TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Integrierte Systeme

Lastbalancierung und Resequenzierung in Netzwerkprozessoren

Dipl.-Ing. Univ. Michael Meitinger

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. Ulf Schlichtmann
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 13.04.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 01.12.2011 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:

Lastbalancierung und Resequenzierung in Netzwerkprozessoren

am Lehrstuhl für Integrierte Systeme unter der 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 ist mir bekannt.

München, den 17.02.2012

Unterschrift

Kurzzusammenfassung

Die vorliegende Arbeit beschäftigt sich mit der Optimierung der Architektur von Netzwerkprozessoren. Netzwerkprozessoren sind eine spezielle Klasse von Prozessoren zur Paketverarbeitung, die an unterschiedlichen Knoten innerhalb eines Netzwerkes (z.B. Router) ihren Dienst verrichten. Die Anforderungen an Netzwerkprozessoren umfassen dabei neben einer möglichst hohen Rechenleistungsdichte eine ausreichende Flexibilität um wechselnde Einsatzzwecke sowie zukünftige Entwicklungen zu unterstützen.

Die Arbeit entstand im Rahmen des FlexPath-Netzwerkprozessor-Projekts. Das FlexPath-Projekt hat sich zum Ziel gesetzt, durch eine intelligente Pfadentscheidung, Pakete auf optimierten Wegen durch den Netzwerkprozessor zu leiten und so die Leistungsfähigkeit zu erhöhen. Standardaufgaben werden dabei durch eine Hardware-Implementierung abgedeckt, während ein zusätzliches Prozessor-Cluster für ein ausreichendes Maß an Flexibilität sorgt und so die Umsetzung komplexer Aufgaben ermöglicht.

Konkret wird ein Verfahren zur Lastbalancierung zustandslosen Verkehrs im Prozessor-Cluster vorgestellt, welches sowohl Verlustraten als auch Verarbeitungslatenzen gegenüber bekannten Verfahren deutlich senkt. Zustandsloser Verkehr kann dabei grundsätzlich von jedem Prozessor bearbeitet werden, allerdings gilt es dabei aus Gründen der Verbindungsqualität die Paketreihenfolge zusammengehöriger Pakete bestmöglich zu erhalten. Da dies durch das vorgestellte *Spraying* nur bedingt möglich ist, wird zudem ein effizientes Verfahren zur hardwaregestützten Resequenzierung vorgestellt. Ein weiterer Schwerpunkt der Arbeit befasst sich mit der Umsetzung der Paketverteilung im Prozessor-Cluster. Zudem wird eine Paketmodifikationseinheit vorgestellt, welche in Zusammenarbeit mit anderen FlexPath-Komponenten einen vollständigen Hardwareverarbeitungspfad für bestimmte Pakete zur Verfügung stellt.

Die vorgestellten Verfahren wurden ausgiebig mithilfe eines SystemC-Simulationsmodells getestet und evaluiert. Zudem wird eine FPGA-Implementierung des FlexPath-Netzwerkprozessors vorgestellt, mit dem anhand von Messungen die aufgestellten Vermutungen bestätigt werden. Die Implementierung dient zudem als Beweis der Umsetzbarkeit der vorgeschlagenen Module.

Danksagung

An dieser Stelle möchte ich allen danken, die auf direkte oder indirekte Art zur Entstehung dieser Arbeit beigetragen haben.

Zunächst geht mein Dank an Prof. Andreas Herkersdorf, der mir die Gelegenheit gab dieses interessante Thema zu bearbeiten. Hierbei konnte ich nicht nur meine fachlichen Kenntnisse vertiefen, sondern gerade auch in der Zusammenarbeit mit Studenten und in der Lehre wertvolle Erfahrungen sammeln. Ein herzliches Dankeschön sei hierbei auch für die Unterstützung ausgesprochen, die es mir ermöglichte, Arbeit und Familie bestmöglich zu vereinbaren. Mein Dank gilt ferner Prof. Erik Maehle von der Universität zu Lübeck, der sich als Zweitprüfer dieser Arbeit bereitgestellt hat.

Ein herzliches Dankeschön geht an die Mitarbeiter des Lehrstuhls Integrierte Systeme. Vorneweg seien hier Dr. Thomas Wild, Rainer Ohlendorf und Daniel Llorente als Mitglieder der Networking-Gruppe genannt. Die zahlreichen Diskussionen trugen erheblich zum Fortschritt dieser Arbeit bei. Bedanken möchte ich mich auch bei Holm, Stefan, Michael, Johannes, Christopher und Robert für die stets gute Unterstützung, sowohl bei technischen als auch administrativen Problemen. Mein Dank geht ferner an unsere Damen im Sekretariat für die stets schnelle Hilfe bei verwaltungstechnischen Problemen, Prof. Walter Stechele für die gute Zusammenarbeit insbesondere in der Lehre, sowie allen aktuellen aber auch ehemaligen Mitarbeitern, die zu einer sehr angenehmen Arbeitsatmosphäre beitrugen.

Mein Dank geht ferner an die „ITI-Truppe“ aus Lübeck – Prof. Thilo Pionteck, Dr. Carsten Albrecht und Roman Koch – für die gute und angenehme Kooperation.

Zu guter Letzt möchte ich meiner ganzen Familie für die Unterstützung während der letzten Jahre danken. Ganz besonders möchte ich hierbei meine Frau Claudia hervorheben, die mir stets den Rücken stärkte, aber auch meine Kinder Johanna und Katharina, die mir die nötige Motivation verschafften und mich vorantrieben.

Inhaltsverzeichnis

1 Einleitung **13**
- 1.1 Entwicklung des Internets . . . 13
- 1.2 FlexPath - ein Ansatz mit flexiblen, rekonfigurierbaren Verarbeitungspfaden 14
- 1.3 Struktur der Arbeit . . . 15

2 Grundlagen der Paketverarbeitung **17**
- 2.1 Das OSI-Schichtenmodell . . . 17
- 2.2 Definitionen . . . 18
- 2.3 Verarbeitungsprotokolle . . . 19
  - 2.3.1 Layer 1/2: Ethernet . . . 19
  - 2.3.2 Layer 3: Internet Protocol . . . 21
  - 2.3.3 Layer 4: TCP/UDP . . . 24
  - 2.3.4 Sicherheitsprotokolle . . . 26
  - 2.3.5 Quality of Service . . . 29
  - 2.3.6 Deep Packet Inspection . . . 30
  - 2.3.7 Routingprotokolle . . . 30
- 2.4 Netztopologie . . . 31
  - 2.4.1 Local Area Networks . . . 31
  - 2.4.2 Zugangsnetz . . . 33
  - 2.4.3 Wide Area Networks . . . 34
  - 2.4.4 Anforderungen an paketverarbeitenden Instanzen im Netz . . . 35

3 Stand der Technik **37**
- 3.1 Netzwerkprozessoren . . . 37
  - 3.1.1 Überblick über kommerzielle Netzwerkprozessoren . . . 39
  - 3.1.2 Überblick über akademische Forschung zu Netzwerkprozessoren . . . 43
  - 3.1.3 Zusammenfassung . . . 46
- 3.2 Lastbalancierung in Netzwerkprozessoren . . . 46
  - 3.2.1 Statische Lastbalancierungsverfahren . . . 47
  - 3.2.2 Dynamische Lastbalancierungsverfahren . . . 48
  - 3.2.3 Zusammenfassung . . . 50
- 3.3 Packet Reordering und Resequenzierung im Netzwerkbereich . . . 51
  - 3.3.1 Paket-Resequenzierung in Load Balanced Switches . . . 52
  - 3.3.2 Paket-Resequenzierung in Netzwerkprozessoren . . . 54
  - 3.3.3 Zusammenfassung . . . 59
- 3.4 Interrupt Controller für Multiprozessor-Systeme . . . 59

4 FlexPath-Netzwerkprozessor 63
4.1 Konzept . . . 64
4.1.1 Hardware-Unterstützung . . . 64
4.1.2 Paketabhängige Pfadwahl . . . 66
4.1.3 Rekonfiguration der Pfadentscheidung . . . 67
4.1.4 Unterstützung von Quality of Service . . . 68
4.2 Beschreibung der Einzelkomponenten und des Paketflusses . . . 69
4.3 Paketmanipulation im Ausgangsdatenpfad . . . 71
4.3.1 Motivation . . . 71
4.3.2 Konzept: Datenmanipulation am Paketdatenstrom . . . 73
4.3.3 Architektur: Modulare Verarbeitungspipeline . . . 75
4.3.4 Implementierung . . . 82
4.4 Konzeptevaluierung . . . 84
4.4.1 SystemC-Simulationsmodell . . . 84
4.4.2 Simulationsergebnisse . . . 86
4.5 Bewertung . . . 88

5 Paketverteilung und Lastbalancierung im Multiprozessor-Cluster 91
5.1 Motivation . . . 91
5.2 Lastbalancierungsstrategien . . . 93
5.2.1 Lastbalancierungsstrategie für zustandslosen Verkehr . . . 93
5.2.2 Lastbalancierung im FlexPath-NP . . . 95
5.3 Paketverteilung im Multiprozessor-Cluster . . . 98
5.3.1 Motivation . . . 98
5.3.2 Gegenüberstellung von Interrupt und Polling . . . 99
5.3.3 Multiprocessor Interrupt Controller . . . 100
5.3.4 Packet Distributor . . . 104
5.3.5 Externe Interrupts und Zuordnung der Paket-Queues im FlexPath-NP . . . 108
5.3.6 Implementierung . . . 109
5.4 Konzeptevaluierung . . . 114
5.4.1 Funktionales Simulationsmodell . . . 114
5.4.2 Verkehrs-Stimuli . . . 116
5.4.3 Simulationsergebnisse . . . 117
5.5 Bewertung . . . 127

6 Paket-Resequenzierung im FlexPath-NP 129
6.1 Motivation . . . 129
6.2 Konzept der Hardware-Resequenzierung . . . 131
6.2.1 Anpassungen am funktionalen Simulationsmodell . . . 132
6.2.2 Speicherbedarf/-organisation und Sortieralgorithmus . . . 133
6.2.3 Ingress Tagger . . . 137
6.2.4 Aggregation Unit . . . 137

6.3 Dimensionierungsaspekte . . . 142
6.3.1 Hashwert-Kollisionen und Flow-ID-Bitbreite . . . 142
6.3.2 Dimensionierung bei homogenem Verkehr . . . 144
6.3.3 Anpassungen für heterogenen Verkehr . . . 146
6.3.4 Wahl der Zeitspanne beim Timeout . . . 148
6.4 Simulationen und Evaluierung der Hardware-Resequenzierung . . . 150
6.4.1 Szenario 1: Forwarding-Verkehr . . . 151
6.4.2 Szenario 2: Verkehrs-Mix . . . 151
6.5 Implementierung . . . 156
6.5.1 Ressourcen-Verbrauch . . . 158
6.5.2 Paket- und Datendurchsatz . . . 159
6.6 Ausblick: Adaptive Anpassung der Timeout-Zeit . . . 161
6.6.1 Algorithmus . . . 161
6.6.2 Simulation . . . 162
6.6.3 Bewertung des Verfahrens . . . 165
6.7 Bewertung und Zusammenfassung . . . 166

7 FlexPath-Demonstrator 169
7.1 Demonstrator-Plattform: Xilinx Development Board ML410 . . . 169
7.2 Implementierung des FlexPath-NPs . . . 171
7.2.1 Implementierung der Hardware . . . 171
7.2.2 Implementierung des Network Processing Software Stacks . . . 173
7.3 Messungen am FlexPath-NP-Demonstrator . . . 174
7.3.1 Versuchsaufbau . . . 175
7.3.2 Optimierung am Packet Distributor . . . 176
7.3.3 IP-Forwarding unter Verwendung des Post-Processors . . . 178
7.3.4 Einfluss des Softwareeinstiegspunkts auf den Datendurchsatz . . . 181
7.3.5 Packet Reordering und Hardware-Resequenzierung bei Spraying . . . 183
7.3.6 Packet Spraying vs. dedizierte Lastbalancierung . . . 185
7.4 Zusammenfassung . . . 190

8 Zusammenfassung und Ausblick 193
8.1 Zusammenfassung . . . 193
8.2 Ausblick . . . 195

1 Einleitung

1.1 Entwicklung des Internets

Der weltweite Datenverkehr hat aufgrund der zunehmenden Fülle an Diensten und Anwendungen in den letzten Jahren drastisch zugenommen. Anwendungen wie Internet-Browsing sind mittlerweile fest ins Alltagsleben integriert, aber auch der Einsatz von *peer-to-peer* (P2P) Datendiensten sorgt für zunehmenden Verkehr. Daneben haben Anwendungen mit vertraulichen Daten wie Internet Banking eine kontinuierlich Zunahme. Verschlüsselungsprotokolle wie IPsec ermöglichen den Transfer sensibler Daten weltweit. Voice-over-IP (VoIP) erlaubt eine kostengünstige Kommunikation weltweit. Und letztlich verlagern sich auch die klassischen Übertragungswege von Rundfunk und Fernsehen von Kabel und Satellit zunehmend in Richtung Paketdatenverkehr (IPTV, *Video on Demand*).

Dabei steigt nicht nur die Datenrate pro Teilnehmer, sondern auch die Anzahl der Teilnehmer selbst rasant an. Mehr und mehr Menschen erhalten Zugang zum weltweiten Netz. Im Jahr 2010 waren einer Schätzung des *Bundesverbandes Informationswirtschaft, Telekommunikation und neuen Medien* zufolge etwa 1,5 Millarden Menschen online. Dabei werden jährliche Wachstumsraten von etwa 7-8% (siehe [1]) erreicht!

So verwundert auch eine Marktanalyse der Firma Cisco [2] nicht, die derzeit eine durchschnittliche Steigerungsrate des Datenverkehrs von 58% pro Jahr bei den Privathaushalten und etwa 21% bei Unternehmen voraussagt. Dieser Analyse zufolge wird der Verkehr der Privathaushalte von 1,9 Millionen Terabyte pro Monat im Jahr 2006 auf 18,3 Millionen Terabyte pro Monat im Jahr 2011 ansteigen. Der Anteil der institutionellen Nutzung wird dann bei 10,6 Millionen Terabyte pro Monat liegen.

Diese Zahlen geben einen Eindruck davon, wie schnell sich Internet und weltweite Datenkommunikation nach wie vor entwickeln. Das Internet besitzt also - auch wenn es mittlerweile wie selbstverständlich in unserem Alltag integriert ist - nach wie vor keine statische Infrastruktur, sondern ist geprägt durch ein ein hohes Maß an Dynamik und Entwicklung. Insgesamt lassen sich daraus drei wesentliche Herausforderungen an Netz und Infrastruktur definieren, die auch in Zukunft gelten:

- Die Infrastruktur muss den rasant ansteigenden Datenraten gewachsen sein. Das setzt genug **Rechenressourcen** voraus, sowie eine vorausschauende Planung, um auch mit dem Datenverkehr der nächsten Jahre zurechtzukommen.
- Wie schon in der Vergangenheit mehrfach gesehen, können innerhalb weniger Jahre neue Anwendungen entstehen, die einen nennenswerten Anteil am Gesamtverkehr produzieren. Das bedeutet im Umkehrschluss, dass schon heute Anwendungen tech-

nisch mit eingeplant werden müssen, die teilweise noch gar nicht bekannt sind, bzw. von denen man letztlich noch nicht abschätzen kann, inwieweit sie sich durchsetzen. Das erfordert ein hohes Maß an **Flexibilität**.

- Außerdem reicht eine Gleichbehandlung des Verkehrs (sogenannter *Best Effort* Verkehr) - wie sie in der Vergangenheit Standard war - in Zukunft nicht mehr aus. Vielmehr bedürfen eine Reihe von Anwendungen garantierte Zusagen an Durchsatz, Latenz oder Jitter (**Quality of Service**).

1.2 FlexPath - ein Ansatz mit flexiblen, rekonfigurierbaren Verarbeitungspfaden

Hohe Rechenleistung bei hoher Flexibilität sind im Allgemeinen widersprüchliche Anforderungen. Um beide Anforderungen dennoch besser miteinander zu verbinden, hat sich in den letzten 10 bis 15 Jahren eine neue Klasse von Verarbeitungseinheiten - die Netzwerkprozessoren - herausgebildet. Netzwerkprozessoren verrichten an den verschiedensten Stellen im Internet ihren Dienst. Ihre Hauptaufgabe ist das Routing von Paketen, aber auch Anforderungen bis hin zum *Deep Packet Inspection* werden integriert. Ihre Fähigkeiten variieren dabei stark je nach konkreten Einsatzzweck. Dementsprechend vielfältig ist auch die anzufindende Architekturvielfalt. Die Palette reicht von hoch leistungsfähigen, aber wenig flexiblen ASIC-Architekturen, bis hin zu weniger leistungsfähigen, aber sehr flexiblen CPU-Cluster-Ansätzen. Allerdings findet sich in der Regel auch bei CPU-basierten Ansätzen eine speziell für den Einsatzzweck optimierte Hardwareunterstützung.

In diesem Umfeld wurde im Rahmen des Schwerpunktprogramms 1148 *Rekonfigurierbare Rechensysteme* der Deutschen Forschungsgemeinschaft (DFG) das FlexPath-Projekt ins Leben gerufen und von April 2005 bis April 2009 gefördert.

Ziel des Projekts war dabei die Steigerung der Leistungsfähigkeit eines NPs bei Aufrechterhaltung der Flexibilität. Als Basis diente dabei ein zentrales CPU-Cluster. Durch Erweiterungen in Hard- und Software sollen nennenswerte Geschwindigkeitssteigerung in der Prozessierung erreicht werden.

Die grundsätzlichen Ideen des FlexPath-Projektes bestehen dabei aus einer **paketabhängigen Pfadwahl**, **Hardware-Unterstützung** bis hin zur kompletten Bearbeitung in Hardware, einer dynamischen **Rekonfiguration der Pfadentscheidung** bei Bedarf und Mechanismen zur besseren Unterstützung von **Quality of Service**.

Dabei stand nicht die Frage einer konkreten Implementierung im Vordergrund. Vielmehr sollten die gefundenen Ansätze ebenso möglichst leicht auf bestehende CPU-Cluster Architekturen übertragen werden können.

Es wird eine zweigleisige Strategie verfolgt: Standardaufgaben, die effizient in Hardware ausgelagert werden können, werden (weitestgehend) in Hardware bearbeitet, während komplexere Aufgaben wie Ausnahmebehandlungen, Routingprotokolle, Verbindungsaufbau und andere komplexe Protokolle weiterhin innerhalb des CPU-Clusters behandelt

werden. Diese Strategie erscheint vielversprechend, da ein Großteil der Pakete lediglich eine Standardverarbeitung benötigt und nur relativ wenige einer komplexeren Behandlung bedürfen.

Das FlexPath-Projekt selbst ist dabei Thema zweier parallel entstandener Dissertationen. Während sich Rainer Ohlendorf in [3] mit der Realisierung der Pfadentscheidung in Hardware, diverser Module im Eingangsdatenpfad und insbesondere auch um Algorithmen zur Lastbalancierung für zustandsbehafteten Verkehr beschäftigte, lassen sich die Schwerpunkte dieser Arbeit wie folgt definieren:

- Im Rahmen des Gesamtkonzeptes wurde eine **Verarbeitungseinheit im Ausgangsdatenpfad** entwickelt, die es ermöglicht, notwendige Paketmodifikationen *on-the-fly* in Hardware durchzuführen.
- Es wurde untersucht, wie die **Verteilung der Pakete innerhalb eines Multiprozessor-Cluster** erfolgen kann. Hierzu wurde im Rahmen der Untersuchungen zur Lastbalancierung ein effektives Verfahren zum Verteilen von zustandslosem Verkehr geschaffen. Zudem wurde eine Einheit entwickelt, die die Pakete effizient innerhalb des Multiprozessor-Systems verteilt.
- Die flexible Pfadwahl und die Verteilung der Pakete auf unterschiedliche CPUs führt teilweise zur Vertauschung der Paketreihenfolge innerhalb eines Flows. Dies hat jedoch nachweisbare Nachteile für die Netzwerkgeschwindigkeit. Zur Lösung des Problems wurde ein Verfahren entwickelt, das trotz geringen Ressourcenbedarfs die **originale Paketreihenfolge auf Flow-Ebene praktisch vollständig wiederherstellt**.

Der erste Punkt der Arbeit ist dabei sehr eng an das FlexPath-Konzept gebunden. Die weiteren Untersuchungen wurden zwar auf Basis der FlexPath-Architektur durchgeführt, können jedoch auch auf andere Architekturen übertragen und angewandt werden.

1.3 Struktur der Arbeit

Die weitere Arbeit gliedert sich wie folgt: In Kapitel 2 werden zunächst die Grundlagen der Paketverabeitung kurz dargestellt. Vom besonderen Interesse ist hierbei neben *Ethernet* u.a. das *Internet Protocol* (IP) sowie TCP und UDP. Kapitel 3 gibt einen Überblick über den für diese Arbeit relevanten Stand der Technik. In Kapitel 4 wird das FlexPath-Konzept vorgestellt, sowie die Datenmanipulationseinheit im Ausgangsdatenpfad. Kapitel 5 beschäftigt sich mit Lastbalancierungsstrategien im Netzwerkprozessor, sowie der Umsetzung in einem Multiprozessor-System. In Kapitel 6 wird eine Hardware Resequenzierungs-Einheit zur Wiederherstellung der Paketreihenfolge nach der Prozessierung vorgestellt. Im Kapitel 7 wird schließlich der FlexPath-FPGA-Demonstrator beschrieben. Mithilfe des Demonstrators werden unterschiedliche Messreihen im Hardware-System durchgeführt und deren Ergebnisse vorgestellt. In Kapitel 8 wird die Arbeit schließlich zusammengefasst und ein Ausblick für mögliche weitere Entwicklungen gegeben.

2 Grundlagen der Paketverarbeitung

In diesem Kapitel werden zunächst die Grundlagen der Paketverarbeitung behandelt. Ausgehend vom OSI-Referenzmodell wird neben verschiedenen Protokollen wie *Internet Protocol* (IP), TCP oder UDP, auch auf andere Aspekte wie *Quality of Service*, *Deep Packet Inspection* und Sicherheitsanwendungen im Netz eingegangen. Anschließend wird der typische Aufbau eines Netzwerks beschrieben, um die Einsatzmöglichkeiten und Anforderungen an Netzwerkprozessoren besser auszuleuchten.

2.1 Das OSI-Schichtenmodell

Das *Open Systems Interconnection Reference Model* [4] (OSI-Referenzmodell, siehe Abbildung 2.1) der *Internationalen Standardisierungsorganisation* (ISO) ist ein 1983 eingeführtes Modell zur Beschreibung von Kommunikationsprotokollen. Das Referenzmodell soll einen Rahmen vorgeben, um die Vielzahl von Protokollen und Netzwerktechnologien der verschiedenen Anbieter miteinander zu verbinden (siehe [5]). Grundlage ist dabei eine Aufteilung des Protokollstacks in sieben Schichten (*Layer*). Im Referenzmodell wird festgelegt, welche Dienste die jeweilige Schicht zur Verfügung stellen muss. Es handelt sich hierbei aber um keine konkreten Implementierungsvorgaben. Die Schichten 1-3 sind netzorientiert und stellen u.a. das Übertragungsmedium zur Verfügung. Die Schichten 5-7 sind anwendungsorientiert und spezifizieren die höheren Kommunikationsprotokolle. Schicht 4 (Transportschicht) ist die Schnittstelle zwischen Anwendung und Netz und ermöglicht somit die Verbindung zweier Endsysteme auch über verschiedene Übertragungsmedien und Netzprotokolle hinweg.

Die Verarbeitungskomplexität der einzelnen Schichten nimmt dabei von Schicht zu Schicht zu. Während sich insbesondere die unteren Schichten bis Schicht 3 sehr leicht durch generische Hardware unterstützen lassen, ist für die höheren Schichten zunehmend der Einsatz von spezifischen Hardwarebeschleunigern oder flexibel programmierbaren Prozessoren notwendig. Eine Unterstützung durch Hardware wird also generell schwieriger bzw. spezifischer. Um einen möglichst breiten Anwendungsbereich zu erhalten, liegt der Schwerpunkt dieser Arbeit deshalb auf Schicht 3.

Schicht 3 ermöglicht dabei Punkt-zu-Punkt (*Point-to-Point*) Verbindungen über ein bzw. mehrere Übertragungsmedien hinweg. Der Vorgang der Wegfindung eines Pakets durch ein Netz wird dabei als *Routing* bezeichnet. Die einzelnen Instanzen, die dabei passiert werden, bezeichnet man entsprechend als *Router*. Das physikalische Übertragungsmedium selbst ist auf Schicht 1 und 2 angesiedelt. Dieses stellt lediglich eine verbindungslose Kommunikation zur Verfügung. Eine Weiterleitung auf dieser Ebene be-

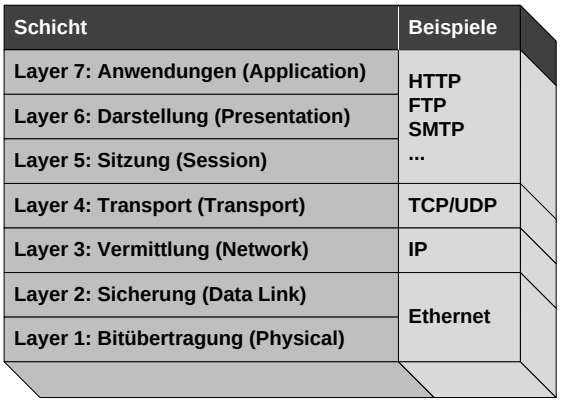

Abbildung 2.1: OSI Referenzmodell

zeichnet man als *Switching*. Das wohl bekannteste physikalische Übertragungsmedium ist das Ethernet, das vor allem in lokalen Netzen, mittlerweile aber auch verstärkt im *Backbone*-Bereich zum Einsatz kommt.

2.2 Definitionen

Im Rahmen dieser Arbeit wird eine Reihe von Begriffen verwendet, deren genaue Definition wichtig für das Verständnis der Arbeit ist und die im Folgenden erläutert werden.

Flow

Ein Flow kann allgemein als eine Menge von Paketen mit gemeinsamen Eigenschaften angesehen werden. Eine übliche und im Folgenden verwendete Definition von Flow beschreibt eine Menge von Paketen mit identischem IP 5-Tupel. Das IP 5-Tupel beinhaltet die Quell- und Ziel-IP-Adresse des Pakets, Ports und die Layer 4-Protokollnummer. Ein Flow ist damit üblicherweise die Menge von Paketen, die zu einer gemeinsamen Verbindung (z.B. TCP-Verbindung) gehören. Sofern keine Ports vorhanden sind oder diese (wie bei IPsec) nicht unverschlüsselt gelesen werden können, wird alternativ auf das IP 3-Tupel (IP-Adressen plus Protokollnummer) zurückgegriffen.

Burst

Ein Burst (engl. für Häufung) beschreibt im Rahmen dieser Arbeit die oft beobachtete Eigenschaft in der Paketübertragung, dass Pakete eines Flows nicht gleichmäßig über der Zeit verteilt eintreffen, sondern oftmals gebündelt. Damit wechseln sich Zeiten mit relativer Inaktivität ab mit Zeiten, bei denen innerhalb kurzer Zeit viele Pakete eines Flows eintreffen. Ein einzelnes, zeitlich befristetes Paketbündel nennt man dabei Burst.

Packet Reordering

Packet Reordering beschreibt das Vertauschen der originalen Paketreihenfolge **innerhalb** eines Flows. *Packet Reordering* kann anhand einer im Paket mitgeführten konsekutiven Paket-Sequenznummer (z.B. TCP-Sequenznummer) erkannt werden. In Übereinstimmung mit RFC4737 [6] gilt ein Paket als *out-of-order*, sofern die Sequenznummer des Pakets kleiner als die erwartete Sequenznummer ist. Als Pseudo-Code lässt sich die Erkennung folgendermaßen ausdrücken:

```
if s >= NextExp then
  NextExp = s + 1;
  Reordered = False;
else
  Reordered = True;
```

Dabei gilt: `NextExp` beschreibt die als nächstes erwartete Sequenznummer, `s` die Sequenznummer des aktuellen Pakets. Damit werden also in der Reihenfolge zu spät eintreffende Pakete als *out-of-order* (`Reordered = True`) gewertet.

Paket-Resequenzierung

Paket-Resequenzierung beschreibt den Vorgang der Wiederherstellung der originalen Paketreihenfolge eines Flows. Es handelt sich hierbei also um den umgekehrten Vorgang zu *Packet Reordering*.

2.3 Verarbeitungsprotokolle

In diesem Abschnitt wird besonderes Augenmerk auf die Bedeutung der einzelnen Schichten und Protokolle für die Verarbeitung innerhalb eines Routers gelegt. Gemäß der zunehmenden Bedeutung - auch in Weitverkehrsnetzen (siehe u.a. [7]) - wird die Betrachtung des physikalischen Übertragungsmediums (Layer 1/2) auf Ethernet limitiert. Neben dem *Internet Protocol* (IP), werden auch die wichtigsten Protokolle der Schicht 4 betrachtet. Außerdem werden Sicherheitsanwendungen und *Quality of Service* kurz erläutert.

2.3.1 Layer 1/2: Ethernet

Ethernet ist eine kabelgebundene Datenübertagungstechnik spezifiert in der IEEE Norm 802.3 [8]. Es basiert auf dem *Carrier Sense Multiple Access/Collision Detection* (CSMA/CD) Prinzip. Es handelt sich um einen paketorientierten Zugriff auf ein gemeinsames Medium. Da der Zugriff nicht näher geregelt ist, können mehrere Teilnehmer gleichzeitig sendend auf das Übertragungsmedium zugreifen, wodurch eine Kollision entsteht. Diese werden erkannt und eine Übermittlung nach bestimmten Regeln wiederholt.

Diese Art der Kollisionskontrolle spielt heutzutage aber nurmehr eine untergeordnete Rolle. Der Einsatz von Switches und der Betrieb im Vollduplexmodus erlaubt eine kollisionsfreie Übertragung. Jeder Teilnehmer ist direkt an einen Switch angeschlossen. Damit wird jede Leitung nur von einem einzigen Sender belegt. Mögliche Kollisionen werden durch Paketpuffer im Switch aufgelöst.

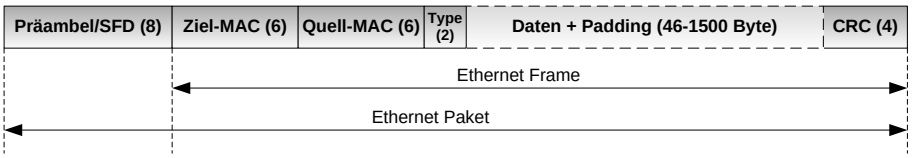

Abbildung 2.2: Aufbau eines *Basic Ethernet Frames* nach IEEE 802.3 3.1a. Längenangaben in Byte.

Der Aufbau eines Ethernet-Frames ist in Abbildung 2.2 gezeigt. Ein Frame beginnt mit einer festgelegten Präambel gefolgt von einem *Start Frame Delimiter* (SFD), der den Beginn des eigentlichen Pakets definiert. Die Präambel dient zur Bit-Synchronisation des Empfängers. Da heute nahezu nur noch synchrone Punkt-zu-Punkt Verbindungen verwendet werden, sind die beiden Felder streng genommen überflüssig. Aus Gründen der Kompatibilität werden sie aber nach wie vor verwendet. Es folgt eine sechs Byte große Ziel-MAC-Adresse (*Media Access Control*), also die Ethernet-Adresse des Empfängers, gefolgt von der Adresse des Senders. Das *Ethertype*-Feld gibt an, welches Protokoll im Datenfeld folgt. Für IP wird beispielsweise der Wert 0x0800 verwendet. Nach den eigentlichen Paketdaten folgt noch eine vier Byte große CRC-Prüfsumme, die die Datenintegrität des Pakets garantiert.

Optional können mit Ethernet Virtuelle Netzwerke (VLANs) gebildet werden. In diesem Fall folgt den Adressen noch ein 32 Bit breites VLAN-Tag.

Jedes Gerät in einem Netz muss seine eigene, eindeutige Adresse besitzen. In der Regel ist dies bereits herstellungsbedingt der Fall. Die ersten drei Bytes der Adresse geben dabei die Herstellerkennung wieder, während die restlichen drei Bytes vom Hersteller frei definiert werden können. Über die Adresse *FF-FF-FF-FF-FF-FF* ist ein Broadcast an alle angeschlossenen Geräte möglich. Ferner sind noch Adressbereiche für Multicast-Anwendungen reserviert.

Nach der Übertragung eines Ethernet-Frames folgt eine Sendepause entsprechend der Länge von mindestens 12 Byte bis zum nächsten Frame. Diese *Interframe Gap* gibt dem Empfänger eine gewisse Zeit, den vorherigen Frame zu verarbeiten, ehe er erneut empfangsbereit sein muss. *Interframe Gap*, MAC-Adressen, *Ethertype* und CRC-Prüfsumme ergeben zusammenaddiert somit einen Overhead von 38 Byte pro Paket. In Abhängigkeit von der Payloadgröße ergibt sich somit bei einer Linkrate von 1 GBit/s eine Nutzdatenrate zwischen 548 MBit/s bei der minimalen Ethernet-Framegröße von 64 Byte (vgl. Abbildung 2.3) und 975 MBit/s bei 1.518 Byte großen Paketen.

Ethernet ist derzeit für Geschwindigkeiten von 10 MBit/s bis 10 GBit/s spezifiziert. Als Übertragungsmedium kommen neben Kupferleitungen auch Glasfaserkabel zum Ein-

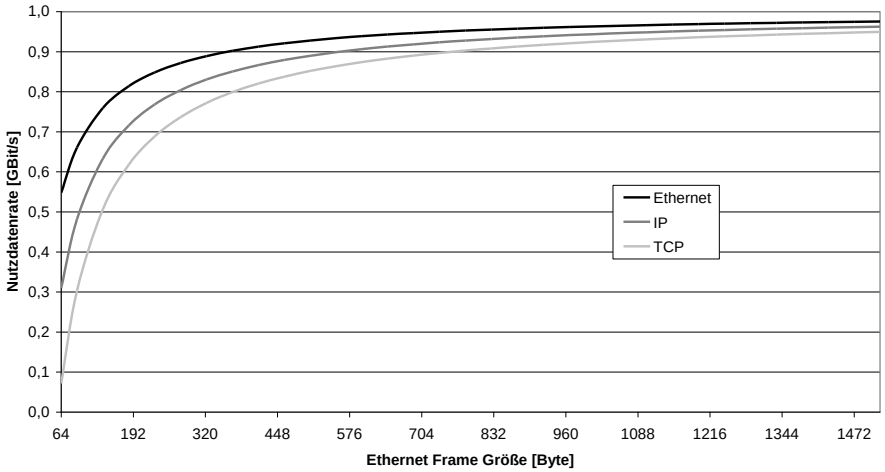

Abbildung 2.3: Nutzdatenrate von Ethernet, IP und TCP in Abhängigkeit der Ethernet-Framegröße bei einer Linkrate von 1 Gbit/s

satz.

2.3.2 Layer 3: Internet Protocol

Das *Internet Protocol* (IP) ist als Protokoll der Vermittlungsschicht für die Weiterleitung von Datenpaketen zuständig. Die Übermittlung der Pakete erfolgt verbindungslos. Jedes Paket enthält im vorangestellten Header die zum Routing notwendingen Information, insbesondere die IP-Adressen. Somit kann jedes Paket individuell durch das Netzwerk geroutet werden. Es ist dabei nicht sichergestellt, das jedes Paket den gleichen Weg verwendet. Dadurch kann es zu Paketüberholungen kommen, die auf IP-Ebene nicht aufgelöst werden. IP garantiert dabei keine zuverlässige Übermittlung. Es ist weder ein Quittierungsmechanismus implementiert, der dem Sender den Empfang der Pakete beim Empfänger bestätigt, noch ist die Datenintegrität gewährleistet. Damit können die Daten auf dem Transport entweder unabsichtlich (durch Fehler) oder absichtlich (durch Manipulation) verändert werden, ohne dass der Empfänger dies auf IP-Ebene erkennen kann. Mechanismen zur Datenintegrität müssen demnach auf den höheren Schichten implementiert werden.

Internet Protocol Version 4

IP in seiner ursprünglichen Version 4 (IPv4) wurde 1981 in der RFC791 [9] spezifiziert.

Die angeschlossenen Netzwerkgeräte sind im Allgemeinen eindeutig durch eine eigene IP-Adresse identifiziert. Bei IPv4 handelt es sich hierbei um eine 32 Bit-Adresse (siehe Abbildung 2.4). Somit stehen etwas mehr als vier Milliarden Adressen zur Verfügung.

Teilnehmer innerhalb eines gemeinsamen Subnetzes kommunizieren bei IP direkt mit-

einander. Dafür verwaltet jeder Teilnehmer eine entsprechende Auflösungstabelle zwischen IP-Adresse und Ethernet-Adresse. Ist einem Teilnehmer die Ethernet-Adresse eines anderen Teilnehmers noch nicht bekannt, kann diese anhand des *Address Resolution Protocols* (siehe [10]) ermittelt werden.

Bei der Kommunikation über Subnetzgrenzen hinweg, kommen sogenannte Router zum Einsatz. Diese haben in der Regel mehrere Netzwerkschnittstellen und sind somit Teilnehmer mehrerer Subnetze. Im Beispiel von Abbildung 2.5 wird ein Paket von Teilnehmer 192.168.0.55 an 129.187.155.218 gesendet Da der Quellrechner das Ziel nicht direkt erreichen kann, erhält zunächst der Router das Paket. Dieser entscheidet anhand seiner Routingtabelle an welchen seiner Anschlüsse das Paket weitergeleitet werden muss. In diesem Beispiel gelangt das Paket direkt über den Router in das richtige Subnetz und danach an den Empfänger. Kann der Router das Subnetz dagegen selbst wiederum nicht direkt erreichen, wird das Paket an den nächsten Router weitergeleitet, der ebenfalls eigenständig entscheidet auf welchen Wege das Ziel-Subnetz am besten zu erreichen ist. Jedes Durchlaufen eines Routers wird dabei als *Hop* bezeichnet. Die Zielfindung ist dabei keine triviale Aufgabe. Neben der Erreichbarkeit einzelner Router (z.B. bei Ausfall eines Routers), hat auch die Auslastung der einzelnen Netzwerkzweige Auswirkungen auf die Zielfindung. Zur Wegfindung und damit zur Pflege der Routingtabelle kommen unterschiedliche Routingprotokolle wie OSPF, BGP oder RIP zum Einsatz (vgl. Abschnitt 2.3.7).

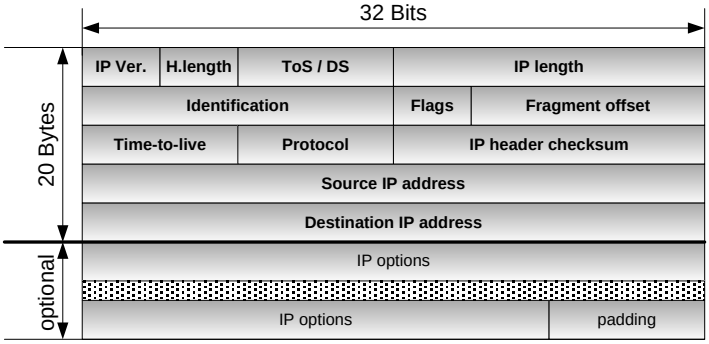

Abbildung 2.4: Aufbau des *Internet Protocol Version 4* (IPv4) Headers.

Der IP-Paketheader (siehe Abbildung 2.4) enthält neben Ziel- und Quell-IP-Adresse außerdem die Länge des IP-Pakets, eine Checksumme über den Header und ein *Time-to-live* Feld. Dieses Feld wird bei jedem *Hop* dekrementiert. Bei Erreichen des Wertes 0 wird das Paket verworfen und eine entsprechende Benachrichtigung an den Sender zurückgeschickt. Dadurch kann verhindert werden, dass bei ungünstigen Bedingungen (z.B. Nichterreichbarkeit des Zielsubnetzes) Pakete endlos in einem Netz kreisen. Mit dem *Type of Service*-Feld lässt sich eine Priorisierung des Pakets erreichen. Das Protokoll-Feld schließlich markiert die nächste Protokollschicht (z.B. TCP, UDP). Die optionalen *IP options* sind in der Realität von untergeordneter Bedeutung.

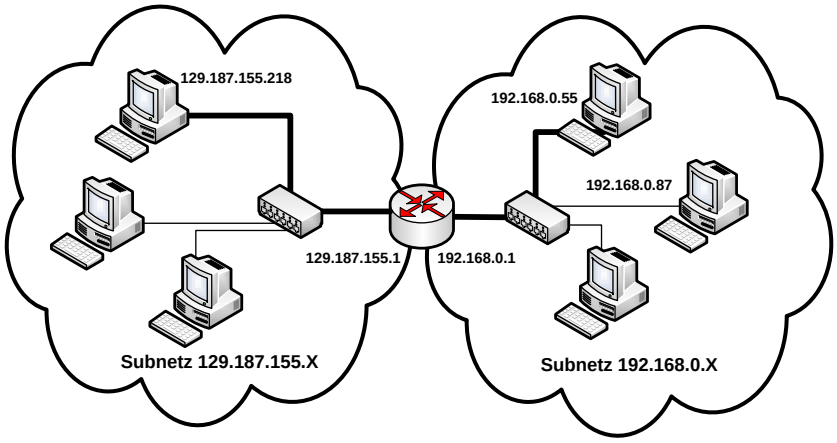

Abbildung 2.5: Beispiel für IP-Routing mit zwei Subnetzen.

Ein IP-Router muss für jedes Paket zur Weiterleitung (IP-Forwarding) mindestens folgende Schritte ausführen:

- Berechnen der Header-Checksumme zur Überprüfung der Datenintegrität des Headers.
- Ausführen eines *Next-hop Lookups*, d.h. Bestimmen des nächsten Zielrouters anhand des Zielsubnetzes.
- Ersetzen der Ziel-MAC-Adresse durch die Ethernet-MAC-Adresse des Zielrouters.
- Ersetzen der Quell-MAC-Adresse durch die eigene Ethernet-MAC-Adresse des jeweiligen Ausgangsports.
- Dekrement des *Time-to-live* (TTL) Feldes.
- Neuberechnung der IP Header-Checksumme. Nachdem die einzige Veränderung des Headers das dekrementierte TTL-Feld ist und die Checksumme ein CRC ist, kann dies inkrementell erfolgen.

Zusätzlich muss bei Ethernet noch für jedes Paket die Ethernet-Prüfsumme aktualisiert werden. Diese Funktionalität ist dabei in der Regel im Ethernet-MAC realisiert. Nachdem ein Router keine Vorkenntnisse über einen Flow benötigt und somit jedes Paket unabhängig bearbeitet werden kann, spricht man bei konventionellem IP-Forwarding auch von **zustandslosem Verkehr**.

Internet Protocol Version 6

Das *Internet Protocol* Version 6 wurde in der RFC 2460 [11] als Nachfolger des IPv4 spezifiziert. Der wesentliche Unterschied zum Vorgänger und damit auch der Hauptgrund für die neue Version liegt in der Aufweitung der Adressen von 32 auf 128 Bit (siehe Abbildung 2.6), da mit zunehmender Teilnehmerzahl und einer Vielzahl von netzwerkfähigen Geräten der Adressraum bei IPv4 in Kürze vollkommen ausgeschöpft zu sein droht. Die-

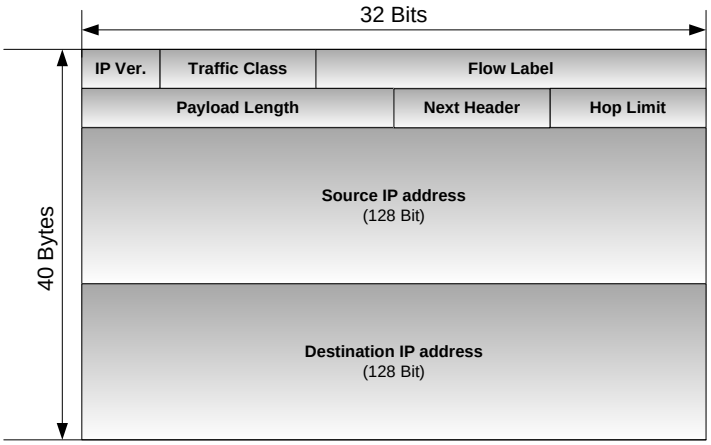

Abbildung 2.6: Aufbau des *Internet Protocol Version 6* (IPv6) Headers

ses Problem wäre mit der neuen Version schlagartig beseitigt. Dennoch versucht man durch Techniken wie *Classless Routing* oder *Network Address Translation (NAT)* den IPv4-Adressraum optimal auszunutzen und damit den endgültigen Umstieg auf IPv6 hinauszuzögern. Ein Grund liegt hierbei sicherlich in den Kosten zur Vorbereitung der Hardwareinfrastruktur bei den Netzbetreibern.

Viele Felder des IPv4-Headers findet man auch bei IPv6 in zum Teil leicht abgewandelter Form wieder. Für die Priorisierung wurde das *Traffic Class Feld* reserviert. Die IP-Länge wurde durch eine *Payload* Länge ersetzt und der *Hop Limit* entspricht dem TTL-Feld bei IPv4. Mit dem *Next Header* können nachfolgende Erweiterungsheader angezeigt werden. Beispiele sind *Routing Header* oder *Fragment Header*. Ersatzlos gestrichen wurde die IP Header-Checksumme. Hier setzt man auf die Datenintegritätsmechanismen der untergeordneten Schicht (z.B. Ethernet). Die Headergröße hat sich durch die breiteren Adressen von 20 auf 40 Bytes erhöht.

Für das Forwarding bedeutet IPv6 somit, dass die Überprüfung und Neukalkulation der Header-Checksumme entfällt. Das macht die Prozessierung zunächst einfacher. Andererseits muss ein *Lookup* auf eine 128 Bit-Adresse ausgeführt werden, welcher aufgrund des größeren Adressraums wesentlich aufwändiger ist.

2.3.3 Layer 4: TCP/UDP

Das *Transmission Control Protocol* (TCP) und das *User Datagram Protocol* (UDP) bilden die Sicherungsschicht im OSI Referenzmodell ab. Es sind die beiden am meisten benutzten Protokolle der Ebene 4. Rund 80-90% des Verkehrs wird über TCP abgewickelt, weitere 5-10% basieren auf UDP (siehe [12]).

Transmission Control Protocol (TCP)

TCP [13] ermöglicht einen gesicherten, verbindungsorientierten Datenaustausch zwischen zwei Rechner-Endpunkten im Netz. Die zu übertragenden Daten werden im Quellrechner in einzelne Segmente zerlegt und mit einem nummerierten TCP-Header ergänzt. Anschließend können die TCP-Pakete über das unsichere IP als zusammenhanglose IP-Pakete versendet werden. Die Daten werden empfangsseitig rekonstruiert. Ankommende Pakete werden vom Empfänger quittiert. Vertauschte oder verlorengegangene Pakete können anhand der Sequenznummer (vgl. Abbildung 2.7a) identifiziert werden. Während vertauschte Pakete beim Empfänger sortiert werden können, werden verlorengegangene Pakete erneut versendet. Fehler in der Übertragung können anhand der Checksumme festgestellt werden.

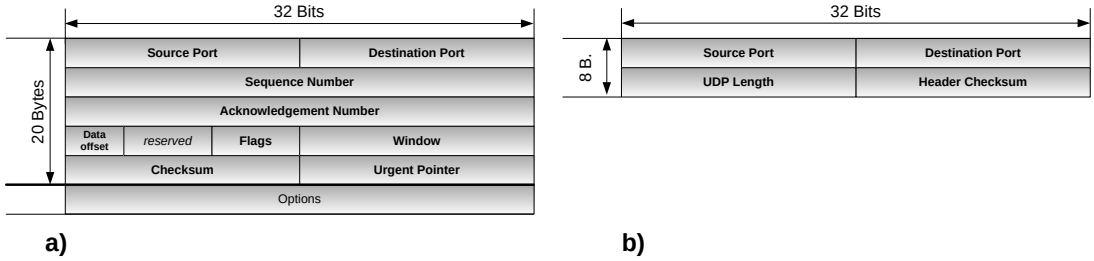

Abbildung 2.7: Header-Aufbau von TCP (a) und UDP (b)

Das Vertauschen der Paketreihenfolge kann negative Auswirkungen auf die Netzwerkleistung haben. Um diese Auswirkung zu verstehen, muss der Quittier-Mechanismus von TCP näher erläutert werden. Jedes empfangene Paket muss bei TCP durch ein *Acknowledgement*-Paket bestätigt werden. Dies erlaubt dem Sender festzustellen, ob alle Pakete beim Empfänger ankamen. Der Sender besitzt ein festgelegtes Sendefenster, d.h. er darf eine bestimmte Anzahl an Paketen absenden, ehe er auf eine Empfangsbestätigung warten muss. Wird der Empfang nicht innerhalb einer bestimmten Zeit bestätigt, wird das Paket senderseitig als verloren eingestuft und erneut versendet. Damit ein verlorengegangenes Paket nicht zu lange die Übertragung blockiert, verwendet TCP das sogenannte *Fast Retransmit* [14]. Für jedes empfangene Paket, das außerhalb der Reihenfolge empfangen wurde, wird die Bestätigung des letzten in Reihenfolge empfangenen Paketes wiederholt (sogenannte *Duplicate Acknowledgements*). Dies zeigt dem Sender, dass zwar das eigentlich nächste Paket nicht beim Empfänger eintraf, die Verbindung aber nach wie vor aktiv ist. Bei drei *Duplicate Acknowledgements* stuft der Sender das Paket als verloren ein und sendet es bereits vor Ablauf des Timers erneut.

Pakete, die drei oder mehr Plätze hinter ihrer originären Reihenfolge eintreffen, werden also fälschlicherweise als verloren eingestuft und erneut gesendet. Gehen zu viele Pakete „verloren“, so wird dies von der *Congestion Control* als eine Überlast des Netzes interpretiert und die Senderate entsprechend gedrosselt.

In [15] wurden die Auswirkung von *Packet Reordering* auf einen Backbone Link un-

tersucht. Ab *Packet Reordering*-Raten von 0,1-1% konnten nennenswerte Einbrüche im Durchsatz festgestellt werden. Der minimale Datendurchsatz wird in etwa ab einer *Packet Reordering*-Rate von 8-10% erreicht. Die Einbrüche hierbei sind beträchtlich. In Abhängigkeit von diversen Variablen wurden Einbrüche beim Applikationsdurchsatz von 50% und mehr gemessen.

Packet Reordering kann dabei zweierlei Gründe haben:

- Paketreihenvertauschung aufgrund unterschiedlicher Wegwahl im Netz.
- Paketreihenvertauschung innerhalb eines Routers.

Während der erste Punkt v.a. durch Lastbalancierung auf unterschiedliche, alternative Links verursacht wird, ist für diese Arbeit v.a. der zweite Aspekt von besonderem Interesse. In [16] konnte mit einem Intel IXP 2400 gezeigt werden, dass bereits in einem Netzwerk mit zehn Hops allein durch Vertauschungen innerhalb des Netzwerkprozessors *Reordering* Raten von bis zu 75% und daraus resultierende *Retransmission* Raten von bis zu 61% auftreten können.

User Datagram Protocol (UDP)

Anders als TCP stellt UDP [17] nur einen verbindungslosen, ungesicherten Dienst zur Verfügung. Der Header (siehe Abbildung 2.7b) besteht nur aus wenigen Elementen. Verloren gegangene oder vertauschte Pakete können nicht erkannt werden. Somit ist auch keine Flusskontrolle wie bei TCP möglich. Ebenso ist eine Resequenzierung bei UDP grundsätzlich nicht möglich. *Packet Reordering* kann damit direkt zu Datenverlust führen. UDP wird häufig für Streaming-Anwendungen wie zum Beispiel VoIP verwendet. Für eine hohe Sprachqualität sind neben geringen *Packet Reordering*-Raten auch geringe Übertragungslatenzen wichtig.

2.3.4 Sicherheitsprotokolle

Sicherheitsrelevante Anwendungen gewinnen mehr und mehr an Bedeutung. TCP/IP selbst bietet jedoch keinerlei Schutzmechanismen. Damit ist die gesamte Kommunikation zwischen zwei Teilnehmern relativ leicht mitlesbar bzw. manipulierbar. Gerade aber sensible Daten benötigen einen entsprechenden Schutz. Als Anwendungsbeispiele sei hier nur auf die Vernetzung zweier Firmenstandorte verwiesen oder auch auf den externen Zugriff von Mitarbeitern auf Firmen-Server. Eine mögliche Lösung wäre dabei die Verwendung physikalisch separierter Standleitungen. Dieser Ansatz ist jedoch teuer und zudem unflexibel. Ein *Virtual Private Network* (VPN), also ein gesichertes virtuelles Netz, das physikalisch über das Internet kommuniziert ist dagegen eine preiswerte und flexible Lösung. Letztlich kann der Zugriff auf das VPN bei vorhandenem Internetanschluss weltweit erfolgen.

Ein Verschlüsselungsmechanismus stellt dabei insbesondere die folgenden Dienste bereit:

- **Vertraulichkeit**: Die übermittelten Daten können unterwegs nicht von anderen

Personen gelesen und ausgewertet werden.

- **Datenintegrität**: Die Daten können unterwegs nicht unbemerkt verändert werden.
- **Authentizität**: Die Daten können eindeutig einem Absender zugeordnet werden.

Grundsätzlich lassen sich VPNs auf Layer 2 (insbesondere L2TP) oder Layer 3 (IPsec) realisieren. L2TP ermöglicht selbst keine Sicherheitsmechanismen, kann aber mit IPsec entsprechend kombiniert werden.

IPsec

IPsec ist in der RFC4301 [18] definiert und besteht aus den zwei Protokollen *Authentication Header* (AH, RFC4302 [19]) und *Encapsualting Security Payload* (ESP, RFC4303 [20]).

AH sichert dabei lediglich die Datenintegrität und Authentizität, bietet jedoch keinerlei Verschlüsselung. Bei ESP dagegen werden die Daten verschlüsselt übertragen. Beide Protokolle können entweder im *Transport Modus* oder *Tunnel Modus* betrieben werden. Im *Transport Modus* wird der ursprüngliche IP-Header weiterverwendet. Dahinter wird der entsprechende AH- oder ESP-Header eingefügt, gefolgt vom restlichen (bei ESP verschlüsselten) Paket. Damit sind die ursprünglichen IP-Adressen nach wie vor im Klartext lesbar. Dieser Modus ist bei Ende-zu-Ende-Verbindungen relevant, die im Folgenden nicht weiter betrachtet werden. Die Verbindung zweier Subnetze im Router erfolgt über den *Tunnel Modus*. Dabei wird ein zusätzlicher IP-Header, der nunmehr lediglich Start- und Endpunkt des Tunnels festlegt (also jeweils der Router), eingefügt. Der alte IP-Header befindet sich mit dem restlichen Paket im verschlüsselten Bereich. Die ursprünglichen IP-Adressen sind also ohne Entschlüsselung nicht mehr nachvollziehbar.

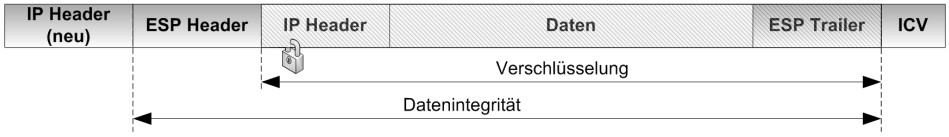

Abbildung 2.8: Header Struktur bei einem IP-Paket im ESP-Tunnel Modus

Der Aufbau des ESP-Paketes ist in Abbildung 2.8 veranschaulicht. Das Paket wurde, durch einen ESP-Trailer ergänzt, verschlüsselt übertragen. Der Trailer enthält im Wesentlichen ein *Padding*, um die Größe des zu verschlüsselnden Bereichs auf ein Vielfaches der Algorithmus-Blocklänge zu bringen. Der ESP-Header enthält neben einem *Security Parameter Index* (SPI) - einem Index auf den Sicherheitseintrag der Tunnel-Verbindung - eine fortlaufende Sequenznummer. Eine angehängte Prüfsumme (*Integrated Check Value*, ICV) sichert die Datenintegrität.

IPsec-Datenbanken

IPsec verwaltet im Wesentlichen zwei verschiedene Datenbanken: die *Security Policy Database* (SPD) und die *Security Association Database* (SAD).

Die SPD enthält für eine Verbindung die durchzuführende Aktion auf ein ankommendes Paket. Dabei gibt es drei mögliche Aktionen: Das Paket soll verworfen werden (*Discard*), das Paket wird ohne IPsec weiterverarbeitet (*Bypass*), oder eine Verschlüsselung ist notwendig (*Protect*). In letzterem Fall enthält die Datenbank zusätzlich den bereits erwähnten SPI, der letztlich auf einen Eintrag in der SAD verweist.

Die SAD enthält für jede zu verschlüsselnde Verbindung alle notwendigen Informationen. Darin enthalten sind u.a. [21]:

- Der zu verwendende Modus (AH/ESP, Tunnel/Transport).
- Die aktuelle Sequenznummer.
- Der zu verwendende Algorithmus für Verschlüsselung und Datenintegrität.
- Notwendige Schlüssel und Initialisierungsvektor (falls benötigt).
- Ziel- und Quell-IP-Adresse des Tunnels.

Zur Verschlüsselung und für die Datenintegrität stehen verschiedene Algorithmen zur Verfügung, die sich in Aufwand und Sicherheit unterscheiden können. Möglich sind beispielsweise AES, DES und Triple-DES zur Verschlüsselung bzw. HMAC-SHA1-96 oder AES-XCBC-MAC-96 für die Datenintegrität.

Die Einträge in der SAD können auf zwei verschiedene Arten verwaltet werden. Zum einen durch *Manual Keying*, d.h. Einträge werden per Hand eingetragen. Hierzu muss beiden Endpunkten ein (statischer) Schlüssel zuvor bekannt sein. Beim *Internet Key Exchange Protocol (IKE)* [22] dagegen werden die Schlüssel beim Verbindungsaufbau eines neuen Tunnels auf sichere Weise festgelegt und ausgetauscht.

Im Folgenden werden nochmals die einzelnen Schritte zusammengefasst, die für die Prozessierung eines *Outbound*-, respektive eines *Inbound*-Pakets erforderlich sind. Als *Outbound*-Verkehr werden dabei die Pakete bezeichnet, die nach der SPD zu verschlüsseln sind. *Inbound*-Verkehr besteht aus verschlüsselten Paketen, die entschlüsselt werden müssen. Es wird lediglich der Datenpfad betrachtet, d.h. es wird von einer bereits bestehenden Verbindung mit ausgetauschten Schlüsseln ausgegangen. Exemplarisch wird ESP-Verkehr behandelt.

IPsec ESP Prozessierung Outbound Traffic

Folgende Schritte müssen durchgeführt werden:

1. Überprüfen der Integrität des ankommenden IP-Pakets (TTL, IP-Header-Checksumme)
2. Abfrage der SPD anhand der Paketdaten.
3. *Die folgenden Schritte beziehen sich auf den Fall, dass diese Abfrage eine Ver-*

schlüsselung ergibt:

a) Abfrage aller relevanten Informationen in der SAD anhand der SPI (aus SPD).
b) Erstellen und Einfügen des ESP-Headers und ESP-Trailers.
c) Erstellen und Einfügen eines Tunnel-IP-Headers.
d) Verschlüsselung.
e) Erstellen und Einfügen der ICV.

4. IP-Forwarding.

IPsec ESP Prozessierung Inbound Traffic

Folgende Schritte müssen durchgeführt werden:

1. Überprüfen der Integrität des ankommenden IP-Pakets (TTL, IP Header-Checksumme).
2. *Folgende Schritte werden durchgeführt, sofern ein zu entschlüsselndes Paket anhand Ziel-IP-Adresse und ESP-Header festgestellt wird:*
   a) Abfrage aller relevanten Informationen in der SAD anhand der SPI (aus ESP-Header)
   b) Überprüfen und Löschen der ICV.
   c) Entschlüsseln.
   d) Löschen von Tunnel IP-Header, ESP-Header und ESP-Trailer.
3. IP-Forwarding.

Damit muss bei IPsec aktiv ein Zustand verwaltet und abgefragt werden. Pakete können somit nicht unabhängig voneinander prozessiert werden. IPsec ist daher ein Beispiel für einen **zustandbehafteten Verkehr**.

2.3.5 Quality of Service

Als *Quality of Service* (QoS) bezeichnet man im Allgemeinen die Dienstgüte eines Netzes beim Übertragen von Paketen. QoS definiert sich dabei über die folgenden vier Messgrößen:

- **Latenz:** Zeit, die benötigt wird, um ein Paket vom Sender zum Empfänger zu übertragen.
- **Jitter:** Abweichung der Übertragungszeit (Latenz) vom Mittelwert eines Flows.
- **Paketverlustrate:** Der Anteil an Paketen, der bei der Übertragung verloren geht.
- **Durchsatz:** Die Datenrate, die im Mittel übertragen werden kann.

Je nach Anwendung können unterschiedliche Anforderungen relevant sein. Bei reinen Filetransfers ist beispielsweise vor allem der Durchsatz von Bedeutung, die restlichen Größen spielen eine untergeordnete Rolle. Bei VoIP dagegen spielen Latenz, Jitter und

Verlustrate eine maßgebliche Rolle, um eine hohe Sprachqualität zu erreichen. Standardmäßig werden IP-Pakete jedoch als *Best Effort*-Verkehr behandelt, d.h. es gibt keinerlei Garantien bezüglich QoS.

Für die Priorisierung von IP-Paketen kommen vor allem zwei Verfahren zur Anwendung: die *Integrated Services (IntServ)* [23] und die *Differentiated Services (DiffServ)* [24].

Bei *IntServ* wird vor der Übertragung von Datenpaketen eine virtuelle Verbindung aufgebaut. Mithilfe des *Resource Reservation Protocols (RSVP)* werden bei jedem beteiligten Router die für die Verbindung notwendigen Ressourcen reserviert. Somit muss jeder Router *IntServ*-kompatibel sein, die aktiven Verbindungen verwalten und die Pakete entsprechenden klassifizieren. Bei *DiffServ* werden die Pakete bereits am Rand des Netzwerks durch einen speziellen Router klassifiziert. Innerhalb des Netzwerks wird das Paket lediglich entsprechend seiner Klasse weitergeleitet. Die Klassifizierung wird dabei im *Differentiated Service Codepoint* (DSCP) festgehalten. Dieser entspricht bei IPv4 dem *Type of Service*-Feld, bei IPv6 dem *Traffic Class*-Feld [25]. Die Router innerhalb des Netzwerks müssen der Einstufung vertrauen. Beim Übergang zwischen zwei Netzwerken (z.B. zwischen zwei Netzbetreibern) kann eine erneute Klassifikation erfolgen.

2.3.6 Deep Packet Inspection

Unter *Deep Packet Inspection* (DPI) versteht man die Prozessierung von Paketen auf den Layern 4 und höher. Dabei werden nicht nur die Header einzelner Pakete betrachtet, sondern ebenso dessen Paketdaten. In der Regel werden die Pakete dabei nicht einzeln verarbeitet, da diese ja nur ein kleines Fragment der zu übertragenden Information beinhalten. Vielmehr ist eine Auswertung des ganzen Flows notwendig und damit verbunden eine aufwändige Verwaltung eines aktuellen Status. DPI ist damit mit entsprechend hohen Anforderungen an die Rechenleistung verbunden.

Mögliche Anwendungen für DPI sind z.B. [26]:

- E-Mail Spam Filter
- Anti-Virus Filter
- Intrusion Detection
- Firewalls
- Network Monitoring

DPI ist dabei nicht unumstritten, da sich die verwendeten Techniken ebenso gut zur Überwachung der übermittelten Inhalte und damit auch Zensur einsetzen lassen.

2.3.7 Routingprotokolle

Zur Erstellung von Routingtabellen kommen, wie im Abschnitt 2.3.2 bereits erwähnt, unterschiedliche Routingprotokolle zum Einsatz. Dabei unterscheidet man grob zwischen *Interior Gateway Protocols* (IGP) und *Exterior Gateway Protocols* (EGP). IGP kommen

dabei innerhalb sog. autonomer Systeme zum Einsatz und EGPs zum Austausch von Routinginformation zwischen autonomen Systemen. Ein autonomes System ist dabei eine Ansammlung von IP-Netzen, die gemeinsam verwaltet werden (z.B. ein *Internet Service Provider*).

Bekannte Vertreter der IGPs sind *Open Shortest Path First* (OSPF) und das *Routing Information Protocol* (RIP). Bei den EGPs kommt v.a. das *Border Gateway Protocol* zum Einsatz (siehe auch [5]).

Ohne auf die Details der Routingalgorithmen, die nicht Gegenstand dieser Arbeit sind, näher einzugehen, lässt sich folgendes festhalten: Das Erlernen der Routingtabellen in einem Router ist eine vergleichsweise aufwändige und komplexe Verarbeitung. Während des Betriebs müssen die Teilnehmer laufend über den aktuellen Stand des Netzwerks informiert werden. Dies ist v.a. wichtig beim Ausfall einzelner Komponenten. Dazu werden die Routingtabellen laufend aktualisiert. Dennoch ist die Paketrate der dazu notwendigen Kontrollpakete verglichen mit der Nutzdatenpaketrate vergleichsweise gering. Es handelt sich hierbei um keine zeitkritische Verarbeitung.

2.4 Netztopologie

Um die Einsatzvielfalt von Netzwerkprozessoren in Netzwerk-Routern aufzuzeigen, wird in diesem Abschnitt ein Blick auf eine typische Netzwerktopologie geworfen. Innerhalb eines typischerweise hierarchisch aufgebauten Netzwerks, gibt es eine Reihe von Routern an unterschiedlichen Stellen. Je nach Funktion im Netzwerk, spricht man von verschiedenen Routertypen. Dies ist nicht unbedingt verbunden mit einer anderen Hardware-Architektur, aber mit unterschiedlichen Aufgaben und damit eventuell anderen Anforderungen an zum Beispiel Durchsatz oder Protokollunterstützung. Die Klassifizierung der verschiedenen Routertypen ist dabei nicht immer ganz eindeutig, da ein Router auch unterschiedliche Aufgaben gleichzeitig wahrnehmen kann.

Wie in Abbildung 2.9 ersichtlich lässt sich ein Netz typischerweise aufteilen in lokale Netzwerke (*Local Area Network*, LAN), Zugangsnetze (*Access Network*, AN) und Weitverkehrsnetze (*Wide Area Network*, WAN). Ein LAN erstreckt sich dabei in der Regel über eine kurze Distanz, wie z.B. innerhalb eines Gebäudes oder einer Wohnung. Ein WAN ist dagegen ein Netz über große Distanzen, häufig über Länder oder Kontinente hinweg und wird z.B. von Internetanbietern oder auch großen Organisationen genutzt. Das Zugangsnetz schließlich befindet sich in der Regel in der Vermittlungsstelle und ist die Schnittstelle zwischen LAN und WAN.

2.4.1 Local Area Networks

Switched LANs

Ein LAN besteht aus mehreren lokalen Netzwerkteilnehmern die über ein gemeinsames Netz miteinander verbunden sind. Im LAN-Bereich dominiert dabei v.a. Ethernet. In der

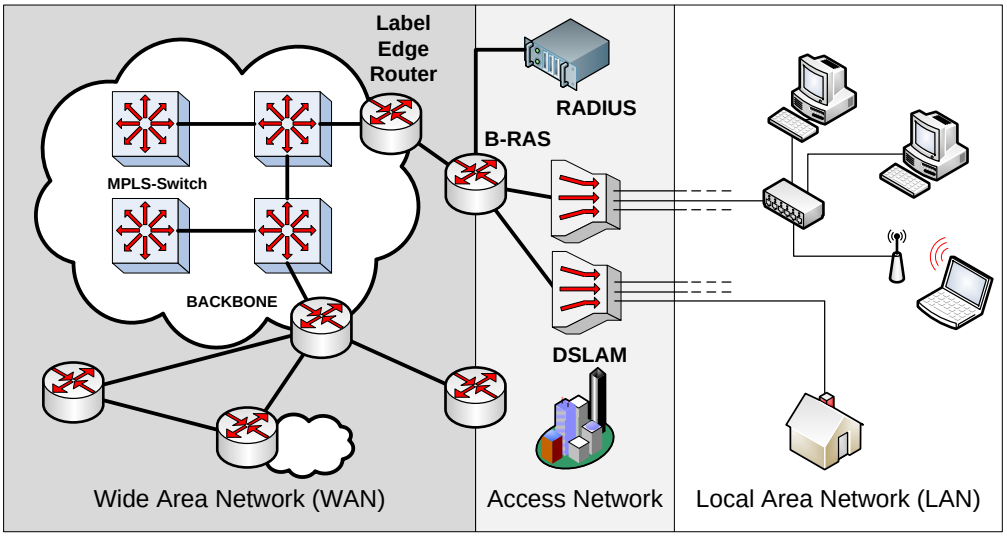

Abbildung 2.9: Exemplarische Netzwerk-Infrastruktur

Regel werden dafür sogenannte *Switched LANs* verwendet. Hierbei ist jeder Teilnehmer separat an einen Switch angeschlossen, womit eine kollisionsfreie Übertragung möglich ist. Das LAN kann über einen gemeinsamen Router mit der Außenwelt verbunden sein. Im sogenannten SOHO-Bereich (*Small Office - Home Office*), also bei privaten Nutzern bzw. Kleinbüros, kommen hier häufig sogenannte *Residential Gateways* zum Einsatz. Das sind in der Regel verhältnismäßig kleine Geräte mit integriertem Modem (beispielsweise DSL- oder Kabelmodem) und eingebauten Switch zum direkten Anschluss mehrerer Teilnehmer. Die Geräte beinhalten typischerweise eine *Network Address Translation*, d.h. im Router werden die lokalen IP-Adressen gegen eine einzige gemeinsame IP-Adresse ausgetauscht, mit der Daten dann ins Internet versendet und empfangen werden. Auch stellen die Geräte teilweise Funktionen wie eine Priorisierung des Verkehrs oder *Virtual Private Network*-Unterstützung bereit. Es handelt sich hier um Geräte mit im Allgemeinen geringen Anforderungen an Durchsatz und Leistung und damit auch niedrigen Anschaffungskosten.

