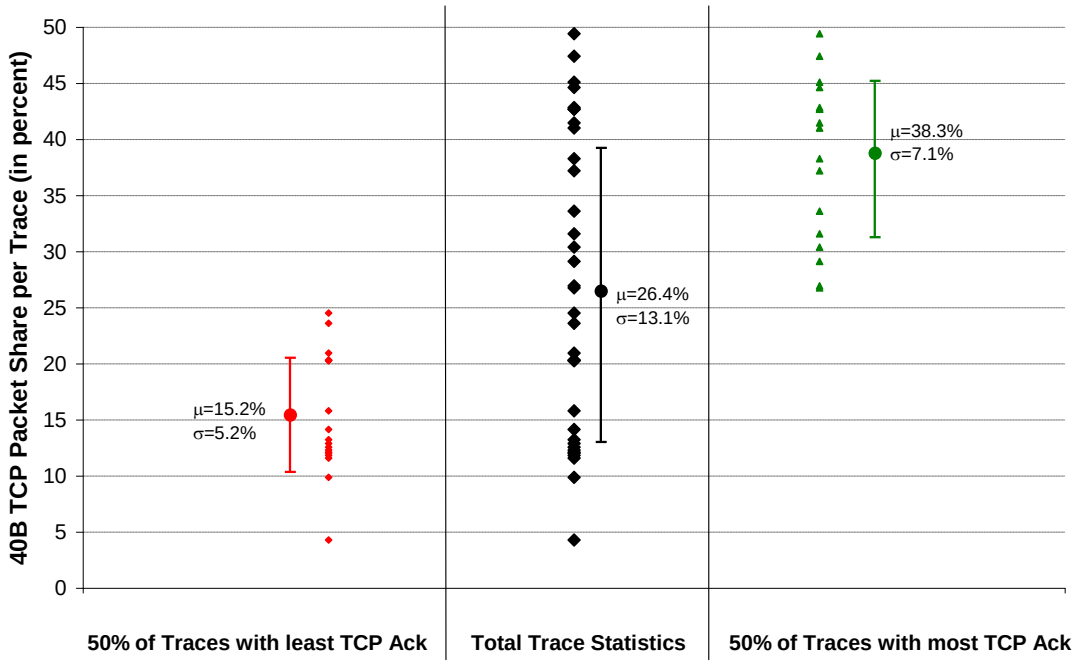


Figure 25: 40 Byte TCP Packet Shares from Internet Links recorded in 2004/2005

While the percentage of 40 byte packets from all traces is around 26.4%, the value is changing for different traces between 4.3% and 49.43% with a standard deviation of 13.1% (central black column). It can be seen, that the total statistic can be separated into two parts, a set of link traces (red column in the left) exhibit fewer short packets (average of 15.2% with a standard deviation of 5.2%) and the remaining traces (green column in the right) contain 38.3% of 40 byte packets on average with a standard deviation of 7.1%. Another side effect addressed by this

investigation is the fact, that it is exactly the shortest size packets are well suited for AutoRoute. These packets otherwise cause the highest event rates onto the central processor cluster (see Table 6). The remaining packets, which carry more information may be harder to process (there is more content to be inspected), but they also arrive with significantly smaller packet rates.

Another case for AutoRoute can be extracted from the wireless networking scenario described in chapter 2.2.5. Here, we observed that by chaining several network elements on a common link towards the next-higher hierarchical element, up to 90% of the traffic would simply be forwarded, while the remaining traffic is subject to more computationally challenging protocol conversions (see Figure 7, Figure 8).

The same mix of forwarding traffic and gateway traffic can also be observed at several places within the Internet hierarchy as described in Figure 1 (chapter 1) and chapter 2.2.6. The vast majority of traffic in the routers within the network will only have to be forwarded from one connecting interface to another on its way to the final destination. As laid out before, this forwarding is typically achieved by layer 2 (e.g. carrier Ethernet, ATM) or 2.5 (MPLS) switching. This switching can easily be assumed to be performed by AutoRoute, if the implementations of the Pre- and Post-Processors in the FlexPath NP are adapted to the respective protocol stack. In essence, not significantly more than a simple lookup and a few basic header modifications are necessary to complete the switching function. However, there will also be a (smaller) share of traffic, that has reached the final destination within the current network at the concerned router; and thus has to be forwarded into the neighboring network (either towards the core or access network). Here, the gateway functions necessary for protocol conversion (e.g. termination of MPLS forwarding and conversion to ATM over SDH) and access control or traffic shaping required for valid entry into the neighboring network have to be performed. These conversions may not be mapped to dedicated hardware in such a straightforward fashion as the previously described switching functions, and will therefore be performed with the use of the programmable processing resources of the NP. Still, the required number of programmable resources can be dimensioned to provide sufficient processing power for the fraction of gateway traffic, while the FlexPath NP hardware functions may route the significant share of switching traffic via an AutoRoute path.

3.3.2. Simulative Evaluation of Hardware-Offload in FlexPath NP

As the calculations in the previous section have already highlighted the benefits of the AutoRoute path in a FlexPath NP on a very high level of abstraction, I would now like to discuss the presented hardware-offload features offered by the Pre- and Post-Processor units. In order to get more precise results that also cover contention effects in multi-processor SoC designs with shared resource access, I will perform simulations that are able to capture some of the runtime effects, which are hard to be captured on a pure mathematical level. The following system-level simulation results have already been published in [82] and [83].

3.3.2.1. TAPES Simulator for FlexPath NP

The system-level simulations will be performed with the TAPES simulation framework that has been extensively discussed in [81]. TAPES is based on the SystemC language [8] and models the different functional entities of the system as abstract modules communicating with each other over defined interfaces. The model does not implement the functionality associated with each specific module, but is limited to executing traces describing the interactions of each module with the outside world. In this fashion, internal processing is abstracted to a simple processing delay, whereas the communication is performed by transactions across direct communication interfaces or a model of the central system bus.

A cycle-accurate model of the IBM CoreConnect PLB bus is used in the simulation that resembles arbitration delays and parallel read or write transactions with address pipelining and burst transfers like in a real implementation. By using such a highly detailed model, contention effects caused when several parallel entities perform concurrent transactions towards the memory modules can be accurately described. Figure 26 shows the resulting model of the FlexPath NP.

While the TAPES simulation framework in general allows using both artificial traffic and real traffic traces by importing pcap-files, only artificial traffic has been used for the subsequent simulations in order to better demonstrate worst case and best case results.

After reception of the packets, the Buffer Manager model initiates a sequence of bus write accesses, modeling the DMA operation of the segmented packet. In parallel, the Pre-Processor model spends a processing delay determined by the packet length of the actual packet, in order to model parsing of the header fields. After both the Buffer Manager and Pre-Processor models have finished, the Path Dispatcher model is activated, which synchronizes the results of the two previous elements (i.e. the Packet Descriptor coming from the Buffer Manager and the Packet Context coming from the Pre-Processor). The classification function is abstracted to a few cycles delay, and the packets can be configured to be routed either to the CPUs or the Traffic Manager (i.e. AutoRoute). However, this distribution is not based on

actual header fields, but packets are assigned in a preconfigured sequence. The Path Dispatcher model also comprises the queuing models of the Packet Distributor, i.e. packets can be held while the processors are busy working on previous packets and not reacting on new interrupt notifications from the Path Dispatcher model.

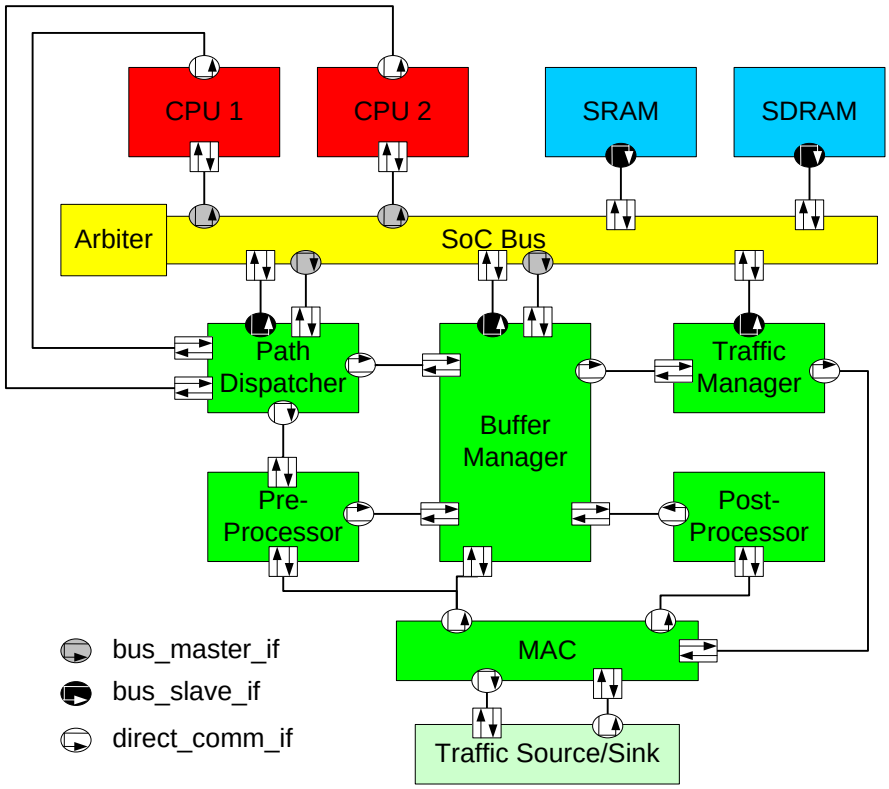


Figure 26: TAPES Model of FlexPath NP

The CPUs are executing a processing trace, which models the processing delay and intermittent bus accesses for both instruction fetches and packet data load operations. The actual trace will be based on a real-world application profiling, which is presented in section 3.3.2.2.

After processing, the packets are forwarded to the Traffic Manager model that contains a number of output queues and performs a simple round-robin scheduling among packets destined to four output interfaces. The destinations are not chosen based on packet data, but are randomly assigned.

The Buffer Manager model is again activated to perform the egress side DMA transactions and the Post-Processor model is reduced to a simple processing delay, consuming a number of cycles proportional to the length of the packet.

3.3.2.2. Calibration of System-Level Model with FPGA Measurements

In order to calibrate the system-level simulation model with real-world measurement results we implemented a rudimentary network processor on a Xilinx Virtex-II Pro development board with the functional modules according to Figure 27. It is important to note that this FPGA design does not implement the full functionality of the FlexPath NP concept (this will be presented later in chapter 6). However, by using measured data concerning the DMA operations and by profiling the forwarding performance of a real networking stack on an embedded PowerPC processor, we expect to gain more accurate results than by relying on published benchmarking results and datasheet performance figures.

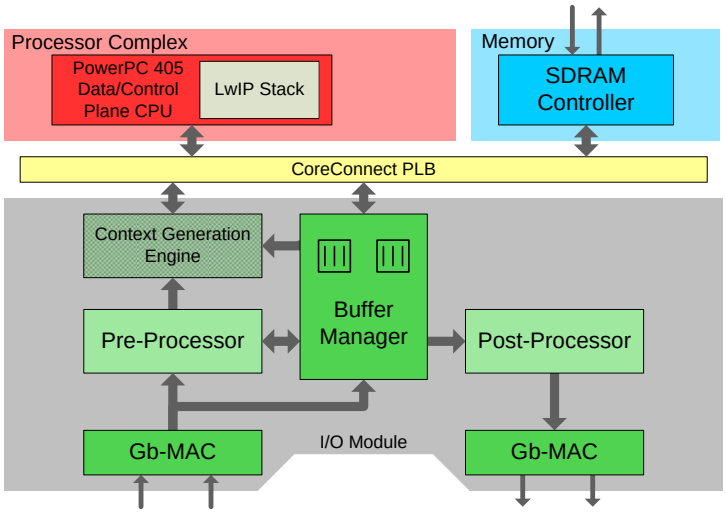


Figure 27: Calibration Prototype Implementation on a Virtex-II Pro FPGA

The Pre-Processor extracts relevant packet fields from the incoming packets using a set of field extraction units that are controlled by a static FSM that provides appropriate control for ARP, IPv4, TCP and UDP packets. In parallel to the Pre-Processor, the Buffer Manager [94], which is a simplified DMA engine used later as reference to the improved SmartMem architecture ([88], [89], [108]), splits the packets into 64 byte segments and stores them with a linked-list structure in the external SDRAM. A packet descriptor is generated that contains the pointers to the linked list of segments. In this stage, no Path Dispatcher and Packet Distributor units are available. The main purpose of the system level simulations is to demonstrate the expected benefits of the FlexPath NP architecture with respect to the hardware offload and AutoRoute scenarios. This can be achieved by statically assigning the traffic either to the CPU or the Traffic Manager. Instead, another module called Context Generation Engine [102] performs the DMA operation of the extracted header fields and status flags from the Pre-Processor in a data structure referred to as CII (Context Information Input).

A single embedded PowerPC is used to execute a slightly adapted form of the open-source lightweight IP stack (LwIP, [95]), that works together with the DMA offered by the Buffer Manager and may make use of the CII information generated by the Pre-Processor. If some of the necessary packet manipulations shall be performed by the Post-Processor (hardware offload), a CIO (Context Information Output) data structure may be added that contains the assembler-like instructions for the Post-Processor along with the respective data fields.

After processing, the CPU sends the packet descriptor back to the Buffer Manager, which retrieves the packet and an eventual CIO from memory and transmits it over the gigabit Ethernet MAC through the Post-Processor.

The FPGA development platform features an external 32 MB SDRAM memory that will be used for packet and context storage and the instructions for the PowerPC. The linked-list structures of the Buffer Manager are stored in a small on-chip SRAM (BlockRAM).

The processing traces for the CPU traces have to be derived from an application profiling of the IP stack. In order to obtain the full instruction count for the packet processing functions, the interrupt service routine and all forwarding functions were profiled by sending a single packet through the FPGA prototype. In addition to stepping through the individual code lines on assembler level, an integrated Xilinx ChipScope Bus Analyzer core allowed recording number and frequency of cache line transactions of the PowerPC on the PLB bus and measuring the execution time of the entire packet processing routine. Table 8 and Table 9 summarize the profiling results. The figures in the reference solution column refer to the implemented version of the LwIP stack that makes use of the Buffer Manager as autonomous DMA engine, but the processing itself is based purely on the packet header, i.e. CII and CIO is not integrated into the software. As the DMA function is not performed by the CPU, the amount of instructions that need to be executed for each packet is independent of the packet size. Although implementations of the Pre- and Post-Processor units were already available when the calibration prototype was generated, using the packet context had not been integrated into the LwIP stack. Therefore, the potential savings effect that may be achieved by performing operations based on the CII information (integrity checks are already performed in hardware) and writing a short CIO to offload checksum recalculation, TTL decrement and replacement of the MAC addresses to the Post-Processor have to be estimated. The expected figures are found in the right columns of Table 8 and Table 9.

Table 8: Profiling Results of modified LwIP Stack on Calibration Demonstrator

Function	*Instruction Count (SW reference)*	*Instruction Count Estimate (HW offload)*
Entry into ISR	20	20
Data structure initialization	362	362
Update ARP table	535	477
Receive Integrity Checks	523	153
Next-hop Gateway lookup & forwarding	96	86
ARP query and packet modifications	568	470
Transmission of packet descriptor and freeing of data structure	237	237
Function call returns and end of ISR	80	80
SUM	2,421	1,885

Table 9: CPU Execution Time and Bus Access Patterns

	SW reference	*Estimate w/ HW offload*
Processing Delay	5,080 clock cycles	3,955 clock cycles
Instruction Cache Fills	41 × 32B	32 × 32B
Packet Descriptor read/write	2 × 16B	2 × 16B
Packet Header read/write	2 × 64B	--
Packet Context read/write	--	2 × 64B
CIO descriptor write	--	32b

The estimates performed for the hardware offload scenario show that we are able to save up to 22% of the originally required instructions. The major contribution (15%) comes from offloading the receive side integrity checks. As far as the bus transactions are concerned, the number of instruction cache fills is reduced proportionally to the number of instructions. There are only minor differences on the data cache operations, as fetching the CII information and writing back a CIO, consist of two 32 byte cache line operations each, and this is equal to accessing the first 64 bytes of the packet data.

When analyzing the bus transactions for the different memory types in the system, we obtained single access patterns for accessing on-chip SRAM (i.e. BlockRAM) and an asymmetric 10-4-4-4 cycle burst read and 4-1-1-1 burst write pattern on the external SDRAM with the given Xilinx memory controller as PLB slave.

3.3.2.3. Conventional NP Reference Performance and System Scalability

The system model is stimulated by four gigabit Ethernet interfaces, each carrying a load of 750 Mbit/s, such that the aggregate traffic arriving at the NP is 3 Gbit/s. This amount of traffic exceeds the forwarding performance of a PowerPC by far, and therefore allows determining the maximum throughput the investigated architecture would be able to deliver.

In the following simulations, the SW reference traces are used as presented in the center column of Table 8 and Table 9. The simulation results for a single CPU running at 200 MHz, while the rest of the system is running at 100 MHz, directly correspond to the implementation on the calibration prototype.

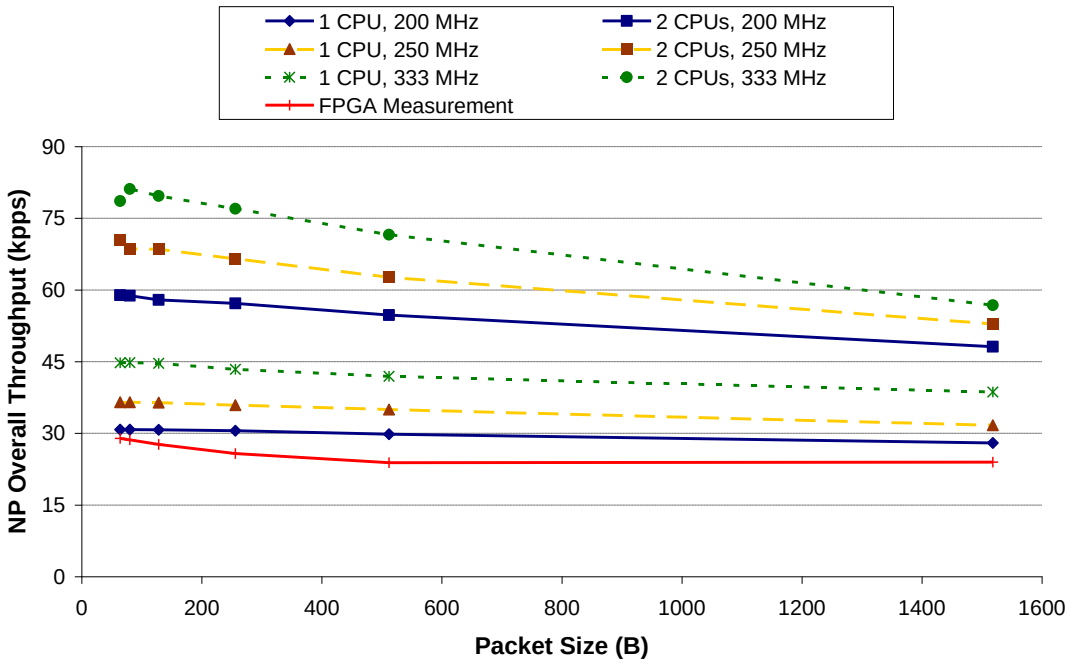


Figure 28: SW Forwarding Performance of Reference Scenario

The simulation results shown in Figure 28 yield a single CPU forwarding performance of 30.77 kpps (kilo packets per second) for 64 byte packets and 27.99 kpps for 1518 byte packets. This compares to measurements taken on the prototype, which show a decline from 28.95 kpps to 23.95 kpps. While the simulation mismatch is only 6% for the smallest size packets, the error becomes larger for packet sizes beyond 256 bytes. The simulations have also been repeated with a dual CPU setup and scaled CPU frequencies of 250 MHz and 333 MHz.

An interesting aspect is the scaling efficiency of the investigated system by either increasing the processor clock frequency or adding additional cores:

– When a second CPU is added to the system (i.e. 100% more processing power), the forwarding performance increases from 30.77 kpps to 58.90 kpps, which is an

increase of 91.4%. The resulting scaling efficiency can be computed as 91.4% ÷ 100% = 91.4%.

– In turn, by accelerating the clock frequency of a single CPU by 66.5% to 333 MHz, the forwarding performance is only increasing to 44.74 kpps, which is an increase of 45.4%. This results in a scaling efficiency of 45.4% ÷ 66.5% = 68.3%.

From these results it can be concluded that increasing the number of processor cores is more efficient than scaling the frequency of the CPUs alone.

In general, the decline in forwarding performance is little for small packets (i.e. small throughput, as the packet rate is constant) and a single CPU with a slower clock frequency. This result is also well expectable as the same amount of code has to be executed for every packet. As the packet size grows (i.e. the system throughput increases) or additional CPUs are added to the system (i.e. the packet rate in the system increases), the load on the system bus increases. This leads to more collisions between the different bus master modules. The longer average access times result in a performance degradation, especially when considering that the CPUs need to read 41 cache lines (i.e. 1312 bytes!) from the shared instruction memory while processing a single packet. These instruction fetches are also independent from the packet length.

3.3.2.4. CPU Offload Performance Evaluation

In this section, the estimated performance improvement achievable by making use of the FlexPath hardware offload possibilities is presented. The results of the conventional baseline NP simulation showed that scaling to more cores works better than increasing the processor frequency. Consequently, the CPU frequency is fixed at 200 MHz but the investigations are extended to system setups with four processor cores in the NP. An additional scenario investigates the effect of moving the software code from the SDRAM into on-chip SRAM which has a significantly reduced access latency.

As I have laid out in chapter 3.3.2.2, the hardware offload available through Pre- and Post-Processor in a FlexPath NP allows reducing the executed number of instructions by 22%. This should lead to an increase of the NP's forwarding performance by 28% $\left(\frac{1}{1-22\%} = 1.28\right)$. The simulation results for a single CPU exhibit a slightly smaller improvement by 27.5% for 64 byte packets and 24.7% for 1518 byte packets (see Figure 29). Again, this may be explained by the fact that increasing bus loads lead to a deterioration of the system's overall throughput and stresses the fact that the commonly shared resources (bus, memory) may become potential bottlenecks.

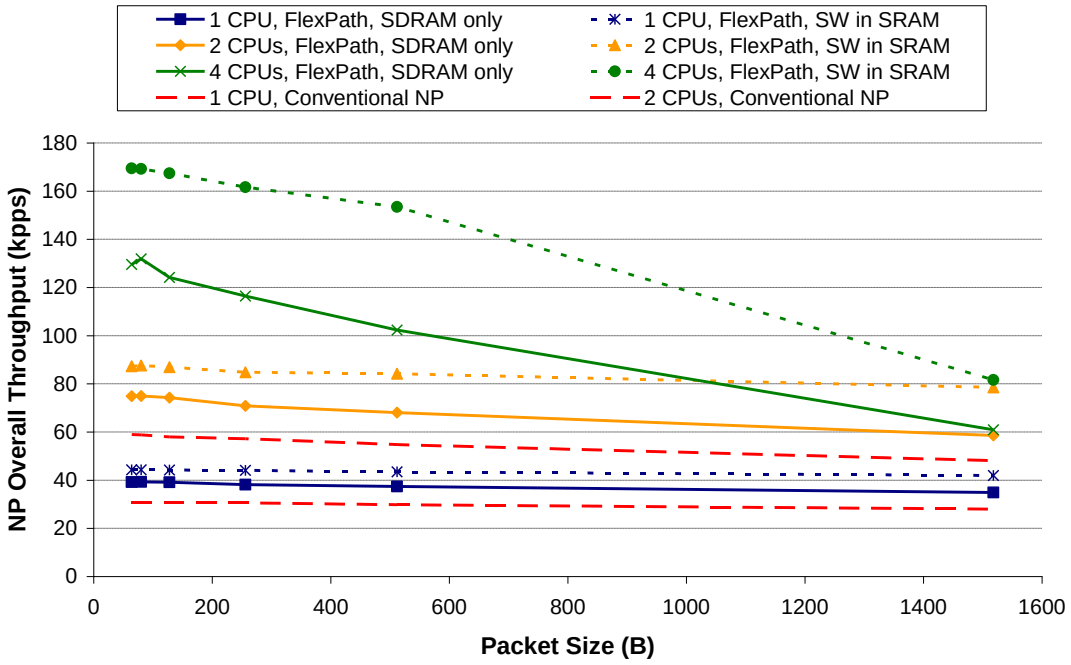


Figure 29: SW Forwarding Performance of FlexPath NP using HW Offload

Moving the software code from the external SDRAM memory into an internal SRAM increases the single CPU performance by 12.8% for 64 byte packets and even 20.3% for the 1518 byte packets compared to the FlexPath performance with software being mapped to the external SDRAM.

When scaling the processor complex to four parallel processors, another system bottleneck is revealed. While the scaling works quite well for smaller packet sizes, the forwarding rate of the 1518 byte packets is reduced to 81.6 kpps, while it is still above 150 kpps for the 512 byte packets. The packet rate of 81.6 kpps translates into a throughput of 990.9 Mbit/s, which is only one sixth of the PLB read or write bandwidth of 6.4 Gbit/s (64 bit bus clocked at 100 MHz). Assume only the transfer of packet data to and from the SDRAM memory and neglect all other transfers for a moment. The Buffer Manager transfers data in bursts of 64 bytes, which means eight consecutive accesses of 64 bits. Taking into account the memory access patterns for the SDRAM, such an 8 word burst can be written in 11 cycles, but it takes 38 cycles to retrieve the same amount of data from the memory. Multiplying the raw bus bandwidth of 6.4 Gbit/s with the memory read efficiency of $\frac{8}{38}$, we receive a maximum throughput of 1.347 Gbit/s. When taking into account the other necessary transfers like processor instruction cache refills and fetching the CII from the memory (which is in case of 1500 byte packets small compared to the packet length), the data rate limitation at 1 Gbit/s can be explained.

3.3.2.5. AutoRoute Performance Evaluation

Figure 30 shows the performance of a single CPU FlexPath NP using the hardware-offload capabilities discussed in the previous section for CPU-destined traffic shares and the AutoRoute path taken by 20%, 40%, 50% and 70% of the incoming traffic. This performance is compared to the FlexPath NP architecture without AutoRoute for one, two and four CPUs as discussed in Figure 29 and the baseline conventional NP with a single processor. For this set of simulations, the CPU code is mapped to the SDRAM memory.

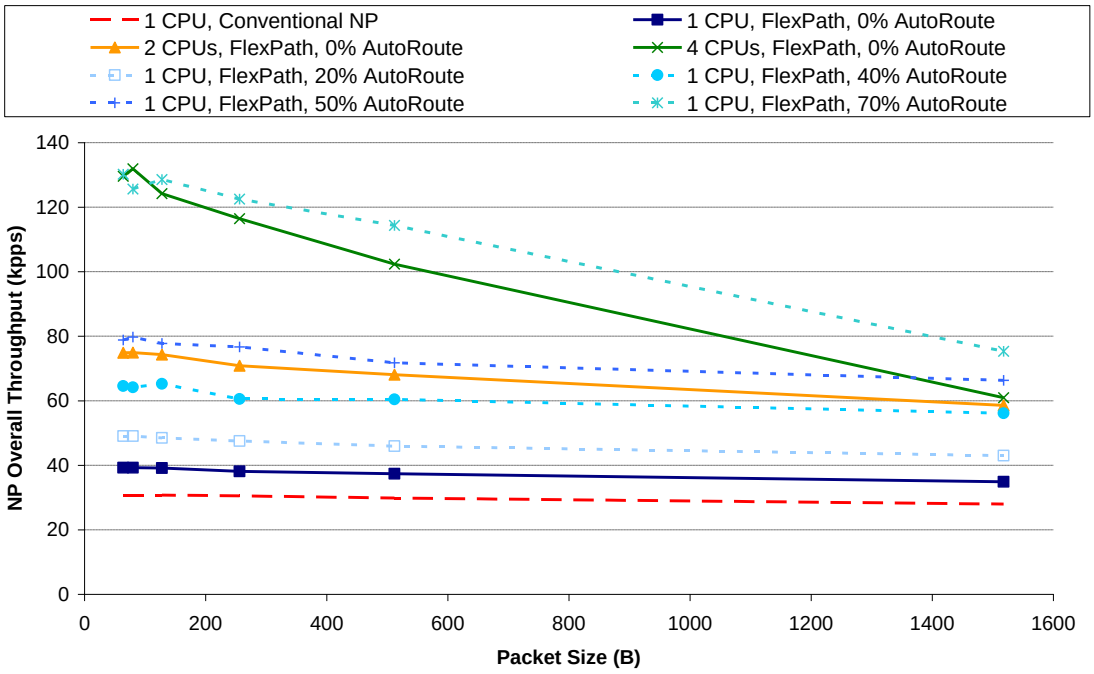


Figure 30: Forwarding Performance of FlexPath NP with AutoRoute

Although the AutoRoute path itself should be capable of significantly higher forwarding rates as a single CPU, head-of-line blocking effects in the Path Dispatcher model disallow AutoRoute packets passing through the system while the buffer for CPU-bound packets is filled (backpressure).

However, it can be seen that the forwarding performance of the system with 50% AutoRoute packets is 5% better than when a second CPU would be added to the processor complex. As the AutoRoute packets do not add to the system bus load by the instruction cache accesses associated with software-based forwarding, the performance decrease previously observed for larger packets is also less significant than in the previously inspected scenarios. The performance of a system with 70% AutoRoute and a single processor even exceeds that of a four-processor software-only forwarding FlexPath NP for packet sizes beyond 128 bytes, with an increasing benefit for the larger packet sizes.

But AutoRoute is not only an interesting feature from the point of view of overall NP throughput. AutoRoute also has a latency advantage over CPU-processed packets, which is shown in Figure 31 below. In this simulation, the NP is not any longer driven into overload in order to obtain the peak system throughput; but traffic consisting of 64 byte packets is slowly increased until the processing resources are fully utilized.

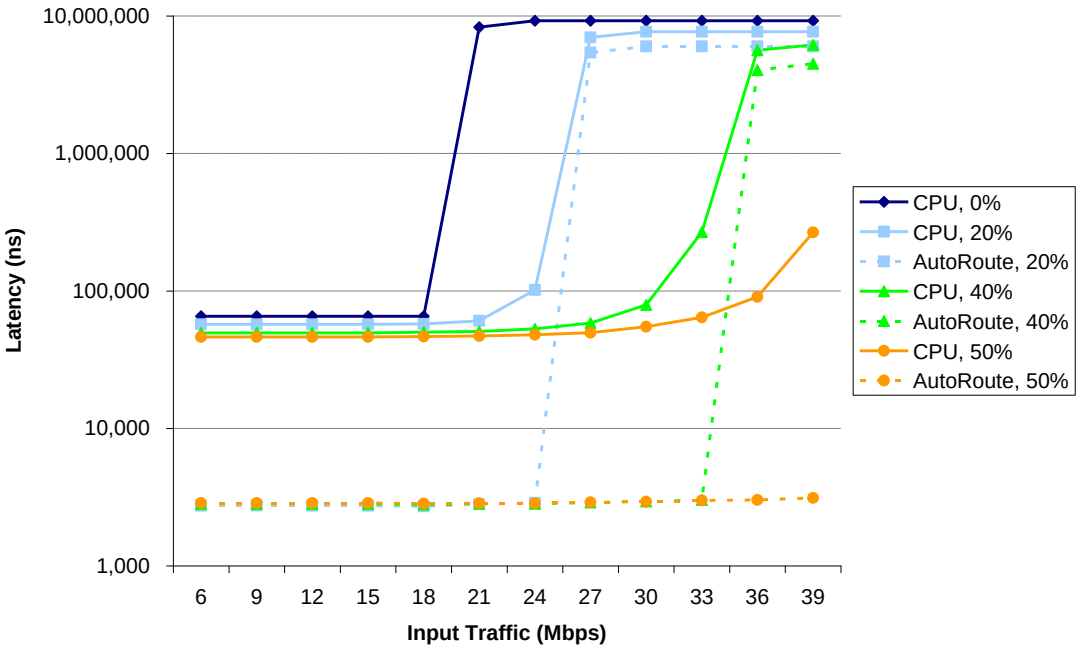


Figure 31: Latency Comparison CPU Path vs. AutoRoute Path

As the processing on the AutoRoute path is accomplished in pure hardware in a pipelined architecture with aggregated line-speed capability, the processing latency is significantly lower than any CPU-based forwarding implementation could be. When the maximum throughput of the NP system is approached, the buffers in the system are filled and the latency of the individual packets rises to a level that is defined by the maximum buffer size. The maximum latency for CPU-bound packets is 9.2 ms and for AutoRoute packets it is 6.0 ms. The difference can be explained by the fact, that AutoRoute packets potentially get queued in the receiving MAC and the Buffer Manager model - CPU-bound packets have to traverse the additional FIFO in the Path Dispatcher model in front of the processor complex.

The latency advantage of AutoRoute packets is however depending on the packet length. As it can be expected, short packets will consume only little DMA transfer times, and thus the resulting latency will be the lowest. The simulation results presented in Figure 32 show the simulated processing latencies for increasing packet sizes for traffic at a fixed rate of 5 Mbps, so that no queuing effects due to processor overload or output port contention can be observed.

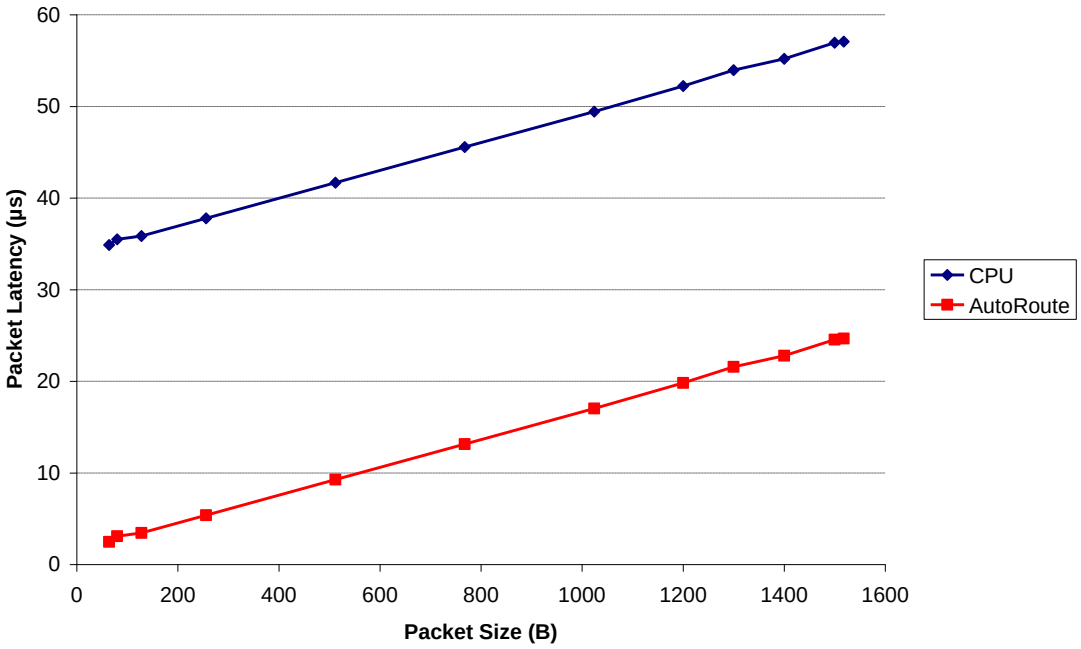


Figure 32: Processing Latencies of AutoRoute and CPU-processed Packets over Increasing Packet Size

In accordance with the results shown in Figure 31, the latencies for 64 byte packets are 2.5 µs for AutoRoute and 34.9 µs for the CPU path. As the packet size increases, we can observe a linear increase in the packet latency, which reaches 24.7 µs for AutoRoute and 57.1 µs for CPU-bound 1518 byte packets. Here, the time needed to store the packet in the system memory and retrieve it from there approaches the processing time in the CPU, which is independent from the packet size for a simple IP forwarding application. Therefore, the latency advantage of AutoRoute packets is reduced from an order of magnitude in case of 64 byte packets to a factor of two for the maximum length Ethernet frames.

3.4. Conclusions

In this chapter I have derived the concept of FlexPath NP based on observations about current networking applications. A FlexPath NP appends state-of-the-art network processor architectures with special hardware units for pre-processing, packet classification, packet sequence control and post-processing. Apart from offloading a central, software-programmable processor cluster with the presented hardware modules, a pure hardware-based processing path through the system called AutoRoute is proposed.

When comparing the FlexPath approach with other architectures from the state-of-the-art, the conceptually closest architecture is the SafeNet inline security engine (see section 2.1.1.2) of which first details were released about half a year after the first FlexPath publication ([7]). The fact that this architecture is still actively marketed today underlines that the concept of completely bypassing programmable elements for certain traffic types is a viable and successful approach. Another "close competitor" would be the cache-based NP proposed by Hitachi in 2006 ([34]). Here, the NP also possesses Pre- and Post-Processor units and a classification unit, but every packet of a certain stream has to go through the programmable units first, before they may be routed over the hardware path. In FlexPath, AutoRoute may be enabled for certain traffic types without the requirement of going through the processor complex with at least one packet per flow.

Initial analytical investigations show the theoretical benefits of combining AutoRoute and software-based processing in the NP. Several examples of current networking applications are discussed with respect to a suitable mapping of certain traffic shares to either on-chip processing path. Traffic shares ranging between 20% and 90% for different applications, which are suitable for hardware-only processing, make the FlexPath approach relevant enough for further investigation.

In a next step, system simulations have been performed to further investigate the expected performance improvements that can be achieved with the FlexPath-specific extensions. The simulation model, which abstracts processing to simple delays and transactions between the individual system modules, has been calibrated with a "first shot" implementation on an FPGA-based prototyping platform. The presented investigations suggest the following propositions:

– Based on the application profiling results of an open-source networking stack, 22% of the instructions could be saved by using context information (CII and CIO) instead of analyzing and manipulating the packet data directly. The most dominant relief (68% of the savings) is due to the Pre-Processor, which completely offloads packet integrity checks to hardware.

– AutoRoute packets traverse the NP with significantly reduced latency compared to packets being processed by the CPUs. Pre- and Post-Processors work on the packets on a hardware pipeline structure, such that only a few clock cycles of latency will be added during reception and transmission of the packet. The total latency of AutoRoute packets in the system is dominated by the DMA time required for storing and retrieving the packet in the system memory and any possible queuing delays associated with output port contention. As the DMA time is directly proportional to the packet length, the total latency of AutoRoute packets in contrast to CPU-bound packets varies from roughly 10% for 64 byte packets to 50% for 1518 byte packets.

In addition to analyzing the benefits of hardware-offload in the FlexPath NP architecture, some more general results about the scaling efficiency and potential bottlenecks in multi-processor systems have been revealed:

– The system interconnect and the common shared dynamic memory for storing packet data and processor instructions becomes the main bottleneck in a system with more than two processors. By providing a separate SRAM with single cycle access patterns for the instruction code, the performance of the system can be improved by an additional 13% to 21%. This roughly matches the expected benefits that can be achieved by the hardware-offload features in the FlexPath NP architecture.

– The SDRAM packet memory was identified as an additional performance bottleneck. Considering the measured access patterns, the throughput of the demonstrator system is limited around 1 Gbit/s, although the ingress and egress packet processing entities would be able to process the packets with a full line speed of 3.2 Gbit/s assuming a 32 bit data path operated at 100 MHz. The performance-limiting operations are the read accesses of the buffer manager, which are constrained by the slow read access patterns of the DRAM.

In summary, the achieved simulation results support the expected benefits of the FlexPath NP architecture. Consequently, the following two chapters focus on the classification problem in the ingress path of the proposed architecture and on advanced load balancing and QoS provisioning techniques in a FlexPath NP.

4. Concept and Implementation of Path Dispatcher

The Path Dispatcher is a crucial element in the ingress data path pipeline of the FlexPath NP architecture and executes the most challenging task - classification of the arriving packet stream under hard real-time constraints and assigning them to an appropriate processing path. Because of the importance of the Path Dispatcher component, I have devoted a whole chapter for this topic, which is structured as follows:

- Section 4.1 outlines the constraints of the on-chip packet classification problem found in the FlexPath NP and compares them to existing classification problems.
- Section 4.2 derives the Heterogeneous Decision Graph Algorithm (HDGA), which I propose to solve the classification problem in the Path Dispatcher.
- Section 4.3 presents simulation results that illustrate the performance of HDGA.
- Section 4.4 focuses on an efficient hardware implementation of HDGA. The elaboration includes an extensive design space exploration and optimizations of the HDGA data structures that allow a more efficient implementation.
- Section 4.5 concludes the chapter by summarizing the main characteristics of the HDGA concept and implementation and highlights the contributions to the state-of-the-art in packet classification.

The concepts of HDGA have already been published in [56] and [84].

4.1. Introduction and Problem Formulation

The problem of packet classification is not new, and has gained increasing attention since the advent of newer QoS-sensitive applications in the Internet after the late 1990's. In the prior art chapter 2.3, a number of hardware and software algorithms have been introduced, which handle the task of separating different service classes in networking equipment and give them an application-specific treatment with respect to queuing, forwarding, traffic shaping, etc.

Our proposed FlexPath NP architecture, which improves system performance by provisioning different processing paths through the NP system, also requires a packet classification function in the ingress data path pipeline before the packets reach the Network Processing Complex (see Figure 23 in section 3.2). Arriving packets are classified according to the specific application classes they belong to and are then mapped to an appropriate, optimized processing path in the NP. The mapping should also encompass the problem of load balancing among different forwarding paths and multiple processors in the central network processing complex. While the task of the Path Dispatcher unit is formally a multi-field packet

classification task, it has its own domain-specific requirements and constraints, which differ significantly from the traditional five-tuple packet classification problem addressed by prior art schemes.

In the following paragraphs, I will outline the specific environment in which the Path Dispatcher classification problem is situated and derive an example rule base, which exhibits the typical requirements found within the FlexPath NP system. The analysis of these requirements is then used to motivate and illustrate the derivation of the HDGA classification algorithm (see 4.2) and highlights the important differences to the problem setting in the prior art.

As we have seen in 3.2 (further implementation details on the ingress data path follows in 6.2), all packets arriving from the networking or fabric interfaces pass through the Pre-Processor and Context Assembler units. The Pre-Processor checks the packets' integrity (correct packet and frame lengths, MAC and IP header checksums, etc.) and extracts a selection of header fields depending on the protocol stack of the packet. In addition, these fields are also compared against a given set of values and depending on the result of such comparisons certain flags are generated. Examples for these flags could be IPv4, Control Plane Protocol (e.g., ARP/RARP, ICMP, IPsec, TCP, UDP, etc.). For each incoming packet, the Pre-Processor generates only the fields and flags present in the respective packet; this information is then transformed into a uniform standard representation called Raw Context covering all possible packet types and protocols by the Context Assembler unit. The Raw Context is then used in the Path Dispatcher to determine the further processing path of the packet inside the NP system.

Figure 33 shows the important functional units of a FlexPath NP implementation with n parallel data plane processors, a single control plane CPU and a hardware accelerator in the network processing complex. The necessary DMA functions and the egress path processing elements have been excluded for clarity, as they are not relevant to the Path Dispatcher function. The full view of all elements is later described in chapter 6.2. Based on the FlexPath NP system of Figure 33, I will introduce an example application mix from which I derive the different processing paths and explain the conditions under which they will be chosen. The application example should give the reader a good insight into the properties and effects of our multi-field classification problem and may be transferred to a more general setup in a straightforward way.

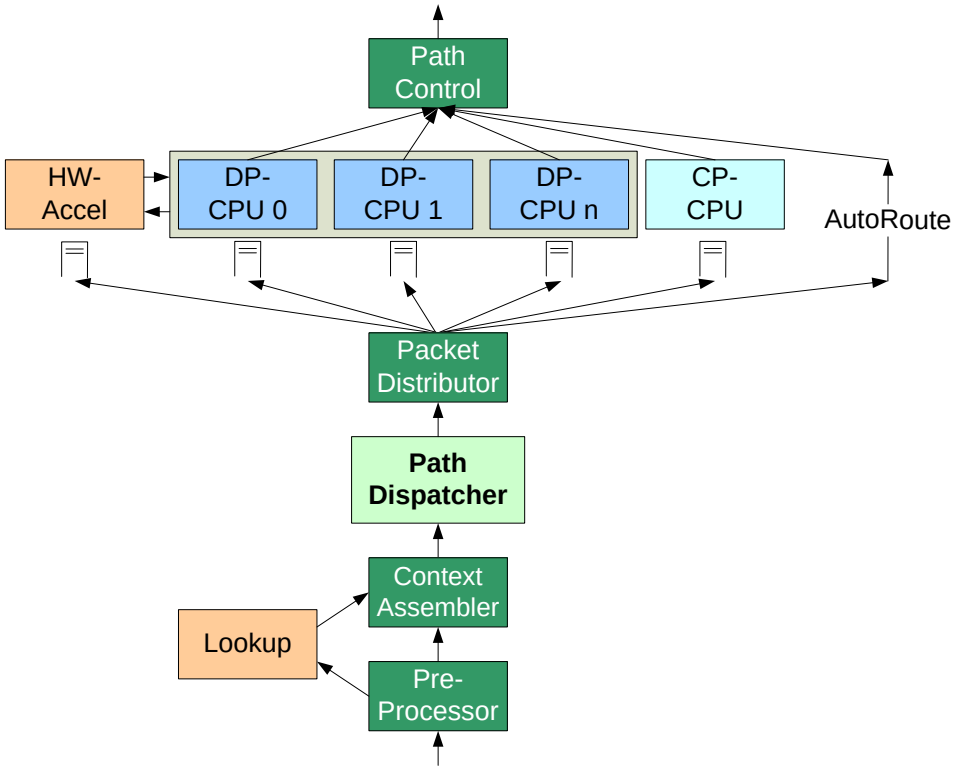


Figure 33: FlexPath NP with Data and Control Plane CPUs, Hardware Accelerator and AutoRoute

The FlexPath NP as described above supports the following path decisions:

– *Discard:* Packets that are received with invalid checksums (e.g. Ethernet or IP header) are not forwarded to the processor core but may be silently discarded in the ingress data path pipeline. This saves both processor resources and bus cycles by potentially omitting the DMA in SmartMem. The path is chosen by rule 1 in Table 10.

– *Control Plane CPU:* Packets with expired TTL values may not be silently discarded as above, but an ICMP error message has to be generated, which is typically a task allocated to the control plane CPU. In addition, all packets belonging to the network control and management plane (e.g. routing protocols, SNMP, etc.) have to be forwarded to the control plane. These packets can be easily identified by their IP address, which must be the router's own address (i.e. the Own Flag will be set by the Pre-Processor). This path is chosen by rules 2 and 5 in Table 10.

– *Hardware Accelerator:* Encrypted packets of terminating tunnel connections may be sent directly to a decryption core before the payload can be processed effectively by a data plane CPU. This saves unnecessary CPU interrupts and increases processing efficiency. These packets may be identified by a set Own Flag (because the router must be the end point of the IPsec tunnel) in combination

with an IPsec protocol (i.e. ESP and/or AH) as layer four payload. This case is represented in rules 3 and 4 in Table 10. Unencrypted packets from specific networks that require IPsec encryption (cf. provider-based VPN services as outlined in chapter 2.2.3) may also be separated from the remaining traffic and will be guided to a specific CPU. This can effectively be seen as a hardware offload of the necessary IPsec SPD check to the Path Dispatcher and is reflected in rule 10 of Table 10.

– *AutoRoute:* As discussed in chapter 3.2, packets known to require only plain forwarding functionality can be offloaded to the AutoRoute path, if a valid next-hop lookup result could be determined by the Pre-Processor (cf. Figure 33). Rules 7, 8 and 9 in Table 10 allow such an offload for VoIP packets that are assumed to be marked with a unique DSCP value in the IP header and for plain TCP acknowledgment packets without further protocol payload. According to the specified DSCP forwarding priority, these packets are still assigned to different queuing priorities in the system.

– *Data Plane Processors:* Essentially all remaining packets would have to be forwarded directly to the data plane CPU cluster. Here again, a differentiation according to IP DSCP field values might be useful in determining "high priority" processing queues and "best effort" queues. In addition, based on further details of the traffic, different load balancing strategies may have to be applied for stateful and stateless packet applications, which will be further elaborated in chapter 5. Rules 11, 12 and 13 in Table 10 reflect a high priority dedicated queue, flow-preserving hash-based assignment (e.g. HLU, see chapter 5.2.2) to a specific processor and best effort traffic that can be worked off by any free CPU (e.g. spraying, see chapter 5.2.1).

The total number of paths that needs to be differentiated in our example is limited to 10, assuming that two priorities (for DSCP = 0 and DSCP > 0) are available for both AutoRoute and data plane CPU bound packets. As the number of supported service classes in a specific system can scale up to ten different classes [38] and we can also conceive systems with several different hardware accelerators, scenarios with up to 30 different paths may be encountered in a generalized FlexPath NP implementation, but certainly not many more.

The traffic shares that will be routed to the different functional entities (i.e. hardware accelerators, processors, AutoRoute) will remain quasi-static during operation of a system, i.e. the assignment will change only very infrequently. The assignment of flows to paths will change more frequently with usage statistics and traffic load variations during system runtime for flow-aware classification rules, which could for example be load balancing rules or IPsec network-specific rules (cf. rules 10 and 11 in Table 10).