Komplexe LAN Strukturen

Prinzipiell sind im LAN-Bereich auch komplexere Strukturen und der Einsatz von Routern denkbar. Dies ist besonders dann sinnvoll, wenn aus Gründen der Sicherheit oder des Datendurchsatzes das LAN besser strukturiert sein soll. Denkbar sind z.B. separate Subnetze für einzelne Abteilungen einer Firma. Diese Subnetze werden untereinander mit Routern verbunden. Sinnvoll ist dies v.a. dann wenn der Hauptteil des Datenverkehrs innerhalb einer Abteilung stattfindet und nur ein kleinerer Teil zwischen den Subnetzen. Zudem können im Router Filterfunktionen oder Zugangskontrollen implementiert werden.

2.4.2 Zugangsnetz

Das Zugangsnetz ist die Schnittstelle zwischen LAN und WAN. Hierbei werden in der Regel viele LANs mit einem übergeordneten WAN verbunden. Bei DSL beispielsweise werden die separaten Anschlüsse der Teilnehmer im *Digital Subscriber Line Access Multiplexer* (DSLAM) zusammengeführt und mit dem WAN des *Internet Service Provider* (ISP) verbunden. Der DSLAM stellt also das Gegenstück des DSL-Modems beim einzelnen Teilnehmer dar. Die herkömmliche Kommunikation sowohl zwischen DSLAM und Teilnehmer als auch zwischen DSLAM und Backbone ist ATM-basiert. In diesem Szenario dient der DSLAM lediglich als Layer-2 ATM-Switch, der die Daten an einen *Broadband Remote Access Server* (BRAS) des ISP weiterleitet [27]. Die Benutzerdaten werden dabei mittels eines RADIUS (*Remote Authentication Dial-In User Service*) Servers verifiziert.

Access Router

Access Router befinden sich an der Verbindung zwischen Zugangsnetz und Weitverkehrsnetz und sind damit die Schnittstelle zum Teilnehmernetz (siehe [28]). Im Abschnitt zuvor wurde bereits der BRAS eingeführt. Ganz allgemein kann man definieren, dass ein *Access Router* private oder Unternehmensnetze mit einem größeren Netzwerk oder dem Internet verbindet. Unterstützt werden neben der Authentifizierung insbesondere Funktionen für Sicherheit, Verschlüsselung, *Accounting* oder *Tunneling*. Neben VPN-Anwendungen kann auch die Integration einer Firewall von Bedeutung sein.

IP-DSLAMs

ATM besitzt eine im Allgemeinen schlechte Skalierbarkeit und wird deshalb auch im Zugangsbereich immer häufiger durch Ethernet/IP-basierte Systeme ersetzt. Die Kommunikation zwischen Teilnehmer und DSLAM bleibt jedoch meist ATM-basiert. Die Umsetzung auf Ethernet erfolgt dabei innerhalb des DSLAM (IP-DSLAM) [29]. Dies erfordert eine Reihe von sogenannten *Interworking Functions* (IWF) zwischen ATM und Ethernet, die je nach eingesetztem Protokoll benötigt werden. Hinzu kommt, dass immer mehr Funktionalität in die DSLAMs verlagert wird. Hierzu gehören zum einen einzelne Funktionen die ursprünglich im BRAS angesiedelt waren bis hin zur vollwertigen BRAS-Funktionalität innerhalb des DSLAMs. Aber auch QoS, Filterung oder die Klassifikation von Paketen werden innerhalb des DSLAMs angestrebt. Zukünftige DSLAMs sollen sowohl IP-Routing beherrschen, als auch schon als Zugang zu einem *Multiprotocol Label Switching* Netz dienen und Funktionen eines *Label Edge* oder *Label Switch Routers* übernehmen [30] (siehe auch 2.4.3).

Triple Play bei DSLAMs

Ein weiterer zunehmend wichtiger Aspekt bei DSLAMs ist die Unterstützung von *Triple Play*. Darunter versteht man die Benutzung des Netzes für Datenanwendungen, Inter-

nettelefonie (VoIP) und für Video-/Fernsehverkehr (IPTV, *Video on Demand*). Da die Datenvolumina insbesondere der Multimediaanwendungen rasant zunehmen, versucht man auch hier neue Wege zu gehen [31]. Anstatt den gesamten Verkehr über den BRAS des ISP zu senden, kann der Verkehr bereits im DSLAM aufgeschlüsselt werden (siehe Abbildung 2.10). Das hat zweierlei Vorteile: zum einen kann das Backbone-Netz des Anbieters entlastet werden, da ein großer Teil des Verkehrs direkt im Zugangsnetz zum richtigen Server geleitet wird. Zum anderen kann dadurch auch eine höhere Dienstgüte durch geringere Latenzen, geringere Ausfallwahrscheinlichkeiten etc. erreicht werden. Insbesondere erfolgt bereits eine Priorisierung, z.B. von VoIP-Paketen im DSLAM.

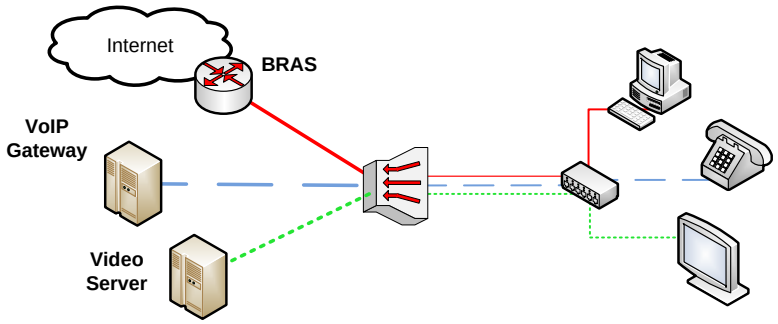

Abbildung 2.10: Aufsplittung des *Triple Play* Verkehrs im DSLAM.

Aber auch weitere Anwendungen können in DSLAMs integriert werden. So kann der Aufbau einer gesicherten VPN-Verbindung (z.B. zwischen zwei Firmenstandorten) innerhalb der DSLAMs beim *Service Provider* realisiert werden [32]. Dies spart unter Umständen eigene Rechenkapazitäten ein, die für die Verschlüsselung vorgehalten werden müssten.

2.4.3 Wide Area Networks

Ein Weitverkehrsnetz eines ISP lässt sich meist hierarchisch strukturieren. Dementsprechend lassen sich verschiedende Typen von Routern definieren, die je nach Hierarchiestufe zum Einsatz kommen.

Edge Router befinden sich am Übergang zwischen zwei Netzen. Dies kann zum Beispiel der Übergang zu einem Hochgeschwindigkeitsnetz (*Backbone*, siehe nächsten Abschnitt *Core Router*) sein oder der Übergang zum Netz eines anderen ISP. Beim Übergang zum Hochgeschwindigkeitsnetz kommen zum Beispiel sogenannte *Label Edge Router* zum Einsatz. Die Kommunikation zu anderen ISP basiert dagegen weitestgehend auf dem *Border Gateway Protocol* (siehe Abschnitt 2.3.7).

Router innerhalb Hochgeschwindigkeitsnetze (manchmal auch als *Core*-Netzwerk oder *Backbone* bezeichnet) werden als *Core Router* bezeichnet. Als Übertragungstechnik innerhalb des *Backbones* kommt verstärkt das *Multiprotocol Label Switching* (MPLS) [33] zum Einsatz [5]. Pakete werden hierbei über virtuelle Verbindungen versendet. Das hat den Vorteil, dass im Gegensatz zum verbindunglosen Routing bereits zuvor der Weg des

Paketes durch das Netz festgelegt werden kann und damit innerhalb des bekannten, eigenen Netzes eine effektive Lastbalancierung betrieben werden kann. Die Weiterleitung basiert auf einem Label, das jedem Paket vorangestellt wird. Das Label stellt somit eine Identifikation für das Paket bzw. die Route dar. Die Verarbeitung beschränkt sich auf die Analyse und das Austauschen des Labels, die aufwändige Analyse der IP-Header entfällt.

Bei MPLS kommen zwei Arten von Routern zum Einsatz: der *Label Edge Router* (LER) und der *Label Switch Router* (LSR). Der LER sitzt am Rand eines MPLS-Netzes und besitzt neben den klassischen Routerfunktionen zusätzlich die Möglichkeit der Paketklassifizierung. Die Pakete werden hier mit einem zusätzlichen Label versehen. Die LSR haben die Aufgabe, das Paket anhand des Labels weiterzuleiten und dabei die Labels ggf. zu aktualisieren. Hier handelt es sich also um kein klassisches Routing (die Route wurde ja zuvor schon durch den LER festgelegt), sondern eher um ein Layer 2-Switching. Aufgrund dieser Ansiedlung zwischen Layer 2 und Layer 3 spricht man beim MPLS auch oft von einem Layer 2,5-Protokoll.

2.4.4 Anforderungen an paketverarbeitenden Instanzen im Netz

Der Überblick über die Netztopologie verdeutlicht bereits die Vielfalt an Aufgaben, die in einem Netz erledigt werden müssen. Je nachdem, an welcher Stelle im Netz ein Router eingesetzt wird, stehen unterschiedliche Anforderungen im Vordergrund. Im *Backbone*-Bereich ist eine möglichst schnelle Paketverarbeitung bei hohen bis sehr hohen Datenraten notwendig. Allerdings ist der Umfang der Paketverarbeitung und die Anzahl der zu unterstützenden Schichten und Protokolle sehr gering. Im *Access*-Bereich dagegen ist eine Vielzahl an Anwendungen denkbar. Von Forwarding über Zugangskontrolle, *Filtering* und QoS, bis hin zu *Deep Packet Inspection* kann hier praktisch alles gefordert sein. Zudem sind auch hier immer noch beträchtlichten Datenraten notwendig, wenngleich deutlich niedriger als im *Backbone*-Bereich.

Genau diese Vielfalt stellt die große Herausforderung an Router-Architekturen dar. Sie müssen - auch in Abhängigkeit vom konkreten Einsatzzweck - mitunter eine **hohe Rechenleistung** bei **hoher Protokollkomplexität** erreichen. Zudem muss eine ausreichende **Flexibilität** gewährleistet sein, um einen möglichst weiten Einsatzbereich abzudecken. Schließlich besitzen zu spezialisierte Lösungen nur limitierte Einsatzmöglichkeiten und sind damit wegen geringer Stückzahlen unwirtschaftlich.

Router können dabei als Software- oder Hardware-Router realisiert sein. Software-Router sind häufig aus Standard-PC Komponenten aufgebaut und decken die Funktionalität komplett in Software ab. Sie besitzen jedoch eine im Allgemeinen niedrige Rechenleistung und sind daher nicht für den *Backbone*- oder *Access*-Bereich geeignet. Hardware-Router sind dagegen speziell für den Routing-Einsatz optimiert. Zur Verbindung der einzelnen Ports verfügen sie i.A. über entsprechende Hochleistungs-Interconnects bzw. *Cross-Bars*. Als Verarbeitungseinheit kommen häufig spezialisierte Prozessoren – die **Netzwerkprozessoren** – zum Einsatz, die im folgenden Kapitel näher beschrieben werden.

3 Stand der Technik

Im folgenden Kapitel wird der für diese Arbeit relevante Stand der Technik näher beschrieben. Dazu wird zunächst ein Überblick über Netzwerkprozessoren im kommerziellen und akademischen Bereich gegeben. Anschließend werden bekannte Lastbalancierungsverfahren im Netzwerkprozessor-Bereich gezeigt, bevor verschiedene Paket-Resequenzierungsverfahren vorgestellt werden. Abschließend wird ein *Multiprozessor Interrupt Controller* vorgestellt.

3.1 Netzwerkprozessoren

Netzwerkprozessoren (NP) werden seit den 1990er Jahren entwickelt, um die beiden im Allgemeinen gegensätzlichen Hauptanforderungen der Paketverarbeitung – nämlich Geschwindigkeit und Flexibilität – bestmöglich zu kombinieren. In den Ursprüngen der Paketverarbeitung war die Verwendung von *General Purpose Prozessoren* (GPP) üblich. Durch die Abbildung der Funktionalität auf Software konnte eine hohe Flexibilität erreicht werden. Allerdings stießen die GPPs mit steigenden Datenraten und steigender Verarbeitungskomplexität sehr schnell an ihre Grenzen. Durch den sehr allgemeinen Instruktionssatz ist die Rechenleistungsdichte der GPPs sehr gering (vgl. auch Abbildung 3.1). Abhilfe schaffte der Einsatz spezieller, dedizierter Hardware, also einer direkten Implementierung der Paketverarbeitung in Hardware. Diese *Application Specific Integrated Circuits* (ASICs) besitzen zwar eine sehr hohe Rechenleistungsdichte, allerdings können Anpassungen an den unterstützten Protokollen oder dem jeweiligen Anwendungsbereich nur im sehr beschränktem Umfang vorgenommen werden, da die jeweilige Funktionalität direkt in Hardware implementiert wird.

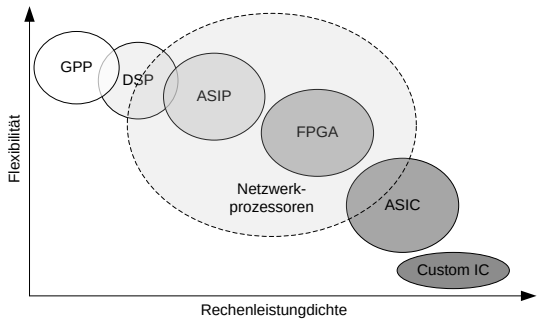

Abbildung 3.1: *Design Space* bei NP-Implementierungen

Die Definition von NPs ist nicht ganz eindeutig. Ganz allgemein handelt es sich hierbei um Verarbeitungsinstanzen mit angepassten Eigenschaften für den *Networking*-Bereich. NPs sind typischerweise in Software programmierbar, besitzen allerdings eine für *Networking* optimierte Hardwareunterstützung. Das beginnt häufig mit einem spezifisch optimierten Instruktionssatz, umfasst aber auch gesonderte Hardwarebeschleuniger für spezielle Aufgaben (beispielsweise Verschlüsselung, *Pattern Matching* etc.; vgl. hierzu auch [34]).

An dem folgenden Überblick über verschieden NP-Architekturen wird deutlich, wie unterschiedlich die gewählten Ansätze sein können. Häufig hängt die Architektur vom konkreten Anwendungsfall des einzelnen NPs ab. Je nachdem, ob ein NP z.B. eher im *Access*- oder im *Backbone*-Bereich eingesetzt werden soll, wird mehr Wert auf Flexibilität oder Leistung gelegt.

In der Regel besitzen die meisten NPs zur Steigerung des Durchsatzes eine Vielzahl an Verarbeitungseinheiten. Man unterscheidet im Allgemeinen zwischen einem parallelen Cluster und einer Verarbeitungs-Pipeline (vgl. Abbildung 3.2).

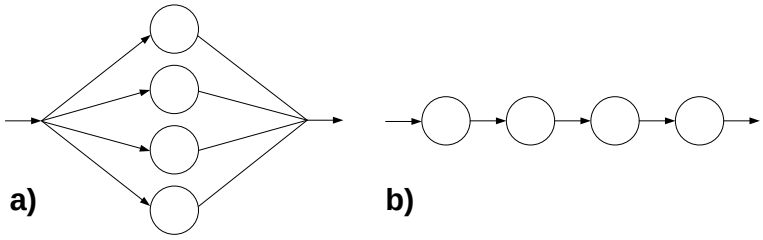

Abbildung 3.2: Paketverarbeitung in einem Cluster (a) und in einer Pipeline (b).

Im **Cluster** finden sich viele, meist identische Verarbeitungsinstanzen, die parallel Pakete verarbeiten können. Die einzelnen Verarbeitungsinstanzen sind dabei grundsätzlich gleichberechtigt und können prinzipiell jedes Paket komplett bearbeiten. Da keine Übergabe an eine andere Verarbeitungsinstanz notwendig ist, spricht man hier auch von *run-to-completion*. Cluster sind prinzipiell sehr einfach skalierbar und programmierbar. Grundsätzlich kann der Programmcode ohne Berücksichtigung der Parallelität für eine Instanz geschrieben und optimiert werden. Er wird anschließend lediglich für alle Instanzen vervielfältigt. Probleme ergeben sich beim Cluster insbesondere bei der Verteilung der Pakete auf die Verarbeitungsinstanzen und beim Paketspeicher. Wie in Kapitel 3.2 noch erläutert wird, ist die Lastbalancierung keineswegs ein triviales Problem. Cluster besitzen in der Regel einen zentralen Paketspeicher. Die einzelnen Verarbeitungsinstanzen laden die zu verarbeitenden Pakete aus diesem Speicher und speichern sie nach erfolgreicher Bearbeitung wieder dort ab. Damit wird der Speicher und dessen Anbindung ans System sehr schnell zu einer potentiellen Engstelle im System.

Bei einer **Pipeline** dagegen wird die Verarbeitung in mehrere Teilschritte aufgeteilt. Jede Verarbeitungseinheit führt nur einen Teilschritt aus und übergibt das jeweilige Paket danach der nächsten Verarbeitungseinheit. Jede Instanz erhält einen lokalen Paketspeicher, somit wandert das Paket von Instanz zu Instanz. Ein zentraler Speicher

und die damit verbundenen Probleme entfallen. Nachdem sich die Geschwindigkeit der Pipeline an der langsamsten Stufe orientiert, muss die Verabeitung in möglichst gleich große Teilschritte eingeteilt werden. Diese anspruchsvolle Partitionierung ist bei der Programmierung zu berücksichtigen und wird in aller Regel vom Entwickler vorgenommen. Die Problematik verschärft sich bei einer heterogenen Verarbeitung, also einer Umgebung mit unterschiedlichen Verarbeitungsanforderungen der einzelnen Pakete. Damit einher gehen unterschiedliche Verarbeitungslatenzen, die eine effiziente Partitionierung schwierig bis unmöglich machen. Pipelines eignen sich daher vor allem bei homogener Verarbeitung, wie sie z.B. im *Backbone*-Bereich zu finden ist.

Grundsätzlich sind auch Mischformen zwischen Cluster und Pipeline möglich und teilweise zu finden.

Bei NPs wird in der Regel zwischen einer *Data* und einer *Control Plane* (oft auch *Fast / Slow Path* genannt) unterschieden. Die *Data Plane* bearbeitet hierbei den Großteil der Pakete. Sie ist für die Verarbeitung und Weiterleitung der Datenpakete zuständig und hat daher hohe Anforderungen an Durchsatz und Verarbeitungslatenz. Die *Control Plane* bearbeitet insbesondere Kontrollpakete (z.B. Routingprotokolle) oder Ausnahmen in der Paketverarbeitung (z.B. fehlerhafte Pakete, erfolglose Weiterleitung etc.). Sie hat daher geringere Anforderungen an Durchsatz und Latenz bei hoher Protokoll-Komplexität. Die *Control Plane* wird aus diesem Grunde häufig durch einen GPP realisiert.

3.1.1 Überblick über kommerzielle Netzwerkprozessoren

Die Liste kommerziell verfügbarer NPs ist lang. Dabei werden – wie schon erläutert – konzeptionell unterschiedliche Ansätze verfolgt. Die folgende Auflistung soll einen Überblick über wichtige, kommerziell verfügbarer NPs geben. Hierbei wird auch die Vielfalt der gewählten Ansätze verdeutlicht. Die Liste ist dabei nicht als abschließend zu verstehen.

Xelerated Synchronous Dataflow Architecture

Die synchrone Datenfluss-Architektur von Xelerated [35] besteht aus einer Pipeline von sogenannten *Packet Instruction Set Computers* (PISC) mit regelmäßigen Unterbrechungen durch *Engine Access Points* (EAP) (siehe Abbildung 3.3). Jeder PISC-Prozessor kann innerhalb eines Takts vier unterschiedliche Operationen (8-/16-Bit *ALU*, *Copy*, *Branch* und *Load Offset*) ausführen. Mit jedem Takt wandern die Paketdaten in den Paketspeicher des jeweils nächsten PISC-Prozessors. Parallel hierzu werden mit dem Paket sogenannte *General-* und *Special-Purpose*-Registerinhalte übergeben. Das *Special-Purpose*-Register enthält Kontroll- und Statusinformationen, wie Paketlänge, Empfangsport, ALU-Flags oder Fehlerstatus. Ferner werden Kontrollflags zum Auslösen einer Operation und ein Zeiger auf die jeweils auszuführende Instruktion übergeben. Die Instruktionen selbst werden in einem lokalen Speicher des Prozessors gehalten. Die EAPs dagegen sind I/O Prozessoren, die die Kommunikation mit diversen On-Chip Beschleunigern, wie *Hash Engine*, *Ternary Content-Addressable Memories* (TCAM) oder externen

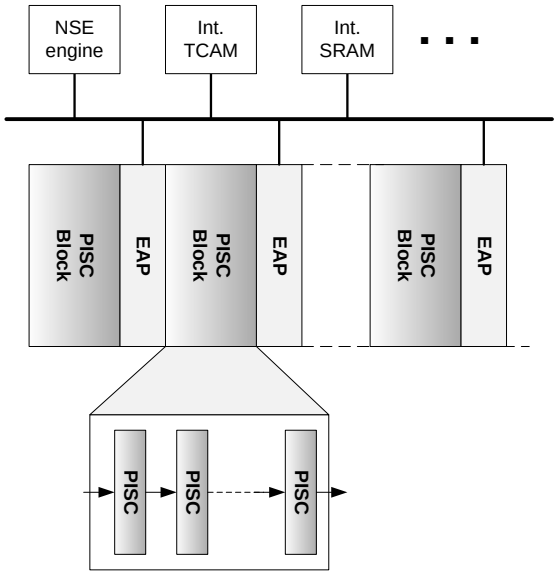

Abbildung 3.3: Xelerated Synchrone Datenfluss Architektur

Beschleunigern ermöglichen. Damit unterscheidet sich der Datenfluss-Ansatz grundlegend von der von-Neumann-Architektur. Bei einem von-Neumann-Prozessor bestimmt der Instruktionsfluss die zu ladenden und speichernden Daten. Dies erweist sich bei bestimmten Anwendungen (insbesondere bei solchen mit hohen Datenvolumina) allerdings als Nachteil, da das Laden und Speichern von Daten die Leistung des Systems limitiert. Bei der Paketverarbeitung stellen die Pakete einen quasi-kontinuierlichen Datenfluss dar, was durch die Datenfluss-Architektur ausgenutzt wird. Hier bestimmen die Paketdaten letztlich die auszuführenden Instruktionen. Dadurch, dass die Daten die Pipeline passieren, ist ein gemeinsamer Speicher nicht mehr nötig.

Die synchrone Datenfluss-Architektur wird bei unterschiedlichen Prozessoren der Firma Xelerated angewandt. Der X10q [36] besteht beispielsweise aus einem linearen Array von 200 PISC Prozessoren mit insgesamt elf EAPs. Die Architektur erlaubt hohe Paketverarbeitungsraten bei niedrigem Leistungsverbrauch (z.B. X10q-e: 60 Mpps bei 6,5 Watt). Neuere Prozessorgenerationen, wie der X11 [37] oder die neue HX320- [38] bzw. HX330-Serie [39] sind dabei grundsätzlich nach dem gleichen Prinzip aufgebaut und unterscheiden sich im Wesentlichen durch eine angepasste Peripherie und höheren Durchsatz.

Netronome NFP-3200

Netronome als Nachfolger der Intel NP-Sparte stellt mit seinem NFP-3200 *Network Flow Processor* [40] den Nachfolger der bekannten IXP28XX-Familie. Als konsequente Weiterentwicklung der IXP-Reihe, setzt auch der neueste Chip auf die Verwendung von *Microengines* – kleiner, spezialisierter Prozessoren zur Paketverarbeitung. Damit ist der

NFP-3200 ein klassischer Vertreter der Cluster-Architektur. Während die 40 *Microengines* die *Data Plane*-Funktionalität übernehmen, kommt ein ARM11 Prozessor für *Control Plane* und *Management*-Funktionen zum Einsatz. Die *Microengines* unterstützen jeweils acht parallele Threads und sind mit 1,4 GHz getaktet. Integrierte Kryptografie-Beschleuniger ermöglichen die Unterstützung von IPsec-Verkehr mit bis zu 10 GBit/s.

Cisco QuantumFlow Prozessor

Der Cisco QuantumFlow Prozessor [41] ist eine relativ neue Entwicklung von Anfang 2008, der hohe Durchsatzraten bei hoher Flexibilität erreichen soll. Die Architektur zielt dabei sowohl auf den Einsatz bei *Service Providern* als auch im *Enterprise*-Bereich.

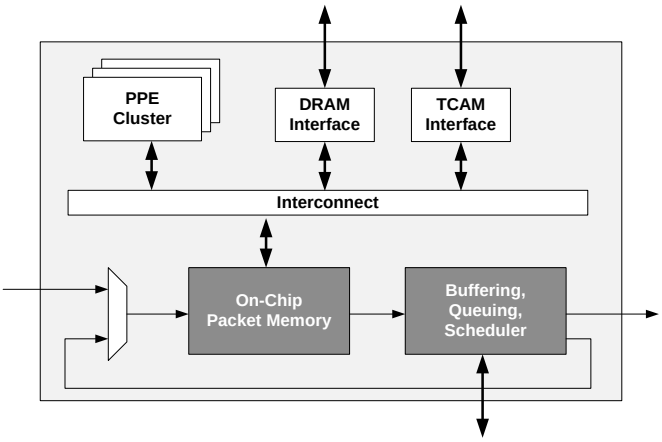

Abbildung 3.4: Architektur des Cisco QuantumFlow Processors.

Der QuantumFlow Processor besteht aus einem Cluster von 40 parallel arbeitenden *Packet Processor Engines* (PPE) – 32 Bit-RISC-Cores mit einer Taktfrequenz von 900 MHz – 1,2 GHz (vgl. Abbildung 3.4). Jede PPE unterstützt bis zu vier Threads. Die Programmierung erfolgt komplett in ANSI-C. Die PPEs sind über ein gemeinsames *Interconnect* an einen zentralen Speicher angebunden. Pro PPE existiert ein 16 KByte Level 1-Cache, alle PPEs teilen sich einen 256 KByte großen Level 2-Cache. Zusätzlich stehen diverse Hardwarebeschleuniger, wie *Network Address / Prefix Lookup, Hash Lookup, Traffic Policer* oder *TCAM* zur Verfügung. Jeder PPE erreicht 1.200 Millionen Instruktionen pro Sekunde (MIPS). Statusinformationen werden zentral in einer allgemein zugänglichen Datenbank gespeichert. Somit kann grundsätzlich jedes Paket von jeder PPE bearbeitet werden. Eine feste Zuordnung Flow zu PPE ist nicht notwendig. Die Pakete werden grundsätzlich im *run-to-completion*-Modus von einer PPE komplett bearbeitet. Nach der Bearbeitung werden die Pakete einem *Traffic Manager* zum *Queuing* und *Scheduling* übergeben. Eine Kommunikation der PPEs untereinander findet grundsätzlich nicht statt. Über einen *Hardware Lock Manager* kann bei Bedarf die Paketreihenfolge eines Flows garantiert werden. Der *Lock Manager* wird dabei über

Software gesteuert.

Cavium Octeon II

Als ein Beispiel für die Verwendung von (Standard-) RISC-Cores kann die Cavium Octeon Baureihe herangezogen werden. Die Prozessoren bestehen aus cnMIPS64 Cores – angepasster 64 Bit-Cores der Firma MIPS Technologies. Im Octeon II CN63XX [42] finden sich zwischen zwei und sechs (im angekündigten CN68XX sogar 32) parallel arbeitende Cores. Die einzelnen Cores werden mit bis zu 1,5 GHz getaktet und besitzen jeweils zwei MByte an Level 2 Cache.

Der NP enthält mehrere Hardwarebeschleuniger und *Crypto Engines*. Die Kommunikation erfolgt über ein 8 TBit/s *Crossbar Interconnect*. Der Octeon II erreicht Datenraten von bis zu 10 GBit/s für Forwarding mit QoS und 4 GBit/s bei *Deep Packet Inspection*.

EZchip TOPcore

Die NPs der NP-1, NP-2 und NP-3 Baureihen der Firma EZchip basieren auf der in [43] vorgestellten *Task Optimized Processing Core* (TOPcore) Technologie. EZchip geht hierbei davon aus, dass sich sämtliche Paketverarbeitung in vier unterschiedliche Grundaufgaben zerlegen lässt:

- **Parse**: Analysieren und Klassifizieren von Paket-Headern und anderen Feldern
- **Search**: Suche von Tabelleneinträgen (basierend auf den im 1. Schritt klassifizierten Feldern)
- **Resolve**: Bestimmen des Ziels, Auflösen von QoS-Anforderungen und Paket-Routing
- **Modify**: Aktualisierung der relevanten Paketfelder

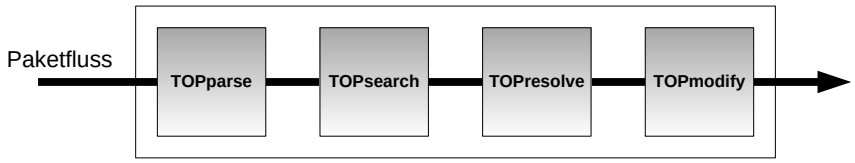

Abbildung 3.5: Paketfluss bei der TOPcore-Technologie von EZchip.

Für jede dieser Aufgaben kann nun ein separater, angepasster Prozessor eingesetzt werden (siehe Abbildung 3.5), der für die jeweilige Aufgabe optimiert ist und damit Geschwindigkeitsvorteile besitzt. Es werden auch einige Prozessierungsbeispiele angegeben. So reduziert sich z.B. die Anzahl der Instruktionen für das Parsen einer URL aus einem Paket von 400 (RISC) auf 60 (TOPcore).

Zur Steigerung des Durchsatzes können mehrere Pipelines parallel zu einer superskalaren Architektur ergänzt werden.

SafeXcel IP Inline Security Engine

Bei der *Inline Security Engine* (ISE) von SafeXcel [44] handelt es sich in erster Linie um einen Coprozessor für Sicherheitsanwendungen. Er wurde erstmals im Jahr 2006 vorgestellt. Der Prozessor ist im Rahmen dieser Arbeit insbesondere deshalb interessant, weil der konzeptionelle Ansatz Ähnlichkeiten mit dem in Kapitel 4 vorgestellten FlexPath-Konzept aufweist.

Die Grundidee bei der ISE von SafeNet besteht darin, bekannte Flows ohne weiteres Zutun eines externen Host-Prozessors in Hardware zu verarbeiten. Der Host-Prozessor selbst übernimmt im Idealfall lediglich den Verbindungsaufbau- und abbau. Die ISE unterstützt dabei *Data Plane*-Prozessierung bis hin zu IP/IPsec. Ein *Packet Classifier / Flow Processor* (vgl. Abbildung 3.6) überprüft alle ankommenden Pakete und entscheidet, ob das Paket der *Inline Packet Engine* bzw. dem Host-Prozessor übergeben wird, oder ob das Paket ausgefiltert und verworfen wird. Im Falle der Weiterleitung instruiert der *Packet Classifier / Flow Processor* automatisch die *Inline Packet Engine* und den *Post Processor*, welche Operationen auszuführen sind. Hierbei können diverse Datenmanipulationen (z.B. Einfügen, Ersetzen, Löschen) und eine Reihe unterschiedlicher Verschlüsselungen, Hash-Funktionen und Checksummen berechnet werden. Die *Inline Packet Engine* ist dabei als eine dreistufige Prozessierungspipeline ausgeführt.

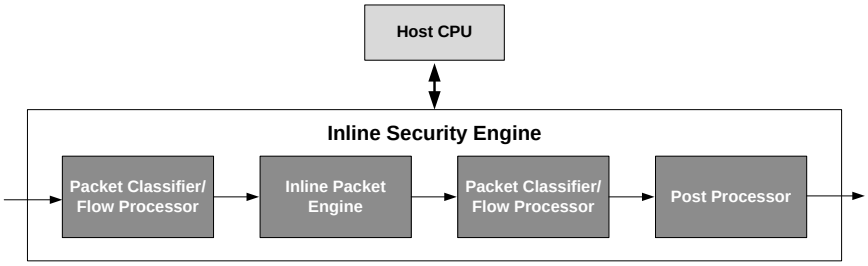

Abbildung 3.6: Architektur der SafeNet *Inline Security Engine*.

Die Inline Security Engine erreicht Durchsatzraten bei IPsec von bis zu 1 GBit/s (bei 1.500 Byte Paketgröße) bzw. 400 MBit/s (64 Byte Pakete) bei einer Taktfrequenz von 200 MHz.

3.1.2 Überblick über akademische Forschung zu Netzwerkprozessoren

Neben den zahlreichen kommerziell verfügbaren NPs, sind NPs auch weiterhin Gegenstand in unterschiedlichen akademischen Forschungsprojekten. Auch hier ist das Spektrum an Konzepten und Architekturen groß. Exemplarisch sollen im Folgenden ausgewählte akademische Ansätze vorgestellt werden.

PRO3 (Technische Universität Kreta)

Die PRO3 Architektur der Technischen Universität Kreta realisiert drei unterschiedliche Verarbeitungspfade innerhalb ihres NPs (vgl. hierzu [45] und [46]). Der PRO3 unterstützt hardwarebasierten Paketempfang und -speicherung. Rechenintensive Aufgaben bis hin zum *Network Layer* werden in programmierbarer Hardware in *wire-speed* behandelt. Komplexere Aufgaben der höheren Schichten werden an interne oder externe RISC-CPUs weitergeleitet.

Zu diesem Zweck werden ankommende Pakete durch einen Präprozessor untersucht. Dieser extrahiert relevante Header-Felder, überprüft die Checksummen und klassifiziert das Paket mithilfe eines externen TCAM. Die resultierende Flow-ID spezifiziert u.a. die für dieses Paket benötigte Recheneinheit und den auszuführenden Programmcode. Ein *Task Scheduler* leitet das Paket entsprechend der notwendigen Prozessierung an die RISC-CPU, ein *Reprogrammable Pipeline Module* (RPM) oder ggf. an den Ausgang weiter.

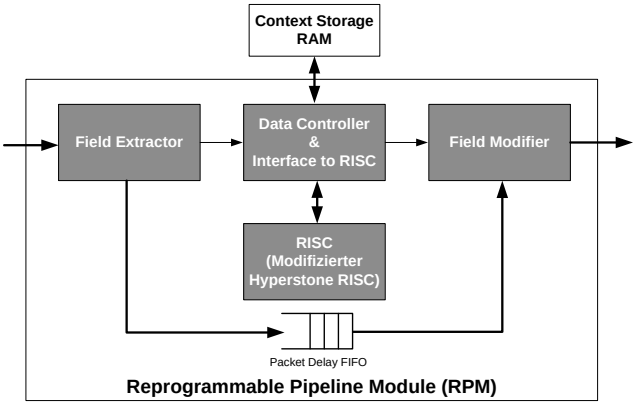

Abbildung 3.7: Vereinfachte Architektur des *Reprogrammable Pipeline Modules* beim PRO3.

Das RPM ist die zentrale Verarbeitungsinstanz im PRO3 und besteht aus einer dreistufigen Pipeline (siehe Abbildung 3.7). Der *Field Extractor*, eine kleine, spezialisierte RISC-CPU, stellt die benötigten Header-Felder bereits aufbereitet der eigentlichen Verarbeitungseinheit, bestehend aus einer angepassten Hyperstone RISC-CPU mit umgebender Hardware, bereit. Diese CPU besitzt zwei parallele Registersätze. Während die CPU auf einem Registersatz arbeitet, kann die umliegende Kontrolllogik parallel auf den zweiten Registersatz zugreifen, um neue Paketinformationen zu laden oder bereits bearbeitete Informationen zu speichern. Durch den Parallelbetrieb von Ein-/Ausgabe und RISC wirken sich Speicherlatenzen nicht auf die Verarbeitung aus, was zu einer entsprechenden Beschleunigung beiträgt.

Die *Field Modification*-Einheit schließlich arbeitet die Ergebnisse in das wartende Paket im Delay FIFO ein. Hierbei können entweder die entsprechenden Modifikation am Paket durchgeführt, oder aber bei Bedarf ein neues Paket erzeugt werden. Die *Field*

Modification-Einheit ist ein spezialisierter und optimierter RISC und voll programmierbar. Er unterstützt 16 unterschiedliche Instruktionen und besitzt fünf interne Register, ein Arbeitsregister, Programmzähler und einen Datenzeiger. Die 32 Bit-Architektur mit 200 MHz erzielt einen maximalen Durchsatz von 6,4 GBit/s. Der tatsächliche Durchsatz liegt jedoch niedriger, da einige Instruktionen mehrere Clockzyklen benötigten und ein 32 Bit-Wort zudem zum Teil aus mehreren separat empfangenen Feldern aufgebaut werden muss.

MIXMAP (Universität Stuttgart)

Der relativ neue MIXMAP-Ansatz [47] versucht die leichte Programmierbarkeit auf Softwareebene von NPs mit der effizienten Implementierung einzelner Aufgaben in FPGAs zu kombinieren. Hiermit soll ein Durchsatz von 100-200 Millionen Pakete pro Sekunde erreicht werden, die für 100 GBit/s-Ethernet ausreichen. Der NP wird komplett auf einem FPGA implementiert.

Um einen deterministischen Durchsatz zu erreichen, wurde als Ansatz eine Pipeline aus Funktionsblöcken gewählt. Jede Pipelinestufe verarbeitet pro Clockzyklus die jeweilige minimale Paketgröße (bei Ethernet also 64 Byte, entspricht im Folgenenden einem Wort), was mit einer entsprechend großen Datenpfadbreite verbunden ist. Parallel werden noch Meta-Daten des Paketes (z.B. Ports, Klassifizierungsergebnisse oder Paketgröße) mit übergeben. Optional kann durch die Pipeline nur das erste Wort (ausreichend z.B. bei Headerprozessierung) oder das ganze Paket transportiert werden. Im zweiten Fall sinkt bei größeren Paketen jedoch der Paketdurchsatz entsprechend.

Die Verarbeitungspipeline selbst ist modular aufgebaut. Jeder Funktionsblock besitzt dieselben Schnittstellen. Zum Einsatz können Funktionsblöcke kommen, die entweder auf Register-Transfer-Ebene beschrieben werden – verbunden mit einer hohen Effizienz. Es sind aber auch Module möglich, die komplett aus softwaregesteuerten Prozessoren aufgebaut sind. Hierbei muss jedoch pro Funktionsblock stets der Durchsatz von einem Wort pro Takt garantiert werden. Hierzu ist es also unter Umständen nötig, innerhalb des Funktionsblocks mehrere parallele Instanzen z.B. eines Prozessors zu verwenden.

Durch diesen Ansatz wird eine Realisierung auf unterschiedlichen Abstraktionsebenen (RTL bis Software) erreicht und kombiniert.

DynaCORE (Universität zu Lübeck)

Ebenfalls auf FPGA-Technologie basiert der dynamisch rekonfigurierbare Coprozessor DynaCORE [48][49] der Universität zu Lübeck. Es handelt sich hierbei um keinen vollständigen NP. Vielmehr werden unterschiedliche Hardwarebeschleuniger, z.B. zur Ent- bzw. Verschlüsselung bereitgestellt. Hierbei wird davon ausgegangen, dass nie alle zur Verfügung stehenden Beschleuniger gleichzeitig benötigt werden. Um Fläche zu sparen, stehen nur die jeweils benötigten Hardwarebeschleuniger bereit. Hierbei macht man sich die Eigenschaft der dynamisch partiellen Rekonfigurierbarkeit des FPGAs, also der Umprogrammierung von Teilen eines FPGAs zur Laufzeit, zu Nutze. Ein NP sendet oh-

ne Kenntnis der aktuellen Konfiguration Pakete zur Verarbeitung an den DynaCORE-Coprozessor. Wird eine Funktionalität angefordert, die aktuell nicht geladen ist, so wird entsprechend rekonfiguriert.

3.1.3 Zusammenfassung

In diesem Abschnitt wurden unterschiedliche Architekturen kommerzieller und akademischer Netzwerkprozessoren vorgestellt. Dabei lässt sich zusammenfassen:

- Die **Architekturvielfalt** bei Netzwerkprozessoren ist groß. Neben spezialisierten Prozessierungseinheiten (z.B. Xelerated, Netronome) kommen auch Standard-Cores (Cavium) und spezialisierte Hardwareeinheiten (z.B. EZchip, SafeXcel) zum Einsatz.
- Alle Prozessoren haben eine für den Networking-Bereich optimierte **Hardwareunterstützung**, allerdings in unterschiedlichem Umfang (z.B. optimierter Instruktionssatz, Speicheranbindung, Beschleuniger für spezielle Aufgaben).
- Neben Pipelinearchitekturen (Xelerated, EZchip) sind v.a. **Prozessierungscluster** stark verbreitet (Netronome, Cisco, Cavium).
- Die Architekturen orientieren sich stark am Einsatzzweck. Je spezialisierter der Einsatzbereich, desto spezialisierter die Architektur (siehe v.a. Architekturen im Kryptobereich, wie SafeXcel oder DynaCORE). Insbesondere NPs mit einem weiten Einsatzspektrum erlauben eine leichte bzw. hohe **Programmierbarkeit** (siehe z.B. Netronome, Cisco).

3.2 Lastbalancierung in Netzwerkprozessoren

Multiprozessor-Architekturen sind ein gängiger Ansatz bei NPs, um mit steigenden Datenraten und Prozessierungsanforderungen Schritt zu halten. Dabei ergibt sich jedoch das Problem, wie der Verkehr auf die einzelnen CPUs zur Verarbeitung verteilt wird. Wie in Kapitel 2.3.3 erläutert, ist insbesondere das TCP-Protokoll sensitiv auf Vertauschungen in der Paketreihenfolge. Dabei ist nicht entscheidend, dass die Reihenfolge des gesamten Verkehrs aufrecht erhalten wird, allerdings sollen Pakete einer TCP-Verbindung und damit eines Flows nicht vertauscht werden. Werden Pakete innerhalb eines Multiprozessor-Clusters jedoch willkürlich auf die einzelnen CPUs verteilt, kann die Einhaltung dieser Anforderung nicht mehr garantiert werden. Die Gründe liegen in unterschiedlichen Verarbeitungszeiten der Pakete aufgrund z.B. unterschiedlicher Speicherzugriffszeiten, parallel laufender Threads oder sonstiger Ressourcenkonflikte.

Die meisten Lastbalancierungsverfahren in der Literatur verfolgen daher den Ansatz, jeden Flow einer festen Verarbeitungseinheit zuzuordnen. Unterschieden werden dabei statische und dynamische Verfahren. Bei statischen Verfahren wird die Zuordnung zu Beginn festgelegt und bleibt danach erhalten. Da die einzelnen Flows jedoch eine zeitlich stark ausgeprägte Dynamik bezüglich der Paketrate haben können, kann es sehr leicht

zu Ungleichgewichten innerhalb eines Clusters kommen. Deshalb nehmen dynamische Verfahren bei Bedarf Umbalancierungen vor. Hierbei werden ein oder mehrere Flows einer neuen CPU zugewiesen, um das Ungleichgewicht zu beseitigen und Paketverluste zu vermeiden. Da jede Umbalancierung die Gefahr der Vertauschung der Paketreihenfolge mit sich bringt, wird generell versucht, die Frequenz der Umbalancierung möglichst klein zu halten. Nichtsdestotrotz sind Vertauschungen auf IP-Ebene grundsätzlich erlaubt und werden durch TCP – wenngleich auf Kosten der Leistung – abgefangen. Allerdings sollte deren Anteil möglichst klein gehalten werden.

3.2.1 Statische Lastbalancierungsverfahren

Bei Cao et al. [50] wurde ein statische Verfahren für die Lastbalancierung auf Internet-Links untersucht. Ziel war die direkte Abbildung des IP 5-Tupels auf eine Ziel-CPU. Hierzu wurden mehrere Abbildungsfunktionen untersucht. Es hat sich gezeigt, dass ein CRC16-Hash auf das 5-Tupel modulo der Anzahl der möglichen Ziele eine recht gute Verteilung ergibt. Da das Verfahren die Flows nur zu gleichen Teilen auf die Ziele aufteilen kann, nicht aber unterschiedliche Aktivitäten der Flowbündel berücksichtigt, wurde das Verfahren anschließend um eine *Hash Table* ergänzt. Der Hashwert teilt die Flows in M gleich große Flowbündel. Ein Flowbündel charakterisiert sich dabei also durch eine Menge von Flows mit identischem Hash-Wert. Diese Flowbündel werden anschließend anhand einer Tabelle einem der N möglichen Ziele zugewiesen. Dabei gibt es grundsätzlich wiederum zwei verschiedene Möglichkeiten (vgl. Abbildung 3.8).

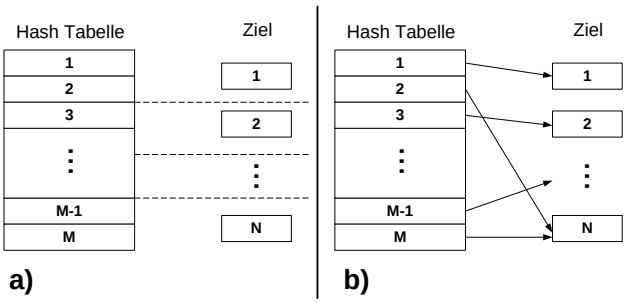

Abbildung 3.8: *Table-Based Hashing* Verfahren nach [50]: mithilfe eines Schwellwerts (a) oder direkter Zuweisung (b)

Bei der ersten Möglichkeit (a) werden die M Hashwerte mithilfe von N-1 Schwellwerten in N Partitionen geteilt. Jede Partition wird einem Ziel zugewiesen. Die Partitionen können bewusst unterschiedliche Größen erhalten. Das ist insbesondere dann hilfreich, wenn einem Ziel z.B. aufgrund geringerer Kapazitäten ein kleinerer Anteil übermittelt werden soll als einem anderen. Eine höhere Flexibilität erreicht man allerdings, wenn jedem Hashwert direkt ein Ziel zugeordnet werden kann (vgl. Abbildung 3.8b). Dies ist insbesondere von Bedeutung, falls manuell Anpassungen vorgenommen werden. Um gezielt Flowbündel von einem Link zum anderen zu transferieren, ist dies mit der direkten Zuweisung sehr einfach möglich. Bei Änderung der Schwellwerte müssen dagegen unter

Umständen sehr viele Schwellwerte angepasst werden (nämlich all jene, die zwischen den beiden anzupassenden Links sind). Dabei werden sehr viele Flowbündel einem neuen Ziel zugeordnet.

Bereits Cao et al. erkannten die Möglichkeit, die Zuordnung in festen periodischen Abständen dynamisch (z.B. anhand der Queue-Länge) anzupassen. Somit wird bereits im *Table-Based Hashing* eine dynamische Lastbalancierung angedacht.

3.2.2 Dynamische Lastbalancierungsverfahren

Lastbalancierung nach Dittmann

Bei Dittmann [51] wird das *Table-Based Hashing* nach Cao et al. aufgegriffen. Bei ankommenden Paketen wird eine Hash-Funktion auf das IP 5-Tupel ausgeführt. Das Ergebnis dient als Adresse für ein *Lookup Memory*, das an eine *Balancing Unit* angeschlossen ist (vgl. Abbildung 3.9). Dort ist jedem Flowbündel eine Verarbeitungseinheit zugeordnet. Im einfachsten Fall wird das Paket direkt in die zugehörige Queue weitergeleitet.

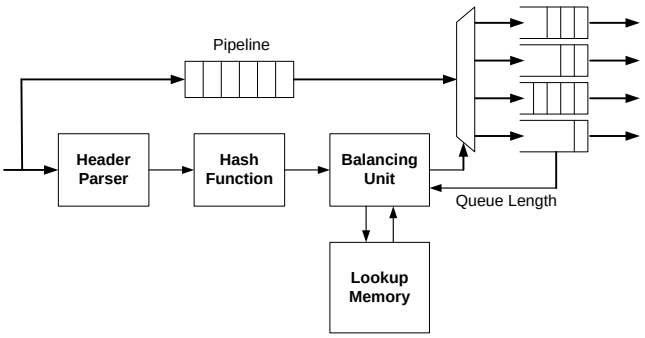

Abbildung 3.9: Lastbalancierungskonzept nach Dittmann (siehe [51]).

Um den teilweise stark variierenden Verkehr und der daraus resultierenden potentiellen Ungleichverteilung gerecht zu werden, sind verschiedene Umbalancierungsmaßnahmen implementiert. Zu jedem Flowbündel ist innerhalb des *Lookup Memories* ein Zeitstempel abgespeichert. Wann immer ein Paket eines Flowbündels ankommt, wird dieser aktualisiert. Zuvor jedoch wird der gespeicherte Wert mit der aktuellen Systemzeit verglichen. Überschreitet die Differenz einen festgelegten Wert – d.h. lag das letzte Paket des Bündels mindestens eine bestimmte Zeitspanne zurück – so kann davon ausgegangen werden, dass dieses Flowbündel zuvor nicht mehr aktiv war. Damit befinden sich keine Pakete des Bündels mehr im System und es kann ohne der Gefahr von *Packet Reordering* einer neuen Verabeitungseinheit zugewiesen werden. Hierbei wird jeweils die aktuell kürzeste Queue verwendet.

Dieses Verfahren kann längerfristige Ungleichgewichte zwar mildern, aber nicht immer verhindern. Insbesondere kurzfristige Ereignisse, wie Bursts innerhalb eines Flows können damit nicht hinlänglich erfasst werden. Deshalb greift hier ein weiterer Mechanismus: bei jedem ankommenden Paket wird zusätzlich untersucht, ob die eigentliche

Zielqueue bereits einen festgelegten Schwellwert überschreitet. Sofern dies der Fall ist, wird wiederum ein neues Ziel für das Flowbündel – nämlich die kürzeste Queue – festgelegt. Im Gegensatz zum ersten Mechanismus ist *Packet Reordering* hier nicht mehr ausgeschlossen.

Ein zusätzliches Problem ergibt sich bei Flowbündeln, die im Schnitt mehr Rechenleistung benötigen als eine Verarbeitungseinheit zur Verfügung stellen kann. Diese *Excessive Flows* werden durch eine Messung identifziert und durch einen weiteren Mechanismus verteilt. Da eine Verabeitungseinheit per se nicht in der Lage ist, diesen Flow zu bearbeiten, werden Pakete des Bündels generell zur jeweils kürzesten Queue gesendet. Da die kürzeste Queue über der Zeit des öfteren wechseln kann, und damit auch die Verarbeitungseinheit, spricht man hier von *Packet Spraying*.

Es bleibt zu erwähnen, dass bei zustandsbehaftetem Verkehr bei Verwendung lokaler Zustandsspeicher eine Umbalancierung zu einem Verlust der Statusinformationen führt. Diese kurzfristige Störung wird jedoch akzeptiert, um längerfristig Paketverluste zu vermeiden.

Adaptives Lastbalancierungverfahren nach Kencl und Shi

Zentrales Element des adaptiven Balancierungsverfahrens von Kencl [52] ist ebenfalls eine Hash-Tabelle. Die Zuordnung zu den Verarbeitungseinheiten wird regelmäßig durch eine Kontrollschleife neu berechnet. Die Berechnung basiert auf dem *Highest Random Weight* (HRW) Algorithmus. Der Algorithmus berechnet für jedes Flowbündel aus dem Flow-Hashwert für alle mögliche Zielindizes eine zufällige Gewichtung. Multipliziert mit der jeweiligen aktuellen Auslastung ergibt sich für jede Verarbeitungseinheit ein *Score*, wobei die Zuordnung mit dem höchsten *Score* in die Hash-Tabelle übernommen wird. Damit sinkt die Wahrscheinlichkeit der Zuordnung zu einer überlasteten Verarbeitungseinheit und steigt für eine in Unterlast. Es kann gezeigt werden, dass bei notwendigen Umbalancierungen nur ein verhältnismäßig kleiner Teil der Flowbündel betroffen ist. Dadurch sinkt auch die Wahrscheinlichkeit für *Packet Reordering*.

Bei Kencl werden Flowbündel zur Umbalancierung also letztlich mehr oder weniger zufällig ausgewählt. Hierbei wird nicht darauf geachtet, welchen Anteil das einzelne Bündel an der jeweiligen Gesamtauslastung hat. Es zeigt sich jedoch, dass bei typischem IP-Verkehr ein Großteil der Flows eine relativ geringe Aktivität besitzt und damit eine relativ geringe Auslastung erzeugt. Dahingegen gibt es verhältnismäßig wenige Flows mit sehr hoher Aktivität [53]. Dies wiederum bedeutet, dass im Schnitt viele Flows, respektive Flowbündel umbalanciert werden müssen, um einen nennenswerten Effekt auf die Auslastung zu erzielen. Deshalb geht Shi [54] einen anderen Weg: während der Großteil der Flowbündel statisch per Hashing zugewiesen wird, werden kontinuierlich die aktivsten Flowbündel (die sogenannten *Aggressive Flows*) identifiziert. Im nicht balancierten Zustand wird parallel der *Load Adapter* aktiv. Pakete eines *Aggressive Flows* werden nun auf die kürzeste Verarbeitungsqueue umgeleitet. Die Pfadentscheidung des Hashingalgorithmus wird damit überschrieben. Da nur die aktivsten Flows umgeleitet werden, kann ein relativ großer Effekt auf die Auslastung der einzelnen Verarbeitungseinheiten mit

wenigen Umbalancierungen erreicht werden. Damit ist auch die Gefahr für *Packet Reordering* relativ gering. Aufgrund der geringen Anzahl an Umbalancierungen wird zudem eine Störung der lokalen Lookup-Caches in den Verarbeitungseinheiten reduziert.

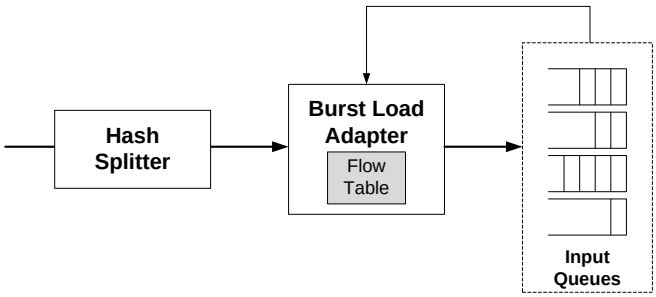

Abbildung 3.10: Das HABS Verfahren nach Kencl und Shi (siehe [55])

Kencl und Shi haben ihre Verfahren in [55] weiterentwickelt und kombiniert. Im *Hashing Adapted by Burst Shifting (HABS)* Algorithmus wird ein *Hash Splitter* einem *Burst Load Adapter* vorgeschaltet (siehe Abbildung 3.10). Der *Hash Splitter* implementiert das *Adaptive HRW Hashing* von Kencl [52], womit längerfristige Unausgeglichenheiten behoben werden. Für kurzfristige Schwankungen wurde der *Burst Load Adapter* von Shi [56] nachgeschalten. Eine Flowtabelle kann für eine begrenzte Anzahl an Flows die Entscheidung des *Hash Splitters* überschreiben. Für jedes Paket wird die Flowtabelle deshalb nach vorhandenen Einträgen durchsucht und ggf. die Pfadentscheidung übernommen. Falls noch kein Eintrag vorhanden ist, so wird ein neuer Eintrag in der Flowtabelle angelegt, falls folgende drei Bedingungen zutreffen:

1. Das Paket soll originär auf eine überlastete Verarbeitungseinheit geleitet werden (erkennbar am Pufferfüllstand)
2. Das Paket ist der Beginn eines Bursts, d.h. es darf sich kein weiteres Paket desselben Flows bereits im System befinden. Nur dann kann ein Umschalten ohne der Gefahr von *Packet Reordering* erfolgen.
3. Es sind freie Einträge in der Flowtabelle vorhanden.

Für alle Einträge der Flowtabelle muss parallel überwacht werden, ob sich aktuell noch Pakete des zugehörigen Flows im System befinden. Falls dies nicht mehr der Fall ist, wird der zugehörige Eintrag gelöscht und steht einem neuen Flow bei Bedarf zur Verfügung.

3.2.3 Zusammenfassung

In diesen Abschnitt wurden unterschiedliche Lastbalancierungsverfahren für homogene CPU-Cluster vorgestellt. Dabei wird zwischen statischen und dynamischen Verfahren unterschieden. Bei **statischen Verfahren** erfolgt eine Zuordnung vom Flow (oder Flowbündel) und Prozessierungseinheit zur Designzeit und wird zur Laufzeit nicht mehr

verändert. Bei **dynamischen Verfahren** dagegen wird versucht, zeitlichen Schwankungen im Verkehr durch Umbalancierungen von einzelnen Flows oder Flowbündeln zu begegnen. Ziel ist ein möglichst ausgeglichenes Prozessierungscluster und die Vermeidung von unnötigen Paketverlusten. Dabei gilt jedoch zu beachten:

- Pakete eines Flows müssen in der Regel von jeweils **derselben Prozessierungseinheit** bearbeitet werden. Nur dadurch kann es innerhalb des Flows zu keinen Paketüberholungen kommen.
- Bei der Umbalancierung besteht die Gefahr von **Packet Reordering**. *Packet Reordering* hat jedoch negative Auswirkungen auf die Netzwerkleistung. Daher ist die Anzahl der Umbalancierungen zu minimieren. Damit schränkt das *Packet Reordering*-Problem die Möglichkeiten und die Effektivität der Lastbalancierungsverfahren massiv ein.
- Alle bekannten Lastbalancierungsverfahren besitzen eine gewisse **Trägheit**. Damit ist es schwierig, auf kurzfristige Änderungen im Verkehr zu reagieren.
- Ein Großteil der Flows hat eine geringe oder keine Aktivitität. Die **Identifizierung von besonders aktiven Flows** kann daher bei der Balancierung helfen. Dies ist allerdings mit einem entsprechenden Aufwand verbunden.

3.3 Packet Reordering und Resequenzierung im Netzwerkbereich

In Abschnitt 3.2 wurden mehrere Lastbalancierungsverfahren vorgestellt, die versuchen, die Paketreihenfolge innerhalb eines Flows weitestgehend aufrechtzuerhalten. Das geschieht in der Regel dadurch, dass Pakete eines Flows von der gleichen Prozessierungseinheit verarbeitet werden und sich dadurch nicht überholen können. Nur in Überlastsituationen wird *Packet Reordering* zugunsten der besseren Ressourcenausnutzung teilweise toleriert. Außerdem kann durch Umbalancierung eines Flows die Paketreihenfolge nicht immer garantiert werden.

Dieser Grundsatz schränkt die Lastbalancierung auf zweierlei Weise ein:

- Die **Anzahl der Umbalancierungen muss klein gehalten werden**, da jede Umbalancierung *Packet Reordering* nach sich ziehen kann.
- Die **Umbalancierung findet in der Regel immer auf ganzen Flows oder Flowbündeln statt**. Damit ist die Granularität der Umbalancierung stark eingeschränkt. Insbesondere ist es teilweise schwierig bzw. mit einem entsprechenden Aufwand verbunden, den passenden Flow (in Abhängigkeit von der Paketrate) zur Umbalancierung zu identifizieren.

Es gibt aber noch eine zweite Herangehensweise, um das Problem des *Packet Reordering* zu lösen. Anstatt *Reordering* im Ansatz zu vermeiden, können die Pakete ausgangsseitig wieder in Reihenfolge gebracht werden. Dadurch können Restriktionen für den Lastbalancierungsalgortihmus beseitigt werden. Freiheitsgrade werden geschaffen,

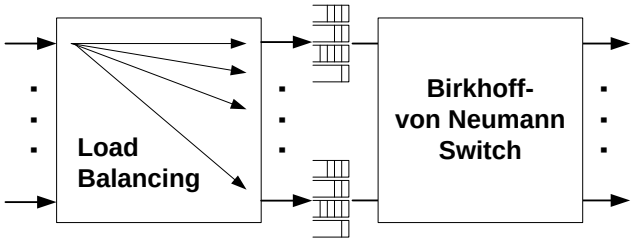

Abbildung 3.11: *Load Balanced Switch* nach Chang et al. (siehe [57])

die insbesondere für eine optimale Ressourcenauslastung hilfreich sind.

Nichtsdestotrotz findet das Prinzip der Paket-Resequenzierung in der Literatur wenig Beachtung. Fast alle Ansätze versuchen, *Packet Reordering* bereits durch den Lastbalancierungsalgorithmus zu vermeiden. Innerhalb von NPs werden bestenfalls softwarebasierte Mechanismen eingesetzt. Diese können jedoch den Durchsatz erheblich einschränken. Im Folgenden wird ein softwarebasierter Ansatz vorgestellt, der mit den eingebauten Mechanismen des Intel IXP 2400 verglichen wird. Außerdem wird eine Hardware-Implementierung vorgestellt und diskutiert. Zuvor wird jedoch auf eine ähnliche Problemstellung bei *Load Balanced Switches* eingegangen, da es auch dort konzeptbedingt zu Paketüberholungen kommt.

3.3.1 Paket-Resequenzierung in Load Balanced Switches

Ein Resequenzierungs-Problem lässt sich auch bei *Crossbar Switch*-Architekturen feststellen, wie sie z.B. innerhalb *Core Router*-Architekturen verwendet werden.

Eine *Crossbar* kann jeden Eingangsport mit jedem Ausgangsport verbinden. Es wird dabei von uniformen Paketgrößen und synchronisierten Zeitscheiben ausgegangen. Zeitgleich können damit genauso viele Pakete an die Ausgänge übertragen werden, wie Eingänge vorhanden sind, allerdings nur solange diese auf unterschiedliche Ausgangsports gesendet werden. Sollen gleichzeitig Pakete auf denselben Ausgangsport übertragen werden, so müssen diese warten. Hierfür sind in der Regel für jeden Eingangsport Puffer angebracht. Der Durchsatz der *Crossbar* ist damit kleiner als 100%.

Dieses Problem soll bei *Load Balanced Switches* umgangen werden (siehe [57]). Grundlage ist eine zweistufige *Crossbar* mit zwischengeschalteten Puffern (siehe Abbildung 3.11). Die Pufferstruktur wurde dahingehend erweitert, dass jeder Eingangsport für jeden Ausgangsport einen Puffer besitzt. Die erste Stufe – der *Load Balancer* – verteilt die Pakete pro Eingangsports auf die Puffer. Die zweite Stufe stellt den eigentlichen *Crossbar Switch* dar. Durch die gleichmäßige Verteilung der Eingangspakete auf die Eingänge des *Crossbar Switch* (also der zweiten Stufe), kann ein 100%iger Durchsatz erreicht werden. Außerdem soll diese Architektur eine gute Skalierbarkeit, geringe Hardwarekomplexität und geringe Verzögerungslatenzen besitzen. *Load Balanced Switches* sind damit eine vielversprechende Alternative zu herkömmlichen Switch-Architekturen. Allerdings ergibt sich dabei ein anderes Problem. Da die Pakete eines Eingangsports unterschied-

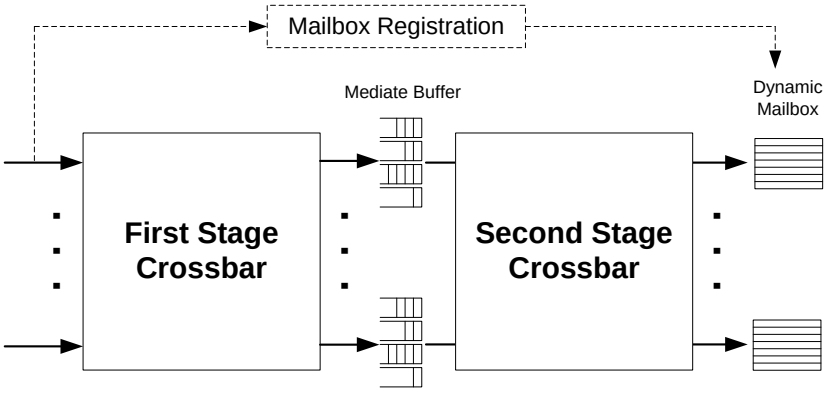

Abbildung 3.12: *Per-flow Resequencing* in Switches nach Cheng et al. (siehe [59])

liche Wege benutzen, führt dies zu unterschiedlichen Latenzen und damit potentiellen Paketüberholungen innerhalb eines Flows.

Eine Möglichkeit, dem *Reordering*-Problem zu begegnen, ist in [58] beschrieben. Dort wird eine effiziente Resequenzierung erreicht, indem Pakete eines Flows gleichmäßig auf die Eingänge der zweiten Switch-Stufe verteilt werden. Durch die deterministischen Grenzen für die Durchlaufzeit kann die Größe der Resequenzierungs-Puffer am Ausgang exakt bestimmt werden. Zu beachten ist, dass der Begriff Flow in diesem Zusammenhang als Pakete mit gleichen Eingang- und Ausgangsport definiert ist.

In [59] wird daher auch argumentiert, dass eine Resequenzierung auf Port-Granularität eine unnötige Einschränkung darstellt. Letztlich ist lediglich die Reihenfolge innerhalb eines Applikations-Flows entscheidend. Deshalb zielt dieser Ansatz darauf ab, die Reihenfolge innerhalb des Applikations-Flows wiederherzustellen (siehe Abbildung 3.12). Ankommende Pakete werden eingangsseitig registriert und dem betreffenden Ausgangsport mitgeteilt. Die übermittelte Nachricht enthält den Flow und eine Paket-ID. Die ausgangsseitige Mailbox überprüft, ob für den Flow bereits eine Sortierungsliste vorhanden ist. Falls nicht, wird eine neue Liste angelegt und die Paket-ID des ersten Pakets festgehalten, ansonsten wird die Paket-ID an die vorhandene Liste angehängt. Somit ist die originale Paketreihenfolge auf Flowebene konserviert. Ausgangsseitig wird für jedes Paket überprüft, ob es sich um das erste Paket der Liste handelt. Sofern das Paket gesendet werden kann, rücken die restlichen registrierten Pakete in der Reihenfolge nach vorn, bzw. falls keine weiteren Pakete des Flows aktuell vorhanden sind, wird die Liste wieder freigegeben. Sofern das Paket noch nicht gesendet werden kann, wird es in einem Puffer zwischengespeichert. Da von mehreren simultan aktiven Flows in *High Speed*-Routern ausgegangen wird, werden ebenso viele Sortierungslisten benötigt. Simulationen mithilfe eines OC-48-Verkehrsmitschnitts ergaben eine Re-Sequenzierungsquote von kleiner 0,1% bei maximal 208 Sortierlisten und maximal 15 Einträgen pro Liste.

Auch wenn diese Ansätze erste Hinweise auf mögliche Resequenzierungs-Konzepte geben können, so kann man innerhalb der Switch-Architektur teilweise von Voraussetzungen ausgehen, die innerhalb eines Netzwerkprozessors nicht unbedingt gegeben sind:

- Die maximale und minimale Latenz innerhalb des Switches ist relativ leicht kalkulierbar. Dies limitiert die Größe der Sortierungspuffer und erleichtert die Handhabung.
- Der Switch ist verlustfrei. Sämtliche Pakete am Eingang verlassen den Switch auch wieder. Damit können Sortierlisten relativ einfach verwaltet werden.

3.3.2 Paket-Resequenzierung in Netzwerkprozessoren

Paket-Resequenzierung innerhalb eines NPs kann durch softwarebasierte oder hardwarebasierte Verfahren durchgeführt werden. Während es bei den meisten NPs Verfahren in Software gibt, um die Paketreihenfolge aufrecht zu erhalten (siehe z.B. Cisco QuantumFlow Processor, Abschnitt 3.1.1), finden sich in der Literatur kaum Verfahren zur Resequenzierung in Hardware.

Softwarebasierte Resequenzierungs-Verfahren

Govind et al. [16] hat die Auswirkungen von *Packet Reordering* eines Intel IXP 2400 NPs [60] untersucht. Außerdem hat er den Durchsatz zweier IXP-Methoden zur Aufrechterhaltung der Paketreihenfolge simuliert. Die beiden Verfahren wurden anschließend einem eigenen, softwarebasierten Sortieralgorithmus gegenübergestellt.

Der Intel IXP 2400 besteht aus acht 32 Bit-*Microengines* mit jeweils acht Threads pro *Microengine*. Neben externen Speichern (DRAM, SRAM), einem 32 Bit-RISC-Xscale Prozessor für Kontroll- und Managementfunktionen, besitzt er u.a. je einen *Receive* und *Transmit Buffer* mit je 8 KByte.

Packet Reordering im IXP kann nun aufgrund von zwei Ursachen auftreten:

- Pakete eines Flows werden unterschiedlichen Threads, u.U. auf unterschiedlichen *Microengines*, zur Verarbeitung zugewiesen. Aufgrund unterschiedlicher Verarbeitungslatenzen (z.B. Ausführung eines anderen Threads auf der *Microengine* oder Zugriff auf DRAM) kann es zu Paketüberholungen kommen.
- Pakete werden nach der Verarbeitung in den *Transmit Buffer* geschrieben. Um einen möglichst effizienten und konfliktfreien Zugriff zu ermöglichen, wird jedem Thread dabei eine feste Speicherzelle im *Transmit Buffer* zugeordnet. Damit hängt der Zustand der Speicherzelle vom Zustand des zugehörigen Threads ab, womit der *Transmit Buffer* nicht der Reihe nach beschrieben wird. Nachdem das Auslesen des *Transmit Buffers* jedoch FIFO-mäßig der Reihe nach geschieht, kann es hierbei zur Vertauschung der Paketreihenfolge kommen.