Table 10: Example Path Dispatcher Rule Base

Rule Number / Priority	*Condition*	*Path*
1	Invalid_Flag = 1	Discard (0)
2	TTL_exp_Flag = 1	CP-CPU (1)
3	Own_Flag = 1 AND ESP_Flag = 1	Decryption Core (2)
4	Own_Flag = 1 AND AH_Flag = 1	Decryption Core (2)
5	Own_Flag=1	CP-CPU (1)
6	Ethertype != 0x0800	CP-CPU (1)
7	DSCP=2 AND Lookup_Flag = 1	AutoRoute_VoIP (10)
8	DSCP!=0 AND TCPAck=1 AND IP_Length=40 AND Lookup_Flag=1	AutoRoute_High (11)
9	DSCP=0 AND TCPAck=1 AND IP_Length=40 AND Lookup_Flag=1	AutoRoute_Low (12)
10	IPsec_network_hit = 1	DP-CPU 0 (20)
11	DSCP=0 AND TCP_Flag=1 AND Hash(IP5Tuple)_hit=1	DP-CPU (*Hash) (100)
12	DSCP!=0	DP-CPU 1 (21)
13	Default rule	Any DP-CPU (30)

The extracted header fields and flags that determine the processing path are dependent on the application. It is important to realize that not all fields are present in every incoming packet. As a consequence, the formulation of the individual classification rules will be quite heterogeneous, in contrast to the classical multi-field classification approaches where the Internet five-tuple of IP source and destination addresses, layer four port numbers and layer four protocol field are fixed inputs. For each path specification or classifier rule, between a single and up to four or five different header fields and flags may be sufficient, while the total set of possible fields and flags that have to be inspected over the whole range of applications can easily grow to an order of 20 to 30. These constraints are compared in Table 11 below.

Table 11: Characteristic Properties of Traditional Single-Field and Multi-Field Classification vs. Path Dispatcher Requirements

	IP Next-Hop Lookup	*IP Five-tuple Multi-Field Classification*	*FlexPath NP Path Dispatcher*
Number of Rules	1,000 - 100,000	100 - 10,000	10 - 100
Number of Actions / Paths	5 - 1,000	3 - 100	10 - 30
Header Fields per Rule	1	5	1 - 5
Header Fields per Classifier / Rule Base	1	5	10 - 50

The Path Dispatcher is to be integrated into our system-on-chip design along with the other NP building blocks. Consequently, it is an important constraint for the classification algorithm to be compact enough to fit into a small part of the available chip area while still achieving aggregated line speed throughput, so that the ingress hardware processing pipeline structure can be maintained and the packet path classification does not become a system bottleneck.

As a common characteristic, the schemes proposed in the prior art are all focused on IP five-tuple classification or subsets thereof. The classification works always on a constant set of header fields and the matching conditions are either direct matches, range matches, prefix matches or wildcard parameters (see also 2.3.4). Range and prefix matches can be effectively addressed by tree or trie structures, however they suffer from an exponential memory size requirement, if exact matches have to be determined. Multi-stage approaches have been proposed (e.g. RFC, HyperCuts, Crossproducting, Grid-of-Tries and DCFL) that first search for matching entries in single dimensions and then successively combine results to compute the final classification output. However, a straightforward application of these approaches to our classification problem with 20 to 30 dimensions would not scale well. The implementation of 20 to 30 parallel single dimension search engines alone would be associated with a significant cost and it is unclear what modifications would be necessary, if the number of dimensions needed in the classification changes during system operation as a new application may be added to the current mix.

In the FlexPath NP Path Dispatcher we are dealing with a much more heterogeneous, but also smaller multi-field classification problem (see also Table 11). At first, much more header fields (referring to higher dimensions according to the terminology introduced in 2.3.2) are relevant to distinguish the appropriate processing path within the rule bases. However, both the number of rules and the possible set of destinations that have to be differentiated in a typical NP system are significantly smaller than the problem size faced in traditional packet classification, where thousands to ten-thousands of flows have to be identified and managed. In the foreseeable future, I don't expect the number of different processing elements to scale that far in single chip designs.

4.2. The Heterogeneous Decision Graph Algorithm (HDGA)

As I have shown in the previous section, rule bases in the FlexPath NP environment have different constraints than typical five-tuple classifiers found in contemporary router designs. The known five-dimensional classification problem is generalized towards more fields and flags that are all extracted by the Pre-Processor unit. However, none of the rules in the classifier will specify distinct values for all of those fields, instead only between one and four fields are relevant for each individual rule (see also Table 10, Table 11).

Especially due to the heterogeneity of the problem setting, a decision tree algorithm that successively checks individual header fields appears to be beneficial. As values in certain header fields exclude existence of further fields for the individual packet, one can assume to successively partition the rule base at each internal tree node into smaller sets of "eligible" rules until an ultimate resolution is found. Thus, the semantic dependencies between the different fields or flags help to reduce the problem size for subsequent steps. This principle can be expected to work in the most efficient way as long as the individual fields relevant for different networking applications are mutually exclusive. An additional observation concerning the problem setting in FlexPath NP is that in most cases the extracted header fields are only compared to a few distinct values. In traditional firewall applications, a significant share of the entire numerical range represented by these fields has to be regarded. In order to reduce the problem size (with respect to the number of input bits that need to be inspected by the classification algorithm), I propose to perform the comparison on the header fields and define a Boolean variable for the outcome of such an arithmetic operation. This Boolean variable can then be used as an input to a decision graph, instead of using the header fields as in HyperCuts or BDD-based algorithms known from the prior art.

Although the majority of the Path Dispatcher rule base consists of simple checks on a various number of header fields and flags, there are some exceptions to this general observation. One example would be the identification of potential IPsec packets in the traffic, where the rules may require checking the IP source address to a range of predefined addresses (see rule 10 in Table 10). This part requires lots of direct or range match operations on the IP source address field and would not be handled as efficiently in the decision tree structure (see section 2.3.4). Instead, I propose to identify these parts of the rule base, which contain a set of expressions on a unique header field, and evaluate them using a table lookup operation. Depending on the problem size, one can conceive either a direct table lookup, a hash table lookup (see 2.3.1.4) or calling an external search engine (see 2.3.1.5). The result of such a table search (i.e. successful or not) can be used to continue the search in the tree data structure.

Inspired by the contribution of Lysecky (2.3.3, [69]), a manually specified rule base should be minimized before starting to build the classification tree data structure.

Concluding the above paragraphs, I devise a new classification algorithm called HDGA (Heterogeneous Decision Graph Algorithm) with the following characteristics:

- Start with a manually-specified rule base containing all applications, which have to be differentiated in the NP system
- Identify homogeneous parts of the rule base (i.e. many comparisons on a single header field / flag) and mark them for mapping into a table lookup
- Re-formulate the rule base using a Boolean variable notation
- Apply logic minimization to the rule base
- Construct a binary decision tree that is subsequently transformed into a directed acyclic graph. The DAG consumes less memory than the original tree and the classification can be accelerated by introducing quaternary decision nodes.

The resulting decision graph is a data structure that can be efficiently implemented in hardware and can thus achieve a high classification throughput. Results of more complex classification problems than the "typical" FlexPath cases can be tackled with established single or multi-field classifiers (e.g. NSEs) and the results are seamlessly integrated into the Path Dispatcher.

4.2.1. Formulation of Rule Base using Boolean Variables

In order to obtain a briefer representation of the rule base, the individual contributions in the rule base are reformulated with Boolean variables:

- The flags generated by the Pre-Processor (e.g. "Packet Invalid", "TTL Expired", "Own Packet", etc.) and the outcome of (hash) table lookup operations (e.g. "IPsec_network_hit", "Hash(IP5Tuple)_hit") are mapped directly onto such a Boolean value.
- The other contributions can be generally formulated as expressions on header fields with masks and arithmetic operations like equality, inequality, smaller than and greater than (see formalism in chapter 2.3.2, formulas 2-4 through 2-8). A Boolean value can then be assigned to the outcome of such an expression, i.e. when the expression is fulfilled by the header field extracted from the current packet, a 1 or "true" would be assigned, otherwise the Boolean value would remain 0 or "false".

This transformation reduces the bit-width relevant for the later classification problem from the width of the extracted header field, which typically lies in range between

eight and 32 bits, to a single bit. The list of Boolean variables extracted from the example rule base is listed in Table 12.

Table 12: Derivation of Boolean Variables from Expressions in Table 10

Boolean Variable	*Expression*	*Type*
A	Invalid_Flag = 1	Flag
B	TTL_exp_Flag = 1	Flag
C	Own_Flag = 1	Flag
D	ESP_Flag = 1	Flag
E	AH_Flag = 1	Flag
F	Ethertype ≠ 0x0800	Expression on Field
G	DSCP = 2	Expression on Field
H	Lookup_Flag = 1	Flag
I	DSCP = 0	Expression on Field
J	TCP_Ack = 1	Expression on Field
K	IP_Length = 40	Expression on Field
L	Hash(IP5Tuple)_hit = 1	Flag, Hash-Table Result
M	TCP_Flag = 1	Flag
S	IPsec_network_hit = 1	Flag, Hash-Table Result

The reduction in relevant bits by regarding Boolean variables rather than entire header words implies that the Path Dispatcher must contain an arithmetic logic unit (ALU) that computes the Boolean variable out of the header fields extracted by the Pre-Processor. By providing a programmable ALU rather than resorting to a hard-wired implementation, the system as a whole gains a lot of flexibility, such that the architecture may easily be adapted in the field towards supporting new protocol stacks without needing to change the design. Provisioning such an ALU into the classification function is one of the differentiators between HDGA and related classification algorithms described in the prior art.

In addition to providing logic that computes the Boolean variables based on regular expressions on Raw Context fields and flags, the ALU also supports integration of table lookups. Of course, table lookup operations require at least one separate memory access, and can therefore not be implemented in a pure combinatorial way as simple comparisons of Context fields. Consequently, traversal of the decision graph structure has to be halted for the duration of the lookup operation. In return, the classification algorithm has gained the flexibility to seamlessly include full-fledged classification engine results into the architecture.

4.2.2. Matrix Representation of Rule Base and Pre-Processing

The (system-operator specified) rule base is represented by a Matrix R_{manual} with ternary contents true (1), false (0) and don't care (-). The lines or rows of the matrix correspond to the individual rules, while each column represents one of the Boolean variables. The processing path IDs, i.e. numbers representing the processing path associated with each rule, are summarized in vector p_{manual}.

$$R_{manual} = \begin{array}{c} \begin{matrix} A & B & C & D & E & F & G & H & I & J & K & L & M & S \end{matrix} \\ \begin{bmatrix} 1 & - & - & - & - & - & - & - & - & - & - & - & - & - \\ - & 1 & - & - & - & - & - & - & - & - & - & - & - & - \\ - & - & 1 & 1 & - & - & - & - & - & - & - & - & - & - \\ - & - & 1 & - & 1 & - & - & - & - & - & - & - & - & - \\ - & - & 1 & - & - & - & - & - & - & - & - & - & - & - \\ - & - & - & - & - & 1 & - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & 1 & 1 & - & - & - & - & - & - \\ - & - & - & - & - & - & - & 1 & 0 & 1 & 1 & - & - & - \\ - & - & - & - & - & - & - & 1 & 1 & 1 & 1 & - & - & - \\ - & - & - & - & - & - & - & - & - & - & - & - & - & 1 \\ - & - & - & - & - & - & - & - & 1 & - & - & 1 & 1 & - \\ - & - & - & - & - & - & - & - & 0 & - & - & - & - & - \\ - & - & - & - & - & - & - & - & - & - & - & - & - & - \end{bmatrix} \end{array}; \quad p_{manual} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 2 \\ 1 \\ 1 \\ 10 \\ 11 \\ 12 \\ 20 \\ 100 \\ 21 \\ 30 \end{bmatrix}$$

This representation of the rule base differentiates HDGA from prior art classification schemes that all use one or several bits from the header fields in each decision node. By defining Boolean values for comparisons of whole header fields to distinct values, the problem size is reduced from at least 210 bits (concatenation of all relevant header fields and flags) to a range between 10 and 50 bits.

We can see that certain rules have overlapping specifications that are finally resolved by the rule's priority, which is equivalent to its position in the rule base. Thus, evaluation of the rule base is dependent on the order in which the rules are searched with every incoming packet. In order to obtain rules that may be searched in an arbitrary order, I propose the following three steps:

- Make the individual lines in the rule base independent by logical-ANDing the negation of higher priority rules with the current rule specification
- Combine different rules with identical action or processing path ID and apply the rules of Boolean arithmetic to simplify the terms to a sum-of-products form and eliminate irrelevant contributions
- Feed the resulting rule base into a logic minimization tool in order to compress the rule base as far as possible. For the current state of the work I have used the well-known Espresso algorithm obtained from [71].

When executing the above steps, the number of lines in the rule base are first increased from 13 to 27 during priority extension and then reduced to 24 during logic minimization. Finally, every entry in the rule base is now independent from each other and may therefore be evaluated in any order. The resulting pre-processed rule base is reflected in matrix R_{pp} and vector p_{pp}.

$$R_{pp} = \begin{bmatrix}
0 & 0 & 0 & - & - & 0 & 0 & - & 1 & - & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & - & - & 0 & 0 & - & 1 & 0 & - & 1 & 1 & 0 \\
0 & 0 & 0 & - & - & 0 & 0 & 1 & 0 & 1 & 1 & - & - & - \\
0 & 0 & 0 & - & - & 0 & 0 & 1 & 1 & 1 & 1 & - & - & - \\
0 & 0 & 0 & - & - & 0 & - & 0 & 1 & - & - & 1 & 1 & 0 \\
0 & 0 & 0 & - & - & 0 & 0 & - & 1 & - & 0 & - & 0 & 0 \\
0 & 0 & 0 & - & - & 0 & 0 & - & 1 & 0 & - & - & 0 & 0 \\
0 & 0 & 0 & - & - & 0 & 0 & - & 1 & - & 0 & 0 & - & 0 \\
0 & 0 & 0 & - & - & 0 & 0 & - & 1 & 0 & - & 0 & - & 0 \\
0 & 0 & 0 & - & - & 0 & - & 0 & 1 & - & - & - & 0 & 0 \\
0 & 0 & 0 & - & - & 0 & - & 0 & 1 & - & - & 0 & - & 0 \\
0 & 0 & 0 & - & - & 0 & 0 & - & - & - & 0 & - & - & 1 \\
0 & 0 & 0 & - & - & 0 & 0 & - & - & 0 & - & - & - & 1 \\
0 & 0 & 0 & - & - & 0 & 0 & - & 0 & - & 0 & - & - & 0 \\
0 & 0 & 0 & - & - & 0 & 0 & - & 0 & 0 & - & - & - & 0 \\
0 & - & 1 & 0 & 0 & - & - & - & - & - & - & - & - & - \\
0 & 0 & 0 & - & - & 0 & - & 0 & 0 & - & - & - & - & 0 \\
0 & 0 & 0 & - & - & 0 & 1 & 1 & - & - & - & - & - & - \\
0 & 0 & 1 & - & 1 & - & - & - & - & - & - & - & - & - \\
0 & 0 & 1 & 1 & - & - & - & - & - & - & - & - & - & - \\
0 & 0 & 0 & - & - & 0 & - & 0 & - & - & - & - & - & 1 \\
0 & - & 0 & - & - & 1 & - & - & - & - & - & - & - & - \\
1 & - & - & - & - & - & - & - & - & - & - & - & - & - \\
0 & 1 & - & - & - & - & - & - & - & - & - & - & - & -
\end{bmatrix}; \quad p_{pp} = \begin{bmatrix}
100 \\ 100 \\ 11 \\ 12 \\ 100 \\ 30 \\ 30 \\ 30 \\ 30 \\ 30 \\ 30 \\ 20 \\ 20 \\ 21 \\ 21 \\ 1 \\ 21 \\ 10 \\ 2 \\ 2 \\ 20 \\ 1 \\ 0 \\ 1
\end{bmatrix}$$

4.2.3. Construction of a Binary Decision Tree

Using the rule base matrix R_{pp}, a heuristic method is needed to construct an optimized binary decision tree that evaluates the pre-processed rule base.

The basic idea is to check one variable (i.e. evaluation of the requested expression) at a time in each node and then proceed to the left or right child node. In this way, the rule matrix is iteratively split into two sub-matrices (one for each child node) where entries with a zero specification for the inspected Boolean variable would be replicated to the left child, a one specification to the right child and don't care contributions would have to be replicated to both children. This replication - as already mentioned in [67] and [68] - may lead to an exponential blowup of the memory requirements of the decision tree.

In order to determine at each level, which Boolean variable is the best to be inspected next, a greedy selection process is chosen based on metrics that are calculated from the current node's rule matrix and the potential child matrices that would result from splitting the rule base at a certain variable.

In order to achieve a more compact representation, I append the processing path ID vector p_{pp} to the rule matrix R_{pp} and obtain the top-level rule matrix

$$M_{pp} = M_{0,0} = \left[R_{pp} \quad p_{pp}\right] = \left[R_{0,0} \quad p_{0,0}\right] \tag{4-1}$$

In addition, for each distinct path present in the rule base (i.e. distinct members p_i of the vector p_{pp}), we can define a weight factor w_i. The individual w_i factors may be summarized in the weight vector $w_{pp}=w_{0,0}$. To simplify matters, the path weights are fixed to be $w_i=1\ \forall i$ for the following discussion.

For the rule matrix $M_{n,k}$, k=0..2^n-1 of the current iteration level n, we derive the metric $P_{n,k}$, which denotes the weighted number of different paths covered by the rule matrix and $C_{n,k}$, which denotes the weighted number of rule contributions or rows.

$$P_{n,k} = \sum_{w_i \in w_{n,k} \mid w_i \neq w_k, k=0..i-1} w_i \tag{4-2}$$

$$C_{n,k} = \sum_{w_i \in w_{n,k}} w_i \tag{4-3}$$

In our example rule base we obtain $P_{0,0}=10$ and $C_{0,0}=24$ for the initial rule matrix $M_{0,0}$.

For all Boolean variables (BV) in $M_{n,k}$, we determine the potentially resulting child rule matrices $M_{n+1,2k,BV}$ (left child) and $M_{n+1,2k+1,BV}$ (right child) and calculate the number of paths $P_{n+1,2k,BV}$, $P_{n+1,2k+1,BV}$ and contributions $C_{n+1,2k,BV}$, $C_{n+1,2k+1,BV}$ covered in analogy to formulas 4-2 and 4-3.

Regarding splits on Boolean variables A (first column in $M_{0,0}$) and I (ninth column in $M_{0,0}$) in the root matrix we receive after removing the respective column in the child matrices:

$$M_{1,0,A} = \begin{array}{c} \begin{matrix} B & C & D & E & F & G & H & I & J & K & L & M & S \end{matrix} \\ \begin{bmatrix} 0 & 0 & - & - & 0 & 0 & - & 1 & - & 0 & 1 & 1 & 0 & 100 \\ 0 & 0 & - & - & 0 & 0 & - & 1 & 0 & - & 1 & 1 & 0 & 100 \\ 0 & 0 & - & - & 0 & 0 & 1 & 0 & 1 & 1 & - & - & - & 11 \\ 0 & 0 & - & - & 0 & 0 & 1 & 1 & 1 & 1 & - & - & - & 12 \\ 0 & 0 & - & - & 0 & - & 0 & 1 & - & - & 1 & 1 & 0 & 100 \\ 0 & 0 & - & - & 0 & 0 & - & 1 & - & 0 & - & 0 & 0 & 30 \\ 0 & 0 & - & - & 0 & 0 & - & 1 & 0 & - & - & 0 & 0 & 30 \\ 0 & 0 & - & - & 0 & 0 & - & 1 & - & 0 & 0 & - & 0 & 30 \\ 0 & 0 & - & - & 0 & 0 & - & 1 & 0 & - & 0 & - & 0 & 30 \\ 0 & 0 & - & - & 0 & - & 0 & 1 & - & - & - & 0 & 0 & 30 \\ 0 & 0 & - & - & 0 & - & 0 & 1 & - & - & 0 & - & 0 & 30 \\ 0 & 0 & - & - & 0 & 0 & - & - & - & 0 & - & - & 1 & 20 \\ 0 & 0 & - & - & 0 & 0 & - & - & 0 & - & - & - & 1 & 20 \\ 0 & 0 & - & - & 0 & 0 & - & 0 & - & 0 & - & - & 0 & 21 \\ 0 & 0 & - & - & 0 & 0 & - & 0 & 0 & - & - & - & 1 & 21 \\ 0 & 1 & 0 & 0 & - & - & - & - & - & - & - & - & - & 1 \\ 0 & 0 & - & - & 0 & - & 0 & 0 & - & - & - & - & 0 & 21 \\ 0 & 0 & - & - & 0 & 1 & 1 & - & - & - & - & - & - & 10 \\ 0 & 1 & - & 1 & - & - & - & - & - & - & - & - & - & 2 \\ 0 & 1 & 1 & - & - & - & - & - & - & - & - & - & - & 2 \\ 0 & 0 & - & - & 0 & - & 0 & - & - & - & - & - & 1 & 20 \\ - & 0 & - & - & 1 & - & - & - & - & - & - & - & - & 1 \\ 1 & - & - & - & - & - & - & - & - & - & - & - & - & 1 \end{bmatrix} \end{array}; \quad M_{1,1,A} = \begin{array}{c} \begin{matrix} B & C & D & E & F & G & H & I & J & K & L & M & S \end{matrix} \\ \begin{bmatrix} - & - & - & - & - & - & - & - & - & - & - & - & - & 0 \end{bmatrix} \end{array}$$

$P_{1,0,A} = 9;\ C_{1,0,A} = 23;\ P_{1,1,A} = 1;\ C_{1,1,A} = 1$

$$M_{1,0,I} = \begin{array}{c} \begin{matrix} A & B & C & D & E & F & G & H & J & K & L & M & S \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 & - & - & 0 & 0 & 1 & 1 & 1 & - & - & - & 11 \\ 0 & 0 & 0 & - & - & 0 & 0 & - & - & 0 & - & - & 1 & 20 \\ 0 & 0 & 0 & - & - & 0 & 0 & - & 0 & - & - & - & 1 & 20 \\ 0 & 0 & 0 & - & - & 0 & 0 & - & - & 0 & - & - & 0 & 21 \\ 0 & 0 & 0 & - & - & 0 & 0 & - & 0 & - & - & - & 0 & 21 \\ 0 & - & 1 & 0 & 0 & - & - & - & - & - & - & - & - & 1 \\ 0 & 0 & 0 & - & - & 0 & - & 0 & - & - & - & - & 0 & 21 \\ 0 & 0 & 0 & - & - & 0 & 1 & 1 & - & - & - & - & - & 10 \\ 0 & 0 & 1 & - & 1 & - & - & - & - & - & - & - & - & 2 \\ 0 & 0 & 1 & 1 & - & - & - & - & - & - & - & - & - & 2 \\ 0 & 0 & 0 & - & - & 0 & - & 0 & - & - & - & - & 1 & 20 \\ 0 & - & 0 & - & - & 1 & - & - & - & - & - & - & - & 1 \\ 1 & - & - & - & - & - & - & - & - & - & - & - & - & 0 \\ 0 & 1 & - & - & - & - & - & - & - & - & - & - & - & 1 \end{bmatrix} \end{array}; \quad M_{1,1,I} = \begin{array}{c} \begin{matrix} A & B & C & D & E & F & G & H & J & K & L & M & S \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 & - & - & 0 & 0 & - & - & 0 & 1 & 1 & 0 & 100 \\ 0 & 0 & 0 & - & - & 0 & 0 & - & 0 & - & 1 & 1 & 0 & 100 \\ 0 & 0 & 0 & - & - & 0 & 0 & 1 & 1 & 1 & - & - & - & 12 \\ 0 & 0 & 0 & - & - & 0 & - & 0 & - & - & 1 & 1 & 0 & 100 \\ 0 & 0 & 0 & - & - & 0 & 0 & - & - & 0 & - & 0 & 0 & 30 \\ 0 & 0 & 0 & - & - & 0 & 0 & - & 0 & - & - & 0 & 0 & 30 \\ 0 & 0 & 0 & - & - & 0 & 0 & - & - & 0 & 0 & - & 0 & 30 \\ 0 & 0 & 0 & - & - & 0 & 0 & - & 0 & - & 0 & - & 0 & 30 \\ 0 & 0 & 0 & - & - & 0 & - & 0 & - & - & - & 0 & 0 & 30 \\ 0 & 0 & 0 & - & - & 0 & - & 0 & - & - & 0 & - & 0 & 30 \\ 0 & 0 & 0 & - & - & 0 & 0 & - & - & 0 & - & - & 1 & 20 \\ 0 & 0 & 0 & - & - & 0 & 0 & - & 0 & - & - & - & 1 & 20 \\ 0 & 0 & 0 & - & - & 0 & - & 0 & - & - & - & - & 1 & 20 \\ 0 & - & 1 & 0 & 0 & - & - & - & - & - & - & - & - & 1 \\ 0 & 1 & 0 & 0 & - & - & 0 & 1 & 1 & - & - & - & - & 10 \\ 0 & 0 & 1 & - & 1 & - & - & - & - & - & - & - & - & 2 \\ 0 & 0 & 1 & 1 & - & - & - & - & - & - & - & - & - & 2 \\ 0 & 0 & 0 & - & - & 0 & - & 0 & - & - & - & - & 1 & 20 \\ 0 & - & 0 & - & - & 1 & - & - & - & - & - & - & - & 1 \\ 1 & - & - & - & - & - & - & - & - & - & - & - & - & 0 \\ 0 & 1 & - & - & - & - & - & - & - & - & - & - & - & 1 \end{bmatrix} \end{array}$$

$P_{1,0,I} = 7;\ C_{1,0,I} = 14;\ P_{1,1,I} = 8;\ C_{1,1,I} = 20$

Now that we have calculated the $P_{n+1,k,BV}$ and $C_{n+1,k,BV}$ values for all possible child matrices of the subsequent iteration level, we have to decide, which of the Boolean values to use for actually splitting the rule base at level n. In the following, I will derive several contributions for a weighted cost function that can be used to make that decision.

The first optimization target is to avoid an excessive replication of rules to both child nodes, which means trying to avoid splitting on a Boolean variable with many don't care values. In addition, we can focus on replication of rule contributions or replication of different paths. As I have mentioned before, it is not necessary to track the decision down to a single path contribution, the path dispatching task is completed when all remaining contributions point to the same destination in the system. The following two terms can be used as metric for the rule and contribution replication and yield a value of 1.0, in case no replication takes place (desired optimum) and are linearly reducing to 0.5, in case of a full replication (worst case, if the entire column consists of don't care entries):

$$CF^{\alpha}{}_{n,k,BV} = \frac{P_{n,k}}{P_{n+1,2k,BV} + P_{n+1,2k+1,BV}} \tag{4-4}$$

$$CF^{\gamma}{}_{n,k,BV} = \frac{C_{n,k}}{C_{n+1,2k,BV} + C_{n+1,2k+1,BV}} \tag{4-5}$$

A second optimization target is to achieve a well balanced decision tree, which could be achieved when roughly the same number of different path decisions would be found in both child matrices. The benefit of a balanced decision tree is that pathological cases, like a linear search can be avoided, and a more uniform decision time across all possible processing paths may be achieved. The term

$$CF^{\beta}{}_{n,k,BV} = \frac{P_{n+1,2k,BV} + P_{n+1,2k+1,BV} - \left| P_{n+1,2k,BV} - P_{n+1,2k+1,BV} \right|}{P_{n+1,2k,BV} + P_{n+1,2k+1,BV}} \tag{4-6}$$

has its maximum value of 1.0, if the left and right child matrices contain the same number of paths and converges towards zero with increasingly unbalanced splits. A value of zero is achieved, when all paths are represented in a single child and none in the other one. This case would however not lead to a valid decision tree and must therefore be excluded from the tree construction algorithm.

In the following it is important to find out, which of the two optimization criteria formulated above leads to better decision trees with respect to crucial performance metrics such as decision tree size and average and maximum search time. The individual terms presented above are combined into a weighted sum, and trees can be constructed based on the resulting column fitness metric:

$$\begin{aligned} CF_{n,k,BV} &= \alpha \times CF^{\alpha}{}_{n,k,BV} + \beta \times CF^{\beta}{}_{n,k,BV} + \gamma \times CF^{\gamma}{}_{n,k,BV} = \\ &= \alpha \frac{P_{n,k}}{P_{n+1,2k,BV} + P_{n+1,2k+1,BV}} + \beta \frac{P_{n+1,2k,BV} + P_{n+1,2k+1,BV} - \left| P_{n+1,2k,BV} - P_{n+1,2k+1,BV} \right|}{P_{n+1,2k,BV} + P_{n+1,2k+1,BV}} + \gamma \frac{C_{n,k}}{C_{n+1,2k,BV} + C_{n+1,2k+1,BV}} \end{aligned} \tag{4-7}$$

Let's reconsider the situation at the root node with the two columns A and I as described before. For a split at Boolean variable A (i.e. checking the Invalid_Flag from the Pre-Processor), the column fitness contributions are computed as follows:

$$CF_{0,0,A} = \alpha \frac{10}{9+1} + \beta \frac{9+1-|9-1|}{9+1} + \gamma \frac{24}{23+1} = 1 \times \alpha + 0.2 \times \beta + 1 \times \gamma \qquad (4\text{-}8)$$

As we have seen before, there is no replication in the rule base, leading to CF^{α} and CF^{γ} terms of 1 and the asymmetrical splitting leads to a CF^{β} term of only 0.2.

Regarding Boolean variable I, which means comparing the extracted DSCP field against zero, we receive:

$$CF_{0,0,I} = \alpha \frac{10}{7+8} + \beta \frac{7+8-|7-8|}{7+8} + \gamma \frac{24}{14+20} = 0.67 \times \alpha + 0.93 \times \beta + 0.71 \times \gamma \qquad (4\text{-}9)$$

which reflects the more even split between the paths with its CF^{β} metric of 0.93, but the replication is punished with the CF^{α} and CF^{γ} values of 0.67 and 0.71 respectively.

The next step is to find out practical values for α, β and γ, such that the resulting tree best achieves the requested performance criteria, namely compact storage space and low average and worst case search times. As it may be desirable to restrict the computation to integer arithmetic, large integers are used for the weighting factors α, β and γ. As the CF^{α} and CF^{γ} terms are optimizing towards the same criterion and we focus on resolving the processing path, the CF^{α} term should be seen as dominant, with the CF^{γ} term tipping the result into the final direction, if the weighted sum $\alpha CF^{\alpha}+\beta CF^{\beta}$ yields the same column fitness values for several Boolean variables. I have performed a set of simulations for different weighting factors on artificially generated rule bases (details on rule base generation are described in section 4.3), which are shown in Figure 34 and Figure 35. Across all simulations, γ is fixed at 5.

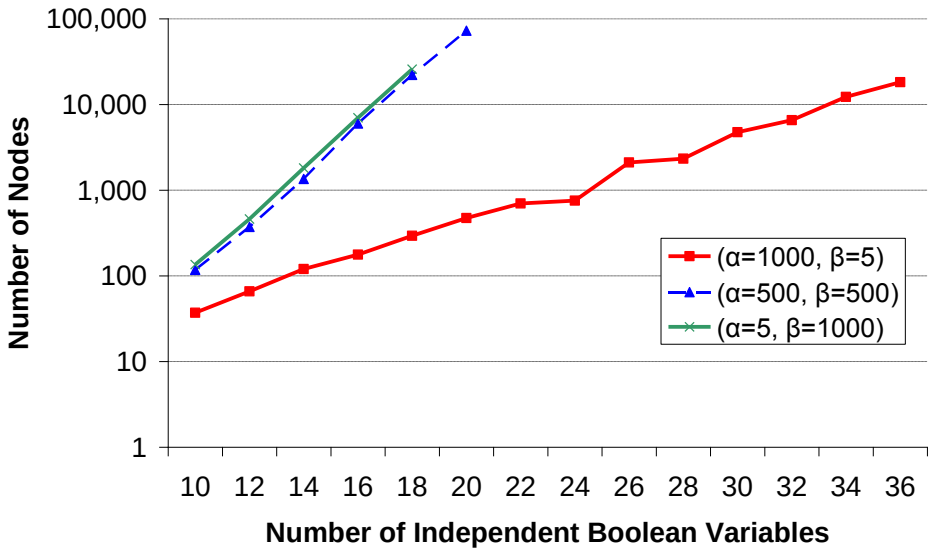


Figure 34: Decision Tree Size for Different α- and β-Weights (γ=5)

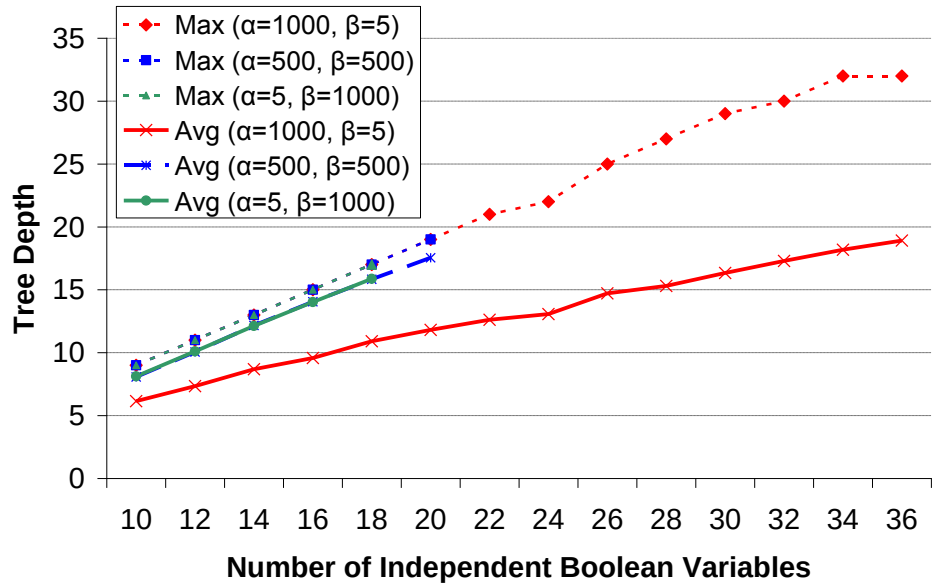


Figure 35: Maximum and Average Decision Tree Depth for Different α- and β-Weights (γ=5)

The three investigated scenarios in the simulations are heavily overweighting CF^{α} (α=1000, β=γ=5), even weighting between CF^{α} and CF^{β} (α=β=500, γ=5) or heavily overweighting CF^{β} (α=γ=5, β=1000). When CF^{α} dominates, decision trees with relatively small memory footprints are obtained, but the depths are quite unevenly distributed. When CF^{β} dominates, the trees are better balanced, but the required storage space grows significantly and the worst case search time (i.e. tree depth) is not improving. Even worse, in addition to the largely inflated storage requirements, also the average tree depth obtained by equally weighting all leaf nodes in the

decision tree is larger than in the CF^{α} dominated trees, and the average depth is only marginally smaller than the worst case depth.

From the above results I conclude that optimizing for minimum replication is the only meaningful approach, as both memory requirements remain within limits and a better average case search time can be achieved. However, CF^{β} should not be completely neglected, as it may still be used to determine a better choice when two columns share the same CF^{α} and CF^{γ} metric, and cases where $CF^{\beta}=0$ have to be excluded. For the following parts of the chapter, I have used the column fitness function with α=1,000, β=5 and γ=5. Weighting of individual paths in the calculations of the split number of paths and contributions has not been used until now, i.e. $w_i=1$ $\forall i$, but could be employed in order to guarantee that certain paths, which might be relevant for a majority of the traffic or belong to extremely critical applications, are evaluated faster than the remaining destinations.

In the example rule base concerned, $CF_{0,0,A}$ has the largest column fitness, so that the root node splits the rule base along Boolean variable A. The matrices and metrics required for the subsequent level of the decision tree are derived as follows:

$$M_{1,0}=M_{1,0,A} \tag{4-10}$$

$$M_{1,1}=M_{1,1,A} \tag{4-11}$$

$$P_{1,0}=P_{1,0,A} \tag{4-12}$$

$$P_{1,1}=P_{1,1,A} \tag{4-13}$$

$$C_{1,0}=C_{1,0,A} \tag{4-14}$$

$$C_{1,1}=C_{1,1,A} \tag{4-15}$$

Construction of the decision tree is continued in an iterative fashion on the child matrices as defined above. In summary, the decision tree can be constructed as follows:

- 1: Obtain root rule matrix as output of rule pre-processing $M_{0,0}$
- 2: Compute $P_{0,0}$ and $C_{0,0}$ according to formulas 4-2 and 4-3
- 3: Iterate over all columns in $M_{n,k}$ except for the last (i.e. inspect all Boolean variables) and derive the two child matrices $M_{n+1,2k,BV}$ and $M_{n+1,2k+1,BV}$ resulting when the current node splits the rule base by inspecting the Boolean variable BV
- 4: For both child matrices and every possible value of BV, compute $P_{n+1,2k,BV}$ and $C_{n+1,2k+1,BV}$
- 5: For every value of BV, compute the column fitness $CF_{n,k,BV}$

– 6: Find the Boolean variable BV_{split}, that yields the highest column fitness

$$BV_{split} : CF_{n,k,BV_{split}} > CF_{n,k,BV} \forall BV \wedge CF^{\beta}{}_{n,k,BV} \neq 0 \quad \text{(4-16)}$$

– 7: Assign the chosen child matrices $M_{n+1,k}$ for the subsequent iterations and compute their $P_{n+1,k}$ and $C_{n+1,k}$ values according to formulas 4-10 through 4-15

– 8: Recursively iterate steps 3 through 7, until the tree is completely built. If a child matrix in any intermediate step contains only a single path ID, the path ID is memorized in the tree as a leaf node instead of the child pointer.

Figure 36 shows the complete resulting binary decision tree constructed from the example rule base. Ovals represent the tree nodes, in which the operation associated with a Boolean variable according to Table 12 is evaluated. The rectangles represent the packet paths or action identifiers that are stored inside the leaf nodes instead of the pointers to further decision tree nodes. The path identifiers are colored differently according to the application-specific processing paths.

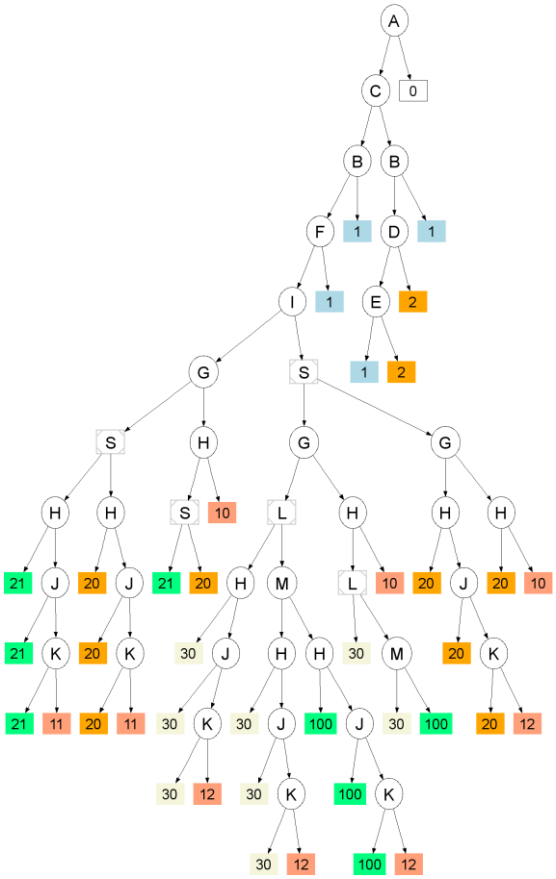


Figure 36: Binary Decision Tree for Example Rule Base

The resulting tree has a maximum depth of 12, i.e. 12 cycles worst case search latency assuming that a memory access and the comparison may be executed within the same cycle. This calculation neglects additional hash table lookup

latencies in the specially marked "L" and "S" nodes. The average depth is 8.58 and the tree consists of 39 nodes. Due to the locally optimal splitting metric (greedy selection algorithm), the order in which the Boolean variables are evaluated may differ in various sub-trees. In addition, it is interesting to realize that the maximum number of inspected Boolean variables is 12, which is two less than the total number of Boolean variables present in the rule base. This effect may be explained by the fact that the Boolean variables' values are not independent for a given packet. The proposed decision tree algorithm helps to evaluate only relevant header fields for each specified path for a distinct application and differentiates HDGA from methods like crossproducting ([60], [63], [65]), where all possible evaluations have to be made before being able to resolve the classification problem.

4.2.4. Transforming the Tree into the HDGA Decision Graph

Although the binary decision tree structure presented in Figure 36 could already be used for rule base evaluation, there is still potential for further optimizations.

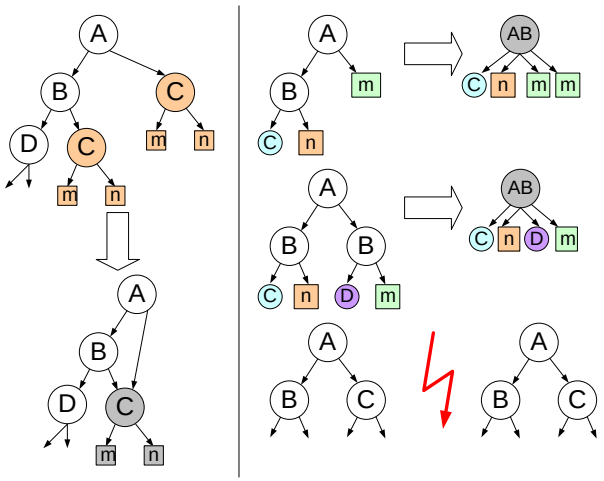


Figure 37: Possible Decision Tree Optimizations: DAG Construction (left) and Quaternary Decision Nodes (right)

Foremost, there exist several isomorphic sub-trees in the decision tree, i.e. both the splitting variable in the nodes and all child nodes and path identifiers are identical. In order to save memory for the search structure, I propose to store these nodes only once and redirect all child pointers of isomorphic sub-trees to the first occurrence in the tree. This optimization, which is also reflected in the left part of Figure 37, transforms the binary search tree into a directed acyclic graph (DAG), which shows some amount of similarity with the binary decision diagrams presented by Prakash et.al. in [66] (see Figure 17 in chapter 2.3.2.6).

In their original form, BDDs are used as canonical forms of representing Boolean functions that perform a mapping from a multi-bit input to a single true/false value symbolized by exactly two terminal nodes in the graph. The requirement of being a function with only a single Boolean value had already been lifted by Prakash in his

routing table algorithm. My decision graph is used to obtain the processing path IDs associated with certain functional elements in our NP design, which can also be symbolized with integer numbers and leads to a larger set of terminal nodes rather than a single Boolean value. However, due to the non-uniform variable ordering, the constructed DAG resembles a free BDD. In addition, the operations at each node in the graph are not based directly on individual header bits, but may also rely on results of whole header field comparisons. This scheme is more effective as long as only a few distinct header values are relevant for larger header fields. By merging the isomorphic sub-trees and constructing the DAG, the number of nodes in our example rule base can be reduced by 7.7% to 36 nodes.

A second optimization is conceivable, which reduces the average and worst case search times and thus facilitates a better scaling towards larger rule bases. By extending the storage space for the individual tree node entry and provisioning additional comparator logic for parallel evaluation of a second Boolean variable, it is possible to execute two decisions within a single clock cycle in a quaternary tree node. However, this will only be effective if the variables in both children of the current node are identical or contain no further splits but resolve the processing path (right part of Figure 37).

I have also made experiments that try to generate a pure quaternary decision tree with a modified column fitness function, which attempts to optimize directly for the two best-fitting Boolean variables. However, these experiments did not deliver competitive results. In contrast, it turned out to be the better choice to construct a binary decision tree with the before presented greedy algorithm, and then try to merge nodes from two adjacent levels in the tree where possible according to the principle shown in Figure 37.

An additional constraint has to be considered with respect to merging nodes that execute hash table lookups rather than arithmetic operations on the Raw Context fields. As I have mentioned before, the hash table lookups need some additional clock cycles for memory accesses and the evaluation of the Boolean variables in the ALU of the Path Dispatcher has to be stalled. In order to simplify the implementation of the Path Dispatcher, it appears to be reasonable to execute hash table operations only in binary decision nodes.

The principle of merging several adjacent binary decision nodes into a quaternary decision node might be extended to even more levels, yielding 8-fold or 16-fold decision nodes. However, the additional cost in the hardware implementation (more comparators and a significantly more complex control and branching logic) is not justified for rare occasions where three or four nodes on consecutive levels in the original decision tree inspect the same Boolean variables. Such a behavior would contradict the properties of the free BDD structure, which is generated by our

heuristic splitting metric. The further parallelization would work in a straightforward way only for the ordered BDDs as they have been used and presented by Prakash et.al. [66]. However, Prakash hinted in his own work, that such an ordered BDD suffers from excessive memory consumption; however his proposed algorithm was not able to construct free BDDs.

An important observation is that although the implementation of quaternary branching nodes comes at a cost of additional comparator and branching logic and larger memory for the quaternary tree node, the total storage space for a given rule base does not change: two binary decision nodes with one Boolean variable specification and two pointers consume as much memory as a single quaternary node with two Boolean variable specifications and four pointers.

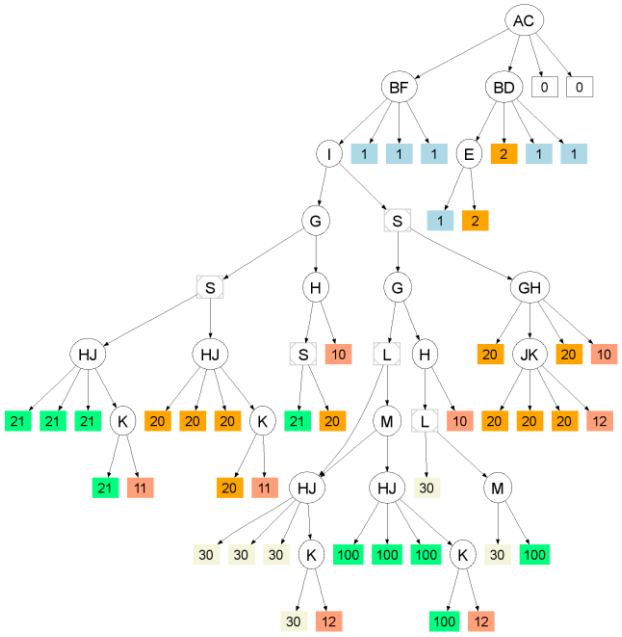


Figure 38: HDGA Decision Graph with Binary and Quaternary Nodes

Figure 38 shows the final HDGA decision graph obtained from the initial binary decision tree by DAG construction through consolidating isomorphic sub-trees and merging suitable binary nodes into quaternary decision nodes. The final decision graph uses only 26 nodes (17 binary and 9 quaternary nodes, consuming the same amount of memory as 35 binary nodes). The maximum depth is reduced to nine levels (-25% compared to the tree of Figure 36) and the average depth is 5.61 levels (-35%).

4.2.5. Updates of the Rule Base and HDGA Data Structures

An important aspect in every classification algorithm is its behavior concerning updates of the rule base during system runtime. As the HDGA data structure

presented in the previous sections consists of several components, I will highlight the updatability of each of them separately.

The decision tree is derived from the pre-processed rule base and the sequence, in which certain packet header fields or flags are inspected, is determined by a greedy selection algorithm that tries to optimize the storage space and search time complexity of the entire search structure. When new rules are added to the rule base that refer to a new protocol class, it is possible that the respective rules are finally mapped into a distinct sub-tree of the decision graph, which can be preconfigured into the tree memory and activated by setting a pointer in the respective parent node in a single atomic operation. However, due to the optimizations that are performed on the initial rule base, such a behavior cannot be guaranteed under all circumstances and it may be necessary to construct a new search graph in a shadow memory and then perform an atomic switch from one graph to another. This would pose a requirement of a sufficiently large memory in the Path Dispatcher implementation to allow holding several configurations in parallel. As additions of new protocols (with additional relevant header fields and thus new Boolean variables) are not very frequent, it might also be possible to assume that such an update could be carried out offline, while the processing elements also need to be supplied with new software code.

Another situation refers to parts of the rule base that consider flow based specifications for certain applications (e.g. the list of currently active IPsec connections) or load balancing. As I have mentioned before, these rule base contributions are mapped to table lookup operations or even external classification engines. In contrast to the before mentioned decision graph, these table contents can be updated easily during system runtime, if the table memories are implemented in dual-port memory technology.

By separating the contributions of the global rule base into quickly changing parts, which are mapped to lookup tables, and quasi-static parts, which are mapped to a decision graph data structure, the system is capable of supporting frequent updates of flow-aware rules (e.g. IPsec or load balancing) without hurting the overall classification throughput.

4.3. HDGA Performance and Scalability

In order to quantify the scaling properties of HDGA and gain objective numbers for comparing it to schemes from the prior art, a set of simulations with randomly generated, i.e. artificial, rule bases of different size has been performed. These rule bases are then used to evaluate the range in which storage requirements and latencies vary. Randomly generated rules will show less statistical dependency than real-world rules, and therefore they exhibit less structure that may be exploited by both the logic minimization and during construction of the decision graph.

The following simulation results show critical performance figures of the proposed HDGA decision graphs for synthetic rule bases with 10 and 20 processing paths. Each rule consists of the conjunction of up to four different Boolean variables in accordance with the observations made in Table 11 (section 4.1). The variables are drawn using independent uniformly random distributed variables out of a set of between 5 and 35 Boolean variables. Therefore, the rules do not reflect the correlation present in real world classifiers and offer less mutually exclusive structure, which can be exploited in the graph. The shares of four-variable to three-variable to two-variable to single-variable rules are 15% to 20% to 35% to 30%. The individual rules are assigning the incoming packets to 10 or 20 different paths, which are again randomly chosen. The resulting figures present average values computed over 100 randomly chosen rule bases for each data point; worst case latencies in Figure 39 reflect the maximum depth of the worst case rule base from the set of 100. In general, the generated decision graphs for the synthetic rule bases tend to become wider and more balanced than the decision graph for the presented real-world example (Figure 38).

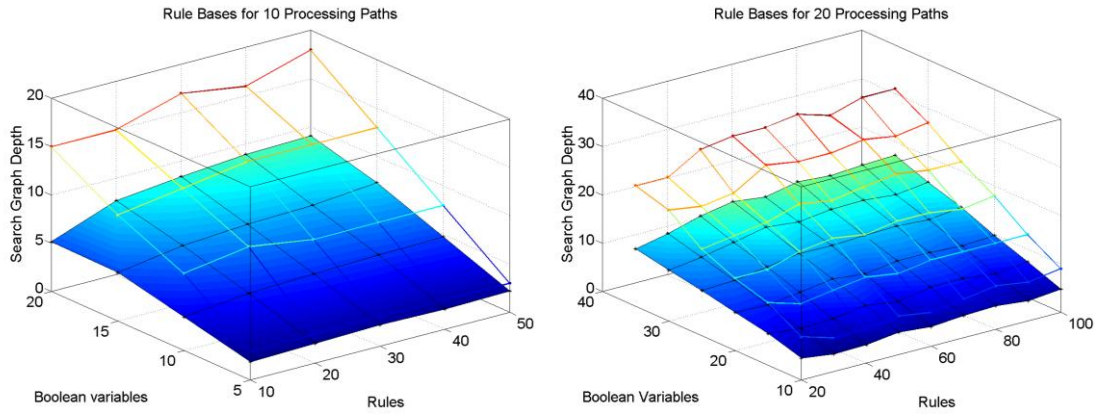


Figure 39: HDGA Average and Worst-Case Search Time Performance

The search time performance figures presented in Figure 39 show that the average decision graph depth (solid surface) is typically half as much as the worst case depth recorded for any of the simulated cases (mesh grid). In addition, the worst

case depth is always less than the maximum number of Boolean variables present in the regarded scenarios.

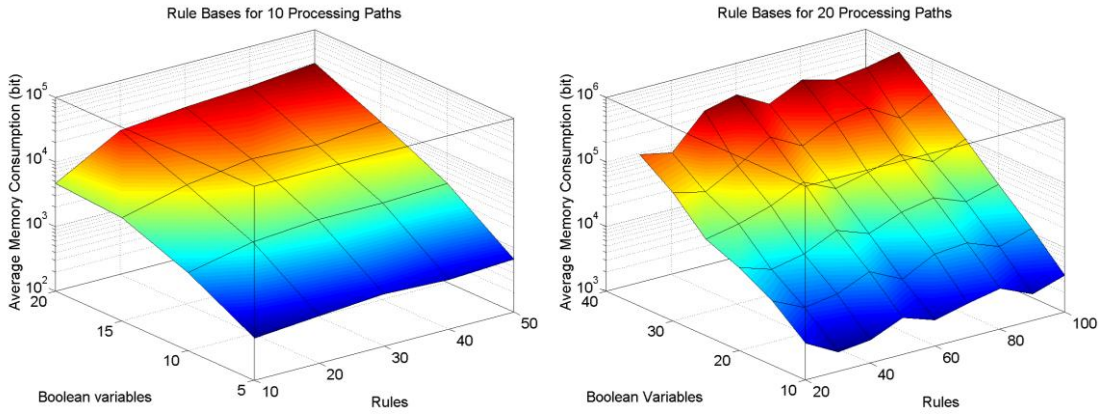


Figure 40: HDGA Average Memory Requirements

With respect to the memory requirements of the HDGA decision graph, an exponential increase can be observed with increasing number of Boolean variables in the rule base (Figure 40). Increasing the number of rules and keeping the number of Boolean variables constant leads to a smaller effect on the storage space requirements. The maximum storage needed for a rule base with 100 rules over 35 Boolean variables is in the order of 750 kbit; this figure corresponds to roughly 3,600 quaternary decision graph nodes and could be mapped into 42 BlockRAM memories of a current Xilinx FPGA.

Figure 41 quantifies the effectiveness of merging isomorphic sub-trees from the initially constructed decision tree and obtaining a DAG. While only about 10% of the memory can be saved for very small rule bases on few Boolean variables, the compression ratio increases to about 40% for rule bases constructed with 20 Boolean variables and to over 50% for rule bases with 35 Boolean variables.

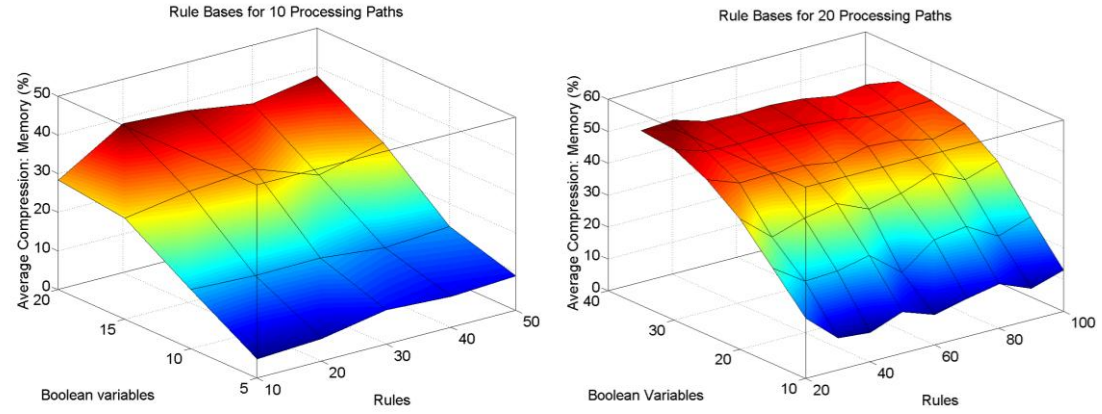


Figure 41: Memory Requirement Reduction by Merging Isomorphic Sub-Trees

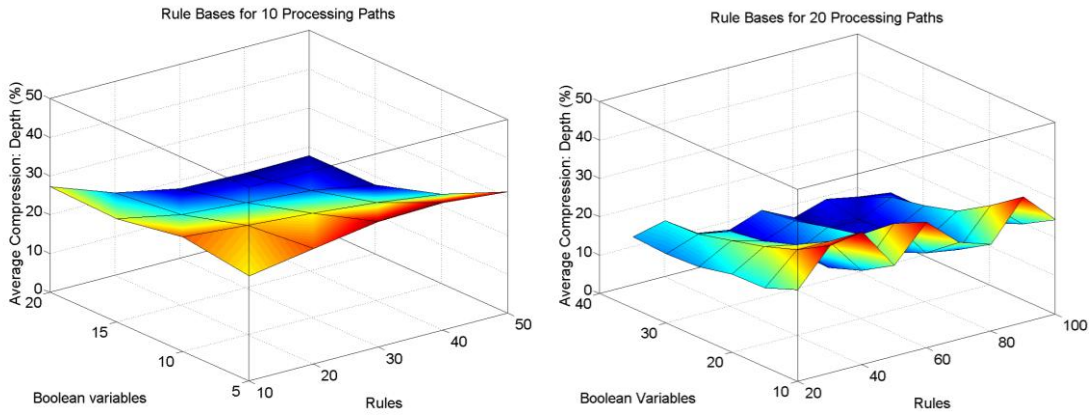


Figure 42: Latency Reduction by Using Quaternary Decision Nodes

An opposite behavior can be observed when analyzing the effect of using quaternary decision nodes in addition to binary nodes to save time (Figure 42). Here, the largest saving effect with roughly 30% can be observed for rule bases with very few Boolean variables, for problem sizes beyond 30 Boolean variables the savings are reduced to less than 20%. Still, 15% fewer cycles on a 30 cycle depth evaluate to four cycles, and four cycles can be an important contribution in deciding whether the resulting decision graph meets real-time requirements in the presented scenario, where packet inter-arrival rates are in the order of a few tens of nanoseconds in the worst case (see Table 6).

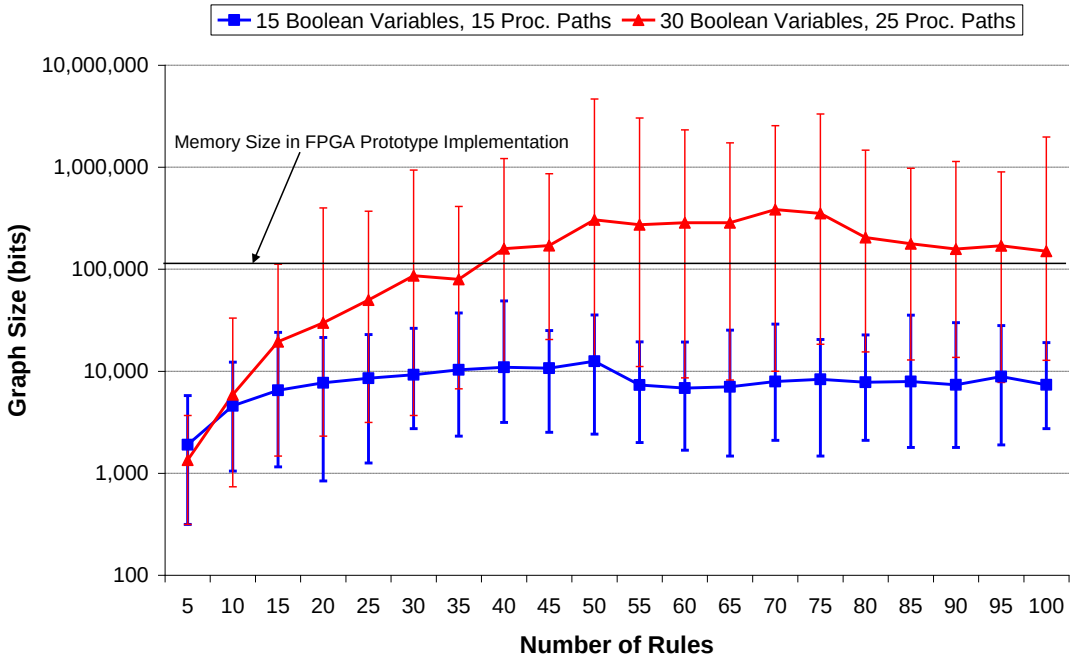


Figure 43: HDGA Decision Graph Size Scaling

Figure 43 focuses on the scaling behavior of HDGA decision graphs with increasing number of rules in the rule base. The first simulation (blue line) refers to a scenario with 15 Boolean variables and 15 processing paths. The resulting size of the

decision graph is growing during addition of the first 50 rules. After this point increasing overlaps in the rule base lead to a saturating effect that eventually reduces the required amount of storage. In addition to the averaged memory requirements of 100 rule bases, the variation range from smallest graph to largest graph is shown. As the presented algorithm is highly data dependent, variations of up to two orders of magnitude can be observed on multiple simulation runs executed with the same input parameter characteristics. Still, the total amount of memory needed is below the (minimally) chosen memory size for the FPGA demonstrator (see also 4.4).

For the larger scenario with 30 Boolean variables and 25 processing paths a similar behavior can be observed. Here, the saturation is reached after about 70 rules in the rule base and due to the more complex classification problem a higher amount of memory is needed.

4.4. Implementation Issues

In the previous sections, I have derived the HDGA packet classification algorithm to solve the packet classification problem faced in the Path Dispatcher. Simulation results have proven the suitability of the chosen algorithm for the given task. Now, an efficient implementation of HDGA is needed. I will start by presenting a straightforward implementation of HDGA. Subsequently, I can show that a few changes in the algorithms' data structures lead to a significantly more area efficient architecture that is able to maintain maximum classification throughput and is adaptable to changes in the Raw Context format by simply modifying the HDGA memory contents. Finally, synthesis results for the Path Dispatcher unit for an FPGA-based demonstrator system (which will be featured in detail in chapter 6) are shown.

4.4.1. Path Dispatcher Interfaces

Before elaborating on the details of an efficient hardware implementation of HDGA, the following figure gives a top-level overview of the Path Dispatcher unit including its external interfaces in the context of our FPGA-based demonstrator platform (see chapter 6).

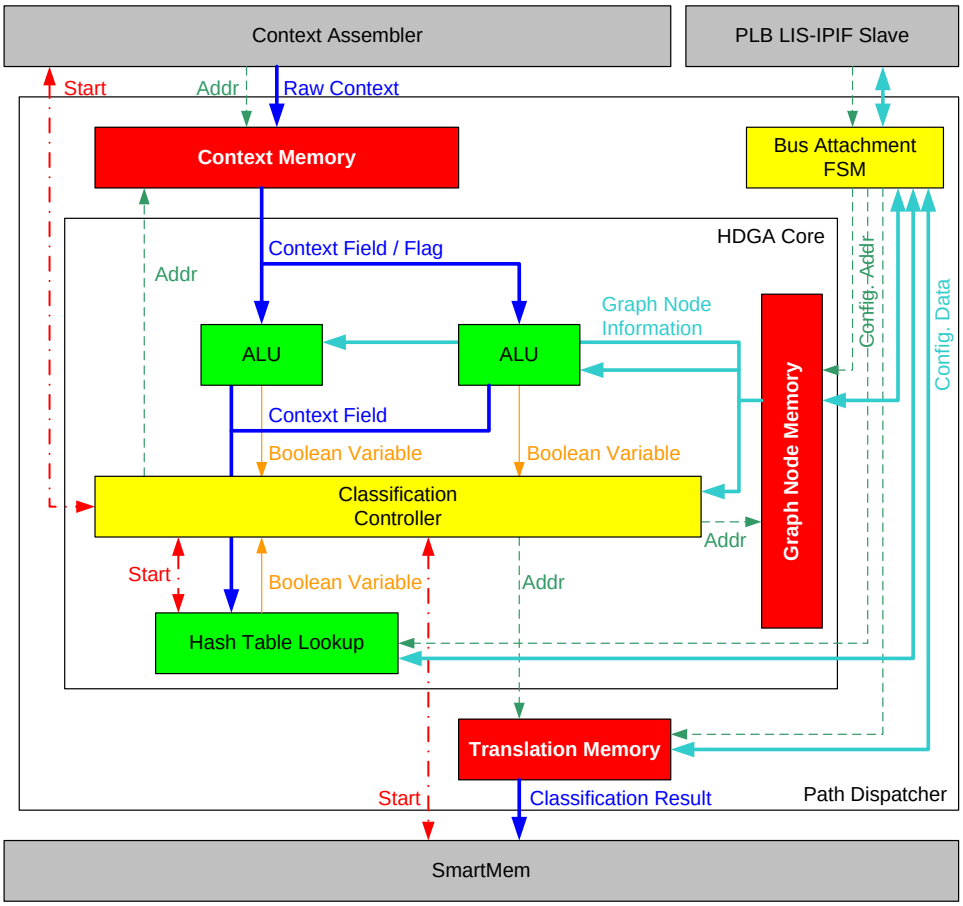


Figure 44: Top-Level Block Diagram of Path Dispatcher

On a very high level of abstraction, the architecture of the Path Dispatcher can be captured as shown in Figure 44. As discussed in the FlexPath concept section (chapter 3.2), the Path Dispatcher implements the packet classification function in the NP ingress data path pipeline. It receives the extracted packet header fields and flags from the Context Assembler unit (described later in chapter 6.2). The final classification result is in turn handed over towards the SmartMem DMA engine and further downstream processing pipeline stages.

The Raw Context is transferred in the data processing pipeline in parallel to the packet data on a 32 bit data path that achieves the same throughput as the packet data. However, while packets may have largely varying lengths (in case of Ethernet this ranges from 64 bytes to 1518 bytes), the context is fixed for our implemented protocol range to 14 words, which is less than the minimum Ethernet frame size of 16 words. Thus, the transmission of packet context is real-time capable for all possible packet arrivals. In addition, the 16 word minimum frame size can also be translated into a 16 clock cycle limit for hard real-time guarantee of the HDGA classification algorithm, i.e. for any arriving packet pattern, real-time constraints are met if the deepest HDGA graph can be worked off within 16 cycles.

The classification algorithm needs to have random access to all context fields and flags, i.e. classification can only be started, when all fields of the current packet have arrived in the Path Dispatcher. This requires at least a double buffering structure (see **Context Memory** in Figure 44), where one context can be received from the Context Assembler, while the second context is used for packet classification. However, if a larger buffer would be implemented, it is also possible to allow a more relaxed timing for average case situations, where classification of a single packet type may take longer than the previously described 16 cycle limit, if such a packet is either followed by a larger packet, or subsequent packets are classified in less than 16 cycles. This would also be interesting in cases, where the classification takes less time than the 14 cycles needed to transfer a single packet context from the Context Assembler to the Path Dispatcher. A larger buffer may be pre-filled and the backlog can be worked off when an arriving packet leads to a shorter than worst-case decision graph. Assuming average case latencies and average packet size in realistic traffic mixes, a good classification performance may still be achieved also for significantly larger problem sizes. The simulation results presented in Figure 35 and Figure 39 suggest that the average-case graph depth in HDGA is on average between 50% and 80% of the worst-case depth.

At the heart of the Path Dispatcher the HDGA classification algorithm has to be implemented. The relevant context words for the current decision graph node have to be transferred to two **ALUs**, which each compute the current Boolean variable by masking the context word and performing a comparison against the predefined value. The Boolean variables are then forwarded to the **Classification Controller**

state machine, which selects the next-level decision graph node or concludes the search process, if the final classification result is resolved. In order to achieve the highest possible classification speed, this calculation has to be performed within a single clock cycle. The **Graph Node Memory**, which contains the HDGA data structure, acts as register stage in an otherwise entirely combinatorial data path. In addition, **Hash Table lookups** can be initiated from the Classification Controller. Hash Table lookups require an additional memory access (apart from accessing the Graph Node Memory) which can not be reasonably assumed to be performed within the same clock cycle as the calculation of the Boolean variables and the resulting child node pointer. In addition, treating the hash table lookup as an additional functional element with its own clocked interface allows to integrate the results of other (possibly off-chip) classification engines for full-fledged five-tuple classification into the Path Dispatcher design.

The decision graph algorithm will at first only provide an ActionID, which determines the further processing in the device. As downstream elements need more precise information like queuing priority, processing latency class, queue ID for the Packet Distributor and information about whether CII or CIO has to be generated for the current packet, an additional lookup in the **Translation Memory** is used to retrieve this kind of information. Storage of a new Raw Context arriving from the Context Assembler and performing the lookup in the Translation Memory plus the handshaking with the SmartMem unit can be performed in separate pipeline stages overlapping the actual HDGA classification task.

The individual processing stages of the ingress NP pipeline use a simple handshaking protocol with Ready / Start signals in order to pass control over the individual packets from one stage to another and provide a backpressure mechanism for accommodating different processing latencies between the individual stages.

4.4.2. Design Space Exploration for HDGA Implementation

4.4.2.1. Constraints on HDGA Graph Evaluation

Before deciding on an implementation of the buffers and the precise structure of the HDGA core, consider the decision graph data structure and the classification process under the assumption of a single clock cycle per decision tree node. A tree structure, as well as the presented decision graph, may be constructed from recursively chaining the individual nodes with data fields as presented in Figure 45.

Binary Node (93 bit):	
CTX_A (4 bit) Mask (32 bit) Value (32 bit) Operation (3 bit)	
Ptr/Action0 (11 bit)	Ptr/Action1 (11 bit)

Quaternary Node (186 bit):			
CTX_A0 (4 bit) Mask_0 (32 bit) Value_0 (32 bit) Operation_0 (3 bit)		CTX_A1 (4 bit) Mask_1 (32 bit) Value_1 (32 bit) Operation_1 (3 bit)	
Ptr/Action00 (11 bit)	Ptr/Action01 (11 bit)	Ptr/Action10 (11 bit)	Ptr/Action11 (11 bit)

Figure 45: Straightforward HDGA Node Contents

The HDGA nodes can be logically separated into two parts:

– The first part contains information about how to compute the Boolean variable within each stage. This includes a reference to the relevant context word (4 bits are needed to address the 14 words of the Raw Context), 32 bit values for defining a mask and comparison value and three bits for selecting the appropriate comparison within the ALU. The ALU itself supports equality, inequality, greater than and less than comparisons, an additional combination is necessary to encode a hash table lookup or invocation of external multi-field classification engines.

– The second part contains the pointers to the two or four child nodes for all possible combinations of the single or two calculated Boolean variables. If the node is a leaf node, the child pointer (which would be NULL in this case) can be used as ActionID consuming the same space as the child pointer. For the prototypic implementation I use 10 bits as pointers or ActionIDs, which allows supporting rule bases with up to 1k nodes. An additional bit is used to differentiate between pointer and ActionID, thus identifying (partial) leaf nodes.

Summing up all the fields mentioned before, we receive a binary node structure of 93 bits and a quaternary node structure of 186 bits. In the final implementation it is beneficial to provision a Graph Node Memory that can hold a quaternary node in a single physical word. At the same time, each word can then also be used to store two binary decision nodes, with an additional bit signaling to the Classification Controller whether the current memory word contains a single quaternary or a pair of binary nodes. The logical addresses used in the child pointers would enumerate binary nodes as elementary elements and the least significant bit is not forwarded to the address lines of the Graph Node Memory. Quaternary nodes have to be assigned to full words, i.e. only even logical addresses.

The main classification routine can now be seen as a combinatorial logic loop, where using the context address field from the node structure one of the fourteen context words is selected. Operation, mask and value are fed into the respective ALU instances and the Boolean outcome of the comparison may be computed. The result now has to be analyzed by the Classification Controller FSM that computes

the address of the subsequent tree node or terminates the classification and initiates the Translation Memory lookup in the next cycle. In case of a hash table lookup instead of a simple comparison operation, the decision graph classification has to be interrupted and the Hash Table Lookup module will be started over the defined interface.

It is the goal of the design space exploration to find an architecture of the Path Dispatcher that achieves maximum HDGA throughput while minimizing the resource consumption in the FPGA environment for the FlexPath NP demonstrator. The basic assumption is that such an efficient architecture would also be a reasonable choice for ASIC implementation, although a standard cell design offers more flexibility especially regarding the availability of custom-sized SRAM blocks.

In order to compare different architectural alternatives, an estimation of the area consumption (for both memory elements and logic) has to be performed. Memory and register sizes can be easily derived, when the dimensions are known by synthesizing a suitable core using the Xilinx CoreGenerator tool. In addition, the ALU, which performs the calculation of the current Boolean variable based on the currently selected Raw Context field and the parameters obtained from the Graph Node Memory, can be easily modeled in VHDL and synthesized using the standard ISE tool chain. The situation is a little bit more complex for the Classification Controller that essentially consists of a large FSM that controls the traversal of the HDGA data structure. Therefore, I have only performed an estimate of the area required for the different multiplexers that split the HDGA data structure into its components and drives the data chunks to the correct functional unit (e.g. the Mask, Value and Operation fields to the ALUs, etc.). The resulting area was determined as 334 slices per ALU, i.e. for a Path Dispatcher with two ALUs (which support evaluating two Boolean variables for a quaternary node in a single cycle) the area estimate is 668 slices. Area requirements for static parts like the LIS-IPIF bus attachment, which is needed for (re-)configuring the Path Dispatcher, the Hash Table Lookup engine and handshaking logic with up- and downstream pipeline elements have been neglected.

Based on the above described area estimates, architectural alternatives will be evaluated in order to obtain the smallest possible solution under the given constraints. The results for all further investigated architectures are summarized in Table 16 on page 143.

4.4.2.2. Path Dispatcher Architecture A

Architecture variant A is based on a straightforward implementation of HDGA based on the tree node structure presented in Figure 45 and the Path Dispatcher interfaces shown in Figure 44.

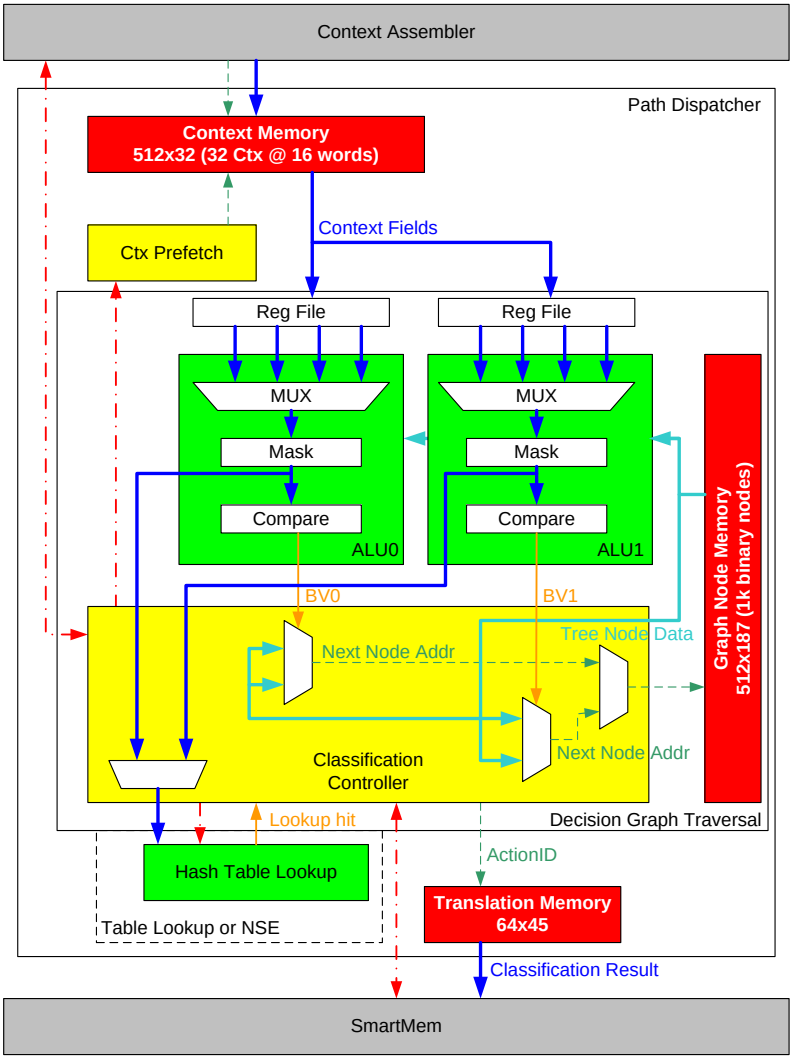


Figure 46: Path Dispatcher - Architecture A

Architecture A1 uses a single BlockRAM instance as Context Memory, which allows storing 32 packet contexts with 16 words per context. While the context is written into the memory from the Context Assembler unit, it has to be fetched into the Register File by a Context Prefetch unit, which has not been included in the area estimation. If the entire packet context of 14 words needs to be copied, each register file will need 448 bits or 224 slices. The Graph Node Memory can be constructed by chaining several BlockRAM instances in parallel in order to obtain the required data width of 187 bits (see Figure 45 and include one bit to distinguish binary and quaternary nodes). This can be achieved with 6 parallel BlockRAMs offering 192 bits using the 512x32 primitive. The Translation memory will be implemented in Distributed Memory technology, i.e. using the LUT resources within

the FPGA as memories, because of its dimensions. The word width of 45 bits would require using two BlockRAM resources in parallel, but the associated depth of 512 entries, i.e. 512 processing paths is beyond the need for the demonstrator implementation. Distributed memory can be parametrized in single bit width increments at multiples of 16 words deep. I have chosen a depth of 64 entries, i.e. supporting a maximum of 64 processing paths as a suitable implementation for the demonstrator. The resulting total area for this solution without the Context Prefetch and Hash Table Lookup units would therefore be 1,388 slices and 7 BlockRAMs, equivalent to 5.5% of the slices and 3.0% of the BlockRAM resources of the used Xilinx Virtex-4 FX 60 FPGA.

Architecture A2 would slightly improve the logic resource consumption by compressing the fields in the Raw Context provided by the Context Assembler, as not every extracted field or flag consumes the full 32 bits of each word (cf. structure of the Raw Context in Figure 88 in the Appendix section). When also ignoring the two fields that contain the result from the next-hop lookup engine; only 210 bits out of the remaining 12 context words are relevant for the supported networking applications. The total area consumption can be reduced to 1,150 slices or 4.5% of the FPGA. While this compression may appear lucrative from the area consumption standpoint, it has to be pointed out that in turn a lot of flexibility is lost with respect to changing the order in which certain header fields and flags are appearing in the context. The larger implementation of architecture A1 would make it possible to replace any context field from the currently used set with another protocol field and the uniform access in 32 bit words would allow to include additional protocols by simply changing the field address in the Graph Node Memory. In other words, a plain control plane update of one configuration memory within the Path Dispatcher unit is sufficient to support in-the-field changes in the supported networking protocols.

Table 13: Area Estimates for Path Dispatcher Architecture A

Architecture	*Unit*	*FPGA Slices*	*FPGA BlockRAMs*
A1	Context Memory	0	1
	Register Files	448	0
	ALUs + CC Multiplexers	668	0
	Graph Node Memory	0	6
	Translation Memory	272	0
	TOTAL	**1,388**	**7**
A2	Context Memory	0	1
	Register Files	210	0
	ALUs + CC Multiplexers	668	0
	Graph Node Memory	0	6
	Translation Memory	272	0
	TOTAL	**1,150**	**7**

4.4.2.3. Path Dispatcher Architecture B

Architecture B1 (see Figure 47) tries to eliminate the two parallel register files by instantiating a wide Context Memory, where the entire packet context can be stored in a single word using distributed memory technology. The multiplexers in the ALU can then access the entire context and the Context Prefetch unit can be eliminated. In turn, the packet context arriving from the Context Assembler has to be converted to the wider data path width by means of an additional serial to parallel converter (SPC), which can be implemented as a single shift register of 512 bits length. Similar as in other units (see chapter 6.2), the Context Memory holds up to 16 packet contexts. As the memory is quite shallow, distributed RAM appears to be a reasonable choice, however the estimated figures show a high area cost of 1,708 slices (6.8%) and 6 BlockRAMs (2.6%).

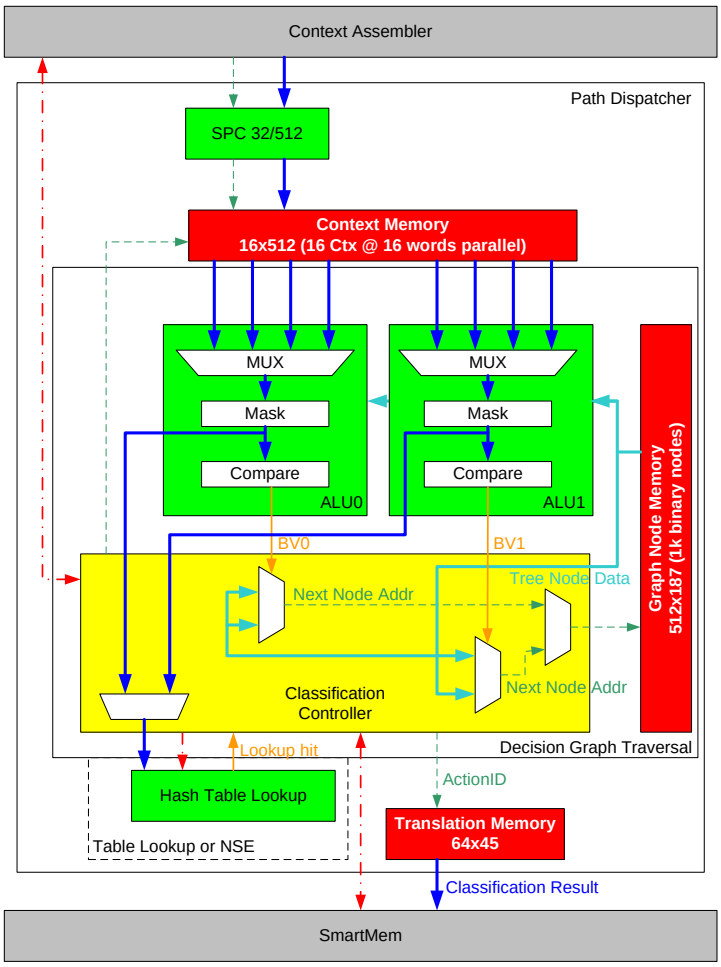


Figure 47: Path Dispatcher - Architecture B

As an alternative, the Context Memory could be implemented using BlockRAM memories that can also be configured with variable read and write data widths. Thus, the need for the SPC could be eliminated by instantiating an asymmetrical BlockRAM core with 32 bit write interface and 512 bit read interface (Alternative B2).

The total system cost estimate evaluates to 940 slices (3.7%) but consumes 22 BlockRAMs (9.5%), which are for the most part only sparsely utilized.

Table 14: Area Estimates for Path Dispatcher Architecture B

Architecture	*Unit*	*FPGA Slices*	*FPGA BlockRAMs*
B1	SPC	256	0
	Context Memory	512	0
	ALUs + CC Multiplexers	668	0
	Graph Node Memory	0	6
	Translation Memory	272	0
	TOTAL	**1,708**	**6**
B2	Context Memory	0	16
	ALUs + CC Multiplexers	668	0
	Graph Node Memory	0	6
	Translation Memory	272	0
	TOTAL	**940**	**22**

4.4.2.4. Path Dispatcher Architecture C

As we have seen, there exists a fundamental tradeoff in the Path Dispatcher architecture exploration between small area consumption (both by means of logic slices and embedded SRAM blocks), generality and extensibility of the design towards future protocols with different context formats and ease of implementation, where complex control and prefetching logic might be eliminated by a simple shift register or a wider memory block. Architecture B2 is the most resource efficient implementation by means of slices and is extensible to rearranged packet contexts in a straightforward fashion, but more than twice the amount of embedded SRAM blocks are needed - measured in percent of the resources offered by the targeted FPGA device - than for the remaining logic. In order to work off a decision graph node in a single clock cycle, it is necessary to be able to access the entire range of context words in a parallel fashion. As we have seen before, this can be achieved by using either a set of registers or a wide memory.

Still, as we need at most two different context fields in every classification cycle, it might be more efficient, if we could read the required words directly out of a (narrow) memory block. This can be achieved with high performance, if the format of the decision graph nodes is rearranged as shown in Figure 48.

Essentially, the address of the context word involved in computing the Boolean variable is moved from the node part into the pointer part of the graph node data structure. In addition to specifying the pointers, i.e. address of the subsequent node, the address of the upcoming context word is specified for both possible outcomes (0 or 1) for the binary node. With this information, the address may be routed to the

Context Memory at the same time as the Graph Node Memory address, and both the mask/value information as well as the correct context field is available in the subsequent clock cycle. As it is necessary to include two addresses for binary nodes and four addresses for the quaternary nodes, the size of the graph node entry rises to 105 or 210 bits respectively. The resulting 211 bit wide Graph Node Memory can however still be implemented with a chain of six BlockRAM memories, when selecting the 512x36 primitive and using the parity check bits as additional data bits.

Binary Node (105 bit):	
Mask (32 bit) Value (32 bit) Operation (3 bit)	
Ptr/Action0 CTX_A0_0 CTX_A0_1	Ptr/Action1 CTX_A1_0 CTX_A1_1

Quaternary Node (210 bit):			
Mask_0 (32 bit) Value_0 (32 bit) Operation_0 (3 bit)		Mask_1 (32 bit) Value_1 (32 bit) Operation_1 (3 bit)	
Ptr/Action00 CTX_A00_0 CTX_A00_1	Ptr/Action01 CTX_A01_0 CTX_A01_1	Ptr/Action10 CTX_A10_0 CTX_A10_1	Ptr/Action11 CTX_A11_0 CTX_A11_1

Figure 48: Optimized HDGA Node Contents

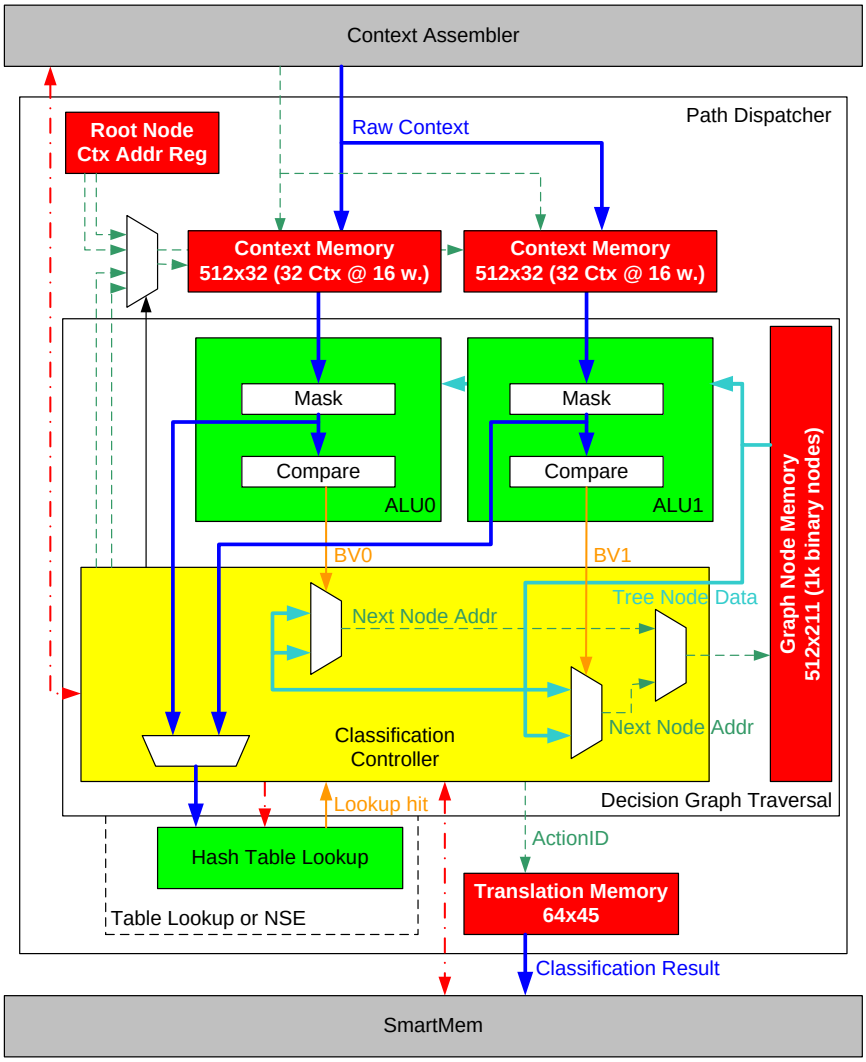


Figure 49: Path Dispatcher - Architecture C

As shown in Figure 49, the Context Memory can now be implemented with two parallel instances of a single BlockRAM memory that allows storing 32 packet contexts with 16 words each or 16 contexts (as used in other processing pipeline elements) and up to 32 words. There is no need for additional prefetching logic and the next context word addresses are provided by the Classification Controller out of the Graph Node data structure. The multiplexers previously required in the ALUs can now be saved. An additional set of registers is necessary to define the context fields required for evaluating the root node; this initial access is performed in parallel to fetching the root node information from the Graph Node Memory. The total area estimate is now reduced to only 872 slices (3.4%) and 8 BlockRAMs (3.4%), so Architecture C is also a very balanced solution. In addition to being the smallest possible solution, a lot of freedom is retained that allows easily changing contents and formats of the packet context without needing to redesign the Path Dispatcher.

Table 15: Area Estimates for Path Dispatcher Architecture C

Architecture	*Unit*	*FPGA Slices*	*FPGA BlockRAMs*
C	Context Memory	0	2
	ALUs + CC Multiplexers	600	0
	Graph Node Memory	0	6
	Translation Memory	272	0
	TOTAL	**872**	**8**

Table 16 summarizes the estimates of the resource utilization for the different presented architectural alternatives for the HDGA graph evaluation logic.

Table 16: Estimated Resource Requirements of Various Architecture Alternatives

Architecture	*FPGA Slices*	*FPGA BlockRAMs*	*Share of slices*	*Share of BRAMs*
A1	1,388	7	5.5%	3.0%
A2	1,150	7	4.5%	3.0%
B1	1,708	6	6.8%	2.6%
B2	940	22	3.7%	9.5%
C	872	8	3.4%	3.4%

Architecture C is finally chosen to be implemented in the FPGA prototype as it achieves the desired functionality with the least amount of resources and provides a good balance between logic elements and memory primitives.

4.4.2.5. Hash Table Lookup

As described in chapter 4.2, there are two cases for which hash table lookups are needed in HDGA. The first application is distribution of stateful processing loads among several parallel processor entities, which is discussed in further detail in chapter 5.2. Here, packets are assigned to specific processors by performing a lookup using a hash value of the packet's Internet five-tuple. A complete list with all possible hash values must be maintained along with the corresponding processing path assignment. The second application is matching a certain header field against a larger set of distinct values, which is too large to scale efficiently in the decision tree structure. Here, a hash table lookup with a possible collision resolution scheme is required and performs significantly better than working off an exponentially sized decision graph over several clock cycles.

Based on the previous observations, a simple generic table lookup engine can be implemented in a straightforward fashion that allows performing direct and hash table lookups with an optional collision resolution scheme. As the table lookups involve accessing at least the additional table memory, and an asynchronous memory access would significantly deteriorate the length of the combinatorial path delay in the HDGA decision graph traversal block (see Figure 49), the table lookup will be included in a separate entity with a defined synchronous handshaking protocol interface. In addition, it is then also possible to replace the table lookup function as implemented in the Path Dispatcher prototype with any other external classification engine like e.g. an NSE.

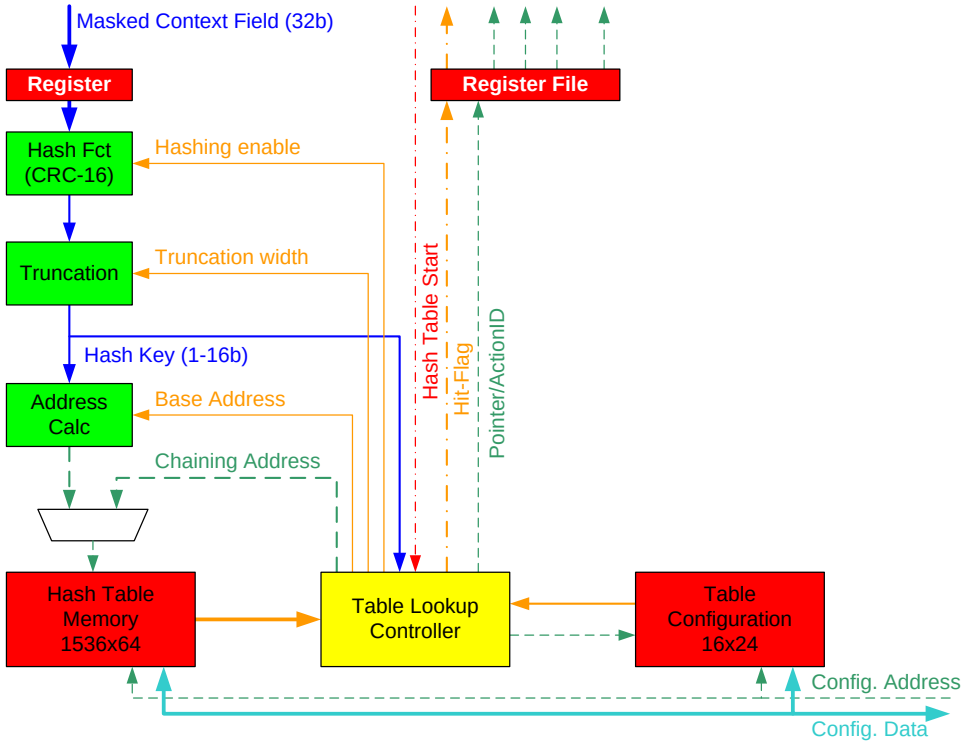


Figure 50: Block Diagram of Table Lookup Unit

When the Classification Controller of the Decision Graph Traversal unit detects the table lookup Opcode in the current graph node, the (masked) 32 bit wide context field is forwarded from the ALU to the Table Lookup Unit, and the lookup is initiated by asserting the HashTable Start signal. The table lookup module supports several tables in parallel, so important information like whether the requested lookup is direct or hashing based, the key width (which is directly related to the logical address width of the lookup table) and the base address of the lookup table in the physical table memory is obtained from a configuration memory. In the subsequent clock cycle, the masked field may be hashed, using a 16 bit CRC function, is truncated to the specified key length and the initial lookup address is calculated by adding the key to the table base address. If a direct lookup or a hash table lookup without collision resolution is requested, the lookup result can be communicated back to the Classification Controller in the subsequent cycle. If a hash table lookup with collision resolution was necessary, the original context field is compared to the value stored in the hash table. If they correspond, the result may also be communicated to the Classification Controller. If they don't match, a simple chaining mechanism is provided in the table lookup unit, i.e. the Table Lookup Controller obtains the chaining pointer from the hash table entry and performs another lookup in the following cycle. This may be continued until either the correct key entry was found or the end of the chain of entries is reached.

4.4.3. FPGA Implementation Results

The Path Dispatcher has been implemented in the way derived above on our Virtex-4 FX60-based FPGA development platform, which will be discussed in detail in chapter 6.1. In the following, I would like to briefly highlight the final synthesis results for the Path Dispatcher as a standalone element; cumulative figures for the entire prototype platform are deferred to chapter 6. Table 17 lists the synthesis results of the Path Dispatcher implementation according to Architecture C and including the Hash Table Lookup module as described in 4.4.2.5. Figures for the LIS-IPIF [99] needed in the final system as PLB bus master attachment are excluded. However, the logic in the synthesized core includes the bus attachment FSM implementing the LIS-IPIC control signals between the Path Dispatcher core and the LIS-IPIF slave.

Table 17: Stand-alone FPGA Synthesis Results for the Path Dispatcher

Resource Type	*Resource Quantity*
FPGA Slices	1,368 of 25,280 (5.41%)
Slice Flip-Flops	368 of 50,560 (0.73%)
Slice LUTs	2,450 of 50,560 (4.85%)
FPGA BlockRAM memories	16 of 232 (6.90%)
Critical Path	8.971 ns (i.e. 111.473 MHz)

Concerning real-time capabilities of the current implementation, the following timing behavior has to be considered:

– If the rule base can be mapped exclusively on the decision graph structure of HDGA, the classification within the ingress processing path pipeline is real-time capable, if the maximum depth of the graph is 15. As already discussed in section 4.4.1, the shortest inter-arrival time between two consecutive packets is 16 cycles, and one cycle is necessary for accessing the first Raw Context word and HDGA root node information, before the actual graph traversal starts.

– The currently implemented Hash Table Lookup engine consumes three cycles in non collision-resoluted operation, which reduces the maximum graph depth by two additional cycles for each hash table access. The implemented chaining mechanism increases the consumed cycles by one for every additional collision.

As I have shown in section 4.3, a HDGA tree depth of 10 to 15 nodes is already enough for a significant range of relevant scenarios. However, depending on the actually configured rule base, the decision graph may become deeper. In order to tackle the problem of deeper graphs or to enable rule bases with several table lookup operations, it is possible to introduce pipelining into the architecture of the Path Dispatcher.

A pipelined version of the Path Dispatcher would need to replicate Context Memories, ALUs, Classification Controllers and Graph Node Memories. The first stage would remain unchanged from the current implementation and traverses the first 15 cycles of the HDGA decision graph. In its last cycle, the root node of the next-stage HDGA graph would have to be communicated to the subsequent pipeline stage along with the next context memory pointers. Starting with this information, the subsequent pipeline stage may work off the remainder of the graph. By provisioning two pipeline stages, the maximum depth constraint can be raised to 30, which is sufficiently large to work off all problem sizes investigated in the context of this thesis as shown in Figure 39.

4.5. Conclusions

In the present chapter, I have introduced HDGA as a new, modular packet classification algorithm tailored for the specific environment faced in the path dispatching problem within a FlexPath NP. However, the classification scheme may also be easily applied to more general on-chip path selection or task assignment functions relevant in modern multi-processor SoCs designs. HDGA is a hybrid approach that combines a decision graph classifier with table lookups. Various optimization goals for constructing the decision graph have been proposed and evaluated. Heterogeneous parts of the rule base are dealt with in the graph structure, which is constructed optimizing for compact implementation and short average search times. Homogeneous and potentially quickly changing parts of the rule base are mapped to table lookups (which may be either direct table lookups or hash table lookups depending on the actual situation) or other specialized classifiers, e.g. off-chip TCAM-based NSEs.

Before the decision graph is constructed, the classification rule base is minimized using techniques from logic synthesis and isomorphic sub-trees are merged into a single instance in order to save memory without hurting lookup performance. In order to further accelerate the classification process, binary decision nodes from the graph are merged into quaternary decision nodes where possible.

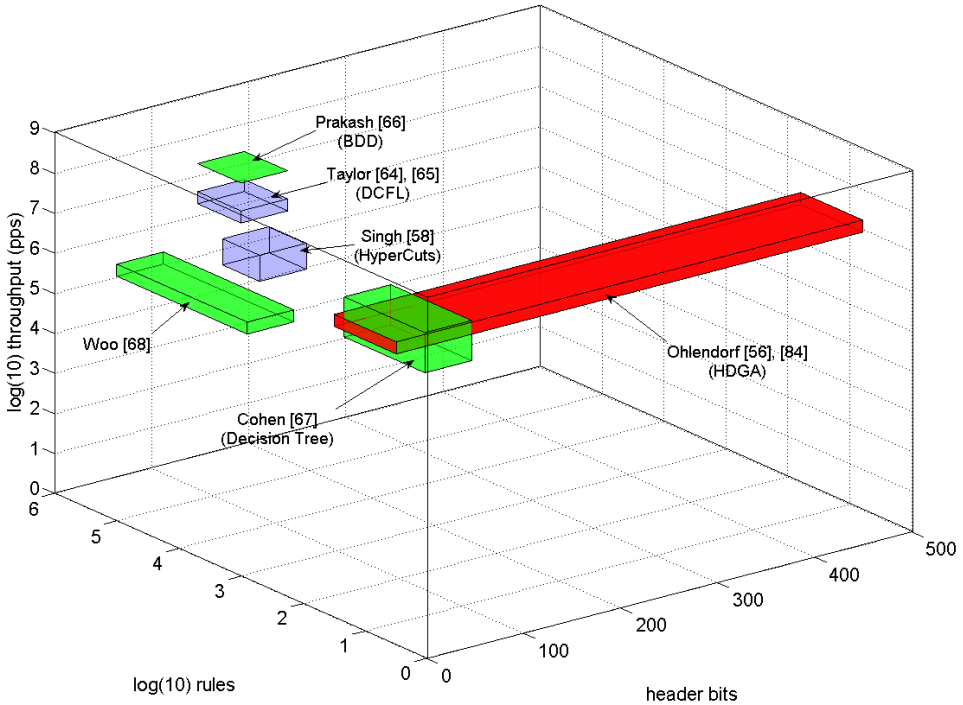


Figure 51: Throughput Performance of HDGA vs. Several Prior Art Schemes

Throughput and storage requirements of the proposed HDGA classification scheme (obtained from simulations described in 4.3) are compared to published

performance figures quoted for classification schemes from the prior art in Figure 51 and Figure 52.