Der IXP-Prozessor unterstützt nun zwei verschiedene Mechanismen, um die Paketreihenfolge aufrecht zu erhalten:

- **Inter Thread Signaling (ITS)**: Hierbei wird Start und Ende der jeweiligen Paketverarbeitung zwischen den einzelnen Threads synchronisiert. Die einzelnen Threads beginnen ihre Verabeitung sequentiell und warten am Ende auf das ent-

sprechende Signal des vorhergehenden Threads um die Bearbeitung auch wieder sequentiell zu beenden. Dazwischen laufen die Threads unabhängig voneinander. Da nun auch die Allokation des *Transmit Buffers* sequentiell stattfindet, werden die Pakete in der korrekten Reihenfolge gesendet.

- **Asynchronous Insert Synchronous Remove (AISR)**: Hierbei wird die Verarbeitung in vier Stufen aufgeteilt: *Speicherung*, *Prozessierung*, *Resequenzierung* und *Versenden*. Das Paket wird hierbei im DRAM gespeichert. Dabei wird ihm eine eindeutige Sequenznummer zugewiesen. Anschließend können die Pakete unabhängig voneinander prozessiert werden, ehe sie in der nächsten Stufe anhand der Sequenznummer wieder in die korrekte Reihenfolge gebracht werden. Die Resequenzierung erfolgt anhand eines einfachen *Countingsort* Verfahrens. Abschließend werden die Pakete durch die letzte Stufe in den *Transmit Buffer* geschrieben.

Govind hat die Leistung des IXP für IPv4-Forwarding anhand eines Petri-Netz Modells simuliert. Der IXP erreicht eigentlich Forwardingraten von ca. 3 GBit/s. Durch das ITS-Verfahren wird der Durchsatz auf rund 2,3 GBit/s beschränkt, durch AISR sogar auf lediglich 1,1 GBit/s. Dadurch erfolgt eine Reduktion des Durchsatzes um 23% bzw. 63% durch die Paket-Resequenzierung. Damit können die für den IXP2400 angepeilten 2,5 GBit/s eines OC-48-Links nicht mehr erreicht werden.

Govind hat deshalb einen eigenen Paketsortieralgorithmus beschrieben und simuliert. Er teilt die Prozessierung in drei Stufen auf. Die erste Stufe ist für die Speicherung im DRAM (in Abhängigkeit von der Flowinformation) und die eigentliche Paketprozessierung zuständig. Die Prozessierung auf den einzelnen Threads erfolgt unabhängig voneinander. In der zweiten Stufe werden die Pakete anhand der Adresse mithilfe eines *Insertionsort*-Algorithmus resquenziert und schließlich in der dritten Stufe vom DRAM zum *Transmit Buffer* transferiert. Anhand der Simulationen hat sich folgende Aufteilung der *Microengines* als günstig erwiesen: vier *Microengines* sind für die Speicherung/Prozessierung zuständig, einer für die Sortierung und drei für den Speicher bzw. *Transmit Buffer*-Transfer. Abhängig von der Anzahl der prozessierten Flows konnte ein Durchsatz von 2,5 GBit/s erreicht werden. Lediglich für weniger als zehn Flows wurden kleinere Durchsatzraten (1,7 GBit/s bei einem Flow) erreicht. Da sich in der Realität aber durchwegs mehr Flows gleichzeitig im System befinden, können die für OC-48 geforderten 2,5 GBit/s erreicht werden.

Die Verwendung von softwarebasierten Verfahren deutet bereits an, dass eine Aufrechterhaltung, respektive Wiederherstellung der originalen Paketreihenfolge mitunter eine negative Auswirkungen auf den Datendurchsatz haben kann. Selbst bei dem vergleichsweise schnellen Verfahren von Govind wird der Durchsatz von 3 auf 2,5 GBit/s reduziert, was in etwa 17% entspricht. Bei den IXP-Verfahren wiegt aber ein anderer, zunächst wenig augenscheinlicher Effekt schwerer. Sowohl beim ITS- als auch beim AISR-Verfahren wird die Paketreihenfolge global aufrecht erhalten. Beim AISR-Verfahren wird jedem Paket eine inkrementelle Sequenznummer zugeteilt – unabhängig vom jeweiligen Flow. Solch ein Verfahren ist jedoch nur solange unproblematisch, solange eine (nahezu) uniforme Verarbeitungslatenz erreicht werden kann. In dem dargestellten Beispiel

ist dies dadurch gegeben, dass für jedes Paket IPv4-Forwarding ausgeführt wird – eine wohldefinierte Aufgabe, bei der sich ein Verarbeitungsjitter in Grenzen hält. Problematisch wird dies in einer heterogenen Applikations-Umgebung, wie sie sich z.B. in einem *Edge Router* wiederfindet. Neben reinem Forwarding können auch andere Tasks, wie z.B. VPN / IPsec gefordert sein, die wesentlich höhere Rechenanforderungen und damit Verarbeitungslatenzen besitzen. Wird nun die Reihenfolge von Paketen mit kurzer Prozessierungsdauer mit Paketen mit langer Prozessierungsdauer gemischt aufrechterhalten, so führt dies im besseren Fall zu großen Resequenzierungs-Puffern und/oder im schlechteren Fall zur Blockierung von Rechenressourcen. Am Beispiel ITS kann zum Beispiel ein Thread mit sehr langer Rechendauer dazu führen, dass andere Threads unnötig lange auf dessen Freigabe warten. Somit sind diese Threads für die Dauer der Berechnung blockiert und können keine neuen Pakete prozessieren.

Hardwarebasierte Resequenzierungs-Verfahren

Wu et al. [61] versuchen die Probleme mit einer Hardware-Resequenzierung zu lösen: durch den Einsatz von Hardware wird das Verarbeitungscluster entlastet, somit steht die volle Rechenleistung zur Prozessierung zur Verfügung. Außerdem findet die Resequenzierung auf Flow-Granularität statt. Damit stellen unterschiedliche Verarbeitungslatenzen der verschiedenen Flows kein Problem mehr bei der Resequenzierung dar.

Grundsätzlich wird ein ähnlicher Ansatz wie er bereits in Abschnitt 3.3.1 vorgestellt wurde gewählt: die Reihenfolge der Pakete wird eingangsseitig vor der Verteilung auf die einzelnen Verarbeitungseinheiten festgehalten und nach der Verarbeitung durch einen *Aggregator* auf Flowebene wiederhergestellt (vgl. Abbildung 3.13).

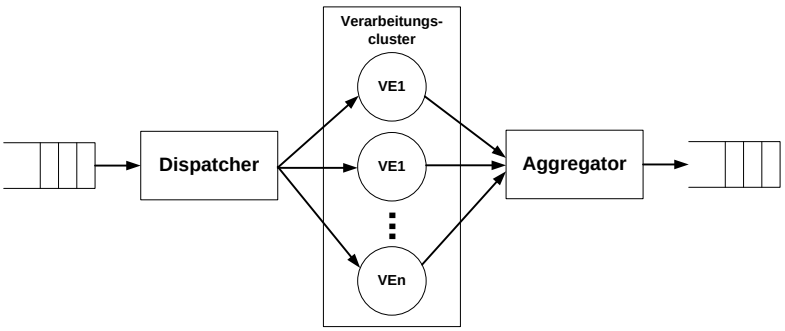

Abbildung 3.13: Verarbeitungscluster mit nachgeschalteten *Aggregator* zur Paket-Resequenzierung (vgl. [61]).

Das Herzstück der Implementierung von Wu ist die Flow-Tabelle mit angehängter Paketverkettungsliste (siehe Abbildung 3.14). In dieser zentral verwalteten Datenstruktur wird die originale Paketreihenfolge auf Flow-Ebene gespeichert. Die Flow-Tabelle hat ganz allgemein Einträge für maximal k unterschiedliche Flows. Bei Ankunft eines Pakets wird mittels eines *Content Addressable Memories* (CAM) überprüft, ob bereits ein Eintrag für diese Flow-ID existiert. Wu liefert hier keine genaue Definition zur Flow-ID,

es wird aber von einem Wert von mindestens 32 Bit ausgegangen. Sofern bereits ein Eintrag vorhanden ist, verweist der Index (also die Adresse) des Speichereintrags des CAMs auf eine Adresse im SRAM. Ist kein Eintrag vorhanden und sind damit aktuell noch keine weiteren Pakete des Flows im System, so wird ein freier Eintrag verwendet und die Flow-ID im CAM gespeichert. Der SRAM enthält neben der Anzahl der aktuell vorhandenen Pakete im System ferner einen Zeiger auf das erste und letzte Paket der verketteten Liste des Flows. Die verkettete Liste wird in der Block-Tabelle – einer weiteren Datenstruktur – verwaltet. Wu geht von einem Block pro Paket aus. Die Block-Tabelle enthält für jedes Paket u.a.:

- Ein *Head*-Flag, das anzeigt ob das Paket der Beginn einer verketteten Liste ist.
- Ein *Finished*-Flag, das anzeigt, ob das Paket bereits prozessiert wurde.
- Die Flow-ID des Pakets.
- Ein Zeiger auf die Adresse des nächsten Pakets der verketteten Liste.

Bei Ankunft eines Pakets muss somit im SRAM der Zeiger auf das letzte (und ggf. erste) Paket geändert werden. Außerdem wird der Eintrag in der Block-Tabelle für das aktuelle Paket und ggf. der Zeiger des letzten Pakets auf das nun aktuelle Paket aktualisiert. Dies schließt mit ein, dass für jedes neue Paket zunächst ein freier Eintrag in der Block-Tabelle gefunden werden muss.

Eine Thread-Tabelle verwaltet schließlich die Zuordnung zwischen Thread und Paket. Sobald der Thread die Bearbeitung des Pakets abgeschlossen hat, wird diese Zuordnung wieder gelöscht und das *Finished*-Flag der Block-Tabelle gesetzt. Ein weiterer Suchalgorithmus durchforstet unabhängig hiervon die Block-Tabelle nach Einträgen die bereits bearbeitet und Kopf einer verketteten Liste sind. Nur diese Pakete können ohne der Gefahr von *Packet Reordering* gesendet werden. Anschließend wird der Eintrag der Block-Tabelle gelöscht und der Folgeeintrag der verketteten Liste aktualisiert (Setzen des Head-Flags). Außerdem muss der entsprechende Zeiger in der Flow-Tabelle aktualisiert werden bzw. sofern das Paket das letzte der Liste war, der Eintrag komplett gelöscht werden.

Sowohl eingangsseitig als auch ausgangsseitig wird ein *Round Robin*-Algorithmus zur Suche freier Einträge in der Block-Tabelle bzw. zur Suche sendefertiger Pakete verwendet. Die Autoren erwähnen aber die Möglichkeit der Nutzung komplexerer Algorithmen, unter Umständen auch unter Berücksichtigung weiterer Flags wie Prioritätsflags.

Zur Verifikation des Ansatzes wurde ein System mit *Dispatcher/Aggregator* und vier *OPENRISC*-Prozessoren auf einem Altera Stratix FPGA implementiert. *Dispatcher* und *Aggregator* belegten dabei 3.963 Logikelemente des FPGAs bei einem Speicherbedarf von rund 65 KByte. Sie konnten mit einer Taktfrequenz von 133 MHz betrieben werden. Die Block-Tabelle war auf 32 Einträge beschränkt.

Zur Validierung wurden Simulationen mit unterschiedlichen Verkehrs-Traces durchgeführt. Dabei wurde der Verkehr in eine paketlängenunabhängige Prozessierung (IPv4-Forwarding) eingeteilt und in eine paketlängenabhängige. Für das Forwarding wurden 50 Instruktionen angesetzt, für die längenabhängige Prozessierung dagegen sechs Instruk-

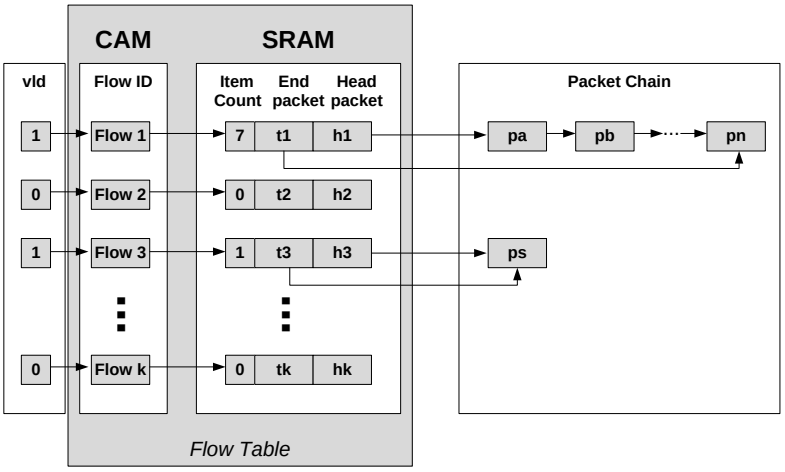

Abbildung 3.14: Struktur der Flow-Tabelle in der Implementierung von Wu et al. (vgl. [61])

	Längenunabh. [%]	Längenabh. [%]	min. Paketlänge	max. Paketlänge
Trace 1	100%	0%	-	-
Trace 2	95%	5%	40	80
Trace 3	90%	10%	60	120

Tabelle 3.1: Parameter der drei verwendeten Traces bei Wu et al.

tionen pro Byte Paketlänge. Der Anteil der Pakete mit längenabhängiger Prozessierung wurde in Trace 2 und 3 auf 5% bzw. 10% festgelegt, die Paketlänge schwankte dabei zwischen 40 und 80 Byte, bzw. 60 und 120 Byte (siehe Tabelle 3.1).

Die Paketreihenfolge konnte bei allen Tests aufrecht erhalten werden. Dabei ergab sich ein Systemdurchsatz von 3,4 GBit/s (Trace 1), 1,8 GBit/s (Trace 2) und 1,2 GBit/s (Trace 3). Dabei konnte nur mit Trace 1 eine relativ hohe CPU-Auslastung erreicht werden. Zu 90% der Zeit waren drei oder vier CPUs ausgelastet. Nur in 0,04% der Fälle waren alle Paketspeicherplätze besetzt.

Ein anderes Bild ergibt sich, sobald die paketlängenabhängige Verarbeitung (Trace 2 und 3) betrachtet wird. Durch die Payload-Prozessierung ergibt sich rein rechnerisch eine Verlängerung der durchschnittlichen Verarbeitungszeit von 50 Instruktionen pro Paket (Trace 1) auf 65,5 (Trace 2), respektive 99 Instruktionen (Trace 3). Die gesteigerte Rechenanforderung würde deshalb theoretisch einen Durchsatz von 2,6 GBit/s (Trace 2) und 1,7 GBit/s (Trace 3) erwarten lassen. Die Differenz zu den gemessenen Werten lässt sich größtenteils durch das *Packet Reordering* erklären. Die paketlängenabhängige Verarbeitung erzeugt bei Paketen eines Flows verhältnismäßig große Prozessierungsjitter und somit auch *Packet Reordering*. Pakete außerhalb der Reihenfolge können jedoch noch nicht versendet werden und blockieren somit einen der limitierten Paketspeicherplätze. Tatsächlich misst Wu zu 3% bzw. 16% (Trace 2 bzw. 3) der Zeit keinen einzigen frei-

en Speicherplatz und zu 10% bzw. 14% keine einzige aktive CPU. Dadurch ergeben sich Hinweise, dass die Speicherplatzbelegung aufgrund der Resequenzierung zu einer zeitweisen Blockierung der CPUs und damit einer verminderten Leistung des Gesamtsystems führt.

3.3.3 Zusammenfassung

In Abschnitt 3.3 wurden unterschiedliche Resequenzierungsmöglichkeit im Netzwerkbereich vorgestellt. Die aktive Resequenzierung ist dabei eine Alternative zur Vorgehensweise der Lastbalancierungsverfahren aus Abschnitt 3.2, die *Packet Reordering* aufgrund der Paketverteilung gar nicht entstehen lassen. Durch die Resequenzierung entstehen **neue Freiheitsgrade zur Balancierung**. Dabei kann zwischen software- und hardwarebasierten Verfahren unterschieden werden.

- **Softwarebasierte Verfahren** zur Paket-Resequenzierung benötigen einen nennenswerten Anteil der zur Verfügung stehenden Rechenleistung. Zudem beschränken sich die Verfahren auf eine globale Resequenzierung ohne Berücksichtigung von Flows.
- **Hardwarebasierte Verfahren** sind dagegen kaum bekannt. Zwar existiert das Problem auch in sogenannten *Load Balanced Switches* und wurde dort – wenn auch unter anderen Randbedingungen – auf verschiedene Arten gelöst. Für Netzwerkprozessoren konnte jedoch nur ein Verfahren vorgestellt werden, das die Reihenfolge in Hardware mit Flow-Granularität wiederherstellt. Hierbei sind v.a. eine komplizierte Speicherstruktur und eine ressourcenintensive Realisierung auffällig. Zudem konnte das Verfahren in den gezeigten Simulation zu einer Blockierung des Prozessierungsclusters führen.

3.4 Interrupt Controller für Multiprozessor-Systeme

Bei einem Cluster von Prozessoren (oder auch anderen Verarbeitungseinheiten) ergibt sich automatisch das Problem der Benachrichtigung der einzelnen Prozessoren bei Eintreffen neuer Pakete bzw. Tasks. Grundsätzlich gibt es hierfür zwei unterschiedliche Ansätze:

- **Polling:** Beim Polling liest ein freier Prozessor von sich aus einen Speicherinhalt aus und überprüft, ob eine Aufgabe zum Abarbeiten bereitsteht. Am Beispiel Paketverarbeitung kann dies zum Beispiel ein Eingangspuffer sein. Der Prozessor überprüft also, ob sich im Puffer ein Paket befindet, das er laden und verarbeiten kann. Ist dies nicht der Fall, wird der Vorgang in regelmäßigen Abständen wiederholt. Dieser Abstand ist dabei mit Bedacht zu wählen. Ein zu häufiges, erfolgloses Auslesen erzeugt eine unnötig hohe Buslast. Ein zu großer Abstand führt dazu, dass ein Prozessor zu langsam auf neue Pakete reagiert.
- **Interrupts:** Interruptbasierte Systeme dagegen informieren einen Prozessor aktiv über eine anstehende Aufgabe. Trifft ein Paket ein, so wird der zugehörige

Prozessor mithilfe eines Interrupteingangs darüber informiert. Der Prozessor liest anschließend einen *Interrupt Controller* aus, der ihm die Nummer des auslösenden Interrupts mitteilt. Damit verbunden ist beim Prozessor eine *Interrupt Service Routine*, also eine genaue Funktionsbeschreibung, die beim Auslösen dieser Interrupt-Nummer abzuarbeiten ist. Dabei kann auch eine bestehende Bearbeitung unterbrochen werden, um z.B. eine Aufgabe mit höherer Priorität vorzuziehen. Da der Prozessor die Priorität des Interrupts erst nach dem Auslesen der *Interrupt Controllers* feststellen kann, bedeutet dies eine Unterbrechung der Prozessierung bei jedem Ereignis, d.h. also auch bei einem eigentlich niederprioren Ereignis. Der *Interrupt Controller* wird in der Regel über einen externen Interrupt Eingang über eine anstehende Aufgabe informiert (z.B. ein Interrupt-Eingang pro Paketpuffer).

Zwischen Interrupt-Quelle und Prozessor kann bei einem Multiprozessorsystem grundsätzlich eine *eins-zu-eins* oder eine *eins-zu-n* Zuordnung exisitieren. Exisitieren beispielsweise in einem paketverarbeitenden System mehrere Paketeingangspuffer, so ist im ersten Fall genau ein Prozessor für die Abarbeitung eines konkreten Puffers zuständig. Im zweiten Fall dagegen übernehmen mehrere Prozessoren parallel die Abarbeitung eines Puffers.

IBM hat in [62] die Architektur und Funktionsweise eines *Multiprocessor Interrupt Controllers* (MPIntC) beschrieben. Dieser basiert auf dem *Open Programmable Interrupt Controller (PIC)* Standard. *Open PIC* ist dabei ein von den Firmen AMD und Cyrix definierter Industriestandard für Symmetrische Multiprozessorsysteme.

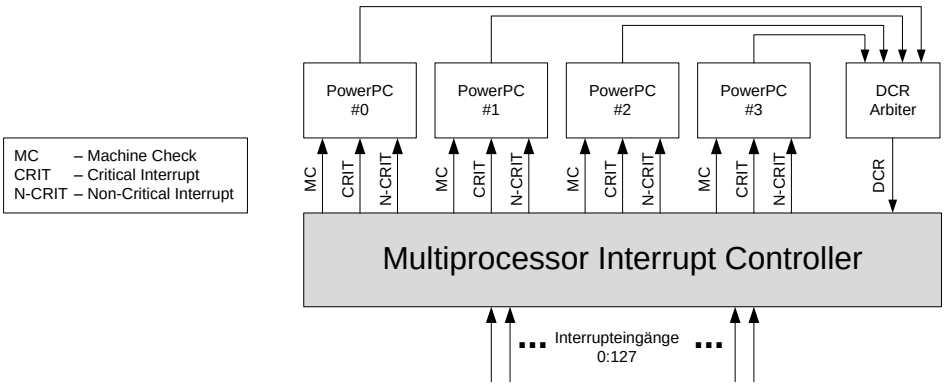

Abbildung 3.15: *Multiprocessor Interrupt Controllers* von IBM mit vier angeschlossenen PowerPC-Prozessoren.

Der MPIntC unterstützt 128 externe Interrupt-Eingänge und bis zu vier Prozessoren. Zusätzlich existieren vier *Inter-Processor Interrupts*, mit deren Hilfe die Prozessoren untereinander Interrupts auslösen können, und vier Timer für periodische Interrupts. Die Interrupt-Eingänge besitzen jeweils eine programmierbare Priorität zwischen 0 und 15. Die Prioritäten können dabei in *Machine Check*, *kritische* und *nichtkritische Interrupts* unterteilt werden. Für jeden dieser Interrupttypen steht dabei ein separater Interrupt-

Eingang zu jedem Prozessor bereit (siehe Abbildung 3.15). Die Prozessoren kommunizieren mit dem *Interrupt Controller* über den *Device Control Register* (DCR) Bus. Dies ist ein bei IBM PowerPCs verwendeter, sehr einfach gehaltener Bus zur Konfiguration der Peripherie. Der Bus hat eine Zugriffslatenz von nur einem Takt und ermöglicht damit einen schnellen Zugriff, ohne der Gefahr von größeren Latenzen auf dem eigentlichen Systembus. Da es sich hier eigentlich um einen *Single-Master* Bus handelt, setzt ein *DCR Arbiter* die potentiell parallelen Zugriffe der PowerPCs in sequentielle Zugriffe um.

Der MPIntC erlaubt *Directed, Multicast* und *Distributed Interrupts*. Bei *Directed Interrupts* wird grundsätzlich nur ein vorher definierter Prozessor benachrichtigt, bei *Multicast* dagegen alle für diesen Interrupt zuvor festgelegten Prozessoren. Die *Multicast*-Funktionalität beschränkt sich dabei auf Interrupts, die durch interne Timer des Controllers ausgelöst wurden. Für *Distributed Interrupts* wählt der MPIntC einen Prozessor aus einer zuvor definierten Gruppe aus. Bei *Distributed Interrupts* wird pro Interrupt-Eingang immer nur maximal ein Interrupt gleichzeitig bearbeitet. Selbst wenn ein Interrupt bereits in Bearbeitung ist und ein weiterer am Eingang folgt, wird bis zur Abbarbeitung des ersten Interrupts gewartet, ehe ein weiterer Prozessor benachrichtigt wird. Die Prozessoren geben nach Abarbeitung eines Interrupts hierfür dem Controller eine Rückmeldung. Auf diese Weise kennt der MPIntC den aktuellen Status aller Prozessoren.

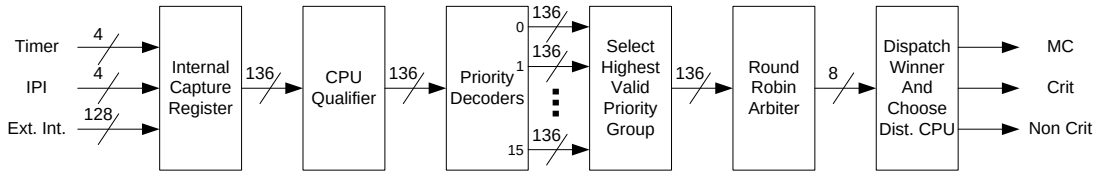

Abbildung 3.16: Vereinfachte Architektur der Verarbeitungspipeline des IBM *Multiprocessor Interrupt Controllers*.

Die Verarbeitungspipeline des MPIntC ist in Abbildung 3.16 gezeigt. Ankommende Interrupts werden im *Interrupt Capture Register* identifiziert und gespeichert. Bereits sich in Bearbeitung befindliche Interruptnummern werden an dieser Stelle ausmaskiert. Der *CPU Qualifier* wählt der Reihe nach jeweils einen Prozessor aus und maskiert den Eingangsvektor entsprechend. Auf diese Weise gelangen nur die für den Prozessor relevanten Interrupts zur nächsten Stufe. Die *Priority Decoder* schlüsseln den Vektor in die 16 Prioritätsklassen auf. Somit ergibt sich pro Klasse ein eigener Vektor mit den jeweils aktiven Interrupts. Die nächste Stufe wählt aus den 16 Vektoren den höchstprioren aus, der einen aktiven Interrupt enthält. Da jeder Interrupt aus einer Prioritätsklasse die gleichen Chancen zu Bearbeitung haben soll, wählt ein Arbiter einen aktiven Interrupt nach dem *Round Robin*-Verfahren aus. Hierfür ist es notwendig, dass für jeden Prozessor und jede Prioritätsklasse die jeweils zuletzt ausgewählte Interruptnummer gespeichert wird.

Pro Prozessor existiert ein eigener *Dispatcher*. Ein *Inservice Register* hält hier fest, welche Prioritätsklasse(n) aktuell beim Prozessor in Bearbeitung ist/sind. Der vom Arbiter ausgewählte Interrupt kann an dieser Stelle verworfen werden, sofern er nicht höher

als der aktuell höchstpriore, bereits in Bearbeitung befindliche Interrupt ist. Sofern der Interrupt akzeptiert wird, wird der Prozessor informiert. Zudem wird das *Interrupt Capture Register* darüber informiert, dass sich der Interrupteingang nun in Bearbeitung befindet. An dieser Stelle erfolgt zudem die Auswahl des bestgeeigneten Prozessors für *Distributed Interrupts*. Die Entscheidung wird pro Taktzyklus getroffen und wird mit dem nächsten Taktzyklus an die entsprechenden Stellen verteilt. Hierfür wird ein Algorithmus eingesetzt, der zunächst nach einem freien Prozessor sucht. Sind alle Prozessoren belegt, so erfolgt die Auswahl anhand anderer Kritierien (Taskpriorität der Prozessoren, Priorität der wartenden Interrupts, Priorität der aktuell in Bearbeitung befindlichen Interrupts). Führt dies immer noch zu keinem Ergebnis, wird eine weitestgehend zufällige Entscheidung getroffen.

Der IBM MPIntC erlaubt also v.a. durch die konfigurierbare Interrupt-Priorität eine hohe Flexibilität. Zudem wird mit den *Distributed Interrupts* eine auslastungsabhängige Verteilung anstehender Interrupts ermöglicht. Zusammenfassend besitzt der MPIntC jedoch eine Reihe von Nachteilen:

- Die konkrete Implementierung ist **limitiert auf vier CPUs**. Damit deckt sie keine höheren *Multi-* oder gar *Many-Core*-Systeme ab.
- Die **Komplexität** der gezeigten Architektur ist verhältnismäßig hoch (vgl. auch Abbildung 3.16). Insbesondere lassen Komponenten wie *CPU Qualifier* oder *Priority Decoder* eine **schlechte Skalierbarkeit** erwarten.
- Bei *Distributed Interrupts* wird **gleichzeitig nur jeweils ein aktiver Interrupt pro Interrupt-Eingang weitergeleitet**. Somit können nicht mehrere Interrupts des gleichen Typs gleichzeitig im Cluster prozessiert werden. Genau dies kann sich jedoch bei der Paketverarbeitung als nachteilig erweisen.

4 FlexPath-Netzwerkprozessor

Das FlexPath-NP-Konzept beruht auf einer NP-Architektur mit zentralem CPU-Cluster. Wie bereits in Kapitel 3.1 erläutert verspricht dieser Ansatz – insbesondere im Vergleich zu einer Verarbeitungspipeline – grundsätzlich ein hohes Maß an Flexibilität (und damit ein breites Einsatzspektrum) und wird daher auch in aktuellen, kommerziellen Produkten häufig gewählt. Im Rahmen des FlexPath-Projekts stand allerdings nicht die Entwicklung einer konkurrenzfähigen Implementierung im Vordergrund. Vielmehr sollte ein Konzept gefunden werden, mit dem sich die Leistung eines CPU-zentrischen NPs ganz allgemein verbessern lässt. Dabei sollen das FlexPath-Konzept – als Ganzes oder wahlweise auch in Teilen – möglichst einfach auf andere NP-Architekturen übertragbar sein.

Konkret werden bei FlexPath die folgenden Ansätze vorgeschlagen:

- **Hardware-Unterstützung:** Die Hardware-Unterstützung soll über das bekannte Maß bei NPs (optimierte Instruktionssätze, Hardwarebeschleuniger) hinausgehen. Damit soll zumindest ein Teil der Pakete vollständig in einem zum Cluster parallelen Hardwarepfad bearbeitet werden.
- **Paketabhängige Pfadwahl:** Im Gegensatz zu herkömmlichen Architekturen sollen ankommende Pakete direkt am Eingang klassifiziert werden und dabei ein für das Paket optimierter Weg durch das System bestimmt werden.
- **Rekonfiguration der Pfadentscheidung:** Die Kriterien der Pfadentscheidung müssen aufgrund der großen Dynamik im Verkehr nicht statisch die beste Lösung sein. Deshalb soll sich das System bei entsprechenden Änderungen anpassen. Dies schließt v.a. die in dieser Arbeit noch näher beschriebene Lastbalancierung innerhalb des CPU-Clusters mit ein.
- **Unterstützung von Quality of Service:** Bei herkömmlichen Ansätzen erfolgt eine Einstufung in verschiedene Service-Klassen erst nach der Prozessierung durch eine CPU. Eine Analyse der Pakete in Hardware erlaubt dagegen bereits eine frühzeitige Priorisierung im Eingangsdatenpfad. Hochpriore Pakete können damit schon frühzeitig identifiziert und bevorzugt behandelt werden.

Das FlexPath-Konzept wurde im Rahmen eines DFG-Schwerpunktprogramms (SPPRR 1148) erarbeitet und vorgestellt. Neben dieser Arbeit beschäftigt sich ebenso die Dissertation von Rainer Ohlendorf [3] mit dem Projekt. Außerdem wurde auf die Ergebnisse und Implementierung des von Daniel Llorente erarbeiteten SmartMem Buffer Managers [63] zurückgegriffen.

Während sich Rainer Ohlendorf insbesondere mit den Hardwarekomponenten des Eingangsdatenpfades, der Umsetzung der Pfadentscheidung und einem Lastbalancierungs-

konzept für zustandsbehafteten Verkehr beschäftigt, können die **Schwerpunkte der vorliegenden Arbeit** folgendermaßen zusammengefasst werden:

- Zur Umsetzung einer effektiven Hardware-Unterstützung und einer vollständigen Bearbeitung einzelner Pakete in Hardware bedarf es einer **Paketmanipulationseinheit im Ausgangsdatenpfad** (*Post-Processor*). Der *Post-Procssor* und dessen Implementierung wird in Kapitel 4.3 näher beschrieben.
- Während der Schwerpunkt bei Rainer Ohlendorf in der Lastbalancierung zustandsbehafteten Verkehrs liegt, beschäftigt sich diese Arbeit mit der **Lastbalancierung zustandslosen Verkehrs** (siehe Kapitel 5.2).
- In Kapitel 5.3 wird zudem eine Einheit zur **Paketverteilung im Multiprocessor-Cluster** (*Packet Distributor*) vorgestellt, die die Pfadentscheidung sowie unterschiedliche Lastbalancierungsstrategien im Cluster effektiv umsetzt.
- Eine dynamische Pfadentscheidung kann zu Paketüberholungen innerhalb eines Paketflows führen. Diese hat, wie schon in Abschnitt 2.3.3 beschrieben, negative Auswirkungen auf die Netzwerkleistung. In Kapitel 6 wird deshalb ein effektives **Verfahren zur Aufrechterhaltung der Paketreihenfolge** vorgestellt.

Der Rest dieses Kapitels gliedert sich wie folgt: Im Folgenden werden zunächst die genannten konzeptionellen Ansätze näher erläutert (Abschnitt 4.1). In Abschnitt 4.2 wird die Architektur des FlexPath-NPs mit den zugehörigen Einzelkomponenten beschrieben. Anschließend wird in Abschnitt 4.3 der *Post-Processor* im Detail vorgestellt, bevor im letzten Abschnitt 4.4 die Ergebnisse erster Simulationen die Vorteile der Hardware-Unterstützung in FlexPath zeigen.

4.1 Konzept

In diesem Abschnitt werden die grundlegenden Ideen des FlexPath-NPs näher erläutert. Abbildung 4.1 veranschaulicht hierzu die wesentlichen Erweiterungen der FlexPath-Architektur (b) im Vergleich zu einem herkömmlichen CPU-Cluster (a).

4.1.1 Hardware-Unterstützung

Hardware-Unterstützung ist wie in Kapitel 3.1 beschrieben, ein gängiges Mittel, um den Durchsatz eines NPs zu steigern. Neben speziellen Hardwarebeschleunigern für z.B. *Address Lookups*, *Policing*, aber auch Kryptoanwendungen ist meist auch der Instruktionssatz der Prozessoren für Netzwerkanwendungen optimiert. Wie am Beispiel der SafeXcel *Inline Security Engine* gesehen (siehe Abschnitt 3.1.1), kann aber auch die vollständige Verarbeitung zumindest bestimmter Applikationen auf Hardware ausgelagert werden.

Der FlexPath-NP-Ansatz basiert auf einem Cluster von Standard-RISC-Prozessoren, die um eine Reihe von Hardwarefunktionen ergänzt werden. Ziel ist es, Standardaufgaben auszulagern, um somit Freiraum im Prozessor-Cluster zu erhalten. Um eine Auslagerung in Hardware zu rechtfertigen, müssen zwei Bedingungen eingehalten werden:

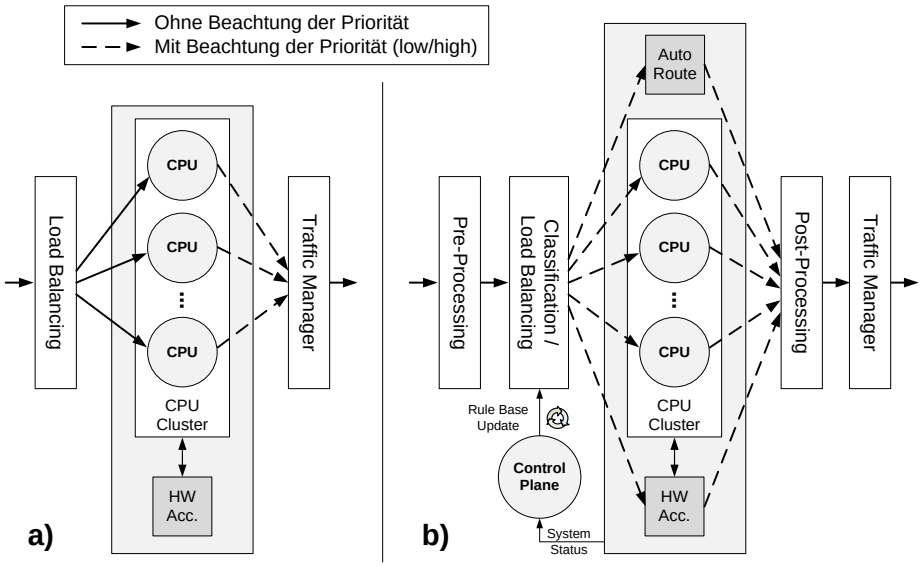

Abbildung 4.1: Gegenüberstellung zwischen herkömmlichen NP mit CPU-Cluster (a) und den grundsätzlichen Erweiterungen im FlexPath-NP (b).

1. Die Funktionen können effizient in Hardware implementiert werden. Es muss dabei eine deutliche Ausführungszeitverkürzung gegenüber einer Softwareimplementierung erreicht werden.
2. Die ausgewählten Aufgaben müssen für einen nennenswerten Anteil der Pakete nutzbar sein, um den Mehraufwand an Hardware zu rechtfertigen.

Typischer Internetverkehr besteht zum überwiegenden Teil (>90%) aus TCP- oder UDP-Datenpaketen (siehe Abschnitt 2.3.3). Die Bearbeitung dieser Pakete schließt auf IP-Ebene in der Regel immer IP-Forwarding mit ein. Je nach Anforderungen im Netz werden neben Forwarding zusätzliche Anwendungen wie Ver-/Entschlüsselung, *Filtering* etc. (siehe hierzu auch Abschnitt 2.4.3, *Wide Area Networks*) ausgeführt. Forwarding ist daher eine Standardaufgabe, die auf einem Großteil der Pakete durchgeführt werden muss. Eine Analyse der notwendigen Schritte (siehe Abschnitt 2.3.2) verspricht eine effiziente (Teil-)Implementierung in Hardware.

Generell werden drei unterschiedliche Stufen für die Hardwareunterstützung vorgeschlagen:

- IP-Pakete, deren Bearbeitung sich auf IP-Forwarding beschränken, sollen komplett in Hardware bearbeitet werden. Dies wird im Folgenden *AutoRoute* (AR) genannt.
- Bei IP-Paketen, die aufgrund weiterer Applikationen von einem Prozessor bearbeitet werden müssen, wird zumindest das IP-Forwarding in Hardware ausgelagert.
- Bei sonstigen Paketen (z.B. Kontrollpaketen oder Ausnahmebehandlungen), sollen Ergebnisse der Hardware-Analyse bestmöglich genutzt werden (z.B. Checksummen-Überprüfung, *Lookup*-Ergebnis, Länge). Zudem soll eine hardwaregestützte Paket-

modifikation die Bearbeitung beschleunigen.

Der Vorteil von *AutoRoute* wurde bereits zu Beginn des FlexPath-NP-Projekts analytisch in [64] abgeschätzt. Dabei wurde ein Prozessor-Cluster mit sechs ASIP-Cores (Taktfrequenz 232 MHZ, CPI = 1,0; Daten basieren auf dem Intel IXP1200 NP) mit einem FlexPath-System mit zwei Standard-RISC-Prozessoren (Taktfrequenz 667 MHz, CPI = 1,4) verglichen. Dabei war das FlexPath-System ab einem AR-Anteil von 31,5% dem ASIP-Ansatz überlegen.

Auch im Falle einer Prozessor-Bearbeitung werden Vorteile aus der Hardwareunterstützung erwartet. Für eine erste Abschätzung wurde ein frei verfügbarer Netzwerkstack, der *Lightweight TCP/IP Stack* (lwIP) [65], näher untersucht (siehe [66]). Tabelle 4.1 vergleicht für IP-Forwarding die originale Anzahl der Instruktionen mit der Anzahl, die bei vorhandener Hardwareunterstützung voraussichtlich notwendig ist. In diesem Fall wird davon ausgegangen, dass kein gültiges *Lookup*-Ergebnis vorhanden ist, entsprechend der Bearbeitung eines Pakets mit ungültiger Route. Dabei kann die Anzahl der Instruktionen voraussichtlich um ca. 22% auf 1.885 gesenkt werden.

Funktion	**Beschreibung**	**Referenz (Zählung)**	**mit HW-Unterst. (Schätzung)**
BMG_intr	Begin of ISR	20	20
BMGif_input	Data structure initialization	362	362
Etharp_input	Update ARP table	535	477
IP_input	RX integrity checks	523	153
IP_forward	Default GW lookup & forwarding	96	86
Etharp_output	ARP query and TX modification	568	470
BMGif_output	Transmit & free data structure	237	237
BMG_intr	Return from functions & ISR	80	80
SUMME		**2.421**	**1.885**

Tabelle 4.1: Anzahl der Assembler Instruktionen für lwIP bei IP-Forwarding mit und ohne Hardwareunterstützung.

Insbesondere für AR-Pakete ist eine Datenmanipulationseinheit zwingend erforderlich. Diese muss die notwendigen Anpassungen am Paket-Header in Hardware durchführen. Die Realisierung dieser Einheit wird in Kapitel 4.3 detailliert beschrieben.

4.1.2 Paketabhängige Pfadwahl

Der letzte Abschnitt verdeutlicht, dass im FlexPath-NP mit *AutoRoute* und dem Prozessorpfad wenigstens zwei verschiedene Wege existieren. Damit wird eine Hardware-Instanz im Eingangsdatenpfad benötigt, die die Pakete klassifizieren und das Paket auf den jeweils optimalen Pfad senden kann. Zudem soll die Pfadentscheidung um weitere Pfade ergänzt werden. Zusammengefasst können folgende Pfade identifiziert werden:

- Pakete können direkt und komplett in **Hardware** bearbeitet werden (*AutoRoute*)
- Pakete müssen von einem **Data Plane-Prozessor** bearbeitet werden. Die Pfadentscheidung umfasst auch die Auswahl des Prozessors (spezieller oder beliebiger Prozessor im Cluster).
- Pakete können direkt zum **Control Plane-Prozessor** gesendet werden (z.B. Routingprotokolle, ARP, ICMP).
- Pakete können vor der Bearbeitung durch einen Prozessor zu einem **Hardwarebeschleuniger** zur Vorverarbeitung gesendet werden (z.B. Entschlüsseln eines Pakets vor der Weiterleitung).
- Pakete können direkt verworfen werden (**Discard**). Dies ist z.B. hilfreich bei Paketen die aufgrund falscher Checksummen oder auch aufgrund bestimmter Filtereinstellungen nicht weiter prozessiert werden sollen.

Die paketabhängige Pfadwahl verringert durch optimierte Wege die Latenz der einzelnen Pakete. Sie entlastet jedoch auch das CPU-Cluster. Bei herkömmlichen Cluster-Architekturen entscheidet nämlich letztlich eine CPU bei Ankunft über den Weg des Pakets. Bei FlexPath hingegen wird diese Aufgabe auf die Hardware ausgelagert und damit die Interrupt-Rate des CPU-Clusters gesenkt.

Die Pfadentscheidung wird aufgrund von extrahierten Paketfeldern, dem Ergebnis der Integritätsprüfungen, sowie dem *Lookup*-Ergebnis gefällt. Für eine schritthaltende Verarbeitung muss die Pfadentscheidung pro Paket in Echtzeit getroffen werden. Die Zeit zwischen zwei Paketen liegt bei einer Gigabit-Ethernet-Verbindung mit minimaler Ethernet-Frame-Größe von 64 Byte (plus Präambel, plus *Interframe Gap*; siehe Abschnitt 2.3.1) lediglich bei 672 ns. Bei einem System mit vier Gigabit-Ports verkürzt sich somit die zur Verfügung stehende Zeit zwischen dem Beginn zweier Pfadentscheidungen auf 168 ns. Um in dieser kurzen Zeit den Pfad zu bestimmen wurde in [67] und [68] das Konzept eines *Path Dispatchers* vorgestellt. Es basiert auf einer konfigurierbaren Regelbasis. Details zum Konzept und zur Implementierung finden sich in [3].

4.1.3 Rekonfiguration der Pfadentscheidung

Internetverkehr hat eine sehr hohe Dynamik. Dies gilt zum einen für die aggregierten Datenraten, die über der Zeit starken Schwankungen ausgesetzt sein können, als auch für die im Verkehr enthaltenen Verbindungen, die ständig auf- und abgebaut werden. Mit einer zur Laufzeit konfigurierbaren Regelbasis wird die Möglichkeit geschaffen zur Laufzeit auf Änderungen im Verkehr zu reagieren und das System entsprechend anzupassen.

Eine Rekonfiguration kann dabei unterschiedliche Auslöser haben, z.B.:

- Am Ende eines IPsec-Tunnels besteht die Möglichkeit, Pakete gezielt zur Entschlüsselung direkt zu einem Hardwarebeschleuniger zu senden. Dies ist aber nur für bereits fertig konfigurierte Tunnel möglich, bei denen Schlüssel etc. bereits bekannt sind. Beim **Aufbau einer Verbindung** kann somit ein entsprechender Eintrag zur Regelbasis hinzugefügt werden. Beim **Verbindungsabbau** kann der

Eintrag wieder gelöscht werden und steht dann für andere Zwecke zur Verfügung.

- Aufgrund sich ändernder Verkehrssituationen kann es zu einer unterschiedlichen Auslastung einzelner Prozessierungselemente kommen. Dem resultierenden Ungleichgewicht kann durch **Lastbalancierungsstrategien** entgegengewirkt werden. Dabei wird die Zuordnung zwischen Paketflows bzw. Flowbündel und Prozessor zur Laufzeit in der Regelbasis angepasst. Wie im Kapitel 5 noch gezeigt wird, eignet sich der FlexPath-NP hervorragend zur Anwendung diverser Strategien.

Die Rekonfiguration der Pfadentscheidung kann jedoch zu Vertauschungen in der originalen Paketreihenfolge (*Packet Reordering*) führen. Die daher notwendige Resequenzierung wird später in Kapitel 6 behandelt.

4.1.4 Unterstützung von Quality of Service

Bei vielen NP-Architekturen findet eine Klassifizierung der Pakete und damit eine Einstufung in unterschiedliche Prioritätsklassen erst im verarbeitenden Prozessor statt (siehe Abbildung 4.2a). Vor dem Prozessor ist die Priorität unbekannt und hochpriore Pakete können nicht gesondert behandelt werden. Die eingangsseitigen Puffer sind aus diesem Grunde klein zu halten. Werden die Puffer zu groß gewählt, wächst in Überlastsituationen die Latenz aller Pakete an. Somit erfahren auch die hochprioren Pakete eine hohe Latenz – dies ist im Sinne von QoS aber zu vermeiden.

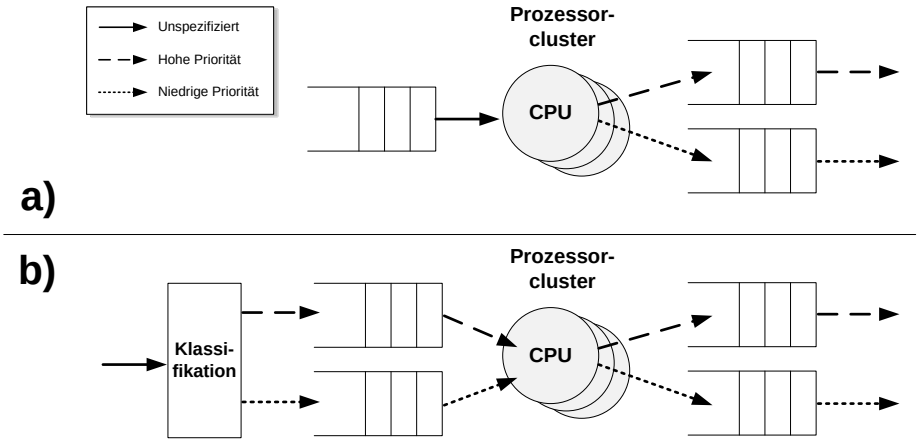

Abbildung 4.2: Klassifikation in hoch- und niederprioren Verkehr bei herkömmlichen NP Architekturen (a) und bei FlexPath (b)

Beim FlexPath-NP (siehe Abbildung 4.2b) dagegen erfolgt bereits eingangsseitig eine Einteilung. Somit können die hochprioren Pakete bevorzugt von den Prozessoren bearbeitet werden. In Überlastsituationen ist es möglich, gezielt niederpriore Pakete zu verwerfen, die hochprioren Pakete aber dagegen weiterzuleiten. Sowohl Latenz als auch Verlustraten der hochprioren Pakete profitieren.

4.2 Beschreibung der Einzelkomponenten und des Paketflusses

Im FlexPath-NP wird das Prozessor-Cluster durch eine Reihe umgebender Hardware im Ein- und Ausgangsdatenpfad ergänzt. Der logische Aufbau des NPs ist in Abbildung 4.3 dargestellt.

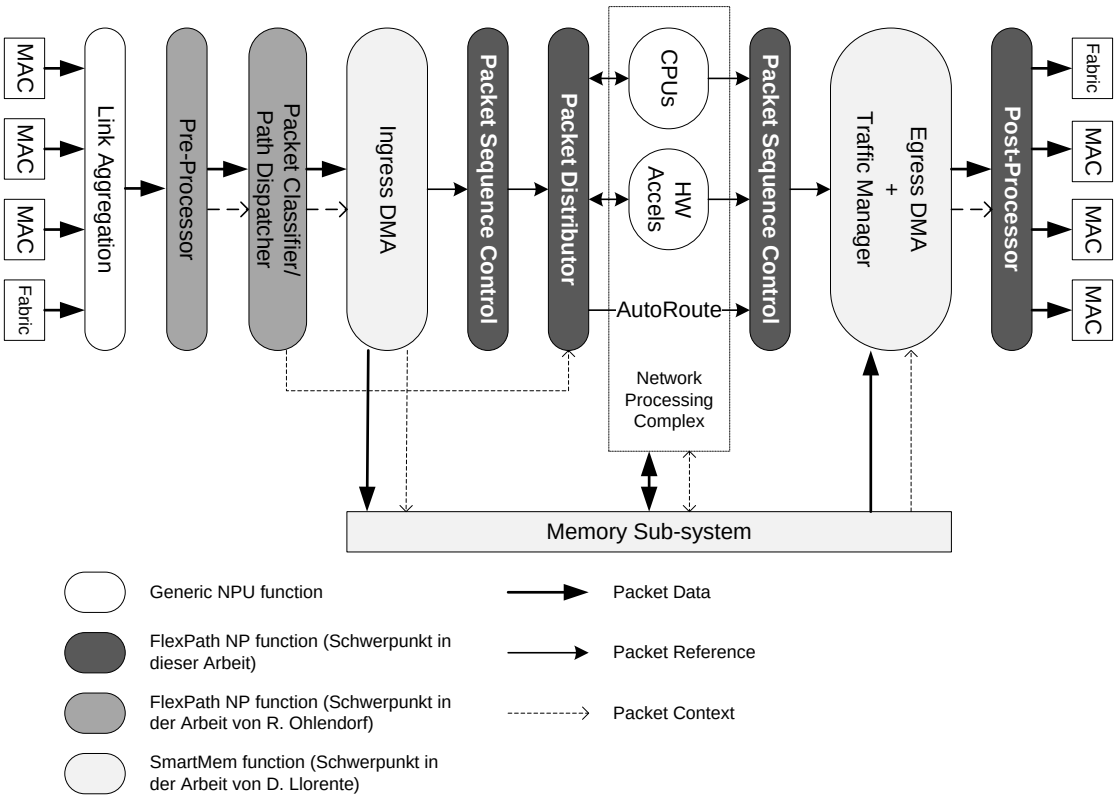

Abbildung 4.3: Konzept des FlexPath-Netzwerkprozessors.

Man erkennt dabei eine Reihe von Komponenten, wie sie auch in anderen auf CPU-Cluster basierenden NPs üblich sind. Dazu zählt insbesondere der zentrale **Network Processing Complex**, wozu neben den eigentlich CPUs auch diverse Hardwarebeschleuniger (z.B. Kryptobeschleuniger) gezählt werden können. Ebenfalls als Standardkomponente darf aus FlexPath-Sicht der *SmartMem Buffer Manager* gezählt werden. Dieser kann in die beiden Einheiten **Ingress DMA Engine** und **Egress DMA Engine** aufgeteilt werden. Die *Ingress DMA Engine* speichert ankommende Pakete automatisch in den zentralen Paketspeicher ab. Sie übernimmt die Verwaltung des Speichers selbst, so dass dies ohne das Zutun eines Prozessors geschehen kann. Die *Ingress DMA Engine* legt für das Paket eine Paket-Referenz (Paket-Deskriptor) an. Der Paket-Deskriptor enthält neben grundlegenden Paketinformationen wie der Paketlänge insbesondere einen Zeiger auf den Speicherort. Im Folgenden wird innerhalb des Systems nur der Paket-Deskriptor als Referenz auf das Paket an die nachfolgenden Instanzen weitergeleitet.

Die folgenden Verabeitungseinheiten können anhand des Paket-Deskriptors auf die eigentlichen Paketdaten und den Paketkontext im Paketspeicher zugreifen. Als Gegenstück empfängt die *Egress DMA Engine* in Zusammenarbeit mit dem **Traffic Manager** ausgehende Paket-Deskriptoren aus dem Verabeitungscluster und versendet die Pakete je nach Linkauslastung. Dazu werden die Paketdaten aus dem Paketspeicher geladen und der entsprechende Speicherbereich anschließend wieder freigegeben.

Neben dieser Standardkomponenten wird der FlexPath-NP um eine ganze Reihe spezialisierter Hardware ergänzt:

Die Klassifikation der Pakete und die Pfadentscheidung finden direkt nach Empfang der Pakete an den Eingängen statt. Ein ankommendes Paket wird dazu zunächst im **Pre-Processor** auf Datenintegrität überprüft. Zudem werden einzelne Paketfelder aus dem Paket extrahiert. Die extrahierten Felder (z.B. Addressen, Ports) und weitere Informationen (z.B. Ergebnis der Checksummen-Überprüfung) werden in einer separaten Datenstruktur – dem Paket-Kontext – abgelegt und der *DMA Engine* zur Speicherung mit dem Paket zusammen übergeben. In der folgenden **Paket-Klassifikation** im **Path Dispatcher** werden die Ergebnisse des *Pre-Processors* ausgewertet. Anhand einer konfigurierbaren Regelbasis wird entschieden, auf welchem Weg das Paket das System passieren soll.

Der **Packet Distributor** hat die Aufgabe, die Pakete auf die korrekten Verarbeitungsinstanzen zu verteilen. Er führt damit in erster Linie die Pfadentscheidung der Paket-Klassifikation aus. Allerdings unterstützt er auch ein *Spraying* über mehrere Instanzen. Das bedeutet, dass eine Pfadentscheidung nicht immer mit einer konkreten Verarbeitungseinheit verbunden sein muss, sondern sich auch auf eine Klasse, bzw. einen Pool mehrerer Instanzen beziehen kann. Der *Packet Distributor* wählt dann aus allen möglichen Einheiten eine verfügbare Einheit zur Verarbeitung aus. Ferner dient er ebenso zur Übermittlung von Paket-Deskriptoren zwischen den Verarbeitungseinheiten. Der *Packet Distributor* wird in Abschnitt 5.3 genauer beschrieben.

Die **Packet Sequence Control**-Einheit ist für die Aufrechterhaltung der Paketreihenfolge im System zuständig. Dazu werden die Pakete vor dem eigentlichen Verarbeitungscluster markiert und die Reihenfolge im Anschluss bei Bedarf wieder hergestellt. Die Vertauschung der Paketreihenfolge kann wie in Abschnitt 2.3.3 bereits erwähnt negative Auswirkungen auf die Gesamtnetzwerkleistung haben. *Packet Reordering* kann in allen Systemen auftreten, bei denen Pakete eines Flows unterschiedliche Verarbeitungseinheiten durchlaufen können. Die genaue Untersuchung der Paket-Resequenzierung ist Gegenstand von Kapitel 6.

Eine weitere wichtige Hardwareeinheit ist der **Post-Processor**. Durch das Bereitstellen diverser Manipulationsfunktionen ermöglicht er für eine bestimmte Klasse von Paketen einen eigenständigen Hardwarepfad am CPU-Cluster vorbei (*AutoRoute*, siehe Abschnitt 4.1.1). Er soll aber auch das Prozessor-Cluster durch Übernahme rechenintensiver Operationen entlasten. Der *Post-Processor* ist im folgenden Abschnitt näher beschrieben.

4.3 Paketmanipulation im Ausgangsdatenpfad

4.3.1 Motivation

In den vergangenen Abschnitten wurde das FlexPath-Konzept vorgestellt und besprochen. Kernelemente des Konzepts sind neben einer flexiblen Pfadwahl v.a. eine sinnvolle Hardwareunterstützung. Der bereits in Abbildung 4.3 erwähnte *Post-Processor* hat dabei zweierlei Funktionen:

- Durch die Hardware-Paketmanipulation wird in erster Linie **AutoRoute** ermöglicht, also eine vollständige Bearbeitung von bestimmten Paketen in Hardware (insbesondere IP-Forwarding).
- Für Prozessor-Pakete sollen – soweit möglich – einzelne Aufgaben auf den *Post-Processor* übertragen und somit das **Prozessor-Cluster entlastet** werden.

Dabei gilt genauso wie in den Überlegungen zur paketabhängigen Pfadwahl (siehe 4.1.2), dass sämtliche Modifikationen schritthaltend durchgeführt werden müssen.

Paketmanipulationen bei AutoRoute / IP-Forwarding

AutoRoute ist wie in Abschnitt 4.1.1 erläutert eine effektive Methode um ein Prozessor-Cluster zu entlasten. Dort wurden bereits die Pakete, die auf IP-Forwarding beschränkt sind, als geeignete *AutoRoute* Pakete identifiziert. Im Abschnitt 2.3.2 (IPv4) wurden ferner die notwendigen Manipulationen aufgeführt. Diese sind:

- Ersetzen von Ziel- und Quell-MAC-Adresse im Ethernet-Header
- Dekrementierung des TTL-Feldes im IP-Header
- Neuberechnung der IP-Header-Checksumme

Damit lässt sich der notwendige Funktionsumfang für *AutoRoute* relativ eindeutig definieren. Diese Paketmanipulationen können aber natürlich auch bei allen anderen Forwarding-Paketen, die aus diversen Gründen von einem Prozessor verarbeitet werden müssen, genutzt werden.

Einfügen / Löschen von Paketheadern

Ein weiterer interessanter Einsatzzweck des *Post-Processors* lässt sich im Umfeld von IPsec identifizieren. Bei IPsec ergibt sich das Problem, dass am Tunnelanfang, bzw. Tunnelende größere Mengen an Daten (IP-/ESP-/AH-Header etc.) in das Paket eingefügt bzw. aus diesem gelöscht werden müssen.

Sofern das Paket zusammenhängend im Speicher abgelegt ist (vgl. Abbildung 4.4), bedeutet das Einfügen des ESP-Headers ein Verschieben des kompletten Pakets mit Ausnahme des Ethernet-Headers, der konstant im Speicher bleibt. Hierzu muss ein Prozessor die Daten von hinten beginnend auslesen und sukzessive an die neue Speicheradresse kopieren (max. 1.500 Byte). Dies ist zudem nur problemlos möglich, sofern von ausrei-

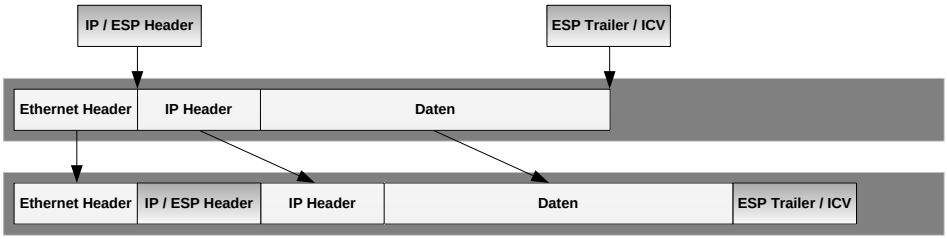

Abbildung 4.4: Einfügen von IP- und ESP-Header und Trailer bei IPsec bei fester Paketspeichergröße.

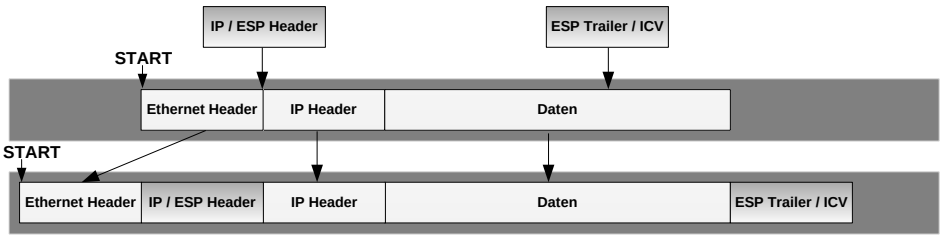

Abbildung 4.5: Einfügen von IP- und ESP-Header und Trailer bei IPsec unter Verwendung der Linux *Socket Buffer* Speicherstruktur.

chend großen Paketpuffern ausgegangen wird. Andernfalls müssen evtl. noch zusätzliche Speichersegmente bzw. größere Paketspeicher angefordert werden, um die zusätzliche Datenmenge aufnehmen zu können.

Eine potentielle Alternative stellt die Verwendung einer geeigneten *Packet Buffer*-Struktur (siehe z.B. den Linux *Socket Buffer (SKB)* [69], Abbildung 4.5) dar. Hierbei wird das Paket beim Empfang grundsätzlich nicht am Anfang der Paket-Speicherzelle abgelegt, sondern ein zuvor definierter Offset eingehalten. Dieser muss groß genug gewählt werden, um denkbare einzufügende Header aufzunehmen. Ein Zeiger in der Paketspeicherstruktur verweist auf den eigentlichen Beginn der Paketdaten. Wird ein zusätzlicher Header eingefügt, kann somit der Großteil des Pakets an der originalen Stelle im Speicher belassen werden. Lediglich der Anfang – in diesem Fall der Ethernet-Header (14 Bytes) – muss umkopiert werden. Ebenso muss am Ende des Pakets genügend Platz für den Trailer vorhanden sein. Der entscheidende Nachteil bei dieser Variante ist der ineffiziente Umgang mit Speicherplatz. Bereits bei der Unterstützung von IPsec muss der zusätzliche Speicherplatz pro Paket bereitgehalten werden – immerhin über 300 Byte z.B. bei IPsec mit ESP und Tunnel-Header. Dies muss zudem unabhängig von der tatsächlichen Nutzung erfolgen. Schließlich stellt sich erst bei der späteren Prozessierung heraus, ob ein Paket tatsächlich zu verschlüsseln ist.

Die gleichen Überlegungen lassen sich (mit umgekehrten Vorzeichen) schließlich auch auf das Löschen von Paketdaten übertragen. Aus diesem Grunde soll mit dem *Post-Processor* die Problematik des Einfügens/Löschen von Paketdaten in den Ausgangsdatenpfad verschoben werden. Auf diese Weise wird das Problem der Re-Allokation von

zusätzlichem Speicher durch die Speicherverwaltung bzw. übermäßige Kopiervorgänge umgangen.

4.3.2 Konzept: Datenmanipulation am Paketdatenstrom

Ein wichtiges Kriterium bei der Manipulation der Paketdaten ist die schritthaltende Verarbeitung im Ausgangsdatenpfad. Mit *AutoRoute* wird ein Hardwareverarbeitungspfad bereitgestellt, der (von der Speicherschnittstelle abgesehen) einer durchgehenden Pipeline entspricht. Damit ist der Datendurchsatz direkt abhängig von der Datenwortbreite und der Frequenz der Schaltung. Um dieses Zeitkriterium auch im Ausgangsdatenpfad aufrecht erhalten zu können, muss ein Datendurchsatz vom im Schnitt einem Datenwort pro Takt angepeilt werden.

In Abschnitt 3.1.2 wurde im Rahmen des PRO3 Projekts eine *Field Modification*-Einheit vorgestellt. Diese besteht aus einem angepassten RISC-Prozessor, der mit 32 Bit und 200 MHz einen maximalen Durchsatz von 6,4 GBit/s erreichen kann. Ein RISC Prozessor ist dabei durch den Kontrollfluss bestimmt. Das bedeutet, dass der Prozessor ein Programm ausführt, bei dem sequentiell ein Befehl nach dem anderen abgearbeitet wird. Die jeweils benötigten Daten müssen zunächst geladen und anschließend wieder gespeichert werden.

Dieser Ansatz kann je nach Anwendungsfall hilfreich sein. Bei der Paketverarbeitung besitzt er jedoch zwei entscheidende Nachteile:

- Nachdem nicht von vornherein festgelegt ist, wie viele Instruktionen pro Datenwort ausgeführt werden, lässt sich der Durchsatz dieses Systems nicht verlässlich festlegen. So können mehrere Instruktionen durchaus auf das gleiche Datenwort zugreifen. Jede Instruktion wird sequentiell für sich abgearbeitet. Zudem können einzelne Instruktionen auch mehrere Taktzyklen benötigen. Der erreichbare Durchsatz beim PRO3 liegt damit in Abhängigkeit der Manipulationen unter den theoretischen 6,4 GBit/s.
- Ein RISC-Prozessor erfordert einen wahlfreien Zugriff auf die Daten. Dabei ist grundsätzlich vor Ausführung der Instruktion nicht bekannt, welches Datenwort als nächstes benötigt wird. Ein Paket muss demnach zunächst in einem Zwischenspeicher für einen wahlfreien Zugriff vorgehalten werden. Neben dem zusätzlichen Speicherbedarf erhöht das zudem die Latenz, da das Paket zunächst komplett vor der Verarbeitung abgespeichert werden muss und erst nach der Verarbeitung weitergesendet wird.

Beim FlexPath-NP werden die Pakete nacheinander aus dem Speicher gelesen und auf dem Ausgangsdatenpfad gesendet. Durch die kurze Aufeinanderfolge von Paketen ergibt sich dabei ein quasi-kontinuierlicher Paketdatenstrom. Dieser Paketdatenstrom soll nun im *Post-Processor* [70] bearbeitet werden. Würde hier ein RISC-ähnlicher Ansatz gewählt werden, würde dieser Paketdatenstrom (wie gesehen) unterbrochen: die Pakete müssten erneut einzeln in einen weiteren Zwischenspeicher gespeichert werden. Deshalb wird hier ein anderer Ansatz gewählt: die Paketdaten sollen beim Durchlaufen des *Post-*

Processors on-the-fly manipuliert werden (siehe Abbildung 4.6). Der *Post-Processor* wird dabei als Verarbeitungspipeline realisiert. Die einzelnen Paketdatenwörter werden beim Durchlaufen der Pipeline modifiziert. Sofern die Pipeline ohne Pausezyklen auskommt, kann der volle Pipelinedurchsatz (Datenwortbreite · Frequenz) erreicht werden.

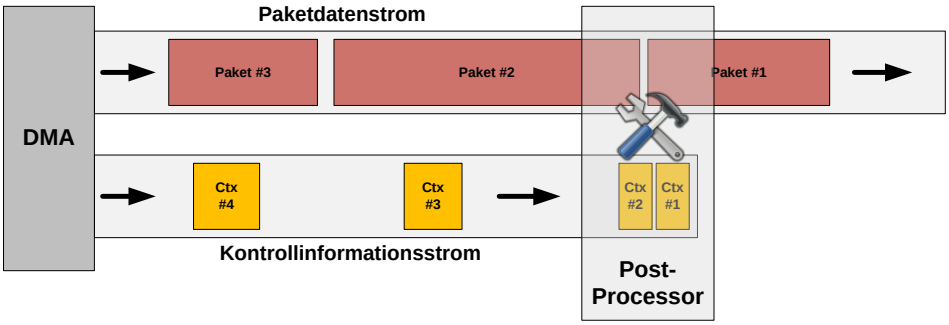

Abbildung 4.6: Paketdatenstrom und *on-the-fly* Verarbeitung im *Post-Processor*.

Neben den Paketdaten werden zur Bearbeitung noch die jeweiligen Instruktionen benötigt. Der *Post-Processor* soll und muss dabei keine eigene weitreichende Intelligenz besitzen. Er hat nicht die Aufgabe, ein Paket zu analysieren und basierend auf den Ergebnissen selbst Modifikationen zu veranlassen. Die Paketanalyse und Verarbeitung findet vielmehr bereits im Eingangsdatenpfad (*AutoRoute*) bzw. im Prozessor-Cluster statt. Basierend auf diesen Ergebnissen sind lediglich entsprechende Modifikationen durchzuführen. Diese sind dem *Post-Processor* in geeigneter Form mitzuteilen. Zu diesem Zweck existiert parallel zum Paketdatenstrom ein weiterer Datenstrom, der pro Paket ein assemblerartiges Programm enthält (siehe Abbildung 4.6). Das Programm kann dabei nicht nur auszuführende Instruktionen enthalten, sondern wird – in Abhängigkeit der Instruktionen – um notwendige Daten (z.B. einzufügende Header) ergänzt. Diese Datenstruktur wird als Ausgangs-Kontext – im Rahmen von FlexPath auch *Context Information Output* (CIO) – bezeichnet. Dieser CIO eilt im Idealfall dem Paket ein wenig voraus und trifft parallel zum vorangehenden Paket ein. Nur so kann eine notwendige Vorprozessierung und Analyse des CIO im *Post-Processor* erfolgen, so dass ein unterbrechungsfreier Datenstrom ermöglicht wird. Um eine eindeutige Zuordnung zwischen Paket und CIO sicherzustellen wird dabei ein CIO pro Paket gefordert. Dieser besteht im einfachsten Fall aus einem *NOP* (*No Operation*) – d.h. es wird keine Modifikation ausgeführt.

Instruktionssatz und Ausgangskontext

Basierend auf den in Abschnitt 4.3.1 geforderten Paketmanipulationen wird der folgende, eingeschränkte Instruktionssatz gewählt. Hierbei wird versucht, Instruktionen möglichst generisch zu halten. Nur so ist ein großes Einsatzspektrum möglich. Im Einzelnen werden folgende Instruktionen vorgehalten:

- **NOP**: *No Operation*. Führt keine Funktion aus.