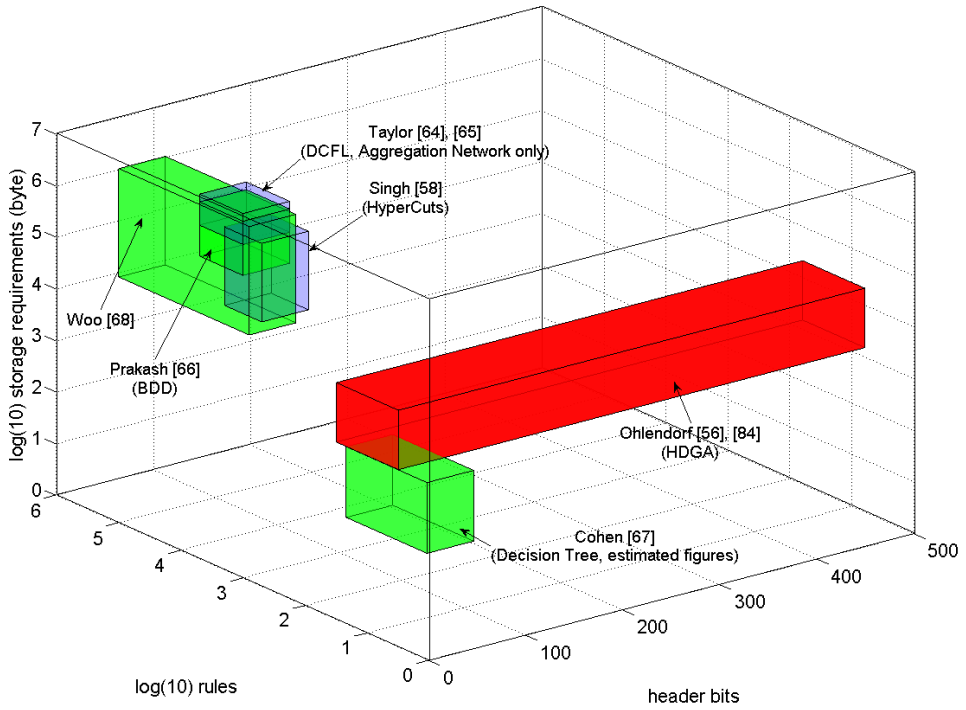


Figure 52: Storage Requirements of HDGA vs. Several Prior Art Schemes

The published performance figures by Woo [68] refer to a software implementation, while Taylor [65] and Prakash [66] describe an ASIC concept; thus these figures are hard to compare directly to our FPGA targeted implementation. Cohen [67] only states performance figures by means of tree nodes and memory accesses, so I had to estimate the performance by means of bytes and packets per second for inclusion into Figure 51 and Figure 52. In accordance with the derivation of tree node sizes by Woo in [68], I favorably assumed one node to consume only 10 bits, and considered a range of 10 ns to 30 ns for a memory access.

5. Multi-Processor Load Balancing in FlexPath NP

5.1. Introduction

The FlexPath NP I have derived in chapter 3.2 features a network processor complex with parallel CPUs or processing elements (PE). Parallelization of the incoming packets onto several processing entities is necessary in order to meet the processing requirements imposed by the networking applications on the network processing infrastructure. A fundamental problem faced by every parallel compute architecture is the problem of load balancing, which will be investigated in the context of FlexPath NP in the subsequent sections.

The problem of load balancing is not new, and several schemes that deal with the load balancing problem in the context of network processing have been discussed in chapter 2.4. However, a FlexPath NP features a Pre-Processor and Path Dispatcher in an ingress processing pipeline in order to provide different processing paths for various networking applications. These capabilities, which allow different treatment for packets of different traffic classes, should in the following be used for load balancing. After all, solving the load balancing problem is effectively making a decision on the further processing path of the arriving packet. Therefore, the Path Dispatcher of a FlexPath NP is the straightforward instance, onto which the load balancing function should be mapped.

The Path Control, which is covered in detail in Michael Meitinger's dissertation [107], solves the problem of packet reordering in FlexPath, which has also been given an important focus in the prior art schemes for NP load balancing. Therefore, we have investigated how the specific functional enhancements in the NP system can be exploited to further optimize the NP system performance with respect to satisfying QoS requirements and maximizing the individual processing element utilization.

Our approach to the load balancing problem is driven by the following question: What would be the optimum load balancing strategies with respect to overall system utilization, minimum packet loss rates and processing latencies when considering a heterogeneous application mix with different QoS requirements?

As the different application classes can be identified within the FlexPath ingress processing pipeline, it is possible to apply different load balancing strategies for different application types. This can be seen as a straightforward utilization of the functionality offered by the proposed network processor architecture, but such a strategy has not been investigated by other researchers before. In accordance with the findings in chapter 2.2.7, we have focused our efforts on stateful and stateless network processing applications and use QoS-aware IP forwarding (see chapters 2.2.1 and 2.2.2) and IPsec encryption (see chapter 2.2.3) as representative examples

for the more general application classes. After identifying load balancing techniques that are well suited for each individual application class, we propose a combination of the two most promising techniques in systems that process a mix of stateful and stateless traffic classes. The most important concepts and results described in detail in the following parts have already been published in [72].

Section 5.2 presents the individual load balancing techniques for stateless and stateful network processing applications. In addition, a combination of two specific techniques is proposed for system scenarios that process different application classes at the same time. Section 5.3 evaluates the performance of the proposed load balancing techniques and compares them to the performance of prior art solutions by functional simulation of a parallel processor cluster NP architecture. Finally, the chapter is concluded in section 5.4.

5.2. Load Balancing Strategies for Different Application Classes

5.2.1. Stateless Network Processing Applications

In the following discussions, we distinguish two different traffic types within the class of stateless networking applications: best effort traffic (referred to as BE in the following), which is the bulk of Internet traffic being forwarded without any QoS provisions or guarantees, and DiffServ high priority traffic (referred to as QoS in the following) where a DSCP other than zero defines an application-dependent per-hop forwarding behavior that has to be applied to the respective packet stream. In our example, we will simply provide a higher processing and output port queuing priority to such packets.

We propose to use a slightly modified form of packet spraying (definition see chapter 2.4.2) for all stateless traffic classes. In order to implement the requested QoS behavior, it is possible to provide separate queues for each DSCP value and provision separate queue servicing schemes in order to achieve the requested per-hop forwarding behavior in the NP. The idea behind implementing a packet spraying approach is that the packets will experience optimum processing by the PE cluster as we can exploit a pooling gain from distributing the packets over a multitude of PEs. In contrast to the spraying mechanism as described by Dittmann in [73], we do not maintain a single queue in front of each processor, into which the packets are sprayed. The spraying is performed in the Packet Distributor (details see Michael Meitinger's dissertation [107], chapter 5) out of a single queue per traffic class, which is shared among a configurable set of PEs that are responsible for processing this traffic. As long as a packet sits in the queue, an interrupt will be forwarded to all PEs associated with the respective traffic type. The interrupt priority for the QoS queue will be higher than that for the BE traffic. When a PE is busy with processing a packet, it will mask all its interrupts, so that only idling processors will react to the interrupts.

As a consequence of the statistical distribution of packets a well-balanced distribution of the load among all involved PEs can be expected, and each arriving packet will experience the shortest possible waiting time until it gets serviced. The modified spraying technique avoids head-of-line blocking effects associated with queues that are dedicated for individual PEs and also reduces packet reordering probabilities in comparison to Dittmann's spraying, as packets may only experience varying processing latencies, but not different queuing delays. Of course, it is not guaranteed that packets from the same traffic flow are processed by the same processor, which is acceptable as long as no shared state information is required for packet processing. The resulting higher packet reordering rates in comparison to

the hashing-based dedicated flow assignments are eliminated in a FlexPath NP by the Path Control unit before the packets reach the output buffers of the NP.

5.2.2. Stateful Network Processing Applications

While packet spraying is a good solution for the stateless network processing applications, it is ill-suited for stateful processing due to the consistency and performance implications arising when state information is distributed among several parallel processing entities. Stateful flows are more efficiently processed on a single dedicated PE that can hold a local copy of the required processing state. In case that the aggregate of flows that are assigned to a single PE exceeds the overall processing capacity, rebalancings have to be performed with a possibly costly state information migration among the involved PEs. The class of adaptive hashing-based load balancing schemes (AHH, see chapter 2.4.3 and [74] or HABS, see chapter 2.4.5 and [77]) presented in the prior art section appear to be suitable candidates for this type of traffic.

I have analyzed the behavior of these two schemes by means of our functional NP simulation framework (see chapter 5.3.1 for details) and came up with the following conclusions:

- Implementation and evaluation of the highest random weight (HRW) scheme in AHH is quite computationally intensive, especially as the hash function has to be computed N times in a system with N PEs. In addition, the maximum of the N weighted hash values has to be determined (see formula (2-11) in section 2.4.3). These calculations cause a significant processing burden for larger processor clusters. In addition, as weights are adapted in order to reduce the load from excessively loaded PEs, the algorithm guarantees a minimum disruption property, i.e. only few flows are shifted, but it is not guaranteed that the shifted flows are migrated towards the least-loaded PE in the system. In contrast, the flows are randomly spread among the remaining processors, with a weighting according to the relative load of all PEs. I have also observed that when the algorithm is exposed to a system state near the total system capacity that successive adaptations may lead to oscillating flow assignments between the same pair of PEs. As the load in the system is reduced, the AHH algorithm stops adaptation, which is beneficial from a standpoint of keeping flows where they are, but the uneven load observed on different PEs in the processor cluster leads to varying queuing latencies for different flows belonging to the same application class. In addition, for short-lived bursts, which are a commonly observed phenomenon in Internet traffic, the highest-loaded CPUs are shortly moved into overload with possible packet loss in the respective queue. This happens although other PEs in the cluster still have sufficient processing resources available.

- For HABS, the implementation effort is even higher. In addition to the computations associated with the HRW algorithm in AHH, a flow table has to be maintained that keeps track of the set of currently active flows in the NP. With

every packet arrival a counter in the table has to be incremented and it is reduced with every packet departure. If the flow table is fully populated, no further burst shifting may be performed. While the authors state in [77] that up to 300,000 flows are active in one of their investigated traffic traces, the flow table is dimensioned to hold only 200 entries. Although the authors make no comment concerning possible implementation, such a table might be implemented as a hash table with collision resolution just as described in the context of the Path Dispatcher in chapter 4.4.2.5. Another problem with the scheme as it is described in [77] can be identified with respect to packet loss in the system. Whenever an NP system is driven near the performance limit, some packets may be lost due to temporal overloads on single PEs, or flows from lower priority traffic classes might be willfully discarded in order to guarantee the QoS for higher-priority packets. Here, additional efforts are necessary, e.g. timeouts or a flow aging mechanism, in order to avoid blocking the flow table by packet arrivals that never depart from the system again.

In order to minimize the effort spent for load balancing, while maintaining a close to optimal PE resource utilization, I propose a new, simple, adaptive, hashing based load balancing scheme referred to as HLU (hash lookup). The following sections describe the load assignment process of HLU that has to be performed within the control plane CPU of the NP. The resulting flow to PE assignment can be easily configured into the Path Dispatcher rule base by performing a hash table lookup without collision resolution with the IP five-tuple hash computed by the Pre-Processor.

At system startup, an initial assignment is performed for all possible flows (distinguished by a hash value that is computed from the Internet five-tuple, called FlowID in the following) to the individual PEs in the processor cluster or at least a subset of these. A FIFO list is maintained in the control plane software for each processor that stores all FlowIDs that are currently assigned to the respective PE (see Figure 53). From these FIFOs, the hash table entries for the Path Dispatcher can be easily constructed. Initially, the FIFOs will be filled with an equal amount of flows. It is known from previous publications (especially [74]) that this assignment is not optimal due to a bias in the hash value distribution for realistic Internet traffic.

During system runtime, the load of the individual PEs is measured and a load adaptation that shifts FlowID assignments away from the heaviest-loaded PE to the least-loaded PE is performed when an unbalanced load situation is observed. In contrast to the schemes of the prior art presented in chapter 2.4, we do not rely on queue length as indicators for processor load; instead the load is measured directly on the respective processors. This can be achieved for CPUs by inserting two additional instructions in the processing code that inform a set of hardware counters of beginning and end of the processing routine for each arriving packet. Two

different counters can be provisioned for each CPU, such that it is also possible to record the load contributions of dedicated and sprayed traffic classes separately.

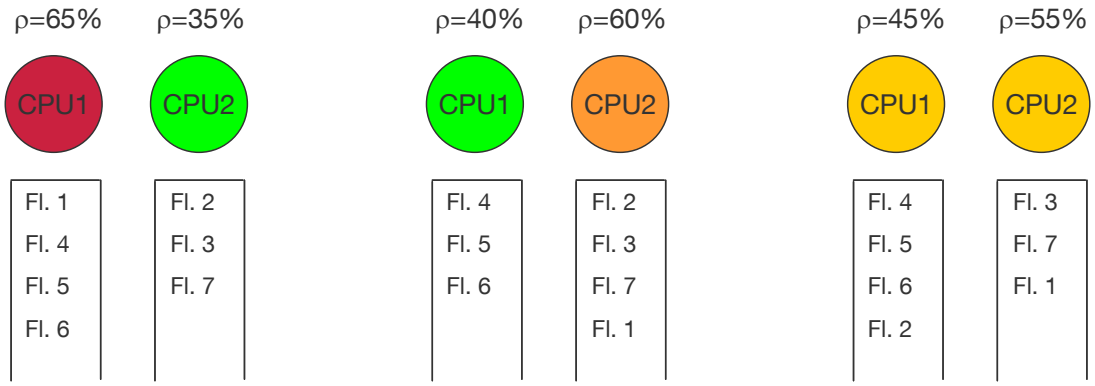


Figure 53: HLU Load Adaptation Scheme

By removing FlowIDs from the front of the overloaded PE's FIFO and appending it to the end of the least-loaded PE's FIFO, I insure that flows that have been rebalanced stick with the new assignment as long as possible. This behavior is in contrast to AHH, where load variations may lead to oscillations of flow assignments due to the nature of the HRW algorithm. The assignment persistence is beneficial in the context of stateful networking applications, where rebalancings not only pose the risk of packet reordering, but also come at the cost of migrating processing context from one PE to another.

The current load figures $\rho(i,t)$ measured on PE i at time t caused by the HLU-assigned traffic (i.e. disregarding the load caused by spraying of stateless applications) are gathered for each PE and are low-pass filtered according to the following iterative formula:

$$\rho_{low\_pass}(i,t) = .05 \times \rho(i,t) + .95 \times \rho_{low\_pass}(i, t - T_{adapt}) \tag{5-1}$$

From these individual PE loads, maximum, minimum and average utilization figures are computed as follows:

$$\rho_{\max} = \max_i\left(\rho_{low\_pass}(i,t)\right),\ \rho_{\min} = \min_i\left(\rho_{low\_pass}(i,t)\right) \tag{5-2}$$

$$\rho_{avg} = \frac{\sum_i \rho_{low\_pass}(i,t)}{i} \tag{5-3}$$

An adaptation is triggered, when the utilization of the highest-loaded PE $\rho_{\max}$ exceeds an adaptation threshold AT_1 and the imbalance between highest and least-loaded PE exceeds an adaptation threshold AT_2. If $\rho_{\max}$ is excessively exceeding the average load, flows are moved towards the least-loaded PE. The number of flows moved depends on the amount of relative overload $\left(\rho_{\max} - \rho_{avg}\right)$ and number of flow

bundles currently assigned to the highest-loaded PE (FIFO[max].size()). The term is multiplied with a low-pass factor of s_{over} to factor in the risk of moving an aggressive flow. Analogous to this, flows are assigned towards an excessively under-utilized PE with a slower low-pass factor of s_{under}. The low-pass factors help to evenly balance the loads over several adaptation periods, and wildly oscillating load assignments caused by aggressive flows are avoided.

Code Listing 1 describes the adaptation routine of HLU, which is executed periodically (period T_{adapt}) within the control plane software.

```
if(rho_max > AT1)
{
      if(rho_min < rho_avg-AT2 or rho_max > rho_avg+AT2)
      {
            if(rho_max-rho_avg > rho_avg-rho_min)
                  flows=sover*(rho_max-rho_avg)*FIFO[max].size();
            else
                  flows=sunder*(rho_avg-rho_min)*FIFO[max].size();
            while(flows>0)
            {
                  FIFO[min].push_back(FIFO[max].pop_front());
                  flows--;
            }
      }
}
```

Code Listing 1: HLU Adaptation Routine

The algorithm's parameters have been determined by a set of simulations with realistic Internet backbone traffic (see details in chapter 5.3.1) and yield good results for the considered traffic with the values according to Table 18. If the algorithm is applied on traffic with different statistical properties as observed in the traces used for our simulations, an adaptation of these parameters may be necessary. This adaptation may also be accomplished during system runtime by implementing a learning algorithm in the control plane of the NP. However, I have not performed a detailed analysis of such learning methods within the scope of the work covered in this dissertation.

Table 18: HLU Adaptation Parameters

Parameter	*Value*
AT_1	40%
AT_2	15%
s_{over}	$\frac{1}{8}$
s_{under}	$\frac{1}{16}$
T_{adapt}	50 ms

5.2.3. Combination of Stateless and Stateful Networking Applications

In real-world NP deployments, it is often the case that the device has to process a traffic mix that consists of both stateless and stateful networking applications. As the processing requirements with respect to packet order and assignment of subsequent packets of a single connection to the same PE are stricter than for packets belonging to the stateless application class, load balancing in actual NP deployments usually implement one of the hashing-based techniques as presented in section 2.4.

As I have shown in section 5.2.1, packet spraying is a suitable alternative for stateless traffic that achieves almost perfect load balancing and may exploit a pooling gain effect due to the statistical distribution of arriving packets onto the available processor resources.

For stateful networking applications, dedicated load assignment schemes that insure processing of packets of a specific flow on a distinct processor are required. In section 5.2.2, I have shown that this can be achieved by two techniques from the prior art (AHH and HABS), but as both techniques are rather complex to implement, I have proposed HLU as an alternative load balancing technique that requires less implementation effort.

A distinct feature of the FlexPath NP architecture is its capability to distinguish different applications in the ingress hardware data path and subsequently assign the arriving packets onto different processing paths. This feature can now be exploited for the load balancing problem by separating the arriving traffic into stateless flows, which may be sprayed among the PEs in the parallel processor cluster and stateful flows, for which load balancing can be achieved with HLU. The combination of these two load balancing techniques, which are respectively applied to different applications in the actual traffic mix is referred to as S&H (spraying and HLU) in the following.

Depending on the requirements of the individual applications, different queuing priorities can be chosen for each individual traffic type. As Michael Meitinger shows in chapter 5 of his dissertation ([107]), the Packet Distributor in our demonstrator implementation supports sixteen queues with a static priority, but each of the queues can be configured to be used either for packet spraying or implement a direct mapping to a single processing element. The Path Dispatcher is used to classify the incoming traffic and determines the Packet Distributor queue, into which the current packet is assigned. The combination of Path Dispatcher and Packet Distributor in a FlexPath NP thus provides a powerful framework for deploying sophisticated load assignment techniques that allow a fine-grained control of the packet assignment onto the available processing resources.

5.3. Functional Simulation of Load Balancing Techniques

5.3.1. Simulation Model

In order to evaluate and compare the performance of the various regarded load balancing techniques, we have developed a functional level SystemC model of the FlexPath NP system as depicted in Figure 54.

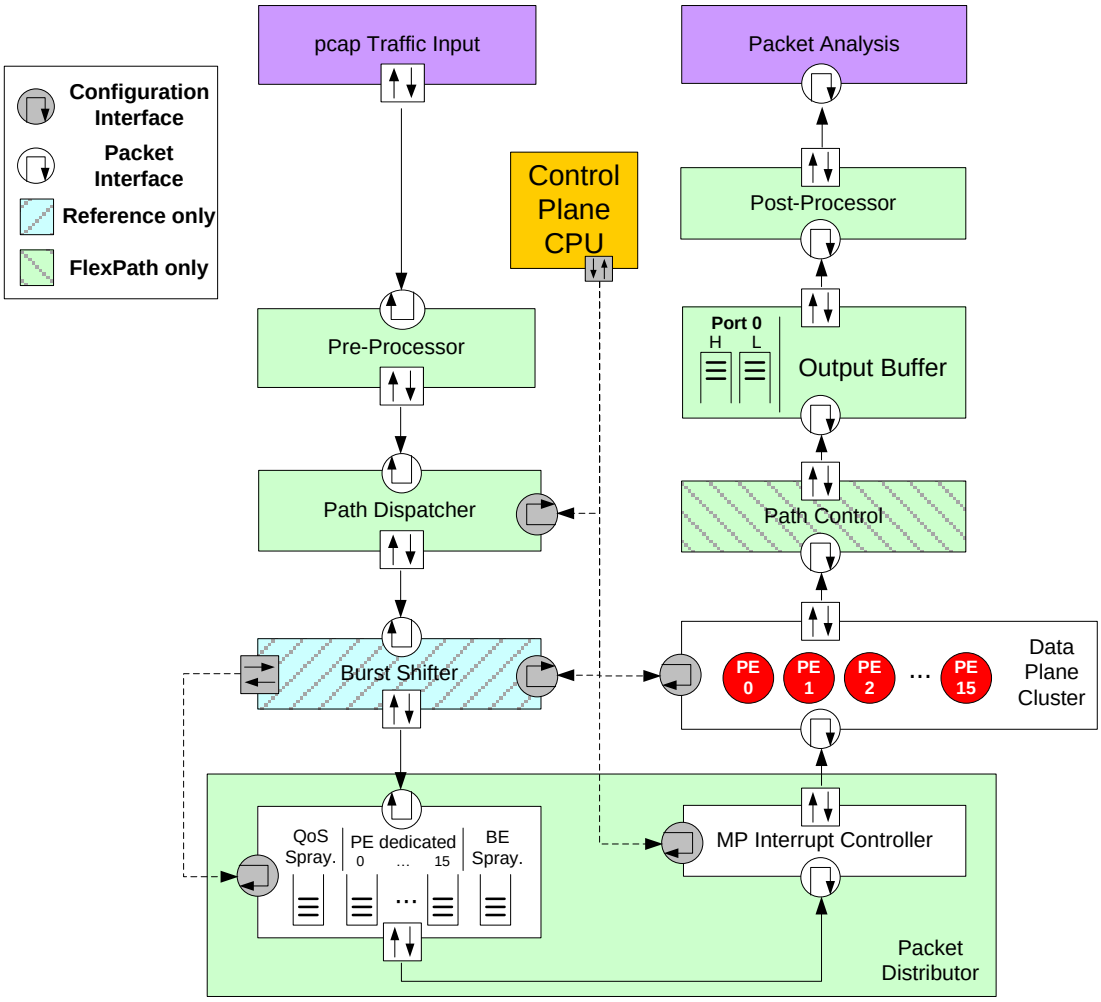


Figure 54: Functional Simulation Model of FlexPath NP and Reference Architecture for Load Balancing

Packet classification and hash table lookup (which is needed in both HLU and AHH) are performed in the Path Dispatcher model of the system simulator.

The reference scenario is not assumed to feature the extensions of a FlexPath NP like pre-processing and packet classification, thus the model of the Path Dispatcher is used to perform the load assignment according to the AHH or HABS schemes irrespective of the application type to which the arriving packet belongs. In order to implement the HABS scheme, a model of the Burst Shifter is necessary, which is absent in the FlexPath simulations. The burst shifter remaps flows in overload

situations based on queue fill levels and the current flow table entries as described in chapters 2.4.4 and 2.4.5. In the following simulations, the flow table size has been set to 16 entries.

For FlexPath NP, I demonstrate S&H as described in section 5.2.3. Stateless QoS and BE traffic is separated and assigned into two queues in the Packet Distributor, from which they are sprayed among the data plane processors on two different priority levels. The stateful IPsec traffic is assigned to dedicated queues that are each associated with a single PE and load balancing is achieved with HLU.

The Packet Distributor model supports 16 queues for up to 16 dedicated CPUs and two additional queues for high and low priority packet spraying. The QoS spraying queue has the highest interrupt priority, followed by the dedicated assignment queues and BE traffic is sprayed with lowest interrupt priority. The queue size is initially set to 32 packet descriptors, and packet descriptors are lost, when they are assigned to a full queue, i.e. there is no backpressure mechanism that could cause head-of-line blocking effects in the Packet Distributor. By implementing such a scheme in the Packet Distributor, it is possible to investigate average-case dimensioning of the NP architecture, where the provided processing performance in the processor cluster is sufficient to deal with average case traffic from the links, but not with worst-case scenarios (e.g. all packets require most complex processing and arrive with maximum possible rate). Here, it is possible that traffic variations lead to (temporary) overloads in the processor cluster, resulting in a certain amount of packet loss. By instantiating parallel, non-blocking queues for the different traffic types in the system, it is possible to guarantee QoS at least for the higher-prioritized traffic types and can limit the packet losses mainly to the BE traffic class.

The Packet Distributor queues should not be confused with queuing for solving output port contention. This is achieved in the output buffers in front of the transmit interfaces, i.e. our FlexPath NP model behaves like an output-buffered switch.

The processing latencies in the data plane cluster are derived from a networking stack implemented on our Virtex-4 FPGA-based demonstrator (see [106], NB: this version works only with the old Buffer Manager DMA as presented in section 3.3.2.2 and is not compatible with the SmartMem as presented in chapter 6) and have been measured to be 10 µs for plain IPv4 forwarding and

$$t_{proc,IPsec} = 310\mu s + \frac{packet\_length}{64byte} \times 112\mu s$$

for IPsec encryption. These latencies were measured on a single running CPU, which cannot be used in a straightforward fashion to model effects in processor clusters with significantly more cores. In order to cover processing jitter effects that appear in more parallelized architectures due to shared resource conflicts, 20% of

the packets are processed with a 50% processing time penalty and another 10% of the packets are processed with twice the latency obtained from the single CPU measurements.

In contrast to the original AHH implementation, which assumes uniform processing latencies for all traffic types, we are using the real CPU loads as input to the AHH algorithm. Kencl calculated the CPU load by multiplying the packet rate with the processing latency per packet, leading to a theoretical processor load that may exceed 100% in overload situations. Since in a heterogeneous application mix the processing latency cannot be predicted in such a simple fashion, we had to use the actually measured processor loads as described in section 5.2.2, but can thus not measure loads beyond 100%.

The following simulations have been performed with a set of real backbone traffic traces obtained from CAIDA ([96], [97], [98]). In order to obtain a comparable data throughput, traces from different points in time were multiplexed, thus preserving original traffic characteristics like packet inter-arrival times and flow characteristics, but increasing the overall bandwidth. One other trace, which came from a highly utilized link and which would have exceeded the processing capability of the implemented simulation model had to be slowed by a factor of four, but the high amount of bursts, which is in contrast to the characteristics observed in the other traces, makes it attractive for simulation in order to cover a wider range of cases investigated. Table 19 summarizes the main characteristics of the employed traces.

Table 19: Key Characteristics of Utilized Internet Traces

Trace Name	*Packets*	*Avg. Data Rate*	*IPsec*	*QoS*	*BE*
OC-48_mux [96]	22,086,716	1.955 Gbit/s	0.07%	4.14%	95.79%
OC-192_mux [97]	41,223,895	2.819 Gbit/s	0.40%	4.04%	95.56%
OC-192_quarter [98]	26,473,646	1.320 Gbit/s	0.63%	7.39%	91.98%

The OC-48_mux trace is generated from four original traces taken on an OC-48 backbone link in 2002 [96]. The original link was only about 30% utilized, so we multiplexed traces taken at 15-minute intervals into a single trace and limited the file to one minute duration. Intermediate bursts were limited to 3.2 Gbit/s as our simulation model assumes a 32 bit data path running at 100 MHz as models for the FlexPath NP hardware pipeline in accordance with the implementation results on our FPGA-based demonstrator platform (see chapter 6).

The OC-192_mux trace was obtained in the same fashion as the OC-48_mux trace, but using data from a 2008 snapshot [97].

The OC-192 trace in the opposite direction is the highly utilized one with bursts exceeding 9 Gbit/s for periods of a few seconds and intermediate idle times that was slowed down by a factor of four in order to get into the less than 3.2 Gbit/s range also during most of the original bursts in the trace.

When comparing the traces from 2002 to those taken in 2008, it can be seen that both the IPsec and QoS-marked traffic shares have increased significantly. Still, the best effort traffic consumes more than 90% of the traffic in current high-speed Internet links.

Although the FlexPath NP architecture was designed with networking application mixes in mind, which are typically found at the network edges, rather than in the core network, it was necessary to resort to those backbone traces, as edge or access network traces are not made publicly available for both privacy and security concerns. We still consider the later obtained results and conclusions to be valid, as backbone traffic is essentially only a multiplex of a multitude of edge traffic streams, thus important characteristics like protocol distribution and flow-specific characteristics like data rates, burstiness and packet inter-arrival times are preserved through the multiplex.

5.3.2. Individual Performance of Load Balancing Techniques

The load balancing techniques presented in the prior art section were all described in an environment with homogeneous processing, i.e. no QoS classification of application differentiation was regarded with respect to the load balancing problem. In order to make the individual proposed load balancing techniques, i.e. packet spraying and HLU, better comparable with those from the prior art, they will be evaluated against each other in the following chapter using a simple NP scenario, where the entire traffic is subject to plain best effort forwarding. S&H as a combination of two load balancing techniques will be presented later after having evaluated its individual components. I have not simulated all prior art schemes (AHH, Burst Shifting and HABS), but use the most advanced scheme from the prior art (i.e. HABS) as a reference, against which the newly proposed packet spraying, HLU and later also S&H will be compared. In the following scenarios, the Path Dispatcher is configured to perform only the various load assignment strategies without classifying the traffic into the QoS, IPsec and BE classes. The CPUs also apply the forwarding latency with the previously mentioned jitter behavior. I show the simulation results obtained with the OC-48_mux trace (see Table 19) in the following; the general behavior does not change significantly when simulating the system with the other traces. Some results obtained for the other traces will be shown later in chapter 5.3.3, when the proposed S&H technique is applied to a heterogeneous traffic mix.

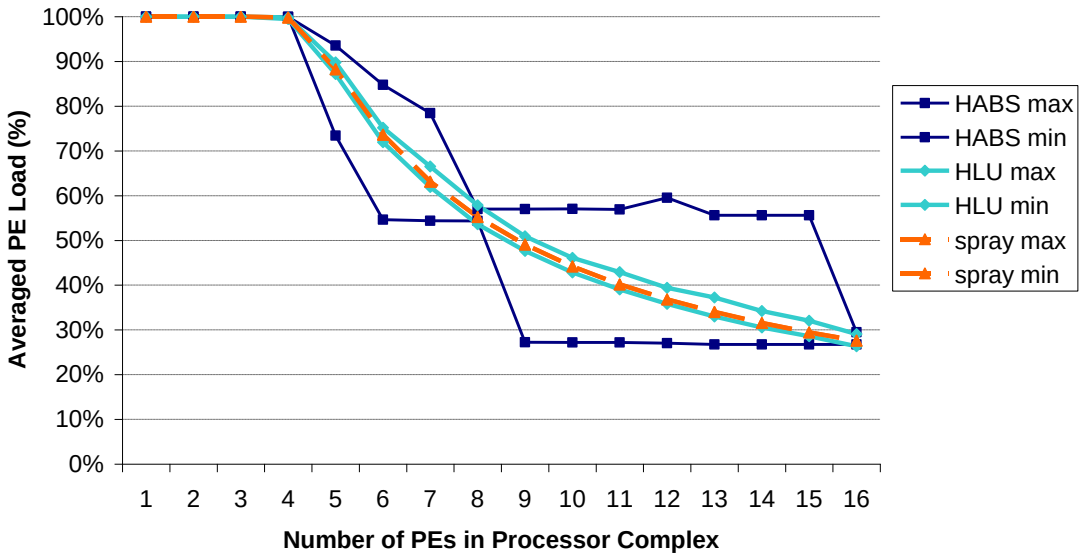


Figure 55: Minimum and Maximum CPU Loads Observed with Different Load Balancing Strategies

Figure 55 shows the ranges between minimum and maximum of the averaged individual PE utilization for each of the three investigated load balancing schemes with an increasing amount of processors in the central network processing cluster. It can be seen that for packet spraying we receive a single line indicating that all processors are sharing exactly the same load, which can be seen as an optimally balanced workload. In HLU, the adaptation threshold AT_2 limits the difference between most and least utilized processor to a maximum of 15%, in reality this imbalance is even lower. Still, there is a residual load imbalance associated with the flow persistent load assignment. As not all flow bundles cause the same amount of processing effort, any dedicated split-up of the entire load will eventually lead to slightly varying workloads on the individual processors. In stark contrast to the two before-mentioned schemes are the results for the HABS load balancing. Both the AHH as the Burst Shifting components, which are part of the HABS algorithm, are designed to eliminate temporal overload in processor utilization. However, none of the two schemes explicitly optimize the load distribution in underload situations, i.e. if none of the processors reaches its capacity limit, no further loads are remapped. This may lead to grossly imbalanced loads especially when the average system load is below 60%, where certain processors remain around 60% utilized while other processors are starved at less than 30%.

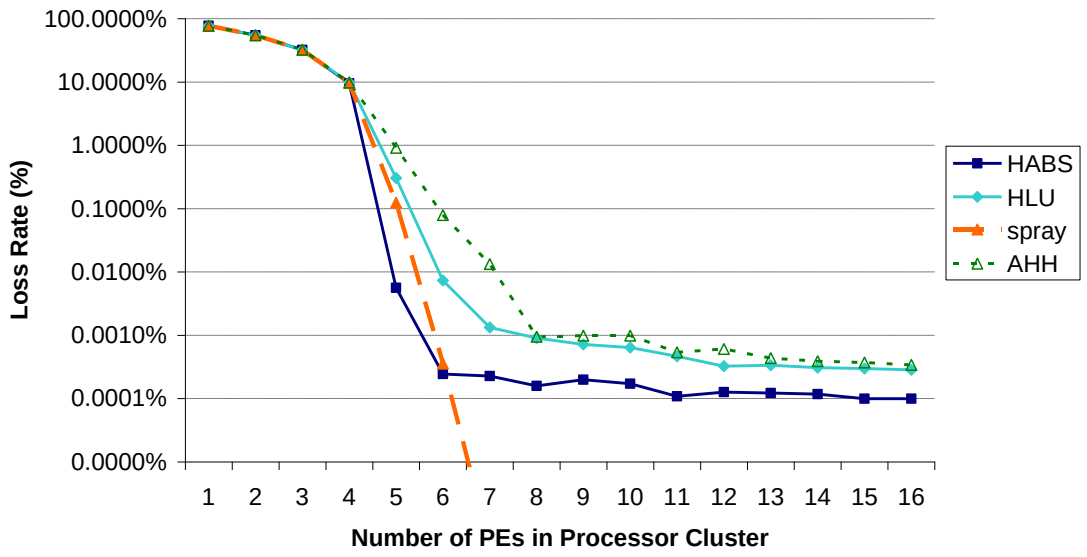


Figure 56: System Packet Loss Rates for Different Load Balancing Strategies

Figure 56 shows the resulting packet loss rates achieved with each of the different load balancing strategies. In addition to packet spraying, HLU and HABS, I have also included a simulation with plain AHH, as this scheme is conceptually closest to the newly proposed HLU assignment (both are based solely on hash bundle load assignment, only the adaptation strategy is different). It is important to realize that a system configuration with less than five CPUs is not sufficient to process the incoming traffic, i.e. the simulated NP system is in overload and loses significant amounts of the incoming packets. As the overall processor load declines between five and six processors, the packet loss rate is reduced to less than 10^{-5}. With more than 7 processors, the provided processing power is greater than necessary to cope with the offered load also during temporary bursts, thus the packet loss rate can be interpreted as a measure for the effectiveness of the individual load balancing mechanisms.

The worst packet loss can be observed for both AHH and HLU; beyond eight processors, these two schemes almost converge on a similar performance level. This can be explained by the fact that both AHH and HLU base their decision on a hash split of the traffic among the available processing resources and imbalances are leveled out with an adaptation interval in the millisecond range. However, network traffic also has very short-lived bursts that lead to brief temporary overflows in the Packet Distributor's queues. Reducing the adaptation interval of the two load balancing algorithms does not really help, as this would lead to a high amount of flow bundle rebalancings and the system would not converge to a steady-state, in which a certain level of flow persistence can be maintained. In the transitional range for five to seven processors, where the system emerges from the overload situation, HLU with its more even balancing of the traffic performs better than AHH.

It also appears that HABS performs better than spraying, which might be surprising at first. However, these results can easily be explained by the fact that the buffer space in front of the processors is higher for HABS (and AHH, HLU) than in packet spraying, as the sprayed packets go through a single queue (with 32 entries), the dedicated assignment schemes all feature a queue with 32 entries per CPU, i.e. in case of five CPUs, the buffer space is 80 packets for the dedicated assignment and only 32 for packet spraying. Such queuing effects will be studied in further detail later in Figure 58.

The HABS load balancing scheme performs about half an order of magnitude better than both AHH and HLU. As in HABS the AHH algorithm is extended with the Burst Shifter described in chapter 2.4.4, temporary overloads caused by short-lived bursts are effectively distributed to less-utilized processors also in between two AHH adaptation times. However, this increased performance has to be paid with a rather high additional effort, as a flow classification has to be performed on the ingress side of the NP and the flow table must be maintained for all flows that are currently active in the system. In a FlexPath NP architecture this could be achieved by sharing some resources in the Path Control unit (although this works on flow bundles rather than microflows); otherwise a similar implementation effort has to be performed.

The best performance of the previously discussed schemes is achieved with packet spraying, and a true lossless operation is achieved for any scenario beyond seven CPUs. Packet spraying achieves the best results as there is no aggregation of packets from a burst in front of a single PE.

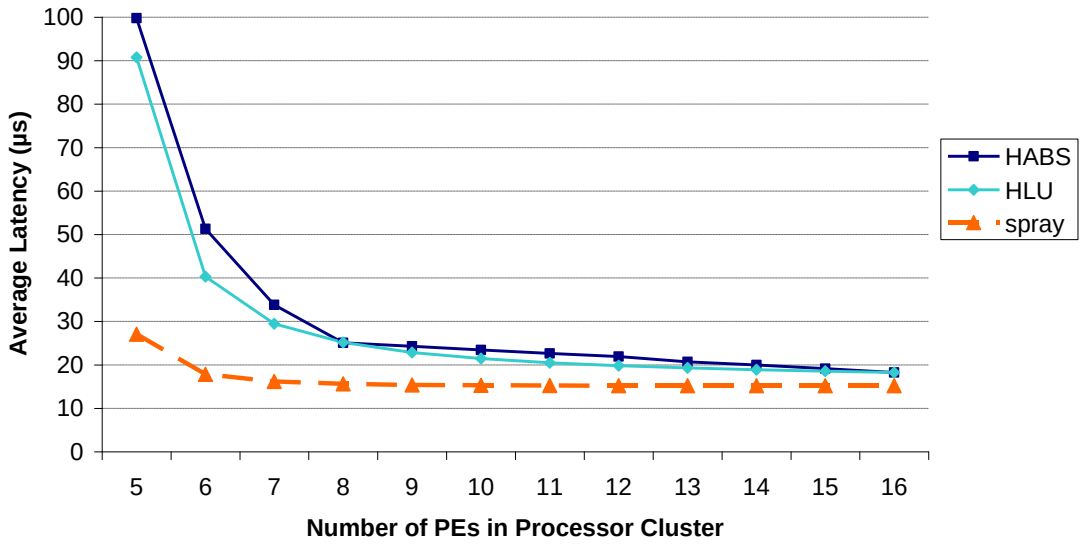


Figure 57: Average Packet Latency for Different Load Balancing Strategies

For the remaining investigations, I focus on the range between five and 16 processors, so the dramatic overload situation is avoided. Figure 57 shows the averaged latency of the packets subject to HABS, HLU and packet spraying

algorithms. The stated latency figures are measured from receive interface to transmit interface and thus include the pre-processing delay, CPU processing delay and possible packet re-sequencing delays. For packet spraying, the latency is reduced very effectively, until a minimum floor is reached, which is defined by the processing time in the central processing cluster without any further queuing delays. As the individual processor loads are more evenly balanced in HLU compared to HABS, the latency which is achievable with HLU is also slightly smaller than that of HABS as shorter average queue lengths may be assumed in front of each processor.

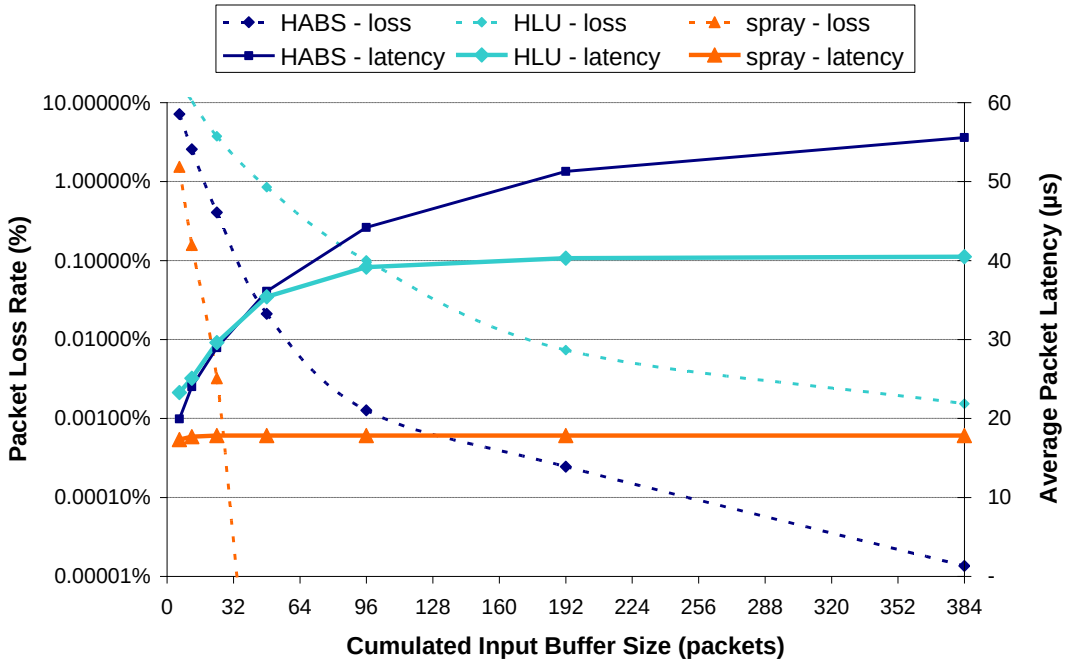


Figure 58: Packet Loss Rate and Average Latency for Different Packet Distributor Buffer Sizes (6 PEs)

Packet loss rates and latencies are not only dependent on the number of provisioned processors, but are heavily dependent on dimensioning the buffers in the system. As it can be seen in Figure 58, the packet loss rate may be reduced effectively by provisioning larger buffers holding packet descriptors in the Packet Distributor. This may be explained by the fact that during those previously discussed packet bursts the buffers are not any longer overflowing, but are able to hold all incoming packets. When the burst is over, the backlog may be worked off. In turn, the average observed packet latency is increasing, because the packets accumulated during bursts are still sitting in the queue and suffer a longer delay in comparison to when they were lost (where we would not count a latency of infinity!).

In general, we can see that while architecting an NP system, we can trade off additional processing resources with an increased buffer size. If the bursty nature of realistic Internet traffic is figured into the dimensioning process, significant amounts

of processing resources can be saved by adding some extra buffer space in order to accommodate more packets during relatively short bursts and still being able to forward the offered traffic. However, the increased latency resulting from larger buffers might have a negative effect on interactive applications like VoIP or Internet video. While packet spraying operates losslessly and with a constant latency beyond a buffer size of 48 packet descriptors for the given processor cluster size of 6 PEs, the other schemes require significantly larger buffer space to reduce the packet losses.

The FlexPath NP provides different processing paths for packets of different applications. The Path Dispatcher in the ingress path of the architecture determines the actual path, to which each arriving packet is assigned. In this context, the before mentioned queue length vs. number of PEs vs. packet loss vs. packet latency tradeoff might be evaluated differently for various traffic classes. In consequence, it is conceivable that best effort traffic types are tackled with relatively fewer processing resources and excessive packet loss is avoided by larger queues in the Packet Distributor. In turn, more processors may be used for QoS-sensitive applications in combination with shorter queues in the Packet Distributor in order to minimize packet latency.

5.3.3. Performance of S&H Load Balancing

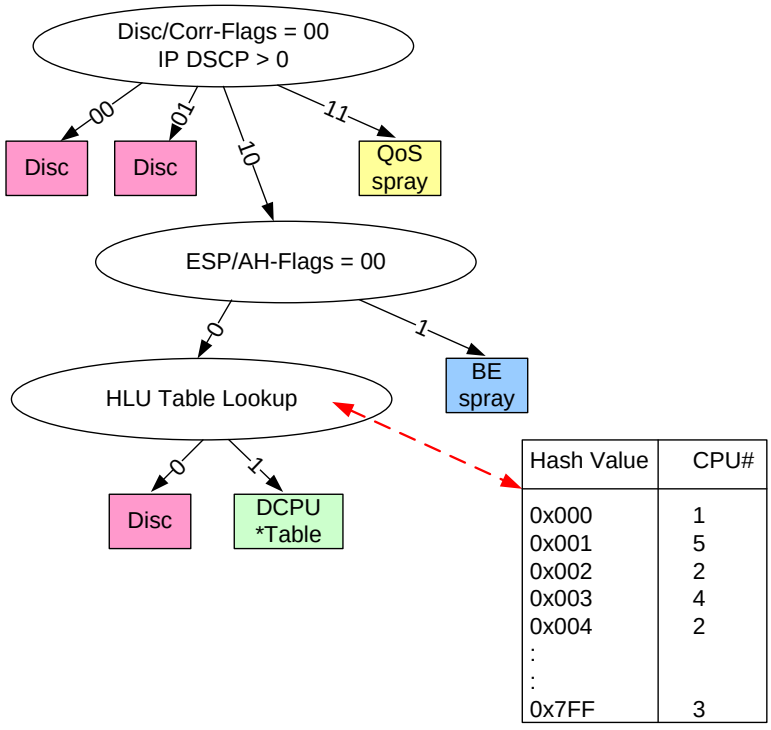


Figure 59: HDGA Decision Graph for FlexPath NP Load Balancing Simulation

In the following chapter, I show the achieved load balancing and forwarding performance by combining a multi-priority packet spraying for stateless traffic with

HLU for stateful IPsec processing. The combined scheme for the previously described heterogeneous application mix is in the following referred to as S&H. The Path Dispatcher is configured to assign the QoS and BE packets directly to the two queues that spray the traffic over all processors. Only packets identified as IPsec are assigned to dedicated CPUs using the HLU algorithm. The resulting HDGA decision graph is shown in Figure 59.

As described in chapter 4.2.5, the control plane CPU executing the HLU algorithm only has to manipulate the contents of a hash table in order to rebalance the dedicated load between the CPUs in the processor cluster. We assume a constant application scenario during runtime, so that the decision graph itself is not changing during runtime. As the load balancing schemes from the prior art do not consider such a heterogeneous processing approach and have typically no provisions to differentiate between various applications in the ingress path of the NP, we use HABS to balance the entire traffic load, irrespective of the actual packet processing requirements. However, in both simulation scenarios (FlexPath and reference) the processors determine whether to apply forwarding or IPsec latencies to the incoming packet. Therefore, the overall processing requirements remain the same for the reference simulations and the S&H load balancing in a FlexPath NP.

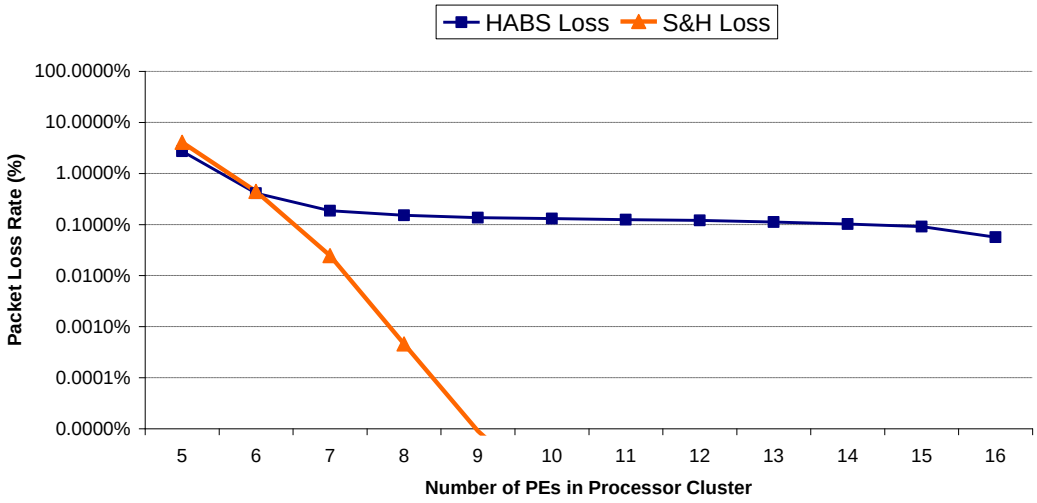


Figure 60: Packet Loss Rates of S&H (FlexPath) and HABS (Reference)

The results shown in Figure 60 show a consistent behavior with respect to the individual characteristics observed for the load balancing schemes in isolation that I have presented in chapter 5.3.2. As the vast majority of the packets in the simulated Internet traffic trace belongs to the BE class (see Table 19), the same kind of "waterfall" packet loss rate can be observed for S&H in the FlexPath NP simulation. However, as IPsec processing takes roughly three orders of magnitude longer than plain IP forwarding, lossless operation is achieved only beyond nine processors, in contrast to the seven processors previously needed for plain forwarding as shown in Figure 56.

Figure 61 shows the packet latencies differentiated by the respective application types in addition to the used load assignment scheme. By giving priority to the QoS packets in FlexPath, we are able to forward them with almost minimum latency, even while the buffers for BE traffic are in overload and packets are lost in the system. The latency figures for BE traffic and QoS high priority traffic converge towards the minimum latency which is determined by the plain processing latency from nine processors onwards. In contrast, the latency of the IP forwarding packets is about a factor of three to four larger in the reference simulation scenario (HABS), as the plain forwarding packets occasionally get stuck in the queue behind IPsec packets (head-of-line blocking effect). In addition, as QoS packets can not be recognized at the ingress path of the NP, no performance advantage can be observed for them. Differences might still be achieved in the reference simulations, if the processors assign the packets to prioritized output queues, so that the QoS packets may receive beneficial treatment with respect to output port contention resolution and queuing at the egress side of the NP architecture. This kind of output queuing and scheduling is a widely accepted standard in NP architectures and is typically implemented by the Traffic Manager hardware resources found in commercial NPs (see chapter 2.1.1). However, output port scheduling and port contention resolution effects have not been captured in our simulation model, as we have not implemented an explicit routing functionality in the processing software model.

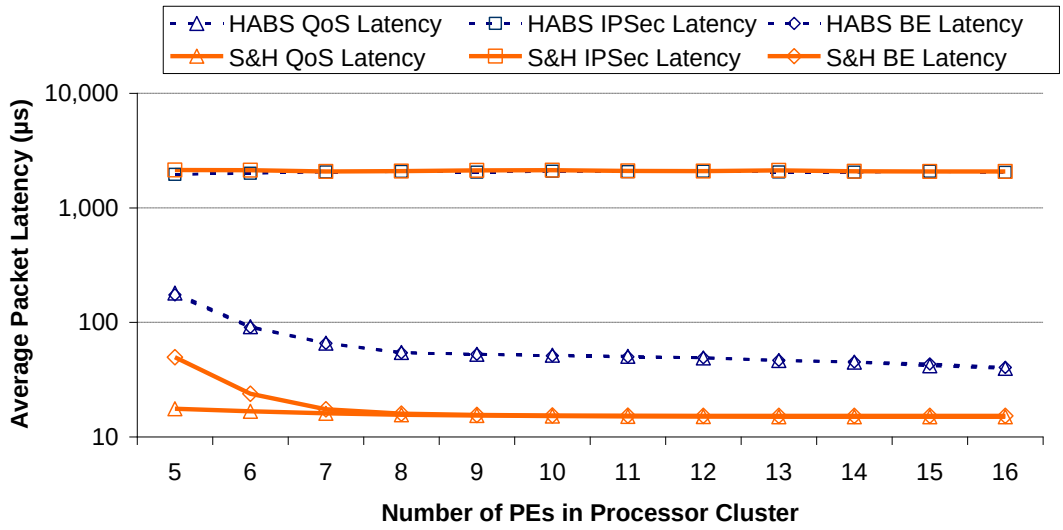


Figure 61: Packet Latencies for S&H (FlexPath) and HABS (Reference)

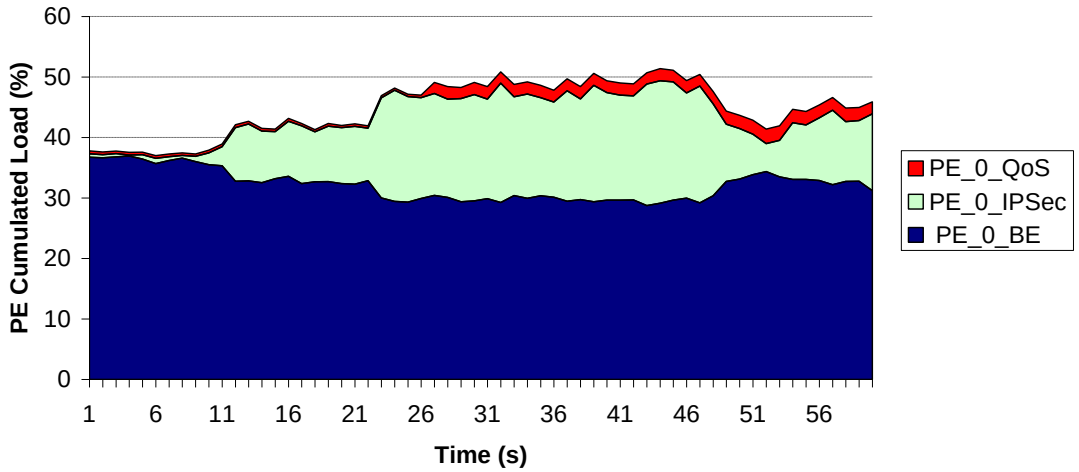


Figure 62: Individual PE Load Share over Time (S&H)

Figure 62 investigates the variations in the load of an individual PE over the course of the simulation for the different processing classes. The proposed combination of packet spraying and dedicated load assignment with HLU in a FlexPath NP allows some "load breathing" on the individual processors. While a larger share of IPsec packets is assigned to the PE (e.g. between 25 and 45 seconds in the simulation shown above) and consumes a larger share of the available processing performance, sprayed traffic is superseded (and in consequence processed by other PEs, which carry less IPsec traffic at the same moment). The supersession is not explicitly triggered by the control plane CPU, thus it happens instantaneously and packets assigned to the spraying queues don't get stuck waiting for the IPsec packet to finish.

Finally, we wanted to investigate the performance of our proposed load assignment scheme with respect to packet reordering, which has been given great attention in all prior art schemes for NP load balancing. If the Path Control is properly dimensioned, the packet reordering problem is solved on the system level. Figure 63 shows that the packet reordering rate resulting from packet spraying in the proposed form for FlexPath measured in front of the Egress Path Control is roughly 0.35% of all packets, which is quite significant and roughly two orders of magnitude more than the 2×10^{-5} achieved with the HABS scheme. However, at the output of the Path Control unit packet reordering can be completely eliminated except for the two simulation runs performed with six and seven processors. More details about packet reordering in FlexPath and how to properly dimension the Path Control unit can be found in Michael Meitinger's dissertation ([107]).

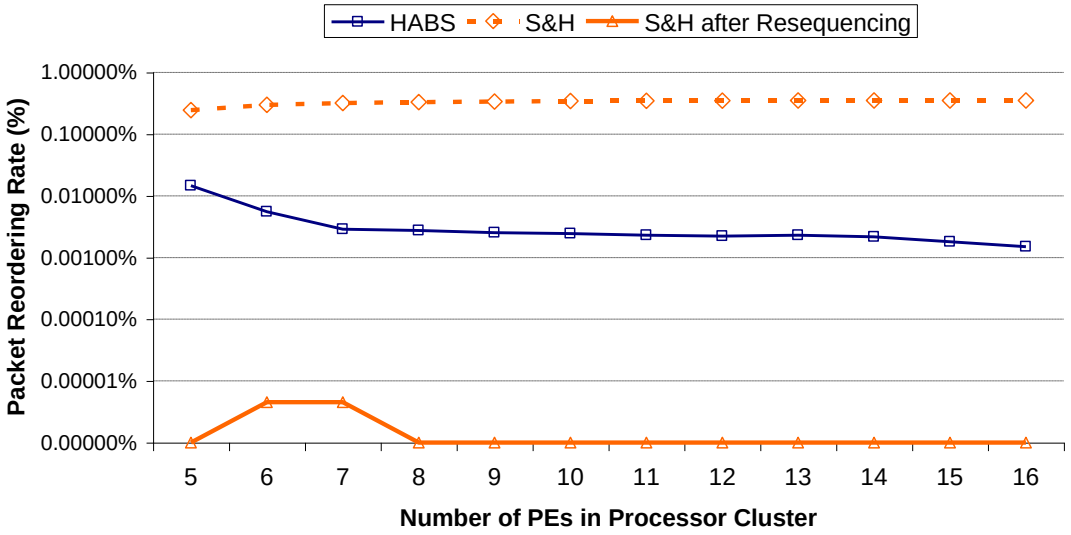


Figure 63: Packet Reordering Rates

The same set of simulations and investigations presented before with the OC-48_mux trace have also been repeated with the two other traces (OC-192_mux and OC-192_quarter). The general behavior of the prior art and S&H load assignment schemes has been confirmed, so that all resulting plots look quite similar. However, due to the different shares of IPsec traffic and different average throughput of the traces, the performance figures are generally shifted towards the right, i.e. lossless operation and low latencies across all traffic types are only achieved with more processors in the processor cluster. The key performance figures obtained for all three traces for a comparison between S&H and HABS in the multi-application mix are summarized in Table 20 below. The performance figures are quoted for the system architecture with the minimum number of processor cores necessary for lossless operation of the sprayed traffic, i.e. QoS and BE. In both S&H and HABS scenarios, packets with dedicated load assignment (i.e. IPsec for S&H) may be lost due to temporary traffic bursts that exceed the provisioned buffer capacity.

Table 20: NP Performance Characteristics for S&H (FlexPath NP) and HABS (Reference Architecture)

Trace	*# of PEs*	*Scheme*	*Packet Loss*	*QoS Latency*	*IPsec Latency*	*BE Latency*
OC-48_mux	10	S&H	0.0000%	15,177 ns	2,131,125 ns	15,378 ns
		HABS	0.1311%	51,600 ns	2,044,273 ns	52,624 ns
OC-192_mux	16	S&H	0.0002%	15,926 ns	3,122,052 ns	16,139 ns
		HABS	0.2865%	66,928 ns	1,586,312 ns	69,614 ns
OC-192_quarter	15	S&H	0.0010%	15,266 ns	3,896,399 ns	15,058 ns
		HABS	0.4792%	159,945 ns	2,124,968 ns	157,635 ns

A common characteristic is that in the FlexPath NP architecture, the latency of the QoS packets is always slightly smaller than that for the BE packets, due to the

higher interrupt priority in the Packet Distributor and the pre-classification of the incoming traffic in the ingress hardware processing pipeline of the NP. For the same reasons, it can also be observed that the IPsec latency in FlexPath is also consistently higher than in the reference simulation with HABS. In the reference scenario, IPsec and non-IPsec traffic is assigned into the same queue in front of the processors. Thus the probability for an arriving IPsec packet to be stuck behind another IPsec packet is significantly less, as there are much more BE packets than IPsec packets in the traffic. In FlexPath NP, IPsec packets have their private queue, separated from the BE and QoS traffic classes, so that they always get stuck behind other IPsec packets. This effect can also be seen, as the latency of the BE and QoS packets in the reference simulation is also significantly higher than in S&H, where such packets may only get stuck behind other forwarding packets, but not behind IPsec packets. QoS and BE packets in the reference simulation are similar across the different traces, but QoS packets are not guaranteed to have a lower latency than BE packets.

5.4. Conclusions

In the previous chapter, I have focused the investigations on the generic problem of load balancing among parallel processing elements with respect to the FlexPath NP architecture. As the FlexPath NP with its Path Dispatcher unit in the ingress data path pipeline provides a dedicated unit for traffic classification and differentiation, I have shown how to capitalize on this infrastructure in order to achieve superior load balancing behavior and QoS performance. Based on networking application characteristics, three different cases are regarded:

- **Stateless Networking Applications** don't rely on a shared connection state and processing of the arriving packets can be performed by any PE in an independent fashion. As packet reordering is addressed separately by the Path Control unit in a FlexPath NP architecture, we have identified packet spraying as a viable load balancing technique for this traffic class. Packet spraying is very beneficial as it achieves the most evenly balanced load distribution among the involved processors and is completely self-organizing, i.e. no additional monitoring and rebalancing effort is needed in the system.

- **Stateful Networking Applications** should be load balanced using an adaptive hashing-based load assignment scheme in order to keep processing state information local on a specific PE for each flow. Following an analysis of state-of-the-art techniques, I proposed HLU as a simpler but equally effective load balancing technique. HLU can be very efficiently mapped to the table lookup functionality offered by the HDGA algorithm in the Path Dispatcher.

- **Traffic Mixes** containing both stateful and stateless traffic types are typically encountered in real-world scenarios. For these circumstances, I have proposed to combine packet spraying and HLU into a new technique called S&H.

The simulations of the different load balancing techniques with realistic Internet traffic traces have revealed that packet spraying achieves the biggest performance improvements with respect to reduced packet latencies and loss rates and also by equally distributing the arriving load over the different processors. Fortunately, as most of the traffic in current networks belongs to the stateless traffic class, these benefits are preserved in S&H load balancing, as only a minor share of the traffic is assigned by HLU. Imbalances caused by the dedicated assignment onto specific processors can be filled with sprayed traffic without the need for dedicated control from the system management plane.

In addition to combining different load balancing techniques for different traffic types as in S&H, the FlexPath NP architecture allows to differentiate the QoS-levels of the incoming traffic in the Path Dispatcher rule base. Therefore, it is possible to prioritize performance-critical traffic streams even before they reach the central

network processing complex of the NP in the Packet Distributor. In both the simulation model and the FPGA prototype implementation (see chapter 6), we have implemented a strict priority-based scheduling in the Packet Distributor, but it would be possible to implement more sophisticated strategies, if this was mandated by the QoS requirements in the respective per-hop-behavior. As the classification in the Path Dispatcher is happening under hard real-time constraints, an additional latency advantage can be achieved for high-priority packets in a properly configured FlexPath NP in comparison to a reference architecture, in which the prioritization is performed in software.

6. FlexPath NP Demonstrator

In the following chapter I conclude the technical part of this dissertation with measurement results obtained on an FPGA-based prototype implementation of a FlexPath NP in conjunction with the SmartMem buffer manager. In section 6.1, I will outline the goals of the prototype implementation and introduce the FPGA development board. Section 6.2 presents the implemented FlexPath NP on the FPGA platform describing the most important features of the implemented functional modules. In order to give the reader sufficient insight about the system view of the FlexPath demonstrator, I have also included brief descriptions of the functional modules implemented by my colleagues Michael Meitinger and Daniel Llorente, who present the conceptual and implementation details in their respective dissertations ([107] and [108]). Section 6.3 describes the lab equipment and measurement setup that has been used to obtain the demonstrator results. The results are then presented in three separate sections: a processor-centric reference scenario, which is used to determine the baseline performance of the implemented NP without using any of the FlexPath NP-specific functions is described in section 6.4. In section 6.5, I investigate the effects of the hardware-offload capabilities in a FlexPath NP on the system performance. The load balancing techniques are then addressed in section 6.6, before the chapter is concluded in section 6.7.

6.1. Demonstrator Goals and Platform

In order to prove the validity of the FlexPath NP architectural approach and in order to support the results obtained from the various simulations presented before, we decided to implement a full-featured prototype of a FlexPath NP on an FPGA development platform. The main objectives of the demonstrator are twofold:

- By implementing the crucial functional elements of the FlexPath NP architecture (i.e. Pre-Processor, Post-Processor, Path Dispatcher, Path Control) we can prove the feasibility of the proposed elements. In addition, we can gain real-world performance figures like area consumption and packet throughput that would not be accessible purely by simulation.

- As we have also implemented an entire NP system, we are able to perform performance measurements and to compare those figures to the projected behavior obtained through our various system-level simulations. In addition to justifying the previous simulation results, a real implementation will also reveal behavior which is typically not captured by simulations, due to the higher level of abstraction and simplifying assumptions of the simulation model. While a simulation model is well suited for exploring new ideas and giving initial support for new hypotheses, only a full implementation of the proposed architecture is able to finally prove the viability of each concept.

Of course, such an FPGA-based demonstrator also has some problems, which should be briefly discussed here.

At first, implementing a complex multi-processor system-on-chip design such as a network processor is an inherently hard task, which consumes a lot of effort from a pure practical and engineering standpoint, in addition to the scientific and theoretical hurdles. In order to reduce the implementation effort, we used readily available IP (here: intellectual property) as far as possible and tried to get along with optimized solutions only in the design and implementation of the FlexPath-specific performance-critical entities. In turn, some of the out-of-the-box implementations may not be performance-optimal as they have not been specifically optimized for a high performance use case. The same also holds true for the software development process. While we initially tried to build the IP stack for the two PowerPC cores on the lightweight IP stack [95] supported by Xilinx; we discovered that this solution had several severe drawbacks. At first, data plane and control plane functions were not separated as the software was originally intended for an embedded microcontroller scenario with TCP/IP communication functions rather than implementation of a router. Therefore, it was not possible to execute the code on several processors in parallel, while sharing a common configuration among the cores. In addition, the stack also lacked an IPsec implementation, which we wanted to include for demonstrating the effects of different networking applications on the overall system performance. Finally, we ended up implementing our own stack with some control plane functions centralized on one PowerPC and a set of data plane functions that can be executed on both processors. Although the structure chosen for this implementation basically matches the architecture found in parallel processor cluster NPs, it is a plain C-code program, which is compiled with the standard EDK/gcc tool chain and has not been specially optimized for maximum performance, e.g. by heavily using inline assembly. In addition, as the two PowerPC processors execute different executables (they share the same packet processing functions, but the Control Plane executable includes additional code and the initialization routines and Packet Distributor drivers are slightly differently configured for the two processors), the achievable forwarding performance is not equal, probably due to non-linear effects when compiling and linking the slightly different code bases.

The second problem of the achievable measurement results are properly judging their relevance as they would have to be compared to ASIC implementations from the commercial domain. As mentioned before, the implementation of the demonstrator could not be optimized as far as a competing commercial architecture would be. In addition, we are constrained in our implementations by the FPGA environment provided by Xilinx. The embedded PowerPC cores run at a clock frequency of 200 MHz, which compares to 1.5 GHz in some commercial NPs. Implementing the application-specific logic in FPGA technology, based on mapping

logic functions to lookup-tables and interconnecting different functional entities by means of the FPGA switching fabrics can not be compared to a standard cell ASIC design flow, where the functionality is implemented in dedicated logic gates and the wiring is also customized.

At this point, it is also important to stress that the FlexPath NP concept is claimed as a general architectural extension to current network processors, and is not constrained to implementation in an FPGA environment. The FPGA demonstrator should be understood as a suitable tool for a university research group to easily (and cheaply) achieve valid implementation results.

Finally, the results obtained by the demonstrator implementation deliver a good insight into the behavior of the proposed FlexPath NP architecture, even though the performance level is not competitive with current commercial designs. The flexibility associated with the FPGA design flow also allows to (relatively) easily reconfigure the device to implement different features and provide also a reference for the non-FlexPath NP case. By doing this, the gain associated with the proposed architectural enhancements can be quantified.

An initial effort was undertaken to implement Pre- and Post-Processor along with the lightweight IP stack [95] to obtain first estimations for the system simulations as presented in chapter 3.3.2.2. As the Virtex-II Pro FPGA board used for this initial demonstrator had insufficient resources to implement an entire network processor in it, we moved our efforts to the Xilinx ML410 development board [100], which features a significantly larger Virtex-4 FX 60 device. The FPGA features two hard-macro PowerPC cores and Gigabit Ethernet MACs. The configurable logic comprises 25,280 slices with two Flip-Flops and two 4-input lookup tables each and the device has 4,176 kb of embedded SRAM distributed over 232 BlockRAM instances [101].

On the development board itself, there are two Gigabit Ethernet PHYs (one of them connected to the FPGA via RGMII and the other one via SGMII using the high-speed differential serial I/Os of the Virtex-4). In addition, there are two types of dynamic memory: a 64 MB DDR-SDRAM and a 256 MB DDR2-SDRAM. We had to use the DDR-SDRAM memory as shared memory for both the software and packet memory, due to clock tree limitations. The involved clock region has to support different clocks for the PowerPC hard cores, receive and transmit clocks for the two MAC blocks plus the 100 MHz system clock for the PLB bus and all attached logic in addition to the DDR clock signals needed for driving the memory interface. Using the DDR2 memory would have been very attractive from the performance standpoint as it has a 64 bit data bus that matches the PLB width. The DDR memory only features a 32 bit data bus which reduces the available memory access bandwidth for the NP demonstrator.

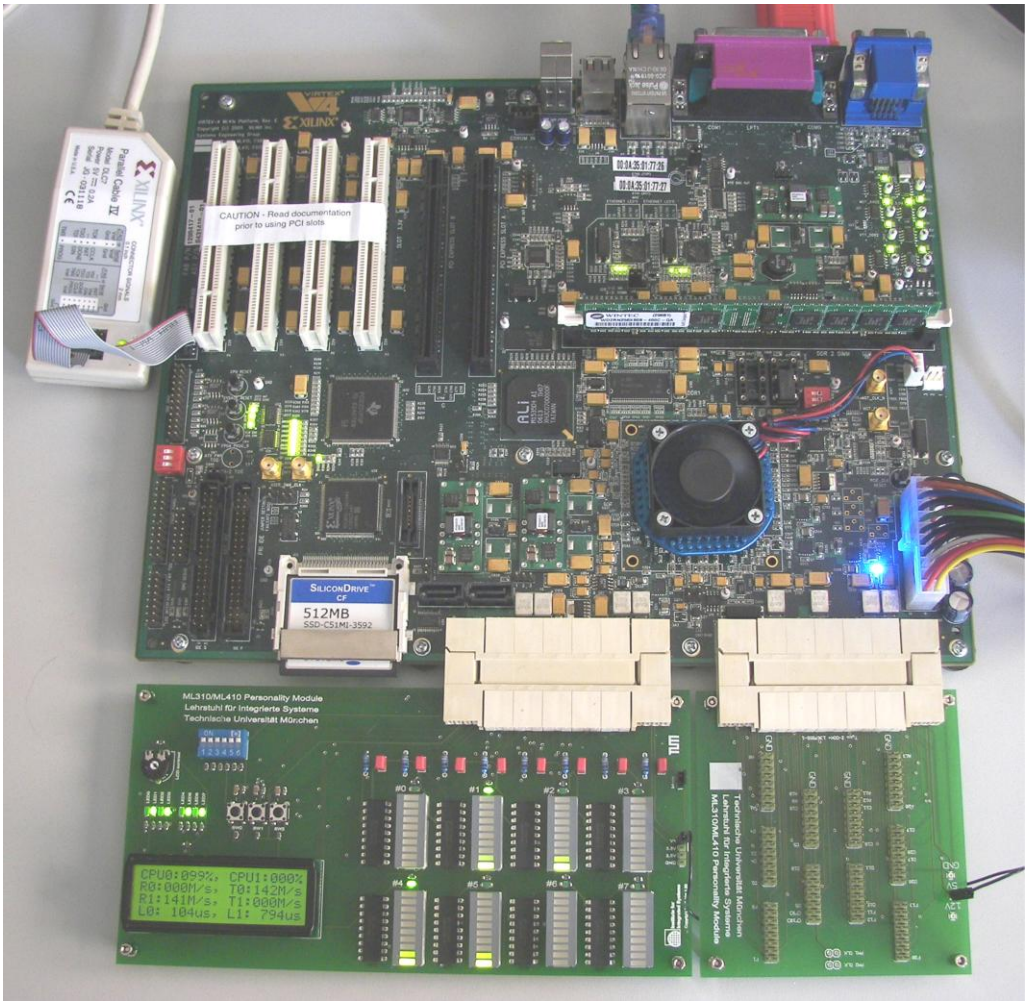


Figure 64: Photo of ML410 Development Board with Two Customized Extension Boards

We have produced two extension boards for the ML410, which are connected using the personality module expansion ports. A smaller board routes 80 general-purpose pins from the FPGA to 0.1" test headers for debugging and analysis of the implemented designs with our Logic Analyzer. The second expansion boards features a set of LEDs, a LCD and some push-buttons for visualization of the internal state and parameters of the system and providing a simple means of I/O for triggering various system configurations. Figure 64 shows a photo of the ML410 board with the extension boards as we have used it for our final measurements.

6.2. FlexPath NP System Overview

Figure 65 shows the functional blocks and major data flow through the implemented FlexPath NP demonstrator system as implemented on the ML410 development board. The darker green colored modules in the hardware processing pipelines have been implemented by me or by students under my supervision and I have added top-level block diagrams for these modules in the Appendix section.

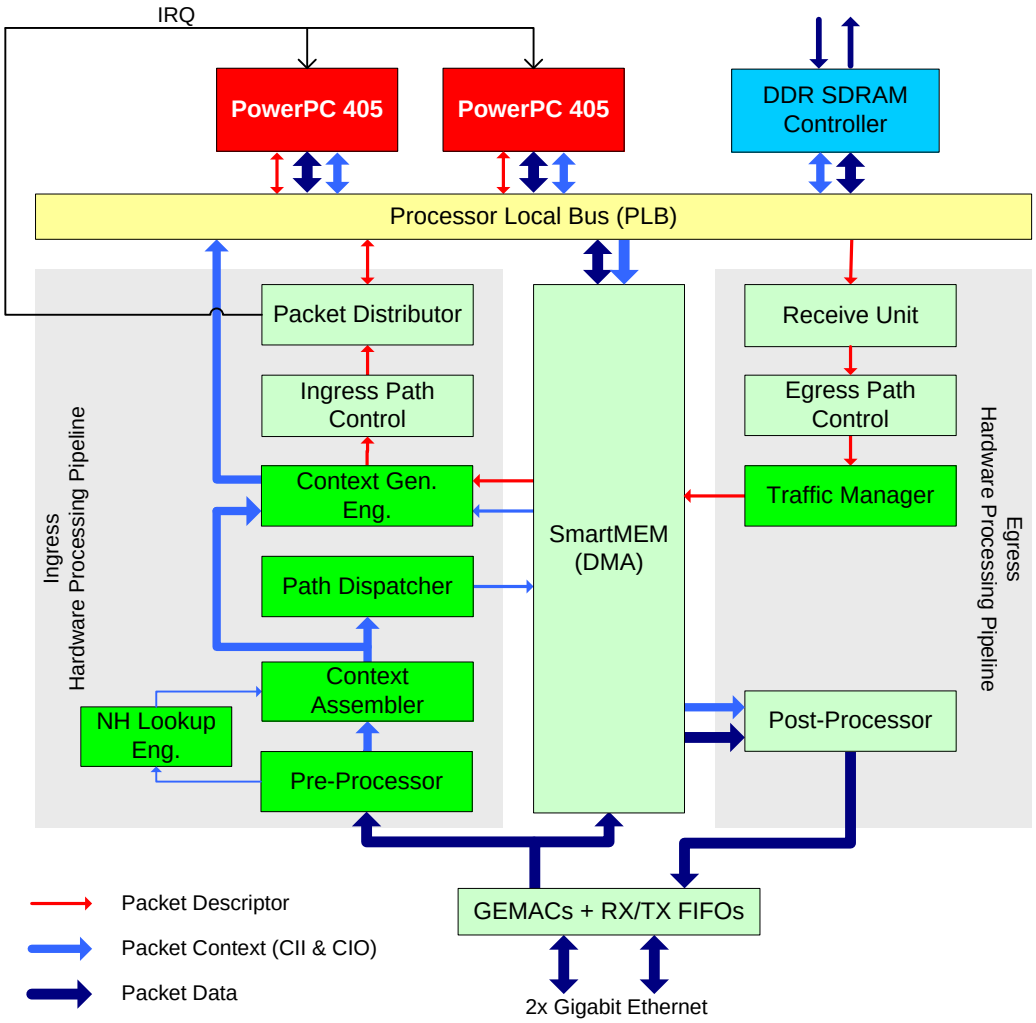


Figure 65: Building Blocks and Data Flow through FlexPath NP Demonstrator

The **Pre-Processor** is the first element in the Ingress Hardware Processing Pipeline. It performs a round-robin receive port scheduling among the two attached MAC ports. The extraction of relevant header fields happens in real-time as the packet is read out from the MAC receive buffers. If the arriving packet is an IPv4 packet, the next-hop lookup engine is triggered with the destination address. After the packet is completely received and the integrity checks (packet length, IP checksum) are passed, the Context Assembler as next downstream element is triggered.

The **Context Assembler** reads out the extracted context fields from the Pre-Processor and consolidates the obtained information into a protocol-independent format referred to as Raw Context (for details refer to Figure 88 in the Appendix). In addition, a next-hop lookup result (hit or miss) has to be synchronized with the Raw Context of the current packet. The Raw Context is forwarded to the Path Dispatcher and Context Generation Engine for further use.

The functionality of the **Path Dispatcher** has already been extensively discussed in chapter 4 of this dissertation, so I will not repeat this discussion here. A detailed description of the reconfiguration interface, which is provided over the PLB bus, is presented along with the detailed descriptions of the other modules in the Appendix.

After the classification result is determined by the Path Dispatcher, the information is passed on to the SmartMem DMA engine, which may use this information for storing the packet data in different memories. In the implemented version of the FlexPath NP demonstrator, all packets are stored in the central shared DDR SDRAM memory. The SmartMem DMA engine is discussed in detail in Daniel Llorente's dissertation [108], but I have included a brief overview of its top-level components and external interfaces in the Appendix to support understanding the module interactions in the implemented demonstrator system.

The SmartMem delivers the packet descriptor (see Figure 91 in the Appendix) containing the addresses of the memory locations, in which the packet has been stored, along with the classification information obtained from the Path Dispatcher to the **Context Generation Engine**. Depending on the further processing path of the packet, a corresponding Context has to be saved in the main memory depending on the subsequent processing element. If the packet is headed for one of the PowerPC processors, a Context Information Input (CII) will be generated that arranges the extracted packet header fields sorted by their relevance for IP forwarding in a segment at the beginning of the first packet data segment. This data can be read in by the processor in a single cache line transfer and as the fields are already 32 bit aligned, the access efficiency is greater than if masking operations would have to be carried out on the packet data section. In addition, as the Pre-Processor has already executed integrity checks on the packet, the processor doesn't have to check that again. However, if the packet is headed for AutoRoute, a Context Information Output (CIO) is necessary that contains the Assembler-like instructions for the Post-Processor that trigger replacement of the MAC addresses, TTL decrement and IP checksum recalculation. The required instructions that will be copied into the Context and the sequence, in which Raw Context words are copied, are fully reconfigurable through another PLB slave attachment. More detailed descriptions for the Context Generation Engine and the typically used CII/CIO context contents are added in the Appendix.

After the packet context has been stored in SDRAM, only the packet descriptor is forwarded through the remaining modules of the NP. If a unit needs context or packet data information, it can be obtained from the shared SDRAM. The Ingress Path Control tags the packet descriptor with a continuous flow-specific sequence number. As all arriving packets traverse the ingress pipeline in a strictly deterministic fashion without being able to pass each other, the tagging records the precise arrival sequence of the packets in the NP system. Further details about the Path Control are found in Michael Meitinger's dissertation [107].

The Packet Distributor is the final element in the ingress hardware processing pipeline and provides queuing and interrupt functions to allow an efficient distribution of packets to their respective processing elements. Sixteen queues have been provisioned for CPU-bound traffic, each queue holding up to 16 packet descriptors. By configuration in the interrupt controller, each queue may be associated with either processor or packets may be sprayed by interrupting both processors, while packet descriptors are present in the queue. An additional queue is provisioned for AutoRoute traffic and Discard packets, from which packet descriptors are written over the PLB bus interface to either the Receive Unit (AutoRoute) or the SmartMem (silent discards). A detailed discussion of the Packet Distributor can be found in Michael Meitinger's dissertation [107].

In the FlexPath NP demonstrator, the Network Processing Complex consists of the two PowerPC cores and an AutoRoute path. The first PowerPC core, which is used as a plain data plane processor, runs the IP stack with IPv4 forwarding code. By means of compiler flags, the stack can be configured to either use the FlexPath-specific hardware-offload features by using CII and / or CIO information for the forwarding or process the packets by accessing the packet data. The second PowerPC shares the same IP forwarding functionality, but it also takes over Control Plane functions. After system startup, it configures the Path Dispatcher, Context Generation Engine and Packet Distributor. If an active load balancing strategy is chosen for the actual scenario (e.g. AHH or HLU), it periodically extracts load measurements and updates the load balancing tables in the Path Dispatcher. Even in static load assignment, the Control Plane processor regularly updates load figures on the LCD-display of the ML410 extension board (see Figure 64). Due to the additional functionality of the Control Plane software, the forwarding performance of the second CPU is slightly smaller than that of the data plane processor. The IPsec stack functionality, which was implemented in addition to the IP forwarding functions (see [106] and section 5.3.1) had to be removed, as the en- and decryption functions work on the 64 byte segmentation of the previous Buffer Manager version and could not be updated to the SmartMem memory management during the final months of the FlexPath NP project. Therefore, the final measurements can only be performed with QoS-aware IP forwarding.

Once the packets have passed the Network Processing Complex, the Packet Descriptors are sent to the Receive Unit as first element in the Egress Hardware Processing Pipeline. The Receive Unit is essentially a small FIFO that may provide a backpressure towards the PLB interface and re-serializes the flow of packets through the Egress Pipeline.

The next function in the egress side of the NP is the Egress Path Control. Here, the sequence numbers in the packet descriptors are checked on correct transmit sequence, and out-of-order packet descriptors are queued in reordering queues to restore correct packet order. More information on the Path Control can be found in [107].

Next, packets are forwarded to the **Traffic Manager**, which in the demonstrator supports two queues per physical port (high and low priority) that resolve output port contention. The queues can hold up to 128 packet descriptors and the Traffic Manager performs a strict priority-based round-robin scheduling and traffic policing at 1 Gbit/s per port. A more detailed description of the Traffic Manager is available in the Appendix section.

As packet descriptors are scheduled for retransmission from the NP, they are again passed to the SmartMem DMA engine. The packet data and the (optional) CIO information for the Post-Processor is read from the SDRAM memory and forwarded to the Post-Processor.

The Post-Processor is able to perform basic packet modifications like field substitutions, insertions, deletions and TTL decrement and IP checksum calculation operations. The modifications are supplied as CIO information, which may be generated either by the Context Generation Engine (as in case for AutoRoute packets) or the PowerPC processors (e.g. as an offload of tunnel header insertions). More information about the Post-Processor may also be found in Michael Meitinger's dissertation [107].

Table 21: FPGA Synthesis Results of Combined FlexPath / SmartMem Demonstrator System

Resource Type	*Resource Quantity*
FPGA Slices	19,391 of 25,280 (76.35%)
Slice Flip-Flops	17,573 of 50,560 (34.76%)
Slice LUTs	31,319 of 50,560 (61.94%)
FPGA BlockRAM memories	124 of 232 (53.45%)
PPC405 Hard Macros	2 of 2 (100.00%)
1 Gbit EMAC Hard Macros	2 of 2 (100.00%)
Critical Path	11.428 ns (i.e. 87.507 MHz)

The before-mentioned FlexPath-specific hardware modules are implemented in a system with the two PowerPC processor cores, PLB bus, Ethernet MACs and the multi-port memory controller provided in the Xilinx EDK IP library. The synthesis results of the entire system, including debug interfaces are summarized in Table 21. In contrast to the post-synthesis estimate of only 87.5 MHz, the Place and Route tools are able to bring the final design slightly above the 100 MHz margin, so that we are able to run the FlexPath demonstrator at 100 MHz for most of the logic elements, while the PowerPC cores run at 200 MHz.

6.3. Measurement Setup

In order to stimulate the NP system, I generate IP traffic with a Spirent SPT-2000 network tester [103], which is also analyzing the traffic forwarded by the device under test and allows to easily gather crucial performance information like packet loss rates, forwarding speed, packet latencies and jitter. The network tester also features automated test runs based on the RFC 2544 [104] benchmarking suite.

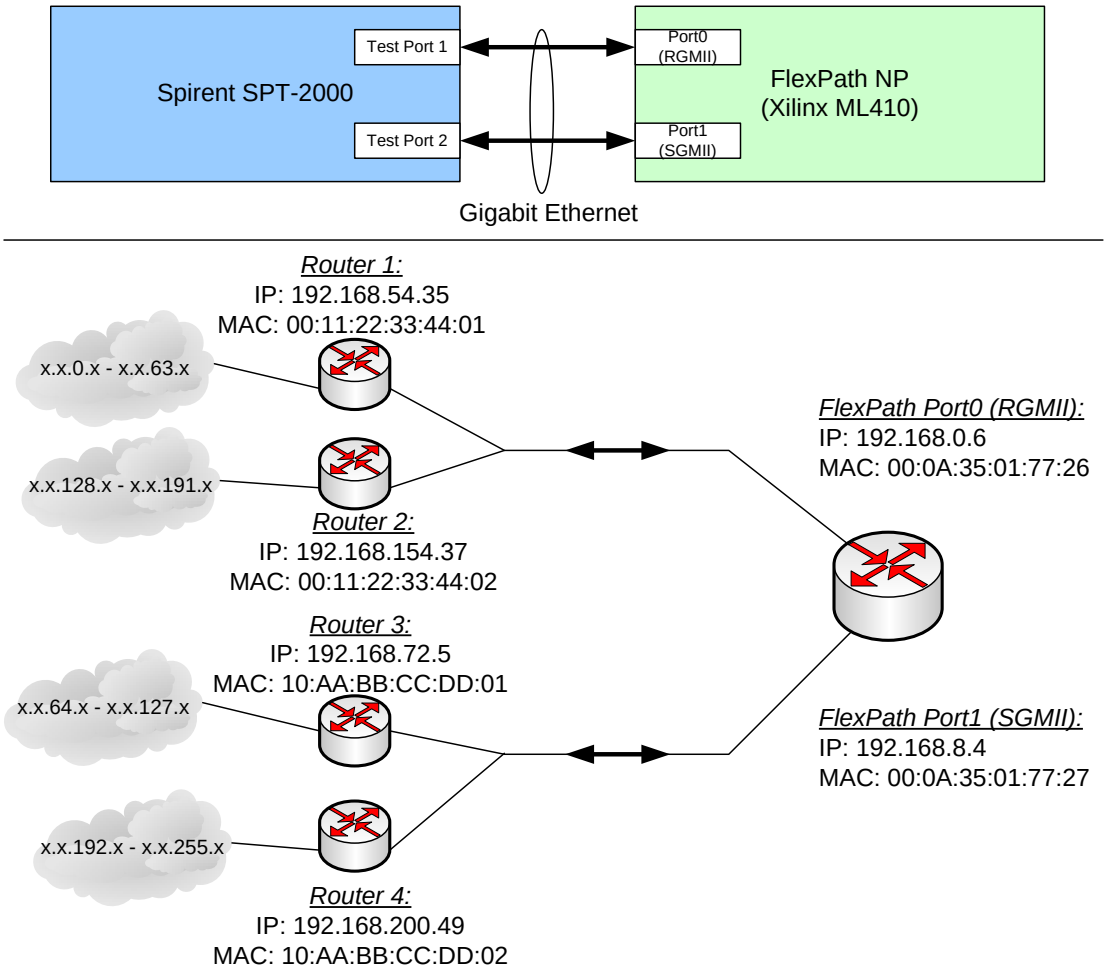


Figure 66: Test and Measurement Setup

The FlexPath NP demonstrator is connected to the network tester using both Ethernet links of the FPGA platform as shown in Figure 66. While each MAC of the FlexPath NP is assigned one IP and MAC address, the Spirent Test Center allows provisioning multiple nodes to be aggregated behind each physical test port interface. In order to obtain a simple measurement scenario, we have added two Routers for each physical test port, which may be reached using distinct MAC and IP addresses. It is important to notice, that the routers and the FlexPath NP prototype do not exchange real routing protocol messages between each other. Instead, the Routers emulated by the Spirent network tester serve simple as sources and destinations of IP packets that are to be forwarded by the FlexPath NP data path.

The routing table in the FlexPath next-hop lookup engine and the algorithm in the IP stack supports this test setup by inspecting only the third octet of the destination IP address as shown in Table 22 instead of performing a longest prefix match operation. Traffic generated by the network tester will always be sent to the FlexPath NP, where the output port will be determined based on the destination address in the IP header. Thus it is possible for a packet to be sent back to the network tester via the same physical link or to be forwarded to the other port depending on the IP destination address. In any case, the FlexPath NP will perform all necessary packet modifications like exchanging the MAC addresses and updating the TTL and checksum values.

Table 22: FlexPath NP Next-Hop Lookup Engine Routing Table

Third Octet	*Egress Port*	*MAC Destination Address*
0 - 63	0 (RGMII)	00:11:22:33:44:01
64-127	1 (SGMII)	10:AA:BB:CC:DD:01
128-191	0 (RGMII)	00:11:22:33:44:02
192-254	1 (SGMII)	10:AA:BB:CC:DD:02
255	None (i.e. test case for next-hop miss)	Result must be determined by software, which would forward these packets to Router 4

The provisioning of two router devices per Spirent test port allows easily implementing traffic patterns that are forwarded on the AutoRoute or CPU paths. By convention, rule bases in the Path Dispatcher can later be configured in a way that all packets destined for Routers 1 and 3 will be forwarded by the processors and packets destined for Routers 2 and 4 will be taking the AutoRoute path through the FlexPath NP.

6.4. Processor-centric Reference Measurements

Figure 67 and Figure 68 show the results of an RFC 2544 throughput test performed on the reference setup, with arriving traffic being directed over either PowerPC (designated as Data Plane or Control Plane) or sprayed over both processors.

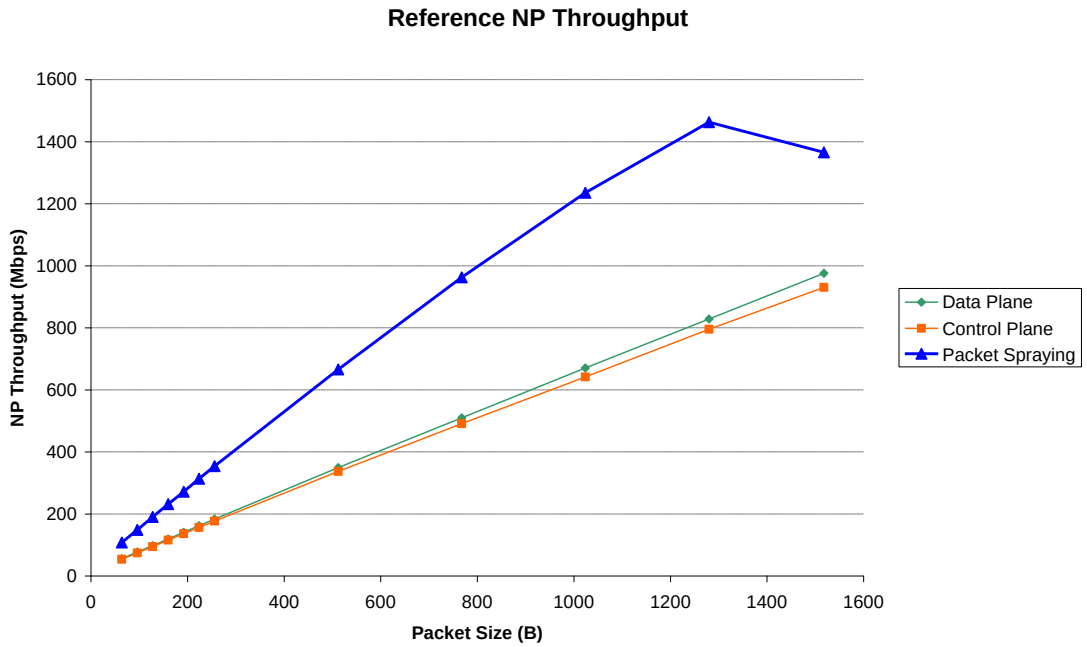


Figure 67: Processor-centric NP Throughput

For only a single CPU in the system, a linear increase of the maximum achievable throughput can be observed, which indicates that the system yields a constant packet forwarding rate. As IP forwarding is not dependent on the length of the individual packets, the constant forwarding rate can be directly related to the processing delay caused in the processor(s). The constant forwarding performance can also be seen well in Figure 68. The Control Plane processor achieves roughly 80 kpps, while the Data Plane Processor is able to forward around 83 kpps. This divergence can be explained by the fact that the code base for the two processors is slightly different and the Control Plane processor is interrupted periodically (every 50 ms) to gather load information of the two cores and display them on the LCD display of the ML410 extension board. As the packet size is increased towards maximum length Ethernet frames with 1518 bytes, the throughput on the shared PLB bus and the memory interface is increasing and in turn the processing performance of the CPUs is slightly decreasing. This observation is in line with the decline in processing performance predicted by the system level simulations in chapter 3.3.2.3 (Figure 28), although the absolute figures (30 kpps in the system simulations) could be increased by a factor of 2.7. This increase can be explained by the differences in the hardware platform, moving from the Virtex-II Pro platform to the Virtex-4 FPGA. The final FlexPath NP demonstrator features a faster DDR-

SDRAM as shared packet memory device. In addition, the SmartMem DMA engine provides better performance compared to the previously used Buffer Manager (see discussions in [108]) and the software stack is more optimized than the LwIP stack with which the system simulation model was calibrated.

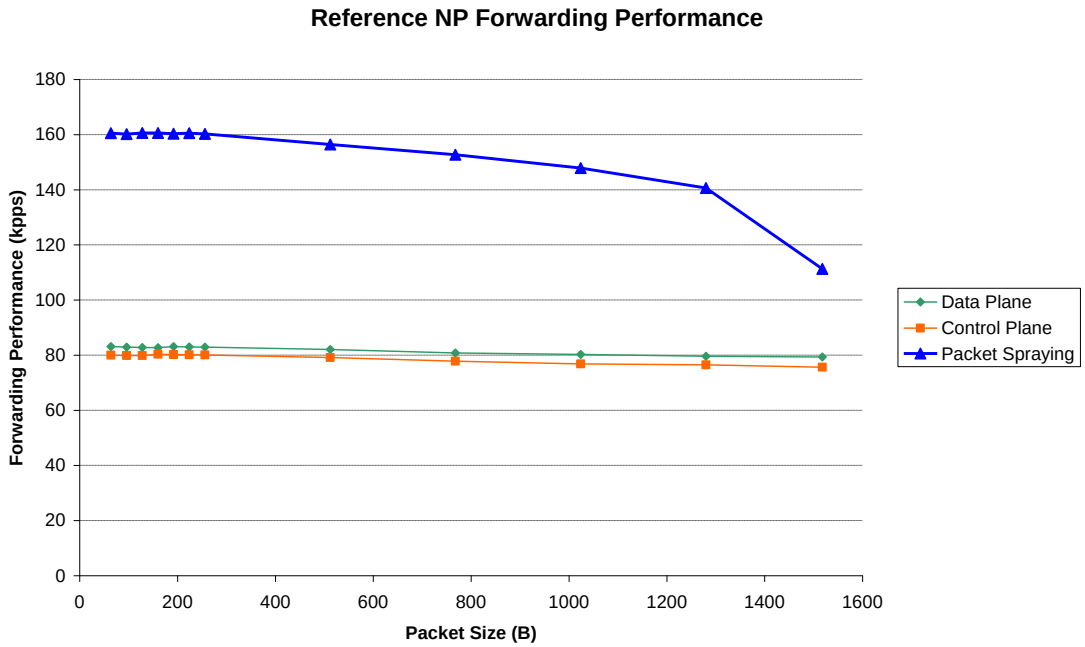


Figure 68: Processor-centric NP Forwarding Rate

When the incoming traffic is sprayed among both PowerPC processors in the system, the forwarding rate is increased to 160 kpps, which is 98.2% of the sum of the individual forwarding rates. As we can see, this figure is even better than the prediction made during the system simulations in chapter 3.3.2.3, which was 91.4%, but again these figures were based on a different hardware platform with a less efficient memory subsystem.

However, when the packet size increases beyond 256 bytes, the resulting higher loads on the bus and memory interfaces reduce the forwarding performance and we reach a maximum throughput of 1460 Mbps for 1280 byte packets. The further sharp decline for the largest 1518 byte packets is due to the reduced bursting effectiveness experienced in the SmartMem DMA operations for packet sizes, which are not powers of two and are discussed in detail in Daniel Llorente's dissertation [108].

6.5. Hardware-offload Aspects of FlexPath NP

In the following chapter, I will successively evaluate the various levels of hardware offload associated with the FlexPath NP concept, and discuss their influence on improving the overall system performance.

6.5.1. Forwarding Performance Using Pre-Processor

The first step in hardware offloading is achieved by moving the ingress packet integrity checks to the Pre-Processor and using the extracted header fields from the input context (CII) in the forwarding software. In this first step, the software still has to perform the necessary egress side packet manipulations directly on the packet data, i.e. the Post-Processor is not yet used for the forwarding task.

The Path Dispatcher is now configured in a way that supports differentiating corrupt packets, control plane packets (e.g. ARP or ICMP) and standard IP forwarding packets. In addition, AutoRoute is only enabled for flows with a valid lookup result and that are destined to a certain IP destination address range. This allows exposing the system to various AutoRoute vs. CPU forwarding shares by using different flows from the Network Tester with varying bit rates. The configured HDGA graph is depicted in Figure 69.

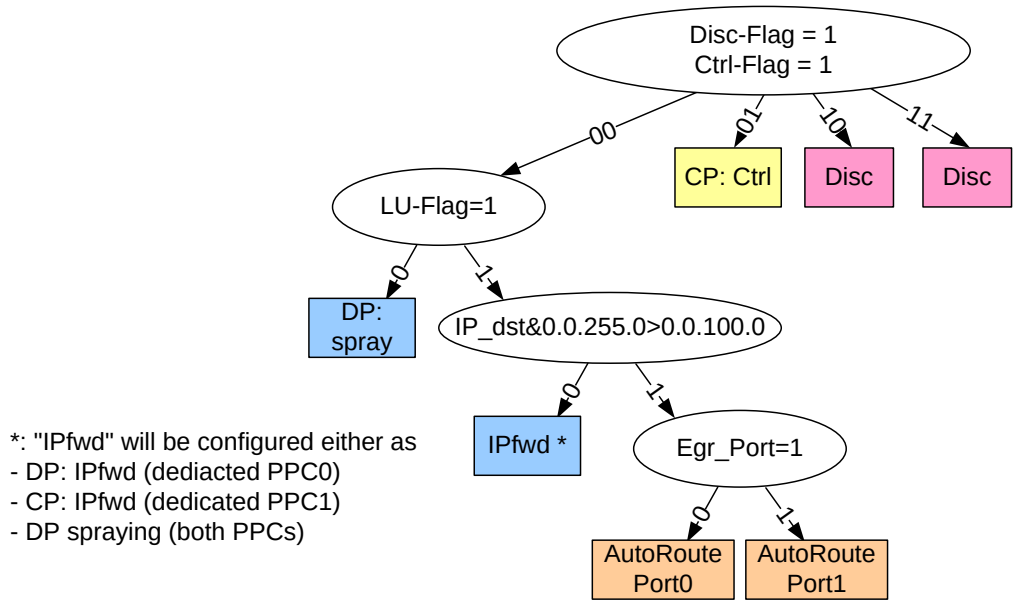


Figure 69: HDGA Graph for Static FlexPath FPGA Measurements

In contrast to the software reference scenario presented in chapter 6.4, the Context Generation Engine has to store the 60 byte CII information according to Figure 93 (in the Appendix section) in the DDR-SDRAM. The processors retrieve this context to determine the appropriate actions, and write the modifications back into the packet data segment of the memory. These additional operations lead to a significant

overhead in required memory access bandwidth, especially for small packets, where the 60 byte CII almost doubles the required space of the packet data itself.