- **Einfügen**: Hiermit kann eine beliebige Anzahl an Bytes in den Paketdatenstrom eingefügt werden. Der Instruktion wird die Startbyte-Adresse und die Anzahl der einzufügenden Bytes mit übergeben. Die Funktion zielt v.a. auf das Einfügen von zusätzlichen Paketheadern/-trailern in ein existierendes Paket. Ferner kann es auch dazu benutzt werden, ein komplett neues Paket zu generieren.
- **Löschen**: Hiermit kann eine beliebige Anzahl an Bytes aus dem Paketdatenstrom gelöscht werden. Startadresse und Byteanzahl werden definiert. Die Funktion dient v.a. zum Löschen nicht mehr benötigter Header bzw. Trailer.
- **Ersetzen**: Hiermit kann eine beliebige Anzahl an Bytes aus dem Paketdatenstrom durch neue Daten ersetzt werden. Startadresse und Byteanzahl werden definiert. Die Funktion erlaubt z.B. das Ersetzen von Adressen im Paket.
- **Dekrement**: Ermöglicht das Dekrementieren einzelner Bytes im Paket. Wird beim Forwarding für das TTL-Dekrement eingesetzt.
- **IPv4-Checksumme**: Aufgrund der sehr speziellen Berechnung der IPv4-Checksumme wurde für diesen Befehl kein generischer Ansatz gewählt. Lediglich die Startadresse des IP-Headers lässt sich per Parameter verändern und somit unterschiedliche *Layer 1/2*-Protokolle unterstützen.

4.3.3 Architektur: Modulare Verarbeitungspipeline

Der *Post-Processor* ist als modulare Verarbeitungspipeline aus unabhängigen Verarbeitungsstufen aufgebaut (siehe Abbildung 4.7). Jede Stufe übernimmt dabei eine den Instruktionen entsprechende Datenmanipulationsfunktion (z.B. Einfügen, Löschen, Ersetzen). Die Paketdatenwörter durchlaufen die Verarbeitungspipeline Stufe für Stufe und werden bei Bedarf verändert. Dabei passiert grundsätzlich jedes Datenwort jede Stufe – unabhängig von einer möglichen Manipulation. Jede Manipulationseinheit kann aus einer oder mehreren Pipelinestufen bestehen. Die Architektur der einzelnen Module teilt sich auf in einen Daten- und einen Kontrollpfad. Der Kontrollpfad verarbeitet den CIO und steuert die eigentliche Datenmanipulation im Datenpfad. Der CIO durchläuft unabhängig von den Paketdaten die einzelnen Kontrolleinheiten. Die für die jeweilige Stufe benötigten Instruktionen bzw. Daten werden zwischengespeichert. Der CIO wird danach an die folgenden Verarbeitungsstufen weitergeleitet. Durch den modularen Aufbau ergibt sich eine sehr flexible Architektur. Weitere Verarbeitungsstufen können bei Bedarf implementiert und sehr leicht in den *Post-Processor* integriert werden. Nicht benötigte Stufen, z.B. das Löschen/Einfügen von Daten in einer reinen Forwarding-Umgebung, können aus der Architektur entfernt werden.

Die Anordnung der einzelnen Verarbeitungsstufen muss entsprechend der chronologischen Abarbeitung angepasst werden. Auf ein beliebiges Datenwort kann letztendlich nur zuerst die Operation der ersten Stufe, gefolgt von der Operation der zweiten Stufe etc. ausgeführt werden. Die umgekehrte Reihenfolge ist nicht möglich. Dies stellt eine Einschränkung gegenüber einem RISC-Ansatz dar, bei dem die Aufeinanderfolge der Operationen grundsätzlich nicht beschränkt ist. Aufgrund der sehr regulären Abfolge in

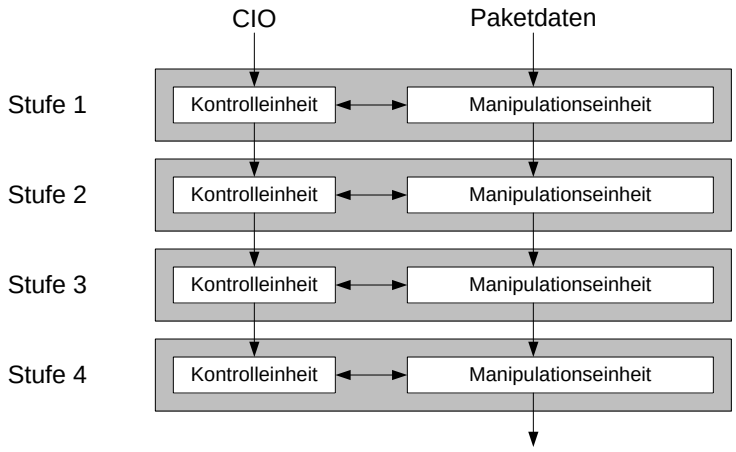

Abbildung 4.7: Modulare Architektur des *Post-Processors*.

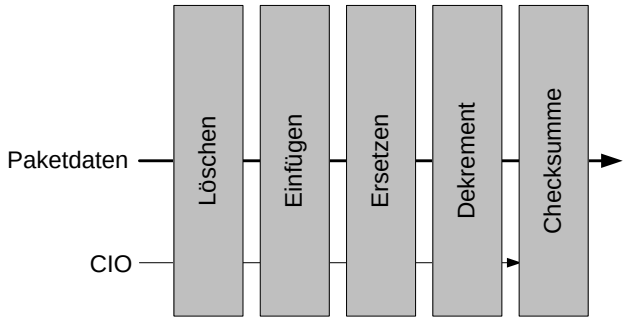

Abbildung 4.8: Anordnung der Verarbeitungseinheiten im *Post-Processor*.

der Paketverarbeitung stellt dies allerdings praktisch keinen gravierenden Nachteil dar.

Für die im Rahmen der Arbeit genutzte Kombination ergibt sich folgende Anordnung (siehe Abbildung 4.8): Die erste Verarbeitungseinheit übernimmt das Löschen von Daten, gefolgt von der *Einfügen*- und *Ersetzen*-Einheit. Die Reihenfolge dieser Einheiten ist bei dem verwendeten Anwendungsszenario (IP-Forwarding und IPsec) grundsätzlich austauschbar. Es mag in manchen Anwendungsfällen allerdings hilfreich sein, Daten aus einem neu eingefügten Header nachträglich verändern zu können. Denkbar sind Szenarien, bei dem z.B. ein *Lookup*-Ergebnis aus einer Hardwareeinheit noch in den neuen Header eingefügt werden muss. Es folgt die *Dekrement*-Einheit, die somit ein TTL-Dekrement auch auf einen eingefügten Header erlaubt. Die *Checksummen*-Einheit wird als letzte Einheit angeordnet, da sie alle vorherigen Änderungen im IP-Header berücksichtigen muss.

Datenpfad-Architektur

Um die Datenmanipulation schritthaltend durchführen zu können, muss in jedem Takt ein Paketdatenwort mit 32 Bit verarbeitet werden können. Dieses Vorgehen lässt sich an der *Dekrement*-Einheit veranschaulichen, welche funktional die einfachste Einheit darstellt (siehe Abbildung 4.9a). Mit jedem Takt wird neben dem Paketdatenwort ein Vektor mit Flags angelegt, der der Verarbeitungseinheit signalisiert, ob eine Manipulation durchgeführt werden soll. Die Granularität ist dabei auf einzelne Bytes beschränkt. Die Bereitstellung der korrekten Flags ist Aufgabe der jeweiligen Kontrolleinheit (siehe Kontrollpfad-Architektur im übernächsten Abschnitt). Im dargestellten Beispiel signalisiert der Flagvektor „0010“, dass eine Dekrementierung des dritten Datenbytes durchgeführt werden muss. Diese wird in der Manipulationseinheit durchgeführt und das Paketdatenwort schließlich im Ausgangsregister gespeichert. Beim Ersetzen von Paketdaten wird dem Datenpfad zusätzlich ein vom Kontrollpfad aufbereiteter Datenvektor zur Verfügung gestellt (siehe Abbildung 4.9b). Die Daten sind dabei bereits innerhalb des Worts korrekt ausgerichtet (siehe auch Kontrollpfad-Architektur).

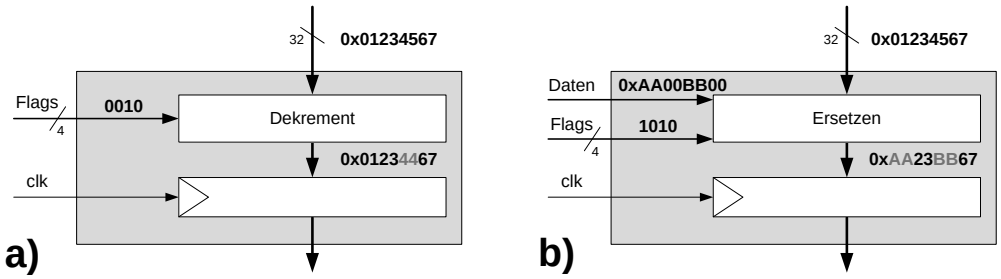

Abbildung 4.9: Datenpfad bei der Datenpfadeinheit für Dekrement (a) und zum Ersetzen von Daten (b).

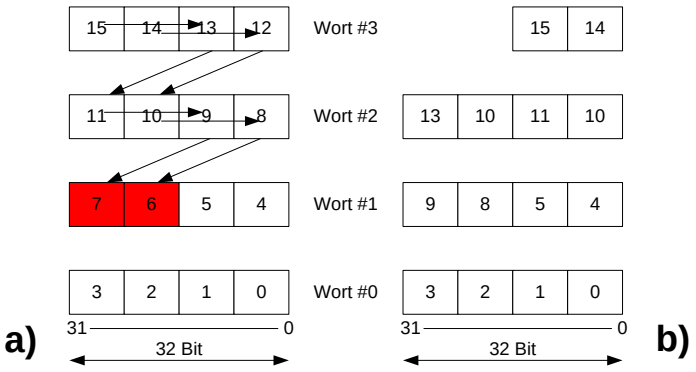

Abbildung 4.10: Paketaufbau beim Löschen einzelner Bytes (a) vor und (b) nach der eigentlichen Datenmanipulation.

Sowohl Dekrementierung, als auch das Ersetzen von Paketdaten sind aus Daten-

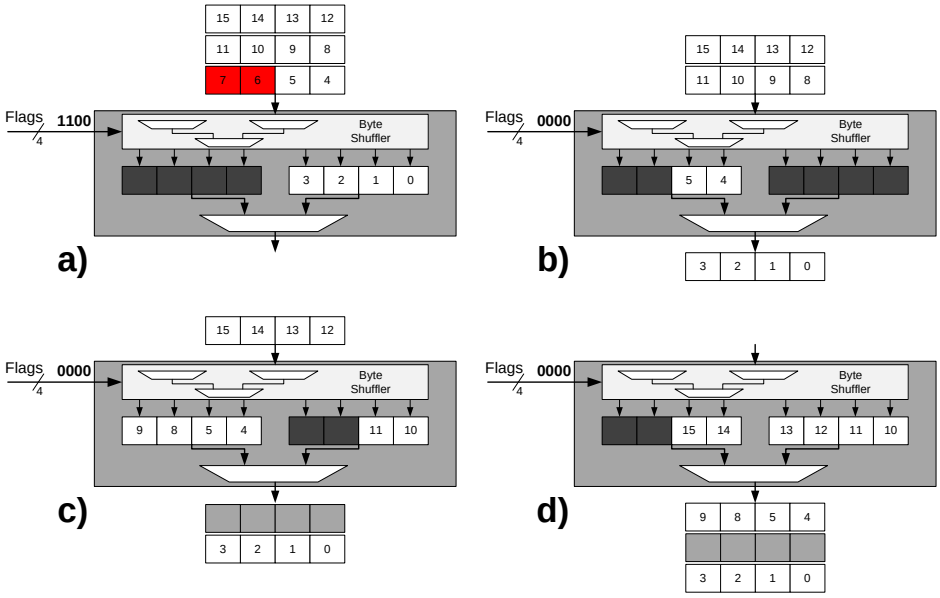

Abbildung 4.11: Schrittweises Vorgehen beim Löschen von Bytes aus dem Paketdatenfluss.

flusssicht sehr einfach zu realisieren. Beim Einfügen oder Löschen dagegen ergibt sich das Problem, dass die originale Anordnung der Bytes in Datenwort nicht mehr zwangsweise eingehalten wird. Zudem ändert sich die Größe des Pakets. In Abbildung 4.10a ist veranschaulicht, wie zwei Bytes (6 und 7) aus dem Paket gelöscht werden sollen. Nachdem Lücken innerhalb eines Wortes nicht erlaubt sind, müssen alle folgenden Bytes im Paket nachrücken. Während das Byte 8 z.B. im 32 Bit-Vektor [31-0] zuvor an der Bitposition [7-0] in Datenwort #2 lag, rückt es anschließend auf Bitposition [23-16] im Datenwort #1 (Abbildung 4.10b).

Kernelement der *Löschen*-Einheit ist neben einem 64-Bit Hilfsregister eine *Byte Shuffle*-Einheit (siehe Abbildung 4.11). Diese ermöglicht es, jedes Eingangsbyte an jede der acht möglichen Stellen im Register zu schreiben. Zur Berechnung der jeweils korrekten Byteposition dienen neben dem aktuellen Zustand des Registers v.a. die eingehenden Flags. Das Register wird immer wechselseitig beschrieben und kann damit als eine Art Ringpuffer gesehen werden (a). Sobald ein vollständiges Datenwort in einer der beiden Registerhälften anliegt, wird dieses am Ausgang ausgegeben. Sofern Daten aus dem anstehenden Wort gelöscht werden sollen, wird das Register nur teilweise beschrieben (b). Die nachfolgenden Wörter verteilen sich dadurch auf beide Register. Das Löschen von Bytes unterbricht den kontinuierlichen Datenstrom durch wortgröße Lücken (c+d). Diese werden durch ein begleitendes *word valid*-Signal den nachfolgenden Modulen mitgeteilt. Die Lücken werden später am Ausgangspuffer des MACs wieder geschlossen. Dieser empfängt das Paket grundsätzlich vor Sendebeginn vollständig.

Ein ähnliches Vorgehen wird auch beim Einfügen von Paketdaten gewählt. Die Kon-

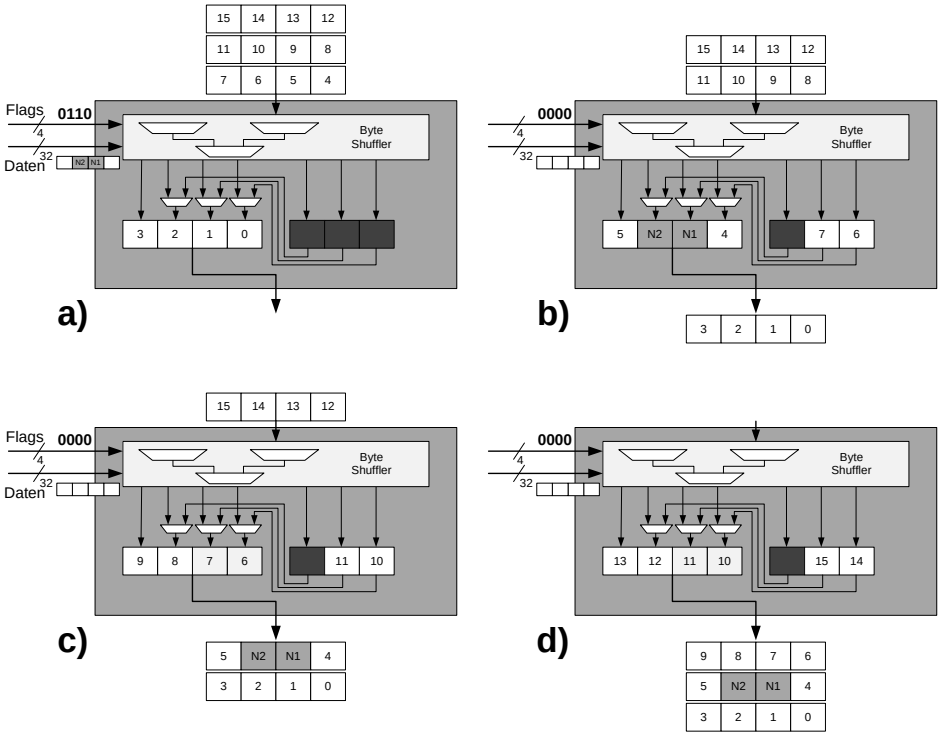

Abbildung 4.12: Schrittweises Vorgehen beim Einfügen von Bytes in den Paketdatenfluss.

trolleinheit stellt die korrekten Flags und Daten bereit. Basierend auf den Flags und den Zuständen der internen Register berechnet auch hier ein *Byte Shuffler* die korrekte Position der einzelnen Bytes im nächsten Takt (siehe Abbildung 4.12a). Im Beispiel sollen zwei Bytes eingefügt werden. Diese verdrängen die Bytes 6 und 7 aus ihrem ursprünglichen Datenwort. Aus diesem Grund müssen diese beiden Bytes in einen zusätzlichen temporären Register zwischengespeichert werden (b). Sie werden am Anfang des nächsten Datenwortes eingefügt (c), dabei werden wiederum die hinteren Bytes des nachfolgenden Datenworts in den Zwischenspeicher verdrängt. Sofern die Anzahl der eingefügten Bytes die Anzahl vier überschreitet (unabhängig ob auf einmal oder gestückelt), muss der Eingang um einen Takt angehalten werden. Damit wird Platz für das nun vollständige neue Paketdatenwort im Datenstrom geschaffen. Das gleiche gilt am Ende des Pakets auch bei weniger als vier Bytes, sofern durch die eingefügten Bytes die Anzahl der Paketdatenworte vergrößert wird.

Die Architektur der *Checksummen*-Einheit unterscheidet sich von den restlichen Einheiten. Es handelt sich hier um eine direkte Hardware-Implementierung der IPv4-Checksummen-Berechnung (vgl. auch Abschnitt 2.3.2). Die zugehörige Kontrolleinheit übergibt lediglich ein Startsignal, das den Beginn des IP-Headers signalisiert. Daraufhin berechnet die Einheit die entsprechende Checksumme und fügt Sie in das korrekte Feld im IP-Header ein.

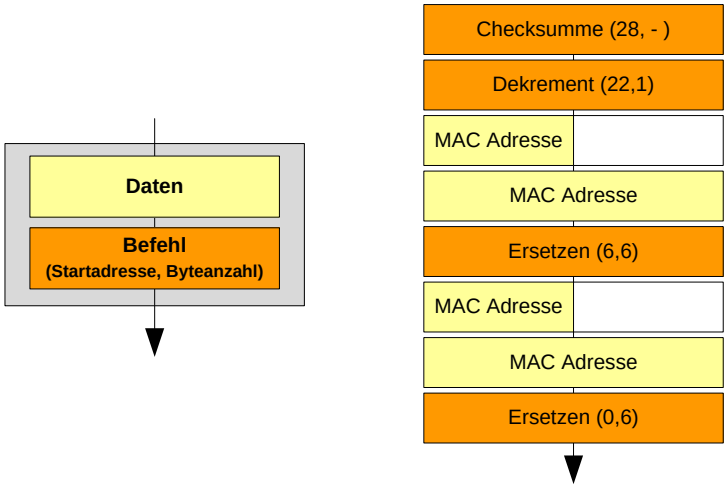

Abbildung 4.13: CIO eines *AutoRoute*-Pakets.

Struktur des CIOs

Die Kontrolleinheit muss den eingehenden CIO zunächst analysieren und aufbereiten. Um dies zu gewährleisten, muss der CIO einem bestimmten Muster (vgl. Abbildung 4.13 links) folgen. Kernelemente des CIOs sind einzelne Befehle. Jeder Befehl enthält neben der Manipulationsart eine Startadresse (in Bytes) innerhalb des Pakets und die Anzahl der konsekutiven Paket-Bytes, auf die der Befehl ausgeführt werden soll. Jeder Befehl ist in einem eigenen 32 Bit-Wort zusammengefasst. Sofern Daten benötigt werden (z.B. beim Einfügen oder Ersetzen), so folgen diese Daten dem Befehl direkt in der benötigten Anzahl an Wörtern. Als Konvention gilt dabei, dass der CIO bereits in der aufsteigenden Startbytereihenfolge der einzelnen Befehle sortiert sein muss. Dies erleichtert später die Analyse des CIOs in der Kontrolleinheit (siehe nächster Abschnitt) und erspart eine komplexe Hardwaresortierung der einzelnen Befehle.

Ein Standard-CIO für AR-Pakete ist in Abbildung 4.13 rechts zu sehen. Der erste Befehl ersetzt die Ziel-MAC-Adresse im Paket. Da hierbei sechs Byte an Daten mit übergeben werden, folgen zwei Wörter an Daten. Es schließt sich das Ersetzen der Quell-MAC-Adresse, der TTL-Dekrement und die Checksummen-Berechnung an. Prinzipiell könnte dem *Post-Processor* das Ersetzen der Ziel- und Quell-Adresse auch in einem Befehl übergeben werden. Es würden sich dann drei Datenwörter (2x6 Byte = 12 Byte) anschließen. Damit wäre der CIO tatsächlich um zwei Datenwörter kürzer. Wenngleich beide Varianten erlaubt sind, ist die hier gezeigte Struktur letztlich durch die eingangsseitige Erzeugung des CIO im Falle von *AutoRoute* bedingt (vgl. Abschnitt 7.2.1).

Kontrollpfad-Architektur

Jede Verarbeitungsstufe ist mit einer Kontrolleinheit ausgestattet, die die jeweilige Datenmanipulationseinheit mit den notwendigen Flags und – beim Einfügen und Ersetzen –

mit den notwendigen Daten versorgt. Es handelt sich dabei (mit Ausnahme der *Checksummen*-Einheit) für alle Verarbeitungsstufen um die identische Kontrolleinheit, die für jede Verarbeitungseinheit neu instantiiert und ggf. unterschiedlich konfiguriert wird. Die Kontrolleinheit untergliedert sich in unterschiedliche Submodule (siehe Abbildung 4.14).

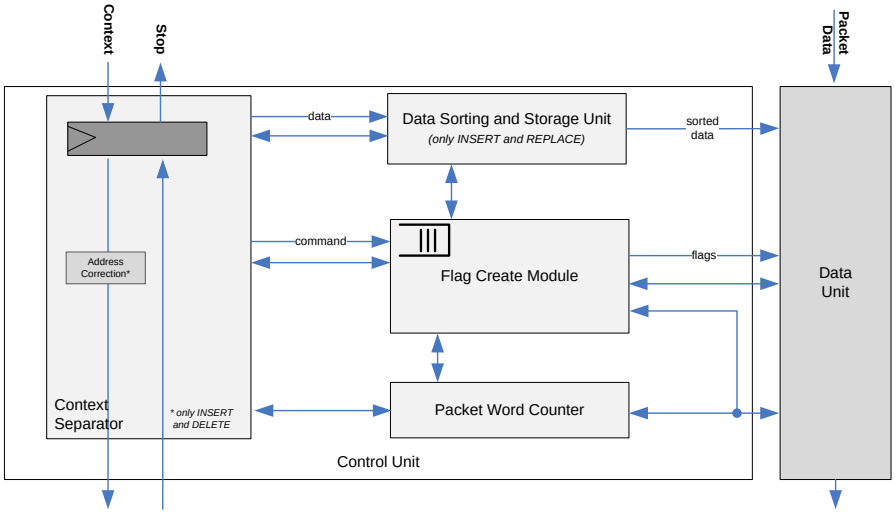

Abbildung 4.14: Architektur der Kontrolleinheiten der einzelnen Manipulationseinheiten im *Post-Processor*.

Der *Context Separator* analysiert den kompletten CIO und filtert die für diese Stufe relevanten Daten und Befehle aus. Um Befehle von Daten unterscheiden zu können, muss jede Kontrolleinheit alle Befehle mit Daten erkennen können. Der CIO wird anschließend direkt an die nachfolgenden Module weitergeleitet. Dabei werden die Startadressen der Befehle in der *Löschen*- und *Einfügen*-Einheit ggf. korrigiert. Die Startadressen im CIO beziehen sich grundsätzlich immer auf die Adressen des originalen, also noch nicht veränderten Pakets. Zur Veranschaulichung soll ein Beispiel-CIO dienen, der ein Einfügen von vier Bytes an Adresse 0 und einem Dekrement an Adresse 15 veranlasst. Die *Dekrement*-Einheit befindet sich nach der *Einfügen*-Einheit. Nachdem nun vier Bytes eingefügt wurden, befindet sich das originale Byte 15 nun aber an Adresse 19. Nachdem der CIO zunächst die *Einfügen*-Einheit durchläuft wird also bereits hier die Adresse angepasst, die *Dekrement*-Einheit erhält einen Befehl mit der für sie relevanten Startadresse 19. Gleiches gilt umgekehrt beim Löschen von Daten. Auf diese Weise müssen beim Erstellen des CIOs Adressveränderungen aufgrund von Einfügen- bzw. Löschen-Befehlen nicht berücksichtigt werden.

Kontext-Daten werden der *Data Sorting and Storage Unit* übergeben. Diese Einheit wird nur bei der *Einfügen*- und *Ersetzen*-Einheit instantiiert. Die Einheit enthält ein FIFO, das die Daten bis zum Gebrauch zwischenspeichert. Zuvor werden die Daten jedoch bereits für die spätere Verarbeitung korrekt ausgerichtet. Im Beispiel in Abbildung 4.15a startet der Befehl mit Startbyteadresse 6 nicht mit dem Paketdatenwortbeginn, sondern um zwei Bytes versetzt. Entsprechend muss das Datenwort ausgerichtet in zwei

Wörtern im FIFO gespeichert werden. Dabei kann es u.U. auch erforderlich sein, Daten aus zwei unterschiedlichen Befehlen vor dem Speichern zu einem Wort zu kombinieren (siehe Beispiel in Abbildung 4.15b).

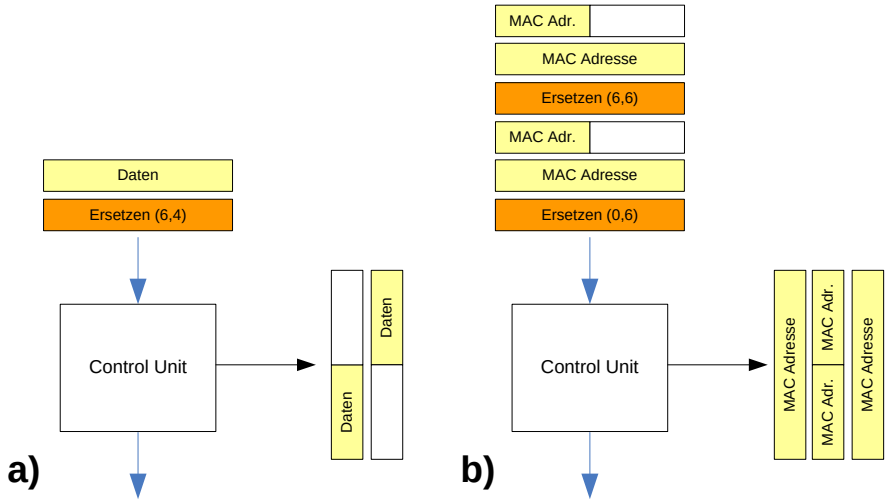

Abbildung 4.15: Datensortierung in der Kontrolleinheit bei unterschiedlichen Eingangskombinationen.

Kontext-Befehle werden dem *Flag Create Module* übergeben. Dieses erzeugt anhand der gespeicherten Befehle und in Abhängigkeit des an der *Data Unit* anliegenden Paketdatenworts die korrekten Flags. Außerdem wird der *Data Sorting and Storage Unit* mitgeteilt, wann das nächste Kontext-Datenwort dem Datenpfad zur Verfügung gestellt werden muss.

Kontroll- und Datenpipeline arbeiten parallel weitestgehend unabhängig voneinander. Die Kontrollpipeline muss die Datenpipeline stoppen, sofern noch potentiell Befehle eintreffen könnten, die für ein anstehendes Datenwort relevant sein könnten. Zuständig ist hierbei in erster Linie der jeweilige *Context Separator*. Nachdem der CIO nach der Startadresse sortiert sein muss und die *Context Separator* den vollständigen CIO analysiert, kann dieser relativ leicht entscheiden, ob noch Befehle eintreffen können. Ein kurzes Beispiel: an der Datenpipeline liegt aktuell das Datenwort Nr. 8. Ein CIO Befehl, der sich auf das Datenwort Nr. 15 bezieht hat die *Control Unit* bereits passiert. Somit kann kein Befehl für Datenwort Nr. 9 mehr eintreffen – ein Anhalten der Datenflusspipeline ist nicht erforderlich. Das gleiche gilt unabhängig von den Startadressen, sobald der vollständige CIO die *Control Unit* passiert hat (erkennbar an einem *Last Word*-Signal).

4.3.4 Implementierung

Der *Post-Processor* wurde in VHDL auf einem Xilinx Virtex-4 FX FPGA implementiert und synthetisiert. Der Ressourcenverbrauch ist in Tabelle 4.2 angegeben. Der gesamte *Post-Processor* benötigt 1.841 Virtex-4 Slices, entsprechend 3.353 *Lookup*-Tabellen

		LÖS	EIN	ERS	DEK	CHK	Post-Processor
CU	**Slices**	226	380	332	212	38	-
	Flip-Flops	123	210	196	112	36	-
	4-input LUTs	421	698	611	397	67	-
	RAMBs	0	1	1	0	0	-
DU	**Slices**	263	203	21	47	92	-
	Flip-Flops	169	71	36	36	154	-
	4-input LUTs	489	382	34	82	153	-
CU+DU	**Slices**	419	584	356	229	123	**1841**
	Flip-Flops	297	285	241	156	197	**1383**
	4-input LUTs	786	1093	654	422	200	**3353**
	RAMBs	0	1	1	0	0	**3**
	Max. f	137,1	136,3	162,2	156,7	258,0	**111,0**

Tabelle 4.2: Ressourcenverbrauch beim *Post-Processor* auf Xilinx Virtex-4 FX, aufgeschlüsselt nach Kontrollpfad (CU), Datenpfad (DU) und Summe der einzelnen Pipelinestufen (CU+DU)[1].

mit je vier Eingängen und 1.383 Flip-Flops. Zusätzlich werden drei chipinterne Block-SRAM Speicher benötigt. Diese fungieren zum Einen als Eingangsspeicher und innerhalb der *Insert*- und *Replace*-Einheit als Datenspeicher. Interessant ist der Blick auf den Ressourcenverbrauch der einzelnen Verarbeitungseinheiten im Datenpfad. Wie erwartet sind die *Einfügen*- (203 Slices) und die *Löschen*-Einheit (263 Slices) die größten Einheiten. Hierbei spielt v.a. das *Byte Shuffling* eine Rolle, das im FPGA relativ ineffizient in eine Multiplexer-Architektur umgesetzt werden muss. Die restlichen Datenpfad-Einheiten sind vergleichsweise klein. Bei den Kontrolleinheiten stechen v.a *Einfügen* (380 Slices) und *Ersetzen* (332 Slices) hervor. Die Kontrolleinheit muss hier neben den Kontext-Befehlen auch die Kontext-Daten aufbereiten und speichern. Es wird im Vergleich zu *Löschen* und *Dekrement* also auch die *Data Sorting und Storage Unit* benötigt. Die *Checksummen*-Einheit als direkte Hardware-Implementierung ist mit insgesamt 123 Slices eindeutig die kleinste Verarbeitungsstufe.

Die maximale geschätzte Taktfrequenz des *Post-Processors* liegt bei 111,0 MHz. Er wird innerhalb des FlexPath-Systems mit 100 MHz betrieben. Damit ergibt sich ein Datendurchsatz bei einem 32 Bit-Datenpfad von maximal 3,2 GBit/s. Der Datendurchsatz wird nur erreicht, sofern der CIO rechtzeitig vor dem zugehörigen Paket eintrifft. Sofern der CIO erst mit oder unmittelbar vor dem Paket beim *Post-Processor* eintrifft, kann es zu kurzen Wartezeiten kommen, in der die Datenflusspipeline angehalten werden muss. Die Wartezeit ist neben der genauen Ankunftszeit vom Kontext abhängig und kann sich auf wenige Takte pro Paket summieren. Der tatsächliche Datendurchsatz kann dadurch

[1] Die Zahlen basieren auf einer Schätzung nach der Synthese im Xilinx ISE. Tatsächliche Werte können durch Optimierung, Platzierung etc. abweichen. Dadurch ergeben sich auch Abweichungen in der Summe der Ressourcen von CU plus DU im Vergleich zur kombinierten Synthese mit CU und DU.

geringer ausfallen.

4.4 Konzeptevaluierung

In diesem Abschnitt werden die Kernelemente des zuvor vorgestellten FlexPath-Projektes anhand eines Simulationsmodells näher untersucht. Besonderes Augenmerk liegt dabei auf der Hardwareunterstützung bei FlexPath, wie sie v.a. auch durch den Einsatz des *Post-Processors* ermöglicht wird. Dabei steht v.a. die Beschleunigung von CPU-Paketen durch Auslagern von einzelnen Funktionen auf die Hardware (vgl. Tabelle 4.1) im Vordergrund. Es sollen aber auch die Vorteile von *AutoRoute* untersucht werden. Da diese Untersuchungen nicht den Schwerpunkt dieser Arbeit bilden, wird lediglich kurz auf das zugrundeliegende Modell und die wichtigsten Ergebnisse der Simulationen eingegangen. Detailierte Ergebnisse finden sich in [71] und [66].

4.4.1 SystemC-Simulationsmodell

SystemC

Als Simulationsumgebung wird SystemC [72] verwendet. SystemC stellt alle Funktionen zur Modellierung und Simulation von komplexen Systemen bestehend aus Hard- und Software-Komponenten bereit. Es handelt sich hierbei um keine eigenständige Sprache, sondern vielmehr um eine Erweiterung (Klassenbibliothek) für C++. Ähnlich wie VHDL erlaubt es eine nebenläufige Beschreibung der einzelnen Module. Es eignet sich aber auch sehr gut für die Beschreibung auf höheren Abstraktionsebenen. Entsprechend kann mit SystemC eine deutlich höhere Simulationsgeschwindigkeiten als z.B. mit VHDL erreicht werden.

TAPES-Simulator

Das zur Simulation genutzte Simulationsmodell basiert auf den am Lehrstuhl für Integrierte Systeme der TU München entwickelten TAPES-Simulator (*Trace-based Architecture Performance Evaluation with SystemC*, siehe [73], [74]). Bei TAPES steht grundsätzlich nicht die Funktion der einzelnen Module im Vordergrund. Vielmehr wird das Systemverhalten durch sogenannte *Traces* abstrahiert. Ein *Trace* gibt dabei das zeitliche Verhalten eines Moduls bei Eintreten eines Ereignisses wieder. Es wird also beschrieben, welche Zugriffe auf andere Module nacheinander ausgeführt werden bzw. welche zeitlichen Verzögerungen im Modul zustande kommen. Beispielsweise wird angegeben, welche Zugriffe die *DMA Engine* bei Ankunft eines Pakets durchführt. Dies beinhaltet z.B. das Abspeichern des Pakets im Paketspeicher (inklusive der zu übertragenden Datenmenge) oder die Information der nachfolgenden Einheiten über das abgespeicherte Paket. Auf diese Weise werden die zeitlichen Zusammenhänge im System nachgebildet. Konflikte beim Zugriff auf gemeinsam genutzte Ressourcen (z.B. Speicher, Bus) werden

dynamisch aufgelöst. Dies erlaubt eine erste Geschwindigkeitsabschätzung des Gesamtsystems. Zudem werden Engstellen und Dimensionierungsprobleme im System frühzeitig erkannt. Durch die Abstraktion wird eine ausreichend hohe Simulationsgeschwindigkeit erreicht.

FlexPath-Systemmodell

In Abbildung 4.16 ist der Aufbau des FlexPath-Simulationsmodells abgebildet (vgl. auch [71], [66]). Es orientiert sich an der angestrebten Demonstrator-Architektur in einem Xilinx Virtex-II Pro / Virtex-4 FPGA (vgl. auch Kapitel 7). Der Prozessorkomplex besteht aus einer variablen Anzahl an Prozessoren, die über einen gemeinsamen Bus verbunden sind. Entsprechend dem Ziel-FPGA wird ein *Processor Local Bus* (PLB) verwendet – ein von IBM spezifizierter 64 Bit-Bus. Da der gemeinsame Bus eine potentielle Schwachstelle im System darstellt, wird ein zyklenakkurates Modell verwendet, das zudem die wichtigsten Eigenschaften (paralleler Schreib- und Lesebus, *Address Pipeling*, *Burst Access* etc.) abbildet. Die zentrale *DMA Engine* (*SmartMem Buffer Manager*) simuliert die notwendigen Speicherzugriffe beim Abspeichern und Senden der Pakete. *Pre-Processor*, *Path Dispatcher* (Paket-Klassifizierer) und *Post-Processor* sind im Simulator weitgehend funktionslos, lediglich die Verzögerungen der Module werden nachgestellt. Zudem enthält der *Path Dispatcher* Queues für jeden Prozessor und verteilt die ankommende Pakete im Cluster. Das System wird mit vier Gigabit-Ethernet Ports betrieben. Der jeweilige Zielport wird für jedes Paket zufällig gewählt. Die Wahrscheinlichkeit ist dabei gleichverteilt. Zudem enthält der *Traffic Manager* die jeweiligen Ausgangs-Queues. Wahlweise kann das Simulationsmodell als Referenz auch ohne die FlexPath-spezifischen Komponenten betrieben werden.

Die Qualität der Simulationen und damit der zu treffenden Aussagen hängt maßgeblich von der Güte der verwendeten *Traces* ab. Aus diesem Grunde wurde ein einfacher Prototyp auf einem Virtex-II Pro realisiert. Ziel war es, den Verlauf eines Pakets zu verfolgen und die Zugriffsmuster der einzelnen Module (insbesondere der *DMA Engine* und der Prozessoren) mitzuschneiden und auszuwerten. Als Prozessor wurde der *on-chip* PowerPC 405 des Virtex-II Pros verwendet. Als Software wurde der bereits in Abschnitt 4.1.1 erwähnte lwIP [65] verwendet.

Aus den Messungen konnten die in Tabelle 4.3 angegebenen Werte (linke Spalte) ermittelt werden. Insbesondere die Prozessierungslatenz und die Anzahl der *Cache Refills* bei den Instruktionen sind für die Simulation substantielle Werte. Die Werte für FlexPath (rechte Seite der Tabelle) wurden dagegen auf Basis der Werte in Tabelle 4.1 geschätzt. Nachdem sich die Anzahl der Instruktionen pro Paket durch Nutzung der Hardware-Unterstützung von 2.421 auf 1.885 reduziert, wird angenommen, dass sich die Anzahl der Zyklen (bei konstanten CPI) um denselben Faktor von 5.080 auf dann 3.955 senkt. Auch *Cache Refills* wurden um den gleichen Faktor angeglichen. Da das FlexPath-Modell auf dem Paket-Kontext anstatt auf den echten Paketdaten arbeitet, entfällt das Lesen und Schreiben des Paket-Headers. Stattdessen müssen jedoch die passenden Paket-Kontexte (CII, CIO) gelesen und geschrieben werden.

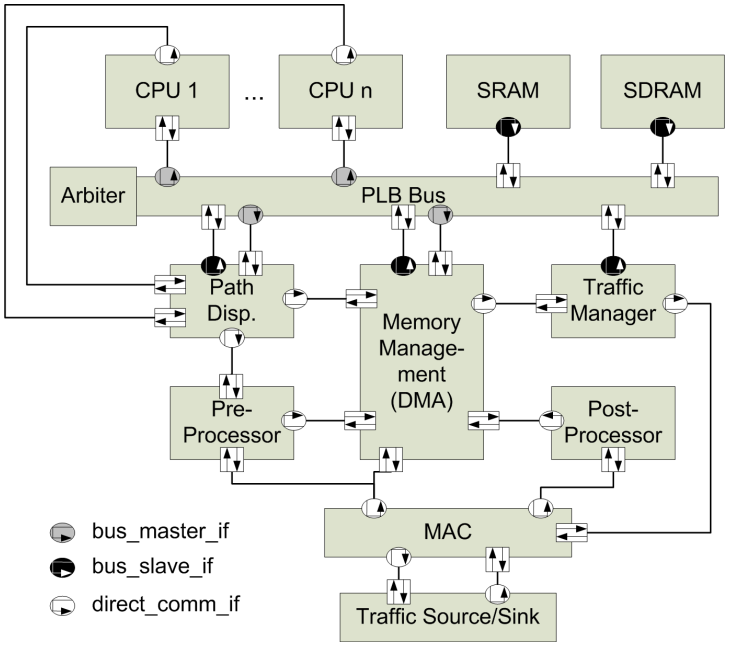

Abbildung 4.16: SystemC-Modell zur Geschwindigkeitsabschätzung der FlexPath-SoC-Architektur.

4.4.2 Simulationsergebnisse

Das Simulationsmodell kann durch die Verwendung von *pcap*-Dateien [75] sowohl mit künstlichem als auch mit realem (mitgeschnittenem) Verkehr stimuliert werden. Zur Stimulation von *worst case*-Szenarien wurde künstlicher Verkehr mit fester Paketgröße verwendet.

Zunächst wurden Simulationen ohne *AutoRoute* durchgeführt und die Hardwareunterstützung des Prozessors mit unterschiedlichen Paketgrößen simuliert. Nachdem die Prozessierungslatenz nach Tabelle 4.3 um 22% gesenkt werden konnte, würde man eine Steigerung des System-Durchsatzes auf $1/(1-0,22) = 128\%$ erwarten. Tatsächlich fällt die Steigerung aufgrund der steigenden Buslast mit 27,5% (64 Byte-Pakete) bzw. 24,7% (1.518 Byte-Pakete) etwas geringer aus.

Im nächsten Schritt wurden die Prozessorpakete nun mit wachsendem *AutoRoute*-Anteil kombiniert. In Abbildung 4.17 ist der simulierte Maximaldurchsatz durch das FlexPath-System in Paketen pro Sekunde bei unterschiedlichen Paketgrößen und unterschiedlichen *AutoRoute*-Anteilen gezeigt. Nachdem bei IP-Forwarding lediglich der Header prozessiert wird, lässt sich eine konstante Paketrate erwarten, zumindest solange die CPU das limitierende Element darstellt. Tatsächlich bleibt die Paketrate für alle Messreihen mit steigender Paketgröße zunächst nahezu konstant. Insbesondere bei größeren Paketen sieht man jedoch einen wachsenden Einbruch. Da Prozessor und DMA sich den Bus teilen, führt dies bei größer werdenden Paketen (und damit steigender

	Software-Referenz (FPGA-Messung)	**FlexPath m. Hardware-Unterstützung (Schätzung)**
Prozessierungslatenz	5.080 Clock-Zyklen	3.955 Clock-Zyklen
Bus Transaktionen (jeweils Lesen/ Schreiben)	Instr. Cache Refill 41x32B Paket-Deskriptor 2x16B Paket-Header 2x64B	Instr. Cache Refill 32x32B Paket-Deskriptor 2x16B Paket-Kontext 2x64B Out. Ctx. Deskriptor 32 Bit

Tabelle 4.3: Prozessierungslatenz & Bus Transaktionen bei der Referenzimplementierung und mit Hardwareunterstützung (geschätzt).

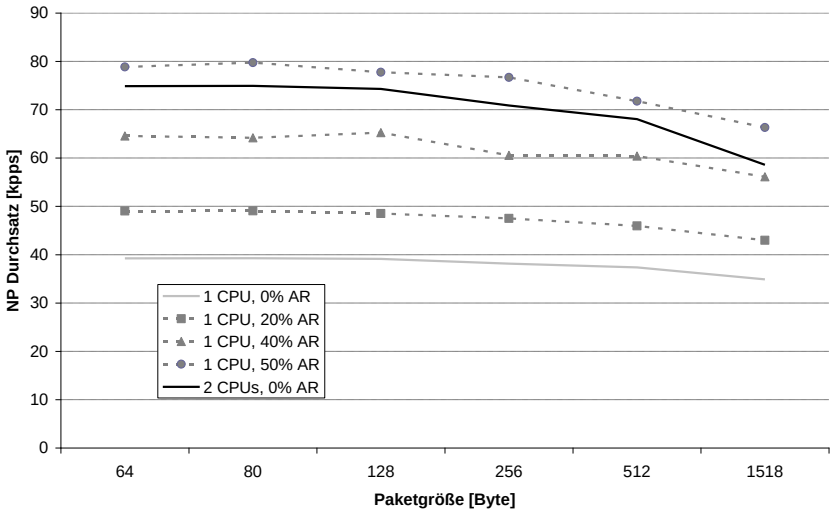

Abbildung 4.17: Simulierter Systemdurchsatz bei unterschiedlichen Anteilen an *AutoRoute*-Paketen bei FlexPath.

Datenrate) zu steigenden Konflikten und damit Geschwindigkeitseinbußen.

Wie erwartet steigt mit wachsenden *AutoRoute*-Anteil auch der Gesamtdurchsatz im System. Auffällig ist hierbei, dass dieser Anstieg nicht linear, sondern tatsächlich etwas stärker ausfällt. Man erkennt z.B., dass die Erhöhung des Durchsatzes zwischen 40% und 50% *AutoRoute*-Anteil in etwa gleich groß ist, wie der zwischen 20% und 40%. Jedes Prozessor-Paket erzeugt auf dem Bus eine zusätzliche Last durch das Nachladen der Instruktionen. Sofern ein Paket nicht vom Prozessor sondern per *AutoRoute* bearbeitet wird, sinkt damit nicht nur die Auslastung des Prozessors, sondern zudem die Bus- und Speicherauslastung. Damit werden Ressourcenkonflikte vermindert. Dies erklärt auch, warum das System mit einer CPU und 50% *AutoRoute*-Anteil, das System mit zwei CPUs ohne *AutoRoute* vom Durchsatz her übertrifft.

Der verbesserte Systemdurchsatz ist allerdings nur ein Aspekt beim Einsatz von *AutoRoute*. In Abbildung 4.18 erkennt man, dass die Durchgangslatenzen von *AutoRoute*-

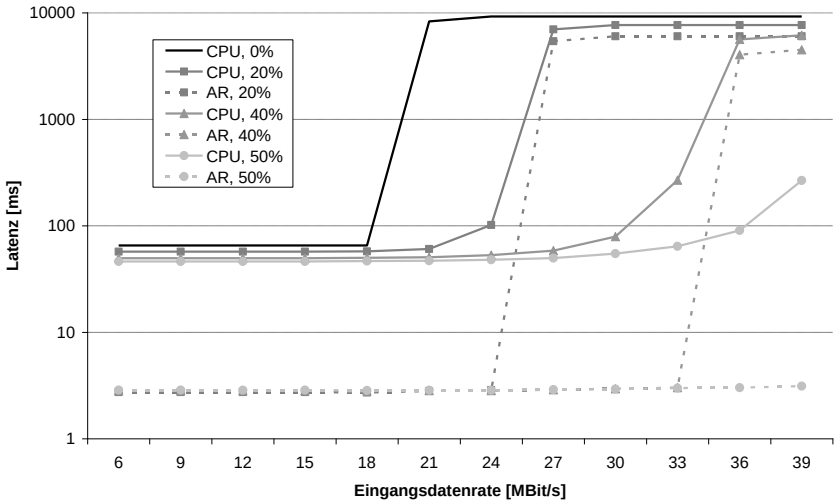

Abbildung 4.18: Simulierte Systemlatenz bei unterschiedlichen Anteilen an *AutoRoute*-Paketen und wachsender Eingangsdatenrate bei FlexPath (Paketgröße: 64 Byte). CPU: Prozessor-Paket, AR: *AutoRoute*-Paket, %: *AutoRoute*-Anteil am Verkehr.

Paketen deutlich kleiner sind als die der Prozessor-Pakete. Damit werden die *AutoRoute*-Pakete wesentlich schneller weitergeleitet, was – wie in Abschnitt 2.3.3 erläutert – ein wesentliches Qualitätskriterium bei bestimmten Anwendungen sein kann. Steigert man nun die Eingangsdatenrate kontinuierlich, so sieht man, dass das System allmählich in Überlast geht. Die Latenzen der einzelnen Pakete steigen deutlich an. Dies lässt sich v.a. durch die sich langsam anfüllenden Paketpuffer erklären. Die Latenzen erreichen schließlich ihr Maximum bei vollständig gefüllten Paketpuffern. Ein höherer *AutoRoute*-Anteil zögert dabei den Anstieg der Latenzen auf höhere Eingangsdatenraten hinaus. Hiervon profitieren ebenso die Prozessor-Pakete.

4.5 Bewertung

In diesem Kapitel wurden die Grundzüge des FlexPath-Konzepts erläutert und dabei insbesondere ein Schwerpunkt auf die Hardwareunterstützung gelegt. FlexPath erlaubt mit Hilfe des vorgestellten *Post-Processors* einen vollständigen Hardware-Bearbeitungspfad (*AutoRoute*) für Pakete mit Standardaufgaben (insbesondere IP-Forwarding). Aber auch andere Pakete, die von einer CPU prozessiert werden müssen, können von der Hardwareunterstützung profitieren. Die Vorteile von *AutoRoute* wurden in den Abschnitten 4.1.1 und 4.4.2 ausführlich erläutert.

Der Haupteinsatzbereich des *Post-Processors* liegt daher in der Unterstützung von *AutoRoute* im FlexPath-NP. Es ergeben sich aber auch andere, interessante Einsatzmöglichkeiten, z.B. im Zusammenhang mit IPsec. Dabei kann er das CPU-Cluster vom

verhältnismäßig aufwändigen Einfügen und Löschen einzelner Header und Trailer in und aus dem Paket entlasten.

Durch den modularen Aufbau wurde eine sehr flexible Architektur geschaffen, die den jeweiligen Anforderungen verschiedener Applikationen sehr gut angepasst werden kann. Sofern der *Post-Processor* z.B. nur im Rahmen von *AutoRoute* eingesetzt werden soll, können die *Einfügen-* und *Löschen*-Einheit sehr leicht aus dem Design entfernt werden. Wie in Tabelle 4.2 gesehen benötigen diese beiden Einheiten einen Großteil der Gesamtressourcen. Der Ressourcenverbrauch sinkt dabei um mehr als den Faktor zwei. Es ist aber ebenso denkbar, neue Module in das Design aufzunehmen.

Der *Post-Processor* selbst besitzt noch Verbesserungspotential. Ein entscheidender Faktor bei der Bewertung der Effizienz des *Post-Processors* stellt die Größe des notwendigen CIO dar. Ein Standard-CIO kann, wie in Abbildung 4.13 dargestellt, durchaus aus acht Wörtern oder auch mehr bestehen. Damit sind neben den Paketdaten 32 Byte an CIO zu übertragen. Bei einer *worst case*-Betrachtung unter der Annahme einer Paketgröße von 64 Byte, ergibt dies einen zusätzlichen Aufschlag von 50%, die das Speicher- und *Interconnect*-Subsystem zusätzlich zu bewältigen hat. Je nach Architektur und Auslastung kann dies zu erheblichen Geschwindigkeitseinbußen führen. Aus diesem Grunde ist es wünschenswert, die Größe des CIOs auf ein Minimum zu reduzieren. Ein Ansatzpunkt ist, dass viele Pakete einen sehr ähnlichen oder identischen CIO besitzen. Bei Standard-*AutoRoute*-Paketen ist die Befehlsreihenfolge im CIO praktisch immer gleich, lediglich die Daten (sprich Ethernet-Adressen) variieren. Es ist daher naheliegend, den *Post-Processor* dahingehend zu erweitern, Befehls-Templates verwenden zu können. Damit könnten vier Befehle auf einen reduziert werden. Die originale Befehlsreihenfolge könnte innerhalb des *Post-Processors* vor den eigentlichen Verarbeitungseinheiten wiederhergestellt werden. Aber nicht nur Befehlsmuster, auch Daten wiederholen sich für eine Vielzahl von Paketen. Schließlich ist die Anzahl an möglichen Ziel- und Quell-MAC-Adressen innerhalb des NPs in der Realität doch sehr eingeschränkt. Hier könnte ein lokaler kleiner Speicher im *Post-Processor* häufig verwendete Daten vorhalten. Innerhalb des CIOs würde nur ein verhältnismäßig kleiner Zeiger auf den jeweils benötigten Speicherinhalt zeigen. Diese Vorgehensweise könnte auch innerhalb von IPsec Anwendung finden. Auch die einzufügenden Header/Trailer sind für eine Tunnelverbindung weitestgehend identisch und könnten lokal zwischengespeichert werden.

5 Paketverteilung und Lastbalancierung im Multiprozessor-Cluster

5.1 Motivation

Um die notwendige Leistungsfähigkeit eines Netzwerkprozessors zu erreichen, werden, wie in Kapitel 3.1 skizziert, häufig parallele, überwiegend homogene Verarbeitungseinheiten eingesetzt. Auch wenn die Gestaltung dieser Verarbeitungseinheiten bei verschiedenen Architekturen höchst unterschiedlich ausfallen kann, so besteht ein und dasselbe Grundproblem bei allen Architekturen: Wie sollen ankommende Pakete auf die zur Verfügung stehenden Verarbeitungseinheiten verteilt werden, um eine möglichst effiziente Auslastung des Gesamtsystems zu erreichen? Wie können also mit einem gegebenen System die Anforderungen an Gesamtdurchsatz, Verarbeitungslatenz und QoS bestmöglich erfüllt werden?

Eine sehr einfache Möglichkeit wäre die Verwendung eines *Round Robin*-Verfahrens: Hierbei werden ankommende Pakete reihum auf die zur Verfügung stehenden Verarbeitungseinheiten verteilt. Damit müsste sich eigentlich eine gleichmäßige Verteilung ergeben. Dabei gilt es aber, wie auch schon in Kapitel 3.2 zum Teil gezeigt, eine Reihe von Nebenbedingungen zu beachten, die gegen ein solches Verfahren sprechen:

- Die **Paketreihenfolge** innerhalb eines Flows soll im NP aufrecht erhalten bleiben. Auch wenn es sich hierbei um keine *harte* Forderung handelt (protokolltechnisch werden Vertauschungen durchaus abgefangen), so handelt es sich doch um ein deutliches Qualitätskriterium (siehe hierzu auch Abschnitt 2.3.3). Wie wichtig dies einzuschätzen ist, erkennt man schon daran, dass alle in Abschnitt 3.2 vorgestellten Lastbalancierungsverfahren dem Rechnung tragen, indem Paketvertauschungen auf ein Minimum reduziert werden.
- Pakete haben mitunter unterschiedliche Prozessierungsanforderungen. Damit gehen auch **unterschiedliche Prozessierungslatenzen** einher. Die Last, die ein Paket auf einer CPU erzeugen wird, kann daher nicht von vornherein bestimmt werden. Ausnahmen finden sich nur bei homogenem Verkehr mit identischen Prozessierungsanforderungen (z.B. IP-Forwarding ohne weitere Verarbeitung).
- Die Aktivität der einzelnen Flows kann stark variieren. In der Regel gibt es sehr viele Flows mit geringer Aktivität und wenige mit hoher Aktivität (siehe auch [53]). Diese Verteilung ist zudem teilweise starken **zeitlichen Schwankungen** unterworfen, die sowohl längerfristiger Natur (zeitlicher Verlauf z.B. im Tagesverlauf), als auch kurzfristiger Natur (Bursts) sein können.

Die in Abschnitt 3.2 vorgestellten Verfahren versuchen diese Gegebenheiten bestmöglich zu berücksichtigen. Zeitlichen Schwankungen im Verkehr wird durch Umbalancierungen begegnet, die Lasten der einzelnen CPUs also angeglichen. Pakete eines Flows werden dabei in der Regel von ein und derselben CPU verarbeitet. Auf diese Weise wird *Packet Reordering* weitestgehend vermieden. Nur während der Umbalancierung wird das Riskio von *Packet Reordering* in Kauf genommen. Dennoch besitzen die vorgestellten Verfahren eine Reihe von Nachteilen, die je nach Anwendungsfall und Anforderungen unterschiedliche Relevanz haben:

- Während langfristige Schwankungen im Verkehr relativ gut in den Griff zu bekommen sind, machen v.a. kurzfristige zeitliche Schwankungen den Lastbalancierungsverfahren Probleme. Hierbei stellt sich v.a. die Frage, wie schnell auf Veränderungen im Verkehr reagiert werden soll. Reagiert das Verfahren zu träge, läuft es Gefahr, eine temporäre Überlast einzelner CPUs nicht zu verhindern. Reagiert es zu schnell, wird ein häufiges Umbalancieren und damit auch erhöhtes *Packet Reordering* gefördert. Kencl und Shi [55] haben durch die Kombination ihrer Systeme versucht, diesen Umstand etwas abzumildern, wenngleich auch nicht beseitigt.
- Gerade das Verfahren nach Kencl [52][55] berücksichtigt in erster Linie homogenen Verkehr. Für die Berechnung der CPU-Auslastung wird eine (nahezu) konstante Prozessierungslatenz angenommen. Inhomogener Verkehr (unterschiedliche Prozessierungslatenzen) wird nicht ohne weiteres unterstützt.
- *Quality of Service* wird bei den bekannten Verfahren nicht berücksichtigt. Die Zuordnung erfolgt ohne jegliche Kenntnis der Prioritäten. Erst nach der Zuordnung wird in der CPU erkannt, welche Priorität ein Paket besitzt.
- Es werden nur homogene Multiprozessorsysteme unterstützt, d.h. dass jede CPU die gleiche Funktionalität besitzt.

Um die Schwächen der vorgestellten Verfahren zu umgehen, wurde im Rahmen des FlexPath-Projekts ein eigenes, neuartiges Lastbalancierungsverfahren vorgestellt. Grundidee dieses Verfahrens ist die eingangsseitige **Klassifizierung** des Verkehrs und Anwendung unterschiedlicher Strategien in Abhängigkeit vom Pakettyp. Insbesondere werden **unterschiedliche Balancierungsstrategien für zustandslosen und zustandsbehafteten Verkehr** vorgesehen. Es lassen sich aber auch unterschiedliche Prioritäten berücksichtigen und damit der **Quality of Service** verbessern. Schließlich können somit auch **heterogene Systeme** besser unterstützt werden.

Im Rahmen dieser Lastbalancierungsstrategie lässt sich der Schwerpunkt der vorliegenden Arbeit wie folgt definieren:

- Es wurde eine **Lastbalancierungsstrategie für zustandslosen Verkehr** entwickelt (siehe Kapitel 5.2.1). Zustandslose Pakete können dabei unter Beachtung des *Packet Reordering*-Problems zur Prozessierung auf unterschiedliche CPUs verteilt werden. Es wird ein selbst-adaptives Verfahren vorgestellt, das im Rahmen von FlexPath in Kombination mit zustandsbehaftetem Verkehr verwendet werden kann. Das Verfahren kann jedoch auch eigenständig verwendet werden, sofern es in einem homogenen System mit ausschließlich zustandslosem Verkehr (v.a. im

Backbone-Bereich) eingesetzt wird.

- Mit Hilfe der Lastbalancierungsstrategie wird eine Pfadentscheidung getroffen. Dabei kann das Ziel eine konkrete (dedizierte) CPU sein, oder auch ein Pool von CPUs, von denen grundsätzlich jede beliebige ausgewählt werden darf. Zur Ausführung der Pfadentscheidung wird eine Einheit zur **Verteilung der Pakete innerhalb des Multiprozessor-Clusters** (*Packet Distributor*, siehe Abschnitt 5.3) benötigt. Sie informiert die CPUs über ankommende Pakete und wählt eine CPU aus, sofern das Ziel aus mehreren Alternativen besteht.

Im Folgenden wird zunächst die Lastbalancierungsstrategie für zustandslosen Verkehr erläutert. Im Anschluss wird das kombinierte Lastbalancierungsverfahren für FlexPath vorgestellt. Nach der Einführung des *Packet Distributors* erfolgt eine simulative Evaluierung und Bewertung der vorgeschlagenen Mechanismen.

5.2 Lastbalancierungsstrategien

Bevor auf das kombinierte Lastbalancierungsverfahren im FlexPath-NP eingegangen wird, wird zunächst ein Verfahren für zustandslosen Verkehr vorgestellt. Das Verfahren lässt sich dabei sowohl in FlexPath, prinzipiell aber auch als eigenständiges Verfahren in einer Umgebung mit rein zustandslosen Paketen einsetzen.

5.2.1 Lastbalancierungsstrategie für zustandslosen Verkehr

Zustandsloser Verkehr zeichnet sich bereits dem Namen nach dadurch aus, dass für zusammengehörige Pakete, d.h. Pakete eines Flows, kein gemeinsamer Zustand gespeichert und verwaltet werden muss. Eine Verarbeitung zustandsloser Pakete basiert deshalb rein auf dem Paketinhalt des zu verarbeitenden Pakets. In einem homogenen Prozessor-Cluster kann damit jedes Paket grundsätzlich immer von jeder CPU verarbeitet werden. Lediglich aus Gründen des *Packet Reorderings* kann es empfehlenswert sein, Pakete eines Flows derselben Verarbeitungseinheit zu übergeben (vgl. Abschnitt 3.2).

Basierend auf dem Ansatz von Dittmann [51] (siehe Abschnitt 3.2) wird für diese Paketklasse ein *Packet Spraying* vorgeschlagen. Dittmann hat *Spraying* für jene Flows angewandt, deren Prozessierung die Kapazität einer CPU übersteigt und damit nicht von einer dedizierten CPU verarbeitet werden kann. Dabei wird jedes ankommende Paket eines solchen Flows an die jeweils kürzeste CPU-Queue übergeben (siehe Abbildung 5.1). Das Paket soll also von der vermeintlich am wenigsten belasteten CPU verarbeitet werden.

Im Gegensatz zu Dittmann wird das *Packet Spraying* in dieser Arbeit nicht als Notlösung angesehen, sondern bewusst für alle zustandslosen Pakete angewandt. Die beiden Verfahren unterscheiden sich dabei in einem wesentlichen Punkt: bei Dittmann erfolgte das *Spraying* **vor** den CPU-Queues. Der *Sprayer* wählt hierbei eine (nämlich die kürzeste) Queue aus. Nachdem bei Dittmann jede Queue genau einer CPU zugeordnet ist, wird dadurch auch die CPU festgelegt. Bei dem hier vorgeschlagenen Verfahren er-

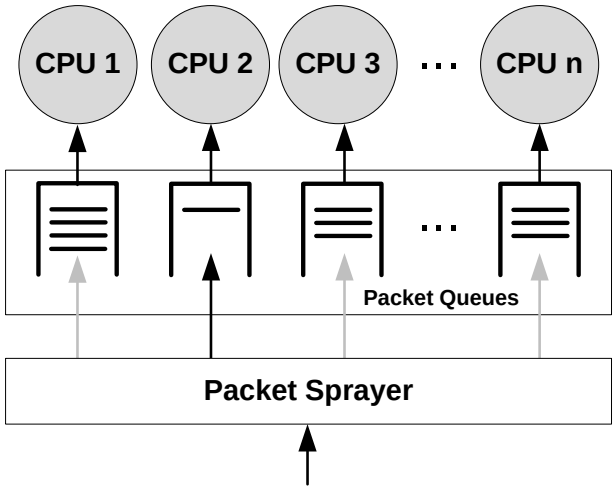

Abbildung 5.1: *Packet Spraying* nach Dittmann vor den CPU-Queues.

folgt das *Spraying* **nach einer gemeinsamen** Queue (siehe Abbildung 5.2). Dabei wird das jeweils erste Paket der Queue einer freien bzw. der jeweils nächsten freien CPU zur Verarbeitung übergeben.

Damit hat das Verfahren bezüglich *Packet Reordering* entscheidende Vorteile gegenüber Dittmann. Nachdem das *Spraying* erst nach der Queue erfolgt, startet die Prozessierung selbst noch sequentiell. Nachdem bei zustandslosem Verkehr eine homogene Verarbeitung zu erwarten ist (z.B. IP-Forwarding), können sich Paketüberholungen praktisch nur durch Verarbeitungsjitter innerhalb des CPU-Clusters, z.B. aufgrund unterschiedlicher Zugriffszeiten auf Bus oder Speicher, ergeben. Bei Dittmann muss zudem die unterschiedliche Verweildauer zweier Pakete in den parallelen CPU-Queues hinzugerechnet werden. Diese kann ein Vielfaches der Prozessierungsdauer im CPU-Cluster betragen und ist damit gegenüber dem Jitter innerhalb des Clusters dominant. Damit steigt auch die Gefahr von *Packet Reordering* entsprechend. Die gleiche Überlegung gilt im Übrigen bei Umbalancierungen bei den anderen vorgestellten dynamische Lastbalancierungsverfahren.

Ungeachtet des potentiellen Ausmaßes von *Packet Reordering*, welches noch in Abschnitt 5.4.3 und Kapitel 6 eingehend untersucht wird, besitzt das vorgestellte Verfahren eine Reihe entscheidender Vorteile gegenüber den in Abschnitt 3.2 genannten Verfahren:

- Bei n CPUs kann im Vergleich zur dedizierten Zuweisung eine n-fach große Queue verwendet werden, ohne dass die Prozessierungslatenz ansteigt. Schließlich bearbeiten nun n statt einer CPU diese Queue. Bei den anderen Verfahren besteht grundsätzlich das Problem, dass bei schneller Änderung des Verkehrsverhaltens die Balancierung nicht schnell genug gegensteuern kann. Während einzelne CPUs in Unterlast bleiben, gehen andere in Überlast. Es kommt zu Paketverlusten, obwohl das CPU-Cluster insgesamt genug Rechenressourcen besitzt. Bei *Spraying* kann durch die eine große Queue ein wesentlich größerer Burst abgefangen werden,

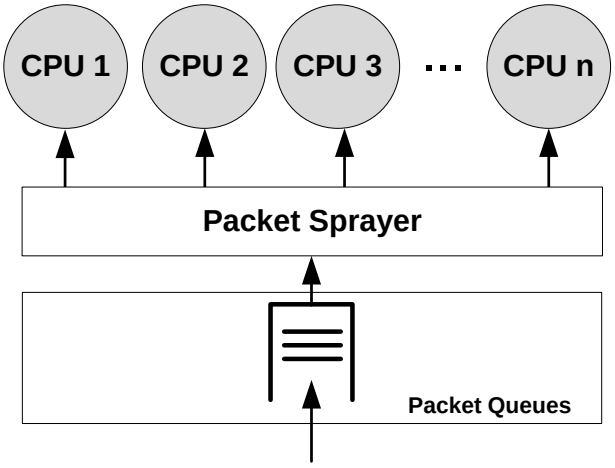

Abbildung 5.2: *Packet Spraying* im FlexPath-NP.

ehe es zu Paketverlusten kommt. Es entsteht also ein **Bündelungsgewinn**.

- *Spraying* führt zu einer quasi **optimalen Lastverteilung** auf den CPUs, da nur freie CPUs Pakete aus der *Spraying*-Queue erhalten. Belegte CPUs werden nicht berücksichtigt.
- Damit ist das Verfahren auch vollständig **selbst-adaptiv**. Es werden dabei selbst kurzfristige Schwankungen ausgeglichen. Es ist keinerlei Regelalgorithmus innerhalb des Systems notwendig. Eine komplexe Implementierung entfällt damit.

An dieser Stelle bleibt noch anzumerken, dass Argumente wie der teilweise in NPs zu findende Einsatz von *Lookup*-Caches gegen dieses Verfahren sprechen können. Abhängig von der konkreten NP-Architektur werden lokale Caches verwendet, die z.B. ein *Lookup*-Ergebnis eines Flows zwischenspeichern. Dieses Vorgehen ist durchaus sinnvoll, da in der Regel in einem begrenzten Zeitraum jeweils mehrere Pakete eines Flows eintreffen. Sofern die Pakete von derselben CPU bearbeitet werden, kann das Ergebnis direkt aus dem Cache verwendet werden, eine erneute Anfrage an eine externe *Lookup-Engine* ist nicht erforderlich. Werden die Pakete dagegen verteilt, benötigt jede CPU einen eigenen Eintrag in ihrem Cache. Das erhöht die Anzahl der *Lookup*-Anfragen, zugleich muss jede CPU mehr Einträge vorhalten. Die Wahrscheinlichkeit von Cache-Verdrängungen steigt deutlich, was die Leistungsfähigkeit des NPs insgesamt senkt. Im FlexPath-NP spielt diese Problematik keine Rolle. Ein *Lookup*-Ergebnis wird bereits eingangsseitig erzielt und im Paket-Kontext der CPU mitgeteilt (siehe Abschnitt 4.2, *Pre-Processor*). Ein lokaler Cache für *Lookup*-Ergebnisse ist deshalb im FlexPath-NP nicht notwendig.

5.2.2 Lastbalancierung im FlexPath-NP

Während sich *Spraying* hervorragend zur Paketverteilung von zustandslosem Verkehr eignet, ist die Verwendung von *Spraying* in einer heterogenen Umgebung mit gemischt

zustandslosem und zustandsbehaftetem Verkehr nicht mehr ohne weiteres möglich. Bei zustandsbehaftetem Verkehr handelt es sich um Pakete, für deren Bearbeitung die Verwaltung von Zustandsinformationen notwendig ist. Dies kann z.B. sowohl beim *Policing*, als auch *Shaping* der Fall sein. Aber auch IPsec-Pakete am Anfang bzw. Ende eines IPsec-Tunnels können hierzu gezählt werden. Bei IPsec ist beispielsweise die Verwaltung der Schlüssel, der Sequenznummer etc. notwendig. Zustandsbehaftete Pakete eines Flows werden idealerweise von derselben CPU bearbeitet. Bei einer Verteilung auf mehrere CPUs ergäbe sich das Problem der Dateninkonsistenz der Zustandsinformationen. Die Informationen müssten zentral in einem geteilten Speicher liegen (siehe Abbildung 5.3a). Da hierbei mehrere CPUs parallel an den Daten arbeiten können, wird ein Semaphorschutz unbedingt notwendig und damit die Komplexität erhöht. Die resultierenden Ressourcenkonflikte ergeben Geschwindigkeitseinbußen.