Initially, the network tester generates traffic with equal shares among four connections between the following routers (see Figure 66):
– 00-CPU (loopback on Port0, CPU forwarding): Router2 to Router1
– 01-CPU (switching Port0 to Port1, CPU forwarding): Router1 to Router3
– 10-CPU (switching Port1 to Port0, CPU forwarding): Router4 to Router1
– 11-CPU (loopback on Port1, CPU forwarding): Router4 to Router3

If the traffic volume on each of these connections is kept equal, there will be no output port contention effects by exceeding 1 Gbit/s for the physical interface when scaling the system beyond the gigabit limit. In addition, by combining traffic from both physical interfaces, the latency differences caused by the different MAC cores (the SGMII MAC causes a higher latency than the RGMII MAC) can be averaged out. The results of the RFC 2544 throughput test for this scenario are shown in Figure 70 and Figure 71.

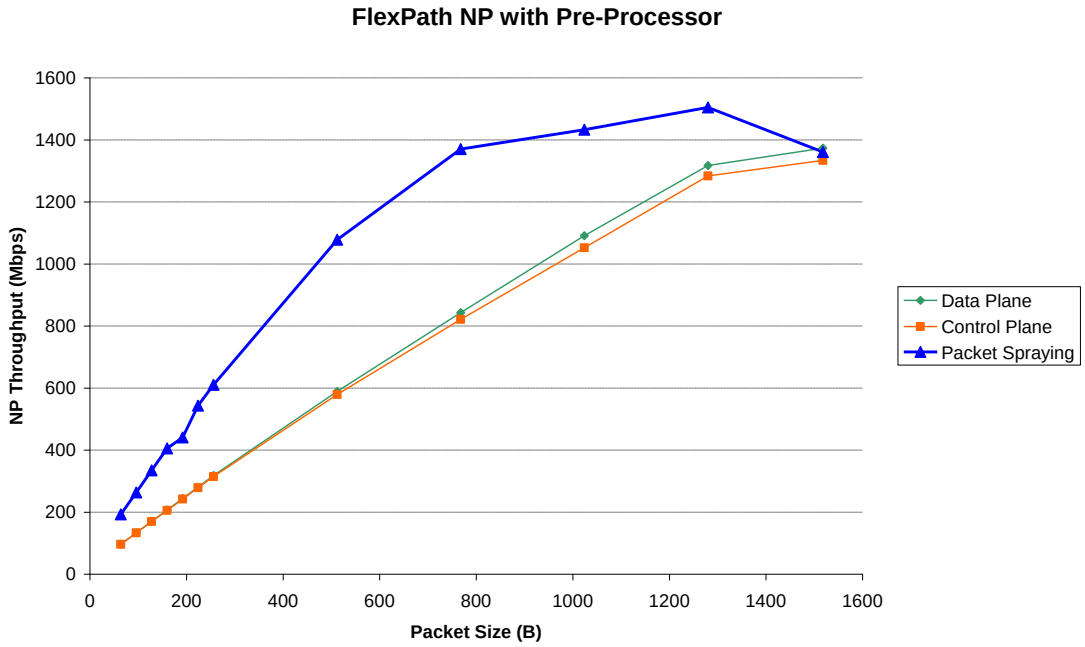


Figure 70: FlexPath NP Throughput using CII (Pre-Processor)

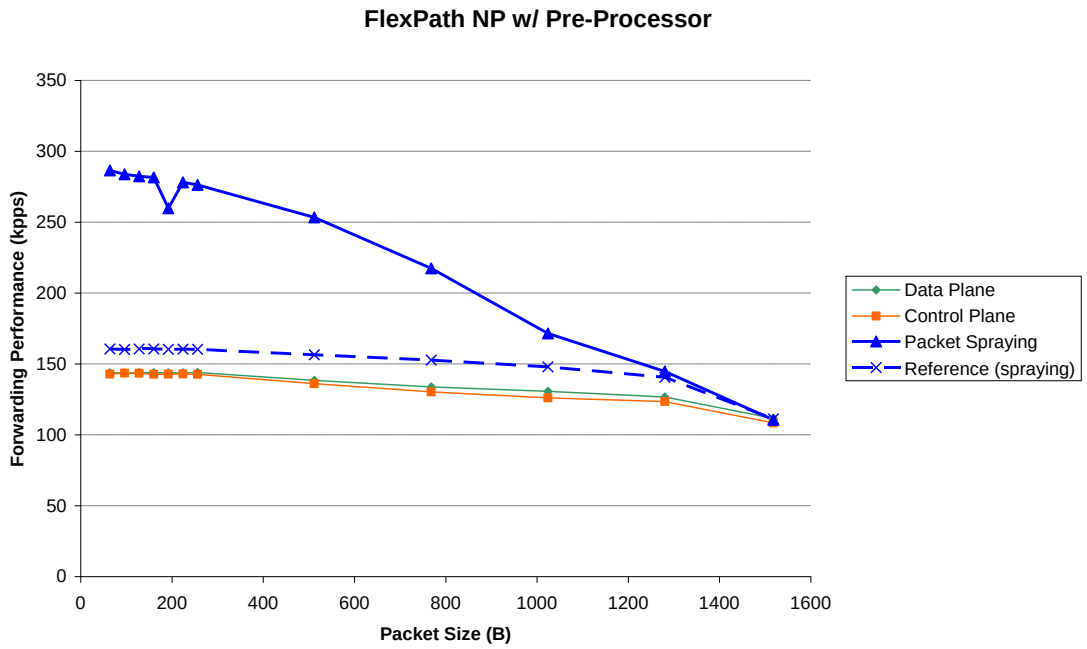


Figure 71: FlexPath NP Forwarding Rate using CII (Pre-Processor)

Offloading the processors by using the results of the pre-processing steps in the FlexPath NP increases the forwarding performance of the single processors by roughly 75% to 144 kpps for the Data Plane processor and 143 kpps for the Control Plane processor. This increase is tripled in comparison to the estimated figures in chapter 3.3.2.4, and this may be explained by the fact that the currently implemented software stack is much smaller than the original lightweight IP stack, on which the application profiling in the system simulation has been based. The amount of instructions necessary to fulfill e.g. the IP header checksum verification may be assumed to be identical across the implementations. However, as the same amount of instructions is offloaded from a smaller overall program, the relative offload gain increases.

Due to the higher forwarding performance provided by the CPUs, the system reaches higher data rates already with smaller packet sizes. In addition, using CII information adds to the overall necessary memory accesses. In consequence, a single processor is now sufficient to exhaust the available memory access bandwidth for packets beyond the 1280 byte packet measurement, and we see the deteriorating performance effect of increasing bus and memory congestion on the forwarding rate for packet sizes between 256 bytes and 1280 bytes, where the single processor performance is reduced almost linearly from 144 kpps to 125 kpps.

Regarding packet spraying among both processors, the initial forwarding rate of 286 kpps (which is 99.6% of the sum of the individual forwarding rates) cannot be maintained through larger packet sizes. The peak bandwidth of the offloaded system is reached for the 1280 byte measurement at 1505 Mbit/s. The convergence

of the measurement results for packet spraying of the offloaded and reference scenarios at 1280 byte and 1518 byte packets supports the assumption that the current demonstrator system is running into a memory access bandwidth bottleneck around 1.5 Gbit/s. Still, when compared with the system simulations in chapter 3.3.2.4, the system throughput could be raised by 50% from 1 Gbit/s, mainly due to the improved timing of the DDR-SDRAM on the Virtex-4 platform and the advanced memory management algorithms implemented in the SmartMem buffer manager.

6.5.2. Forwarding Performance Using Pre- and Post-Processors

The next step in applying hardware-offload capabilities is to remove the necessity for the processors to perform the packet manipulations on the packet data itself, but instead simply generate an appropriate output context (CIO), which will then be executed by the Post-Processor in the NP egress data path pipeline. Figure 72 and Figure 73 show the results of the corresponding RFC 2544 throughput test.

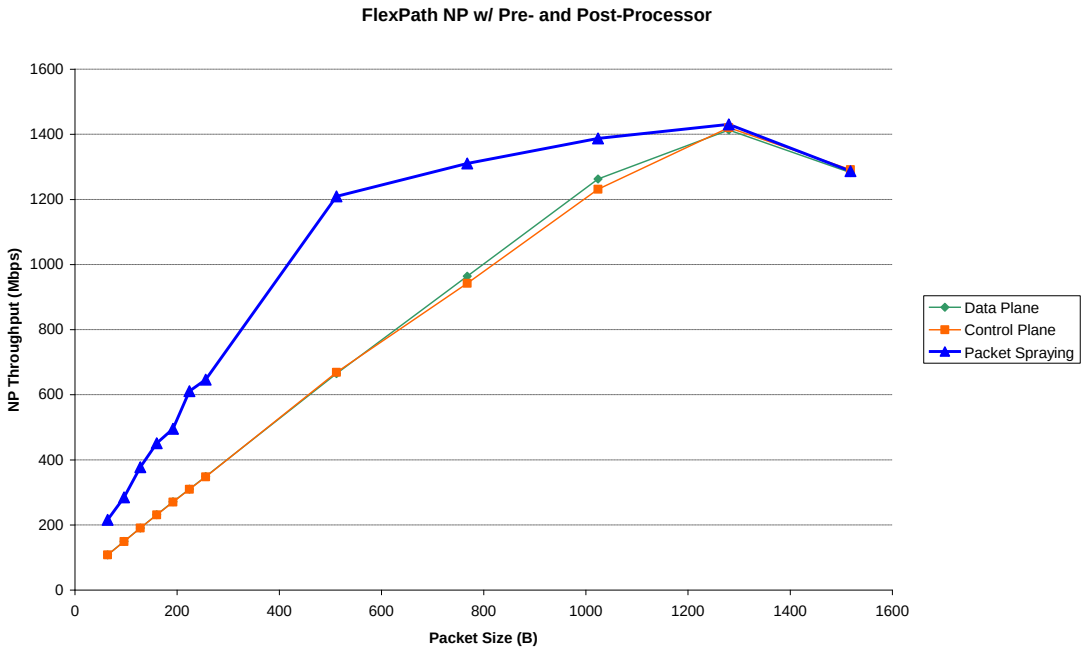


Figure 72: FlexPath NP Throughput using CII and CIO (Pre- and Post-Processor)

The NP throughput chart reveals a very steep increase of the supported bit rate for packet sizes smaller than 512 bytes, where an aggregated throughput of 1.2 Gbit/s is reached. After this point the packet spraying graph shows a significantly smaller increase before reaching the highest throughput for 1280 byte packets at 1430 Mbit/s.

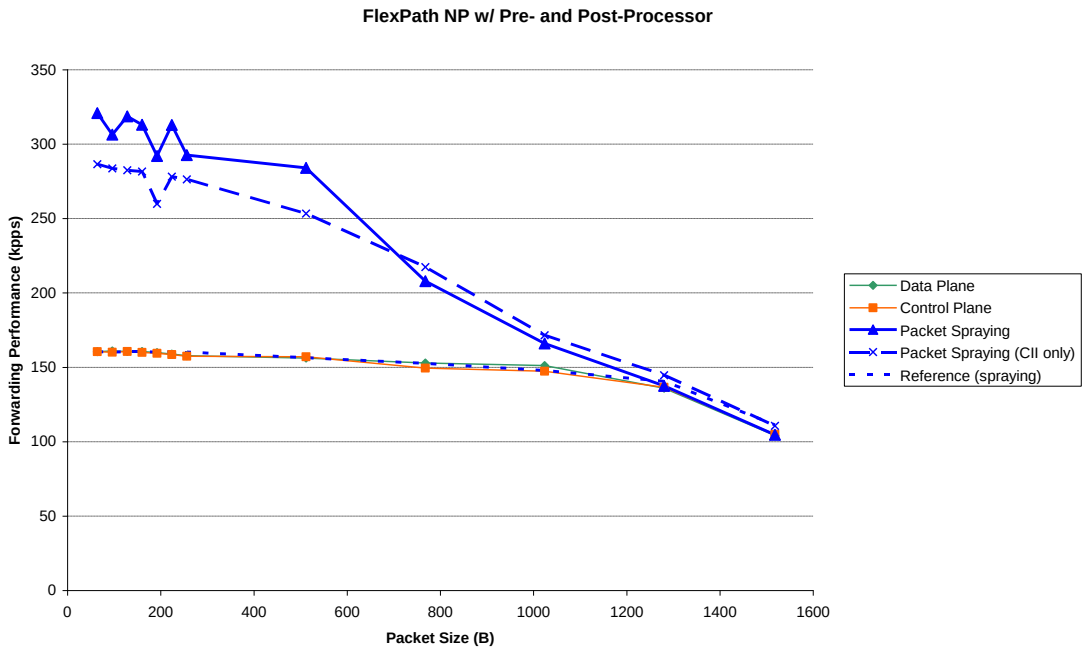


Figure 73: FlexPath NP Forwarding Rate using CII and CIO (Pre- and Post-Processor)

When looking at the forwarding rate comparison in Figure 73, several interesting effects may be seen.

At first, the forwarding performance of a single Control or Data Plane processor (160.5 kpps) matches the forwarding performance for spraying among both processors in the reference scenario described in chapter 6.4. In other words, by using all provided hardware-offload features of a FlexPath NP, the processing performance of each core is effectively doubled. If the system was scaled towards a real multi-core scenario - as in virtually all current commercial NP devices - adding the Pre- and Post-Processor units can help save half of the programmable processor resources on the chip, or double the processing capability of already existing cores to implement more computationally challenging networking applications.

In relation to the previously presented offload of the packet integrity checks to the Pre-Processor unit, the forwarding performance is increased by a further 12%. In line with the estimations presented in chapter 3.3.2.2, the main contribution is coming from the Pre-Processor and CII data structure, such that the effort of implementing a Post-Processor might be avoided, as long as a system designer focuses on a software-centric NP architecture. Of course, the Post-Processor is a necessary pre-requisite for the AutoRoute capability of a FlexPath NP, which will be analyzed later in section 6.5.3.

However, the gains achieved by using CII and CIO for software processing are limited to packets shorter than 512 bytes. Beyond the 768 byte measurement, the

packet spraying performance of the CII-only scenario is slightly higher than that of the combined CII and CIO setup. Both forwarding rates are then linearly decreasing due to the previously identified memory access bandwidth bottleneck. This effect can again be explained with the involved memory bottleneck: if the processors generate the CIO data structure, which is physically stored in the same section as CII in the first segment of the packet, the processors are relieved from reading in the packet header and performing the necessary bit-level manipulations. However, the SmartMem buffer manager has to retrieve the additional context information when retransmitting the packets, and I have already shown in 3.3.2.2 that memory reads from DRAM are more time-consuming than writes. In addition, the output context generated by the processors will only consist of eight words (cf. Figure 94 in the Appendix section), which is also less than what can be transferred over the PLB bus in a maximum length burst of 16 consecutive 64 bit doublewords. In summary, it can be stated that while using the Post-Processor for software-based IP forwarding brings a little relief for the processing resources, the overheads associated with the additional CIO data structure limit the overall system performance, if a memory bottleneck is present.

6.5.3. Forwarding Performance Using AutoRoute

Figure 74 and Figure 75 illustrate the results of the RFC 2544 throughput test for AutoRoute, which is an alternative for software-based IP forwarding in FlexPath NP. The network tester is again configured to mix four different flows, in order to generate several AutoRoute connections with balanced MAC latency and avoiding output port contention. The resulting flows are resembling the CPU-centric scenario used before, replacing destination routers 1 and 3 by 2 and 4 in order to have them routed via AutoRoute by the Path Dispatcher rule base shown in Figure 69.

- 00-AR (loopback on Port0, AutoRoute): Router1 to Router2
- 01-AR (switching Port0 to Port1, AutoRoute): Router2 to Router4
- 10-AR (switching Port1 to Port0, AutoRoute): Router3 to Router2
- 11-AR (loopback on Port1, AutoRoute): Router3 to Router4

As AutoRoute packets are forwarded exclusively by hardware units, which are implemented as ingress and egress data path pipelines of the NP, the performance is dominated by the DMA times consumed in the SmartMem buffer manager during reception and retransmission and for storing the CIO context (cf. Figure 94 in the Appendix section) by the Context Generation Engine. In addition, the packets may experience queuing delays in the Traffic Manager, if there is output port contention. Pre- and Post-Processor work on the packet data on-the-fly and add only a few cycles of latency to the packet. Context Assembler, Path Dispatcher and Context Generation Engine each finish their work within a minimum frame time in order to meet the real-time requirements of the data path pipeline.

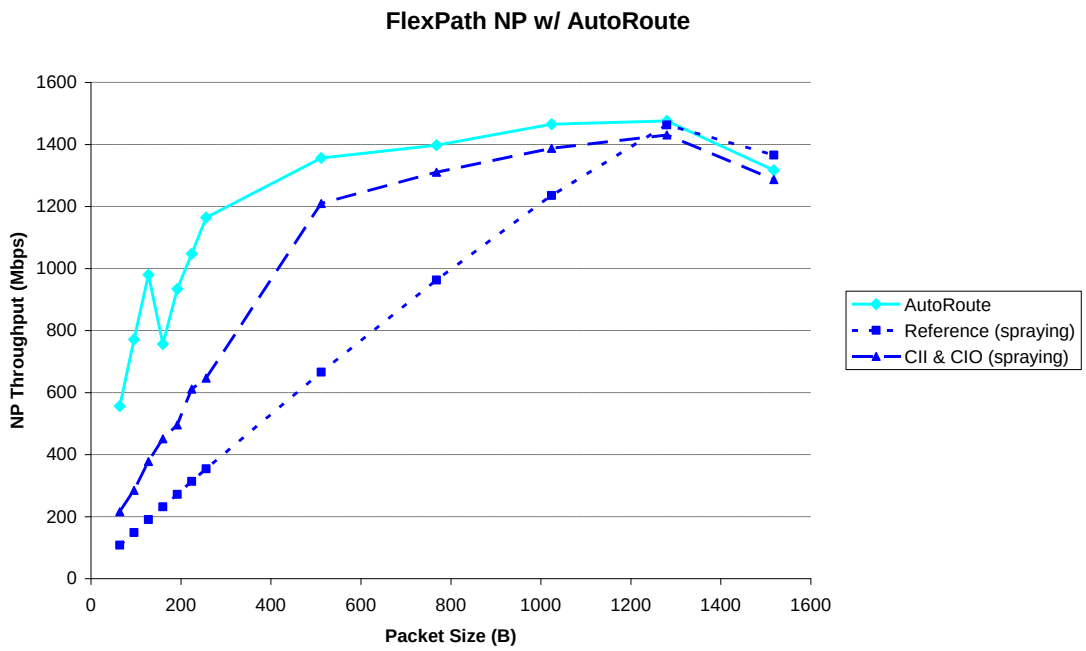


Figure 74: AutoRoute Throughput

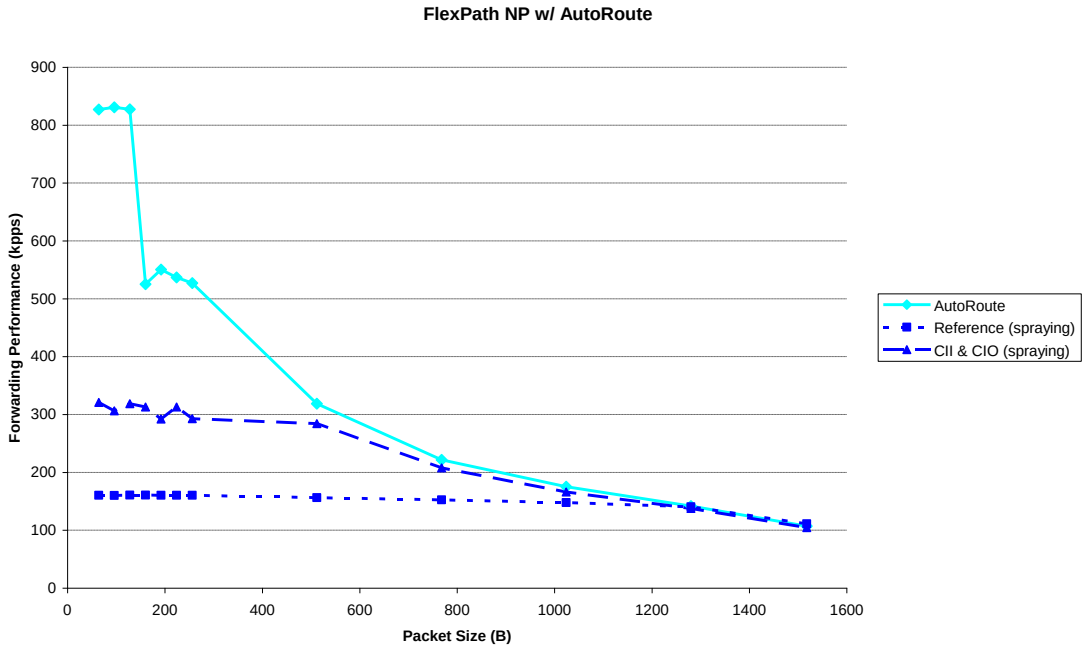


Figure 75: AutoRoute Forwarding Performance

The packet forwarding rates are almost constant at around 830 kpps for 64, 96 and 128 byte packets, which can be transferred between the memory and the SmartMem in a single burst transfer. Although the length of the individual burst transfers increases among these three packet sizes, the delay is dominated by constant overheads like bus arbitration and memory commands (e.g. RAS/CAS). A steep degradation of the performance can again be observed for the 160 to 256

byte packets, which require two burst transfers. A more detailed analysis of these effects can be found in chapter 5.4 of Daniel Llorente's dissertation [108].

Beyond the 512 byte packet measurement, the curves of the AutoRoute and processor-based forwarding performance converge on the previously presented memory bottleneck.

Considering the achievable data rates (Figure 74), it is interesting to observe that AutoRoute performs better than the two processors in the FlexPath scenario. This difference, which is around 80 Mbps for 768 and 1024 byte packets, 46 Mbps for the 1280 byte packets and 29 Mbps for the 1518 byte packets, can be explained with the absence of additional memory accesses by the two PowerPC processors. In case of the 1518 byte packets, forwarding by the software reference scenario (i.e. without context!) performs best with 1366 Mbps, followed by AutoRoute with 1316 Mbps and the FlexPath software with 1287 Mbps.

6.5.4. Packet Latencies

All previously presented measurements were RFC 2544 throughput tests applied to the different configurations of our FlexPath demonstrator yielding maximum achievable packet forwarding rates and throughput. Another important characteristic of an NP system is the latency imposed on the packets traversing the system. In addition to the processing delay imposed by the processors and the DMA delays of the SmartMem, queuing can play an important role, especially when the system is approaching the maximum capacity of a critical system resource. This capacity limit might either refer to the maximum memory transfer bandwidth, which is also packet size dependent as shown in Figure 74 (AutoRoute); or a maximum packet rate as in case of the different processor bottlenecks shown in Figure 68, Figure 71 and Figure 73. However, in a real packet forwarding system, packets with all sorts of different packet sizes will be present at the same time and it may happen that on a very small time scale both data rate and packet rate maxima may be exceeded by a sequence of short or long packets, while over a longer timeframe the traffic might still be forwarded in a lossless fashion.

In the following sections, I will therefore present a series of average latency measurements plotted against an increasing amount of traffic for the Reference and FlexPath scenarios described before. As we have seen in previous results, combining the processing power of both PowerPCs with the FlexPath forwarding software exceeds the memory access bandwidth for packets larger than 512 bytes. Therefore, I decided to run these measurements with only the data plane processor forwarding packets (using the reference software from chapter 6.4 or the FlexPath software using both CII and CIO) and AutoRoute in addition to the FlexPath software processing.

The Reference software and FlexPath without AutoRoute measurements were stimulated with the 01-CPU and 10-CPU flows, each carrying 50% of the aggregate traffic and using a simple IMIX [105] packet size distribution, i.e. packet sizes are not any longer uniform but randomly generated with a distribution of 64B:576B:1518B packets of 7:4:1. For the FlexPath measurements with AutoRoute, the 01-AR and 10-AR flows are added, carrying half of the AutoRoute traffic share each, the remainder of the traffic is generated by the 01-CPU and 10-CPU flows.

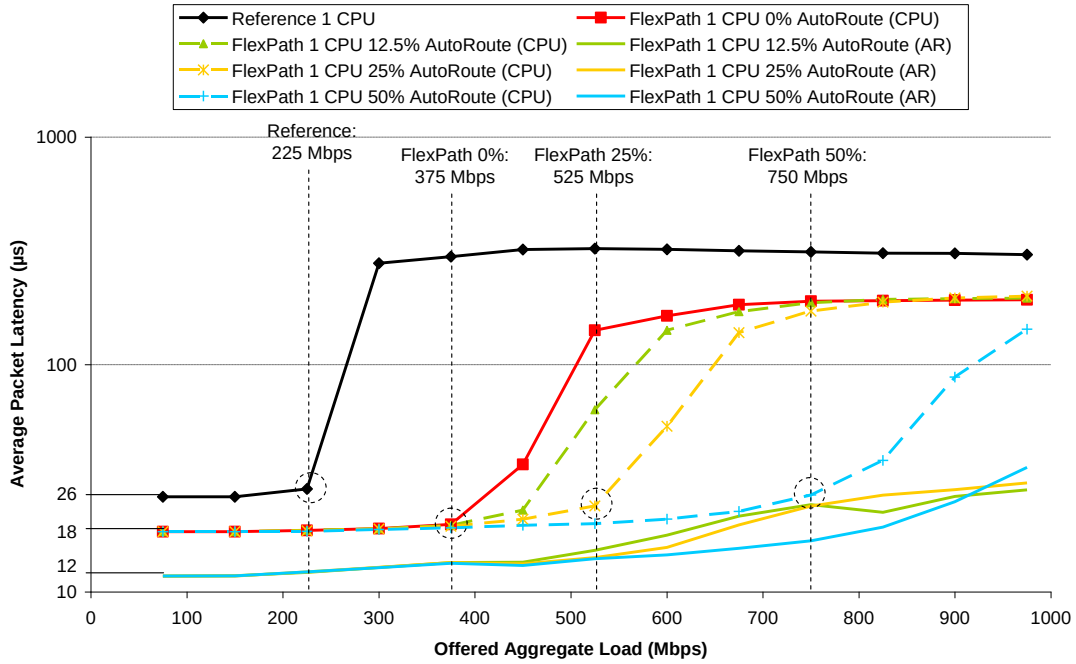


Figure 76: Reference and FlexPath System Latencies for IMIX Traffic (Part I)

Figure 76 shows a lower part of the measurements with aggregate offered traffic load increasing from 75 Mbps in 75 Mbps increments to 975 Mbps. At aggregated loads below 200 Mbps, the system is in underload in all measured scenarios, and the latencies on the three different paths through the NP system are minimal. For the AutoRoute packets the latency is 11.8 µs, the Reference software forwards them in 26.2 µs and the FlexPath software with all possible hardware-offload features accomplishes the task in 18.4 µs. The figures are measured by the network tester from transmission over network tester port to reception on the network tester, i.e. includes all transmission and MAC delays, queuing delays, DMA, latency of the FlexPath NP hardware pipeline and the software processing latency.

As the offered load in the reference measurement exceeds 225 Mbps, the processor is not any longer capable to process the incoming packets. The buffers in the Packet Distributor fill up and the latency levels off at 320 µs. If the traffic load is further increased, the system latency, which is only recorded for the actually forwarded packets, remains constant although an increasing share of packets is discarded by the NP.

A similar effect can be observed for the FlexPath software measurement, however, due to the lower processing latency (remember that FlexPath is able to forward packets at twice the rate as the reference solution!) the cutoff point, where packets start being queued in front of the processor complex is shifted to 375 Mbps. A more gradual increase of the average latency can be observed, visualizing the range, in which the Packet Distributor queues are temporarily holding some descriptors, but are regularly emptied, e.g. when the processor is able to catch up with the load during reception of one or several maximum size packets. The maximum latency of roughly 190 µs is reached only beyond 675 Mbps.

When introducing AutoRoute traffic into the traffic mix, the cutoff points, when the processors starts being overloaded is moved further to the right, as the processing capacity of the NP system is increased. The processing latency of the AutoRoute packets slowly increases with rising amounts of traffic because of congestion on the PLB bus and memory interface. In addition, collisions in the Traffic Manager, when CPU packets and AutoRoute packets reach the egress data path pipeline at the same time also increase the latency. The almost parallel increase of the 50% AutoRoute measurement (light blue curves in Figure 76) beyond 750 Mbps suggests that the system is again approaching the memory access bandwidth bottleneck, which can be seen when comparing the scenarios under higher load as shown in Figure 77.

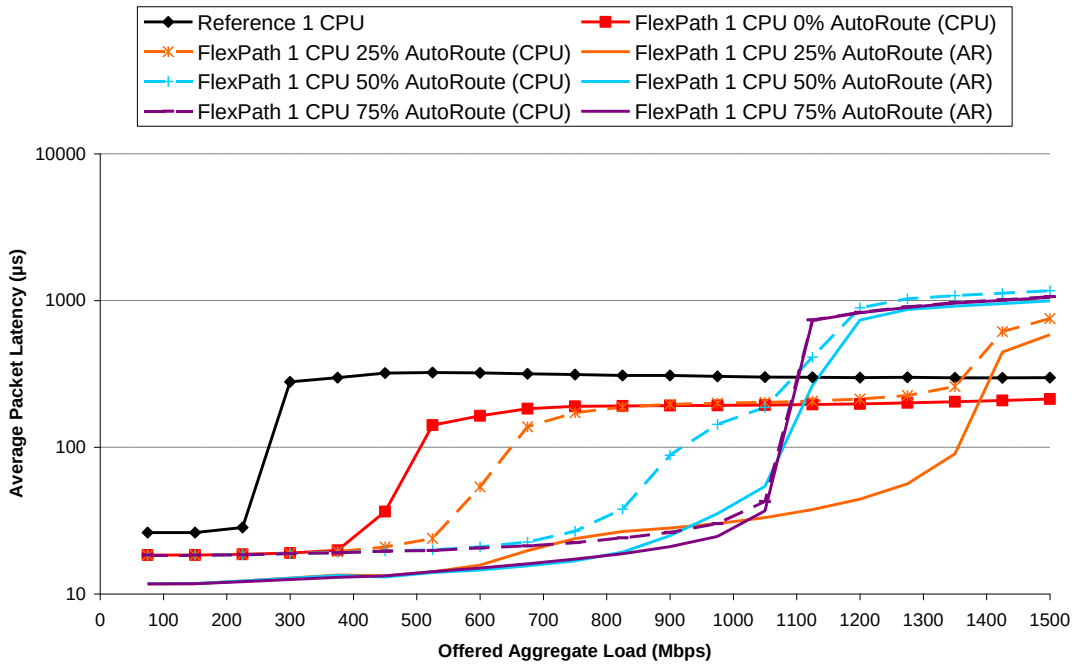


Figure 77: Reference and FlexPath System Latencies for IMIX Traffic (Part II)

For the two measurements with 50% and 75% AutoRoute traffic share, a steep increase of the latency can be observed around 1050 Mbps. Here, the Traffic Manager buffers are suddenly filled, as the packets are received (and forwarded)

faster than the transmit side of the SmartMem can retrieve the packets and packet contexts from memory. As the Traffic Manager acts as output buffer in the system, the queues are much deeper than those in the Packet Distributor, in order to efficiently tackle temporary output port contention and QoS scheduling. In turn, the latency penalty observed for packets that entered an almost full output queue is higher, and we can observe a saturation of the latency around 1100 µs, i.e. 1.1 ms.

The measurement taken for 25% AutoRoute share shows a comparable effect beyond 1350 Mbps. This can be explained by the fact that much more packets are lost on the overloaded CPU path, and thus the overloaded processor works like a policer in front of the egress pipeline of the NP.

In the non-AutoRoute scenarios (red and black curves in Figure 77), the forwarding performance of the single CPU is not sufficient to saturate the memory, and thus the latency of the packets remains almost constant at the level described in Figure 76.

6.5.5. Packet Loss

In order to better evaluate the effects of packet loss in the overloaded queues, Figure 78 plots a "transfer function" of the traffic for the same scenario as described before.

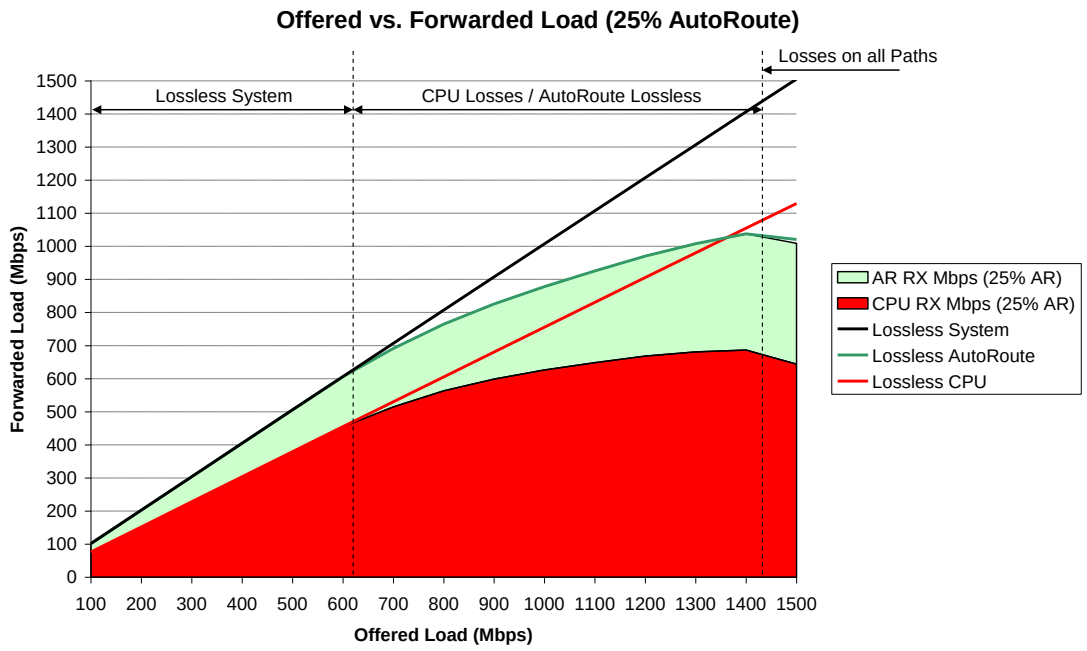


Figure 78: Packet Transfer Function for FlexPath with 25% AutoRoute

Here, the x-axis is again showing the aggregate of all four traffic streams increasing from 100 Mbps to 1500 Mbps. On the y-axis the receive rate at the network tester is shown, differentiated by the path (or flows) that the respective packets should have taken through the FlexPath NP demonstrator. While the system is in underload, the

forwarded load equals the transmitted load; the identity function is also shown as a black line in Figure 78. Just above 600 Mbps, the processor starts being overloaded and packets are lost. This effect is emphasized by the divergence of the red traffic share from the thick red line, which indicates the lossless case for 75% of the offered load (i.e. the share of packets going over the CPU path). As the traffic increases, the CPU is eventually able to forward more traffic, but the gap between the ideal (lossless) case and the actually forwarded traffic amount widens.

In order to visualize the first losses on the AutoRoute path, the thick green line is introduced, which adds 25% of the offered load aggregate (i.e. the AutoRoute traffic share) to the (lossy) CPU forwarding rate. In this case, we observe the first divergence around 1420 Mbps, which is also well in line with the observations in Figure 77, where the 25% AutoRoute curves enter the memory bottleneck saturation around 1400 Mbps. When the memory bottleneck is first reached in the system, the total forwarded load is even reduced by further increasing the input load due to congestion that slows all NP modules accessing the shared memory resources.

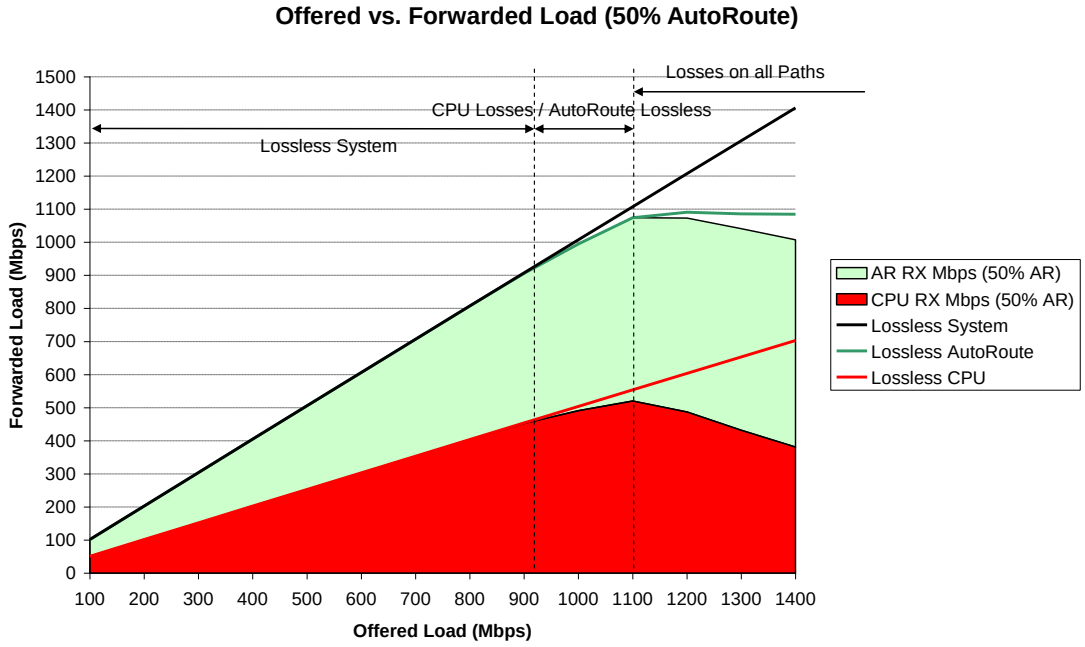


Figure 79: Packet Transfer Function for FlexPath with 50% AutoRoute

I have repeated the same measurements also for the 50% AutoRoute case, where the CPU bottleneck is reached later around 910 Mbps and the memory saturation starts earlier at roughly 1100 Mbps aggregated traffic. The results are shown in Figure 79.

6.6. Load Balancing Algorithms on FlexPath NP

After having thoroughly investigated the individual performance benefits of the various hardware-offload features of a FlexPath NP, I would like to focus on the real-world measurement results for the load balancing schemes discussed in chapter 5. The originally developed IPsec software stack, that was used for profiling in the simulations of [72] and measurements published in [106], has not yet been adapted to the current version of the SmartMem buffer manager with its optimized packet memory organization. Therefore, I will focus in the following only on QoS-aware IP forwarding (using the DSCP value in the IP header to differentiate high priority from best effort traffic classes). As we have no means to replay real-world Internet traces as used for the load balancing simulations, we again have to resort to using traffic generated with the network tester. However, the kind of fixed load, fixed pattern traffic streams used in the previous two chapters would not lead to variations in the traffic volume needed to demonstrate the adaptation effects inherent in each load balancing scheme.

Table 23: Characteristics of Best Effort Traffic Flows

Flow Group	*Packet Size (B)*	*Active Burst Period (ms)*	*Inter Burst Gap (in % of active burst period)*
1	78	100	5
1	160	100	11
1	256	100	19
2	78	80	7
2	160	80	13
2	256	80	9
3	78	140	17
3	160	140	19
3	256	140	27
4	78	180	27
4	160	180	25
4	256	180	23

In order to generate variations in the different logical connections that have to be addressed by the different load balancing techniques, the network tester is configured to generate traffic in a bursty fashion. For a certain period of time, a burst of same length packets are generated for a given period of time and a specified inter-packet gap, followed by a transmission pause, in which no packets are transmitted. By superposing several of these bursts with varying intensity and periods, a quasi-random behavior is achieved. Table 23 lists the timing parameters of four groups of logical flows with three distinct packet sizes per flow that is used for the best effort traffic class generation. Based on these traffic parameters, the

actual burst lengths (in packets) and inter-frame gaps (within a burst) can be derived for different average target bit rates. Due to the gaps within each flow group, short term bit rates are higher than the average bit rate and the hashing-based load balancing techniques are forced to rebalance the load, as the load of a specific hash bundle disappears when the respective flow enters its transmission pause.

The previous chapter has shown that we exceed the capabilities of the memory interface and central interconnect structure when using both PowerPC processors in association with the FlexPath offload features. If a packet size distribution according to the IMIX definition [105] were applied, we would therefore obtain measurement results which are dominated by the performance bottleneck effects of the memory interface and the effectiveness of the traffic distribution among the processing entities would be concealed. Therefore, I chose to reduce the packet sizes to 78 bytes, 160 bytes and 256 bytes (from the original 64, 576 and 1518) while maintaining the 7:4:1 packet shares of the original IMIX. The smallest size packets have been increased from 64 to 78 bytes, so that the Spirent-specific measurement strings fit after a full TCP / UDP packet header. While this modification in the packet size distribution does not resemble the distribution found in the real Internet, I can now guarantee to first run into the processor bottleneck rather than the memory bottleneck.

For the generation of the QoS-marked traffic, I generate two logical connections, but use a constant bit rate for each packet size, in contrast to the bursty pattern presented for the best effort traffic before. This stimulation may be motivated by the fact that in reality, non-best effort traffic is often subject to traffic shaping algorithms, where the packets of each service class are injected with certain predefined timing and bandwidth behavior (e.g. constant bit-rate (CBR)) in order to comply with respective service-level agreements in the actual network.

For all following measurements, 10% of the traffic volume is marked with a non-zero DSCP, and 90% of the traffic volume is generated as bursty best effort traffic with a DSCP of 0x0. The total traffic volume is then increased from low levels until the system resources are fully saturated and packet latency and loss rates are evaluated in a differentiated fashion for each service class.

6.6.1. QoS-aware AutoRoute

The first set of measurements shows the results for a pure AutoRoute scenario, where all packets take the hardware path, but DSCP-marked packets are directed to the dedicated high-priority queues in the Traffic Manager. The corresponding HDGA tree is shown in Figure 80. Although the HDGA tree contains handling rules for differentiated packet spraying they will never actually be used as they refer only to lookup misses in the next-hop lookup engine, and the lookup table covers the entire address space.

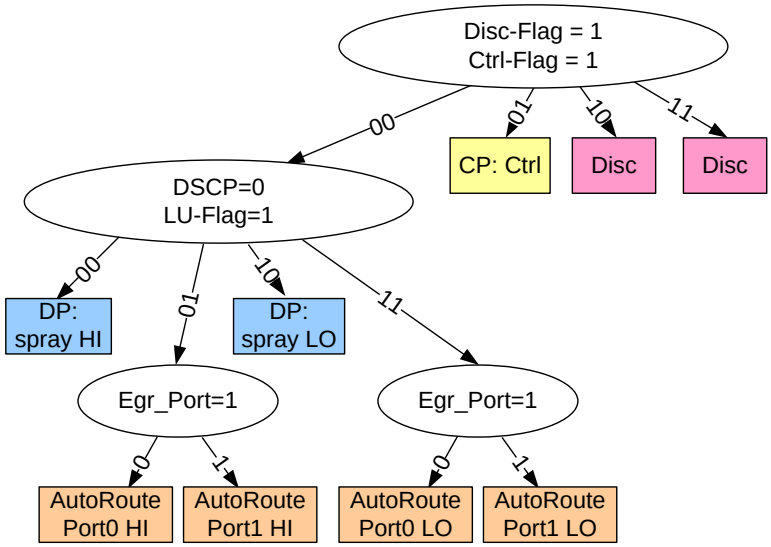


Figure 80: HDGA Decision Graph for FlexPath NP AutoRoute Scenario with QoS Differentiation

As there is no processor involved in AutoRoute forwarding, we can only expect to run into the memory access bandwidth bottleneck previously seen in the RFC2544 throughput tests of our FPGA-based prototyping platform.

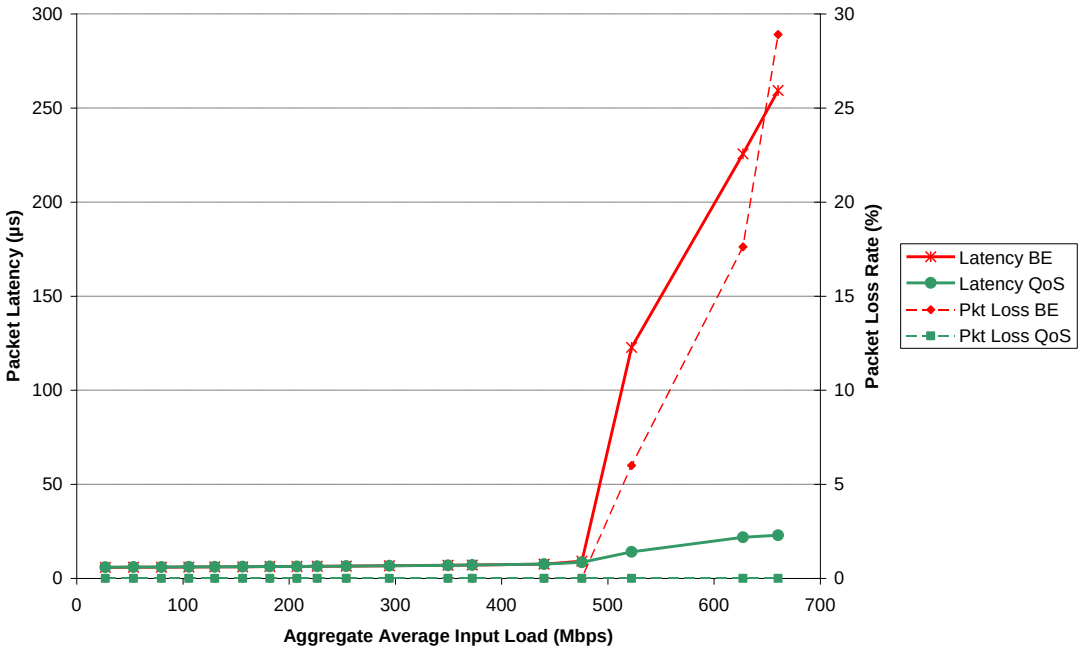


Figure 81: Packet Latency and Loss Rates per Traffic Class for AutoRoute Scenario

When analyzing the measurement results, which are depicted in Figure 81, we see an (almost) identical forwarding latency of 5.9 µs for both traffic classes at 27 Mbps aggregate average input, which is increasing to 7.6 µs at the 440 Mbps measurement. After this point, we are encountering first congestion effects in the egress part of the NP, and the best effort packets suffer from additional output

queuing delay compared to the QoS-marked packet streams. First packet losses can be observed at the 522 Mbps point, where 6% of the best effort packets are lost. Although both packet latency and packet loss increase significantly, the QoS-marked packets can still be forwarded in a lossless fashion and with a latency of less than 23 µs due to the classification in the Path Dispatcher and subsequent assignment to a strictly prioritized queue in the Traffic Manager.

6.6.2. QoS-aware Packet Spraying

In the following, I will reintroduce the processors for the IP forwarding task, and compare the key performance figures for both packet spraying and HLU-based dedicated, flow-specific load assignment. These two schemes have been identified in the simulations laid out in chapter 5, to be the ideal candidates for processing stateful and stateless networking applications respectively. Especially in traffic conditions, where we observe short-term load variations (i.e. bursts), we can expect higher packet loss and latencies for the dedicated assignment performed with HLU compared to packet spraying. The modified HDGA graph for this setup is shown in Figure 82.

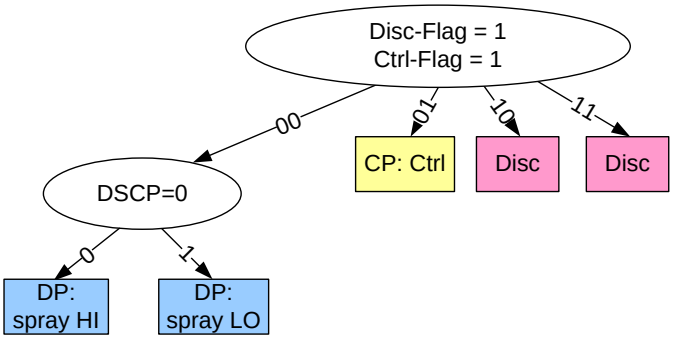


Figure 82: HDGA Decision Graph for FlexPath NP Packet Spraying Scenario with QoS Differentiation

Figure 83 shows the performance differentiated for the two traffic classes while the system is in underload. The packet latency for both traffic types is equally increasing from 11.1 µs to 11.8 µs for the 130 Mbps measurement point. From there on, the best effort packets start experiencing longer delays, as occasionally packets have to wait for higher priority packets to be serviced during bursts that briefly exceed the processing performance of the two PowerPC processors. However, the queues provisioned in the Packet Distributor are dimensioned sufficiently large in order to balance out these bursts.

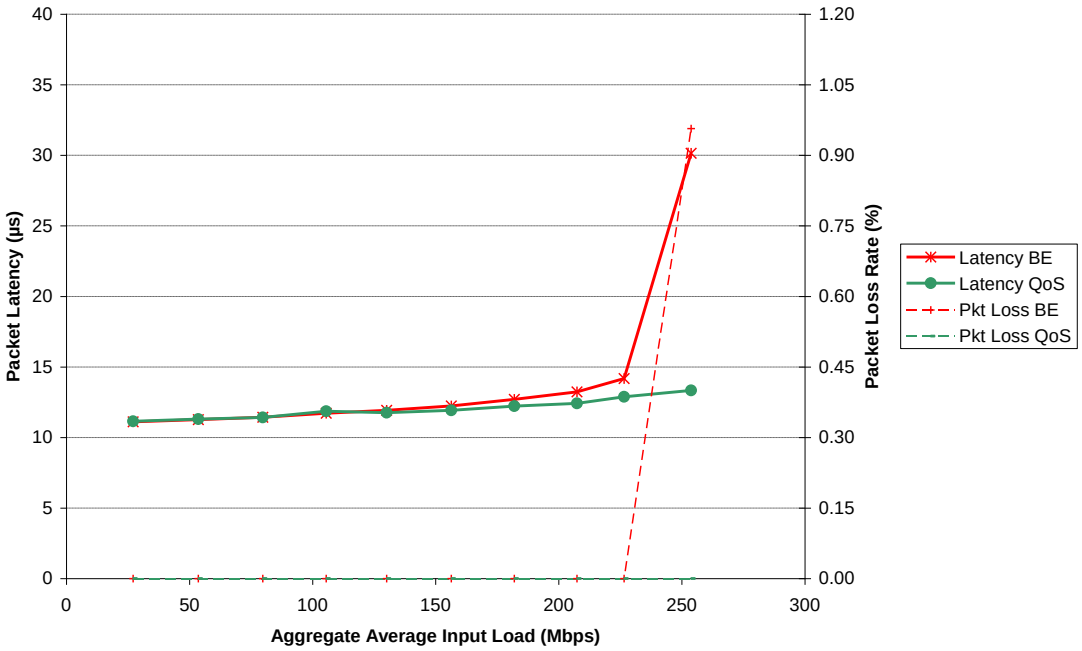


Figure 83: Packet Latency and Loss Rates per Traffic Class for Prioritized Spraying Scenario (Lossless Part)

First packet losses can be observed at the 253 Mbps measurement point, where the latency of the best effort packets rises significantly from 14 µs to 30 µs, and the system drops 0.96% of the best effort packets. Figure 84 shows the system behavior for measurements beyond this point.