Innerhalb des FlexPath-NPs kann man sich nun die Vorteile der Paketklassifizierung zunutze machen. Damit unterscheidet sich das Lastbalancierungsverfahren im FlexPath-NP besonders in einem Punkt deutlich von den vorgestellten Verfahren in Kapitel 3.2, bei denen die Balancierung nur anhand eines kalkulierten Hashwertes ohne Kenntnis von Pakettyp oder Priorität erfolgt. Anhand der Paketmerkmale (Typ, IP-Adressen etc.) kann eine Einteilung in zustandslosen und zustandbehafteten Verkehr erfolgen. Für beide Klassen können nun unterschiedliche Strategien angewandt werden:

- **Zustandsloser Verkehr:** Für zustandslosen Verkehr wird das in Abschnitt 5.2.1 vorgestellte *Spraying* verwendet.
- **Zustandsbehafteter Verkehr:** Für zustandsbehafteten Verkehr eignet sich ein dediziertes Verfahren, dass die Zuordung zwischen Flow und CPU bestmöglich aufrechterhält. Als Kandidaten eignen sich hierzu prinzipiell die in Abschnitt 3.2 vorgestellten Verfahren.

In einem zweiten Schritt lässt sich das Verfahren weiter ausbauen, indem unterschiedliche Prioritätsklassen der Pakete berücksichtigt werden.

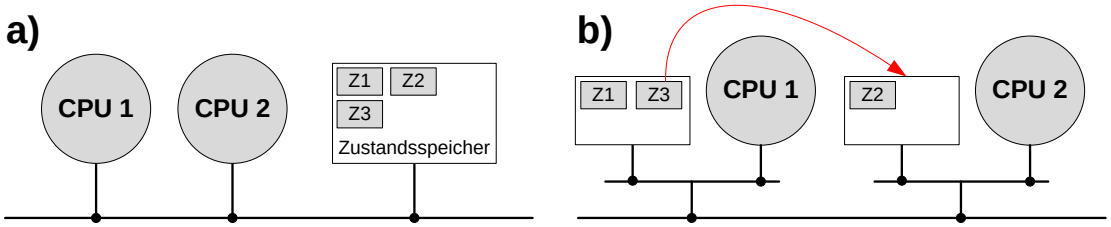

Abbildung 5.3: Speichern von Zustandsinformationen bei zustandsbehaftetem Verkehr: mit zentralem Speicher (a) und lokalem Speicher (b).

Lastbalancierungsstrategie für zustandsbehafteten Verkehr

Bei zustandsbehaftetem Verkehr empfiehlt sich die lokale Speicherung der Zustandsinformationen. Mit dieser Vorgehensweise lassen sich Ressourcenkonflikte und Dateninkonsi-

stenzen vermeiden. Damit ergibt sich letztendlich die gleiche Problematik, wie sie aus den Lastbalancierungsverfahren in der Literatur bereits bekannt ist. Auch dort sollen, wenn auch primär aus Gründen des *Packet Reordering*, die Pakete auf derselben CPU prozessiert werden. Damit lassen sich die bekannten adaptiven Verfahren nach Kencl und Shi [52][54][55] durchaus auf dieses Problem übertragen. Diese Verfahren besitzen allerdings eine vergleichsweise hohe Komplexität, deren Umsetzung mit einem hohen Ressourcenverbrauch einhergeht. Nachdem sich diese Lastbalancierungsstrategie im FlexPath nur auf einen sehr kleinen Teil der Pakete bezieht, wären die Kosten im Verhältnis zum Nutzen relativ hoch. Geht man von einem durchschnittlichen Internet-Verkehrsmix aus, bei dem lediglich die IPsec-Pakete als zustandsbehaftet einzustufen sind, bewegen sich die Zahlen im niedrigen einstelligen Prozentbereich der Gesamtpakete. Aus diesem Grunde wird eine weniger komplexe, aber durchaus leistungsfähige Vorgehensweise angestrebt. Neben möglichst ausgeglichenen CPU-Lasten sind aber auch die Anzahl der Umbalancierungen ein Merkmal für die Qualität des Lastbalancierungsverfahren. Letztendlich zieht jede Umbalancierung eine Migration der Zustandsinformationen eines Flows von einer CPU zur anderen nach sich, also ein Umkopieren von einem lokalen Speicher zum anderen (siehe Abbildung 5.3b). Ein entsprechendes Verfahren, das sogenannten HLU (*Hash-Lookup*), wurde in [76] vorgestellt. Es wird schwerpunktmäßig in [3] erläutert.

Prioritätsbasierte Lastbalancierung

Im FlexPath-NP kann die Kombination aus unterschiedlichen Lastbalancierungsverfahren nun um mehrere Prioritätsklassen erweitert werden. In Abbildung 5.4 ist dieses kombinierte Lastbalancierungsverfahren mit zwei Prioritäten veranschaulicht. Der *Path Dispatcher* trennt eingangsseitig anhand von Header-Informationen wie IP-Adressen oder IPsec-Headern, zustandslosen und zustandsbehafteten Verkehr. Hierbei wird allerdings in aller Regel lediglich eine Vorklassifizierung vorgenommen, also sicher zustandloser Verkehr von möglicherweise zustandsbehaftetem Verkehr getrennt. Eine vollständige Überprüfung ist in der Hardware sehr aufwändig, da z.B. bei IPsec eine Überprüfung der *Security Policy Database* (siehe Abschnitt 2.3.4) anhand des IP 5-Tupel erfolgen muss. Sinnvoller und effektiver ist eine Überprüfung z.B. anhand von IP-Adressbereichen mit potentiell zustandsbehaftetem Verkehr. Als Folge wird u.U. auch zustandsloser Verkehr aus demselben IP-Adressbereich als zustandsbehaftet klassifiziert und dediziert einer CPU nach dem HLU-Verfahren zugeordnet, was funktional allerdings keinerlei Nachteile mit sich zieht. Allerdings erfolgt die Verteilung dann nach dem für diese Pakete suboptimalen, dedizierten Verfahren.

Zustandsbehaftete Pakete werden nach dem HLU-Verfahren dediziert einer CPU übergeben. Die Zuordnung von Flow-Hashwert zu CPU ist dabei intern in einem entsprechenden Speicher im *Path Dispatcher* hinterlegt (siehe [3]). Der eigentliche HLU-Algorithmus wird auf einer *Control Plane*-CPU ausgeführt. Dabei wird in regelmäßigen Abständen die Auslastung der CPUs überprüft und die neue Flowzuordnung berechnet. Die Ergebnisse werden im *Path Dispatcher* abgespeichert und die Pakete in der Folge der entsprechenden CPU-Queue übergeben. Dabei wird zwischen hochprioren und niederprioren Paketen

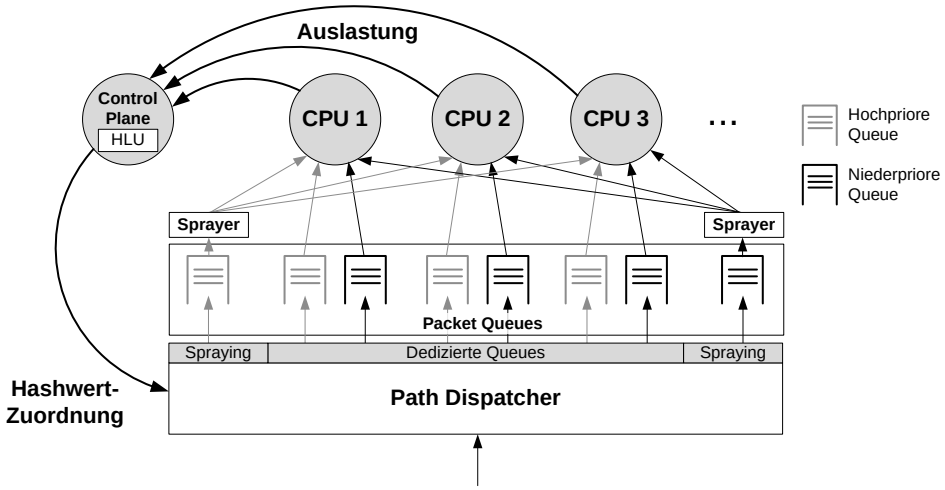

Abbildung 5.4: Kombiniertes, prioritätsbasiertes Lastbalancierungsverfahren im FlexPath-NP.

unterschieden. Hochpriore Pakete werden von der CPU bevorzugt bearbeitet.

Zustandslose Pakete werden je nach Priorität in zwei *Spraying*-Queues aufgeteilt. Von dort werden die Pakete auf die verfügbaren CPUs verteilt. Die niederprioren *Spraying*-Pakete kommen dabei nur zum Zuge, falls keine dedizierten Pakete auf einer CPU verarbeitet werden. Damit werden diese Pakete also bevorzugt von CPUs mit wenig dediziertem Verkehr prozessiert. Die hochprioren Pakete dagegen werden noch vor den dedizierten Paketen mit der höchsten Priorität verarbeitet.

Die effektive Umsetzung der Paketverteilung und damit auch Ausführung des *Sprayings* ist Gegenstand des folgenden Abschnitts 5.3.

5.3 Paketverteilung im Multiprozessor-Cluster

5.3.1 Motivation

In Abschnitt 5.2 wurden unterschiedliche Lastbalancierungsverfahren für den Einsatz in NPs vorgestellt. Hierbei wird für jedes Paket eine Verarbeitungseinheit festgelegt. Dabei kann entweder eine dedizierte Entscheidung getroffen werden, also eine Festlegung auf eine konkrete Verarbeitungseinheit, es kann aber auch ein Pool von Verarbeitungseinheiten ausgewählt werden, auf den die Pakete verteilt werden sollen (*Spraying*). Der *Path Dispatcher* gibt diese Entscheidung in Form einer Pfad-ID aus.

Im Folgenden wird nun untersucht, wie diese Entscheidung möglichst effizient im Multiprozessor-Cluster umgesetzt werden kann. Dabei sind mehrere entscheidende Fragen zu berücksichtigen:

- Wie werden CPUs über ankommende Pakete informiert? Eignet sich hierbei eher

ein interruptbasiertes Verfahren oder ein Polling-Verfahren? Wie wird der Verarbeitungsoverhead aus Sicht der CPU und aus Gesamtsystemsicht minimiert?

- Wie entscheidet sich im Falle von *Spraying*, welche CPU konkret die Prozessierung durchführt? Wie wird insbesondere entschieden, falls mehrere CPUs zur Prozessierung bereitstehen?
- Wie können die unterschiedlichen Prioritäten der Paket-Queues berücksichtigt werden?

5.3.2 Gegenüberstellung von Interrupt und Polling

Bevor eine konkrete Umsetzung der Paketverteilung erfolgen kann, muss entschieden werden, ob ein polling- oder interruptbasiertes Verfahren zur Information der CPUs über ankommende Pakete eingesetzt werden soll. Beide Ansätze wurden bereits kurz in Abschnitt 3.4 beschrieben. Beide Ansätze haben Vor- und Nachteile, die im Zusammenhang mit der konkreten Applikation betrachtet werden müssen.

Beim **Polling** entscheidet die CPU, wann mit dem Beginn einer neuen Aufgabe, bzw. im konkreten Fall der Prozessierung eines neuen Pakets, begonnen werden soll. Nach Abarbeitung eines Pakets kann das nächste Paket angefordert werden. Liegen keine weiteren Pakete vor, findet in regelmäßigen Abständen eine Abfrage statt. Dabei ist die Periodendauer der Abfrage von entscheidender Bedeutung: eine zu lange Periodendauer lässt die CPU evtl. unnötig lange warten und senkt den Wirkungsgrad. Eine zu kurze Periodendauer erzeugt dagegen eine Blindlast im System, insbesondere im Multiprozessor-Cluster mit vielen CPUs. Eine sinnvolle Dimensionierung erfordert die Berücksichtigung von Spitzenlasten. Damit wird jedoch ein Großteil der CPUs im Regelfall nicht benötigt, erzeugt aber dennoch eine Last durch Polling. Polling kann dennoch eine sinnvolle Variante bei rein dedizierten Lastbalancierungsverfahren sein, da nur die CPU-eigenen Paket-Queues überprüft werden müssen. Bei entsprechender separater Anbindung hätten die Anfragen keine Nebenwirkungen auf andere CPUs im Cluster.

Ein **interruptbasiertes System** reagiert schneller auf Ereignisse, respektive ankommende Pakete. Eingehende Interrupts führen bei der CPU dazu, dass die aktuelle Prozessierung unterbrochen wird, um den anstehenden Interrupt zu bearbeiten. Dies ist mit einem entsprechenden Kontextwechsel und damit Overhead verbunden. Die meisten Prozessoren besitzen mehrere Interrupteingänge mit unterschiedlichen Prioritäten. Der beim Virtex-4 verwendete PowerPC 405 unterscheidet kritische und nicht-kritische externe Interrupts. Die Abarbeitung der *Interrupt Service Routine* (ISR) eines nicht-kritischen Interrupts kann durch einen kritischen Interrupt unterbrochen werden. Der Prozessor reagiert innerhalb der ISR aber nicht auf nicht-kritische Interrupts.

Insgesamt scheint die Verwendung von Interrupts aufgrund der schnelleren Reaktion bei ankommenden Paketen dem Polling-Verfahren überlegen. Es muss allerdings verhindert werden, dass die CPU während der Prozessierung eines Pakets zu oft unterbrochen wird. Eine sehr hohe Ereignisrate – wie sie bei der Paketverarbeitung durchaus der Fall ist – führt sonst dazu, dass eine CPU einen Großteil der Zeit mit *Interrupt Handling*, Kon-

textwechsel etc. beschäftigt ist und damit die eigentliche Paketprozessierung behindert wird. Es ist daher sinnvoll, die Prozessierung eines Pakets generell nicht zu unterbrechen. Damit können zwar auch höherpriore Pakete die Prozessierung niederpriorer Pakete nicht unterbrechen. Da es sich aber bei einem Paket um eine zeitlich sehr limitierte Aufgabe handelt, ist der damit verbundene Nachteil minimal. Die Nicht-Unterbrechbarkeit kann durch zwei Möglichkeiten realisiert werden:

- Die CPU deaktiviert den Interrupteingang. Dies geschieht z.B. automatisch während sie sich in der Abarbeitung der ISR befindet, kann aber auch manuell erfolgen.
- Der *Interrupt Controller* berücksichtigt belegte CPUs nicht.

In beiden Fällen kann die Prozessierung nach wie vor durch kritische Interrupts unterbrochen werden.

5.3.3 Multiprocessor Interrupt Controller

Um in einem Multiprozessor-Cluster ankommende Pakete bzw. Tasks mittels Interrupts verteilen zu können, bedarf es einer zentralen Kontrollinstanz – dem *Multiprocessor Interrupt Controller* (MPIntC, siehe Abbildung 5.5). Paket-Queues signalisieren dem MPIntC über eine Interruptleitung das Vorhandensein von Paketen. Zusätzlich können weitere externe System-Interrupts (z.B. Schnittstellen) auf eine nötige Prozessierung hinweisen. Der MPIntC fasst den Status aller aktiven und nicht aktiven Interrupt-Eingänge in einem *Interrupt Status Register* zusammen. Basierend auf diesem *Status Register* entscheidet der MPIntC, welche(r) Prozessor(en) für die Bearbeitung zu benachrichtigen ist/sind (*Interrupt Request*). Der Prozessor reagiert auf den Interrupt üblicherweise mit dem *Interrupt Handling*. Er liest über ein gemeinsames *Interconnect* den Registerinhalt des MPIntC mit den aktiven Interrupts aus und bearbeitet den höchstprioren Interrupt durch Ausführung der zugehörigen ISR. Sofern es sich hierbei um ein Paket-Interrupt handelt, kann dieses aus der entsprechenden Queue gelesen und prozessiert werden.

Konkurrierende Zugriffe im Multiprozessor-System

Das Auslesen des *Interrupt Controllers* in einem **Einprozessor-System** folgt in der Regel dem folgenden Schema (vgl. auch [77]):

1. Die CPU wird mittels *Interrupt Request* benachrichtigt.
2. Sie liest das *Interrupt Status Register* mit allen aktiven Interrupts aus. Daraufhin entscheidet die CPU (sofern mehr als ein Interrupt vorhanden), welcher Interrupt bearbeitet werden soll. Auch wenn dies in der Regel der höchstpriore Interrupt ist, kann theoretisch ein beliebiger Interrupt von der CPU zur Bearbeitung ausgewählt, bzw. alle aktiven Interrupts nacheinander abgearbeitet werden.
3. Das entsprechende Bit im *Interrupt Status Register* wird auf '0' zurückgesetzt und der Interrupt damit wieder deaktiviert. Dies geschieht in der Regel durch eine entsprechende Adresse im *Interrupt Controller*, die nur Auswirkungen auf das

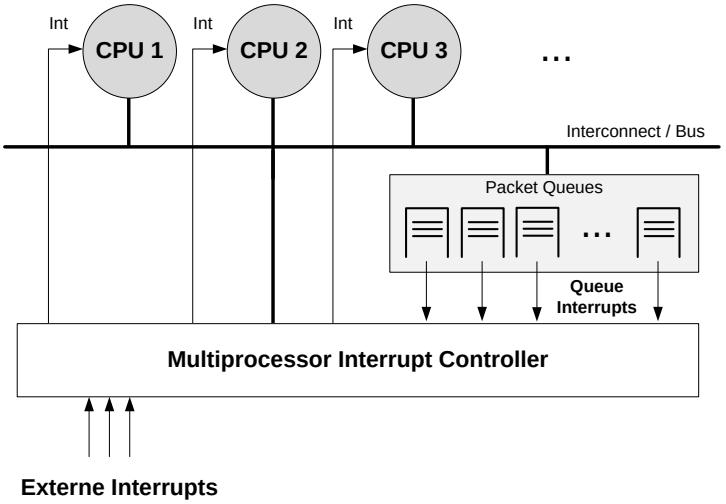

Abbildung 5.5: *Multiprocessor Interrupt Controller* im Multiprozessor-System.

gesetzte Bit hat. Somit wird ein versehentliches Löschen eines seit dem Auslesen neu aktivierten Interrupts verhindert.

Im **Multiprozessor-System** stehen mehrere CPUs zur Bearbeitung von Interrupts zur Verfügung. Somit können – je nach Konfiguration – mehrere CPUs um einen Interrupt konkurrieren. Sofern die Funktionsweise des *Single Processor Interrupt Controllers* übernommen werden würde, könnte es zur Fehlfunktion kommen. Nachdem alle CPUs unabhängig voneinander parallel arbeiten, kann das Auslesen des *Interrupt Status Registers* zeitlich kurz aufeinander folgen. Nachdem die erste CPU einen zweiten Zugriff zum Zurücksetzen des Interrupts benötigt, startet u.U. in der Zwischenzeit fehlerhafterweise eine zweite CPU mit der Bearbeitung desselben Interrupts. Eine praktikable Lösung im Multiprozessor-System fordert also vielmehr einen atomaren Zugriff auf den MPIntC, bei dem das Auslesen und Zurücksetzen des Interrupts in einem Vorgang stattfindet. Das Vorgehen wird deshalb hier wie folgt abgewandelt:

1. Der MPIntC meldet allen in Frage kommenden Prozessoren einen anstehenden Interrupt.
2. Jede freie, informierte CPU versucht, den Registerinhalt des MPIntC auszulesen. Aufgrund des gemeinsamen *Interconnect* geschieht dies bei mehreren CPUs zwangsläufig seriell.
3. Der MPIntC detektiert welche CPU Zugriff erhalten hat und meldet dieser CPU den für sie höchstprioren, aktiven Interrupt. Dieser wird dabei automatisch intern zurückgesetzt.
4. Sofern kein Interrupt mehr zur Bearbeitung bereit steht, eine Abfrage aber schon gestartet wurde, erhält die CPU einen Registerinhalt von 0. Das *Interrupt Handling* kann damit abgebrochen werden. Der MPIntC setzt alle CPU-Interruptleitungen zurück. Zuvor belegte CPUs reagieren damit nicht mehr auf den Interrupt.

Durch dieses Vorgehen entscheidet also der MPIntC, welcher Interrupt von einer CPU bearbeitet werden muss. Die CPU selbst hat keinerlei Wahlmöglichkeit mehr. Dies ist notwendig, um einen zweifachen Zugriff (*read - modify - write-back*) und damit verbundene potentielle Inkonsistenzen zu vermeiden.

Spraying im Multiprozessor-System

Beim *Packet Spraying* stellt sich das Problem, dass eine Queue und damit ein Interrupt-Eingang auf mehrere oder auch alle CPUs zu verteilen ist. Bei Ankunft eines Paketes ergeben sich damit zwei Möglichkeiten:

- Der MPIntC **selektiert eine** freie CPU, die benachrichtigt wird. Alle anderen CPUs erhalten keinen *Interrupt Request*.
- Der MPIntC **informiert alle** in Frage kommenden CPUs. In Abhängigkeit vom Füllstand der Queue erhalten nur die zuerst zugreifenden CPUs einen Interrupt zur Bearbeitung.

Bereits der in Abschnitt 3.4 vorgestellte IBM MPIntC [62] besitzt die Möglichkeit, sogenannte *Distributed Interrupts* zu unterstützen. Dabei kann ein Interrupt auf eine zuvor definierte Menge von CPUs verteilt werden. Der IBM MPIntC realisiert die erste genannte Möglichkeit und wählt aus den zur Verfügung stehenden CPUs eine CPU aus, die mittels *Interrupt Request* benachrichtigt wird. Neben einer verhältnismäßig komplizierten Auswertelogik in Hardware benötigt der MPIntC zur Berechnung v.a. den Kenntnisstand über die jeweilige CPU-Belegung. Dazu müssen die CPUs sich beim MPIntC an- und abmelden. Während das Abmelden durch Zuweisung eines *Interrupt Request* bzw. dem Auslesen des *Interrupt Registers* implizit erfolgen kann, erfolgt die Wiederanmeldung nach Abbarbeitung des Interrupts explizit. Bei der Implementierung des IBM MPIntC ergibt sich zudem das Problem, dass parallel lediglich ein *Distributed Interrupt* pro Interrupt-Eingang im System verarbeitet werden kann.

Um die Komplexität des Controllers klein zu halten, wird hier deshalb die zweite Möglichkeit angewendet: beim *Packet Spraying* werden grundsätzlich alle zugeordneten CPUs per *Interrupt Request* informiert. Da die laufende Bearbeitung von Paketen nicht unterbrochen wird, reagieren darauf nur die jeweils freien CPUs. Wie im Abschnitt zuvor bereits erwähnt, findet der tatsächliche Zugriff auf den MPIntC seriell statt, die erste zugreifende CPU erhält also das Paket. Der Zugriff der restlichen CPUs bleibt dagegen möglicherweise erfolglos. Befinden sich mehrere Pakete in der *Spraying*-Queue so werden diese in der Reihenfolge der restlichen CPU-Zugriffe verteilt. Der Interrupteingang der Queue wird erst zurückgenommen, sobald das letzte Paket aus der Queue verteilt ist. Der MPIntC muss deshalb Kenntnis über die Anzahl der in der Queue vorhandenen Pakete besitzen. Im Gegensatz zum IBM MPIntC kann damit die parallele Bearbeitung von Paketen einer *Spraying*-Queue unterstützt werden. Die Auswertung der Zustände der CPUs in Hardware ist nicht notwendig, was auch die Skalierbarkeit des MPIntC bei wachsender Zahl der CPUs erhöht. Das System reagiert dabei schneller auf einen Burst von ankommenden Paketen, da die Benachrichtigung der einzelnen CPUs nicht seriell

erfolgt, sondern sofort alle zur Verfügung stehenden CPUs benachrichtigt werden.

Es bleibt dennoch festzuhalten, dass ein einzelnes Paket bei mehreren CPUs einen *Interrupt Request* auslöst. Aus Sicht der CPU ist dies nicht weiter entscheidend, da die Prozessierung eines sich bereits in Bearbeitung befindlichen Pakets ja grundsätzlich nicht unterbrochen wird. Beschäftigte CPUs ignorieren den *Interrupt Request* für die Dauer der Prozessierung. Eine möglicherweise erfolglose Abfrage des MPIntC betrifft damit nur freie CPUs, wodurch keine Rechenressourcen verschwendet werden. Lediglich die Auslastung des *Interconnects* wird erhöht. Dennoch halten sich die Einbußen in Grenzen. Die Wahrscheinlichkeit für eine erfolglose Abfrage ist v.a. bei einem nicht ausgelasteten System hoch – wenn also wenige Pakete auf viele freie CPUs stoßen. Dann jedoch ist die Auslastung des *Interconnects* generell sehr gering. Steigt die Auslastung des Systems, sinkt auch die Anzahl der freien CPUs. Damit sinkt die Wahrscheinlichkeit, dass mehrere CPUs gleichzeitig auf einen *Interrupt Request* reagieren und damit eine erfolglose Abfrage provozieren. Andererseits steigt die Wahrscheinlichkeit bei einem hoch ausgelasteten System, dass immer wenigstens ein Paket auch zur Prozessierung bereit steht.

Konfiguration und Interrupt-Zuordnung

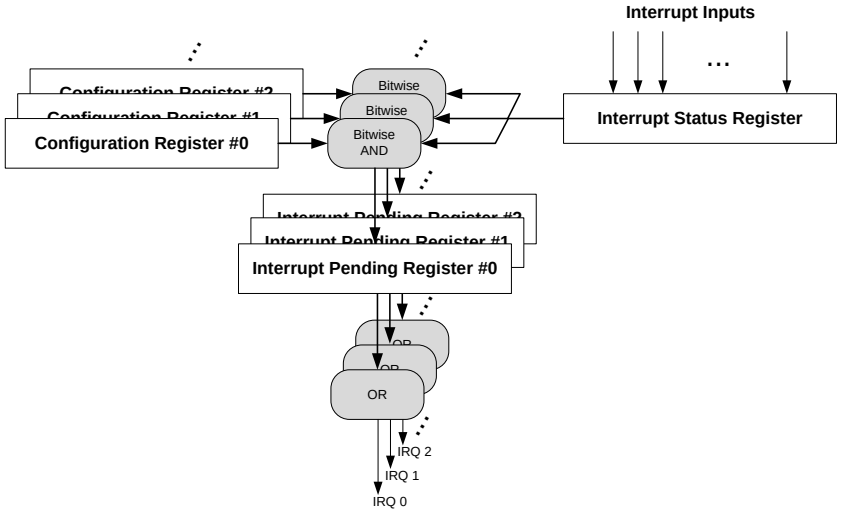

Abbildung 5.6: Interne Registerstruktur des MPIntC.

Die Zuordnung zwischen Interrupt-Eingang und *Interrupt Request* der CPU erfolgt anhand eines durch die *Control Plane* beschreibbaren Konfigurationsregisters (siehe Abbildung 5.6). Der Status der Interrupt-Eingänge ist im *Interrupt Status Register* festgehalten. Pro CPU steht ein Konfigurationsregister zur Verfügung, das alle für die CPU nicht relevanten Interrupt-Eingänge ausmaskiert. Anhand der Konfigurationsregister kann somit eine dedizierte Zuweisung zwischen Interrupt-Eingang und CPU erreicht werden, oder eine Pool-Zuordnung zu mehreren CPUs. Die maskierten Interrupt-Eingänge finden sich im *Interrupt Pending Register*, das für jede CPU also die relevanten, aktiven

Interrupts enthält. Sobald ein Bit des *Interrupt Pending Register* aktiv ist, wird ein *Interrupt Request* (IRQ) an die zugehörige CPU ausgelöst. Jede CPU liest ihr *Interrupt Pending Register* aus, wobei automatisch der höchstpriore Interrupt gelöscht wird.

Die Prioritäten der Interrupts sind statisch. Damit wird die Priorität des Interrupts durch den Anschluss an den MPIntC festgelegt. Die einzelnen Paket-Queues besitzen damit eine feste, ansteigende Priorität. Eine nachträgliche Konfiguration der Priorität, wie sie z.B. beim IBM MPIntC möglich ist, ist nicht vorgesehen.

5.3.4 Packet Distributor

Die Verwendung des MPIntC ermöglicht die Verteilung der Pakete im Multiprozessor-System. Zum Auslesen der Pakete sind dabei zwei Zugriffe einer CPU notwendig – nämlich das Auslesen des *Interrupt Pending Registers* und im zweiten Schritt das Auslesen des Paket-Deskriptors aus der Queue. Der im folgenden beschriebene *Packet Distributor* integriert MPIntC und Paket-Queues in einer Einheit. Gegenüber der in Abbildung 5.5 dargestellten Struktur besitzt er folgende Vorteile:

- Das **Auslesen von Interrupt Pending Register und Paket-Deskriptor kann zu einem Vorgang zusammengefasst werden**. Bei Verwendung eines Burst ist damit nur noch ein Buszugriff notwendig. Durch den Wegfall einer Arbitrierungsphase kann dadurch die Busauslastung gesenkt werden. Dies führt jedoch auch innerhalb der CPU zu einer Beschleunigung der Prozessierung.
- Die **Implementierung von Zählern innerhalb des MPIntC entfällt**. Bei separater Struktur muss der MPIntC über den Füllstand der Queues genau Bescheid wissen, da bei pegelgesteuerten Interrupts der Interrupt der Queue erst beim Auslesen des Paket-Deskriptors – der ja erst zeitlich versetzt in einem zweiten Zugriff erfolgt – zurückgenommen wird. Der MPIntC könnte diesen Interrupt ohne Zähler in der Zwischenzeit fälschlicherweise einer weiteren CPU zuweisen. Durch die Integration der beiden Module entfällt dieser Aufwand, da der Zugriff der CPUs auf MPIntC und Queues atomar und damit streng seriell erfolgt.

Der Aufbau des *Packet Distributor* ist in Abbildung 5.7 gezeigt. Er integriert den MPIntC und die Paket-Queues zu einer Einheit. Der Zugriff einer CPU auf die Queues wird abstrahiert, indem der *Packet Distributor* automatisch den korrekten Paket-Deskriptor bereitstellt. Der Inhalt des *Interrupt Pending Register* wird dem Paket-Deskriptor beim Auslesen des MPIntC vorangestellt. Damit können alle notwendigen Informationen mit einem einzigen Zugriff ausgelesen werden. Anhand des vorangestellten *Interrupt Pending Registers* erkennt die CPU die Interrupt-Nummer und damit den zu verwendenden *Interrupt Handler*.

Der *Packet Distributor* verfügt über eine proprietäre Schnittstelle, mit der Paket-Deskriptoren vom Eingangsdatenpfad in die einzelnen Queues geschrieben werden können. Neben einer festgelegten Anzahl frei zuordenbarer Queues steht ferner eine *AutoRoute-* und *Discard-Queue* zur Verfügung. Es können zudem Paket-Deskriptoren über die Busschnittstelle in die Paket-Queues geschrieben werden. Die Konfiguration und Zuordnung

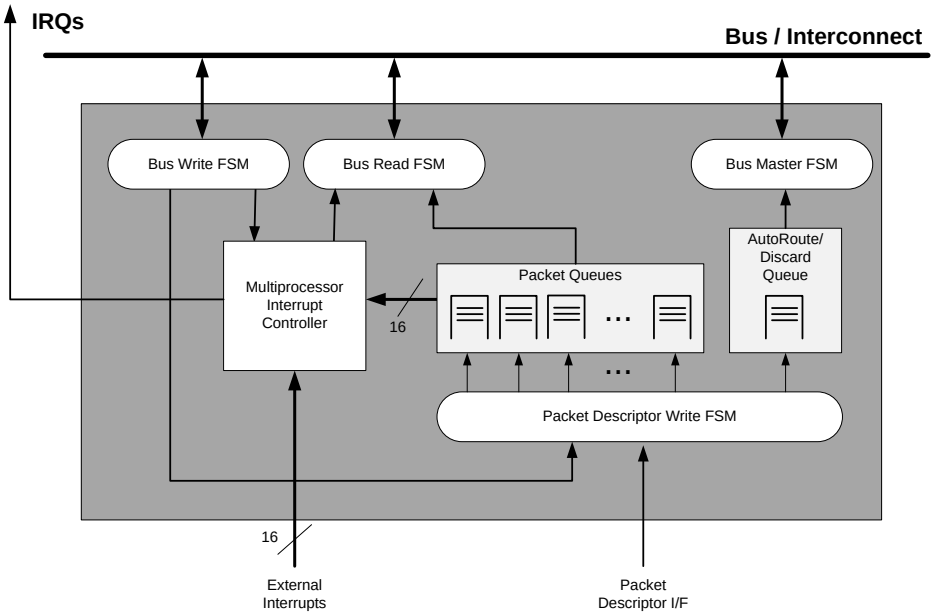

Abbildung 5.7: Architektur des *Packet Distributors*.

zwischen Queue und CPU(s) geschieht dabei über den in Abschnitt 5.3.3 beschriebenen Mechanismus.

Der *Packet Distributor* integriert zusätzlich eine Reihe von Funktionen, die im NP im Allgemeinen und im FlexPath-NP im Speziellen vorteilhaft sind:

AutoRoute und Discard

Mit dem *Path Dispatcher* wird im FlexPath-NP die Möglichkeit geschaffen, Pakete im Eingangsdatenpfad zur reinen Hardwarebearbeitung (*AutoRoute*) zu identifizieren. *AutoRoute*-Pakete werden dabei direkt an den Ausgangsdatenpfad zur weiteren Hardwareverarbeitung gesendet. Es können aber auch Pakete aufgrund bestimmter Filtereinstellungen oder fehlerhafter Integritätschecks eingangsseitig aussortiert werden. Nachdem diese Pakete bereits durch die autonome *DMA Engine* (*SmartMem Buffer Manager*) im Speicher abgelegt wurden, muss der Speicherplatz beim Verwerfen des Pakets wieder freigegeben werden. Dies geschieht durch Übergabe des Paket-Deskriptors an den *SmartMem*. Bei beiden Möglichkeiten muss also der Paket-Deskriptor an eine definierte Adresse gesendet werden, wobei lediglich die Adresse variiert. Der *Packet Distributor* integriert diese Funktion, indem er den entsprechenden Bus Master bereitstellt. Da die Pakete über den Bus generell seriell gesendet werden, wird eine gemeinsame Queue für *AutoRoute* und *Discard* gewählt. Das Ziel wird dabei innerhalb der Queue über ein Flag gespeichert.

Paketübergabe im Multiprozessor-System

In einem paketverarbeitenden System sind des öfteren Paketübergaben von einer Verarbeitungseinheit zu einer anderen notwendig. Dies kann z.B. die Übergabe eines Pakets von der *Data Plane* zur *Control Plane* betreffen (z.B. falls keine gültige Route gefunden wird bzw. das Paket korrupt ist), es kann sich aber auch um die Übergabe eines Pakets zu einem Hardwarebeschleuniger oder zurück handeln. Während beim Hardwarebeschleuniger die Implementierung einer internen Queue möglich ist, ist eine Übergabe an eine CPU schon deutlich aufwändiger. Denkbar ist die Verwendung eines gemeinsamen Speicherbereichs. Der Paket-Deskriptor kann in diesen Speicherbereich geschrieben und die Empfänger-CPU anhand eines Interrupts benachrichtigt werden. Anschließend kann die Empfänger-CPU den Paket-Deskriptor auslesen und verarbeiten. Während dieser Ansatz bei zwei CPUs noch praktikabel ist, nimmt die Komplexität in einem Multiprozessor-System deutlich zu. Da mehrere CPUs einer dedizierten CPU Pakete übergeben können, ist die Verwendung von Queues unumgänglich. Durch eine verhältnismäßig einfache Erweiterung ist es beim *Packet Distributor* möglich, Paket-Deskriptoren über die Busschnittstelle in die Queues zu schreiben. Über eine dediziert zugewiesene Queue kann damit einer bestimmten CPU (z.B. *Control Plane*) ein Paket übergeben werden. Damit ist der *Packet Distributor* logisch nicht mehr allein dem Eingangsdatenpfad zuzuordnen, sondern wird u.U. mehrfach von einem Paket während der Bearbeitung passiert.

Head of Line-Blocking und Auto-Discard

In gepufferten Multiprozessor-Systemen kommt es mitunter zu *Head of Line-Blocking*. Das Problem ist an einem einfachen Beispiel mit vier CPUs und einer dedizierten Queue pro CPU in Abbildung 5.8 veranschaulicht. *Head of Line-Blocking* kann in nicht ausbalancierten Systemen auftreten, wenn eine CPU sich in Überlast befindet und dadurch den ihr dediziert zugewiesenen Verkehr nicht mehr schritthaltend verarbeiten kann. Als Folge dessen füllt sich die Paket-Queue. Nachdem die Queue nun keine Pakete mehr aufnehmen kann, muss beim nächsten für diese Queue bestimmten Paket der Eingangsdatenpfad angehalten werden. Ankommende Pakete werden nun wahllos bereits am Eingang des NPs verworfen. Damit gehen auch Pakete verloren, für die u.U. noch genug Rechenressourcen auf anderen CPUs vorhanden wären. Um dies zu verhindern, übernimmt der *Packet Distributor* eine *Auto-Discard*-Funktion. Sobald eine Paket-Queue überfüllt ist, werden weitere Pakete an diese Queue an die *Discard*-Queue übergeben. Der Eingangsdatenpfad kann somit weiterarbeiten. Der Paketverlust ist damit selektiv und beschränkt sich allein auf die überlastete CPU. Der *Discard* ist nötig, um den bereits belegten Speicher wieder freizugeben und ein Speicherleck zu vermeiden.

Das Verhalten kann in Überlastsituationen jedoch auch negative Auswirkungen haben, nämlich dann, wenn sich das gesamte CPU-Cluster in Überlast befindet. Durch den *Auto-Discard* gelangen auch in dieser Situation Pakete weiterhin ins System und werden durch den *SmartMem Buffer Manager* abgespeichert, nur um im Anschluss wieder gelöscht zu werden. Damit wird zusätzlicher Verkehr innerhalb des Systems erzeugt und die

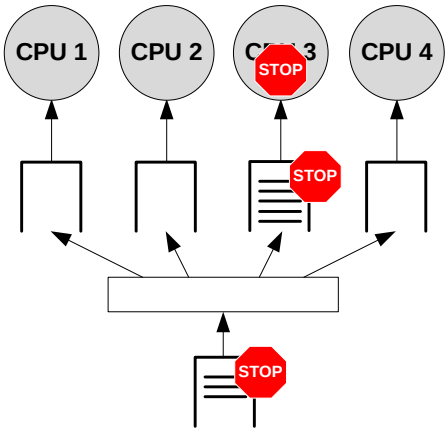

Abbildung 5.8: Effekt von *Head of Line-Blocking* in einem gepufferten Multiprozessor-System.

Abarbeitung behindert. In dieser Sitation wäre ein *Backpressure* und anschließendes Verwerfen der Pakete direkt am Eingang des NPs vorteilhafter.

Architektur

Der *Packet Distributor* besteht, wie in Abbildung 5.7 zu sehen ist, im Kern aus dem MPIntC und den 16 Paket-Queues. Er besitzt neben der proprietären Schnittstelle je eine *Bus-Slave*-Schnittstelle zum Auslesen und Schreiben, sowie über einen *Bus-Master* zum Abbarbeiten der *AutoRoute*- und *Discard*-Queue.

Über den *Read-Slave* können Paket-Deskriptoren ausgelesen werden. Der Slave erlaubt ferner den direkten Lesezugriff auf die Konfigurationsregister des MPIntC. Die *Read Finite State Machine (FSM)* erkennt anhand der Adresse die zugreifende CPU. Beim Auslesen der MPIntC-Register wird der Zugriff transparent durchgeleitet. Soll dagegen ein Paket-Deskriptor ausgelesen werden, wird zunächst das entsprechende *Interrupt Pending Register* im MPIntC gelesen. Anhand des Registerinhalts kann der *Packet Distributor* die abzuarbeitende Queue für den Prozessor erkennen. Der *Packet Distributor* liest daraufhin den Paket-Deskriptor aus der Queue aus und stellt ihn der CPU mit dem Inhalt des *Interrupt Pending Registers* noch im gleichen Lesezyklus bereit. Die CPU kann daraufhin direkt mit der Bearbeitung des Pakets beginnen.

Über den *Write-Slave* können Paket-Deskriptoren in die verschiedenen Queues geschrieben werden. Der *Packet Distributor* ermöglicht dabei sehr effizient die Übergabe von Paket-Deskriptoren von CPU zu CPU bzw. von CPU zu Hardwarebeschleuniger und umgekehrt. Beim Schreiben regelt die *Packet Descriptor Write FSM* den Zugriff auf die Queues. Der Zugriff von der proprietären Schnittstelle und der *Bus Write FSM* muss seriell erfolgen. Bei gleichzeitigen Zugriffen genießt die Busschnittstelle höhere Priorität und die proprietäre Schnittstelle wird für die Dauer des Zugriffes gestoppt. Die *Bus Write FSM* übernimmt zudem den *Auto-Discard* beim Schreiben in eine volle Queue. Über

den *Write-Slave* ist ferner der transparente Schreibzugriff auf die Konfigurationsregister des MPIntC möglich.

Sobald sich ein Paket-Deskriptor in der *AutoRoute/Discard*-Queue befindet, startet die *Bus Master FSM* eine Bus-Transaktion. *AutoRoute*- und *Discard*-Pakete werden durch ein Flag in der Queue unterschieden und an unterschiedliche Adressen geschrieben. *AutoRoute*-Pakete werden direkt an den Ausgangsdatenpfad (*Egress Traffic Manager*) gesendet, *Discard*-Pakete dagegen zum Löschen an die entsprechende *SmartMem*-Schnittstelle.

5.3.5 Externe Interrupts und Zuordnung der Paket-Queues im FlexPath-NP

Der MPIntC besitzt eine limitierte Zahl an Interrupt-Eingängen, die sowohl für Paket-Queues als auch für externe Interrupt-Eingänge verwendet werden können. Die Interrupt-Priorität ist dabei – im Gegensatz zum IBM MPIntC – durch die Reihenfolge des Anschlusses an den MPIntC festgelegt und kann nicht für jede CPU separat konfiguriert werden. Am Beispiel des FlexPath-Systems soll hier kurz gezeigt werden, wie eine sinnvolle Konfiguration der Paket-Queues / Interrupt-Eingänge aussehen kann.

Entsprechend dem Aufbau des Demonstrators im Kapitel 7 wird von zwei CPUs ausgegangen. Als Verkehr kommt reiner IP-Forwarding-Verkehr, sowie zu ver-/entschlüsselnder IPsec-Verkehr zum Einsatz, der bereits vom *Path Dispatcher* separiert wird. Auch wenn das System mit lediglich zwei CPUs noch keinem echten Multiprozessor-System entspricht, können die hier angestellten Überlegungen durchaus auf mehrere Prozessoren übertragen werden. Die beispielhafte Anordnung ist in Abbildung 5.9 dargestellt. Es wird von 16 Queues und 31[1] zur Verfügung stehenden Interrupt-Eingängen ausgegangen. Die Priorität der Interrupts steigt von Eingang 30 (niedrigste Priorität) bis hin zum Eingang 0 (höchste Priorität). Um sowohl hochpriore, als auch niederpriore externe Interrupts im System zu integrieren, wurden die 16 Queue-Interrupts mittig angeordnet. Es empfiehlt sich, am oberen und unteren Ende der Queues jeweils Queues zum *Spraying* vorzuhalten. Damit können sowohl hochpriore *Spraying*-Pakete (mit der Anforderung nach geringer Latenz, z.B. VoIP) verteilt werden, als auch niederpriorer *Best Effort*-Verkehr, der sich auf die jeweils nicht dediziert belegten, freien CPUs im Cluster verteilen soll. Im mittleren Bereich finden sich dediziert zugeordnete Queues – je ein Bereich für CPU 0 und CPU 1. Innerhalb eines dedizierten Bereichs wird zwischen dediziert zugewiesenen Forwarding Verkehr und dediziert zugewiesenen IPsec-Verkehr unterschieden.

Die Verwendung von *Spraying*-Queues und dedizierten Forwarding-Queues ist im Grunde redundant, erlaubt aber – bei identischer *Packet Distributor*-Konfiguration – später die Untersuchung unterschiedlicher Lastbalancierungsverfahren (dedizierte Verfahren vs. *Packet Spraying*). Ebenso scheint zunächst die Unterscheidung zwischen dediziertem Forwarding- und IPsec-Verkehr und damit die Vorhaltung von zwei separaten dedizierten Queues pro CPU überflüssig. Die Queue-Nummer enthält jedoch für die CPU bereits

[1] Die eigentlich 32 Bit werden durch die konkrete Implementierung des *Packet Distributors* um ein Bit reduziert (siehe nächster Abschnitt).

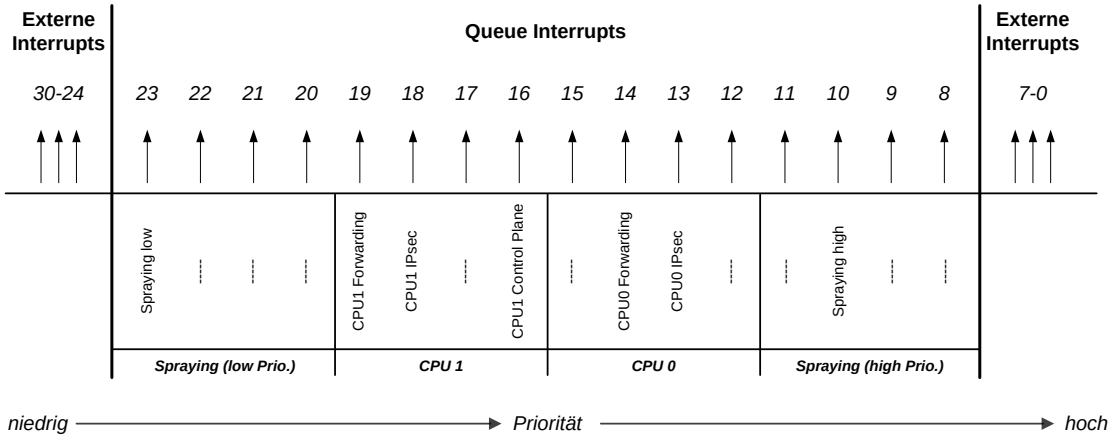

Abbildung 5.9: Anschluss der Interrupt-Eingängen an den MPIntC im FlexPath-NP.

vorab eine Information über den Pakettyp. Dadurch kann direkt der passende, optimierte Software-Code ausgeführt werden, was zu nennenswerten Geschwindigkeitssteigerungen führt (siehe Kapitel 7). CPU 1 übernimmt im FlexPath-Demonstrator zudem die *Control Plane*-Funktionalität. Hierfür wird eine weitere Queue bereitgestellt, die eine höhere Priorität besitzt. Dies garantiert die Ausführung der *Control Plane*-Funktionalität auch bei hohen Lasten. Man erkennt, dass bei diesem Beispiel neun Paket-Queues unbesetzt bleiben. Mit dem gleichen Schema könnten demnach noch vier weitere CPUs bedient werden. Bei einer höheren Anzahl an CPUs müsste die Anzahl der Queues bzw. der MPIntC entsprechend skaliert werden (siehe hierzu auch Skalierbarkeit der Implementierung auf S. 111)

5.3.6 Implementierung

Der *Packet Distributor* und der zugehörige MPIntC wurden auf einem Xilinx Virtex-4 FPGA implementiert. Der *Packet Distributor* ist dabei entsprechend der im Demonstrator maximal zu erwartenden CPU-Clustergröße von vier auf 16 Paket-Queues ausgelegt (vgl. Abschnitt 5.3.5). Die Speichertiefe ist auf max. 16 Paket-Deskriptoren (à 128 Bit) pro Queue konfigurierbar. Ferner ist eine *AutoRoute*- und *Discard*-Queue implementiert. Der Anschluss an die CPUs erfolgt über eine PLB-Schnittstelle. Es werden bis zu 16 CPUs unterstützt. Der *Packet Distributor* erlaubt dabei sowohl 32 Bit- als auch 64 Bit-Zugriffe als Burst oder *Single Access*.

Die Adresstabelle der PLB-Schnittstelle ist in Abbildung 5.10 dargestellt. Sie unterteilt sich im Wesentlichen in einem Bereich zum Auslesen und Schreiben von Paket-Deskriptoren (`0x0000-0x7fff`) und einem Bereich für den Zugriff auf die internen Register des MPIntC zur Konfiguration (`0x8000-0xffff`). Beim Lesen und Schreiben von Paket-Deskriptoren identifiziert sich jede CPU anhand eines separaten Adressbereiches. Beim Schreiben bestimmt der hintere Adress-Offset zudem die Ziel-Queue. Beim Auslesen dagegen wird die Queue automatisch mit Hilfe des MPIntC bestimmt (siehe Ab-

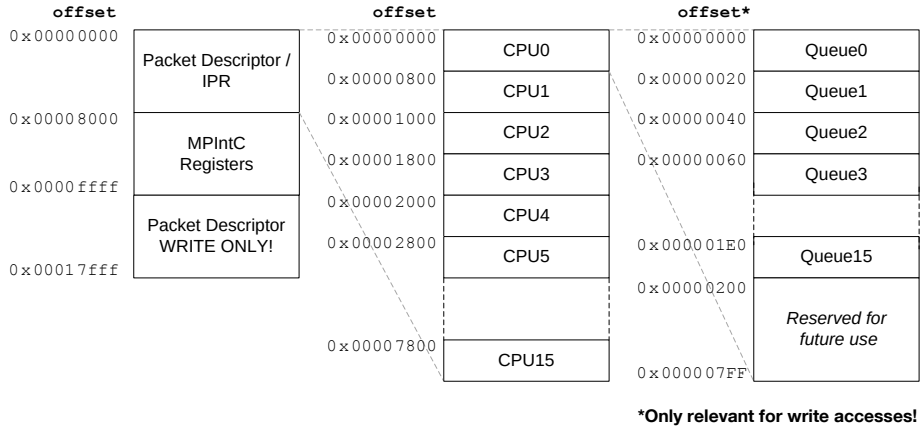

Abbildung 5.10: Adresstabelle des PLB-Interfaces beim *Packet Distributor*.

schnitt 5.3.4). Zum Schreiben der Paket-Deskriptoren existiert zudem ein gespiegelter Bereich am Adress-Offset `0x10000-0x17fff`. Dieser Bereich erlaubt den Einsatz von gecachten Zugriffen durch die verwendete PowerPC-CPU, wobei die volle Busbreite von 64 Bit mit Burstzugriff, anstatt der sonst eingesetzten 32 Bit mit *Single Access* verwendet wird. Beim Einsatz des *Cache Controllers* ergibt sich das Problem, dass beim Anlegen der Cacheline im PowerPC zunächst ein Lesezugriff an die entsprechende Adresse erfolgt. Dies würde vom *Packet Distributor* jedoch fälschlicherweise als geplanter Lesezugriff der CPU gewertet, wodurch – sofern vorhanden – ein Paket-Deskriptor übergeben wird. Sofern kein *Cache Controller* eingesetzt wird, kann der Adressbereich ignoriert werden.

Burst und Single Access

Beim Lesen und Schreiben der Paket-Deskriptoren von 128 Bit plus 32 Bit beim Lesen des *Interrupt Pending Registers* ergibt sich das Problem, dass der Zugriff u.U. auf mehrere Buszugriffe aufgeteilt werden muss. Zwar unterstützt der PowerPC mit *Cache Controller* Burstzugriffe von 32 Byte (256 Bit), jedoch soll der *Packet Distributor* auch andere Prozessoren, u.a. den Xilinx MicroBlaze, unterstützten. Dieser unterstützt jedoch nur 32 Bit-Einzelzugriffe. Damit kann der Paket-Deskriptor nicht mit einem Buszugriff geschrieben/gelesen werden. Aus diesem Grund ist für jede CPU ein temporärer Speicher zum Lesen und Speichern implementiert. Dieser hält die Daten bei Einzelzugriffen, bis der komplette Paket-Deskriptor gespeichert oder gelesen wurde. Das Abspeichern in die jeweilige Queue beim Schreiben erfolgt erst, sobald der Paket-Deskriptor vollständig ist. Der Einsatz des temporären Speichers ist für die CPU transparent.

MPIntC

Der MPIntC basiert auf der Xilinx Implementierung des *Interrupt Controllers* für Einprozessor-Systeme [77] und wurde funktional zum MPIntC erweitert (siehe [78]). Die

Erweiterungen betreffen insbesondere die Implementierung der Konfigurationsregister, die Unterstützung mehrerer CPUs und die Auswertung des jeweiligen höchstprioren Interrupts. Der MPIntC kann als selbstständiges Modul gemäß der originalen Xilinx-Spezifikation sowohl mit DCR-, als auch OPB-Anschluss betrieben werden (vgl. auch Abschnitt 3.4). Er unterstützt sowohl levelsensitive als auch flankengetriggerte Interrupts.

Synthese-Ergebnisse

Tabelle 5.1 gibt einen Überblick über den Ressourcenverbrauch des *Packet Distributors* bei einem System mit zwei CPUs und 24 Interrupt-Eingängen (16 Paket-Queues, acht externe Interrupts). Die Ergebnisse sind aufgeschlüsselt nach MPIntC, dem eigentlichen *Packet Distributor Core* ohne MPIntC und Interfaces, sowie die Summe des vollständigen *Packet Distributors*. Neben den 1.112 Virtex-4 Slices verbraucht der *Packet Distributor* v.a. acht chipinterne SRAM Blöcke (BRAMs). Diese realisieren neben den Paket-Queues auch die temporären Zwischenspeicher, die für Einzelzugriffe notwendig sind.

	MPIntC	**P.Dist. Core**	**Packet Distributor**
Slices	418	537	**1.112**
Flip-Flops	389	434	**1.084**
4-input LUTs	695	1.007	**2.033**
BRAMs	0	8	**8**
Max. f [MHz]	259,0	128,5	**121,9**

Tabelle 5.1: Ressourcenverbrauch beim Packet Distributor auf Xilinx Virtex-4, aufgeschlüsselt nach *Multiprocessor Interrupt Controller*, *Packet Distributor Core* (ohne MPIntC und Bus IP Interface) und Gesamtcore **inkl.** Bus-Interface bei zwei CPUs und 24 Interrupt-Eingängen.

Skalierbarkeit

Auch wenn die Implementierung im FlexPath-Demonstrator zunächst auf zwei CPUs beschränkt ist, ist die Skalierbarkeit ein wichtiges Kriterium bei der Bewertung des *Packet Distributors* für Multi- und Many-Core-Systeme. Dabei haben zwei Parameter entscheidenden Einfluss auf die Größe: die Anzahl der Interrupt-Eingänge und die Anzahl der Interrupt-Requestleitungen (sprich CPUs).

Die Anzahl der Interrupteingänge beeinflusst insbesondere den MPIntC, der die Auswertung der Eingänge übernimmt. In Abbildung 5.11 erkennt man, dass der Ressourcenverbrauch (aufgeschlüsselt nach benötigter Anzahl von Virtex-4 Slices, Flip-Flops und *Lookup-Tables*) mit der Anzahl der Interrupt-Eingänge deutlich ansteigt. Während der Verbrauch an Slices bei zwei CPUs und einem Eingang bei 68 liegt, steigt er über 152 (8 Eingänge) und 325 (16 Eingänge) auf 508 Slices bei 32 Eingängen. Damit ergibt sich ein näherungsweise linearer Anstieg von ca. 14 Slices pro Eingang.

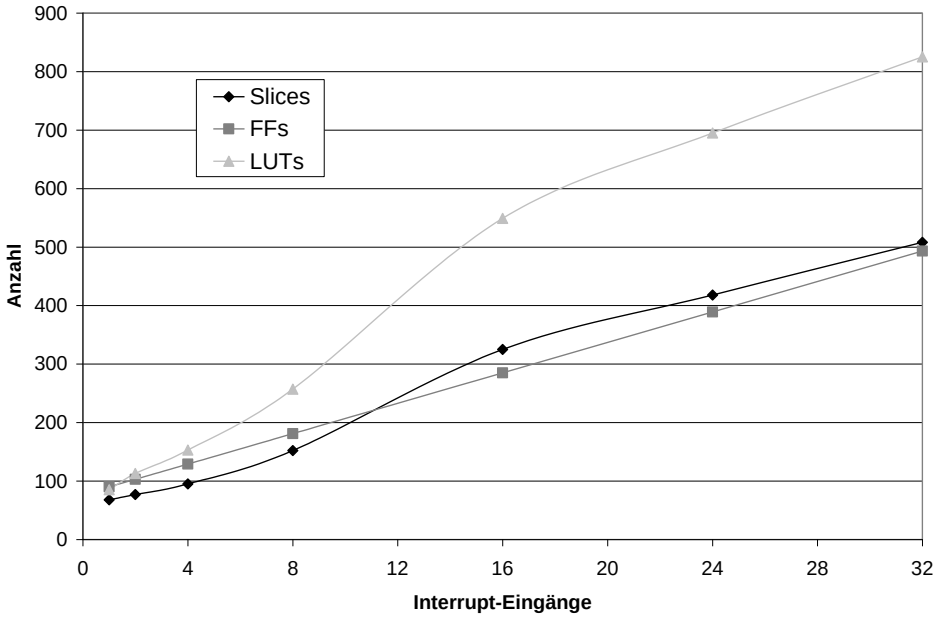

Abbildung 5.11: Skalierbarkeit des MPIntC in Abhängigkeit von den angeschlossenen Interrupteingänge bei zwei CPUs, aufgeschlüsselt nach Virtex-4 Slices, Flip-Flops (FF) und *Lookup-Tables* (LUT).

Für jede CPU muss im MPIntC zusätzlich ein eigener Satz an Konfigurations- und Auswerteregistern implementiert werden. Auch hier ist der Anstieg wie in Abbildung 5.12 ersichtlich näherungsweise linear, wobei die Steigung bei nur 24 Interrupt-Eingängen mit ca. 90 Slices pro CPU niedriger liegt, als bei 32 Eingängen mit ca. 120 Slices pro CPU.

Der Ressourcenverbrauch des *Packet Distributors* (inkl. MPIntC) hat zum eigenständigen MPIntC dagegen einen mehr oder minder festen, von der CPU-Anzahl unabhängigen Zuschlag von rund 550-650 Slices. Die Anzahl der Queues blieb bei den angegebenen Werten konstant bei 16. Eine Erhöhung der Queue-Anzahl hätte – neben den ansteigenden Interrupt-Eingängen und damit Ressourcenverbrauch beim MPIntC – v.a. einen entsprechenden Mehrverbrauch an Speicher zur Folge.

Software-Treiber des Packet Distributors

Die Hardware-Implementierung des *Packet Distributors* wurde noch um einen zugehörigen Software-Treiber für die CPU erweitert. Der Treiber basiert dabei auf dem originalen Treiber des Xilinx *Interrupt Controllers*. Der Treiber muss insbesondere das *Interrupt Handling* übernehmen. Dazu muss bei einem ankommenden Interrupt der *Packet Distributor* ausgelesen und die zugehörige *Interrupt Service Routine* gestartet werden. Die Datenstruktur für einen 32 Bit-Lesezugriff ist in Tabelle 5.2 dargestellt. Anhand des ersten Bits im ersten Datenwort erkennt der Treiber, ob es sich um einen Paket-Deskriptor oder um einen externen Interrupt handelt. Bei einem externen Interrupt kann sofort nach

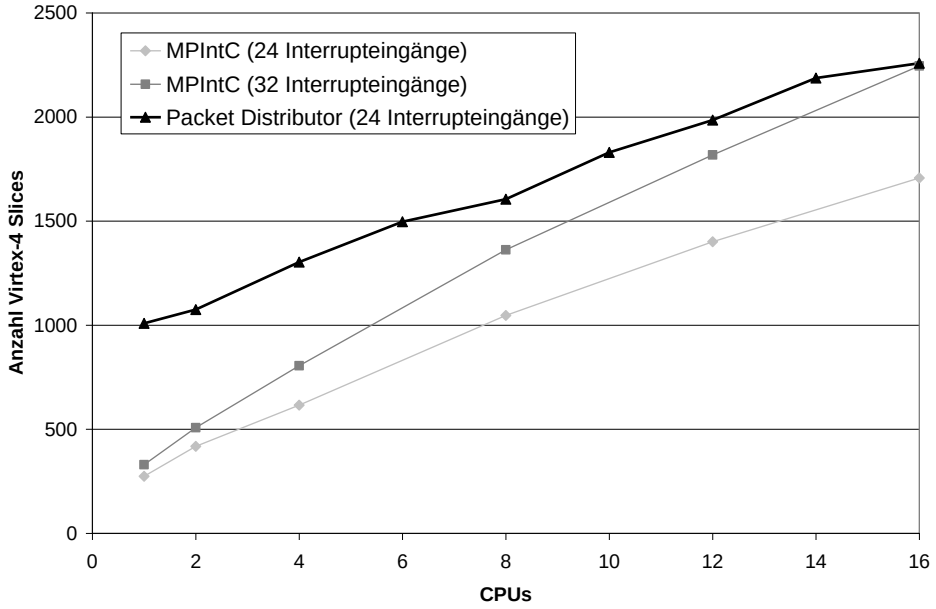

Abbildung 5.12: Skalierbarkeit des MPIntC und des *Packet Distributors* bei variabler Anzahl an CPUs mit 24 bzw. 32 Interrupt-Eingängen. Angegeben ist der Verbrauch in Virtex-4 Slices.

dem Auslesen des ersten Datenworts die entsprechende *Service Routine* gestartet werden. Im Falle eines Paket-Deskriptors werden zusätzlich die nachfolgenden Datenwörter ausgelesen und der *Service Routine* als Paket-Deskriptor übergeben.

Der Treiber ist außerdem für die korrekte Adressumsetzung anhand der jeweiligen CPU-ID zuständig. Er stellt der CPU zudem eine Reihe von Zugriffsfunktionen zum *Packet Distributor* bereit, um z.B. die Konfiguration des MPIntC, die Übergabe oder das Auslesen von Paket-Deskriptoren innerhalb der Software zu erleichtern.

Offset	31	30 0
0x0000	PD/IRQ	CPUx IPR
0x0004	Paket-Deskriptor [127:96]	
0x0008	Paket-Deskriptor [95:64]	
0x000C	Paket-Deskriptor [63:32]	
0x0010	Paket-Deskriptor [31:0]	

Tabelle 5.2: Struktur des Interrupt Pending Registers und Paket-Deskriptors beim Auslesen aus dem *Packet Distributor*. Das erste Bit (`PD/IRQ`) definiert, ob es sich um ein Paket-Deskriptor oder Interrupt handelt.

5.4 Konzeptevaluierung

In diesem Abschnitt werden die Eigenschaften und Vorteile der FlexPath-Lastbalancierungsstrategie anhand eines Simulationsmodells untersucht und mit den aus der Literatur bekannten Verfahren verglichen. Entsprechend dem Schwerpunkt dieser Arbeit wird dabei ein besonderes Augenmerk auf das *Spraying* gelegt. Dazu wird zunächst das Verhalten von *Spraying* im Vergleich zu anderen Lastbalancierungsverfahren bei zustandslosem Verkehr untersucht. Im einem zweiten Schritt wird zusätzlich beleuchtet, welche Effekte und Auswirkungen *Spraying* im kombinierten FlexPath-Verfahren mit zustandslosem und -behaftetem Verkehr besitzt.

Für die Untersuchungen wird ein funktionales SystemC-Simulationsmodell verwendet (siehe Abschnitt 5.4.1). Für eine realistische Stimulation des Modells mit aussagekräftigen Ergebnissen zu den Balancierungsverfahren muss ein möglichst realistischer Verkehr verwendet werden. Künstlich erzeugter Verkehr besitzt nämlich i.A. den Nachteil, dass die Anzahl der enthaltenen Flows und auch die zeitliche Abfolge (z.B. Bursts) nicht hinreichend genau modelliert werden kann. Dies ist aber bei der Überprüfung der Lastbalancierungsverfahren essentiell. Aus diesem Grunde werden reale Verkehrsmitschnitte aus Internet-*Backbone*-Knoten verwendet (siehe Abschnitt 5.4.2). Die Ergebnisse der Simulation werden in Abschnitt 5.4.3 vorgestellt.

5.4.1 Funktionales Simulationsmodell

In Abschnitt 4.4 wurde bereits ein *Trace*-basiertes SystemC-Simulationsmodell vorgestellt, das zur Abschätzung der Geschwindigkeitssteigerung im FlexPath-NP verwendet wurde. In diesem Modell stand v.a. die zeitliche Abfolge der einzelnen Transaktionen (z.B. Speicherzugriffe) im Vordergrund, das funktionale Verhalten wurde dagegen weitestgehend abstrahiert. Für die nun vorliegenden Untersuchungen spielt die Gesamtsystemleistung eine untergeordnete Rolle. Wichtig ist dagegen die funktionale Modellierung aller in die Lastbalancierung involvierten Module.

Der Aufbau des funktionalen Simulationsmodells ist in Abbildung 5.13 dargestellt. Die einzelnen Module besitzen jeweils ein Paket-Interface und je nach Modul ein Konfigurations-Interface. Die Pakete (bestehend aus Paketdaten und einer Reihe von Statusinformationen wie Pfadentscheidung etc.) werden dabei über das Paket-Interface von Modul zu Modul übergeben. Die einzelnen Module können auf die Paketdaten zugreifen und paketabhängige Funktionen ausführen. Nach einer festen oder paketabhängigen Latenz wird das Paket an das nächste Modul übergeben.

Mit dem Modell sollen insbesondere zwei unterschiedliche Lastbalancierungsverfahren verglichen werden. Aus der Literatur wird das adaptive Lastbalancierungsverfahren nach Kencl und Shi (HABS, siehe Kapitel 3.2.2) implementiert. Dieses bietet – wenngleich bei hoher Komplexität – die besten bekannten Resultate. Dem gegenübergestellt wird das in Kapitel 5.2 vorgestellte kombinierte Verfahren aus HLU für zustandsbehafteten Verkehr und *Spraying* für zustandslosen Verkehr. Der *Path Dispatcher* übernimmt dabei die Ausführung des *Lookups* im Referenzszenario mit HABS. Das *Burst Shifting* wird

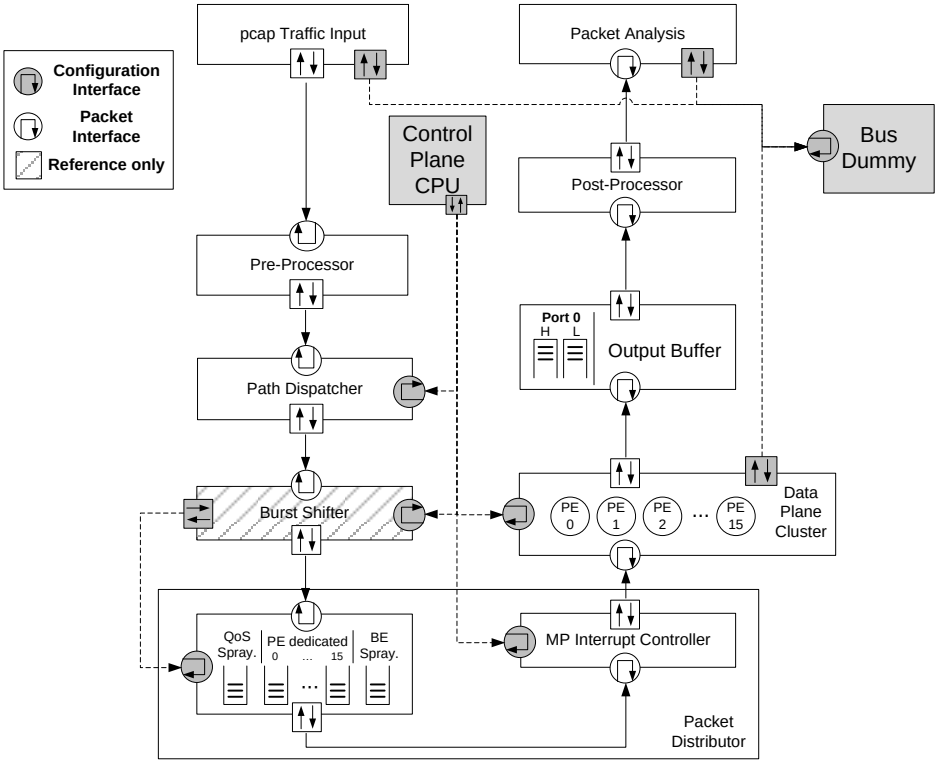

Abbildung 5.13: Funktionales SystemC-Simulationsmodell.

im Referenzfall in einem separaten Modul durchgeführt. Im Falle von FlexPath separiert der *Path Dispatcher* den Verkehr in hoch- und niederprioren zustandslosen Verkehr. Ferner wird für zustandsbehafteten Verkehr der HLU-*Lookup* ausgeführt und die Pakete werden in separate, dedizierte CPU-Queues aufgeteilt. Die unterschiedlichen Adaptionsroutinen für Referenz- und FlexPath-Szenario werden in regelmäßigen Intervallen auf einer *Control Plane*-CPU ausgeführt. Als Zeitintervall werden 50 ms gewählt.

Die Pakete werden aus dem *Packet Distributor* auf ein *Data Plane*-Cluster verteilt. Das Cluster ist konfigurierbar mit einer CPU-Anzahl von bis zu 16 CPUs. Die Prozessierungslatenzen wurden aus einer initialen Softwarestack-Implementierung am FlexPath Demonstrator gewonnen (siehe Kapitel 7). Für IPv4-Forwarding wurde dabei eine Verarbeitungsdauer von 10 μs pro Paket gemessen. Die Verarbeitung (inkl. Verschlüsselung) von IPsec-Paketen ist dagegen paketlängenabhängig und kann mit

$$t_{proc,IPsec} = 310\mu s + \frac{Paketlänge}{64Byte} \cdot 112\mu s \tag{5.1}$$

der Messung angenähert werden. Konkurrierende Buszugriffe beim Abspeichern und Auslesen des Pakets in den Speicher (RX/TX), sowie der Zugriff der CPUs auf die Daten werden durch ein vereinfachtes Bus-Modell abstrahiert. Prozessierungsschwankungen

aufgrund von Cache-Verdrängungen im Instruktions-Cache, Ressourcenkonflikten etc. werden abstrahiert modelliert. 20% der Pakete erhalten dazu einen Aufschlag auf die Prozessierungsdauer mit einem Faktor 1,5 und weitere 10% einen Faktor 2. In der CPU wird dabei nur die Prozessierungszeit anhand des Paketinhalts bestimmt. Eine tatsächliche Bearbeitung des Pakets findet im Simulator nicht statt. Ausgangsseitig werden die Pakete auf hoch- und niederpriore Queues verteilt. Der folgende *Post-Processor* erzeugt letztlich eine der Verarbeitungs-Pipeline entsprechende Latenz ohne konkrete Verarbeitung. Eine Analyse der Pakete (z.B. Gesamtsystemlatenz) findet schließlich am Ausgang des Systems statt.

5.4.2 Verkehrs-Stimuli

Zur Stimulierung des Simulators wurden Verkehrsmitschnitte von realen *Backbone*-Routern verwendet. Ein realistischer Verkehr ist zur Bewertung der Lastbalancierungsverfahren v.a. daher wichtig, da neben den reinen Datenraten und Paketgrößen v.a. eine realistische Verteilung der Flows inkl. der zeitlichen Abfolge für die Verfahren eine wichtige Rolle spielen. Die Mitschnitte liegen im *pcap*-Format (siehe [75]) vor, einem standardisierten Format zum Abspeichern von Verkehrsmitschnitten.

Es wurden zwei unterschiedliche Quellen ausgewählt, die von der *Cooperative Association for Internet Data Analysis* (CAIDA) zu Forschungszwecken zur Verfügung gestellt werden. Der erste Datensatz basiert auf anonymisierten Mitschnitten eines OC-48 Links [79]. Die Mitschnitte aus dem Jahr 2002 haben eine Dauer von jeweils 5 Minuten. Die Auslastung des Links liegt bei lediglich ca. 30% und damit deutlich unter den anvisierten 3,2 GBit/s (32 Bit @ 100 MHz) des FlexPath-Demonstrators (siehe Kapitel 7). Um die Last unter Beibehaltung der Flowcharakteristika (zeitliche Abstände zwischen zwei Paketen eines Flows, Datenrate pro Flow etc.) zu steigern, wurden deshalb vier Mitschnitte mit je 15 Minuten Abstand zu einem kombinierten Mitschnitt addiert. Der kombinierte Mitschnitt wurde auf eine Minute und maximal 3,2 GBit/s begrenzt. Er wird im Folgenden als *OC48_mux* bezeichnet (siehe auch Tabelle 5.3, Zeile 1).

Der zweite Datensatz, ebenfalls von CAIDA veröffentlicht, stammt aus dem Jahr 2008 und wurde an einem OC-192 Link, jeweils separat für jede Richtung aufgenommen (siehe [80]). Die erste Richtung (Chicago A) wurde aufgrund der sehr geringen Last nach dem gleichen Verfahren wie zuvor kombiniert (*OC192_mux*, Tabelle 5.3, Zeile 2). Die zweite Richtung (Chicago B) weist dagegen eine vergleichsweise hohe Auslastung mit Spitzenraten von 9 GBit/s und mehr auf. Der Mitschnitt wurde um den Faktor vier verlangsamt und auf 3,2 GBit/s limitiert (*OC192_quarter*, Tabelle 5.3, Zeile 3).

Wie aus Tabelle 5.3 ersichtlich, setzen sich alle drei Verkehrsdateien mehrheitlich aus *Best Effort*-Verkehr mit einem Anteil von ca. 92-96% zusammen. Dieser muss ohne konkrete Anforderungen an Latenz o.ä. als IPv4-Forwarding-Verkehr bearbeitet werden. Bei weiteren 4-7% handelt es sich um hochprioren Verkehr (ToS>0, siehe Abschnitt 2.3.5), der beim Forwarding bevorzugt behandelt werden muss. Bei einem vergleichsweise geringen Anteil von 0,6% und kleiner handelt es sich um IPsec-Pakete, die im Simulationsmodell als zu entschlüsseln zu betrachten sind und somit als Beispiel für eine zustandsbehaftete

Mitschnitt	Pakete	ØDatenrate	IPsec	QoS	BE	t	Ref
OC48_mux	22.086.716	1,955 GBit/s	0,07%	4,14%	95,79%	60s	[79]
OC192_mux	41.223.895	2,819 GBit/s	0,40%	4,04%	95,56%	60s	[80]
OC192_quarter	26.473.646	1,320 GBit/s	0,63%	7,39%	91,98%	120s	[80]

Tabelle 5.3: Kenndaten zu den verwendeten Verkehrsmitschnitten (IPsec: IPsec-Verkehr, QoS: hochpriorer Verkehr, BE: *Best Effort*-Verkehr, t: Tracelänge).

Verarbeitung dienen.

Die Verwendung von IPsec-Verkehr eines *Backbone*-Routers birgt eine gewisse Ungenauigkeit, da Ver- und Entschlüsselung in der Realität v.a. im *Access*- bzw. *Edge*-Bereich zu finden sind. Im *Backbone*-Router findet in der Realität lediglich ein Forwarding des IPsec-Verkehrs statt. Wenngleich die Protokollverteilung und Flow-Charakteristik als Kombination vieler *Edge*-Verbindung betrachtet werden kann und somit im Durchschnitt eine sinnvolle Verteilung angibt, kann der Anteil von IPsec-Verkehr bei einzelnen *Edge*-Routern deutlich höher liegen. Dennoch wird hier aufgrund der Nichtverfügbarkeit von entsprechenden Mitschnitten aus dem *Access*- und *Edge*-Bereich das Vorgehen als sinnvolle und gültige Näherung betrachtet.

5.4.3 Simulationsergebnisse

Im Folgenden werden die Simulationsergebnisse im Detail vorgestellt, die sich schwerpunktmäßig auf das für diese Arbeit relevante *Spraying* konzentrieren. Konkret werden die folgenden Untersuchungen durchgeführt:

- Die ersten Simulation beschränken sich auf **zustandslosen Verkehr**. Hierbei wird untersucht, welche Auswirkungen *Spraying* im Hinblick auf Verlustraten und Latenz besitzt. Dabei wird *Spraying* mit dem aus der Literatur bekannten HABS-, aber auch mit dem im Rahmen von FlexPath entwickelten HLU-Lastbalancierungsverfahren verglichen.
- Als nächstes wird die **Auswirkung der Eingangsqueuegröße auf Spraying**, wiederum im Vergleich mit HLU und HABS, untersucht.
- Bereits in Abschnitt 5.2.1 wurde angedeutet, dass **Packet Reordering** potentiell ein Problem bei *Spraying* sein kann, wenngleich die erwarteten Zahlen deutlich niedriger als z.B. beim *Spraying* nach Dittmann sein sollten. Eine erste Untersuchung soll das Ausmaß des *Packet Reordering*-Problems bei *Spraying* zeigen. Die Ergebnisse dienen als Entscheidungsgrundlage für weitere Maßnahmen (z.B. Resequenzierung).
- Schließlich soll das **kombinierte FlexPath-Verfahren** aus *Spraying* (für zustandslosen Verkehr) und HLU (für zustandsbehafteten Verkehr) dem HABS-Verfahren bei gemischtem Verkehr gegenübergestellt werden. Besonderes Augenmerk liegt hierbei auf dem Einfluss von *Spraying* auf das Gesamtsystemverhalten.

Die Untersuchungen werden zunächst auf den in Abschnitt 5.4.2 beschriebenen OC48-Mitschnitt (*OC48_mux*) durchgeführt und im Detail beschrieben. Die grundlegenden Aussagen werden anschließend für die anderen Mitschnitte überprüft.

Verlustraten und Latenz bei zustandslosen Verkehr (IP-Forwarding)

Die erste Simulationsreihe beschränkt sich auf zustandslosen *Best Effort*-Verkehr. Hierbei werden alle Pakete zum CPU-Cluster gesendet und dort als Forwarding-Pakete prozessiert. Zustandsbehaftete Pakete werden hierzu als zustandslos betrachtet. Prioritäten werden nicht unterschieden. Die Simulationsreihen werden jeweils mit einer ansteigenden Anzahl an CPUs (1-16) im Cluster durchgeführt. Zu Beginn wird die Tiefe der einzelnen Queues im *Packet Distributor* auf 32 Pakete festgelegt. Es wird dabei die Eignung der drei Lastbalancierungsverfahren HABS, HLU und *Spraying* separat untersucht.

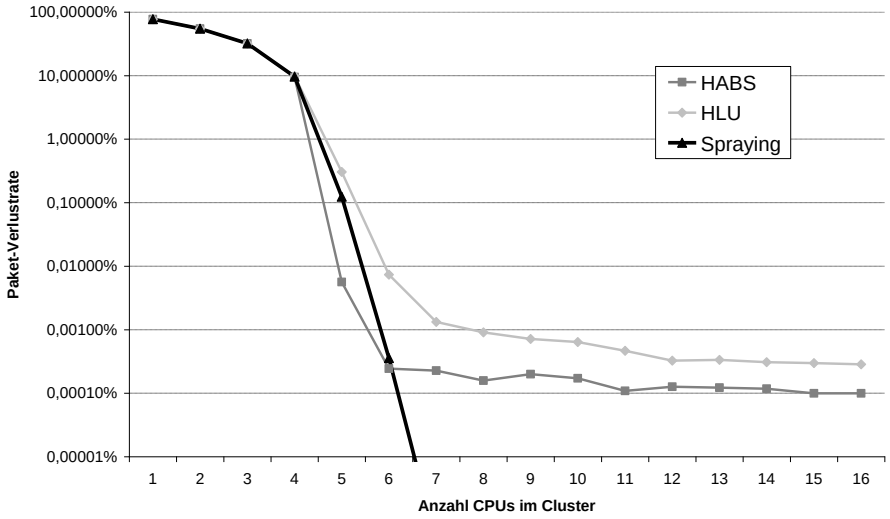

Abbildung 5.14: Paket-Verlustraten für unterschiedliche Verfahren bei variierender CPU-Anzahl im Cluster für *Best Effort*-Verkehr.

In Abbildung 5.14 sind die Verlustraten der einzelnen Verfahren für eine unterschiedliche Anzahl an vorhandenen CPUs dargestellt. Es zeigt sich, dass sich das System bei vier und weniger Prozessoren praktisch durchgehend im Überlastbereich befindet, was in entsprechend hohen Verlustraten von ca. 10% und mehr resultiert. Das Ergebnis unterscheidet sich aus diesem Grunde für alle drei Verfahren nicht. Erst ab fünf Prozessoren zeigen sich die Unterschiede der Lastbalancierungsverfahren. *Spraying* erreicht ab sieben Prozessoren eine verlustfreie Verarbeitung. Wie erwartet erreicht es damit eine optimale Auslastung des Systems. Im ersten Moment erstaunlich scheint, dass bei fünf und sechs CPUs die Verlustraten entgegen der Erwartung höher liegen als bei HABS. Erklären lässt sich dies durch die de facto unterschiedliche Eingangsqueuegröße. *Spraying* verwendet für alle CPUs zusammen lediglich eine gemeinsame Paketqueue mit maximal 32 Paketen, während HABS und HLU für jede CPU eine separate Queue (und damit mehr Spei-

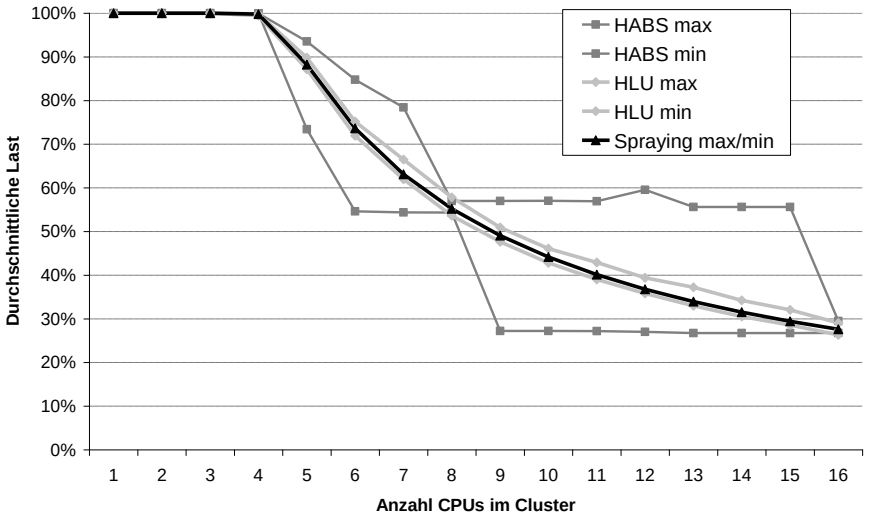

Abbildung 5.15: Maximale und minimale durchschnittliche Last der einzelnen CPUs im Simulations-System für unterschiedliche Verfahren bei variierender CPU Anzahl für *Best Effort*-Verkehr.

cherplatz) belegen. Damit können mitunter größere Schwankungen ausgeglichen werden, ehe es zu Paketverlust kommt. Ab sieben CPUs reicht, wie beim *Spraying* ersichtlich, die Rechenleistung im Cluster aus, um den gesamten Verkehr verlustfrei zu bearbeiten. Dennoch erreichen HLU und HABS selbst für mehrere CPUs keine vollständige Verlustfreiheit und stagnieren bei niedrigen Verlusraten von über 0,0001%.

Damit zeigt sich in den Messungen ein Nachteil der adaptiven Verfahren: der Regelalgorithmus besitzt eine gewisse Trägheit, der auf kurzfristige Schwankungen im Verkehr teils zu langsam reagiert. Somit kann sich eine CPU kurzzeitig im Überlastbereich befinden, was zu Paketverlust führt, sobald die zugehörige Paket-Queue vollläuft – selbst wenn im Cluster in Summe noch ausreichend Reserven vorhanden sind. Die Unausgeglichenheit unter den CPUs zeigt sich in Abbildung 5.15, wo die jeweils minimale und maximale durchschnittliche Auslastung der einzelnen CPUs für unterschiedliche Cluster-Größen dargestellt ist. Gerade bei HABS schwankt die Auslastung zwischen zwei CPUs im Extremfall um den Faktor 2, während HLU bereits viel angeglichenere Auslastungen erreicht und *Spraying* offensichtlich identische Auslastungen für alle CPUs bewirkt. Dies erkennt man auch beim zeitlichen Verlauf der Auslastung am Beispiel von fünf CPUs in Abbildung 5.16. Die Lastwerte sind jeweils gemittelt über eine Sekunde und enthalten damit keine kurzfristigen Lastspitzen. Bei HABS (a) erkennt man sehr gut, wie der Algorithmus das anfängliche Ungleichgewicht auszugleichen versucht. Die Lasten der einzelnen CPUs nähern sich mit der Zeit dem Durchschnitt, wenngleich HABS eine gewisses Ungleichgewicht toleriert. Bei HLU (b) befinden sich die CPU-Lasten durchwegs näher am Durchschnitt als bei HABS, dennoch liegt die Verlustrate im Schnitt höher als bei HABS, das auch kurzfristige Lastspitzen mithilfe des *Burst Shifters* abfedern kann. Lediglich bei *Spraying* gleichen sich die Lasten der CPUs dem Verfahren entsprechend

optimal an. Eine lokale Überlast auf einer CPU ist nicht möglich.

Neben der Verlustrate kann die Systemlatenz als weiteres Qualitätskriterium bei der Paketverteilung gelten. In Abbildung 5.17 ist die durchschnittliche Paketlatenz für die drei Verfahren dargestellt. Die Überlastfälle für weniger als vier Prozessoren werden dabei nicht weiter betrachtet. HABS und HLU erzeugen in etwa vergleichbare Systemlatenzen, die von 90-100 μs bei fünf CPUs zu 18 μs bei 16 CPUs konvergieren. Wiederum zeigen sich die negativen Auswirkungen der beiden Verfahren bei hohen Systemlasten: lokale Ungleichgewichte führen zu einem Anschwellen einzelner CPU-Queues und damit ansteigender Latenz. Die Pakete verbringen also im Schnitt wesentlich mehr Zeit in den Queues als notwendig. Dies zeigt sich insbesondere im Vergleich mit *Spraying* bei dem das erste wartende Paket immer sofort verarbeitet wird, sobald im Cluster eine CPU zur Verfügung steht. Die Latenz liegt bei fünf CPUs demnach mit nur 27 μs wesentlich niedriger. Der Vorteil von *Spraying* schrumpft hier mit steigender CPU-Zahl und damit anwachsender Rechenleistung im System.

Auswirkungen der Eingangs-Queuegrößen auf die Balancierungsverfahren

Bereits bei den Abbildung 5.14 zu Grunde liegenden Untersuchungen hat sich angedeutet, dass unterschiedliche Eingangs-Queuegrößen Auswirkungen auf die Verlustraten besitzen. Kleinere Queues erhöhen die Gefahr, dass kurzzeitige Lastspitzen nicht rechtzeitig abgearbeitet werden können und es bei voller Eingangs-Queue zu Paketverlusten kommt. Größere Queues gleichen Lastspitzen besser aus und senken die Verlustrate, erhöhen bei hoher Auslastung jedoch die Latenz durch lange Wartezeiten in den dann vollen Queues.

Der Effekt der Queuegrößen auf die Verlustrate ist in Abbildung 5.18 dargestellt. Es wurde bereits erwähnt, dass für *Spraying* nur eine einzelne Queue vorgehalten werden muss, während bei HABS und HLU eine Queue pro CPU zu implementieren ist. Um die Verfahren dennoch sinnvoll miteinander vergleichen zu können, wird hier die kumulierte Eingangs-Queuegröße (also Queue-Größe für *Spraying*, bzw. Queue-Größe mal CPU-Anzahl bei HABS und HLU) verglichen. Für die Simulationen wird ein Cluster mit sechs CPUs gewählt, das – wie zuvor gesehen – bei dem gewählten Verkehr eine insgesamt hohe Systemlast erzeugt und dabei selbst für *Spraying* zunächst noch nicht ganz verlustfrei arbeitet. Die Verlustraten sind – wie zuvor – bei *Spraying* am niedrigsten. Ab einer Queuegröße von 48 Paketen kann *Spraying* nun auch mit sechs CPUs verlustfrei arbeiten. HABS benötigt dagegen weit größere Eingangspuffer um vergleichbare Raten zu erzielen und erreicht bei einer Queuegröße von 384 Paketen (6 x 64 Pakete) eine Verlustrate von 0,00001%. HLU liegt bei gleicher Queuegröße mit 0,0015% wenngleich auf niedrigem Niveau noch deutlich darüber.

Gravierender stellt sich der Anstieg der Latenz bei HABS und HLU mit steigender Queuegröße dar (siehe Abbildung 5.19). Bei *Spraying* ist der Anstieg äußerst moderat und stagniert bei einer Queuegröße über 48 Paketen bei 18 μs. Schließlich wird bereits dort Verlustfreiheit erreicht und eine noch größere Paketqueue im laufenden Betrieb zu keinem Zeitpunkt genutzt. Bei HABS und HLU steigt die Latenz von anfangs rund 20 μs

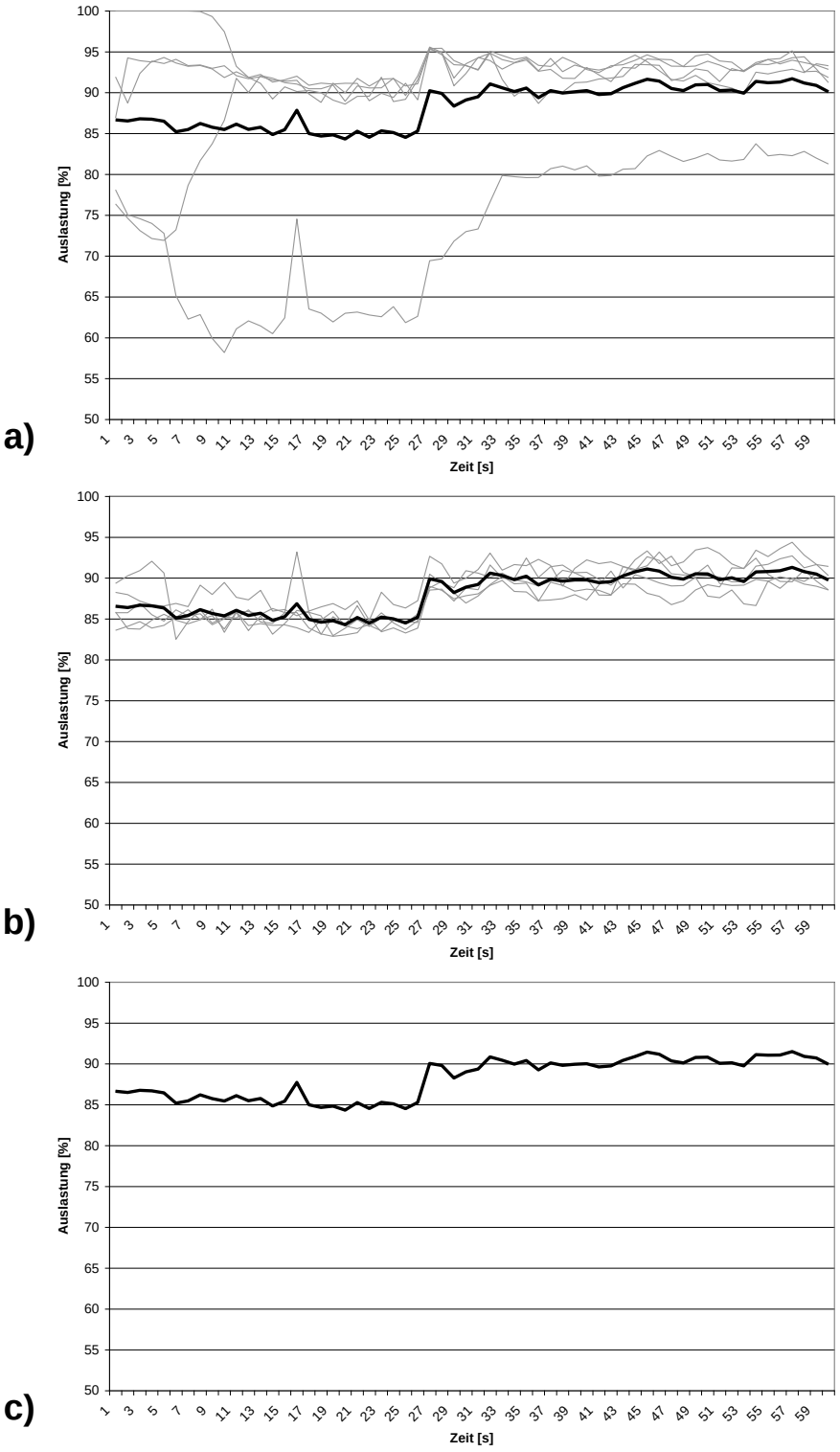

Abbildung 5.16: Zeitliche Auslastung der einzelnen CPUs bei *Best Effort*-Verkehr und fünf CPUs jeweils für die einzelnen CPUs (dünne Linie) und als Durchschnitt über alle CPUs (dicke Linie) für unterschiedliche Lastbalancierungsverfahren: HABS (a), HLU (b) und *Spraying* (c).

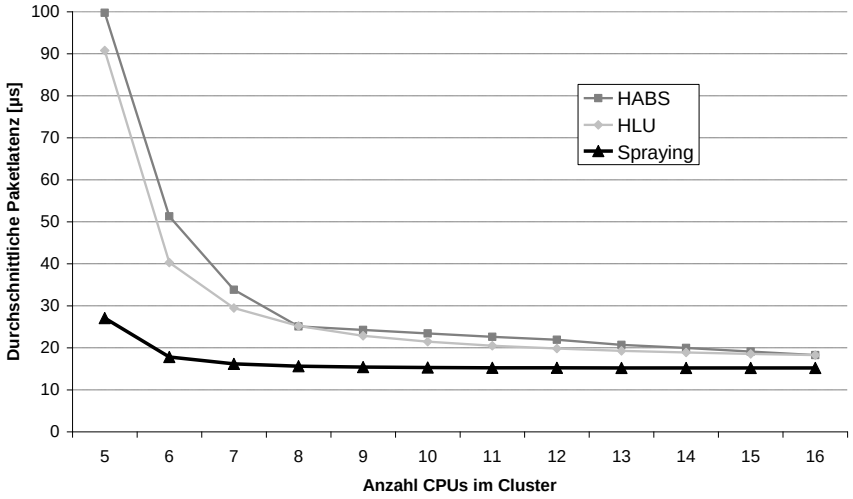

Abbildung 5.17: Paketlatenzen im System für unterschiedliche Verfahren bei variierender CPU-Anzahl im Cluster für *Best Effort*-Verkehr.

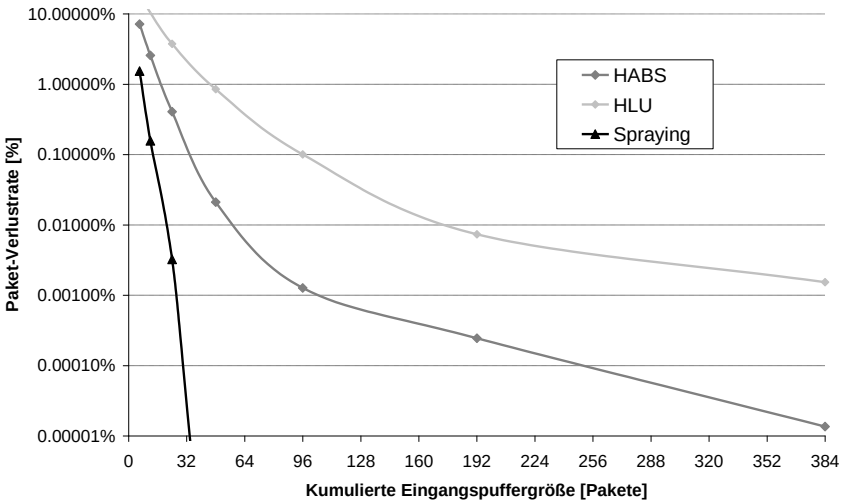

Abbildung 5.18: Paket-Verlustrate für unterschiedliche Eingangs-Queuegrößen.

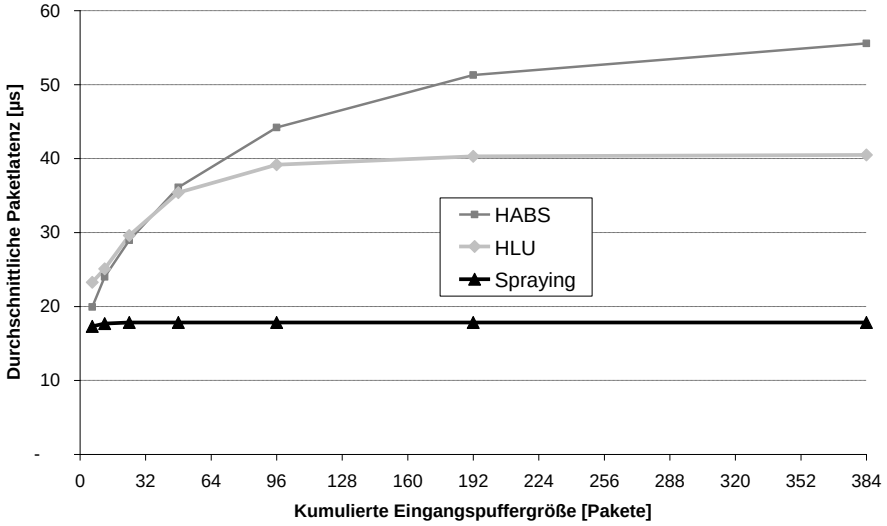

Abbildung 5.19: Durchschnittliche Paket-Latenzen für unterschiedliche Eingangs-Queuegrößen.

(kumulierte Queuegröße: sechs), auf rund 40 μs (HLU), bzw. 56 μs (HABS) bei einer Queuegröße von 384.

Packet Reordering bei Spraying

In Abschnitt 5.2.1 wurde bereits erwähnt, dass *Spraying* zu *Packet Reordering* führen kann, wenngleich vergleichsweise geringe Raten zu erwarten sind. Für eine erste Quantifizierung des Problems wurde die *Packet Reodering*-Rate in Abhängigkeit von der Anzahl der eingesetzten CPUs simuliert (siehe Abbildung 5.20). Es fällt auf, dass im Überlastbereich (vier CPUs und weniger) die Rate vergleichsweise gering ausfällt. Die geringe Anzahl an CPUs und die hohe Auslastung dämmen das *Packet Reordering* weitestgehend ein. Erst ab fünf CPUs und damit zeitweise freien Rechenressourcen im Cluster steigt die Rate sprunghaft auf 0,11% an. Vermutlich spielt hier eine Rolle, dass häufiger zwei oder mehr CPUs gleichzeitig zur Prozessierung bereit sind, womit in sehr kurzen Abständen die Bearbeitung zweier Pakete desselben Flows gestartet werden kann. Der zugrundeliegende Prozesierungsjitter kann dann zur Überholung führen. Der Wert stagniert bei mehr als sechs CPUs bei ca. 0,13%. Der Grund hierfür liegt in der Tatsache, dass ab sechs/sieben CPUs die meiste Zeit mindestens eine CPU bei jedem ankommenden Paket frei sein dürfte – die zusätzlichen CPUs werden also schlichtweg nicht benutzt. Weitere CPUs haben damit weder einen positiven Effekt auf die Verarbeitung, noch einen negativen auf das *Packet Reodering*.

Die *Reordering*-Rate ist damit zwar am unteren Ende der von Laor ([15], vgl. auch Abschnitt 2.3.3) genannten Grenze von 0,1%-1%. Dennoch ist der Wert nicht vernachlässigbar – zumal wenn man bedenkt, dass sich der Effekt beim Durchlaufen mehrerer Router

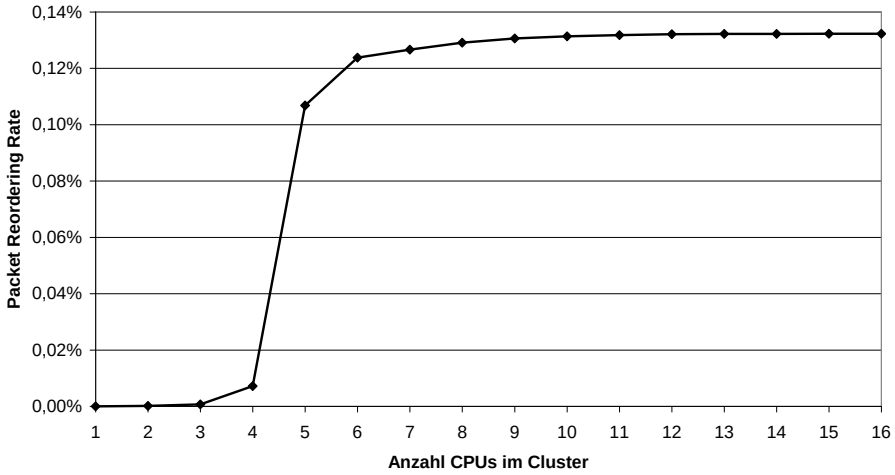

Abbildung 5.20: *Packet Reodering* bei *Spraying* bei variabler CPU-Clustergröße.

verstärkt. Es deutet sich also hier bereits an, dass entsprechende Gegenmaßnahmen getroffen werden müssen. Für weitere Untersuchungen und ein geeignetes Resequenzierungsverfahren wird auf Kapitel 6 verwiesen.

Für HABS und HLU sind die gemessenen Werte erwartungsgemäß dem Verfahren entsprechend niedrig. Während bei HLU kein *out-of-order* Paket gemessenen wurde, erreichte HABS bei fünf CPUs einen Höchstwert von 19 Paketen (0,00009%). Die Werte sind damit praktisch vernachlässigbar.

Kombiniertes Flex-Path-Lastbalancierungsverfahren bei Verkehrs-Mix

Im Folgenden wird das in Kapitel 5.2 vorgestellte FlexPath-Lastbalancierungsverfahren aus *Spraying* und HLU anhand einer Kombination aus zustandslosem und zustandsbehaftetem Verkehr untersucht und mit HABS verglichen. Hierbei wird v.a die Auswirkung des *Sprayings* auf die Lastbalancierung diskutiert. Der HLU-Algorithmus wird dagegen in [3] näher betrachtet.

Bei der Verarbeitung in der CPU werden zu diesem Zweck Forwarding-Pakete und IPsec-Pakete unterschieden und die Prozessorlatenz entsprechend dem vorgestellten Simulationsmodell in Abschnitt 5.4.1 berechnet. Während HABS keinerlei Differenzierung der Pakete vornimmt, wird im FlexPath-System *Best Effort*-Verkehr, IPsec und hochpriorer Verkehr separiert. *Best Effort* und hochpriorer Verkehr wird zwei unterschiedlichen Queues zugewiesen und jeweils mittels *Spraying* im Cluster verteilt (siehe auch Abschnitt 5.3.5). IPsec-Pakete werden mit Hilfe des HLU-Algorithmus verteilt.

Die Simulationen zeigen, dass die Anzahl der Paketverluste beim FlexPath-Verfahren generell unter denen von HABS liegen. Bei variierender CPU-Clustergröße wird ab neun CPUs eine verlustfreie Prozessierung erreicht, während HABS eine Restverlustrate von 0,05% auch bei 16 CPUs nicht unterschreitet. Die durchschnittlichen Prozessierungs-

latenzen aufgeschlüsselt nach den drei Pakettypen sind in Abbildung 5.21 dargestellt. IPsec-Pakete haben für beide Verfahren eine sehr hohe Verarbeitungslatenz von ca. 2 ms. Aufgrund der hohen Prozessierungsdauer spielen Queuing- und Balancierungseffekte eine untergeordnete Rolle. Es sei hier angemerkt, dass die erzielten Verarbeitungslatenzen für IPsec-Pakete im zugrundeliegenden Demonstrator für NPs relativ hoch liegen. In kommerziellen Produkten werden die rechenintensive Ver-/Entschlüsselung in aller Regel durch entsprechende Hardwareunterstützung beschleunigt.

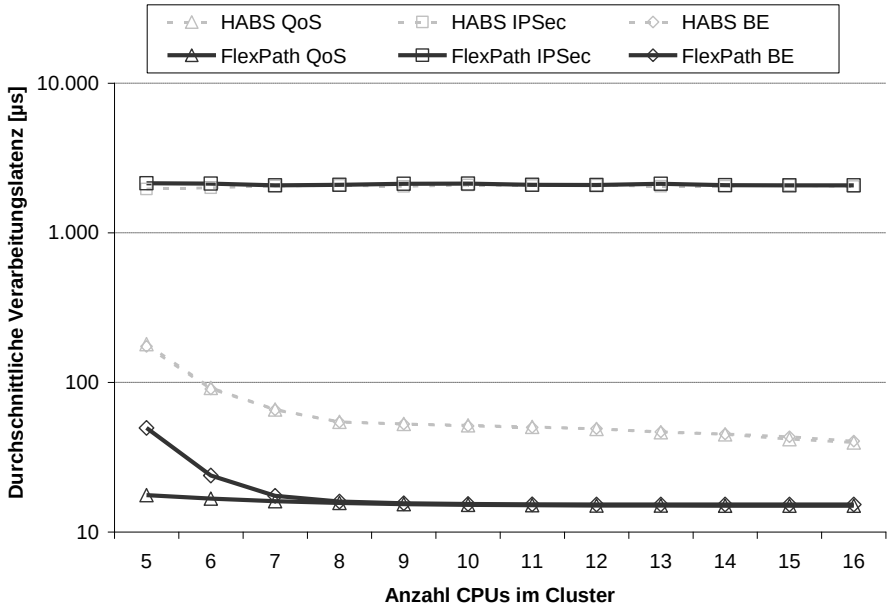

Abbildung 5.21: Durchschnittliche Verarbeitungslatenzen für unterschiedliche Pakettypen bei HABS und FlexPath (*Spraying* und HLU) in Abhängigkeit von der Cluster-Größe.

Für die Forwarding-Pakete erzielt das HABS-Verfahren für hoch- und niederpriore Pakete gleiche Paketlatenzen. Dies ist naheliegend, da diese in den Eingangsqueues nicht unterschieden werden und eine bevorzugte Behandlung der hochprioren Pakete nicht stattfindet. Es zeigt sich, dass die Latenz im Vergleich zu FlexPath deutlich höher liegt. Bei HABS teilen sich Forwarding- und IPsec-Pakete die gleichen Queues. Damit müssen die Forwarding-Pakete bei Blockierung der CPU durch ein IPsec-Paket mitunter recht lange auf ihre Prozessierung warten. Bei FlexPath können die Pakete dagegen eine freie, nicht von IPsec besetzte CPU benutzen. Die hochprioren Pakete profitieren bei FlexPath zudem insbesondere im Überlast-Fall (fünf CPUs) durch die Priorisierung. Während die Latenz der *Best Effort*-Pakete ansteigt, bleibt die Latenz der hochprioren Pakete nahezu konstant niedrig.

Der Einfluss von *Spraying* auf das Gesamtsystemverhalten ist in Abbildung 5.22 dargestellt. Es handelt sich hierbei um den zeitlichen Verlauf der CPU-Auslastungen für CPU 0-3 aufgeschlüsselt nach den drei Verkehrstypen in einem Cluster mit insgesamt acht

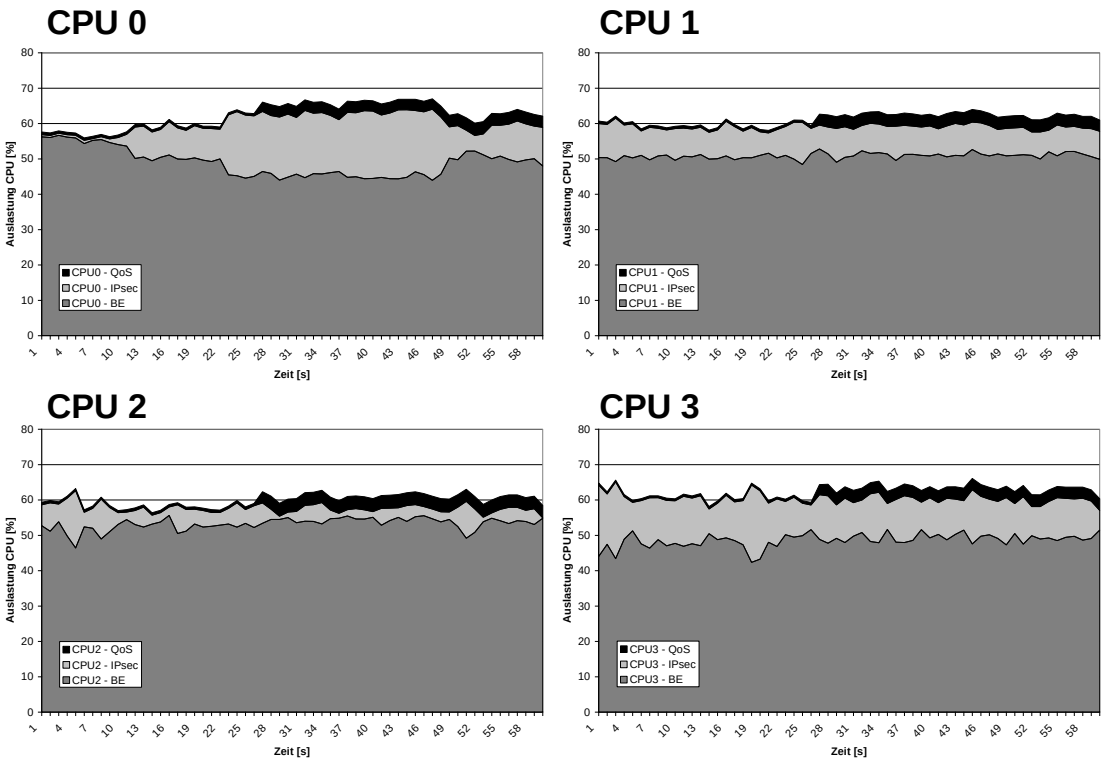

Abbildung 5.22: Kumulierte zeitliche Auslastung unterschiedlicher CPUs mit *Spraying* und HLU im Verkehrs-Mix.

CPUs. Vor allem an CPU 0 erkennt man sehr gut, dass der in der Priorität höher eingestufte IPsec-Verkehr den *Spraying*-Verkehr bei Bedarf verdrängt. Während der IPsec-Anteil ansteigt, sinkt automatisch der *Spraying*-Anteil, der sich auf die restlichen CPUs verteilt. Der in etwa zur Mitte der Simulation verstärkt einsetzende, hochpriore *Spraying*-Verkehr wird ebenso in etwa gleichmäßig auf alle CPUs verteilt. Durch den automatischen Ausgleich des *Spraying*-Verkehrs wird zudem der Lastbalancierungsalgorithmus des zustandsbehafteten Verkehrs effektiv entlastet. HLU arbeitet in diesem Fall nicht auf der Gesamtlast der CPU, sondern berücksichtigt nur die durch zustandsbehafteten Verkehr (hier also IPsec) erzeugte Last. Aufgrund der aus Sicht des HLUs reduzierten Lasten wird eine Umbalancierung unwahrscheinlicher – jede CPU besitzt bei entsprechenden Schwankungen also mehr Reserven für den zustandsbehafteten Verkehr.

Simulationsergebnisse für unterschiedliche Verkehrsmitschnitte

Um die erhaltenen Ergebnisse auf eine fundierte Basis zu stellen, wurden die Simulationen für den kombinierten Verkehr mit den restlichen in Abschnitt 5.4.2 beschriebenen Verkehrsmitschnitten durchgeführt. Für eine sinnvolle Dimensionierung wurde die Clustergröße so gewählt, dass gerade noch Verlustfreiheit für den *Spraying*-Verkehr erreicht

wurde. Es hat sich gezeigt, dass die zuvor gemachten Aussagen auch mit anderem Verkehr Bestand haben (siehe Tabelle 5.4): Bei allen drei Simulationen war die Verlustrate bei der Kombination von *Spraying* und HLU (FP) mit maximal 0,001% deutlich niedriger als die Verlustrate bei HABS (max. 0,48%). Dabei konnten für die Forwarding-Pakete zudem mit Werten um 16 μs für alle Mitschnitte deutlich niedrigere Latenzen erreicht werden als bei HABS mit Latenzen von bis zu 160 μs. Zudem beschränken sich die Paketverluste bei *Spraying* und HLU auf den dedizierten IPsec-Verkehr, während der hochpriore Verkehr vollständig bearbeitet werden konnte.

Mitschnitt	CPUs	Alg.	Verlust	QoS Lat.	IPsec Lat.	BE Lat.
OC48_mux	10	FP	0,0000%	15,18 μs	2.131 μs	15,38 μs
		HABS	0,1311%	51,60 μs	2.044 μs	52,62 μs
OC192_mux	16	FP	0,0002%	15,92 μs	3.122 μs	16,14 μs
		HABS	0,2865%	66,93 μs	1.586 μs	69,61 μs
OC192_quarter	15	FP	0,0010%	15,27 μs	3.896 μs	15,06 μs
		HABS	0,4792%	159,95 μs	2.125 μs	157,64 μs

Tabelle 5.4: Durchschnittliche Verarbeitungslatenzen (Lat.) und Verlustraten bei unterschiedlichen Verkehrsmitschnitten aufgeschlüsselt nach Verkehrsklassen jeweils für die Kombination aus *Spraying* und HLU (FlexPath, FP) und HABS.

5.5 Bewertung

In diesem Kapitel wurde mit *Spraying* ein Lastbalancierungsverfahren für zustandslosen Verkehr vorgestellt. Es konnte gezeigt werden, dass *Spraying* ein sehr einfaches und effektives Verfahren ist. Allerdings erzeugt es in gewissen Maße *Packet Reordering*, welches durch eine anschließende Resequenzierung (siehe nächstes Kapitel) wieder behoben werden sollte.

Da sich *Spraying* nur für zustandslosen Verkehr eignet, empfiehlt es sich für zustandsbehafteten Verkehr aus Gründen der Datenkonistenz der Zustandsinformationen ein dediziertes Verfahren wie HLU oder HABS zu verwenden. Innerhalb des FlexPath-NPs kann mit Hilfe des *Path Dispatchers* der Verkehr eingangsseitig in zustandslos und zustandsbehaftet aufgeteilt werden. Damit kann *Spraying* mit einem dedizierten Verfahren kombiniert werden.

Die Vorklassifizierung des Verkehrs erlaubt die ansonsten unübliche Verwendung von Eingangspuffern. Hochpriore Pakete werden bereits am Eingang separiert und in priorisierte Queues übergeben. Ein selektives Löschen von Paketen überlasteter Routen verhindert effektiv *Head of Line-Blocking*.

Zur Umsetzung der Lastbalancierungsstrategien Bedarf es einer Einheit zur Verteilung der Pakete im Multiprozessor-System. Der vorgestellte *Packet Distributor* dient, neben der Verteilung der Pakete aus dem Eingang, zudem zum Paketaustausch zwischen CPUs

(z.B. *Control* und *Data Plane*), aber auch zwischen Hardware-Beschleuniger und CPUs. Durch die Unterstützung mehrere Queues pro CPU kann außerdem der für das jeweilige Paket optimale Softwareeinsprungpunkt direkt gefunden werden. Es konnte gezeigt werden, dass die vorgestellte Implementierung für eine steigende Anzahl an CPUs oder auch Queues skaliert werden kann. Auch wenn der *Packet Distributor* primär für ein interruptbasiertes System entworfen wurde, kann er ohne Anpassungen in einem Polling-System eingesetzt werden. Schließlich bearbeitet der *Packet Distributor* jederzeit eine Paket-Anfrage einer CPU – unabhängig von einem zuvor ausgelösten Interrupt oder dem tatsächlichen Vorhandensein eines Paket-Deskriptors.

Das effektive Zusammenspiel von *Spraying* und HLU konnte an einem Simulationsmodell gezeigt werden. *Spraying* arbeitet dabei selbst-adaptiv und verteilt die Pakete auf die jeweils am wenigsten ausgelastete CPU. Nachdem nur noch ein Teil der Pakete dediziert zugewiesen wird, können zudem die Anforderungen an den HLU-Algorithmus gesenkt werden. Dies setzt jedoch das Vorhandensein von genügend *Spraying*-Verkehr voraus. Während man hiervon in einem typischen Internet-Verkehr ausgehen kann, ist dies im *Access*-Bereich nicht zwangsläufig gegeben. Im ungünstigsten Fall (kein *Spraying*-Verkehr) konkurriert dann der HLU-Algorithmus direkt mit den anderen vorgestellten Verfahren. Allerdings sind hierbei für den HLU kaum Nachteile erkennbar.

6 Paket-Resequenzierung im FlexPath-NP

6.1 Motivation

Die Einhaltung der originalen Paketreihenfolge eines Flows ist, wie in Abschnitt 2.3.3 beschrieben, für viele, insbesondere aber für das mit Abstand dominante TCP-Protokoll von entscheidender Bedeutung. Bereits vermeintlich sehr kleine *Reordering*-Raten in jedem Router beeinflussen die erreichbare Netzleistung erheblich. Dabei verstärkt sich der Effekt insbesondere dadurch, dass jedes Paket auf dem Weg zum Ziel mehrere Router durchläuft. Es gilt daher, *Packet Reordering* bestmöglich zu verhindern. Dazu können in einem NP zwei verschiedene Ansätze gewählt werden:

- **Pakete eines Flows benutzen (weitestgehend) denselben Pfad**. Dies wird z.B. in hashbasierten Lastbalancierungsverfahren praktiziert, indem jedem Flow eine feste Prozessierungseinheit zugeordnet ist. Nur in Überlastsituationen werden Umbalancierungen vorgenommen, die – in der Regel sehr wenige – Paketüberholungen bewirken können.
- **Pakete werden nach der Prozessierung wieder sortiert**. Damit ergeben sich deutlich mehr Freiheitsgrade bei der Verteilung der Pakete im System und eine erhöhte Gesamtsystemleistung. Andererseits erfordert die Resequenzierung einen Mehraufwand an Ressourcen.

In Kapitel 5 konnte gezeigt werden, dass mit *Spraying* ein sehr effektives, selbstadaptives Lastbalancierungsverfahren entwickelt wurde, das für einen Großteil der Pakete – nämlich den zustandslosen Anteil – angewendet werden kann. *Spraying* zieht aber in gewissem Maße *Packet Reordering* nach sich. *Packet Reordering* stellt dabei ganz allgemein ein Problem für Systeme dar, bei denen – wie im FlexPath-NP – Pakete innerhalb des Systems unterschiedliche Pfade benutzen.

In Abschnitt 3.3 wurden bereits diverse Mechanismen zur Wiederherstellung der Paketreihenfolge vorgestellt. Die Resequenzierung in Software ist dabei aufwändig und bindet einen verhältnismäßig großen Teil der Rechenressourcen im CPU-Cluster. Zudem sind die meisten Verfahren auf eine globale Resequenzierung ausgelegt – die unterschiedlichen Prozessierungslatenzen zwischen den Flows werden nicht berücksichtigt. Dies ist aber insbesondere in einem Umfeld mit heterogenem Applikations-Mix problematisch, da langsame Pakete schnellere Flows aufgrund der Resequenzierung behindern.

Hardwarebasierte Resequenzierungsverfahren für NPs sind hingegen kaum bekannt. Lediglich ein Verfahren (Wu et al. [61]) wurde näher vorgestellt. Es ermöglicht eine flowbasierte Resequenzierung und eignet sich daher auch in einer heterogenen Applikations-Umgebung. Das CPU-Cluster selbst erfährt durch die Hardware-Implementierung kei-

nerlei Geschwindigkeits-Einbußen. Dennoch besitzt das Verfahren nach Wu eine Reihe von Nachteilen, die es für viele NP-Implementierungen nur bedingt einsatzfähig macht:

- Das Verfahren besitzt ein **zentrales Management**. Jedes ankommende Paket muss in den Datenstrukturen registriert und am Ausgang ausgetragen werden. Die zentrale Managementstruktur bedarf daher eines Zugriffs von beiden Seiten sowie einer kommunikativen Verflechtung von Ein- und Ausgang.
- Für jeden Flow im System muss ein neuer Eintrag angelegt werden, ebenso **benötigt jedes Paket im System einen Eintrag** in der verwendeten Block-Tabelle. Deren Größe muss ausreichend dimensioniert werden. Ist dies nicht der Fall, so führt dies, wie in den durchgeführten Tests gezeigt wurde, zur zeitweisen Blockierung der CPUs.
- Das Verfahren benötigt mit CAM, SRAM, Thread-Tabelle und Block-Tabelle eine Vielzahl von z.T. **komplizierten Speicherstrukturen**, die synchron verwaltet werden müssen.
- Ausgangsseitig durchforstet ein *Round Robin*-Suchalgorithmus die Tabellen nach versandfähigen Paketen. Damit werden die Pakete nicht – sofern es die Paket-Sequenz erlaubt – sofort nach der Bearbeitung versendet, sondern je nach Suchzeit später. Damit ergibt sich eine **erhöhte Paketlatenz**. Die Suche wird zudem mit ansteigender Größe der Block-Tabelle zeitintensiver.
- Das Verfahren erlaubt **keine Paketverluste**. Jedes eingehende Paket muss das System wieder verlassen. Nur so werden die entsprechenden Datenstrukturen angepasst. Ein verlorenes Paket würde zu einer Blockierung des betroffenen Flows führen. Paketverluste können im CPU-Cluster jedoch durchaus möglich sein, z.B. wenn ein Paket aufgrund mangelhafter Integrität oder auch wegen gewisser Filtereinstellungen verworfen werden muss (vgl. auch IPsec, Abschnitt 2.3.4).
- Als Summe der zuvor genannten Probleme lässt sich dem Verfahren auch eine **schlechte Skalierbarkeit** attestieren. Insbesondere der Einsatz eines CAMs ist sehr problematisch, da dessen Realisierung insbesondere mit steigender Anzahl der Einträge sehr ressourcenintensiv ist und exponentiell anwächst. Aber auch der Einsatz von linearen Suchalgorithmen wirft bei steigender Paketanzahl Probleme auf.

Die Probleme des Verfahrens von Wu lassen sich folgendermaßen zusammenfassen: Das System verwaltet aktiv alle Flows und alle Pakete, die sich gleichzeitig im System befinden. Allerdings ist, wie die Simulationen im letzten Kapitel nahelegen, nur ein kleiner Anteil der Pakete im maximal einstelligen Prozentbereich wirklich *out-of-order*. Das erhöht sowohl den Speicher- als auch den Verwaltungsaufwand und verschlechtert die Skalierbarkeit deutlich.

Aus dieser Erkenntnis heraus, wurde im Rahmen dieser Arbeit ein Verfahren entwickelt, das die Vorteile der flowbasierten Hardware-Resequenzierung aufgreift und die festgestellten Nachteile auf ein Mindestmaß reduziert (siehe [81]). Anhand von System-Simulationen wird die Wirksamkeit des Verfahrens bestätigt sowie diverse Dimensionie-

rungsaspekte betrachtet. Zudem wird eine Hardware-Implementierung im FPGA vorgestellt. Die Ergebnisse realer Messungen mit dem FlexPath-Demonstrator finden sich in Kapitel 7.

6.2 Konzept der Hardware-Resequenzierung

Ähnlich wie bei dem Verfahren von Wu et al. teilt sich das hier beschriebene Verfahren in einen eingangsseitigen und einen ausgangsseitigen Teil. Wie in Abbildung 6.1 ersichtlich markiert der eingangsseitige *Ingress Tagger* die Pakete mit einer Flow-spezifischen Identifikationsnummer (ID), sowie einer pro ID aufsteigenden Sequenznummer. Ausgangsseitig wird anhand von ID und Sequenznummer erkannt, ob sich ein Paket *in-sequence* befindet. *Out-of-order* Pakete werden bis zur Ankunft der fehlenden Pakete in Puffern festgehalten und resequenziert. Anders als bei Wu wird die originale Paketreihenfolge nicht an einem zentralen Ort gespeichert, sondern implizit dem Paket anhand von Flow-ID und Sequenznummer angehängt. Die Werte können dabei – wie im Falle von FlexPath – innerhalb des Paket-Deskriptors abgespeichert werden. Es kann aber (je nach Architektur) auch ein zusätzlicher Header am Anfang des Pakets benutzt werden.

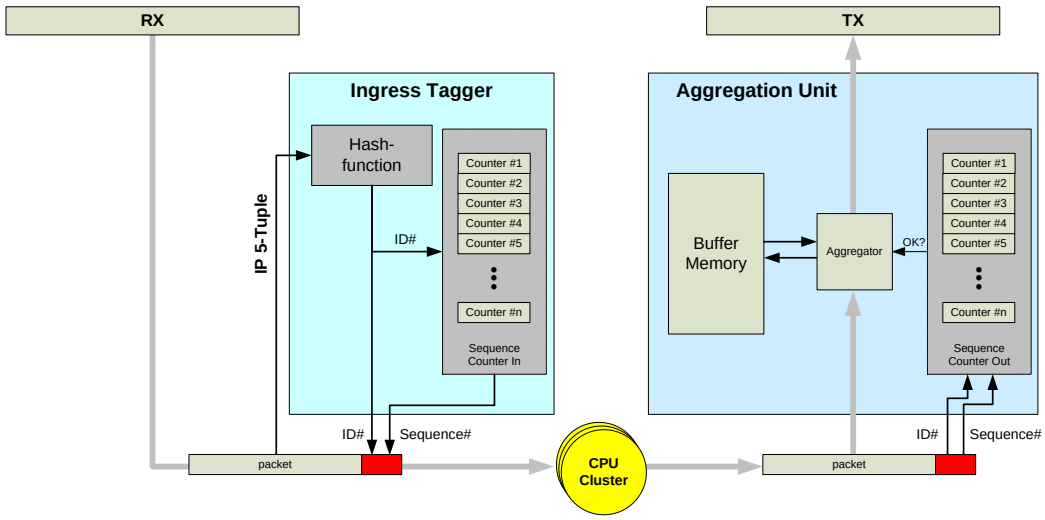

Abbildung 6.1: Paket-Resequenzierung im FlexPath-NP.

Um den Einsatz eines ressourcenintensiven CAMs zu umgehen, wird für jede mögliche Flow-ID ein- und ausgangsseitig ein Zähler angelegt. Der eingangsseitige Zähler im *Ingress Tagger* speichert somit für jeden Flow die nächste zu vergebende Sequenznummer, der ausgangsseitige Zähler in der *Aggregation Unit* die nächste, erwartete Sequenznummer. Die Verwendung des IP 5-Tupels als Flow-ID ist dabei nicht möglich, da sich mit dem 104 Bit breiten Wert mehr als 10^{30} Zählereinträge ergäben. Aus diesem Grund wird ein Hashwert aus dem IP 5-Tupel gebildet. Dieser wird als Flow-ID und damit als Einsprungadresse für den Zählerstand verwendet.

Hashing hat dabei grundsätzlich das Problem, dass viele, eigentlich voneinander unabhängige Flows auf ein- und denselben Hashwert abgebildet werden. Andererseits sind sehr wenige der theoretisch möglichen 2^{104} Flows tatsächlich gleichzeitig im System aktiv. Sofern sich zwei Pakete unterschiedlicher Flows mit gleichem Hashwert (=Flow-ID) gleichzeitig im CPU-Cluster befinden, spricht man von einer Kollision. Um Kollisionen bestmöglich zu vermeiden, muss ein Hashing-Algorithmus verwendet werden, der die Eingangswerte möglichst gleichmäßig auf die möglichen Hashwerte verteilt und Bündelungen vermeidet. Auch wenn prinzipiell eine Vielzahl an Hashing-Algorithmen denkbar sind, wurde in [50] gezeigt, dass eine zyklische Berechnung (CRC) für IP sehr gute Ergebnisse liefert. Es wird daher im Folgenden ein CRC16-CCITT verwendet.

Ausgangsseitig wird die Sequenznummer des an der *Aggregation Unit* eintreffenden Pakets mit dem zugehörigen Zählerstand verglichen. Sofern das Paket *in-sequence* ist, kann es direkt weitergeleitet werden. Der Zählerstand wird anschließend inkrementiert. Gleichzeitig wird geprüft, ob zur zugehörigen Flow-ID weitere Pakete im Pufferspeicher liegen, die nun *in-sequence* mitversendet werden können. Ist die Sequenznummer dagegen größer als der Zählerstand – und damit von der Reihenfolge her zu früh an der *Aggregation Unit* – wird das Paket im Puffer zwischengespeichert. Hier zeigt sich der nächste entscheidende Unterschied zum Verfahren nach Wu. Bei Wu durchsucht ein Suchalgorithmus eine Liste aller Pakete nach versandfertigen Paketen. Bei dem hier vorgestellten Verfahren löst dagegen ein in der *Aggregation Unit* eintreffendes Paket den Paketversand aus. Sofern die Reihenfolge eingehalten wird, kann es unverzüglich weitergesendet werden. Auch die zwischengespeicherten Pakete werden schnellstmöglich ausgelesen und versendet.

6.2.1 Anpassungen am funktionalen Simulationsmodell

Bereits in Abschnitt 5.4.3 wurde eine erste Abschätzung der *Packet Reordering*-Raten bei *Spraying* vorgenommen. Bei *Spraying* ist zunächst von einer uniformen Verarbeitung auszugehen, deren Bearbeitung im Grunde für jedes Paket gleich lang benötigt. Da die Bearbeitung der Pakete aufgrund einer gemeinsamen *Spraying*-Queue sequentiell startet, sind Paketüberholungen nur durch unterschiedliche Verarbeitungslatenzen in der CPU, z.B. wegen dem Zugriff auf eine gemeinsame Ressource wie Speicher, Bus etc. möglich. Diese Effekte wurden bisher in der Simulation dadurch abstrahiert, dass ein gewisser Prozentsatz der Pakete einen Aufschlag um den Faktor 1,5 bzw. 2 auf die Rechenzeit im Simulationsmodell erhält. Hierbei handelte es sich nur um eine erste grobe Schätzung des Prozessierungsjitters. Um fundierte Aussagen bezüglich Resequenzierungsalgorithmus und dessen Umsetzung (siehe Abschnitt 6.2.2) treffen zu können, bedarf es dagegen einer realitätsnäheren Untersuchung. Aus diesem Grund ist es notwendig, das Simulationsmodell zu verfeinern und den Prozessierungsjitter realistisch nachzubilden.

Zu diesem Zweck wurde der FlexPath-Demonstrator (siehe Kapitel 7) genutzt. In einem System mit zwei CPUs und *Spraying* aus einer Forwarding-Queue, wurde bei 1.000 zufällig ausgewählten Paketen die jeweilige Prozessierungsdauer gemessen. Diese Messung erfolgte mit Hilfe des im PowerPC integrierten *Time Base Registers*. Das Re-

gister wurde mit Start der Verarbeitungsroutine auf Null zurückgesetzt und am Ende der Routine wiederum ausgelesen. Das Register wird dabei bei jedem Takt des Prozessors automatisch inkrementiert. Auf diese Weise kann mit geringen Aufwand die genaue Prozessierungszeit der Routine bestimmt werden. Das Ergebnis der Messung ist in Abbildung 6.2 dargestellt. Die Werte sind in einem Raster von je 50 Takten zusammengefasst. Es zeigt sich, dass die ursprünglich gemachte Annahme von 20% des Verkehrs mit einem Faktor 1,5 und 10% mit Faktor 2 sicherlich zu hoch gegriffen war. Andererseits ergibt sich gerade bei der Mehrheit der Pakete (97,7%) eine in etwa gaußförmige Streuung zwischen 1.050 und 1.350 Takten. Lediglich 1% der Pakete erreichen Werte über 1.500 Takte bis maximal 1.750 Takte. Diese Verteilung wird nun als Faktor (1,05-1,75) mit der statischen Prozessierungszeit (IP-Forwarding: 10μs) multipliziert und im Simulationsmodell übernommen.

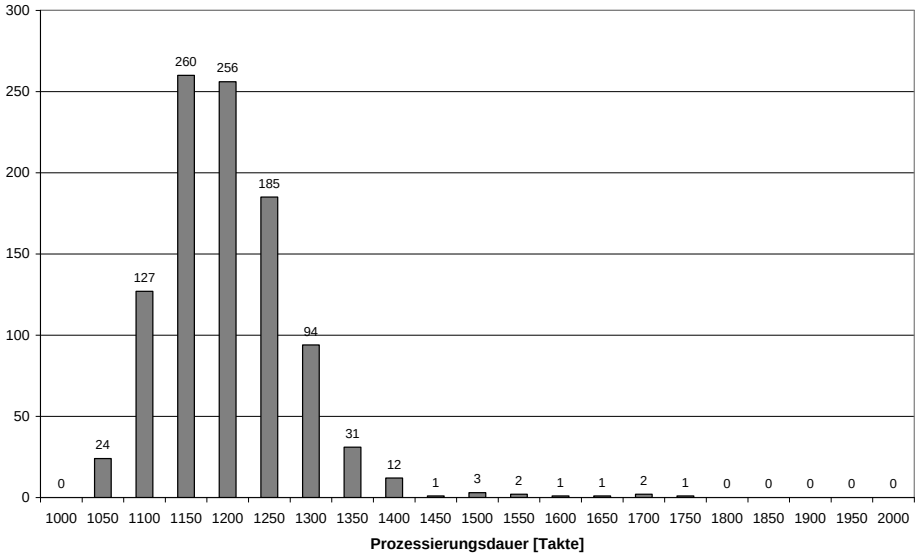

Abbildung 6.2: Jittermessungen für die Verarbeitung eines Pakets (IPv4-Forwarding) im FlexPath-Demonstrator.

6.2.2 Speicherbedarf/-organisation und Sortieralgorithmus

Die Abschätzung des Speicherbedarfs sowie die notwendigen Resequenzierungs-Puffergrößen sind wichtige Informationen, die maßgeblich Einfluss auf die Architektur der *Aggregation Unit* haben. In Abhängigkeit von den zu erwartenden Ergebnissen ergeben sich unterschiedliche Design-Alternativen für die Resequenzierung.

Von allen theoretisch möglichen Flows ist jeweils nur ein kleiner Bruchteil tatsächlich im System aktiv. Zudem wurde aus Abbildung 5.20 bereits ersichtlich, dass nur ein sehr kleiner Anteil der Pakete tatsächlich *out-of-order* ist. Es ist daher nicht sinnvoll, eine Resequenzierungs-Queue für jeden Flow vorzuhalten. Vielmehr wird eine Reihe von

Resequenzierungs-Queues implementiert, die bei Bedarf einem Flow (genauer einer Flow-ID) zugeordnet werden. Die Realisierung dieser Queues kann dabei entweder statisch oder dynamisch erfolgen (siehe hierzu auch [82]). Beide Möglichkeiten werden im Folgenden kurz vorgestellt und bewertet. Mit Hilfe von Simulationen kann im Anschluss eine sinnvolle Festlegung für eine Variante erfolgen.

Alternative 1: Dynamische Queues

Bei der Alternative mit dynamischen Queues teilen sich alle Resequenzierungs-Queues einen gemeinsamen Speicherbereich. Alle freien Speichersegmente, die je ein Paket – bzw. bei FlexPath einen Paket-Deskriptor – aufnehmen können, sind über eine verkettete Liste miteinander verbunden (siehe Abbildung 6.3a). Pro Flow-ID wird neben der nächsten erwarteten Sequenznummer der Zeiger auf den Beginn einer evtl. vorhandenen Resequenzierungs-Queue abgespeichert. Hierzu zeigt ein Zeiger auf das erste verwendete Speichersegment. Die Segmente der einzelnen Queues sind wiederum mit Zeigern auf das jeweilige nächste Segment miteinander verbunden. Ankommende Pakete werden – sofern notwendig – direkt bei der Ankunft in die korrekte Reihenfolge gebracht.

Im dargestellten Beispiel wird davon ausgegangen, dass das Paket mit der Flow-ID 2 und Sequenznummer 19 eintrifft. Der Flow wartet auf die Sequenznummer 16. Eine Überpüfung des Queue-Zeigers ergibt, dass für die Flow-ID bereits eine Queue angelegt wurde. Der Zeiger zeigt auf das Paket mit der Nummer 17. Der Sortieralgorithmus muss nun die komplette verkette Liste abarbeiten, bis die richtige Stelle (zwischen 18 und 20) gefunden wurde. Dann muss noch ein freies Segment angefordert werden und die Zeiger bei Paket 18 und 19, sowie der Zeiger auf das erste freie Segment müssen angepasst werden (siehe Abbildung 6.3b).

Frei werdende Segmente ausgehender Pakete werden hier ans Ende der verketteten Liste angehängt.

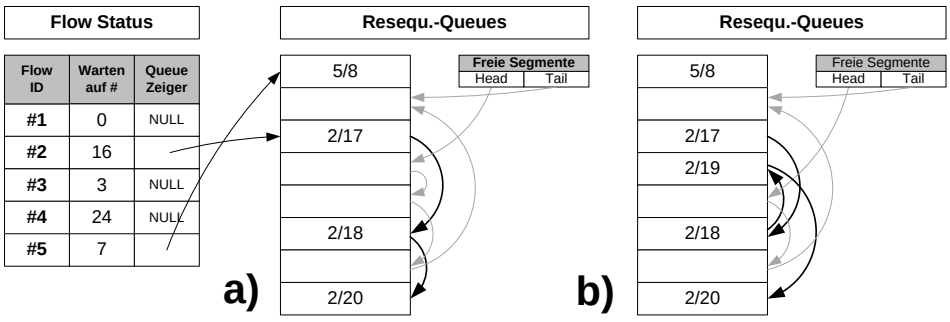

Abbildung 6.3: Resequenzierungs-Beispiel mit dynamischen Queues.

Alternative 2: Statische Queues

Statische Queues sind von vornherein auf Anzahl und Länge festgelegt (siehe Abbildung 6.4). Eine Resequenzierungs-Queue wird bei Bedarf alloziert und einer Flow-ID zugeordnet. Ein Flow-Status hält hierzu neben der Sequenznummer die evtl. vorhandene Zuordnung zu einer Resequenzierungs-Queue fest. Ankommende Pakete können direkt an den korrekten Speicherplatz abgelegt werden. Der Offset innerhalb der Queue ergibt sich aus der Differenz von ankommender und erwarteter Sequenznummer. Ein ankommendes Paket mit Flow-ID 2 und Sequenznummer 19 wird deshalb im Beispiel von Abbildung 6.4 direkt an dritter Stelle der Queue abgelegt – unabhängig von bereits zuvor in der Queue gespeicherten Paketen.

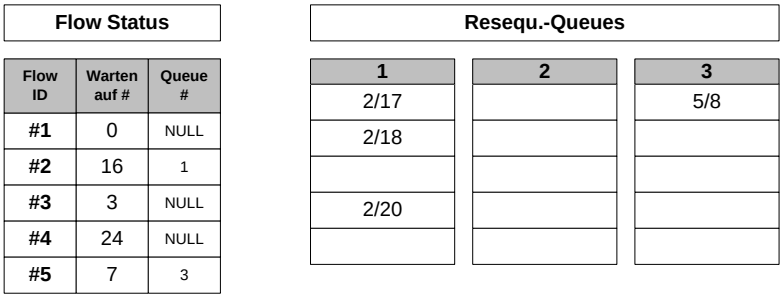

Abbildung 6.4: Resequenzierungs-Beispiel mit statischen Queues.