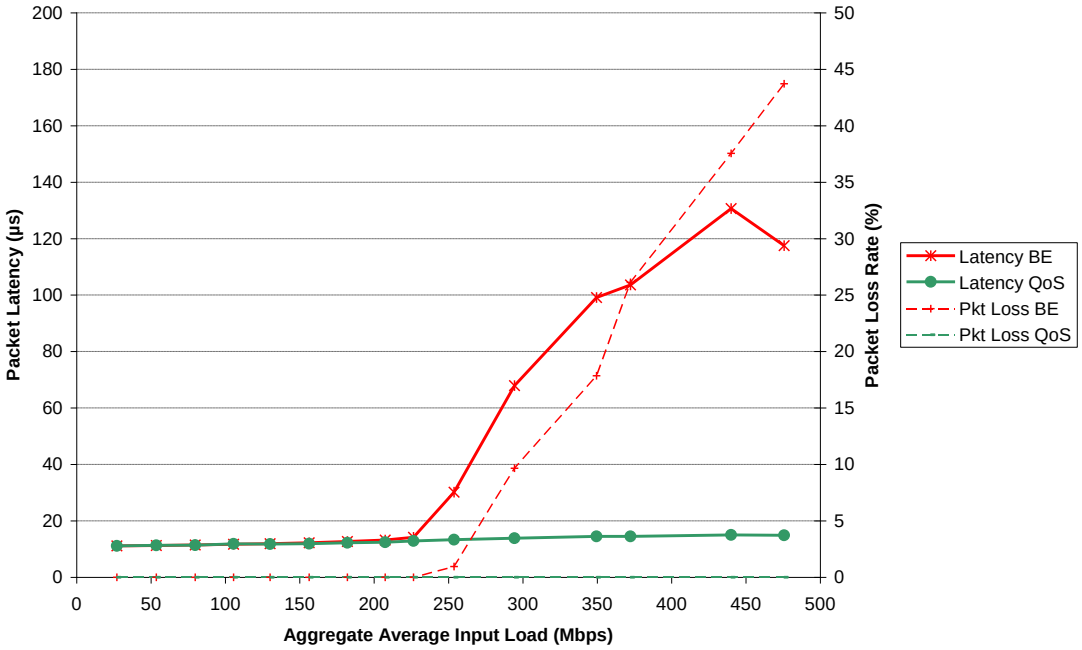


Figure 84: Packet Latency and Loss Rates per Traffic Class for Prioritized Spraying Scenario (Full Range)

As the system is offered increasing traffic on the input interfaces, the PowerPC processors are no longer able to cope with the load and packet descriptors of best effort packets are dropped in the Packet Distributor. Throughout the entire measurement range, the whole offered QoS-marked traffic can be forwarded in a lossless fashion, as a strict priority scheme is applied in the Packet Distributor. The latency of the QoS packets also remains below 15 µs, suggesting that there is (still) no bottleneck in the egress side of the NP; we have previously observed first congestion effects in the AutoRoute scenario beyond 520 Mbps.

6.6.3. Spraying and HLU (S&H)

Finally, I show measurements for a scenario, where the QoS-marked packets are still sprayed among both processors with high priority, but the best effort traffic is now assigned using the HLU load balancing algorithm. This setup effectively represents the S&H load balancing technique. However, the HLU load balancing is applied to the same stateless IP forwarding traffic as packet spraying, as our demonstrator implementation does not support IPsec traffic. The corresponding HDGA graph is depicted in Figure 85.

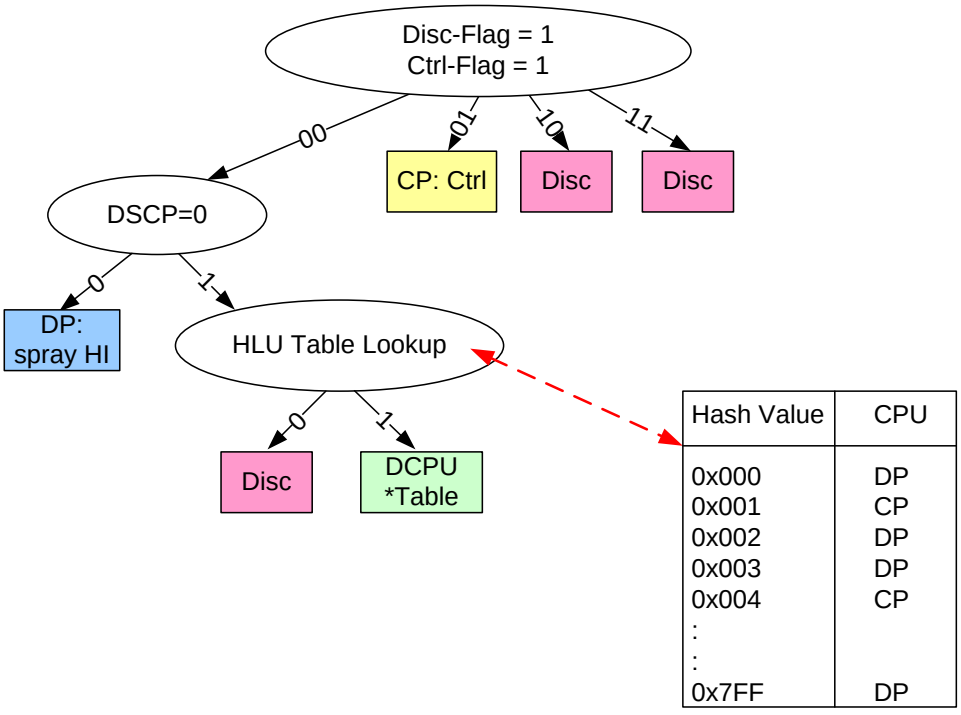


Figure 85: HDGA Decision Graph for FlexPath NP S&H Scenario with QoS Differentiation

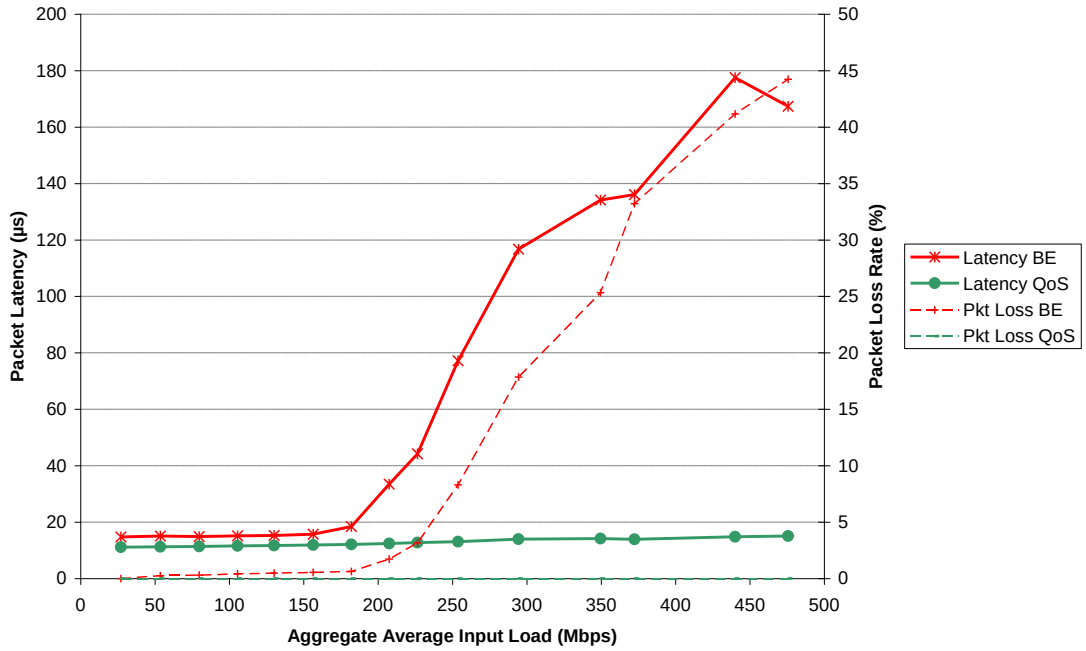


Figure 86: Packet Latency and Loss Rates per Traffic Class for S&H Scenario

In contrast to the scenario with packet spraying, the best effort traffic suffers from a latency penalty of roughly 3 µs also in purely underload situations. This can be explained by the fact that during bursts of packets that belong to the same flow, successive packets are now not served by both processors - leading to a minimal latency - but the processor currently assigned the flow processes all incoming packets in sequence. The other processor might even remain idle during these times. Figure 57 in section 5.3.2 has already predicted this increased forwarding latency of HLU in comparison to packet spraying. Of course, the dedicated mapping of flows to processors is desired when considering stateful networking applications, as a distribution of the packets of the same connection among multiple processors causes significant synchronization overhead and data consistency problems. In turn, the available processor resources may not be maximally utilized for all points in time. In addition to the increased latency of best effort packets, a small amount of packet loss can be observed even for the lowest offered load values. The loss is slowly rising from 0.26% at 54 Mbps to 0.55% at 156 Mbps and is to be compared with the range of minimum packet loss obtained from the HLU simulations in Figure 56.

Beyond the 182 Mbps measurement, the packet loss and latency for best effort flows is increasing much steeper - the processing limit of the two PowerPC processors for this type of traffic is reached. Each individual processor is pushed into overload during the packet bursts, as the load during these peaks can not be distributed over all processors in the system. In comparison, the spraying scenario

started losing packets at 254 Mbps, which is 39% more offered traffic, and shows the "pooling gain" effect that can be exploited by packet spraying.

The packet latency for the QoS-marked packets, which are still forwarded by spraying through a highly prioritized queue in the Packet Distributor, is slowly increasing to 15 µs at 476 Mbps. This latency is even slightly smaller than that observed for the QoS packets in the spraying scenario (Figure 84), as the larger packet loss rates for the best effort traffic class relieve the memory and interconnect resources in the egress part of the NP.

6.7. Conclusions

In this chapter, I have presented a full-featured FlexPath NP prototype demonstrator, which combines all FlexPath NP-specific hardware modules with two PowerPC processors and the SmartMem DMA engine in a single FPGA design. As such, the feasibility and implementability of the previously presented concepts could be proven. Measurements performed with the FPGA demonstrator show the performance of the FlexPath NP concept and allow comparisons with the previously predicted behavior through system simulations.

At first, a reference measurement with the two PowerPC processors was made in order to obtain the baseline performance of a conventional processor cluster architecture in the same system environment as the implemented FlexPath NP. As both the packet processing software and the hardware platform have improved significantly compared to the initial prototype used for calibrating the system-level simulations (cf. section 3.3.2), the current demonstrator achieves almost triple throughput, but the general behavior is still consistent with that predicted by the system simulations.

The hardware-offload possibilities present in a FlexPath NP environment have been explored extensively and it could be shown that the combined offload provided by Pre- and Post-Processing units is able to double the performance of the NP compared to the reference scenario.

The AutoRoute measurements were all memory-constrained, i.e. the full performance of the ingress and egress data processing pipelines could not be exploited due to memory access bandwidth limitations through the PLB bus and the attached shared dynamic memory. Still, for smallest size packets the throughput on the AutoRoute path was measured to be 5.2 times that of a single PowerPC processor. In addition, the reduced processing latency of AutoRoute in comparison to processor-forwarded packets could be shown.

Finally, I have presented measurements that illustrate QoS-aware forwarding and load balancing strategies in a FlexPath NP. While I could not stimulate the prototype with the same backbone traces used in the load balancing simulations of chapter 5.3, the trends observed during the load balancing simulations were confirmed with the available artificial traffic generated by our lab equipment.

The presented measurement results therefore underline the validity and effectiveness of the FlexPath NP concept. In addition, we have identified severe bottlenecks in the central interconnect and shared memory subsystem of the prototype, that would obstruct further scaling of the system towards more processor cores and a higher forwarding performance. However, this also yields

valuable feedback and motivates further research based on the FlexPath NP results about how to overcome these challenges.

7. Conclusion

7.1. Contributions of this Thesis

The current dissertation has presented three major contributions to the state-of-the-art in the network processor field:

FlexPath NP Architecture: Based on observations of current network processor architectures and networking applications, the FlexPath NP architecture has been proposed. The FlexPath NP improves the performance of the network processor by offloading simple, recurring tasks from programmable resources to dedicated hardware units.

- Even when packets are still to be processed by processor cores, the implemented hardware offload in the FPGA-based prototype showed that the plain IP forwarding performance of the available PowerPC processors could be doubled by making use of the Pre- and Post-Processor units in the FlexPath NP.

- In addition, a full hardware offload is proposed for simple applications (AutoRoute) that could be used for significant shares of the overall traffic traversing a network processor, depending on its location and function within the network. The AutoRoute path provides a hardware pipeline architecture that operates at aggregated line speed and it is able to forward packets with significantly reduced latency compared to software-based forwarding.

- In order to utilize the different processing elements in the NP chip efficiently, a packet classification is needed in the ingress data path pipeline of the FlexPath NP architecture. Apart from the before-mentioned CPU processing and AutoRoute paths, arbitrary combinations of dedicated hardware accelerators and software-programmable cores are supported. By moving packet analysis and classification into the ingress hardware data path, the most efficient traversal sequence for each expected packet type can be preconfigured.

Path Dispatcher: The Path Dispatcher is the hardware unit in the ingress data path pipeline that executes the packet classification function in the FlexPath NP architecture under hard real-time constraints. In contrast to packet classification problems regarded in the prior art, the problem faced in the FlexPath NP Path Dispatcher requires a generalization to significantly more header fields.

- The heterogeneous decision graph algorithm (HDGA) has been introduced, which may be used to execute the packet classification function in the Path Dispatcher. HDGA is a new multi-field packet classification technique that combines the advantages of several prior art classification techniques and blends them with some new ideas in order to solve the given classification problem. In a pre-

processing step, the classification rule base is re-formulated with Boolean variables and compacted by logic minimization. The resulting rules are categorized into relatively static and heterogeneous contributions and more frequently changing homogeneous contributions. The static and heterogeneous parts are worked off in HDGA by an optimized decision graph. The homogeneous parts are efficiently handled by table and hash table lookups, which are seamlessly integrated into the decision graph traversal.

– Apart from the concept of HDGA, an area-efficient hardware architecture for implementing the Path Dispatcher has been derived. The presented architecture allows a high degree of flexibility to change protocol details and rule base structure by dynamic updates to configuration memories. In addition, the proposed solution is scalable to larger problem sizes than those discussed within the scope of this work by introducing a pipelined architecture.

– The concept and proposed architecture of the Path Dispatcher allows the system designer to include classification results from (off-chip) classification engines (e.g. TCAM-based NSEs). The external classifiers may communicate with the Path Dispatcher over the same interface as the table or hash table lookup accesses in HDGA.

Load Balancing and QoS:

The concept of assigning arriving packets to different processing paths in the NP system by the Path Dispatcher can also be extended to be used for load balancing. As packets of different traffic classes may be recognized and handled in a differentiated fashion, the Path Dispatcher capabilities may also be used to enable QoS concepts on a chip-wide level.

– It has been shown that the available processor resources in a multi-core NP system can be utilized very efficiently when packet spraying is used as a load balancing strategy. Packet spraying has superior performance in comparison to dedicated load assignment based on flow hashes. However, packet spraying is only suitable for stateless networking applications.

– For stateful traffic shares, hashing-based flow-to-processor mappings as used in state-of-the-art systems are required. After having analyzed the operational characteristics and implementation effort of two current load balancing schemes, HLU (hash lookup) is proposed as another adaptive, hashing-based load balancing technique. HLU produces similar performance as the two reference schemes from the prior art, but at a reduced implementation effort.

– For systems that process both stateful and stateless networking applications, a combination of packet spraying and HLU is discussed. The combined scheme is

referred to as S&H and can be easily deployed in a FlexPath NP system, as the Path Dispatcher can distinguish stateful and stateless networking flows. Since the majority of the traffic belongs to the stateless application class, the additional performance benefits associated with packet spraying dominate the overall system behavior in S&H.

– The Path Dispatcher may be used to prioritize traffic before it reaches the central processor complex. Therefore, QoS features may be implemented more effectively in a FlexPath NP in comparison to conventional NP approaches, where the differentiation has to be performed by software resources. The classification capabilities of the Path Dispatcher give us the opportunity to assign the traffic streams to separated and differently prioritized queues without CPU intervention. Therefore, high-priority traffic may bypass lower-priority packets in the ingress data path before even reaching the central processing elements.

7.2. Outlook to Further Work

The present dissertation has focused mainly on the hardware-offload elements in the ingress data path pipeline of the FlexPath NP architecture. Together with the efforts in the egress data path pipeline and the DMA engine, the conceptual benefits of the FlexPath architecture on an FPGA-based demonstrator have been shown. The performed measurements validate previous results obtained through system-level simulation. Other important aspects of the NP system, like the programmable processor cores, interconnect and memory hierarchy, were just recruited from off-the-shelf IP libraries or constrained by the available (commercial) FPGA development board.

During initial simulations, we predicted memory access bandwidth limitations in the chosen architecture that would eventually limit the achievable throughput of the NP prototype. Through further optimizations in the SmartMem project [108] and by having better physical memory modules on the finally used Virtex-4 development board, the throughput of the system could be raised by 50% from 1 Gbit/s in initial simulations to 1.5 Gbit/s in the prototype implementation. However, for software-based IP forwarding of large packets and the AutoRoute path, the memory interface becomes still saturated before the actual processing elements. In addition, in our demonstrator system with only two PowerPC processors, the shared system bus has to support already nine master and eight slave modules. This architecture is not scalable to a true multi- or even manycore system without running into serious congestion problems.

Recent commercial NP architectures as presented in section 2.1.1.2 (e.g. Cisco [24] or Cavium [26]) feature parallel CPU clusters with 32 cores and more. However, in these systems conventional bus-based architectures have been replaced with crossbar switches. At the same time, academic NP projects (e.g. the GigaNetIC [33]) have considered NoCs (networks-on-chip) and tile-based processing clusters as a more scalable solution than bus-based multi-processor systems.

In the following, I will outline some observations and proposals for further system improvements. They are addressing the identified bottlenecks concerning memory access bandwidth and shared medium interconnect. The demonstrated benefits of the FlexPath NP approach may only be exploited to the full extent, when the identified bottlenecks have been resolved.

– For the system interconnect, NoCs and crossbar switches would certainly be straightforward solutions. However, both alternatives struggle with significant area consumption, as a system is scaled to a larger number of cores. For example, the switch boxes presented in the GigaNetIC consume the same area as a processor cluster with four processing elements. One NoC switch with a locally attached

processor cluster and one Ethernet MAC fill an entire FPGA in the prototyping platform presented in [33].

- First conceptual approaches to resolving the memory access bandwidth bottleneck have been discussed between the FlexPath NP and SmartMem projects, without yet being fully elaborated. The current implementation is already based on interface definitions, which would allow the SmartMem DMA to store the packets (and probably also the packet contexts) in different locations on the chip, depending on the outcome of the Path Dispatcher classification. This allows moving the architecture from a centralized, shared memory system to a distributed memory system. In turn, the performance requirements for every individual memory component can be lowered.

- The network processor architecture would become more scalable by grouping the programmable processing elements and certain dedicated hardware accelerators in clusters of limited size. The overall processing performance is achieved by replicating several smaller-scale clusters in the same fashion as proposed in the GigaNetIC ([33]). In order to address the area consumption and scalability issues of NoC-based systems, hierarchical NoCs [109] are one recently investigated alternative. Cluster-local memories are used to minimize congestion during packet processing and a hierarchically structured interconnect can provide an efficient system-wide communication.

- For processor clusters, the packet data and context can be transferred to the cluster-local SRAMs (referred to as Header Buffer concept in [108]), which can be accessed by the processors through high-capacity crossbar connections. Such SRAMs provide a high memory access bandwidth for irregular access patterns, but they are limited in size. After processing, the packets may be transferred to an external SDRAM, as queuing delays caused by Path Control and Traffic Manager are significantly longer than those in front of the network processor complex and larger memories are required. However, the FIFO behavior of the output queues favor regular memory access patterns, which are well supported by the burst modes in current dynamic memories. AutoRoute packets may of course be stored in the external SDRAM directly as the Packet Descriptor bypasses the network processor complex.

- The Path Dispatcher has to assign the incoming packet stream onto the available processor clusters. An enhanced SmartMem DMA and packet distribution system is required to forward the packets and contexts to the respective processing elements. One open problem would be to find an efficient way of supporting multi-hop processing paths in such a cluster-based, distributed memory NP. Packets traversing several processing elements in different clusters would cause

increased data copy operations between the different local memories and put additional stress onto the system-wide interconnect.

- Another important issue would be how to manage globally shared control plane information, like e.g. the routing table, IPsec databases or connection-specific traffic shaping parameters.

- Load balancing strategies would also have to be revisited, as the cluster regarded in the present thesis is assumed to have a uniform access to a globally shared packet memory. Different communication costs within and in between adjacent processor clusters would certainly influence the choice of suitable load balancing strategies for the proposed NP architecture. Packet spraying has been identified in the current thesis to be a very effective candidate for multi-processor load balancing, as it can exploit a pooling gain effect by distributing the packets over a multitude of parallel processors. However, if the performance constraints on interconnect and memory structures require a migration to a tiled processor cluster structure with local storage and hierarchical interconnects, packet spraying among all processors in the system is no longer feasible. Packet spraying would have to be constrained to be used only within each processor tile, in which typically between four and eight processor cores share a common memory. This would in turn decrease the potential pooling gain, which can be exploited by the spraying technique.

A more detailed elaboration of the before mentioned approaches could not be covered within the scope of the FlexPath NP project and the current dissertation. However, I would consider these aspects as promising starting points for possible future research activities based on the results of the FlexPath NP and SmartMem projects.

Appendix

Implementation Details of selected FlexPath NP-specific Functional Modules

Pre-Processor

As described in chapter 3.2, the Pre-Processor analyzes the incoming packets and extracts important header fields which will be used in further units of the NP. Figure 87 shows the architecture of the Pre-Processor with the main dataflow and the most important external interfaces. The extraction happens in two overlapping functional stages while the packet is streaming in from the receive side interfaces.

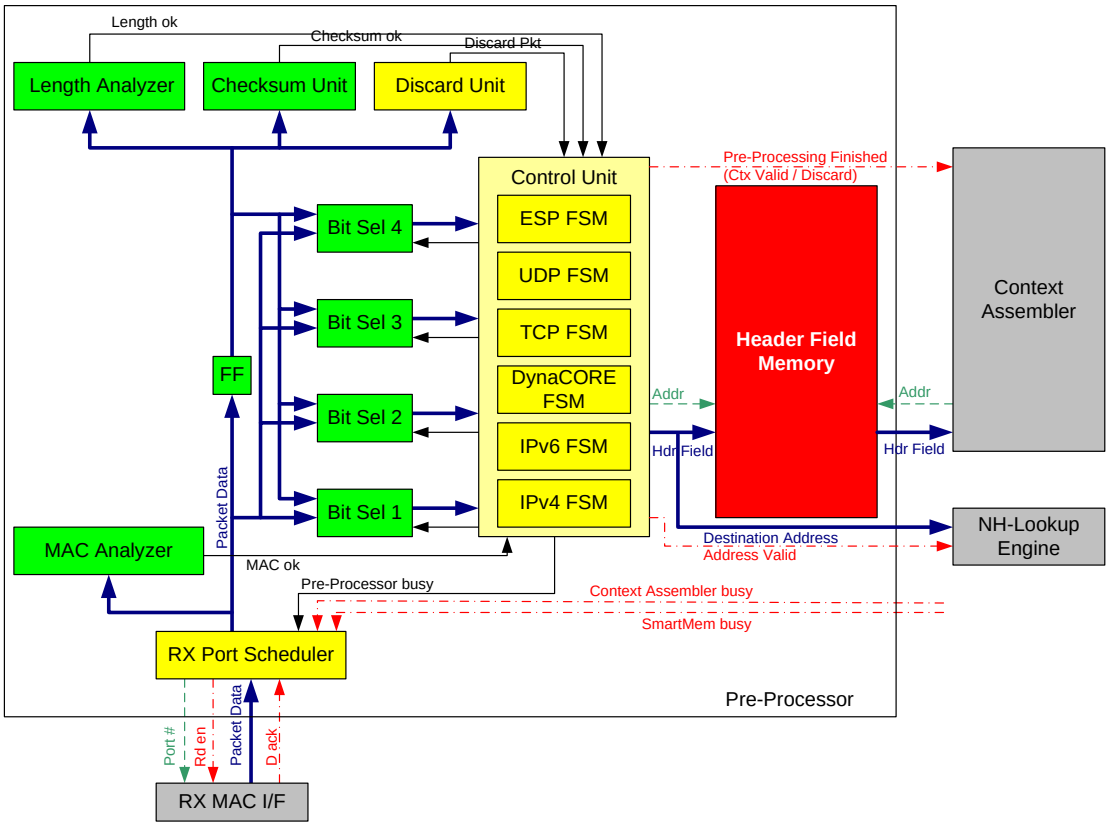


Figure 87: Abstracted Architecture of the Pre-Processor

The first stage consists of an FSM analyzing the MAC layer information of the packet, i.e. frame integrity and L3 protocol field, while the later stage may complete processing of the previous packet.

The second stage consists of a set of FSMs, one for each higher layer protocol, that are activated when the lower protocol level machine detects the corresponding higher layer protocol. In its current implementation, the Pre-Processor supports IPv4, IPv6[3], TCP, UDP, ESP and a proprietary tunneling protocol which was defined

[3] Partial implementation: IPv6 packets will be detected and identified with a special flag in the packet context, but no full parsing and context extraction has been implemented yet.

between the FlexPath group and the DynaCORE group at the university of Lübeck, in order to support a coupling of the FlexPath NP and DynaCORE demonstrators to demonstrate IPsec processing offload to a reconfigurable hardware [91].

The protocol-specific FSMs generate the control signals for four generic bit selectors. These bit selectors consist of multiplexers and registers and store a configurable slice in four bit granularity out of the current 32 bit input word. The results of two adjacent bit selectors can also be combined in order to extract header fields that span two adjacent input words. This is the case for example with the IPv4 source and destination addresses, which are each constructed out of the 16 LSBs of the 7th (8th) word and the 16 MSBs of the 8th (9th) receive word, if the IP packet is transmitted in a standard Ethernet-II frame. The extracted header fields are subsequently stored in a local SRAM to be retrieved later by the Context Assembler unit (see chapter 0). If the arriving packet has an IP header, the destination MAC address is not only extracted and stored in the memory, but also forwarded on a lookup engine interface, that is used to model the behavior of a full-fledged IP next-hop lookup accelerator, e.g. implemented by a network search engine ([53], [54]). A five bit packet ID (PID) is transmitted along with the address to later find synchronization of the extracted header fields and the lookup result in the Context Assembler. In the FPGA demonstrator implementation, the third byte of the IP address is used as an index into a lookup table that allows supporting the simplified routing scenario as described later in chapter 6.3. In addition to the extracted header fields, the Pre-Processor also generates a set of pre-classifying flags that indicate certain properties of the received packet towards downstream elements. Examples for such flags are IPv4 protocol, TCP protocol, UDP protocol, Control Plane protocol, Corrupt frame, etc. These status flags are stored in a single 32 bit word in the header field memory.

While the FSMs in the Control Unit only control the extraction of relevant header fields out of the different protocol headers, three separate units perform verification of the packet length (matching the IP length field to the number of received bytes) and the IP header checksum. The Discard unit may be notified by the MAC Analyzer, Control Unit FSM, Length or Checksum unit in case of a detected error and will subsequently signal a corrupt frame and halts the execution of further packet analysis.

In addition to the pre-processing functions described above, the final implementation of the Pre-Processor also includes a receive port scheduler, that controls the handshaking between the receive side MAC buffers and the ingress hardware processing pipeline. Internal backpressure signals from the Pre-Processor itself, Context Assembler and the SmartMem buffer manager are evaluated and if all attached modules are ready to receive further data, a new packet is read from the receive buffers, iterating through all attached ports in a round-robin fashion.

Once processing of the current packet has finished, the Context Assembler is notified of the completion and is given the PID, with which the corresponding extracted header fields can be retrieved from the Header Field Memory.

Table 24 summarizes the stand-alone synthesis results of the Pre-Processor unit.

Table 24: Stand-alone FPGA Synthesis Results for the Pre-Processor

Resource Type	*Resource Quantity*
FPGA Slices	696 of 25,280 (2.75%)
Slice Flip-Flops	571 of 50,560 (1.13%)
Slice LUTs	1,250 of 50,560 (2.47%)
FPGA BlockRAM memories	1 of 232 (0.43%)
Critical Path	5.774 ns (i.e. 173.178 MHz)

A more detailed description of the Pre-Processor and its implementation can be found in Stefan Lugmair's Master thesis [93].

Context Assembler

The Pre-Processor generates initial packet context information as a protocol stack specific set of extracted header fields. As the protocol stack of a received packet changes, different fields will be extracted along with the uniformly defined flags. Depending on the protocol of the arriving packet, a lookup of the destination IP address may have been started by the Pre-Processor, and if so, the lookup might produce a match or miss in the routing table. The Context Assembler is used to consolidate all this information into a uniform format referred to as Raw Context in the following. Figure 88 shows the contents of the Raw Context, which is generated independently of the protocol stack of the arriving packet. Header fields or flags, which are not present in the current packet, are filled by the Context Assembler with zero values.

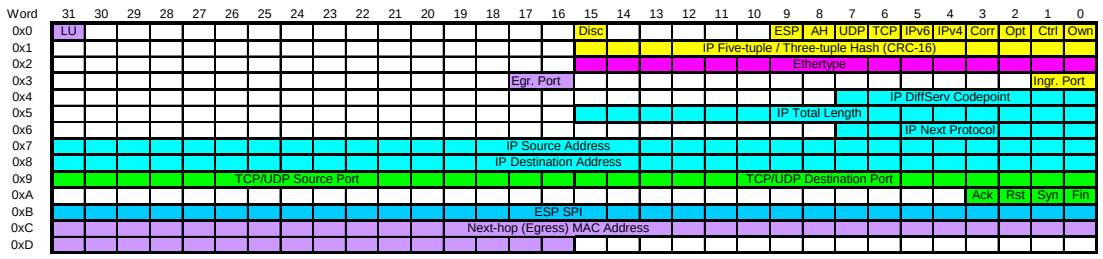


Figure 88: Contents and Layout of Packet Raw Context

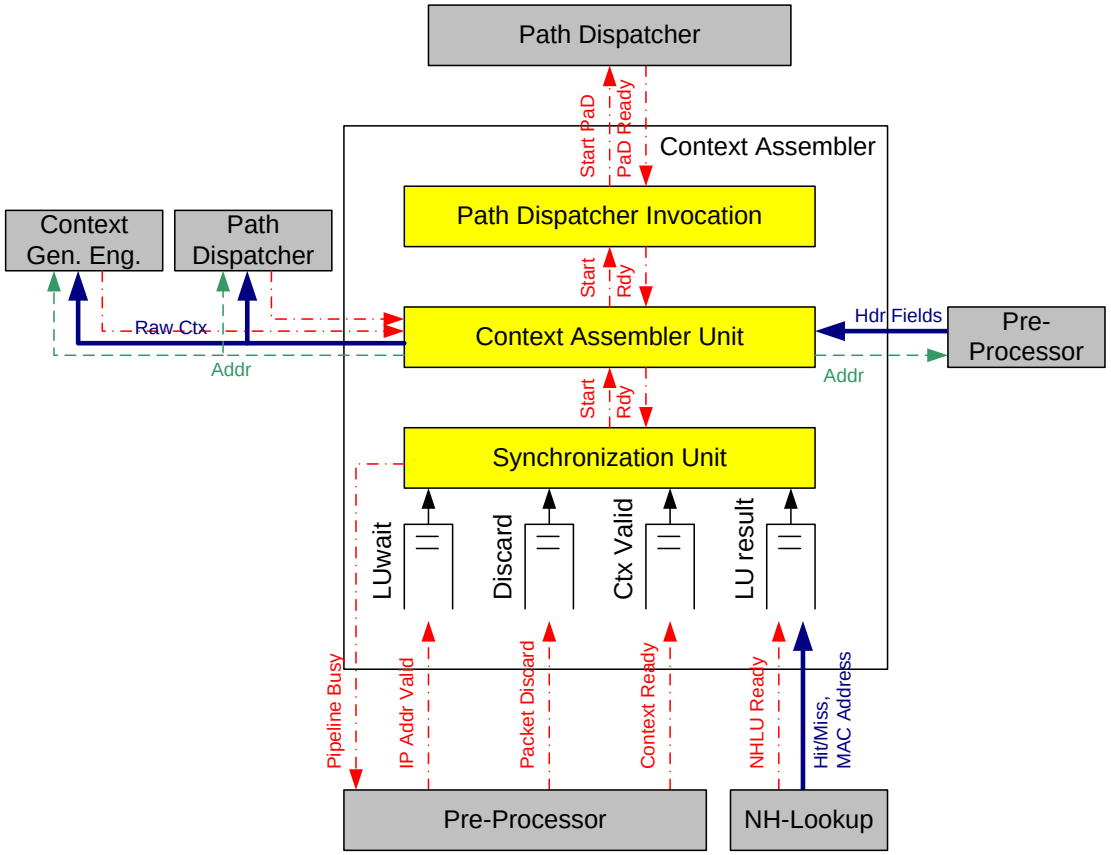


Figure 89: Abstracted Architecture of the Context Assembler

The architecture of the Context Assembler unit is shown in Figure 89. The first task is to synchronize the results generated by the Pre-Processor and next-hop lookup engine. This is implemented by a set of first-word-fall-through FIFOs, which are analyzed by the Synchronization Unit FSM. If a packet contains an IPv4 header, the Pre-Processor initiates a lookup, which is indicated by the IP Address Valid signal in addition to the PID, which is forwarded with every piece of context information or packet descriptor throughout the ingress data path modules. The FIFOs store the respective PIDs. When the packet has been completely received and analyzed, the Pre-Processor sends another transaction consisting of PID and either a Discard or Context Ready signal to the Context Assembler. In case of a corrupt packet (Discard), no header fields and flags are stored for this packet in the Header Field Memory of the Pre-Processor. For packets with a valid IP address, the next-hop lookup engine delivers the lookup result (MAC address of downstream router and output port) along with a hit/miss information.

The Synchronization Unit checks the PIDs of the head of the FIFOs, and when all results for the current packet have arrived, the Context Assembler FSM is triggered.

If the packet is valid, the Context Assembler unit reads the extracted header fields from the Pre-Processor and appends additional fields appropriately in order to

achieve the prescribed Raw Context shown in Figure 88. The Raw Context is saved in local buffers in the Path Dispatcher and Context Generation Engine for later use.

Once the Raw Context has been generated, another FSM controls the handover of the current PID to the Path Dispatcher in order to start the classification of the most recent packet. The stand-alone synthesis results for the Context Assembler unit are summarized in Table 25 below.

Table 25: Stand-alone FPGA Synthesis Results for the Context Assembler

Resource Type	*Resource Quantity*
FPGA Slices	487 of 25,280 (1.93%)
Slice Flip-Flops	250 of 50,560 (0.49%)
Slice LUTs	766 of 50,560 (1.52%)
FPGA BlockRAM memories	0 of 232 (0%)
Critical Path	6.756 ns (i.e. 148.021 MHz)

Path Dispatcher

Detailed descriptions of the concept and implementation of the Path Dispatcher unit have already been presented in chapter 4.4. The following section of the appendix presents the configuration interface of the Path Dispatcher, which is accessed through a slave attachment on the PLB bus.

As described in section 4.4.2.4, the HDGA tree structure as presented in Figure 48 is stored in the Graph Node Memory (Figure 49). From the software driver perspective, the 211 bit wide memory is mapped to a PowerPC cacheline, which is a 256 bit structure. In a C-language level, the individual fields can be accessed through components of the packed struct as shown in Code Listing 2. The most significant 45 bits will be treated as unused padding information.

```
struct HDGA_node
{
      unsigned int padding0:45[4];
      unsigned int quart:1;          // flag indicating Quaternary Node
      unsigned int mask0:32;         // binary or quaternary node 0x..0
      unsigned int value0:32;
      unsigned int oper0:3;          // 0:=, 1:<, 2:>, 3:!=, 4:Hash
      unsigned int is00Ptr:1;
      unsigned int is01Ptr:1;
      unsigned int PtrAID00:10;      // max. 10b Ptr / 6b AID
      unsigned int PtrAID01:10;
      unsigned int CtxA00_0:4;       // next stage ALU0 word
      unsigned int CtxA00_1:4;       // next stage ALU1 word
      unsigned int CtxA01_0:4;       // next stage ALU0 word
      unsigned int CtxA01_1:4;       // next stage ALU1 word
      unsigned int mask1:32;         // binary node 0x..1
      unsigned int value1:32;
      unsigned int oper1:3;          // 0:=, 1:<, 2:>, 3:!=, 4:Hash
      unsigned int is10Ptr:1;
      unsigned int is11Ptr:1;
      unsigned int PtrAID10:10;      // max. 10b Ptr / 6b AID
      unsigned int PtrAID11:10;
      unsigned int CtxA10_0:4;       // next stage ALU0 word
      unsigned int CtxA10_1:4;       // next stage ALU1 word
      unsigned int CtxA11_0:4;       // next stage ALU0 word
      unsigned int CtxA11_1:4;       // next stage ALU1 word
} __attribute__((packed));
```

Code Listing 2: Packed C-struct of Graph Node Memory Contents

The hash table lookup operations that can be interleaved in HDGA with the decision graph traversal have to be configured in two stages in accordance with the functional description in chapter 4.4.2.5. Similar as for the HDGA graph node contents, the information for the hash table memory can be packed into cacheline transfers, while the hash table configuration memory can be accessed with single 32 bit words according to Code Listing 3.

[4] On 32 bit processors, padding fields larger than 32 bits must be split into several fields, e.g. here `padding00:32; padding01:13;`. The above code `padding0:45;` would not compile and is used as a shorthand notation to clarify the total need of 45 padding bits.

```
struct HashTableConfiguration
{
      unsigned int padding0:7;
      unsigned int Hash:1;           // Enables the CRC-16 calculation
      unsigned int TruncSize:4;      // Truncation width (0: 16 bits!)
      unsigned int BaseAddress:16;   // Hash Table Base Address
      unsigned int CollRes:1;        // Enables Chained Coll. Resolution
      unsigned int reserved:3;       // reserved for future extensions
} __attribute__((packed));
struct hashtable_cr
{
      unsigned int padding0:195;
      unsigned int Chain_Valid:1;    // 0: end of list is reached
      unsigned int Hit:1;            // entry contains valid information
      unsigned int isPtr:1;
      unsigned int PtrAID:10;        // max. 10b Ptr/6b AID, as in tree
      unsigned int Key:16;           // the original search key
      unsigned int ChainPtr:16;
      unsigned int CtxAddr_0:4;      // next stage ALU0 word, if Ptr
      unsigned int CtxAddr_1:4;      // next stage ALU1 word, if Ptr
      unsigned int reserved:8;       // reserved for future extensions
} __attribute__((packed));
struct hashtable_ncr
{
      unsigned int padding0:196;
      unsigned int Hit0:1;           // first logical entry, 0x..0
      unsigned int isPtr0:1;
      unsigned int PtrAID0:10;
      unsigned int CtxAddr0_0:4;
      unsigned int CtxAddr0_1:4;
      unsigned int reserved0:8;
      unsigned int padding1:4;
      unsigned int Hit1:1;           // second logical entry, 0x..1
      unsigned int isPtr1:1;
      unsigned int PtrAID1:10;
      unsigned int CtxAddr1_0:4;
      unsigned int CtxAddr1_1:4;
      unsigned int reserved1:8;
} __attribute__((packed));
```

Code Listing 3: Packed C-structs of Hash Table Configuration Register and Hash Table Memory Contents

Depending on whether the hash table features collision resolution, the `hashtable_cr` or `hashtable_ncr` struct has to be used to build the hash table.

Code Listing 4 shows the layout of the individual components in the Translation Memory, which is used after HDGA classification in order to retrieve the classification result from the ActionID delivered by the HDGA leaf node.

```
struct translation
{
      unsigned int padding:211;
      unsigned int TrafficClass:1;    // 0: low latency, 1: high latency
      unsigned int Prio:1;            // 0: low priority
      unsigned int Num_Dest:2;        // 1:1, 2:2, 3:3, 0:4 destinations
      unsigned int PtrIM:9;           // CGE Pointer/Instruction Memory
      unsigned int ListofDest:32;     // NP Processing Path
} __attribute__((packed));
```

Code Listing 4: Packed C-struct of Translation Memory Contents

As the previous paragraphs have shown, the configuration data of the Path Dispatcher unit comprise 32 bit values (or less) for the configuration registers and hash table configuration memory and larger units for the remaining memory contents. In order to support (incremental) updates of the Path Dispatcher data structures, while the system is running, it is important that the individual contents can be written in an atomic operation. Writing less than 32 bits can be achieved by a single bus transfer, and the updated content of the corresponding memory is available in the subsequent cycle, due to the dual-port BlockRAM implementation. However, writing more than 32 bits from a processor cannot be achieved in a single cycle that easily. The Bus Attachment FSM therefore contains a 256 bit register that can hold a full processor cacheline, which will be transferred in a four doubleword burst transfer across the PLB. When the control plane processor wants to write a new configuration, this transfer register will be filled during the consecutive bus cycles, and when the transfer is finished, the entire data can be written to the actual BlockRAM memory in a single cycle. The inverse behavior is applied, when the control plane processor wants to read back configuration information: as the address is transferred over the PLB, the Bus Attachment reads the configuration memory contents into its 256 bit transfer register and can then transmit the requested information in subsequent cycles.

When defining the address map of the Path Dispatcher, care must be taken to include address ranges for both cached and non-cached accesses in order to perform the single-cycle writes (uncached) for the configuration registers and memories with less than 32 bits and cacheline transfers for the other memories. As the PowerPCs available in the Virtex-4 FPGAs have a cache map that allows enabling / disabling the caching behavior at a granularity of 128 MB, the address range for the Path Dispatcher has to be placed in the center of a 256 MB chunk of addresses. Table 26 shows the currently implemented address map of the Path Dispatcher demonstrator, with the configuration addresses mapped around the 0xA7… (uncached) and 0xA8… (cached) address blocks.

Table 26: Address Map of Path Dispatcher

PLB Base Address	*PLB High Address*	*Impl. Range*	*Contents*	*Access Mode*
0xA7FF FF00	0xA7FF FF78	16 WD	Hash Table Configuration Memory	32 bit
0xA7FF FF80		1 WD	HDGA Root Node	32 bit
0xA7FF FF88		1 WD	Root CtxAddr (ALU0)	32 bit
0xA7FF FF90		1 WD	Root CtxAddr (ALU1)	32 bit
0xA800 0000	0xA800 07E0	64 CL	Translation Memory	256 bit
0xA801 0000	0xA801 3FE0	512 CL	Graph Node Memory	256 bit
0xA802 0000	0xA802 BFE0	1,536 CL	Hash Table Memory	256 bit

In order to avoid byte steering effects that occur when 32 bit accesses are performed over the 64 bit PLB bus, the addresses for the Hash Table Configuration Memory and Registers are positioned on 64 bit multiples. With respect to mapping PLB addresses to physical addresses for the Hash Table Configuration Memory, we receive:

Table 27: Mapping PLB to Physical Addresses for Hash Table Configuration Memory

PLB Address	*Physical Address*	*Contents*
0xA7FF FF00	0x0	Configuration of first logical hash table
0xA7FF FF08	0x1	Configuration of second logical hash table
0xA7FF FF10	0x2	Configuration of third logical hash table
0xA7FF FF78	0xF	Configuration of sixteenth logical hash table

The wider memories do not incur the byte steering problem, but for the graph node memory and hash table memory, the additional mapping between logical addresses (that are used in the address / pointer fields within the respective data structure) and the physical address of the BlockRAM memories has to be considered. The different address relations are shown in Table 28 and Table 29.

Table 28: Address Relations for Graph Node Memory

PLB	*Physical*	*Logical*	*Contents*
0xA801 0000	0x000	0x000	Binary Nodes: 0x0 & 0x1; Quaternary Node: 0x0
0xA801 0020	0x001	0x002	Binary Nodes: 0x2 & 0x3; Quaternary Node: 0x2
0xA801 0040	0x002	0x004	Binary Nodes: 0x4 & 0x5; Quaternary Node: 0x4
0xA801 3FE0	0x1FF	0x3FE	Binary Nodes: 0x3FE & 0x3FF; Quaternary: 0x3FE

Table 29: Address Relations for Hash Table Memory

PLB	*Physical*	*Logical (assumed base 0x0, w/o Collision Resolution)*	*Logical (assumed base 0x0, w/ Collision Resolution)*
0xA802 0000	0x000	Offsets 0x000 & 0x001	Offset 0x000
0xA802 0020	0x001	Offsets 0x002 & 0x003	Offset 0x001
0xA802 0040	0x002	Offsets 0x004 & 0x005	Offset 0x002
0xA802 BFE0	0x5FF	Offsets 0xBFE & 0xBFF	Offset 0x5FF

SmartMem

In contrast to the description of the Buffer Manager DMA engine during the system simulations (chapter 3.3.2.2), we have embedded a more advanced DMA engine into our final FPGA demonstrator (chapter 6). This advanced version is referred to as SmartMem Buffer Manager and is described in detail in Daniel Llorente's dissertation [108]. The following paragraphs briefly summarize the main features of the SmartMem architecture and the interfaces to the other FlexPath NP modules.

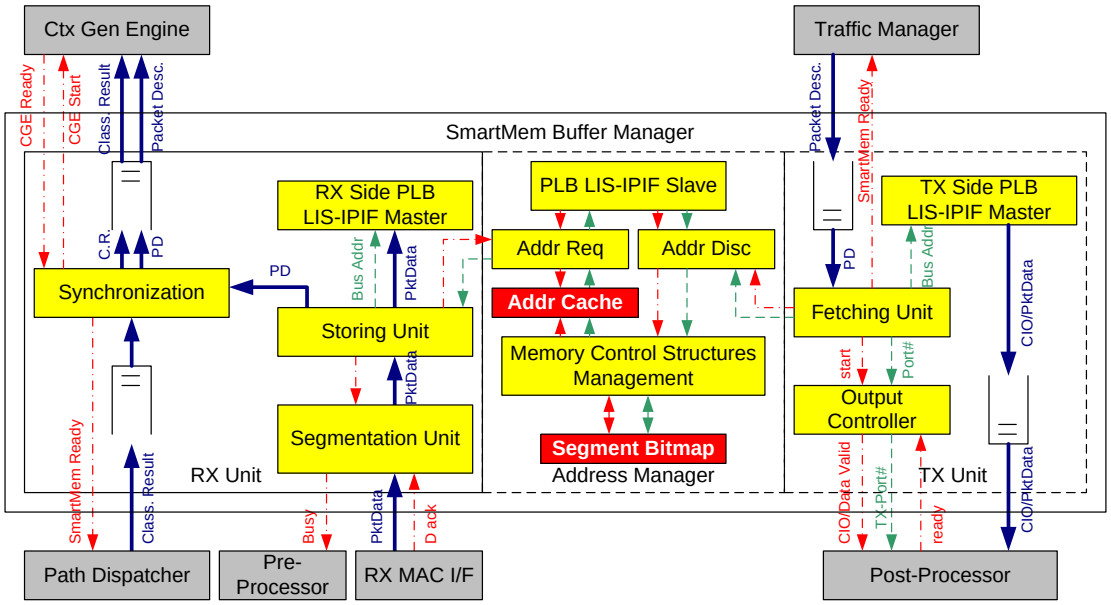


Figure 90: Abstracted Architecture of the SmartMem Buffer Manager (DMA)

When a packet is received from the link, the packet data is forwarded to the SmartMem RX Unit in parallel to the Pre-Processor. The packet data is stored in a BlockRAM buffer in the Segmentation Unit and is then partitioned to fit into a single or a combination of two segments, from a pool of 256 B, 512 B, 1024 B, and 2048 B segments. After segmentation, the packet data is forwarded to the Storing Unit, which stores the packet data in the DDR-SDRAM over the PLB master attachment. The Storing Unit requests the necessary address segments from the Address Manager, which maintains a cache of empty segment pointers. After the complete packet has been stored, the Packet Descriptor (see Figure 91) is generated, which

contains the addresses and segment sizes of the current packet along with other information and is passed through the remaining modules of the FlexPath NP.

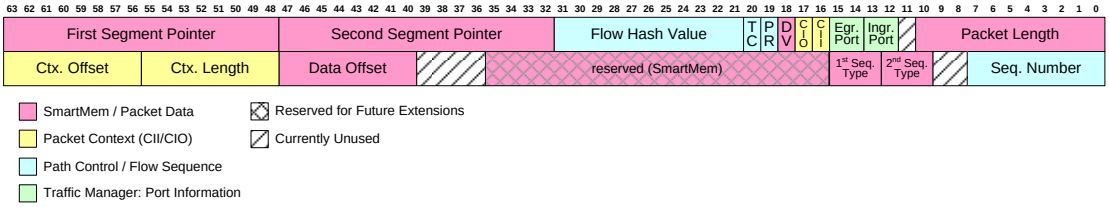


Figure 91: Structure and Contents of the Packet Descriptor

While the segmentation and storage of the current packet take place, the Pre-Processor, Context Assembler and Path Dispatcher also work on the packet data or the generated packet context in parallel. The classification result is transmitted from the Path Dispatcher to the SmartMem after classification. It was planned, that in a later implementation the SmartMem uses these classification results in order to optimize the DMA by storing the packets in different memories, depending on their further processing path. E.g. AutoRoute packets might be kept in a different memory than packets bound for the processor complex or even hardware accelerators, which might be implemented with additional local memories. However, such a feature has not yet been implemented, but the data flow of the classification results through the SmartMem already support that feature.