Bewertung der beiden Alternativen

Beide Verfahren haben je nach Anwendung Vor- und Nachteile. Die Verwendung von dynamischen Queues zeichnet sich durch einen verhältnismäßig niedrigen Speicherverbrauch aus. Da sich alle Queues einen gemeinsamen Pool an Speichersegmenten teilen, kommt es zu Bündelungsgewinnen. Hinzu kommt, dass die Anzahl und Länge der einzelnen Queues nicht von vornherein begrenzt ist. Vielmehr kann einzelnen Queues bei Bedarf mehr Speicherplatz zugeordnet werden. Beschränkend wirkt lediglich der in Summe zur Verfügung stehende Speicherplatz.

Der effizienteren Speicherausnutzung und der erhöhten Flexibilität steht bei dynamischen Queues ein erhöhter Verwaltungsaufwand gegenüber. Mit Hilfe einer verketteten Liste müssen freie Speichersegmente verwaltet werden. Beim Abspeichern eines Pakets muss zunächst ein Segment alloziert und schließlich nach dem Versenden wiederum dealloziert werden. Das Einsortieren von neuen Paketen in eine verkettete Liste erfolgt mit einer linearen Komplexität ($\mathcal{O}$(n)). Damit eignet sich die dynamische Implementierung insbesondere für kurze Queues.

Statische Queues sind dagegen sehr einfach zu verwalten. Eine verkettete Liste und die damit verbundene (De-)Allokation von Speicherplätzen entfällt. Das Einsortieren von ankommenden Paketen in eine bestehende Liste kann direkt mit Hilfe der Differenz von ankommender und erwarteter Sequenznummer erfolgen. Damit ist die Dauer des Sor-

	OC48_mux	**OC192_mux**	**OC192_Qu.**
Max. gespeicherte Pakete	7	7	**8**
Max. gespeicherte Pakete pro Flow	7	7	**8**
Max. Offset	7	8	**9**
Max. aktive Queues	2	**3**	**3**

Tabelle 6.1: Pufferkenndaten bei verschiedenen Verkehrsmitschnitten (12 CPUs, nur BE-Verkehr) bei 16 Bit-Flow-ID.

tiervorgangs in aller Regel zeitlich deterministisch und unabhängig von der Queuegröße.

Der Nachteil von statischen Queues besteht v.a. in der Tatsache der ineffizienten Speicherausnutzung. Die Größe der einzelnen Queues ist von vornherein festgelegt und orientiert sich an den in der Praxis vorkommenden *worst case*-Größen – selbst wenn der Speicherplatz die meiste Zeit nicht verwendet wird. Außerdem muss die Anzahl der zur Verfügung stehenden Queues zuvor festgelegt werden.

Die Entscheidung für eine Variante ist abhängig vom zu erwartenden Resequenzierungs-Aufwand. Um eine fundierte Entscheidung treffen zu können, wird für die bekannten Verkehrsmitschnitte der Aufwand anhand von Simulationen ermittelt. Hierbei erfolgt eine Beschränkung auf Forwarding-Verkehr. Dabei wurden die folgenden Werte bestimmt (siehe Tabelle 6.1):

- **Maximal gespeicherte Pakete**: Dies ist die maximale Anzahl an Paketen, die gleichzeitig in allen Resequenzierungs-Queues gespeichert werden müssen. Dies entspricht somit der Mindestanzahl an Paketspeichern bei einer dynamischen Implementierung.
- **Maximal gespeicherte Pakete pro Flow**: Dies ist die maximale Anzahl an Paketen, die gleichzeitig in **einer** Resequenzierungs-Queue gespeichert werden müssen.
- **Maximaler Offset**: Dies ist die maximale Differenz aus ankommender und erwarteter Sequenznummer. Im Gegensatz zum zuvor genannten Wert schließt dies noch etwaige Lücken in der Resequenzierungs-Queue mit ein. Der Wert kann somit als Mindestgröße der Queues bei einer statischen Lösung gesehen werden, um vollständige Resequenzierung zu erreichen.
- **Maximal aktive Queues**: Maximale Anzahl der zur gleichen Zeit aktiven Resequenzierungs-Queues.

Man erkennt, dass in Summe nie mehr als acht Pakete gleichzeitig zur Resequenzierung abgespeichert wurden. Gleichzeitig waren maximal drei Resequenzierungs-Queues aktiv. Der maximale Offset betrug neun. Für eine dynamische Queue-Implementierung ergibt sich damit ein minimaler Speicherverbrauch von acht Paketen. Für die statische Lösung müssen dagegen $3 \cdot 9 = 27$ Paketspeicherplätze vorgehalten werden. Allerdings ist der benötigte Speicherplatz pro Paket beim Abspeichern des Paket-Deskriptors mit 16 Byte (128 Bit) nicht besonders hoch. Damit ergibt sich ein Speicherverbrauch von mindestens 432 Byte bei der statischen Lösung im Gegensatz zu 128 Byte bei der dynamischen

Lösung. Der Speicherbedarf der statischen Implementierung ist absolut betrachtet sehr moderat, so dass das Einsparpotential des dynamischen Verfahrens als klein gelten muss. Da das dynamische Verfahren zudem einen ungleich höheren Aufwand an Logik in der Implementierung und eine nicht-deterministische, höhere Sortierzeit mit sich bringt, wird die statische Implementierung der Resequenzierungs-Queues gewählt.

6.2.3 Ingress Tagger

Der *Ingress Tagger* berechnet unter Verwendung des CRC16 eine Flow-ID. Er benötigt als Eingangsdaten lediglich den Paketheader. Dabei wird eine Reihe von unterschiedlichen Pakettypen unterschieden:

- **TCP / UDP:** Sowohl für TCP als auch UDP ist die Paketreihenfolge von entscheidender Bedeutung für die Netzwerkleistung bzw. -qualität. Der *Ingress Tagger* kann anhand des IP 5-Tupels den Hashwert und eine Flow-ID berechnen.
- **IPsec:** Aufgrund der Verschlüsselung sind bei IPsec die Layer 4 Ports nicht lesbar. Daher beschränkt sich bei IPsec die Bildung des Hashwertes auf das IP 3-Tupel (Protokoll, Adressen).
- **Kontrollpakete:** Bei Kontrollpaketen wie ICMP oder ARP ist eine Resequenzierung nicht sinnvoll. Zum einen enden viele Kontrollpakete im NP. Zum anderen sind auch Kontrollpakete, die den NP passieren, nicht auf *Packet Reordering* sensitiv. Sämtliche Kontrollpakete liegen im *Slow Path*, sind also weder zeitkritisch noch zahlenmäßig dominant ($< 1\%$). Kontrollpakete werden durch die Flow-ID 0 gekennzeichnet. Dies impliziert, dass die Berechnung des Hashwertes für die restlichen Pakete die 0 ausschließt.

Für jede Flow-ID wird ein Zählerstand im *Ingress Tagger* gespeichert. Bei Ankunft eines Pakets wird der zugehörige Zählerstand dem Paket als Sequenznummer übergeben und der Zähler anschließend inkrementiert. Flow-ID und Sequenznummer werden dem Paket in Form eines Headers angehängt bzw. im Paket-Deskriptor ergänzt.

6.2.4 Aggregation Unit

Ausgangsseitig extrahiert die *Aggregation Unit* zunächst Flow-ID und Sequenznummer aus dem Paket bzw. Paket-Deskriptor. Die Sequenznummer wird mit dem aktuellen Zählerstand verglichen. Außerdem wird überprüft, ob für die Flow-ID bereits eine Resequenzierungs-Queue alloziert wurde. Folgende Fälle können dabei auftreten:

- **Flow-ID = 0:** Diese Pakete werden umgehend ohne eine Überprüfung der Sequenznummer weitergeleitet. Dies ermöglicht das Umgehen der Resequenzierung (z.B. für Kontrollpakete oder vom NP erzeugte Pakete).
- **Sequenznummer = Zählerstand, keine Queue aktiv:** In diesem Fall befindet sich das Paket *in-order*. Es wird direkt zum Versenden an die nachfolgenden Module weitergeleitet. Der der Flow-ID zugeordnete Zählerstand wird um eins inkrementiert.

- **Sequenznummer = Zählerstand, Queue aktiv:** Das ankommende Paket wird weitergeleitet und der zugehörige Zählerstand wird inkrementiert. Zusätzlich wird überprüft, ob weitere, konsekutive Pakete in der zugeordneten Resequenzierungs-Queue vorhanden sind. Zusammenhängende Pakete werden ebenfalls versendet und der Zählerstand um die Anzahl der Pakete inkrementiert. Das Auslesen von Paketen endet an der ersten Lücke in der Queue. Sofern alle Pakete versendet werden konnten, wird die Resequenzierungs-Queue wieder freigegeben.
- **Sequenznummer > Zählerstand, keine Queue aktiv:** Es wird eine freie Resequenzierungs-Queue angefordert und mit der Flow-ID verknüpft. Das Paket wird am entsprechenden Offset der Queue abgespeichert.
- **Sequenznummer > Zählerstand, Queue aktiv:** Die Allokation der Queue entfällt. Das Paket wird direkt am entsprechenden Offset der Queue abgespeichert.

Die Resequenzierungs-Queues sind als Ringpuffer ausgeführt (siehe Abbildung 6.5). Ein Zeiger markiert den aktuellen Beginn der Queue. In der Abbildung ist zudem nochmals ein einfaches Beispiel zur Veranschaulichung der Funktionsweise dargestellt. Es wird davon ausgegangen, dass Flow-ID 3 der Resequenzierungs-Queue 1 (Q#1) zugeordnet ist. In der Queue befinden sich bereits die Pakete mit Sequenznummer 5 und 7. Aktuell wird auf Paket 4 gewartet (a). Sobald Paket 4 eintrifft, wird versucht die Queue zu leeren. Konsekutiv befindet sich allerdings nur Paket 5 in der Queue, welches mit versendet wird. Die Übertragung stoppt beim fehlenden Paket 6. Der Zählerstand wird auf 6 erhöht und Paket 7 wird zum neuen Startpunkt der Queue (b).

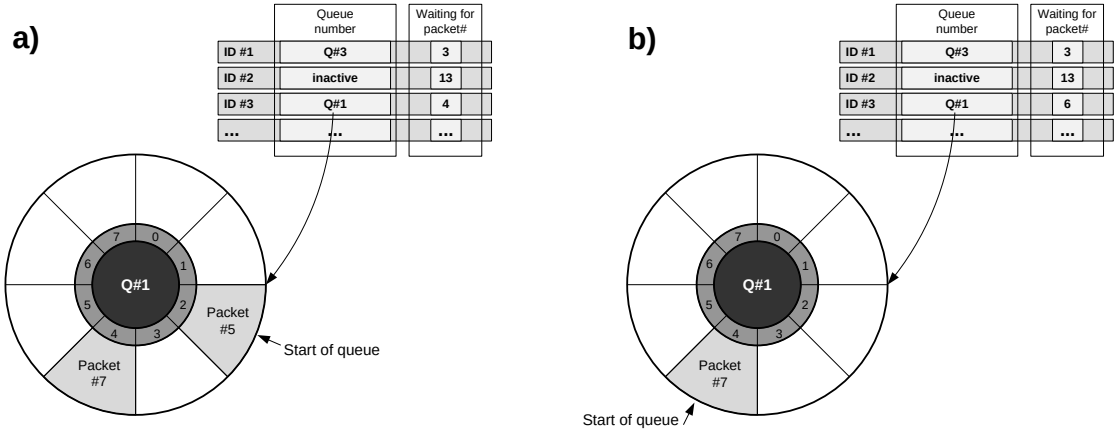

Abbildung 6.5: Ringpuffer-Struktur und Resequenzierungs-Beispiel.

Die *Aggregation Unit* muss ferner eine Reihe von Sonderfällen abdecken. Dazu zählen zum einen Paketverluste, es bedarf aber auch Mechanismen, die bei Überschreiten der vorhandenen Resequenzierungs-Queue-Länge oder -Anzahl greifen. Die Fälle werden im Folgenden einzeln beschrieben.

Paketverluste

Bei der Hardware-Resequenzierung von Wu [61] werden keinerlei Paketverluste erlaubt. Innerhalb eines NP-Systems kann es jedoch an mehreren Stellen zu Paketverlusten kommen. Im FlexPath-NP schließt das zum einen das Verwerfen eines Pakets durch eine CPU (z.B. aufgrund mangelhafter Paketintegrität oder nicht unterstützter Protokolle) und zum anderen die *Auto-Discard*-Funktion des *Packet Distributors* (siehe Abschnitt 5.3.4) mit ein.

Paketverluste führen ohne einen geeigneten Mechanismus zur Blockade des Systems. Nachfolgende Pakete würden in eine Resequenzierungs-Queue abgespeichert werden. Die *Aggregation Unit* würde endlos auf das verloren gegangene Paket warten und die zugehörige Flow-ID blockieren.

Mit Paketverlusten kann prinzipiell auf zwei Arten umgegangen werden, um eine Systemblockade zu vermeiden:

- **Benachrichtigung:** Eine paketverwerfende Einheit informiert die *Aggregation Unit* über den Paketverlust. Die *Aggregation Unit* kann ihren Zählerstand daraufhin anpassen und den Paketverlust berücksichtigen.
- **Timeout:** Für jede Resequenzierungs-Queue wird nach einer zuvor festgelegten Zeit ein Timeout ausgelöst und das noch fehlende, erste Paket als verloren eingestuft. Somit wird eine endlose Blockade des Systems verhindert.

Auch hier besitzen beide Alternativen Vor- und Nachteile. Eine aktive Benachrichtigung verhindert, dass nachfolgende Pakete unnötigerweise in eine Resequenzierungs-Queue gespeichert werden. Durch die Vorab-Information können die folgenden Pakete die *Aggregation Unit*, sofern sie selbst *in-sequence* sind, direkt passieren und erfahren dadurch keinen Aufschlag auf die Latenz. Allerdings erfordert dieser Ansatz, dass sämtliche Einheiten, die sich vom Paketverlauf her nach dem *Ingress Tagger* befinden, die *Aggregation Unit* zuverlässig über Paketverluste informieren. Zudem muss die *Aggregation Unit* von all diesen Einheiten erreichbar sein, sich also z.B. an einem gemeinsamen Bus befinden. Die korrekte Funktionsweise kann also nicht durch die Resequenzierungs-Einheiten alleine garantiert werden, sondern bedarf der Mitwirkung und Anpassung des ganzen Systems.

Beim Timeout stuft die *Aggregation Unit* Pakete dagegen selbstständig als verloren ein und ist somit unabhängig von anderen Modulen. Die Zeitspanne bis zum Timeout ist dabei mit Bedacht zu wählen (siehe Abschnitt 6.3.4). Nachteilig wirkt sich hier aus, dass nachfolgende Pakete bis zum Timeout unnötig in einer Resequenzierungs-Queue abgespeichert werden. Da in einem sinnvoll dimensionierten System ein Paketverlust sehr selten auftritt, hat dieser Nachteil auf das Gesamtsystem kaum messbare Auswirkungen.

Ein fehlerfreies Arbeiten und eine leichtere Testbarkeit lässt sich damit mit dem Timeout erreichen, da hierzu nicht das restliche System zur Überprüfung miteinbezogen werden muss. Dies ist insbesondere wichtig, da die Hardware-Resequenzierung möglichst einfach auf andere, Nicht-FlexPath-Systeme übertragbar sein soll, bei denen eine Kommunikation zwischen der *Aggregation Unit* und den paketverwerfenden Modulen unter

Umständen nicht oder zumindest nicht ohne weiteres möglich ist. Es wird daher der Timeout bevorzugt und umgesetzt.

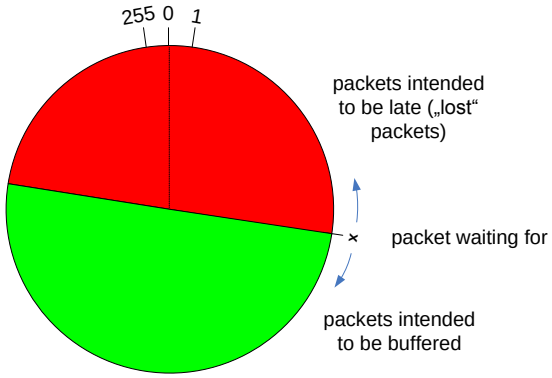

Abbildung 6.6: Überlauf der Sequenznummer und Definition von verspäteten und zukünftigen Paketen (Beispiel mit 8 Bit Sequenznummer).

Durch einen zu frühen Timeout können in Ausnahmesituationen einzelne Pakete mit ungewöhnlich hoher Prozessierungszeit fälschlicherweise als verloren eingestuft werden. Diese Pakete passieren die *Aggregation Unit* nach dem Timeout und somit zu einer Zeit, zu der bereits auf eine höhere Sequenznummer gewartet wird. Die Sequenznummer selbst ist dabei ein in der Bitbreite eingeschränkter Wert, der sich durch einen Überlauf in gewissen Zeitabständen wiederholt. Die *Aggregation Unit* muss somit entscheiden, ob es sich um ein verspätetes Paket handelt oder bereits um ein zukünftiges, welches folglich zur Resequenzierung abgespeichert werden muss. Zu diesem Zweck wird der Zahlenkreis der Sequenznummern in zwei Hälften geteilt (siehe Abbildung 6.6). `X` markiert dabei die aktuelle (= als nächstes erwartete) Sequenznummer. Die Sequenznummern, die sich in dem im Uhrzeigersinn folgenden Halbkreis befinden, werden als zukünftige (= zu resequenzierende) Pakete gewertet, die Sequenznummern im anderen Halbkreis als verspätete Pakete. Der Wertebereich der Sequenznummern und damit die Zahl der Bits zur Repräsentation muss groß genug gewählt werden, dass Fehleinstufungen möglichst ausgeschlossen werden.

Verspätete Pakete werden von der *Aggregation Unit* unmittelbar an den Ausgang weitergeleitet. Nachdem die dieser Sequenznummer folgenden Pakete bereits versendet wurden, ist eine nachträgliche Resequenzierung in diesem Fall nicht mehr möglich.

Überschreiten der Queue-Länge

Bei einer sinnvollen Dimensionierung sollte im Regelfall die implementierte Länge der Resequenzierungs-Queues ausreichen, um eine vollständige Resequenzierung der Pakete zu garantieren. Dennoch ist es schwierig, in einem komplexen, hoch-parallelen System eines NPs bei zudem sehr variablem Verkehr, dies für alle Fälle zu garantieren. Daher ist es sinnvoll, für unterschiedliche Ausnahmefälle Schutzmechanismen einzubauen, die

ein Weiterarbeiten der *Aggregation Unit* garantieren. Ein möglicher Sonderfall ist dabei das Überschreiten der Queue-Länge zur Resequenzierung.

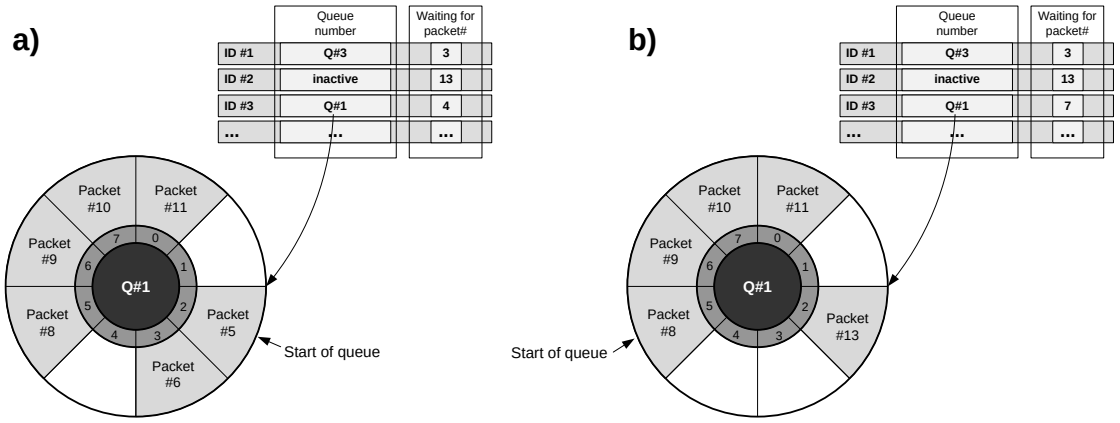

Abbildung 6.7: Resequenzierungs-Beispiel bei Überschreiten der Queue-Länge.

Ein Beispiel in Abbildung 6.7 veranschaulicht die Problematik . Es ist eine Resequenzierungs-Queue mit Länge acht gegeben, in der bereits die Pakete 5 und 6, sowie 8-11 abgespeichert wurden. Aktuell wird auf Paket 4 gewartet. Als nächstes trifft nun jedoch Paket 13 ein. Der Offset von neun überschreitet die Queue-Länge, das Paket kann somit nicht abgespeichert werden (a). Zur Lösung des Problems werden hier so viele Pakete versendet, bis der Offset auf acht sinkt und somit wieder Platz für das aktuelle Paket vorhanden ist. Im konkreten Beispiel beträfe dies lediglich Paket 5. Allerdings können dabei alle konsekutiv folgenden Pakete mitversendet werden, da dies keinerlei zusätzliche negative Auswirkungen auf das *Packet Reordering* erwarten lässt. Im Beispiel betrifft dies Paket 6. Nach dem Versenden der Pakete kann Paket 13 in die Queue geschrieben werden. Der Zählerstand wird auf 7 aktualisiert (b).

Hierbei wird toleriert, dass das möglicherweise noch ankommende Paket 4 als dann verspätetes Paket *out-of-order* ausgesendet werden muss. Das Verfahren greift aber auch bei Paketverlusten, nämlich dann, wenn bis zum Timeout mehr Pakete einer Flow-ID ankommen, als die Queue aufnehmen kann. In diesem Fall wirkt sich der Mechanismus sogar positiv auf die Verweildauer der wartenden Pakete und damit die Latenz aus.

Überschreiten der Queue-Anzahl

Aus den selben wie im vorigen Abschnitt beschriebenen Gründen ist es möglich, dass mehr Resequenzierungs-Queues benötigt werden, als tatsächlich zur Verfügung stehen. In diesem Fall kann somit keine neue Queue mehr alloziert werden. Es erscheint wenig sinnvoll, eine sich bereits in Benutzung befindliche Queue zu leeren und einer anderen Flow-ID zuzuweisen. Ein *out-of-order* Paket, für das keine Resequenzierungs-Queue mehr alloziert werden kann, wird deshalb ohne Resequenzierung versendet. Der Zählerstand des Flows wird entsprechend erhöht. Damit werden nachfolgende Pakete mit niedriger

Sequenznummer als verspätet *out-of-order* versendet. *Packet Reordering* wird in dieser Ausnahmesituation also wiederum toleriert.

6.3 Dimensionierungsaspekte

Nachdem in den vorgehenden Abschnitten bereits die grundsätzliche Funktionsweise der Resequenzierung vorgestellt wurde, werden in diesem Abschnitt weitergehende Untersuchungen zur Auslegung der Resequenzierungs-Einheit am bekannten Simulationsmodell vorgenommen. Soweit nicht anders angegeben, wurde dabei der *OC48_mux*-Mitschnitt (siehe Abschnitt 5.4.2) verwendet. Im Mittelpunkt der Untersuchungen stehen neben grundsätzlichen Dimensionierungsfragen (Bitbreite des Hashwerts, Queue-Länge, Anzahl) auch Auswirkungen von heterogenem Verkehr (QoS, IPsec) auf die Resequenzierung. Außerdem wird eine sinnvolle Wahl der Zeitspanne für den Timeout untersucht. Die Timeout-Zeit wird im Simulator zu Beginn der Untersuchungen initial auf 50 μs gesetzt.

Es sei hier angemerkt, dass sich die hier durchgeführten Untersuchungen auf das vorgestellte FlexPath-System beziehen. Zwar lassen sich hieraus auch allgemeine Aussagen und Bewertungen ziehen, eine sinnvolle Dimensionierung muss jedoch für jedes System unter Berücksichtigung des konkreten Verkehrs und der ausgeführten Applikationen ggf. neu durchgeführt werden.

6.3.1 Hashwert-Kollisionen und Flow-ID-Bitbreite

Wie im letzten Abschnitt erläutert, wird für die Berechnung des Hashwerts ein CRC mit 16 Bit verwendet. Dabei muss für jeden einzelnen Hashwert ein- und ausgangsseitig ein Zähler für die Sequenznummer vorgehalten werden. Um Ressourcen einzusparen, kann es deshalb wünschenswert sein, die Bitbreite des Hashwerts zu reduzieren. Dabei kann der Hash durch Abschneiden der vorderen Bits auf eine geeignete Länge gekürzt werden. Allerdings gilt dabei, dass die Wahrscheinlichkeit von Kollisionen mit kleinerer Bitbreite steigt. Aus diesem Grund wird im Folgenden das Ausmaß der Kollisionen in Abhängigkeit von der Bitbreite des Hashwerts untersucht. Ziel ist die Minimierung der Bitbreite, ohne dabei nennenswerte negative Effekte auf die Resequenzierung selbst zu erhalten.

Grundsätzlich führt eine Kollision nicht zu einer fehlerhaften Prozessierung. Sie bewirkt aber, dass zwei eigentlich unabhängige Flows von der Resequenzierungseinheit als ein einzelner Flow betrachtet werden, der dann als Ganzes resequenziert wird. Inwiefern hierbei negative Konsequenzen zu erwarten sind, hängt maßgeblich von der Ähnlichkeit der Flows ab. Bei Kollisionen zwischen Flows mit gleichen Anforderungen und nahezu konstanter Prozessierungszeit halten sich die negativen Auswirkungen in Grenzen. Kritischer sind dagegen Kollisionen zwischen Flows mit unterschiedlichen Anforderungen (z.B. Priorität) oder auch unterschiedlichen Prozessierungslatenzen. Überquert beispielsweise ein Paket das Cluster (z.B. aufgrund der Priorität) schneller als Pakete des

kollidierenden Flows, kommt es mitunter zu (vermeintlichen) Paketüberholungen. Die Resequenzierungseinheit versucht, die originale Paketreihenfolge des ganzen Flowbündels wiederherzustellen. Das schnelle Paket wird bis zum Eintreffen der überholten Pakete gestoppt. Dies hat negative Konsequenzen für die Prozessierungslatenz und damit für den Flow an sich.

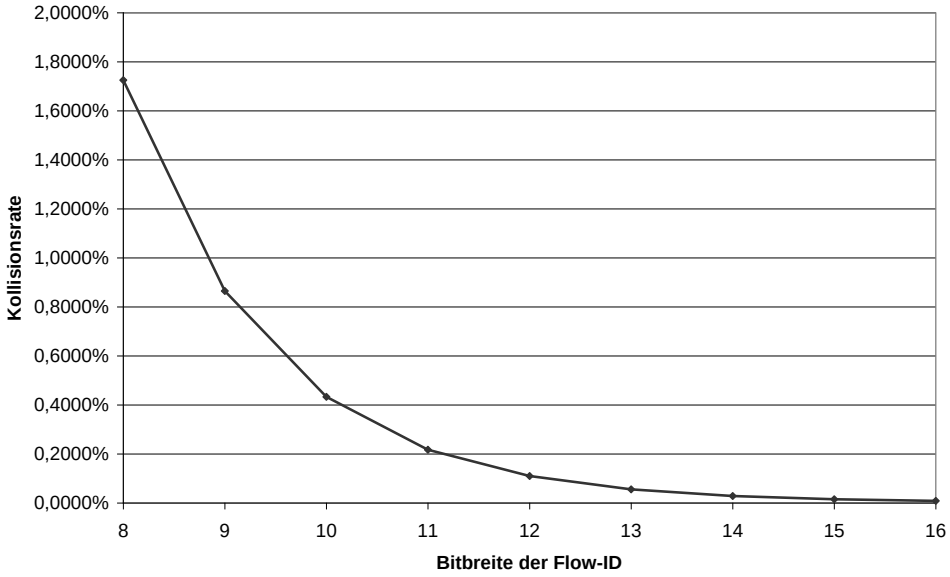

Abbildung 6.8: Hashwert-Kollisionsrate in Abhängigkeit von der gewählten Flow-ID Bitbreite.

Es muss die richtige Balance zwischen niedrigem Ressourcenaufwand (= geringe Bitbreite des Hashwerts) und niedriger Kollisionsrate (hohe Bitbreite) gefunden werden. Zu diesem Zweck wurde die Kollisionsrate am Simulationsmodell für unterschiedliche Flow-ID-Bitbreiten ausgewertet. Hierbei wird zunächst nur homogener Verkehr (IP-Forwarding) betrachtet. Zur Bestimmung der Kollisionen wird am Eingangspfad des Simulators für jede Flow-ID das IP 5-Tupel der eingehenden Pakete gespeichert. Der Ausgangsdatenpfad informiert den Eingangsdatenpfad über ausgehende Pakete, die anschließend wieder aus der Liste gestrichen werden können. Die Simulationen müssen zu diesem Zweck verlustfrei durchgeführt werden. Aus diesem Grunde wurden zwölf CPUs verwendet. Beim Eintritt eines Pakets wird überprüft, ob sich Pakete mit identischer Flow-ID aber unterschiedlichem IP 5-Tupel im System befinden. Falls dies der Fall ist, wird dies als Kollision gewertet. In Abbildung 6.8 erkennt man, dass die Kollisionsrate bei 8 Bit bei ca. 1,7% der Pakete liegt. Sie fällt mit jedem zusätzlichen Bit um näherungsweise den Faktor zwei und liegt bei 12 Bit nur mehr bei knapp über 0,1%. Bei Ausnützen der maximal möglichen 16 Bit kommt es nur noch zu einer Kollisionsrate von 0,009%.

Um den Einfluss der Kollisionsrate auf die Resequenzierung zu ermitteln, wurde im zweiten Schritt die Anzahl der in der Resequenzierungseinheit zwischengespeicherten Pa-

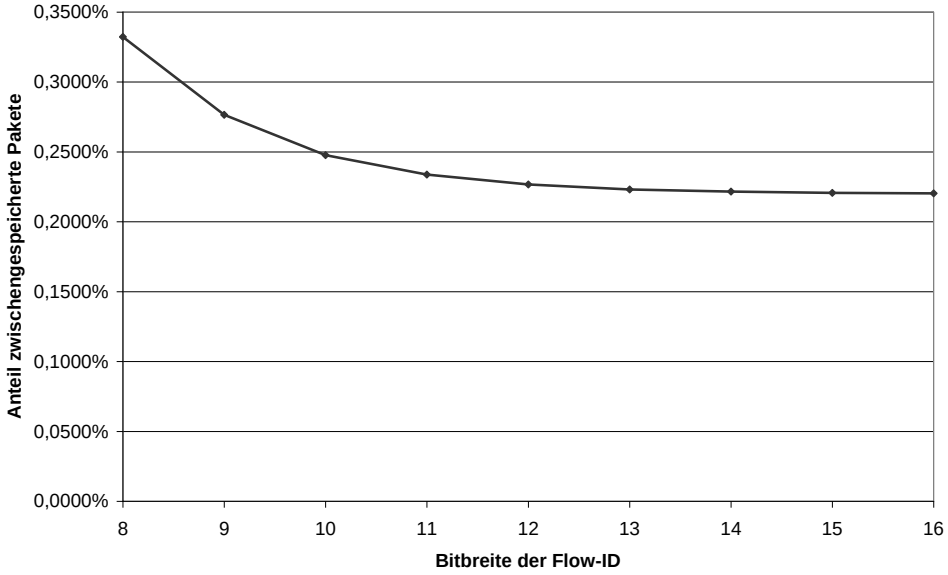

Abbildung 6.9: Anteil der zur Resequenzierung zwischengespeicherten Pakete in Abhängigkeit von der gewählten Flow-ID Bitbreite.

kete gemessen (siehe Abbildung 6.9). Hierbei ist der Anteil der Pakete in Abhängigkeit von der verwendeten Flow-ID Bitbreite dargestellt. Die Kurve nähert sich bei 16 Bit in etwa 0,22% der Pakete. Da hierbei fast keine Kollisionen stattfinden (0,009%, siehe Abbildung 6.8), kommt dieser Wert im Wesentlichen durch tatsächliche Paketüberholungen aufgrund des Jitters im CPU-Cluster zustande. Bei geringeren Bitbreiten steigt der Wert dagegen bis auf 0,33% bei 8 Bit. Diese Differenz lässt sich nun tatsächlich auf Hashwert-Kollisionen zurückführen. Pakete, die eigentlich *in-sequence* sind, werden aufgrund der Kollision als *out-of-order* eingestuft und zur Resequenzierung zwischengspeichert. Die Auswirkungen auf die gemessene durchschnittliche Gesamtsystemlatenz ist mit 15,241μs (8 Bit) zu 15,240μs (16 Bit) jedoch als vernachlässigbar einzustufen.

Aufgrund dieser Ergebnisse wird die Flow-ID Bitbreite im Folgenden zu 12 Bit gewählt. Zwar wird hierbei noch eine Kollisionsrate von ca. 0,1% erreicht, allerdings ist der Einfluss auf die resequenzierten Pakete gering. Jedes zusätzliche Bit würde die Kollisionsrate zwar weiter senken, aber zusätzlich zur Verdoppelung der benötigten Zähler führen.

6.3.2 Dimensionierung bei homogenem Verkehr

In diesem Abschnitt soll eine sinnvolle Queue-Tiefe gefunden werden. Zudem muss die Bitbreite der Sequenznummer festgelegt werden. Hierzu wurden wiederum Simulationen mit zwölf CPUs für die bekannten Verkehrs-Mitschnitte ausgewertet.Die Bitbreite der Flow-ID wurde konform zu Abschnitt 6.3.1 zu 12 Bit gewählt. In Tabelle 6.1 wurden bereits die maximalen Werte der Queuegrößen und -anzahl für *Best Effort*-Verkehr bei 12 CPUs betrachtet.

Es hat sich gezeigt, dass für die gewählten Verkehrsmitschnitte maximal drei Resequenzierungs-Queues bei einer Tiefe von neun ausreichend wären, um eine vollständige Resequenzierung zur erreichen. Die Betrachtung der Maximalwerte ist dabei für eine erste Analyse sicherlich hilfreich. Auf der anderen Seite ist es auch sehr aufschlussreich, wie sehr die einzelnen Queues im Schnitt ausgelastet sind. In Abbildung 6.10 ist die Häufigkeit der einzelnen Offsets für alle drei Verkehrsmitschnitte gezeigt. Man erkennt, dass der Großteil der Pakete ($> 84{,}8\%$) lediglich um ein einziges Paket vertauscht ist (Offset=1). Bei einer Queue-Größe von vier können bereits mindestens 99,72% der Pakete erfolgreich resequenziert werden, bei einer Größe von sechs gar mindestens 99,99%. Ein Offset von mehr als acht konnte in den Simulationen nur bei einem Trace (*OC192_mux*) bei insgesamt zwei Paketen festgestellt werden. Eine Queue-Tiefe von acht Paketen kann damit zwar nicht in allen Fällen eine vollständige Resequenzierung garantieren. In Anbetracht des prozentualen Anteils, der diese Queue-Länge übersteigt, kann dies jedoch zugunsten der geringeren Ressourcen toleriert werden. Im Folgenden wird deshalb die Queue-Länge zu acht gewählt.

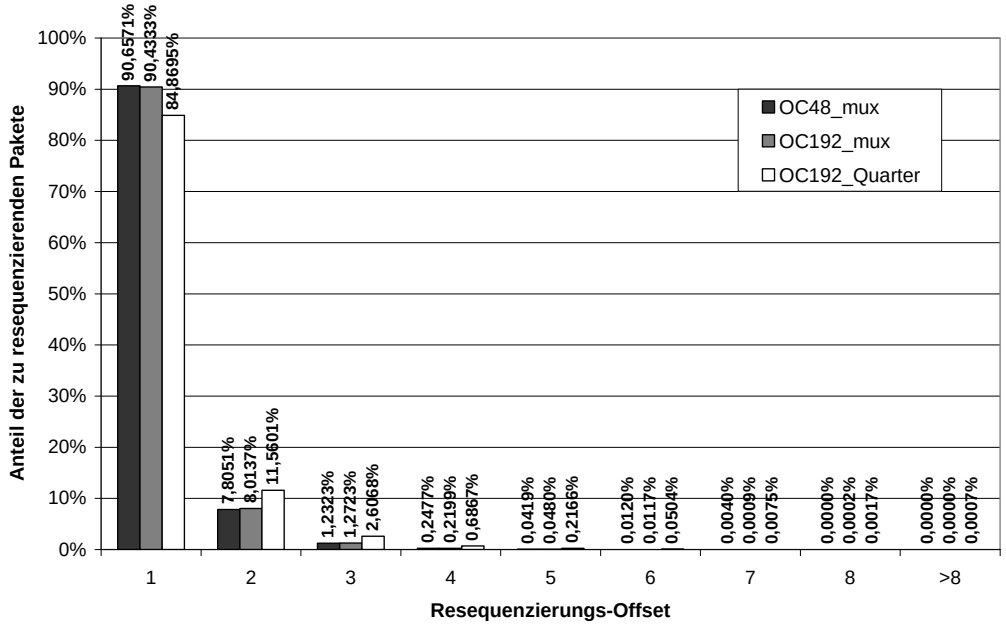

Abbildung 6.10: Offset-Verteilung bei zu resequenzierenden Paketen (12 CPUs, 12 Bit Flow-ID Bitbreite, *Best Effort*-Verkehr)

Tabelle 6.2 gibt einen Überblick über die maximale Anzahl an Paketen, die sich gleichzeitig im NP-System – also zwischen *Ingress Tagger* und *Aggregation Unit* – befinden. Es zeigt sich, dass maximal 36 Pakete parallel verarbeitet werden bzw. sich in den Eingangs-Queues befinden. Dabei sind nie mehr als 26 Pakete eines Flows gleichzeitig im System. In Anbetracht des maximalen Offsets von neun, erscheint ein Wertebereich der Sequenznummern von 256 und damit eine Bitbreite von acht als ausreichend.

	OC48_mux	**OC192_mux**	**OC192_Qu.**
Max. Pakete im System	28	**36**	31
Max. Pakete im System pro Flow	14	23	**26**
Max. Anzahl von Flows im System	28	**34**	27

Tabelle 6.2: Maximale Anzahl an Paketen im System (zwischen *Ingress Tagger* und *Aggregation Unit*, 12 CPUs, 12 Bit Flow-ID Bitbreite).

	a		**b**		**c**	
	12 H., 0 QoS, 0 TC		11 H., 1 QoS, 0 TC		10 H., 1 QoS, 1 TC	
	Max.	**Ø**	**Max.**	**Ø**	**Max.**	**Ø**
BE	102,40 μs	15,27 μs	108,38 μs	15,28 μs	69,87 μs	15,27 μs
QoS	71,26 μs	15,16 μs	26,25 μs	15,15 μs	26,25 μs	15,15 μs
IPsec	32,428 ms	2,034 ms	32,428 ms	2,034 ms	32,428 ms	2,034 ms

Tabelle 6.3: Maximale und durchschnittliche Systemlatenzen aufgeschlüsselt nach den unterschiedlichen Verkehrsklassen für *Best Effort* Verkehr (BE), hochprioren Verkehr (QoS) und zu verschlüsselnden Verkehr (IPsec) bei unterschiedlicher Zusammensetzung der Flow-ID (a-c) (*OC48_mux*, 12 CPUs).

6.3.3 Anpassungen für heterogenen Verkehr

In den vorangegangenen Abschnitten konnte gezeigt werden, dass Resequenzierung bei homogenem Verkehr, also Verkehr, der in der Art der Prozessierung ähnlich ist, sehr gut beherrschbar ist. Für die Dimensionierung ergibt sich eine begrenzte Anzahl an Resequenzierungs-Queues. Die notwendige Tiefe der Queues bleibt mit acht oder auch weniger vergleichsweise überschaubar. Die Anwendung eines Hashing-Verfahrens zur Berechnung einer Flow-ID führt zwar zu Kollisionen unter den Flows, die Anzahl hält sich allerdings in Grenzen. Hinzu kommt, dass Kollisionen bei homogenem Verkehr keine nennenswerten negativen Auswirkungen haben (vgl. Abschnitt 6.3.1).

Im nächsten Schritt wird nun untersucht, welche Auswirkungen heterogener Verkehr auf die Resequenzierung hat. Hierzu wird wiederum das bereits aus Abschnitt 5.4.3 bekannte Beispiel verwendet. Neben dem *Best Effort*-Forwarding-Verkehr wird nun also auch hochpriorer Forwarding-Verkehr und IPsec-Verkehr verwendet. Die Verkehrsarten werden eingangsseitig separiert. Der Forwarding-Verkehr wird mittels *Spraying* auf die CPUs verteilt, während der IPsec-Verkehr dediziert zugewiesen wird. IPsec-Pakete besitzen dabei eine ungleich höhere Prozessierungszeit (siehe Abschnitt 5.4.1).

In Tabelle 6.3a sind die Systemlatenzen (= Latenz zum Durchlaufen des Systems) aufgeschlüsselt nach den drei unterschiedlichen Verkehrsklassen dargestellt. Die Simulation wurde zunächst mit dem bekannten 12 Bit Hashwert als Flow-ID durchgeführt (Spalte a). Man erkennt, dass *Best Effort*-Verkehr und hochpriorer Forwarding-Verkehr wie erwartet mit 15,15 μs bzw. 15,27 μs eine deutlich niedrigere durchschnittliche Latenz besitzen als IPsec-Verkehr mit mehr als 2 ms. Erstaunlich ist jedoch die recht hohe

maximale Latenz beim hochprioren Verkehr von 71,26 μs, was beinahe einen Faktor fünf zur durchschnittlichen Latenz darstellt. Durch die Prozessierung allein lässt sich dieser Faktor kaum erklären, schließlich wird der hochpriore Verkehr bevorzugt von den zwölf CPUs bearbeitet und stellt mit ca. 4% des Verkehrs selbst keine ausreichende Last dar, um nennenswerte Wartezeiten in den Eingangs-Queues zu erzeugen. Es kann davon ausgegangen werden, dass Kollisionen zwischen den Flows eine Rolle spielen.

Die erhöhte Latenz kann folgendermaßen erklärt werden: Ein IPsec-Flow und ein hochpriorer Flow werden durch eine Kollision auf eine Flow-ID abgebildet. Während das IPsec-Paket eine sehr lange Prozessierungszeit besitzt, überholt das hochpriore Paket das langsame IPsec-Paket, wird jedoch anschließend in der *Aggregation Unit* zur vermeintlichen Resequenzierung zwischengespeichert. Dort verbleibt es bis zur Ankunft des IPsec-Paketes, höchstens jedoch bis zum Auslösen des Timeouts, der nach 50 μs ausgelöst wird. Damit lässt sich auch die Differenz von ca. 56 μs zum Durchschnitt erklären. Tatsächlich zeigt sich, dass bei Erhöhen des Timeouts auf 500 μs durchaus maximale Latenzen der hochprioren Pakete von mehr als 500 μs erreicht werden, was die aufgestellte Vermutung stützt.

Gerade für hochprioren Verkehr, der ja bevorzugt und zudem mit geringer Latenz und geringem Jitter prozessiert werden soll, sind solche Ausreißer in der Prozessierungslatenz unbedingt zu vermeiden. Eine Lösung für dieses Problem besteht darin, Kollisionen zwischen hochpriorem und niederpriorem Verkehr bei der Berechnung der Flow-ID zu verhindern. Die notwendige Information der Priorität ist im FlexPath-NP bereits durch den *Path Dispatcher* bekannt. Hierzu wird die 12 Bit Flow-ID im Folgenden nur mehr zu 11 Bit aus dem Hashwert gebildet, während ein zusätzliches Bit die Priorität markiert (Spalte b). Bei Wiederholung der Simulationen erkennt man, dass der Maximalwert für den hochprioren Verkehr tatsächlich auf 26,25 μs absinkt.

Ein nächster Schritt zur Differenzierung des Verkehrs ist die zusätzliche Einteilung in Verkehrsklassen (*Traffic Classes*, TC), die sich durch unterschiedliche (zu erwartende) Prozessierungslatenzen auszeichnen. Auch der in der Latenz wesentlich niedrigere *Best Effort*-Forwarding-Verkehr kollidiert schließlich mit dem IPsec-Verkehr. Dadurch erhält auch dieser Verkehr bei einer Kollision eine hohe Systemlatenz. Die maximal gemessene Latenz kann bei Verwendung eines Verkehrsklassen-Bits von 108 μs (Spalte b) auf immerhin rund 70 μs (Spalte c) gesenkt werden, was aufgrund der niedrigen Priorität immer noch deutlich über der maximalen Latenz der hochprioren Pakete liegt. Die Unterscheidung der Verkehrsklassen bedarf einer eingangsseitigen Klassifizierung und Einteilung, wie sie im FlexPath-System jedoch möglich ist. Zudem müssen die ungefähren Prozessierungszeiten der unterschiedlichen Verkehrsklassen für eine sinnvolle Einteilung bekannt sein.

Die durchschnittlichen Systemlatenzen bleiben bei allen drei Varianten nahezu konstant, was zum Großteil an der recht geringen Kollisionswahrscheinlichkeit liegt (vgl. Abbildung 6.8). Es sei hier dennoch erwähnt, dass die Kollisionswahrscheinlichkeit bei konstanter Flow-ID-Bitbreite innerhalb der *Best Effort*-Pakete steigt, da statt der ursprünglichen zwölf Bit nun nur noch zehn Bit für den eigentlichen Hashwert zur Verfügung stehen. Ein Großteil der Pakete (BE-Anteil nahezu 96%) wird nun also statt auf 2^{12} auf 2^{10}

Hashwerte abgebildet. Damit werden für den *Best Effort*-Vekehr Kollisionswahrscheinlichkeiten erreicht, die nahezu den Werten mit 10 Bit in Abbildung 6.8 entsprechen. Bei Kollisionen zwischen gleichartigem Verkehr (gleiche Prozessierungslatenz in den CPUs) sind jedoch keine nennenswerten, negativen Auswirkungen zu erwarten (siehe Abschnitt 6.3.1). Bei den beiden anderen Verkehrsklassen sinkt aufgrund ihres geringen Anteils am Verkehr (IPsec: <1%, QoS: 4%) die Kollisionswahrscheinlichkeit sogar.

6.3.4 Wahl der Zeitspanne beim Timeout

Der Timeout ist, wie bei der Vorstellung der *Aggregation Unit* in Abschnitt 6.2.4 bereits erläutert, ein wichtiges Instrument, um Paketverluste innerhalb eines Systems zu behandeln. Von entscheidender Bedeutung ist dabei die konkrete Wahl der Zeit bis zum Timeout. Sowohl eine zu kurze als auch eine zu lange Timeout-Zeit haben negative Auswirkungen:

- Eine **zu lange Timeout-Zeit** erhöht bei einem Paketverlust die Verweildauer der (unnötigerweise) in der Resequenzierungs-Queue abgespeicherten Folgepakete. Damit erhalten diese Pakete einen unnötigen Aufschlag auf ihre Verarbeitungslatenz. Zudem werden Resequenzierungs-Queues blockiert und damit der Ressourcenbedarf erhöht.
- Eine **zu kurze Timeout-Zeit** führt zu fälschlicherweise als verloren eingestuften Paketen und zwar dann, wenn der Prozessierungsjitter größer als die Timeout-Zeit ist. Diese Pakete können folglich nicht mehr resequenziert werden und verlassen das System als *out-of-order* Pakete.

Die Wahl der Timeout-Zeit wird durch den Prozessierungsjitter, insbesondere aufgeschlüsselt nach den verschiedenen Verkehrsklassen, beeinflusst. Auch die Lastbalancierungsstrategie kann einen entscheidenden Einfluss haben.

Im Folgenden werden deshalb verschiedene Szenarien bewertet. Zu Beginn wird der Einfluss der Lastbalancierungsstrategie auf den Prozessierungsjitter untersucht. Dies dient als Grundlage für die Wahl der Timeout-Zeit für homogenen und heterogenen Verkehr.

Einfluss der Lastbalancierungsstrategie auf den Prozessierungsjitter

Der für die Resequenzierung relevante Jitter in der Prozessierung zwischen *Ingress Tagger* und *Aggregation Unit* teilt sich auf in einen Jitter innerhalb des CPU-Clusters und einen Jitter innerhalb der Prozessierungsqueues vor den CPUs (je nach Füllstand der Queue). Entscheidend für die Resequenzierung bzw. für den Timeout ist dabei der Zeitunterschied, den zwei unmittelbar nacheinander in das System gelangende Pakete eines Flows maximal erreichen können - also die Zeit, um die ein nachfolgendes Paket ein vorhergehendes Paket maximal überholen kann.

Beim Verteilen der Pakete **nach** der Queue, wie es beim *Spraying* bei FlexPath üblich ist (siehe Abbildung 6.11a) erfahren zwar Pakete je nach Queue-Füllstand durchaus

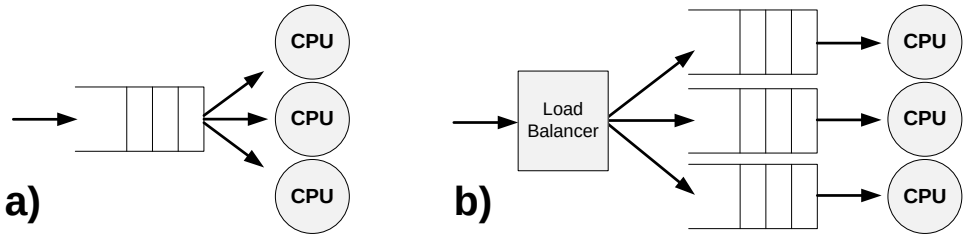

Abbildung 6.11: Einfluss der Lastbalancierung auf den Prozessierungsjitter mit Aufteilung der Pakete nach (a) und vor (b) den Queues.

unterschiedliche Verweilzeiten, diese spielen jedoch keine Rolle, da ein Überholen vor und in der Queue ausgeschlossen ist. Die maximale Zeit, die ein Paket einem in der Sequenz vorangehenden Paket nach der Prozessierung voraus sein kann ergibt sich damit aus dem maximalen Jitter innerhalb des CPU-Clusters und kann wie folgt berechnet werden:

$$\Delta t_{max} = t_{CPU,max} - t_{CPU,min} \tag{6.1}$$

Hierbei ist $t_{CPU,max}$ die maximal mögliche Prozessierungszeit in der CPU, $t_{CPU,min}$ die minimal mögliche. Für Forwarding-Pakete lässt sich die Differenz folgendermaßen bestimmen (vgl. Abschnitt 6.2.1): $\Delta t_{max} = (1,75 - 1,05) \cdot 10\mu s = 7\mu s$.

Im Falle einer Verteilung der Pakete bereits **vor** den Queues (siehe Abbildung 6.11b) ergibt sich ein anderes Bild. Der ungünstigste Fall ergibt sich folgendermaßen: Ein Paket trifft auf eine volle Eingangsqueue. Alle Pakete, die bereits in der Queue warten, werden mit der maximalen Prozessierungszeit $t_{CPU,max}$ bearbeitet. Ein direkt nachfolgendes Paket wird dagegen in eine leere Queue geleitet und direkt mit minimaler CPU-Prozessierungszeit $t_{CPU,min}$ bearbeitet. Damit ergibt sich

$$\Delta t_{max} = (N \cdot t_{CPU,max}) - t_{CPU,min} \tag{6.2}$$

Δt_{max} hängt hier auch direkt von der Queue-Tiefe N ab. Bei einer angenommenen Tiefe von *N=16* ergibt sich damit $\Delta t_{max} = ((16 \cdot 1,75) - 1,05) \cdot 10\mu s = 269,5\mu s$ und liegt damit um einen Faktor von gut 38 höher als im ersten Fall.

Timeout-Zeit bei homogenem Verkehr

Im Falle von homogenem Verkehr ist die Wahl einer sinnvollen Timeout-Zeit vergleichsweise eindeutig. Bei Forwarding-Paketen, die mittels *Spraying*, also nach der Queue verteilt werden, kann bei einer Wartezeit von mehr als 7 μs (siehe vorheriger Abschnitt) mit ziemlicher Sicherheit davon ausgegangen werden, dass das Paket im System verloren ging. Mit einem gewissen Sicherheitspuffer, der auch die variierenden Zugriffszeiten auf den Bus im Simulationsmodell berücksichtigt, scheint somit eine Timeout-Zeit von 10 μs bei *Spraying*-Paketen sinnvoll. Würde statt *Spraying* ein dediziertes Lastbalancierungsverfahren verwendet werden und die Verteilung der Pakete bei Umbalancierungen bereits

vor der Queue stattfinden, müsste entsprechend Formel 6.2 zusätzlich die Queue-Länge berücksichtigt werden.

Timeout-Zeit bei heterogenem Verkehr

Bei heterogenem Verkehr lässt sich innerhalb des FlexPath-NPs eine Einteilung nach Verkehrsklassen vornehmen. Bereits in Abschnitt 6.2 wurde die Möglichkeit genannt, unterschiedliche Verkehrsklassen anhand einzelner Bits innerhalb der Flow-ID zu identifizieren und damit zu separieren. Die notwendigen Informationen können dabei aus der Klassifizierung innerhalb des *Path Dispatchers* erhalten werden. Damit können für alle definierten Verkehrsklassen mit ihren unterschiedlichen Verarbeitungszeiten bzw. -jitter unterschiedliche Timeout-Zeiten bestimmt werden. Die Festlegung der Verkehrsklassen, die Separierung anhand der Flow-ID und die Festlegung einer sinnvollen Timeout-Zeit erfolgt dabei jedoch rein manuell und bedarf eines gewissen Aufwands an Simulationen und/oder Messungen.

Im Beispiel mit hoch- und niederpriorem Forwarding-Verkehr gemischt mit IPsec-Verkehr lässt sich die Timeout-Zeit auf Basis der vorangegangenen Untersuchungen wie folgt wählen:

- Für **Forwarding-Verkehr** ergeben sich die im vorherigen Abschnitt genannten 10μs.
- Bei **IPsec-Verkehr** ergeben sich in einer *worst case*-Betrachtung wesentlich höhere Timeout-Zeiten, da die Prozessierung großen, paketlängenabhängigen Schwankungen unterliegt (siehe Abschnitt 5.4.1). Die maximale Prozessierungszeit (1.518 Byte-Pakete) beträgt 2.967μs (siehe Formel 5.1), die minimale beträgt 422μs (64 Byte-Pakete). Unter Einbeziehung der Queue-Länge von 16 und der Balancierung vor den Queues ergibt sich unter Anwendung von Formel 6.2 eine maximale Schwankung von $((16 \cdot 2.967\mu s) - 422\mu s) = 47.050\mu s$, so dass eine Timeout-Zeit von 50ms als sinnvoll erachtet wird.

6.4 Simulationen und Evaluierung der Hardware-Resequenzierung

Im folgenden Abschnitt wird die Funktionsweise der Hardware-Resequenzierung an unterschiedlichen Simulationen evaluiert. Das Simulationsmodell aus Abbildung 5.13 wird hierzu um die Resequenzierungseinheiten (*Ingress Tagger*, *Aggregation Unit*) ergänzt. Neben der Resequenzierung an sich werden insbesondere auch die Auswirkungen auf den Verkehr (v.a. Latenz) untersucht. Es stehen 16 Resequenzierungs-Queues mit einer Tiefe von acht Paketen (siehe Dimensionierung) zur Verfügung.

Es werden die bereits bekannten Szenarien unterschieden:

- **Szenario 1 (Forwarding):** Hierbei wird der gesamte Verkehr mit *Spraying* auf die CPUs verteilt. Bei allen Paketen wird IP-Forwarding ausgeführt. Die Flow-ID

beträgt 12 Bit und wird komplett aus dem Hash gebildet. Die Timeout-Zeit wird zu 10 μs gewählt.

- **Szenario 2 (Verkehrs-Mix mit Forwarding, QoS und IPsec):** Für den hoch- und niederprioren Forwarding-Verkehr wird *Spraying* verwendet, IPsec-Pakete werden mittels des HLU-Lastbalancierungsverfahrens (siehe Abschnitt 5.2) verteilt. Die Flow-ID setzt sich aus einen 10 Bit-Hashwert plus je ein Bit für Priorität und Verkehrsklasse (Forwarding / IPsec) zusammen. Die Timeout-Zeit wird initial zu 10 μs (Forwarding) und 50 ms (IPsec) gewählt.

6.4.1 Szenario 1: Forwarding-Verkehr

Im ersten Szenario wird lediglich Forwarding-Verkehr verwendet und für die bekannten Verkehrs-Mitschnitte die Resequenzierungs-Rate vor und nach der Resequenzierung bestimmt. Dabei wird die Größe des CPU-Clusters variiert. In Abbildung 6.12 sind die *Packet Reordering*-Raten dargestellt. Betrachtet man den Anteil der *out-of-order* Pakete vor der Resequenzierung (linke y-Achse), so erkennt man, dass der Anteil mit steigender Anzahl an CPUs wächst und sich schließlich einer oberen Grenze annähert. Dieser Effekt wurde bereits in der Messung in Abbildung 5.20 auf Seite 124 beobachtet und erklärt. Die maximale *Reordering Rate* beträgt je nach Verkehrsmittschnitt zwischen 0,22% und 1,28% der Pakete. Damit ist der Anteil der *out-of-order*-Pakete deutlich höher als in Abschnitt 5.4.3 unter Verwendung des noch nicht verfeinerten Modells. Man erkennt, dass nach der Resequenzierung (rechte y-Achse) fast alle Pakete wieder in die korrekte Reihenfolge gebracht werden konnten. Insbesondere bei sehr vielen CPUs wird jedoch keine vollständige Resequenzierung erreicht. Bei einer Analyse dieser Fälle zeigt sich, dass hierbei die Länge der Resequenzierungs-Queues überschritten wurde. Die große Anzahl an CPUs führt dazu, dass bei einem entsprechenden Burst sehr viele Pakete eines Flows parallel bearbeitet werden. Hierbei kommt es in Einzefällen dazu, dass ein Paket mehr als acht vorangegangene Pakete überholt. Innerhalb der *Aggregation Unit* greift daraufhin der in Abschnitt 6.2.4 vorgestellte Mechanismus bei Überschreiten der Queue-Länge. Die Paketreihenfolge kann deshalb nicht mehr vollständig wiederhergestellt werden. Allerdings ist der Anteil nicht sehr groß – im schlimmsten gemessenen Fall mussten sechs Pakete aus 26,5 Millionen (entsprechend einem Anteil von 0,000023%) mit falscher Paketreihenfolge ausgesendet werden. Der Anteil liegt damit im tolerierbaren Bereich. Eine 100%ige Resequenzierung in den genannten Beispielen würde möglich, wenn die Queue-Tiefe auf mindestens zwölf erhöht würde, da der maximale Offset zwölf Pakete betrug.

6.4.2 Szenario 2: Verkehrs-Mix

Im zweiten Szenario wird nun zwischen hoch- und niederpriorem Forwarding- und IPsec-Verkehr unterschieden. Hierbei erfolgt nur eine Betrachtung des Verkehrsmitschnitts mit dem höchsten Anteil an IPsec- und hochprioren Paketen (*OC192_Quarter*, 0,63% IPsec, 7,39% hochpriorer Verkehr, siehe Tabelle 5.3). Die Simulation wird für unterschiedli-

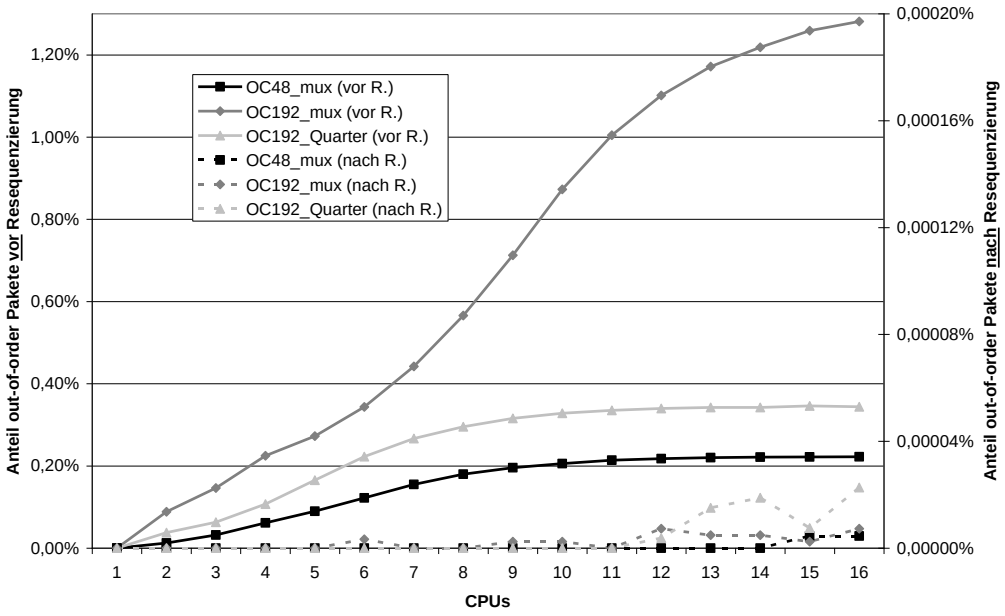

Abbildung 6.12: Anteil der *out-of-order*-Pakete an der Gesamtanzahl **vor** (linke Skala) und **nach** (rechte Skala) der Resequenzierung bei unterschiedlicher CPU-Anzahl (nur Forwarding-Verkehr, 12 Bit Flow-ID).

che CPU-Clustergrößen wiederholt und die Anzahl der *out-of-order* Pakete nach der Resequenzierung gemessen (siehe Tabelle 6.4).

Man erkennt, dass nicht alle Pakete resequenziert wurden und die maximale Anzahl mit 152 bei zwei CPUs auftritt. Der Effekt lässt sich erklären, wenn man den Paketverlust betrachtet. Dieser ist bei zwei CPUs mit 92,5% Verlustrate noch sehr hoch, da bei weitem noch nicht genügend Rechenressourcen zur Bearbeitung des Verkehrs vorhanden sind. Die Paketverluste führen dazu, dass ungewöhnlich viele Resequenzierungs-Queues belegt werden, da die *Aggregation Unit* zunächst auf die verlorenen Pakete wartet. Bei 0,1762% aller Pakete ist daher auch keine Resequenzierungs-Queue frei, obwohl eine benötigt würde. Zwar dürfte der Großteil dieser Pakete wiederum nur aufgrund des Verlusts eines vorangegangenen Pakets als zu resequenzierend eingestuft werden. Dennoch führt dies in Einzelfällen dazu, dass eine tatsächlich notwendige Resequenzierung nicht durchgeführt werden kann. Der Anteil ist mit 0,5250% bei einer CPU zwar noch höher, allerdings kann hier schon aufgrund der seriellen Abarbeitung in nur einer CPU kein *Packet Reordering* auftreten. Mit steigender CPU-Anzahl sinken die Paketverluste sehr schnell und ab fünf CPUs ist auch die Anzahl der Resequenzierungs-Queues ausreichend.

Bei dieser CPU-Anzahl handelt es sich mit Paketverlustraten im hohen ein- und zweistelligen Prozentbereich um keine sinnvolle System-Dimensionierung. Diese wird erst mit Verlusten um 0,01% bei ca. zwölf CPUs erreicht. Wie schon im Kapitel 5 gezeigt, schafft das dedizierte Lastbalancierungsverfahren auch bei mehr CPUs keine vollständige Verlustfreiheit. Dies erklärt auch den ständigen Anteil von ca. 0,003% der Pakete, bei denen

CPUs	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Paketverlust [%]	95,5	92,5	77,8	54,5	31,5	14,8	5,5	1,7
Nicht resequ. Pakete	0	152	4	1	0	2	0	0
Keine Queue frei [%]	,5250	,1762	,0232	,0044	0	0	0	0
Q.-Länge überschr. [%]	,0175	,0264	,0129	,0087	,0081	,0047	,0049	,0041
CPUs	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Paketverlust [%]	0,43	0,10	0,03	0,01	0,01	0,01	0,01	0,01
Nicht resequ. Pakete	0	0	0	1	0	7	3	8
Keine Queue frei [%]	0	0	0	0	0	0	0	0
Q.-Länge überschr. [%]	,0038	,0036	,0036	,0032	,0034	,0034	,0032	,0031

Tabelle 6.4: Kenndaten der Resequenzierung bei unterschiedlicher Anzahl von CPUs (*OC192_Quarter*, Timeout-Zeit 10 μs (Fwd.)/ 50 ms (IPsec)).

die Queue-Länge überschritten wird. Hierbei handelt es sich zum Großteil um Flows mit Paketverlust, bei denen innerhalb der Timeout-Zeit mehr als acht Nachfolge-Pakete eintreffen und somit Pakete vorzeitig ausgesendet werden müssen – in diesem Fall jedoch ohne negative Auswirkungen auf die Paketsequenz. Wie aber auch schon in Szenario 1 gesehen existieren – insbesondere bei sehr vielen CPUs – auch Fälle, bei denen einzelne Pakete aufgrund der Queue-Länge tatsächlich nicht mehr resequenziert werden können. Allerdings ist auch hier dieser Anteil sehr niedrig.

Variation der Timeout-Zeit

Die Timeout-Zeit bei IPsec-Paketen wurde im Abschnitt 6.3.4 anhand der Berechnung des ungünstigsten Falls auf 50 ms festgelegt. Bei dieser *worst case*-Betrachtung wird für den Prozessierungsjitter angenommen, dass einer vollen Queue mit 16 maximal großen Paketen eine leere Queue mit einem Paket minimaler Paketgröße gegenüber steht. Da dieser Fall als sehr unwahrscheinlich angesehen werden kann, wird im Folgenden untersucht, ob diese Zeit ohne negative Auswirkungen auf die Resequenzierung reduziert werden kann und welche Auswirkungen auf die Paketlatenz zu beobachten sind. Dazu werden die Simulationen mit den Timeout-Zeiten 5 ms, 500 μs und 50 μs für IPsec-Pakete wiederholt, während die Timeout-Zeit für Forwarding-Verkehr unverändert bleibt.

In Abbildung 6.13 ist für alle vier Timeout-Zeiten die gemessene Anzahl an *out-of-order*-Paketen nach der Resequenzierung bei unterschiedlicher CPU-Clustergröße dargestellt. Nicht resequenzierte IPsec-Pakete sind dabei schraffiert dargestellt, Forwarding-Pakete dagegen nicht-schraffiert. Man erkennt, dass bei 50 ms Timeout-Zeit bei allen Clustergrößen alle IPsec-Pakete resequenziert werden konnten. Augenfällig ist, dass bei zwei CPUs die Anzahl der nicht resequenzierten Forwarding-Pakete von 152 (50 ms) auf 0 (5 ms - 50 μs) abfällt. Durch die geringere Timeout-Zeit werden die Resequenzierungs-Queues bei Paketverlust wesentlich schneller wieder freigegeben, was den Bedarf in Summe verringert. Allerdings werden nun z.T. nicht mehr alle IPsec-Pakete bei mehreren

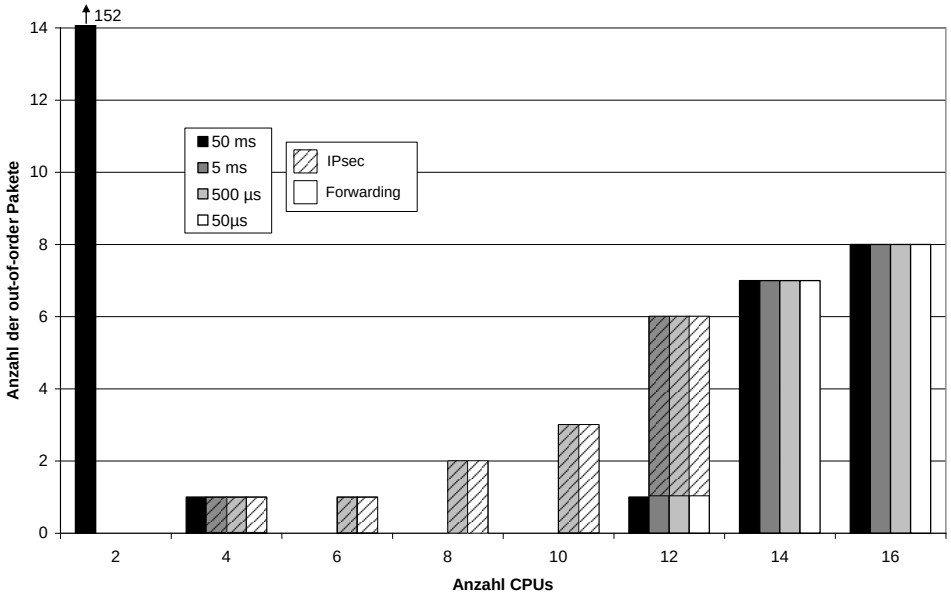

Abbildung 6.13: *Out-of-order*-Pakete nach der Resequenzierung in Abhängigkeit der gewählten Timeout-Zeit für IPsec-Verkehr (*OC192_Quarter*, 12 CPUs, Darstellung für IPsec und Forwarding kumulativ)).

CPUs erfolgreich resequenziert. Die Anzahl der *out-of-order* IPsec-Pakete erreicht sein Maximum mit fünf Paketen bei zwölf CPUs. Der Timeout löst für IPsec offensichtlich bereits zu früh aus. Erstaunlicherweise verschwindet der Effekt bei 14 und 16 CPUs wieder. Allerdings steigt hier für alle Timeout-Zeiten die Anzahl von *out-of-order*-Paketen im Forwarding-Verkehr. Dies wurde bereits beim *Spraying* in Szenario 1 beobachtet.