When both the Packet Descriptor and the classification result of the current packet are present, the whole information is passed on to the Context Generation Engine.

After the packet has been processed by the NP and the Traffic Manager determines that a packet has to be transmitted, the corresponding packet descriptor is passed to the TX Unit of the SmartMem. The Fetching Unit analyzes, if the packet has a valid output context (CIO) for the Post-Processor, and subsequently fetches both the output context and the packet data from the respective memory locations. Afterwards, the (now unused) segment addresses are returned to the Address Manager, so that they are freed (discarded) and are made available for future use. When the data is sitting in the transmit buffer, transmission to the Post-Processor is initiated by using a simple handshaking protocol.

The stand-alone synthesis results for the SmartMem buffer manager are summarized in Table 30. The figures include the two LIS-IPIF master interfaces and one LIS-IPIF slave as shown in Figure 90.

Table 30: Stand-alone FPGA Synthesis Results for the SmartMem Buffer Manager

Resource Type	*Resource Quantity*
FPGA Slices	3,354 of 25,280 (13.27%)
Slice Flip-Flops	2,951 of 50,560 (5.84%)
Slice LUTs	6,240 of 50,560 (12.34%)
FPGA BlockRAM memories	23 of 232 (9.91%)
Critical Path	8.713 ns (i.e. 114.771 MHz)

Context Generation Engine

The SmartMem was initially focused to optimize the storing efficiency of the packet data in a more generalized NP scenario. In consequence, there were initially no provisions made to perform the DMA operation for the packet context. As I have already described in chapter 3.3.2.2, Andreas Schipf had implemented a Context Generation Engine (CGE) [102] that either copied all extracted header fields and flags from the Pre-Processor into memory as Context Information Input (CII), or was able to write a pre-configured Context Information Output (CIO) with the instructions for the Post-Processor. The initial implementation was developed together with the previous version of the Buffer Manager that used linked lists of 64 byte segments to store data in main memory, and the context information was mapped to a separate linked list.

As we decided to merge the FlexPath NP demonstrator with the efforts made in the SmartMem project, the interfaces for packet and context storage changed significantly. Instead of maintaining several linked lists for CII, CIO and packet data, we agreed on consolidating context and data into a shared memory space, by appending a context section of 128 bytes in front of the packet data section in the first segment. Still, the design of the SmartMem does not allow constructing CII or CIO depending on the current type of packet and to perform the DMA of the generated context information into the memory.

In order to adapt the CGE to the current status of the ingress data path pipeline, I re-implemented the CGE to the following functional specification, which includes the later defined interactions between the SmartMem Buffer Manager, Ingress Path Control and Packet Distributor. Figure 92 shows the abstracted architecture of the current version of the CGE.

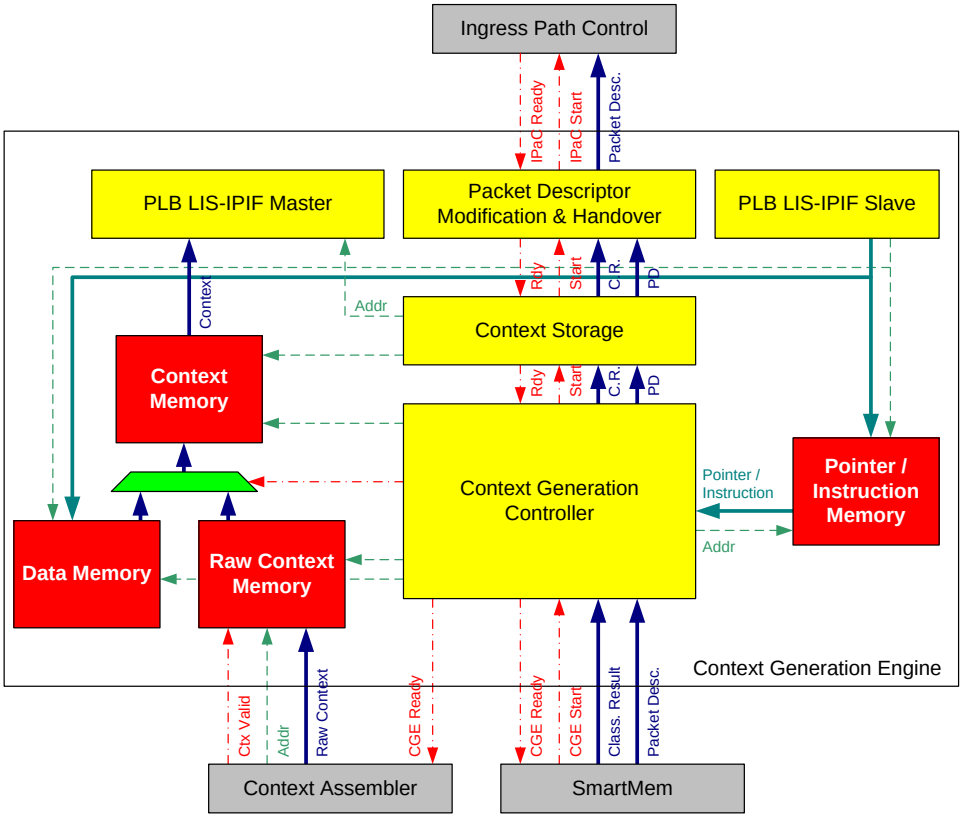


Figure 92: Abstracted Architecture of the Context Generation Engine

The main data flow through the new CGE is organized in three steps. In the first step, the classification result passed on by the SmartMem buffer manager is used to determine which kind of context information has to be generated.

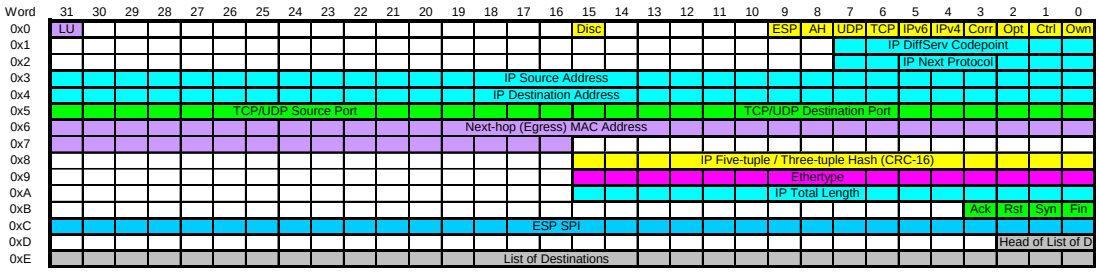


Figure 93: Standard Contents and Layout of Context Information Input (CII)

If the packet is destined for the processor complex, a CII has to be generated basically by copying the Raw Context into the reserved segment in front of the packet data. Figure 93 shows the standardized layout of the CII. The CII is basically a permutation of the Raw Context (as shown in Figure 88). While the Raw Context adheres to the layering of the protocols in the packets, and corresponds to the sequence, in which the fields are typically extracted from the arriving packet, the CII is optimized for the packet processing in the PE complex. The relevant fields for packet forwarding are all concentrated in the first eight 32 bit words, which corresponds to a single PowerPC cache line. Less frequently needed information

follows in the remainder of the CII. At the end of the CII, a list of destinations with up to four entries (at eight bits each) allows distribution of the packets by means of the packet distributor, also in multi-hop scenarios.

However, if the packet is destined for AutoRoute, a CIO has to be generated that contains the Assembler-like instructions necessary for the Post-Processor to perform the required packet modifications. Here, the contents are obtained from the Data Memory, which can be pre-configured with arbitrary instructions by the Control Plane processor. Of course, the lookup result containing the next-hop router's MAC address still has to be copied from the Raw Context Memory as an argument for the first replace instruction. Figure 94 shows an example for an AutoRoute CIO in a plain IPv4 forwarding scenario.

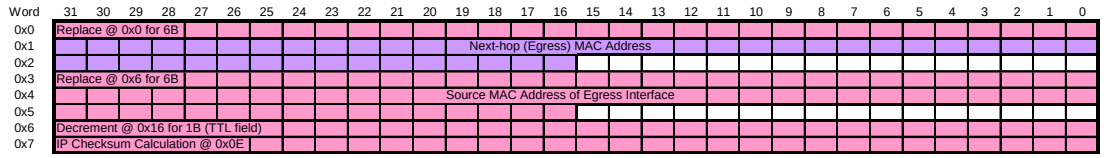


Figure 94: Standard IPv4 AutoRoute Context Information Output (CIO)

It is also possible to skip context generation altogether by specifying a context length of zero, in this case, only the later described modifications in the Packet Descriptor are performed. This feature helps to measure the performance of the system without the FlexPath-specific context information and this has been used for the reference scenario measurements described in chapter 6.4.

In the second processing step, the context information is stored in the system memory at the address extracted from the packet descriptor using the LIS-IPIF master interface of the CGE.

Finally, the CGE completes the following fields in the packet descriptor (see Figure 91):

- The 11 least significant bits of the IP five-tuple hash value are added as Flow Hash value for later use by the Ingress / Egress side Path Control
- Traffic Class and Priority bits are set in accordance with the classification result obtained from the Path Dispatcher
- CII or CIO bits are set if a valid CII or CIO context have been stored in memory
- Ingress and Egress MAC interface information is added in accordance with the information obtained from the next-hop lookup engine and the Pre-Processor

– Context Offset, which describes the offset (in bytes) from the beginning of the first packet segment, is set to zero, as the CGE starts storing any context from the beginning of the segment

– Context Length, which describes the length of valid context information is updated in accordance with the generated context

Table 31 shows the synthesis results of the Context Generation Engine including both LIS-IPIF interfaces (Master and Slave).

Table 31: Stand-alone Synthesis Results for the Context Generation Engine

Resource Type	*Resource Quantity*
FPGA Slices	759 of 25,280 (3.00%)
Slice Flip-Flops	961 of 50,560 (1.90%)
Slice LUTs	1,201 of 50,560 (2.38%)
FPGA BlockRAM memories	5 of 232 (2.16%)
Critical Path	5.541 ns (i.e. 180.486 MHz)

Traffic Manager

As the packet descriptors leave the egress side Path Control in the correct sequence, they have to be queued to resolve output port contention, which might happen as the aggregated processing capabilities of the NP may exceed the maximum transmission bandwidth of a single Gigabit Ethernet interface. In addition to resolving contention, the implemented Traffic Manager also implements a strict priority-based round robin transmission scheduling that allows a simple form of QoS implementation. The abstracted architecture of the Traffic Manager is depicted in Figure 95.

The arriving packet descriptors are enqueued into the correct queue evaluating the egress port and priority bits of the packet descriptor (see Figure 91). Each queue can hold a maximum of 128 packet descriptors, but the capacity can be reduced by specifying a generic parameter in the VHDL code. If a packet descriptor would have to be assigned to a full queue, the descriptor will instead be discarded through the Traffic Manager's PLB LIS-IPIF master interface.

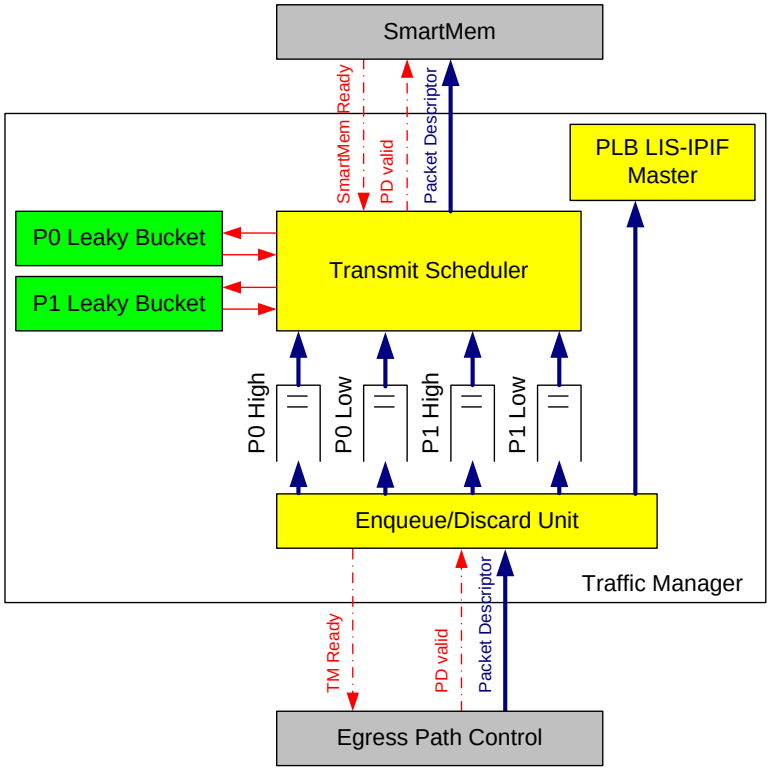


Figure 95: Abstracted Architecture of the Traffic Manager

The Transmit Scheduler determines which packet descriptor is to be sent next by evaluating the backpressure signals from the SmartMem transmit side interfaces, queue fill level and the leaky buckets that are used to limit the transmit rate on each port to 1 Gbit/s. If a port still has transmission capacity, the high priority queues will be worked off first, iterating between both ports (i.e. round-robin) if packets are present for both ports. Low priority packets can be transmitted on the other port, if there are no packets in the high priority queue and more high priority packets cannot be transmitted due to an overflow in the leaky bucket for the respective port.

Table 32 summarizes the resource consumption and synthesis results of the Traffic Manager.

Table 32: Stand-alone FPGA Synthesis Results for the Traffic Manager

Resource Type	*Resource Quantity*
FPGA Slices	441 of 25,280 (1.74%)
Slice Flip-Flops	533 of 50,560 (1.05%)
Slice LUTs	689 of 50,560 (1.36%)
FPGA BlockRAM memories	4 of 232 (1.72%)
Critical Path	5.683 ns (i.e. 175.963 MHz)

References

[1] TeleGeography Research, "Global Internet Geography - Executive Summary", 2009, *available online (Feb 3, 2010):* http://www.telegeography.com/product-info/gig/download/telegeography-global-internet.pdf

[2] F. Baker, Cisco Systems, "Requirements for IP version 4 Routers", IETF RFC 1812, June 1995, *available online (Apr 14, 2009):* http://tools.ietf.org/html/rfc1812

[3] S. Blake et.al., "An Architecture for Differentiated Services", IETF RFC 2475, December 1998, *available online (Apr 14, 2009):* http://tools.ietf.org/html/rfc2475

[4] PacketClearingHouse, "Internet Exchange Directory", *available online (Feb 3, 2010):* https://prefix.pch.net/applications/ixpdir/?show_active_only=0&sort=participants&order=desc

[5] S. Hauger, T. Wild, A. Mutter, A. Kirstädter, K. Karras, R. Ohlendorf, F. Feller, J. Scharf, "Packet Processing at 100 Gbps and Beyond - Challenges and Perspectives", 10. ITG Fachtagung Photonische Netze, Dresden, Germany, May 4-5, 2009

[6] AdvancedTCA Specifications, PCI Industrial Computers Manufacturing Group (PICMG), *available online (Apr 14, 2009):* http://www.picmg.org/v2internal/newinitiative.htm

[7] R. Ohlendorf, A. Herkersdorf, T. Wild, "FlexPath NP - A Network Processor Concept with Application-Driven Flexible Processing Paths", Proceedings of the CODES+ISSS 2005, pp. 279-284, Jersey City, NJ, USA, September 2005, DOI: 10.1145/1084834.1084904

[8] SystemC Homepage, *available online (Apr 14, 2009):* http://www.systemc.org

[9] N. Shah, "Understanding Network Processors", Technical Report, UC Berkeley, September 4, 2001, *available online (Apr 15, 2009):* http://www.gigascale.org/pubs/338/UnderstandingNPs.pdf

[10] Agere Network Processors page, *available online (Apr 15, 2009):* http://nps.agere.com/index.html, (hint: Agere was acquired by LSI on April 2, 2007 thus maintenance of this link is uncertain in the long run)

[11] Brecis Communications, "MSP5000 Multi-Service Processor Product Brief", 2002, *available online (Apr 15, 2009):* http://www.datasheetarchive.com/pdf-

datasheets/Datasheets-319/512401.pdf (hint: original website www.brecis.com is no longer reachable)

[12] J. R. Allen, Jr. et.al., "IBM PowerNP network processor: Hardware, software and applications", IBM Journal of Research and Development, vol. 47, no. 2/3, pp. 177-193, March/May 2003

[13] "Parallel Express Forwarding on the Cisco 10000 Series", Cisco White Paper, *available online (Apr 15, 2009):* http://www.cisco.com/en/US/prod/collateral/routers/ps133/prod_white_paper09186a008008902a.pdf

[14] Intel, "IXP1200 Network Processor", *available online (Apr 15, 2009):* http://download.intel.com/design/network/datashts/27829810.pdf

[15] Xelerated, "Xelerator X40 Network Processor Product Brief", 2001, *available online (Apr 15, 2009):* http://www.icwic.cn/icwic/data/pdf/cd/cd075/Network%20Processor/a/146250.pdf

[16] F. Miller, "Entwicklung von Netzwerkprozessoren am Beispiel der Intel IXP Produktfamilie", seminar paper and presentation (Hauptseminar) held at LIS, TUM in winter term 2008/2009, *in German*

[17] Intel, "Intel IXP2855 Network Processor. Product Brief", *available online (Apr 15, 2009):* http://download.intel.com/design/network/ProdBrf/30943001.pdf

[18] SafeNet, "SafeNet Announces Inline IPsec Security Engine for System on Chip Designs", Press Release, *available online (Dec 18, 2009):* http://www.safenet-inc.com/About_SafeNet/News_and_Media/News_and_Media_Items/2006/2006-01-25_-_SafeNet_Announces_Inline_IPSec_Security_Engine_for_System_on___.aspx#

[19] SafeNet, "SafeXcel IP Flow-Through Packet Engine", Product Brief, *available online (Dec 18, 2009):* http://www2.safenet-inc.com/Library/EMB/SafeNet_Product_Brief_SafeXcel_IP_-_EIP-196.pdf

[20] AMCC, "nP7300 10 Gbps Network Processor with Integrated Traffic Manager Product Brief", *available online (Apr 16, 2009):* https://www.amcc.com/MyAMCC/retrieveDocument/SNP/nP7300_060822.pdf

[21] Netronome, "NFP-3200 Network Flow Processor. Product Brief.", *available online (Apr 16, 2009):*

http://www.netronome.com/files/file/Netronome%20NFP%20Product%20Brief%20(3-09).pdf

[22] Xelerated, "Xelerator X11 Network Processors", *available online (Apr 16, 2009):* http://www.xelerated.com/uploads/files/5.PDF

[23] Linley Group, "Linley Carrier Ethernet Seminar", January 28, 2010, *available online (Jun 2, 2010):* http://www.linleygroup.com/Seminars/carrier_eth_program.html

[24] Cisco, "The Cisco QuantumFlow Processor: Cisco's Next Generation Network Processor", Solution Overview, *available online (Apr 16, 2009):* http://www.cisco.com/en/US/prod/collateral/routers/ps9343/solution_overview_c22-448936.pdf

[25] "Cisco puts high-end silicon on the edge", EETimes article, March 3, 2008, *available online:* http://www.eetimes.com/showArticle.jhtml?articleID=206901479

[26] Cavium Networks, "OCTEON II Internet Application Processor (IAP) Family", *available online (Apr 16, 2009):* http://www.caviumnetworks.com/OCTEON_II_MIPS64.html

[27] J. Lockwood, "An Open Platform for Development of Network Processing Modules in Reprogrammable Hardware", IEC DesignCon 2001, Santa Clara, CA, USA, January 2001

[28] I.A. Troxel, A.D. George and S. Oral, "Design and Analysis of a Dynamically Reconfigurable Network Processor", IEEE Conference on Local Computer Networks (LCN'02), Tampa, FL, USA, November 6-8, 2002, pp. 483-492

[29] I. Papaefstathiou, et.al., "PRO3: A Hybrid NPU Architecture", IEEE Micro, vol. 24, issue 5, September/October 2004, pp. 20-33

[30] K. Ravindran, N. Satish, Y. Jin, K. Keutzer, "An FPGA-based Soft Multiprocessor System for IPv4 Packet Forwarding", FPL 2005, Tampere, Finland, August 24-26, 2005, pp. 487-492

[31] M. Platzner, J. Teich, N. Wehn (Eds.), "Dynamically Reconfigurable Systems: Architectures, Design Methods and Applications", ISBN 978-90-481-3484-7, Springer Science+Business Media B.V. 2010

[32] C. Kachris, S. Vassiliadis, "Analysis of a Reconfigurable Network Processor", IPDPS 2006, Rhodos, Greece, April 25-29, 2006

[33] J.-C. Niemann, C. Puttmann, M. Porrmann, U. Rückert, "Resource efficiency of the GigaNetIC chip multiprocessor architecture", Journal of Systems Architecture, vol 53, issues 5-6, pp. 285-299, May/June 2007

[34] M. Okuno, S. Nishimura, S. Ishida and H. Nishi, "Cache-based Network Processor Architecture: Evaluation with Real Network Traffic", IEICE Transactions on Electronics, vol. E89-C, no. 11, pp. 1620-1628, November 2006

[35] T. Li, X. Zhang, Z. Sun, "DynaNP - A Coarse-grain Dataflow Network Processor Architecture with Dynamic Configurable Processing Path", SNPD 2007, vol. 3, pp. 182-187, Qingdao, China, July 30 - August 1, 2007

[36] V. Fuller, Cisco Systems, et.al., "Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan", IETF RFC 4632, August 2006, *available online (Apr 17, 2009):* http://tools.ietf.org/html/rfc4632

[37] R. Braden, ISI, et.al., "Integrated Services in the Internet Architecture: An Overview", IETF RFC 1633, June 1994, *available online (Apr 17, 2009):* http://tools.ietf.org/html/rfc4632

[38] J. Babiarz, Nortel Networks, et.al, "Configuration Guidelines for DiffServ Service Classes", IETF RFC 4594, August 2006, *available online (Apr 20, 2009):* http://tools.ietf.org/html/rfc4594

[39] S. Kent, BBN Technologies, et.al., "Security Architecture for the Internet Protocol", IETF RFC 4301, December 2005, *available online (Apr 20, 2009):* http://tools.ietf.org/html/rfc4301

[40] H. Schulzrinne, Columbia University, et.al., "RTP: A Transport Protocol for Real-Time Applications", IETF RFC 3550, July 2003, *available online (Apr 21, 2009):* http://tools.ietf.org/html/rfc3550

[41] H. Schulzrinne, Columbia University, et.al., "RTP Profile for Audio and Video Conferences with Minimal Control", IETF RFC 3551, July 2003, *available online (Apr 21, 2009):* http://tools.ietf.org/html/rfc3551

[42] A. Kraas, "Verwendung von RTP/RTCP in Hinblick auf VoIP", seminar paper and presentation (Hauptseminar) held at LIS, TUM in winter term 2004/2005, *in German*

[43] J. Rosenberg, dynamicsoft, et.al., "SIP: Session Initiation Protocol", IETF RFC 3261, June 2002, *available online (Apr 22, 2009):* http://tools.ietf.org/html/rfc3261

[44] Z. Hichem, "Das SIP-Protokoll", seminar paper and presentation (Hauptseminar) held at LIS, TUM in winter term 2004/2005, *in German*

[45] M. Meitinger, R. Ohlendorf, T. Wild and A. Herkersdorf, "Application Scenarios for FlexPath NP", Technical Report TUM-LIS-TR-0501, Technische Universität München, Lehrstuhl für Integrierte Systeme, December 2005

[46] Alan Millard, "2.5G/3G Wireless Networks and the Application of Network Processors", Technical Report, IBM 2002

[47] K. Venken, I. Vinagre, J. de Vriendt, "Analysis of the Evolution to an IP-based UMTS Terrestrial Radio Access Network", IEEE Wireless Communications, October 2003

[48] L. Fang, N. Bitar, R. Zhang, M. Taylor, "The Evolution of Carrier Ethernet Services - Requirements and Deployment Case Studies", IEEE Communications Magazine, vol. 46, no. 3, March 2008, pp. 69-76

[49] D. Fedyk, D. Allan, "Ethernet Data Plane Evolution for Provider Networks", IEEE Communications Magazine, vol. 46, no. 3, March 2008, pp. 84-89

[50] Donald E. Knuth, "The Art of Computer Programming, Second Edition, Volume 3: Sorting and Searching", © 1998 Addison-Wesley, ISBN 0-201-89685-0, 23rd printing, August 2007

[51] Z. Cao, Z. Wang, E. Zegura, "Performance of Hashing-Based Schemes for Internet Load Balancing", IEEE INFOCOM 2000, vol. 1, Tel Aviv, Israel, March 2000, pp. 332-341

[52] Donald R. Morrison, "PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric", Journal of the ACM, vol. 15, no. 4, October 1968, pp. 514-534

[53] IDT, "Taking Packet-Processing to the Next Level Achieving Next-Generation Classification Performance Using Multiple Databases and IP Co-processors", White Paper, *available online (Jan 9, 2009):* http://www.idt.com/products/files/8636/75K6213452134_WP_77739.pdf

[54] IDT, "Network Search Engines", Product Flyer, *available online (Oct 15, 2009):* http://www.idt.com/products/getDoc.cfm?docID=10154

[55] P. Gupta, N. McKeown, "Algorithms for Packet Classification", IEEE Network, vol. 15, no. 2, pp. 24-32, March/April 2001

[56] R. Ohlendorf, M. Meitinger, T. Wild, A. Herkersdorf, "A Processing Path Dispatcher in Network Processor MPSoCs", IEEE Transactions on VLSI Systems, vol. 16, no. 10, pp. 1335-1345, October 2008

[57] P. Gupta, N. McKeown, "Packet Classification on Multiple Fields", ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communication, Cambridge, MA, USA, August/September 1999, pp. 147-160

[58] S. Singh, F. Baboescu, G. Varghese, J. Wang, "Packet Classification using Multi-Dimensional Cutting", ACM SIGCOMM 2003, Karlsruhe, Germany, August 2003, pp. 213-224

[59] P. Gupta, N. McKeown, "Packet Classification using Hierarchical Intelligent Cuttings", Hot Interconnects 7, Stanford, CA, USA, August 1999, pp. 34-41

[60] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel, "Fast and Scalable Layer Four Switching", ACM SIGCOMM 1998, Vancouver, BC, Canada, September 1998, pp. 191-202

[61] F. Baboescu, S. Singh, G. Varghese, "Packet Classification for Core Routers: Is there an alternative to CAMs?", IEEE INFOCOM 2003, San Francisco, CA, USA, April 2003, pp. 53-63

[62] D. Pao, C. Liu, "Parallel tree search: An algorithmic approach for multi-field packet classification", Computer Communications, vol. 30, no. 2, pp. 302-314, January 2007

[63] T. Lakshman, D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching", ACM SIGCOMM 1998, Vancouver, BC, Canada, September 1998, pp. 203-214

[64] D. Taylor, J. Turner, "Scalable Packet Classification using Distributed Crossproducting of Field Labels", IEEE INFOCOM 2005, Miami, FL, USA, March 2005, pp. 269-280

[65] D. Taylor, "Models, Algorithms and Architectures for Scalable Packet Classification", Dissertation, Washington University in St. Louis, St. Louis, MO, USA, 2004

[66] A. Prakash, R. Kotla, T. Mandal, A. Aziz, "A High-Performance Architecture and BDD-based Synthesis Methodology for Packet Classification", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 22, no. 6, pp. 698-709, June 2003

[67] E. Cohen, C. Lund, "Packet Classification in Large ISPs: Design and Evaluation of Decision Tree Classifiers", ACM SIGMETRICS 2005, Banff, AB, Canada, June 2005, pp. 73-84

[68] T. Woo, "A Modular Approach to Packet Classification: Algorithms and Results", IEEE INFOCOM 2000, vol. 3, Tel Aviv, Israel, March 2000, pp. 1213-1222

[69] R. Lysecky, F. Vahid, "On-Chip Logic Minimization", DAC 2003, Anaheim, CA, USA, June 2003, pp. 334-337

[70] R. Brayton, G. Hachtel, C. McMullen, A. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishers, Boston, MA, USA, 1984, ISBN 0-89838-164-9

[71] Instituto Politécnico do Porto, "Espresso for MS-DOS, version 2.3", Porto, Portugal, *available online (November 2, 2009):* http://www.dei.isep.ipp.pt/~acc/bfunc/

[72] R. Ohlendorf, M. Meitinger, T. Wild, A. Herkersdorf, "An Application-aware Load Balancing Strategy for Network Processors", HiPEAC 2010, Pisa, Italy, January 2010, LNCS 5952, pp. 156-170

[73] G. Dittmann, A. Herkersdorf, "Network Processor Load Balancing for High-Speed Links", SPECTS 2002, San Diego, CA, USA, July 2002, pp. 727-735

[74] L. Kencl, "Load Sharing for Multiprocessor Network Nodes", Dissertation, EFPL Lausanne, Switzerland, 2003

[75] W. Shi, M. MacGregor, P. Gburzynski, "Load Balancing for Parallel Forwarding", IEEE Transactions on Networking, vol. 13, no. 4, pp. 790-801, August 2005

[76] W. Shi, M. MacGregor, P. Gburzynski, "A Scalable Load Balancer for Forwarding Internet Traffic: Exploiting Flow-level Burstiness", ANCS 2005, Princeton, New Jersey, October 2005, pp. 145-152

[77] W. Shi, L. Kencl, "Sequence-Preserving Adaptive Load Balancers", ANCS 2006, San Jose, CA, USA, December 2006, pp. 143-152

[78] S. Govind, R. Govindarajan, J. Kuri, "Packet Reordering in Network Processors", IPDPS 2007, Long Beach, CA, USA, March 2007

[79] N. Brownlee, K. C. Claffy, "Understanding Internet Traffic Streams: Dragonflies and Tortoises", IEEE Communications Magazine, vol. 40, no. 10, pp. 110-117, October 2002

[80] P. Dykstra, WareOnEarth Communications, Inc., Protocol Overhead Survey, *available online (Nov 5, 2009): http://sd.wareonearth.com/~phil/net/overhead/*

[81] T. Wild, A. Herkersdorf, R. Ohlendorf, "Performance Evaluation for System-on-Chip Architectures using Trace-based Transaction Level Simulation", DATE 2006, Munich, Germany, March 2006

[82] R. Ohlendorf, T. Wild, M. Meitinger, H. Rauchfuss, A. Herkersdorf, "Performance Evaluation of RISC-based SoC Platforms in Network Processing Applications", IC-SAMOS 2006, Samos, Greece, July 2006, pp. 152-159

[83] R. Ohlendorf, T. Wild, M. Meitinger, H. Rauchfuss, A. Herkersdorf, "Simulated and Measured Performance Evaluation of RISC-based SoC Platforms in Network Processing Applications", Journal for Systems Architecture, vol. 53, no. 10, pp. 703-718, October 2007

[84] R. Ohlendorf, M. Meitinger, T. Wild, A. Herkersdorf, "A Packet Classification Technique for On-Chip Processing Path Selection", WASP 2007, Salzburg, Austria, October 2007

[85] R. Ramaswamy, T. Wolf, "PacketBench: A Tool for Workload Characterization of Network Processing", IEEE 6th Annual Workshop on Workload Characterization (WWC-6), Austin, TX, USA, October 2003, pp. 42-50

[86] C. Jenkins, "NPU Co-Processors", Presentation at Network Processor Conference, San Jose, CA, USA, August 2000

[87] J. Rabaey, "Silicon Architectures for Wireless Systems - Part 2 Configurable Processors", Tutorial at Hot Chips 13, Stanford, CA, USA, August 2001, *available online: http://www.hotchips.org/archives/hc13/*

[88] D. Llorente, K. Karras, T. Wild, A. Herkersdorf, "Advanced Packet Segmentation and Buffering Algorithms in Network Processors", Transactions on HiPEAC, vol. 4, no. 4, 2009, *available online (Jul 2, 2010): http://www.hipeac.net/node/3030*

[89] D. Llorente, K. Karras, T. Wild, A. Herkersdorf, "Buffer Allocation for Advanced Packet Segmentation in Network Processors", Application-specific Systems, Architectures and Processors (ASAP 2008), Leuven, Belgium, July 2008, pp. 221-226

[90] C. Albrecht, J. Foag, R. Koch, E. Maehle, "DynaCORE - A Dynamically Reconfigurable Coprocessor Architecture for Network Processors", 14th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP 2006), Montbéliard-Sochaux, France, February 2006, pp. 101-108

[91] T. Pionteck, R. Koch, C. Albrecht, E. Maehle, M. Meitinger, R. Ohlendorf, T. Wild, A. Herkersdorf, "SPP1148 Booth: Network Processors", FPL 2008, Heidelberg, Germany, September 2008, p. 352

[92] Sprint Nextel, Academic Research Group, IP Data Analysis, Packet Size Distribution, *available online (Nov 16, 2009):* https://research.sprintlabs.com/packstat/packetoverview.php

[93] S. Lugmair, "Entwicklung eines Pre-Prozessors für die Network Processing Prototyping Platform", Diploma Thesis, LIS, TUM, April 2005, *in German*

[94] D. Llorente, K. Karras, M. Meitinger, H. Rauchfuss, T. Wild, A. Herkersdorf, "Accelerating Packet Buffering and Administration in Network Processors", International Symposium on Integrated Circuits 2007, Singapore, Singapore, September 2007, pp. 373-377

[95] A. Dunkels, A lightweight TCP/IP Stack, *available online (Nov 17, 2009):* http://savannah.nongnu.org/projects/lwip/

[96] C. Shannon, E. Aben, K.C. Claffy, D. Andersen, N. Brownlee, "The CAIDA OC48 Traces Dataset", *available online (Dec 14, 2009):* http://www.caida.org/data/passive/passive_oc48_dataset.xml,
files used:
20020814-090000-1-anon.pcap
20020814-091500-1-anon.pcap
20020814-093000-1-anon.pcap
20020814-094500-1-anon.pcap

[97] C. Shannon, E. Aben, K.C. Claffy, D. Andersen, "The CAIDA Anonymized 2008 Internet Traces", *available online (Dec 14, 2009):* http://www.caida.org/data/passive/passive_2008_dataset.xml,
files used:
eq-chic.dirA.20080717-130000.UTC.anon.pcap
eq-chic.dirA.20080717-130500.UTC.anon.pcap
eq-chic.dirA.20080717-131000.UTC.anon.pcap
eq-chic.dirA.20080717-131500.UTC.anon.pcap

[98] C. Shannon, E. Aben, K.C. Claffy, D. Andersen, "The CAIDA Anonymized 2008 Internet Traces", *available online (Dec 14, 2009):* http://www.caida.org/data/passive/passive_2008_dataset.xml, files used: eq-chic.dirB.20080717-132000.UTC.anon.pcap

[99] Technische Universität München, Institute for Integrated Systems, "LIS-IPIF Specification", *available online (February 17, 2010):* http://www.lis.ei.tum.de/?lisipif

[100] Xilinx, "ML410 Embedded Development Platform", UG085, December 2008, *available online (March 8, 2010):* http://www.xilinx.com/support/documentation/boards_and_kits/ug085.pdf

[101] Xilinx, "Virtex-4 User Guide", UG070, December 2008, *available online (March 8, 2010):* http://www.xilinx.com/support/documentation/user_guides/ug070.pdf

[102] A. Schipf, "Entwicklung einer Context Generation Engine für die Network Processing Prototyping Platform", Diploma Thesis, LIS, TUM, July 2006, *in German*

[103] Spirent, "Spirent TestCenter Series 1000 and Series 2000 Gigabit Ethernet Test Modules", *available online (April 22, 2010):* http://www.spirent.com/Broadband/Voice_IMS/~/media/Datasheets/Broadband/PAB/SpirentTestCenter/STC_Series_1000-2000_GbE_Test_Modules_datasheet.ashx

[104] S. Bradner, Harvard University, et.al., "Benchmarking Methodology for Network Interconnect Devices", IETF RFC 2544, March 1999, *available online (Apr 22, 2010):* http://tools.ietf.org/html/rfc2544

[105] Agilent Technologies, "Mixed Packet Size Throughput", *available online (March 31, 2010):* http://advanced.comms.agilent.com/n2x/docs/insight/2001-08/TestingTips/1MxdPktSzThroughput.pdf

[106] M. Meitinger, R. Ohlendorf, T. Wild, A. Herkersdorf, "FlexPath NP - A Network Processor Architecture with Flexible Processing Paths", SoC 2008, Tampere, Finland, November 4-6, 2008

[107] M. Meitinger, "Paketverteilung und Resequenzierung im FlexPath Netzwerkprozessor" (Arbeitstitel), Dissertation, TU München, Germany, 2010, *in German (in preparation)*

[108] D. Llorente, "SmartMem - An Advanced Memory Subsystem for Networking Applications", Dissertation, TU München, Germany, 2010, *(in preparation)*

[109] A. Lankes, T. Wild, A. Herkersdorf, "Hierarchical NoCs for Optimized Access to Shared Memory and IO Resources", DSD 2009, Patras, Greece, August 27-29, 2009, pp. 255-262

General remark about the references used in this work: Most of the documents used during the research for this dissertation were obtained from Internet sources. Therefore, along with the reference description, I have included links to the original online documents. Unless specifically indicated, these links are active at the time of writing this dissertation and the last access date is specified next to the URL. While it can be assumed, that academic research papers published by recognized institutions like IEEE, ACM etc. remain accessible for longer periods of time, this may not be true for material supplied from commercial companies.

List of Figures

List of Tables

Code Listings

Abbreviations

3DES	Triple Data Encryption Standard
ABS	Adaptive Burst Shifting, *load balancing technique proposed by Shi et.al. in [75]*
ACL	Access Control List
ADPCM	Adaptive Differential Pulse Code Modulation
AES	Advanced Encryption Standard
AH	Authentication Header (defined in ➔ IETF RFC 4302)
AHH	Adaptive HRW (highest random weight) Hashing, *load balancing technique proposed by Kencl in [74]*
ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
ARP	Address Resolution Protocol (defined in ➔ IETF RFC 826)
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction set Processor
ATCA	Advanced Telecommunication Compute Architecture, *Industrial standard for telecommunication equipment*
ATM	Asynchronous Transfer Mode, *packet switching protocol operating on L2 of the ➔ OSI stack*
BDD	Binary Decision Diagram
BE	best effort, *default ➔ QoS service class, i.e. no special prioritization*
B-RAS	Broadband Remote Access Server
BV	Boolean Variable
CAM	Content Addressable Memory
CAS	Column Address Select
CBR	Constant Bit Rate
CF	Column Fitness
CIDR	Classless Inter-Domain Routing
CII	Context Information Input, *data structure inside a FlexPath NP*
CIO	Context Information Output, *data structure inside a FlexPath NP*
CMOS	Complementary Metal Oxide Semiconductor
CPI	Cycles Per Instruction, *performance metric of microprocessors*

CPU	Central Processor Unit, *within this thesis widely used as acronym for software-programmable microprocessors in general, not only "conventional" CPUs as e.g. an Intel Pentium, etc.*
CRC	Cyclic Redundancy Check, *error correcting code used in a variety of transmission protocols, e.g. Ethernet, ATM*
DAG	directed acyclic graph
DCFL	Distributed Crossproducting of Field Labels, *packet classification technique proposed by Taylor et.al. in [64], [65]*
DDR	Double Data Rate
DiffServ	Differentiated Services, ➔ *QoS architecture*
DMA	Direct Memory Access
DSCP	DiffServ Codepoint
DSL	Digital Subscriber Line
DSP	Digital Signal Processor
EDK	*Xilinx FPGA development tool for processor-based designs*
ESP	Encapsulating Security Payload (defined in ➔ IETF RFC 4303)
ETSI	European Telecommunications Standards Institute
FIFO	First-in First-out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPRS	General Packet Radio Service, *packet-oriented data transmission standard in ➔ GSM and ➔ UMTS networks*
GSM	Global System for Mobile communications, *formerly groupe spéciale mobile, 2nd generation mobile cellular network standard by ➔ ETSI*
HABS	Hash Adapted by Burst Shifting, *packet classification technique proposed by Kencl et.al. in [77]*
HDGA	Heterogeneous Decision Graph Algorithm, *packet classification technique proposed for use in Path Dispatcher*
HLU	Hash LookUp, *load balancing technique proposed in this dissertation*
HRW	Highest Random Weight
HMAC-SHA1	Hash Message Authentication Code/Secure Hash Algorithm 1
ICMP	Internet Control Message Protocol (defined in ➔ IETF RFC 792)

IEEE	Institute for Electrical and Electronics Engineers, *International professional and standardization organization*
IETF	Internet Engineering Task Force, *standardization body for the IP protocol suite*
IMIX	Internet Mix, *Industry standard packet size distribution used for networking equipment testing*
IntServ	Integrated Services, ➔ *QoS architecture*
I/O	Input/Output
IP	Internet Protocol (defined in ➔ IETF RFC 791), *alternative meaning:* intellectual property
IPsec	IP Security, *group of protocols and architecture defined to provide secure communication across IP networks, namely ➔ ESP and ➔ AH*
ISA	Instruction Set Architecture
ISE	*Xilinx FPGA Development Tool*
ISDN	Integrated Services Digital Network
ISP	Internet Service Provider
ITU-T	International Telecommunications Union, Telecommunication Standardization Section, *standardization body of the United Nations*
kpps	kilo packets per second
LAN	Local Area Network
LCD	Liquid Crystal Display
LED	Light Emitting Diode
LSB	Least Significant Bit
LUT	Lookup table, *basic element in an ➔ FPGA*
MAC	Medium Access Control, *usually refers to the Layer 2 protocol of the ➔ OSI stack*
Mbps	Megabit per second
MIPS	Million Instructions Per Second, *performance metric of microprocessors*
MSB	Most Significant Bit
MPLS	Multi-Protocol Label Switching, *packet switching protocol on L2/L2.5 of the ➔ OSI stack that is used for high-speed switching networks with good ➔ QoS control*
MPSoC	Multi-Processor ➔ System-on-Chip

MUX	Multiplexer
NP	Network Processor
NPU	Network Processor Unit
NSE	Network Search Engine
OSI	Open Systems Interconnection, *seven layer reference model used to classify networking protocols. The layers are: physical (1), data link (2), network (3), transport (4), session (5), presentation (6) and application (7).*
PCB	Printed Circuit Board
PE	Processing Element, *generic term referring to either programmable ➔ CPU resources or application-specific hardware accelerators*
PID	Packet ID
PLB	Processor Local Bus, *IBM processor bus specification relevant for PowerPC systems*
PSTN	Public Switched Telephone Network
QoS	Quality-of-Service
RAM	Random Access Memory
RAS	Row Address Select
RFC	Request for Comment (refers to ➔ IETF standards documents) also: Recursive Flow Classification, *packet classification technique proposed by Gupta et.al. in [57]*
RGMII	Reduced Gigabit Media Independent Interface
RISC	Reduced Instruction Set Computer
ROBDD	Reduced, Ordered ➔ BDD, often simply referred to as BDD
RSVP	Resource Reservation Protocol
RTCP	Real-Time Control Protocol
RTP	Real-Time Protocol
RX	Receive
S&H	Spraying and ➔ HLU, *load balancing technique proposed in this dissertation*
SAD	Security Association Database
SDH	Synchronous Digital Hierarchy, *standardized optical transmission scheme by ➔ ITU-T, used worldwide (except North America, see ➔ Sonet)*

SDRAM	Synchronous Dynamic ➔ RAM
SGMII	Serial Gigabit Media Independent Interface
SIP	Session Initiation Protocol
SLA	Service Level Agreement
SoC	System-on-Chip
Sonet	Synchronous Optical NETwork, *similar to ➔ SDH and interoperable with SDH networks, standardized optical transmission scheme by ➔ ANSI used in North America*
SPC	Serial-to-Parallel Converter
SPD	Security Policy Database
SRAM	Static Random Access Memory
TCAM	Ternary ➔ CAM
TCP	Transmission Control Protocol (defined in ➔ IETF RFC 793)
TTL	time-to-live, *header field in ➔ IP packets*
TX	Transmit
UDP	User Datagram Protocol (defined in ➔ IETF RFC 768)
UMTS	Universal Mobile Telecommunications System, *3rd generation mobile cellular network standard*
VHDL	Very high speed integrated circuit Hardware Description Language
VLAN	Virtual ➔ LAN
VLIW	Very Long Instruction Word, *special type of processor architecture*
VoIP	Voice-over-IP
VPN	Virtual Private Network
WAN	Wide Area Network
WEP	Wired Equivalent Privacy
WLAN	Wireless ➔ LAN
WPA	Wi-Fi Protected Access

List of Prior-Printed Publications

During the course of the FlexPath project, a number of publications were released, covering different aspects of the NP architecture and scientific contributions claimed within this dissertation. The complete list of own (title in **bold** in the following) and co-authored publications was submitted to the faculty of electrical engineering and information technology along with this dissertation in accordance with § 6 (1) 2 and § 6 (5) 3 of the Promotionsordnung der Technischen Universität München (Doctoral Examination Regulations):

1. R. Ohlendorf, A. Herkersdorf, T. Wild, **"FlexPath NP - A Network Processor Concept with Application-Driven Flexible Processing Paths"**, CODES+ISSS 2005, Jersey City, NJ, USA, September 19-21, 2005, ([7])

2. M. Meitinger, R. Ohlendorf, T. Wild, A. Herkersdorf, "Application Scenarios for FlexPath NP", Technical Report, TUM-LIS-TR-0501, December 2005, ([45])

3. T. Wild, A. Herkersdorf, R. Ohlendorf, "Performance Evaluation for System-on-Chip Architectures using Trace-based Transaction Level Simulation", DATE 2006, Munich, Germany, March 6-10, 2006, ([81])

4. A. Herkersdorf, C. Claus, M. Meitinger, R. Ohlendorf, "Reconfigurable Processing Units vs. Reconfigurable Interconnects", Dagstuhl Seminar on Dynamically Reconfigurable Architectures, Dagstuhl Seminar Proceedings 06141, Dagstuhl, Germany, April 2-7, 2006

5. R. Ohlendorf, T. Wild, M. Meitinger, H. Rauchfuss, A. Herkersdorf, **"Performance Evaluation of RISC-based SoC Platforms in Network Processing Applications"**, IC-SAMOS 2006, Samos, Greece, July 17-20, 2006, ([82])

6. M. Meitinger, R. Ohlendorf, T. Wild, A. Herkersdorf, "A Programmable Stream Processing Engine for Packet Manipulation in Network Processors", ISVLSI 2007, Porto Alegre, Brazil, May 9-11, 2007

7. R. Ohlendorf, T. Wild, M. Meitinger, H. Rauchfuss, A. Herkersdorf, **"Simulated and measured performance evaluation of RISC-based SoC platforms in network processing applications"**, Journal for Systems Architecture, vol. 53, no. 10, pp. 703-718, October 2007, ([83])

8. R. Ohlendorf, M. Meitinger, T. Wild, A. Herkersdorf, **"A Packet Classification Technique for On-Chip Processing Path Selection"**, WASP 2007, Salzburg, Austria, October 4-5, 2007, ([84])

9. M. Meitinger, R. Ohlendorf, T. Wild, A. Herkersdorf, "A Hardware Packet Resequencer Unit for Network Processors", ARCS 2008, Dresden, Germany, February 25-28, 2008

10. T. Pionteck, R. Koch, C. Albrecht, E. Maehle, M. Meitinger, R. Ohlendorf, T. Wild A. Herkersdorf, "SPP1148 Booth: Network Processors", FPL 2008, Heidelberg, Germany, September 8-10, 2008, p. 352, DOI: 10.1109/FPL.2008.4629960

11. R. Ohlendorf, M. Meitinger, T. Wild, A. Herkersdorf, **"A Processing Path Dispatcher in Network Processor MPSoCs"**, IEEE Transactions on VLSI Systems, vol. 16, no. 10, pp. 1335-1345, October 2008, ([56])

12. M. Meitinger, R. Ohlendorf, T. Wild, A. Herkersdorf, "FlexPath NP - A Network Processor Architecture with Flexible Processing Paths", SoC 2008, Tampere, Finland, November 4-6, 2008, ([106])

13. S. Hauger, T. Wild, A. Mutter, A. Kirstädter, K. Karras, R. Ohlendorf, F. Feller, J. Scharf, "Packet Processing at 100 Gbps and Beyond - Challenges and Perspectives", 10. ITG Fachtagung Photonische Netze, Dresden, Germany, May 4-5, 2009 ([5])

14. S. Traboulsi, M. Meitinger, R. Ohlendorf, A. Herkersdorf, "An Efficient Hardware Architecture for Packet Re-sequencing in Network Processor MPSoCs", 12th Euromicro Conference on Digital System Design (DSD'09), Patras, Greece, August 27-29, 2009

15. R. Ohlendorf, M. Meitinger, T. Wild, A. Herkersdorf, **"An Application-aware Load Balancing Strategy for Network Processors"**, HiPEAC 2010, Pisa, Italy, January 25-27, 2010, ([72])

16. M. Platzner, J. Teich, N. Wehn (Eds.), "Dynamically Reconfigurable Systems: Architectures, Design Methods and Applications", Chapter 17 - FlexPath NP - Flexible, Dynamically Reconfigurable Processing Paths in Network Processors, pp. 355-374, ISBN 978-90-481-3484-7, Springer Science+Business Media B.V. 2010, ([31], chapter 17)

17. A. Herkersdorf, A. Lankes, M. Meitinger, R. Ohlendorf, S. Wallentowitz, T. Wild, J. Zeppenfeld, "Hardware Support to Exploit Parallelism in Homogeneous and Heterogeneous Multi-Core Systems on Chip", chapter in "Multiprocessor System-on-Chip: Hardware Design and Tool Integration", ISBN 978-1441964595, © Springer, Berlin, November 2010