Der Einfluss der Timeout-Zeit auf das *Packet Reordering* bei IPsec-Paketen scheint also sehr gering. Die Aussage relativiert sich jedoch bei einem Blick auf die Anzahl der *out-of-order*-Pakete **vor** der Resequenzierung. Bei zwölf CPUs stehen hierbei 158.750 Forwarding-Paketen lediglich fünf IPsec-Paketen gegenüber. Mit anderen Worten: während bei einer Timeout Zeit von 50 ms alle IPsec-Pakete resequenziert werden konnten, gelang dies bei zwölf CPUs bei 5 ms und weniger bei keinem einzigen. Dennoch: Die Anzahl der *out-of-order* IPsec-Pakete ist bereits ohne Resequenzierung sehr niedrig. Dies verwundert nicht, wenn man bedenkt, dass diese Pakete durch ein dediziertes Lastbalancierungsverfahren zugewiesen wurden, bei dem Paketüberholungen ausschließlich bei Umbalancierungen möglich sind. Je nach den Anforderungen kann hierbei also individuell abgewogen werden, ob eine Resequenzierung von dediziert zugewiesenen Verkehr überhaupt sinnvoll ist oder sie sich nur auf *Spraying*-Verkehr beschränken soll.

	50 ms	**5 ms**	**500 μs**	**50 μs**	**ohne Resequ.**
IPsec	3,583 ms	3,135 ms	3,104 ms	3,102 ms	3,102 ms
QoS (Fwd.)	15,27 μs	15,27 μs	15,27 μs	15,27 μs	15,27 μs
BE (Fwd.)	15,16 μs	15,16 μs	15,16 μs	15,16 μs	15,15 μs

Tabelle 6.5: Durchschnittliche Latenzen aufgeschlüsselt nach Verkehrsklassen in Abhängigkeit von der IPsec-Timeout-Zeit bzw. ohne Resequenzierung (*OC192_Quarter*, 12 CPUs).

Einfluss der Resequenzierung auf die Paketlatenz

Als nächstes wird der Einfluss der Resequenzierung bei unterschiedlichen Timeout-Zeiten auf die Paketlatenzen untersucht. In Tabelle 6.5 sind die durchschnittlichen Paketlatenzen für Timeout-Zeiten von 50 ms, 5 ms, 500 μs und 50 μs und zusätzlich ohne Resequenzierung als Referenz gegeben. Man erkennt, dass die Resequenzierung bei den Forwarding-Paketen keinen messbaren Einfluss hat. Die Werte sind für alle Fälle nahezu konstant bei 15,27 μs bzw. 15,16 μs. Dagegen ist der Einfluss bei den IPsec-Paketen wesentlich größer. Dabei spielen die echten Resequenzierungen keine allzu große Rolle – wie erläutert handelt es sich lediglich um einige wenige. Einen viel größeren Effekt haben die Paketverluste und die damit verbundene Wartezeit. Demnach steigert sich die durchschnittliche Paketlatenz von 3,10 ms ohne Resequenzierung bis auf 3,58 ms bei einer Timeout-Zeit von 50 ms. Durch eine kleinere Timeout-Zeit kann die Paketlatenz entsprechend verringert werden (z.B. 5 ms: 3,14 ms), allerdings – wie im vorigen Abschnitt erläutert – auf Kosten der Resequenzierungsgüte.

Interessant sind ferner die Auswirkungen auf die maximal gemessenen Paketlatenzen. In Abbildung 6.14 ist der zeitliche Verlauf der maximalen Latenz für hoch- und niederprioren Forwarding-Verkehr jeweils mit und ohne Resequenzierung dargestellt. Die hochprioren Pakete besitzen dabei mit Werten um die 25 μs einen verhältnismäßig stabilen Verlauf. Niederpriore Pakete, die niedriger als die IPsec-Pakete eingestuft sind, erfahren dagegen viel größere Schwankungen. In der Spitze werden Werte von nahezu 160 μs erreicht. Die Kurven mit und ohne Resequenzierung sind dabei nahezu deckungsgleich. Lediglich bei den *Best Effort*-Paketen erkennt man stellenweise minimale Abweichungen. Die Resequenzierung selbst hat damit keine nennenswerte Auswirkung auf die maximale Latenz. Vielmehr scheinen hier Queue-Wartezeiten den dominanten Einfluss darzustellen. Der große Unterschied in der maximalen Latenz verdeutlicht auch den Nutzen der Separierung von hoch- und niederpriorem Verkehr durch ein zusätzliches Bit in der Flow-ID. Auch wenn Kollisionen recht selten vorkommen und damit keine großen Auswirkungen auf die durchschnittliche Latenz haben, so erfahren einzelne hochpriore Pakete ohne Separierung durchaus Latenzen im Bereich der *Best Effort*-Pakete.

In Abbildung 6.15 sind noch einmal zusammenfassend die durchschnittlichen und maximalen Latenzen der drei Verkehrsklassen bei variabler CPU-Clustergröße dargestellt. Die Timeout-Zeit für IPsec beträgt dabei wiederum 50 ms. Gerade in extremer Überlast bei einer CPU werden die Auswirkungen der unterschiedlichen Priorisierung deutlich.

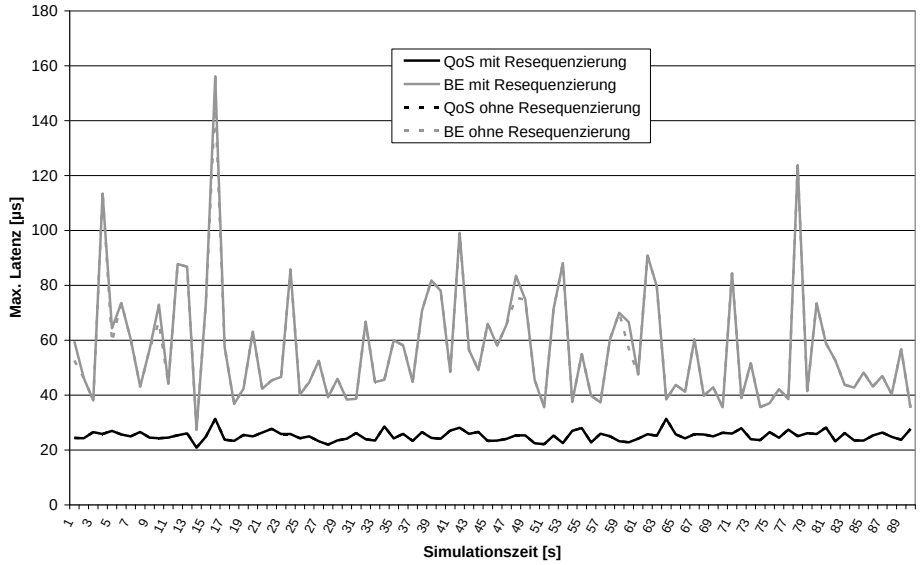

Abbildung 6.14: Maximale Latenz der Forwarding-Pakete (Hoch-/niederprior (QoS, BE)) über der Simulationszeit jeweils innerhalb einer Sekunde (*OC192_Quarter*, 12 CPUs, Timeout-Zeit 50 ms für IPsec-Pakete).

Auch wenn die hochprioren Pakete einen deutlichen Anstieg der Latenz verzeichnen, so liegt diese dennoch deutlich unterhalb der Latenz der *Best Effort*-Pakete, die kaum noch bedient werden. Die durchschnittlichen Latenzen der beiden Forwarding-Klassen nähern sich jedoch mit steigender CPU-Anzahl schnell an. Hierbei zeigt sich erneut die Effizienz des *Spraying*-Mechanismus. Trotz der niederen Priorität können die *Best Effort*-Pakete, die durch IPsec jeweils nicht optimal ausgenutzten CPUs verwenden. Damit erreichen sie z.B. bei neun CPUs trotz hoher Auslastung des Systems (Paketverlust 0,43%) eine Latenz von durchschnittlich 17,4 μs – im Vergleich zu 15,7 μs der hochprioren Pakete. Allerdings erfahren selbst die hochprioren Pakete bei ungünstigen Konstellationen eine hohe maximale Latenz. Bei elf CPUs steigt diese auf immerhin 157,3 μs. Offensichtlich werden hier kurzzeitig alle CPUs von IPsec-Paketen besetzt. Die Nicht-Unterbrechbarkeit der Prozessierung führt in Einzelfällen somit zu einer hohen maximalen Latenz. Abhilfe könnte hier eine exklusiv für (hochprioren) Forwarding-Verkehr reservierte CPU leisten.

6.5 Implementierung

Die Implementierung der Hardware-Resequenzierungseinheit wurde für den Virtex-4 FPGA durchgeführt (siehe hierzu auch [82] und [83]).

Die Übertragung der Flow-ID und Sequenznummer erfolgt innerhalb des FlexPath-NP mit Hilfe des bereits in Abschnitt 4.2 vorgestellten Paket-Deskriptors. Hierbei wurde ein zwölf Bit breites Feld für den Flow-Hash-Wert reserviert, ferner je ein Bit für Priorität

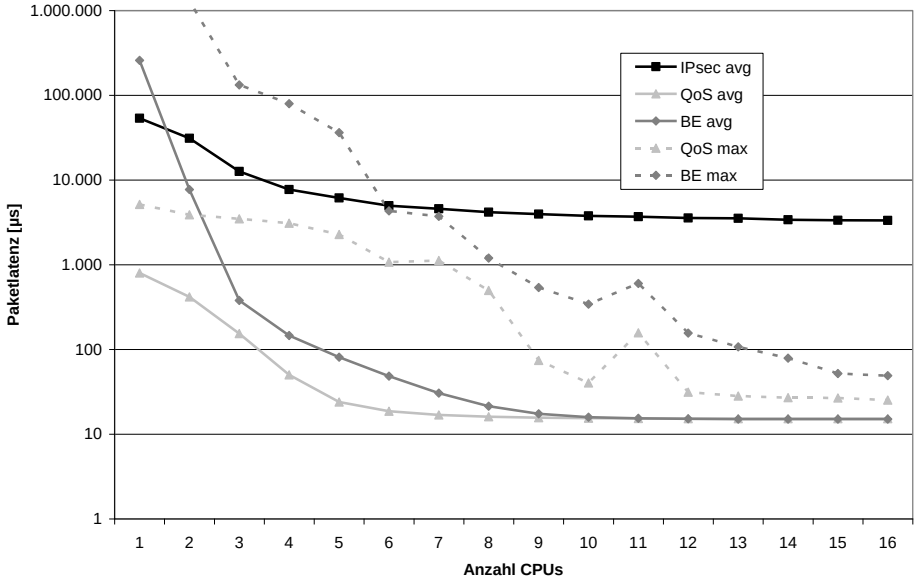

Abbildung 6.15: Durchschnittliche (avg) und maximale (max) Latenzen bei unterschiedlicher Anzahl an CPUs aufgeschlüsselt nach Verkehrsklassen (*OC192_Quarter*, Timeout-Zeit 10 μs (Fwd.)/ 50 ms (IPsec)).

(PR) und Verkehrsklasse (TC). Von den 14 Bit werden zur Resequenzierung letztlich jedoch nur 12 Bit genutzt (10 Bit Hash, PR, TC). Die acht Bit breite Sequenznummer ist im zweiten Teil des Paket-Deskriptors untergebracht (siehe Abbildung 6.16).

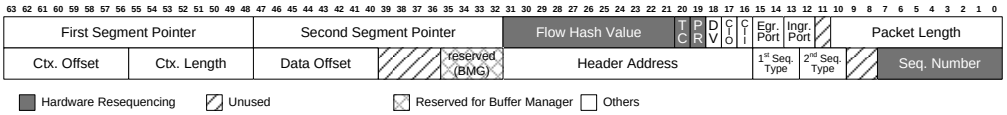

Abbildung 6.16: Aufbau des Paket-Deskriptors in FlexPath mit Flow-ID und Sequenznummer

Der *Ingress Tagger* (siehe Abschnitt 6.2.3) konnte für die Implementierung innerhalb des FlexPath-Systems zusätzlich vereinfacht werden. Nachdem ein Flow-Hashwert bereits für die Lastbalancierung im Eingangsdatenpfad verwendet wird, wird dieser bereits im *Pre-Processor* berechnet und in den Paket-Deskriptor eingesetzt. Ebenso wird auch die Priorität und die Verkehrsklasse dort bestimmt und eingetragen. Die Aufgabe des *Ingress Taggers* beschränkt sich damit auf das Bestimmen und Einfügen der Sequenznummer. Hierfür wurde ein Speicher implementiert, der für alle 4.095 möglichen Flow-IDs die aktuelle Sequenznummer enthält. Die jeweilige Sequenznummer wird beim Eintreffen eines Paket-Deskriptors übernommen, inkrementiert und wieder abgespeichert. Die Implementierung des Sequenznummer-Speichers erfolgt mithilfe der FPGA-internen 16 kBit SRAM-Blöcke (BRAMs).

6.5.1 Ressourcen-Verbrauch

Der *Ingress Tagger* hat auf dem Virtex-4 einen Ressourcenverbrauch von 119 Slices, 173 Flip-Flops und zwei BRAMs. Die maximale Taktfrequenz liegt bei rund 264 MHz (siehe Tabelle 6.6).

Die *Aggregation Unit* ist die sowohl vom Platzverbrauch als auch von der Komplexität her aufwändigere Einheit. Ihr grundsätzlicher Aufbau ist in Abbildung 6.17 dargestellt.

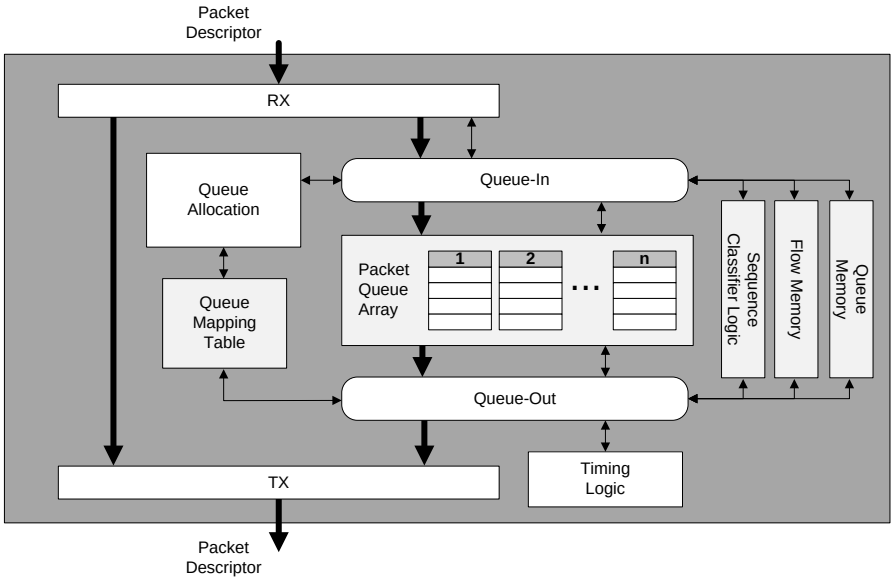

Abbildung 6.17: Blockdiagramm der *Aggregation Unit*

Die eingangsseitigen Pakete werden beim Empfang auf ihre Flow-ID überprüft. Pakete mit Flow-ID 0 werden direkt zum Ausgang weitergeleitet. Für alle anderen Pakete wird über den *Queue-In*-Prozess die Sequenznummer überprüft. Bei Bedarf übernimmt der Prozess den Paket-Deskriptor und speichert ihn im Paketspeicher ab (*Packet Queue Array*). Neue Queues werden mit Hilfe der *Queue Allocation* angefordert. Ausgangsseitig sorgt der *Queue-Out*-Prozess für das Auslesen der Paket-Deskriptoren. Er wird entweder über die Eingangsseite gestartet (Auslesen der Queue nach erfolgreicher Resequenzierung) oder über den Timeout der *Timing Logic*. Sowohl *Queue-In*- als auch *Queue-Out*-Prozess haben dabei Zugriff auf mehrere Datenstrukturen, die neben den Sequenznummern u.a. die Zuordnung von Flow-ID zu Queue bzw. umgekehrt enthalten. Konkurrierende Zugriffe sind dabei über einen Semaphor-Schutz abgesichert. Die aktuelle Implementierung besitzt gegenüber den durchgeführten Simulationen die Limitierung, dass nur einheitliche Timeout-Zeiten für alle Flows unterstützt werden. Die *Aggregation Unit* benötigt 1.145 Virtex-4 Slices, 1.042 Flip-Flops und elf BRAMs (siehe Tabelle 6.6).

Der Speicherverbrauch ist nochmals detailliert in Tabelle 6.7 angegeben. Den größten Anteil hat mit 7.168 Byte die *Flow Table*, die neben den aktuellen Sequenznummern Zeiger auf evtl. zugeordnete Resequenzierungs-Queues enthält. Zusätzliche 2 KByte an Spei-

	Ingress Tagger	Aggregation Unit
Slices	119	1.145
Flip-Flops	173	1.042
4-input LUTs	70	2.204
BRAMs	2	11
Max. Frequenz [MHz]	263,8	144,8

Tabelle 6.6: Ressourcenverbrauch von *Ingress Tagger* und *Aggregation Unit* auf Xilinx Virtex-4 FX (12 Bit Flow-ID).

Speicher	**Tiefe [Bit]**	**Breite [Bit]**	**Größe [Byte]**	**BRAMs**
Flow Table (Sequenznummer, Queue-Zeiger)	4.096	14	7.168	4
Queue Memory (Start-Zeiger, Paketzähler)	16	7	14	1
Paketspeicher	128	128	2.048	4
Queue Mapping Table (Zuordnung Queue → Flow-ID)	16	12	24	1
Queue Free List	16	4	8	1
Gesamt			**9.262**	**11**

Tabelle 6.7: Speicherverbrauch innerhalb der *Aggregation Unit*.

cher werden für die eigentlichen Resequenzierungs-Queues benötigt. Insgesamt benötigt die *Aggregation Unit* damit theoretisch 9.262 Byte an Speicher. Aufgrund der vorgegebenen SRAM-Blockstruktur werden im FPGA elf Blöcke (entsprechend 22 KByte) benötigt. Die maximale Taktfrequenz der *Aggregation Unit* beträgt rund 145 MHz.

6.5.2 Paket- und Datendurchsatz

Ein weiteres, wichtiges Kriterium bei der Beurteilung der Implementierung stellt der Paketdurchsatz durch die Resequenzierungseinheit dar. Die aktuelle Implementierung bearbeitet jedes Paket vollständig. Ein weiteres Paket kann daher am Eingang erst angenommen werden, sobald das vorherige Paket weiterversendet bzw. abgespeichert wurde. Es handelt sich hierbei also nicht um eine Verarbeitungs-Pipeline. Die minimale Dauer zwischen zwei Paketen hängt dabei von den in der Resequenzierungseinheit durchzuführenden Aktionen ab. Dabei sind zu unterscheiden:

- **Kein Packet Reordering:** Pakete können ohne Resequenzierung versendet werden.
- **Packet Reordering:** Pakete müssen zwischengespeichert und später versendet werden.

- **Paketverluste:** Pakete müssen bis zum Timeout bzw. Queue-Überlauf zwischengespeichert werden.

Im ersten Fall (keine Resequenzierung) ergibt sich eine minimale Zeit zwischen zwei Paketen (von Start zu Start) von elf Takten – entsprechend 110 ns bei einer Taktfrequenz von 100 MHz. Dies schließt die Überprüfung und Inkrementierung der Sequenznummer und das Aussenden des Pakets mit ein. Damit ergibt sich eine Paketrate von 9,09 Millionen Paketen pro Sekunde (Mpps). Hierbei sei nochmals erwähnt, dass die Resequenzierungseinheit nur Paket-Deskriptoren verarbeitet. Somit ist der Paketdurchsatz unabhängig von der Paketlänge. Der maximal erreichbare Datendurchsatz liegt entsprechend bei 4,65 GBit/s für 64 Byte-Pakete, bzw. 110,39 GBit/s für 1518 Byte-Pakete und 24,72 GBit/s bei 340 Byte-Paketen (*Simple IMIX*-Durchschnittsgröße).

Bei einer Resequenzierung um eine Stelle erhöht sich der Bedarf auf zwölf Takte für das erste Paket (beinhaltet das Anlegen einer Resequenzierungs-Queue) und 13 Takte für das zweite Paket (beinhaltet das Aussenden der beiden Paket-Deskriptoren und Freigabe der Resequenzierungs-Queue). Dadurch ergibt sich also ein zusätzlicher Aufwand von drei Takten ($12+13-(2\cdot 11)=3$). Bei einer Verschiebung um mehr als eine Stelle ergibt sich für jede Stelle ein zusätzlicher Aufwand beim letzten Paket von drei Takten zum Versenden des zusätzlichen Paket-Deskriptors. Allerdings verkürzt sich die Verarbeitung der mittleren Pakete von elf auf acht Takte. Dies kann damit begründet werden, dass weder das Paket versendet, noch eine Resequenzierungs-Queue neu angefordert werden muss. Damit gleichen sich beide Effekte aus. Somit kann pro *out-of-order*-Paket (unabhängig vom Grad des *Reorderings*) ein konstanter Zusatzaufwand von drei Takten angesetzt werden.

Bei Paketverlusten müssen zwei Fälle unterschieden werden. Werden bis zum Timeout mehr Pakete desselben Flows empfangen, als die Resequenzierungs-Queue aufnehmen kann, so wird diese vorzeitig geleert (siehe Abschnitt 6.2.4). Für das erste Paket sind hierbei aufgrund der Allokation der Resequenzierungs-Queue wiederum zwölf Takte zur vollständigen Bearbeitung notwendig, für die folgenden Pakete dagegen nur acht Takte. Beim neunten Paket wird die Resequenzierungs-Queue geleert und die vorhandenen Pakete ausgesendet. Hierfür werden 43 Takte benötigt. Insgesamt werden also $12 + 7 \cdot 8 + 43 = 111$ Takte verbraucht und damit zwölf mehr als ohne Paketverlust ($9 \cdot 11 = 99$). Sofern der Timeout vor Eintreffen des neunten Pakets eintritt, benötigt dieser zehn Takte zum Aussenden des ersten Paketes plus drei weitere Takte für jedes zusätzliche Paket in der Queue. Es wird also ein konstanter Zuschlag von elf Takten pro verlorenem Paket fällig. Aus beiden Fällen ergibt sich bei Paketverlust ein Maximum von zusätzlich zwölf benötigten Takten.

Die resultierenden Paketraten sind in Abhängigkeit vom *Packet Reordering*-Anteil bzw. den Paketverlusten in Tabelle 6.8 wiedergegeben. Bei einer *Reordering*-Rate von 1,5% (vgl. auch Abbildung 6.12) verringert sich der Paketdurchsatz geringfügig von 9,09 Mpps ohne *Reordering* auf dann 9,05 Mpps (-0,44%).

	Takte/Paket	Max. P.rate	Anmerkungen
Kein P. Reordering	11	9,09 Mpps	
Packet Reordering	$11 + (x \cdot 3)$	$\frac{100}{(11+x \cdot 3)}$ Mpps	*x: P. Reordering Anteil*
Paketverluste	$11 + (y \cdot 12)$	$\frac{100}{(11+y \cdot 12)}$ Mpps	*y: Anteil Paketverluste*

Tabelle 6.8: Maximal erreichbare Paketraten der *Aggregation Unit* in Abhängigkeit von *Packet Reordering* bzw. Paketverlusten.

6.6 Ausblick: Adaptive Anpassung der Timeout-Zeit

Die Wahl der Timeout-Zeit bei heterogenem Verkehr Bedarf einer genauen Analyse und einer expliziten Einteilung in Verkehrsklassen (siehe Abschnitt 6.3.4). Innerhalb des FlexPath-NPs ist die Einteilung in Verkehrsklassen dank der Klassifizierung im *Path Dispatcher* möglich. Problematisch bei der Wahl der Timeout-Zeit bleibt jedoch die weitestgehend manuelle Bestimmung des Prozessierungsjitters, welche neben einer genauen Analyse der Paket-Applikationen und damit einhergehender CPU-Prozessierungszeiten zudem eine gute Kenntnis des Systems (z.B. Queue-Größen) und des Lastbalancierungsverfahrens notwendig macht.

Als mögliche Erweiterung wird aus diesem Grunde ein heuristisches, selbst-adaptives Verfahren zur automatischen Bestimmung der Timeout-Zeit vorgestellt. Dieses basiert darauf, dass sich die Timeout-Zeit automatisch pro Flow-ID auf den aktuellen Prozessierungsjitter einstellt. Eine manuelle Bestimmung des Jitters entfällt damit.

6.6.1 Algorithmus

Das Verfahren läuft folgendermaßen ab:

- Für jedes Paket wird ausgangsseitig die Latenz, die das Paket im System erfahren hat, ausgewertet. Hierzu erhält jedes Paket am Eingang einen Zeitstempel, den es innerhalb des Systems mitführen muss.
- Für jede Flow-ID wird jeweils die maximale und minimale Latenz gespeichert. Die Timeout-Zeit bestimmt sich aus der Differenz von Maximal- und Minimalwert multipliziert mit einem Sicherheitsfaktor x $(1,0 < x < 2,0)$.
- Die Charakteristik der Latenzen kann sich aufgrund wechselnder Bedingungen (z.B. sinkende Systemlast) ändern. Um sich den aktuellen Bedingungen anzupassen, dürfen einmal erreichte Maximal- und Minimalwerte nicht dauerhaft gespeichert bleiben, sondern müssen sich nach bestimmten Regeln mit der Zeit wieder annähern.
- Die Bestimmung der Maximal- und Minimalzeit erfolgt pro Paket einer Flow-ID nach folgenden Regeln:

$$t_{max} = MAX(t_{akt}, (t_{max} - [(t_{max} - t_{min}) \cdot m])) \tag{6.3}$$

$$t_{min} = MIN(t_{akt}, (t_{min} + [(t_{max} - t_{min}) \cdot n])) \tag{6.4}$$

Übersteigt der aktuelle Latenzwert t_{akt} den bisher gespeicherten Maximalwert t_{max}, so wird dieser sofort übernommen. Ansonsten gleichen sich die Werte schrittweise wieder an. Das Gleiche gilt entsprechend für t_{min}. Die Faktoren `m` und `n` bestimmen dabei die Geschwindigkeit der Angleichung.

Die Funktion ähnelt dabei einer Tiefpass-Filterung. Allerdings werden neue maximale oder minimale Extremwerte sofort übernommen. Damit kann der zur Timeout-Bestimmung verwendete Maximalwert t_{max} nie unter dem zuletzt tatsächlich gemessenen Maximalwert t_{akt} fallen (es kommt also zu keinem verzögerten Anstieg). Damit reagiert der Algorithmus sofort und ohne Mittelung auf ein Ansteigen des Prozessierungsjitters. Durch die direkte Übernahme der Extremwerte wird die Timeout-Zeit damit tendenziell hoch angesetzt.

Die Wahl von `m` und `n` bedarf einer genaueren Untersuchung. Die beiden Faktoren haben einen wesentlichen Anteil daran, wie schnell sich die Timeout-Zeit dem jeweils aktuellen Prozessierungsjitter angleicht. Ein sehr kleiner Faktor führt zu einer sehr langsamen Anpassung von Minimal- und Maximalwert. Dies führt zu tendenziell größeren Timeout-Zeiten und damit einer sichereren Resequenzierung, allerdings auf Kosten höherer Latenzen bei Paketverlusten. Ein größerer Faktor führt im Umkehrschluss zu einer schnelleren Adaption mit höherer Gefahr, die Timeout-Zeit zu klein zu wählen und damit *Packet Reordering* zu riskieren. Eine abschließende Untersuchung wird im Rahmen dieser Arbeit nicht durchgeführt. Im Folgenden sollen jedoch erste Abschätzungen gegeben werden.

6.6.2 Simulation

Das adaptive Verfahren wurde in das Simulationsmodell integriert und mit dem Verkehrsmitschnitt *OC48_mux* (IP-Forwarding) bei acht CPUs getestet. In Abbildung 6.18 ist für einen zufällig ausgewählten Flow der zeitliche Verlauf von t_{max} und t_{min} dargestellt. Zudem sind die in jeder Sekunde jeweils gemessenen maximalen und minimalen Werte $t_{akt,max}$ und $t_{akt,min}$ aufgezeichnet. Die Abbildung bezieht sich auf reinen Forwarding-Verkehr und wurde jeweils für `m` und `n` mit Wert 10^{-3} (Abb. 6.18a) und Wert 10^{-4} (Abb. 6.18b) wiederholt.

Bei beiden Kurven erkennt man die zeitlich sehr geringe Varianz der minimalen Werte. Diese werden bei leerer Queue erreicht und stellen im Wesentlichen die reine Prozessor-Latenz dar. Eine leere Queue sollte in einem nicht überlasteten System regelmäßig erreicht werden, so dass der konstante Wert nicht weiter verwundert. Bei den maximalen Werten spielt dagegen der Queue-Füllstand eine größere Rolle. Eine Häufung von Paketen (Bursts) führt zur kurzzeitigen Überlastung des Clusters, was zum Anstieg der Latenz führt. Folgerichtig ergeben sich größere Ausschläge bei der maximalen Latenz. Man sieht im Fall a, wie t_{max} einem Ausschlag von t_{akt} unmittelbar folgt und sich anschließend den reduzierten Werten von t_{akt} wieder annähert. Im Fall b erfolgt die Annäherung dagegen wesentlich langsamer.

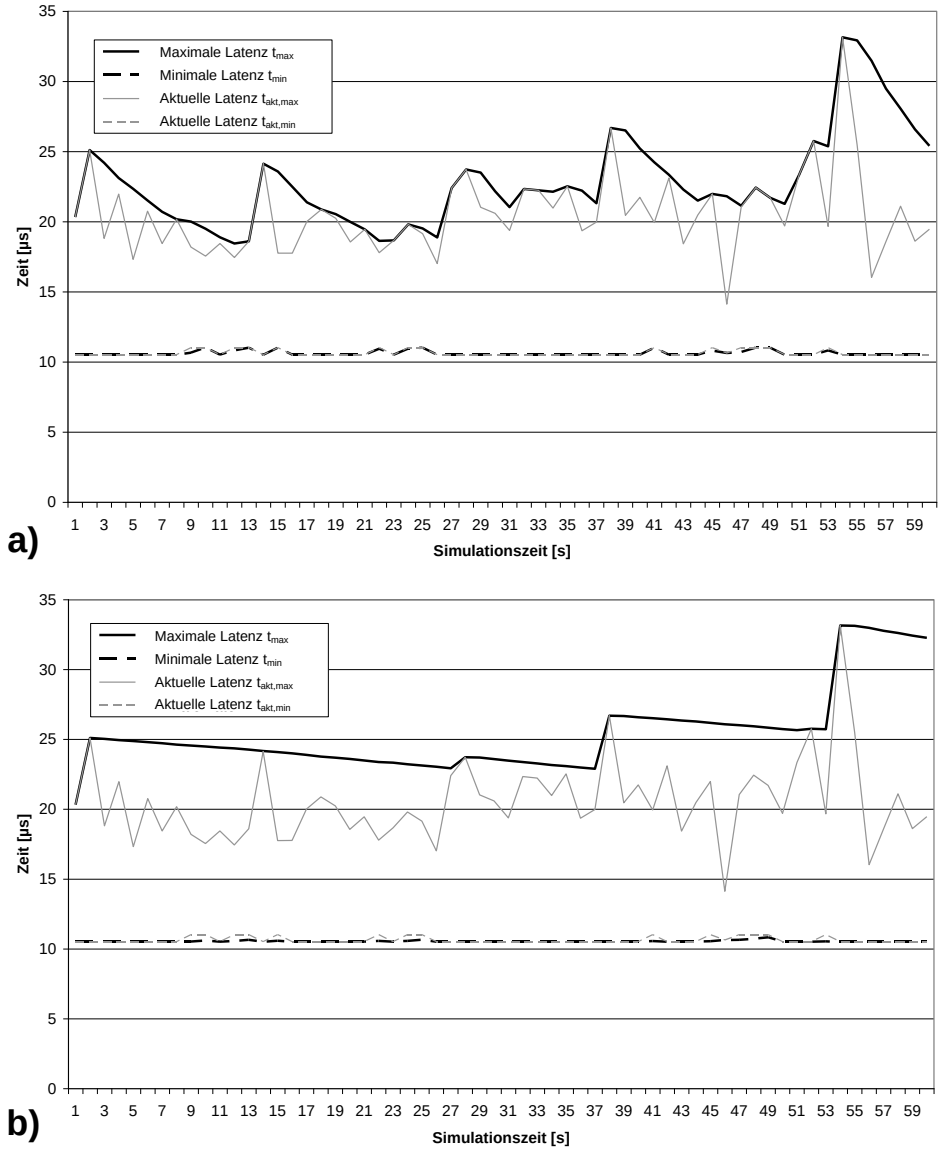

Abbildung 6.18: Minimale und maximale Prozessierungslatenz jeweils absolut gemessen und Tiefpass-gefiltert nach beschriebenen Algorithmus mit $m = n = 10^{-3}$ (a) und $m = n = 10^{-4}$ (b) für eine zufällig ausgewählte Flow-ID (*OC48_mux*, 8 CPUs, nur Forwarding-Verkehr).

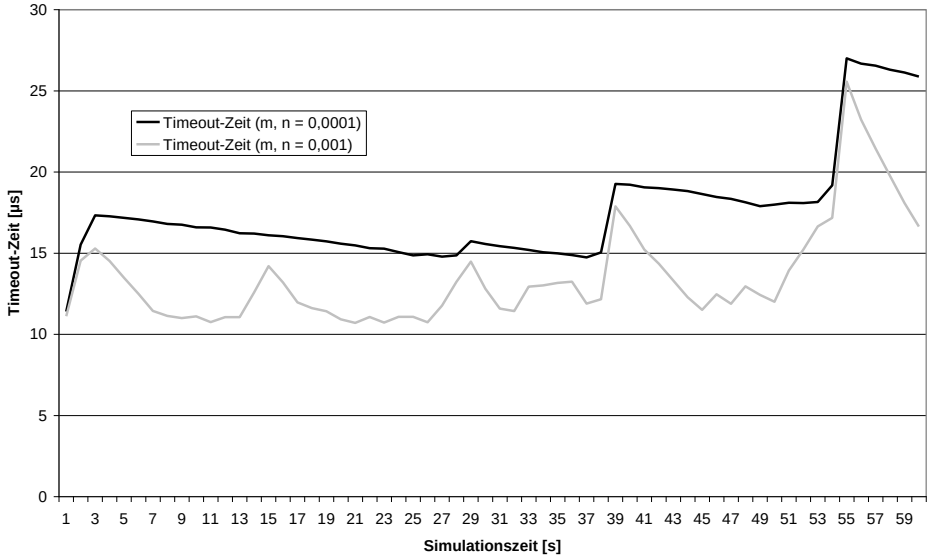

Abbildung 6.19: Timeout-Zeit mit $m = n = 10^{-3}$ bzw. $m = n = 10^{-4}$ für eine zufällig ausgewählte Flow-ID (*OC48_mux*, 8 CPUs, nur Forwarding-Verkehr).

In Abbildung 6.19 ist für beide Werte von `m` und `n` die resultierende Timeout-Zeit (jeweils gemittelt über eine Sekunde) dargestellt. Der Sicherheitsfaktor `x` wurde zu 1,2 gewählt. Damit liegt die Timeout-Zeit also 20% über der aktuellen Differenz von t_{max} und t_{min}. Die ermittelten Werte schwanken zwischen 10,7 μs und knapp 26 μs und liegen damit zum Teil deutlich über dem in Abschnitt 6.3.4 festgelegten Wert von 10 μs. Im gezeigten Beispiel konnten alle Pakete erfolgreich resequenziert werden. Die tatsächliche Paketlatenz wurde aufgrund der nicht vorhandenen Paketverluste bei *Spraying* nicht negativ beeinflusst.

Im nächsten Schritt wurde das Verhalten beim bekannten Verkehrs-Mix mit IPsec- und hochprioren Forwarding-Paketen untersucht. Die Messungen wurden mit dem Verkehrsmitschnitt mit dem höchsten IPsec-Anteil (*OC192_Quarter*, siehe Tabelle 5.3) bei zwölf CPUs durchgeführt. Der zeitliche Verlauf der Timeout-Zeit eines zufällig ausgewählten IPsec-Flows ist in Abbildung 6.20 dargestellt. Man erkennt, wie sehr die Timeout-Zeit im Verlauf der Simulation schwankt. Bei einem Sicherheitsfaktor x von 1,2 werden Werte zwischen 16,4 ms und 65,4 ms erreicht. Damit liegt der Timeout für diesen Flow die meiste Zeit unter den in Abschnitt 6.3.4 ursprünglich gewählten 50 ms. Durch die geringere Timeout-Zeit wird auch eine etwas geringere durchschnittliche Systemlatenz der IPsec-Pakete von 3,55 ms (Latenz bei festen Timeout von 50 ms: 3,58 ms, siehe Tabelle 6.3) erreicht. Allerdings wurden drei IPsec-Pakete *out-of-order* ausgesendet. Durch Reduzierung des Sicherheitsfaktors auf 1,0 kann eine weitere Absenkung der durchschnittlichen Latenz auf 3,48 ms erreicht werden (ebenfalls drei *out-of-order* Pakete).

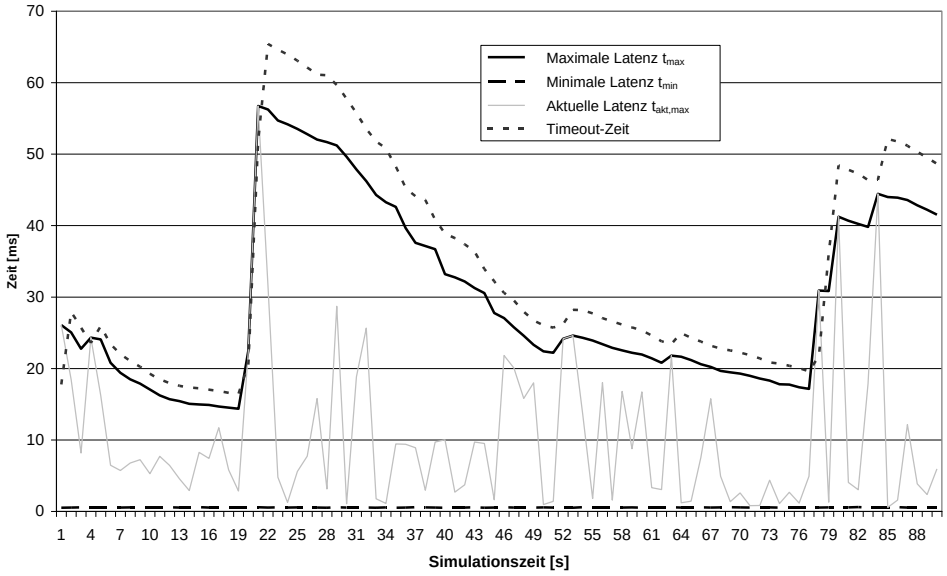

Abbildung 6.20: Zeitlicher Verlauf von Latenzen und Timeout-Zeit für einen zufällig ausgewählten IPsec-Flow (*OC192_Quarter*, 12 CPUs, $m = n = 10^{-3}$, Verkehrs-Mix, x=1,2).

6.6.3 Bewertung des Verfahrens

Mit dem adaptiven Verfahren wurde ein Algorithmus vorgestellt, der die Timeout-Zeit den aktuellen Verhältnissen und Latenzen im System anpasst und somit eine manuelle Anpassung unnötig macht. Es bedarf hierbei keiner tieferen manuellen Analyse des Gesamt-Systems. Das Verfahren kann sowohl in Kombination mit FlexPath und unterschiedlichen Verkehrsklassen als auch in Nicht-FlexPath-Systemen eingesetzt werden. Bei Anwendung ohne FlexPath ist darauf zu achten, dass Kollisionen zwischen Verkehrsklassen mit stark unterschiedlicher Prozessierungszeit so gut wie möglich vermieden werden (z.B. größere Flow-ID-Bitbreiten). Insbesondere Kollisionen zwischen Flows mit hoher und niedriger Latenz haben die bereits beschriebenen negativen Auswirkungen.

Das Verfahren liefert bei geeigneter Dimensionierung zwar brauchbare Ergebnisse, allerdings teilweise auch eher konservative Timeout-Zeiten. Insbesondere bei *Spraying* werden zu hohe Werte kalkuliert. Bei *Spraying* ist für die Resequenzierung nur der Prozessierungsjitter nach der Queue (also im CPU-Cluster) von Bedeutung, da nur dort Paketüberholungen stattfinden können (siehe Abschnitt 6.3). Das beschriebene Verfahren kalkuliert die Timeout-Zeit jedoch anhand des maximalen Systemjitters, also auch unter Berücksichtigung des Jitters in den Queues. Damit werden für *Spraying*-Pakete wie in Abbildung 6.19 dargestellt, Timeout-Zeiten von über 25 μs erreicht, obwohl Werte um 10 μs ausreichend wären.

Durch Variation der Systemparametern (m, n, x) ergeben sich noch eine Reihe von Optimierungsmöglichkeiten. Durch die direkte Übernahme der Extremwerte reagiert das

Verfahren sehr heftig auf Einzelereignisse. Ein einzelnes Paket mit sehr hoher Latenz erhöht direkt den Maximalwert und damit die Timeout-Zeit. Es bleibt zu untersuchen, ob eine höhere Dämpfung und Mittelung über einen längeren Zeitraum evtl. bessere und stabilere Ergebnisse erreichen könnte.

Nachteile ergeben sich für das Verfahren insbesondere bei der praktischen Umsetzung. Die Bestimmung der Systemlatenz erfordert, dass jedes Paket einen geeigneten Zeitstempel innerhalb des Systems mit sich führt. Dieser muss ausgangsseitig mit Minimal- und Maximalwert der Flow-ID verglichen werden. Hierzu müssen beide Werte pro Flow-ID gespeichert werden.

6.7 Bewertung und Zusammenfassung

In diesem Kapitel wurde ein Hardware-Resequenzierungsverfahren vorgestellt, mit dem die originale Reihenfolge der Pakete auf Flow-Ebene effektiv und zuverlässig wiederhergestellt werden kann. Es konnte gezeigt werden, dass sich das Verfahren insbesondere für *Spraying*-Pakete, die in der Regel eine recht gleichmäßige Verarbeitung erfahren, besonders eignet. Genau diese Pakete erfahren bei dem in Kapitel 5.2.1 vorgestellten Lastbalancierungsverfahren das höchste Maß an *Packet Reordering* und sind damit auf die Resequenzierung besonders angewiesen. *Spraying* erzeugt dabei eine ideale Balancierung des CPU-Clusters und schafft zudem Freiräume zur Balancierung zustandsbehafteten Verkehrs. Damit kann die Resequenzierung als wichtiger Bestandteil der Lastbalancierungsstrategie im FlexPath, aber auch für andere NP-Systeme angesehen werden.

Das Verfahren garantiert keine vollständige Resequenzierung, vielmehr kann es in Extremfällen zu einzelnen *out-of-order*-Paketen kommen. *Packet Reordering* an sich führt jedoch nicht zu Fehlern, da Protokolle wie TCP Mechanismen zum Erkennen und Beheben von *Packet Reordering* besitzen. Allerdings kann ein zu hohes Maß an *Packet Reordering* sehr schnell zu einer Degradierung der Netzgeschwindigkeit und -qualität führen, wohingegen vereinzelte *out-of-order*-Pakete keine nennenswerten negativen Auswirkungen haben. Hierbei gilt es abzuwägen, welcher Anteil an *Packet Reordering* toleriert wird bzw. welcher Aufwand zur Vermeidung betrieben wird. Das vorgestellte Verfahren versucht hierbei einen sinnvollen Kompromiss zu erreichen. Der mit Abstand größte Anteil der Pakete wurde mit der gewählten Dimensionierung erfolgreich resequenziert. Lediglich einzelne Pakete, für deren Resequenzierung ein überdurchschnittlicher Aufwand betrieben werden müsste, wird als *out-of-order* ausgesendet. Trotzdem wurden bei allen Simulationen für *Spraying*-Verkehr *in-order*-Raten von 99,99997% und mehr erreicht.

Im Gegensatz zum Verfahren nach Wu [61] unterstützt die vorgestellte Hardware-Resequenzierung Paketverluste innerhalb des Systems. Verluste werden dabei über einen Timeout-Mechanismus erkannt. Zudem wird durch den Einsatz eines Hashes anstatt eines CAMs die Skalierbarkeit deutlich erhöht. Bei Wu kommt es zudem zu temporären Blockierungen insbesondere bei steigenden *Packet Reordering*-Raten. Dadurch, dass hier – im Gegensatz zu Wu – nur die *out-of-order*-Pakete verwaltet werden, kann eine we-

sentlich ressourcenschonendere Implementierung erreicht werden. Zudem können Systeme mit wesentlich mehr Paketen im System verwaltet werden. Bei Wu kommt als Einschränkung zusätzlich zum tragen, dass als ID nur ein (nicht näher spezifizierter) 32 Bit-Wert eingesetzt wird, der damit ebenfalls nicht das IP 5-Tupel abdeckt. Die Untersuchungen bei Wu beinhalten damit ebenfalls Kollisionen – wenngleich auch in geringerer Anzahl.

Die Bestimmung der Timeout-Zeit führt insbesondere bei Verkehr mit besonders hohem Prozessierungsjitter wie bei IPsec (ca. 400 μs bis 3 ms) zu Problemen. Andererseits werden zustandsbehaftete Pakete generell mit einer dedizierte Lastbalancierungstrategie verteilt (siehe Kapitel 5.2.2). Damit reduziert sich die Gefahr von *Packet Reordering* auf die Umbalancierungen. Nachdem der Anteil der *out-of-order*-Pakete damit vergleichsweise gering ist, gilt es abzuwägen, inwieweit der Aufwand der Resequenzierung für diesen Pakettyp überhaupt betrieben werden soll. Je nach Anforderungen kann es ausreichend sein, die Resequenzierung auf die zustandslosen Pakete zu beschränken.

Am Ende des Kapitels wurde ein adaptives Verfahren vorgestellt, das die manuelle Bestimmung des Prozessierungsjitters, respektive einer sinnvollen Timeout-Zeit unnötig macht. Es passt die Timeout-Zeit auf Basis der Flow-ID vielmehr automatisch den aktuellen Umständen an. Erste Ergebnisse die im Rahmen von Simulationen erzielt wurden waren vielversprechend, lassen aber Raum für Optimierungen.

7 FlexPath-Demonstrator

In Kapitel 4 wurde das grundlegende FlexPath-Konzept vorgestellt. Außerdem wurden in den Folgekapiteln ausführlich die Lastbalancierungsstrategie im FlexPath, sowie die Hardware-Resequenzierung diskutiert. Diese Funktionen wurden anhand von Simulationen mit dem im Abschnitt 5.4.1 vorgestellten funktionalen SystemC-Simulationsmodell evaluiert.

Im nächsten Schritt wird nun eine prototypische Implementierung des FlexPath-Systems vorgestellt. Als Demonstrationsplattform dient hierbei ein Xilinx Virtex-4 FX60 FPGA. Mit dem FPGA-Demonstrator sollen insbesondere folgende Ziele erreicht werden:

- Der Demonstrator dient als **Proof of Concept** (vgl. auch [84]). Es soll also gezeigt werden, dass die in dieser Arbeit, sowie in [3] und [63] vorgestellten Verfahren und Ideen und damit die beschriebenen Vorteile tatsächlich mit vertretbarem Aufwand implementierbar sind. Hierbei können Schwierigkeiten, die bei der Implementierung festgestellt werden, ihrerseits wieder Rückwirkungen auf das ursprüngliche Konzept haben.
- Mit dem Demonstrator kann der **Simulator** verbessert werden. Anhand von Messungen und Analysen können wichtige Daten gewonnen werden, mit denen das Simulationsmodell verfeinert und präzisiert werden kann (siehe z.B. Prozessierungsjitter der CPUs in Abbildung 6.2).
- Das Simulationsmodell deckt nicht alle Details hinreichend ab. Mit Hilfe des Demonstrators können **Messungen** durchgeführt werden, die die zuvor gemachten Aussagen unter Einbeziehung aller Randbedingungen stützen.

Es sei hier zu Beginn betont, dass der Demonstrator nicht darauf abzielt, bezüglich Leistungsfähigkeit mit kommerziellen NPs zu konkurrieren. Dies ist aufgrund der FPGA-Implementierung mit einer wesentlich geringeren Taktfrequenz als z.B. bei einer ASIC-Lösung auch nicht zu erwarten. Vielmehr steht die relative Leistungssteigerung im Vordergrund, welche wiederum Rückschlüsse auf zu erwartende Verbesserungen im kommerziellen Einsatz zulässt. Die vorgestellten Konzepte können hierbei ausnahmslos auf ASIC-Technologie übertragen werden.

7.1 Demonstrator-Plattform: Xilinx Development Board ML410

Als Demonstrations-Plattform wurde das *ML410 Development Board* [85] von Xilinx genutzt, das alle wesentlichen Hardware-Komponenten zum Aufbau des Demonstrators

bereit stellt. Die wichtigsten Merkmale sind:

- Xilinx Virtex-4 FX60 FPGA mit u.a.
  - 56.880 Logikzellen / 25.280 Virtex-4 Slices
  - 232 Block-SRAM mit jeweils 18 KBit
  - zwei integrierten IBM PowerPC 405 RISC-CPUs
  - vier integrierten Ethernet-MACs
- 256 MB DDR2 DIMM
- 64 MB DDR *Component Memory*
- Onboard Dual 10/100/1000 Ethernet-PHYs
- Zwei *Personality Module Interfaces* (PMI) zum Anschluss zweier Erweiterungskarten

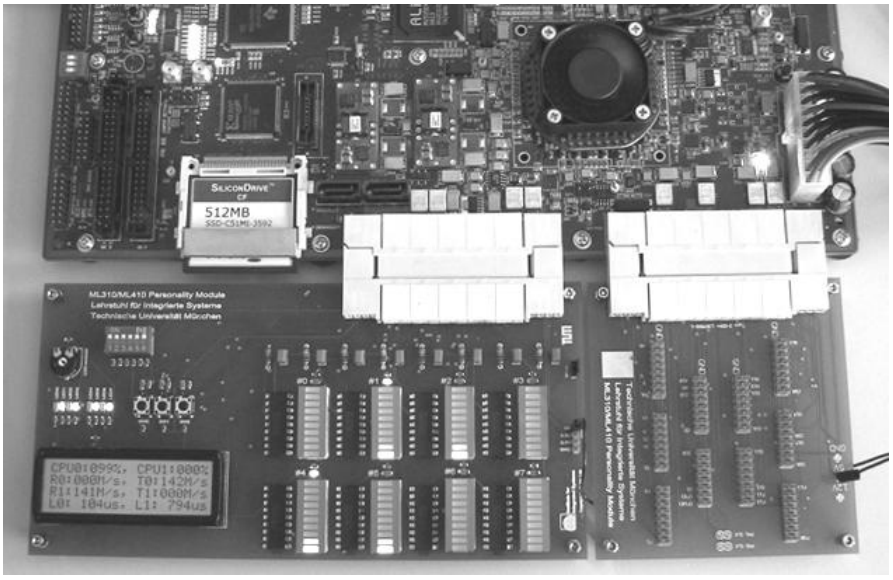

Abbildung 7.1: Zusatzboards am *Xilinx ML410 Development Board*

Im Rahmen dieser Arbeit wurden ferner zwei Erweiterungskarten entwickelt und gefertigt, die über die PMI-Schnittstelle direkt mit dem FPGA verbunden sind (siehe Abbildung 7.1). Die im Bild rechts dargestellte Erweiterungskarte dient zur Verbindung von FPGA-Signalen mit einem externen *Logic Analyzer*, was insbesondere zur Fehlersuche an der Hardware hilfreich ist. Ein weiteres Board (links) stellt eine Reihe von Ein- und Ausgängen zur Verfügung. Es dient zudem mit einer Reihe von LEDs, Bargraphen und einem LCD-Display zur Anzeige diverser Statusmeldungen. Mit den angebrachten Kippschaltern und Tastern lassen sich von extern diverse Einstellungen am System ändern.

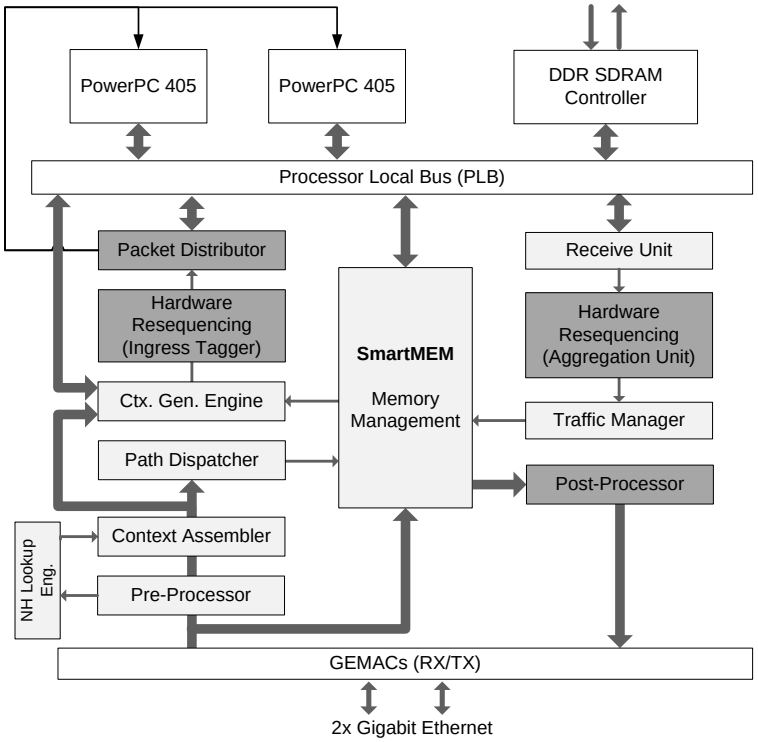

Abbildung 7.2: Blockschaltbild des FlexPath-Systems im Virtex-4 FPGA.

7.2 Implementierung des FlexPath-NPs

7.2.1 Implementierung der Hardware

Die Implementierung des FlexPath-Demonstrators auf dem FPGA ist als Blockschaltbild in Abbildung 7.2 dargestellt. Man erkennt im Grundsatz die Struktur aus Abbildung 4.3 wieder. Das System ist zur Reduzierung von Komplexität und Designaufwand – soweit möglich – aus Standardkomponenten aufgebaut. Als zentrales Kommunikationsmedium dient ein *Processor Local Bus* (PLB). Der PLB besitzt einen separaten Lese- und Schreibbus und hat bei einer Frequenz von 100 MHz und 64 Bit-Datenwortbreite einen maximalen theoretischen Durchsatz von jeweils 6,4 GBit/s. Der angeschlossene DDR SDRAM enthält die Instruktionen der CPUs und dient ebenso als Speicherort für Paket- und Kontextdaten. Als CPU-Cluster dienen die beiden auf dem FPGA integrierten PowerPC-Cores. Die CPU-Frequenz liegt bei 200 MHz. Die Demonstrations-Plattform besitzt damit weitaus weniger Rechenressourcen als z.B. ein Cavium Octeon II (siehe Abschnitt 3.1.1) mit bis zu 32 RISC-Cores und max. 1,5 GHz Taktfrequenz.

Ankommende Pakete werden von der **SmartMem Memory Management** Einheit in maximal zwei Segmente aufgeteilt und direkt im SDRAM abgespeichert. Der *SmartMem* erstellt für das Paket einen Paket-Deskriptor, der als Referenz an die nachfolgenden

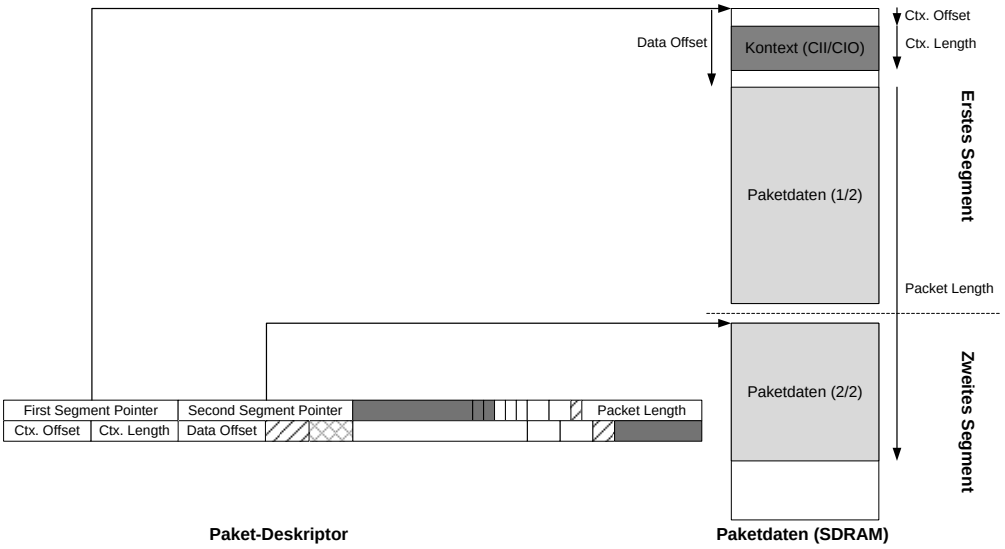

Abbildung 7.3: Speicherorganisation im FlexPath-Demonstrator mit Segmentierung und Paket-Deskriptor

Einheiten übergeben wird. Innerhalb des ersten Segments wird dabei ein Bereich reserviert, der im Folgenden für Ein- oder Ausgangskontext verwendet werden kann (siehe auch Abbildung 7.3).

Parallel zum Abspeichern analysiert der **Pre-Processor** das ankommende Paket. Es werden grundlegende Integritätschecks (IP-Checksumme, Längenüberprüfung) durchgeführt und protokollabhängig unterschiedliche Headerfelder (z.B. IP-Adressen, Ports, Protokoll, Priorität) extrahiert. Abhängig vom vorliegenden Protokoll startet der *Pre-Processor* einen *Lookup* in der **Next-Hop Lookup Engine**. Der **Context Assembler** fasst alle extrahierten Felder sowie das Lookup-Ergebnis zu einem standardisierten Roh-Kontext zusammen, welcher dem **Path Dispatcher** als Entscheidungsgrundlage dient. Anhand der Kontextfelder überprüft der *Path Dispatcher* seine rekonfigurierbare Regelbasis und trifft eine für das Paket passende Pfadentscheidung. In Abhängigkeit von der Pfadentscheidung generiert die **Context Generation Engine** einen passenden Paketkontext. Hierbei kann es sich entweder um einen Eingangs-Kontext (*Context Information Input*, CII) oder einen Ausgangs-Kontext (*Context Information Output*, CIO) handeln. Im Falle von CPU-Paketen wird ein CII angelegt. Hierfür werden alle relevanten Daten des Roh-Kontexts in den CII kopiert. Im Falle eines *AutoRoute*-Pakets wird dagegen ein CIO erstellt. Dieser enthält direkt die für den *Post-Processor* notwendigen Anweisungen zur ausgangsseitigen Manipulation des Pakets. Die Struktur des CIO kann dabei innerhalb der *Context Generation Engine* konfiguriert werden. Sie ist kombinierbar mit aktuellen Feldern des Roh-Kontexts (z.B. *Lookup*-Ergebnis). In beiden Fällen wird der Kontext im dafür vorgesehenen Platz innerhalb des ersten Datensegments im Speicher hinterlegt (siehe Abbildung 7.3). Der vom *SmartMem* erhaltene Paket-Deskriptor wird entsprechend ergänzt. Das Paket ist anschließend bereit zur Bearbeitung. Der Paket-

Einheit	Slices	FFs	LUTs	BRAMs
SmartMem	3.354	2.951	6.240	23
Pre-Processor	696	571	1.250	1
Context Assembler	487	250	766	-
Path Dispatcher	1.368	368	2.450	16
Ctx. Gen. Engine	759	961	1.201	5
Ingress Tagger	119	179	70	2
Packet Distributor	1.112	1.084	2.033	8
Aggregation Unit	1.234	1.036	2.246	11
Traffic Manager	441	533	689	4
Post-Processor	2.341	1.851	4.009	3
Gesamtsystem	**19.391**	**17.573**	**31.319**	**124**
FPGA Ressourcen	25.280	50.560	50.560	232
Belegt	76,4%	34,8%	61,9%	53,5%

Tabelle 7.1: Ressourcenverbrauch des FlexPath Demonstrators[1].

Deskriptor wird über den *Ingress Tagger* an den *Packet Distributor* geleitet und entsprechend der Pfadentscheidung in der korrekten Queue abgelegt. Die CPUs haben über den SDRAM Zugriff auf CII und Paketdaten und können bei Bedarf einen CIO für das Paket erstellen. Nach der Verarbeitung empfängt eine **Receive Unit** die Paket-Deskriptoren am Ausgangsdatenpfad und leitet sie zur Resequenzierung an die **Aggregation Unit** weiter. Ein **Traffic Manager** schlüsselt den Verkehr pro Port in zwei Prioritäten auf und stoppt im Falle einer Überlastung temporär den überlasteten Port. Ausgangsfertige Pakete werden durch den *SmartMem* zusammen mit einem (evtl. vorhandenen) CIO aus dem SDRAM ausgelesen und dem **Post-Processor** weitergeleitet. Im Falle eines nicht vorhandenen CIOs erzeugt der *SmartMem* automatisch für jedes Paket einen CIO bestehend aus einem NOP (vgl. Abschnitt 4.3.2). Nach der Datenmanipulation verlassen die Pakete das System über eine der beiden Gigabit-Ethernet-Schnittstellen.

Der Ressourcenverbrauch des Systems ist in Tabelle 7.1 dargestellt. Das Gesamtsystem hat einen Verbrauch von 19.391 Slices, entsprechend einem Anteil von 76,4% der zur Verfügung stehenden Ressourcen des Virtex-4 FPGAs. Zusammenfassend sind zudem die Werte der wichtigsten Systemkomponenten gegeben. Die Implementierung erreicht eine Taktfrequenz von 100 MHz.

7.2.2 Implementierung des Network Processing Software Stacks

Neben der Hardware-Implementierung musste für das FlexPath-System ein passender Software-Stack geschrieben werden. Um den Entwicklungsaufwand so gering wie möglich

[1]Die Zahlen basieren auf den Synthese-Reports von Xilinx EDK. Die Werte können aufgrund von Optimierungen und teilweise unterschiedlichen Konfigurationen stellenweise geringfügig von den Ergebnissen in den vorherigen Kapiteln abweichen.

zu halten, handelt es sich dabei um einen Stack, der viele Grundfunktionen des *Lightweigth IP*-Stacks [65] nutzt. Der Stack wurde für FlexPath optimiert und um IPsec-Funktionalität erweitert (siehe [86], [21]). Der Stack ist komplett in C implementiert.

Er unterscheidet sich insbesondere in folgenden Punkten von der ursprünglichen *Lightweight IP*-Version:

- Trennung von *Control* und *Data Plane*-Funktionalität.
- Unterstützung der FlexPath-spezifischen Speicherstruktur (Integration des *Smart-Mem Buffer Managers*, inkl. Segmentierung).
- Unterstützung der FlexPath-spezifischen Zusatzfunktionen (CII, CIO), u.a. Verwendung des im CII enthaltenen *Lookup*-Ergebnisses.
- Erweiterung um IPsec-Ver- und Entschlüsselung (beschränkt auf manuelle Konfiguration).

Nachdem der Demonstrator auf zwei CPUs beschränkt ist, wird im vorliegenden System keine reine *Control Plane*-CPU initialisiert. Während eine CPU nur *Data Plane*-Funktionen ausführt, erhält die zweite CPU zusätzlich *Control Plane*-Funktionalität.

Der Code wurde stellenweise auf Geschwindigkeit optimiert, ohne das komplette Optimierungspotential auszuschöpfen. Neben der schnelleren Entwicklungszeit stand auch die einfache Wartbarkeit im Vordergrund. Es ist daher davon auszugehen, dass durch weitere Softwareoptimierungen und/oder Verwendung von Assembler-Code noch nennenswerte Verbesserungen in Hinblick auf Code-Größe und Verarbeitungsgeschwindigkeit erreichbar sind.

7.3 Messungen am FlexPath-NP-Demonstrator

Mit Hilfe des Demonstrators ist es nun möglich eine Reihe von Messungen durchzuführen, die die Vorteile und Eigenheiten des FlexPath-Konzepts verdeutlichen sollen. Dabei können mit dem vorhandenen, auf zwei CPUs eingeschränkten System, nicht im gleichen Umfang Untersuchungen durchgeführt werden, wie mit Hilfe des Simulators. Letztlich handelt es sich hierbei um kein echtes „Multi“core-System. Dadurch sind auch Aussagen bezüglich der Lastbalancierung nur im eingeschränkten Rahmen möglich. Hinzu kommen Probleme bei der Stimulierung. Während beim Simulator „echter“ Internetverkehr mit Hilfe der *pcap*-Dateien verwendet werden kann, ist dies beim Demonstrator aufgrund der fehlenden Möglichkeit zum echtzeitfähigen Abspielen der Dateien nicht möglich. Daher finden die Untersuchungen mit künstlichem Verkehr statt. Dennoch haben die Messungen einen entscheidenden Vorteil. Während es sich beim Simulationsmodell um ein abstrahiertes Modell handelt, dessen Aussagekraft nur so gut wie das Modell ist, handelt es sich hier um reale Hardware. Damit beinhalten die Messungen **alle** Einflussgrößen und Schwankungen im System.

Die hier gezeigten Messungen decken bewusst nicht das ganze Spektrum um FlexPath erschöpfend ab und liefern teilweise nur erste Hinweise. So werden Themen wie *AutoRoute* – eines der Hauptgründe für die Implementierung des *Post-Processors* – in-

tensiv in [3] diskutiert. Dort finden sich auch eine Reihe von weiteren Messungen, die das Gesamtsystemverhalten von FlexPath behandeln.

Gemäß den Schwerpunkten dieser Arbeit beschränken sich die Messungen auf folgende Punkte:

- Es wird untersucht, inwieweit die **Verwendung des Post-Processors** die Prozessierung von IP-Forwarding beschleunigen kann. Dabei wird zwischen der Beschleunigung der CPU-Prozessierung und *AutoRoute* unterschieden.
- Der *Packet Distributor* erlaubt die Verwendung unterschiedlicher Queues und damit **unterschiedlicher Einstiegspunkte** in die Software. Damit kann direkt eine jeweils optimierte Software pro Queue verwendet werden.
- Die Lastbalancierung für zustandslosen Verkehr mit **Spraying** wird bezüglich Datendurchsatz und Verlustraten untersucht und mit dedizierten Verfahren verglichen.
- Schließlich wird der Umfang von **Packet Reordering** beim Einsatz von *Spraying* gemessen sowie die **Resequenzierung** untersucht.

7.3.1 Versuchsaufbau

Die Messungen am FlexPath-NP werden mit einem Netzwerktestgerät von Spirent (SPT-2000A) durchgeführt. Dieses ist ausgestattet mit einer Karte mit vier Gigabit-Ethernet-Anschlüssen. Die beiden Ethernet-Anschlüsse des FlexPath-Demonstrators werden direkt mit je einem Anschluss des Spirent Testgeräts verbunden (siehe Abbildung 7.4). Mithilfe des Testgeräts können unterschiedliche Pakete erzeugt und gleichzeitig analysiert werden. Das Testgerät versieht jedes Paket innerhalb der Payload mit einer eindeutigen ID, so dass sämtliche Pakete auch nach der Verarbeitung (einschließlich Modifikation der Header) identifizierbar bleiben.

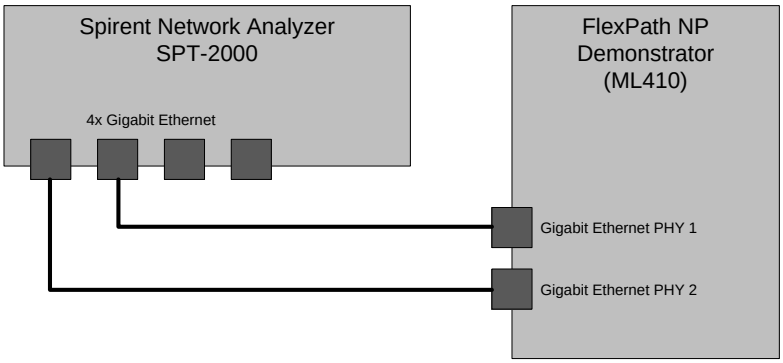

Abbildung 7.4: Versuchsaufbau zur Messung am FlexPath-NP Demonstrator

Mit Hilfe des Netzwerktesters können unterschiedliche Messungen durchgeführt werden. Insbesondere erlaubt der Tester auf Port- oder Flow-Ebene die genaue Bestimmung folgender, relevanter Messgrößen:

- **Paketlatenz**, jeweils gemessen von PHY zu PHY.
- Bestimmung der **Verlustraten** pro Flow.
- Messung der Anzahl der **out-of-order-Pakete**.

Ferner sind mit Hilfe des Netwerk-Testers automatisierte Tests möglich. Hierbei ist insbesondere die Durchsatzmessung nach RFC 2544 [87] interessant. Damit kann automatisch für unterschiedliche Paketgrößen der maximale Durchsatz durch das System bestimmt werden (also der Durchsatz bei dem gerade noch kein Paketverlust auftritt).

Der maximale Datenverkehr ist durch die beiden Ethernet-Schnittstellen auf 2 GBit/s beschränkt.

7.3.2 Optimierung am Packet Distributor

Zu Beginn der Messungen konnte eine Optimierung im *Packet Distributor* erreicht werden. Sie beruht auf der Beobachtung, dass je nach verwendeter Queue im *Packet Distributor* unterschiedliche Durchsatzraten gemessen wurden. In Abbildung 7.5 sind die durchschnittlichen Systemlatenzen bei Verwendung unterschiedlicher Queues dargestellt. Es wurden hierzu 64 Byte große Pakete und *Spraying* verwendet. Bei Queue #0 wurde mit 9,81 μs die geringste Latenz gemessen, bei Queue #15 mit 10,33 μs dagegen die Längste. Der Anstieg dazwischen ist nahezu linear. Entsprechend der Latenz ergibt sich bei Verwendung von Queue #0 ein maximaler Datendurchsatz von 202 MBit/s, bei Queue #15 dagegen von nur 187 MBit/s.

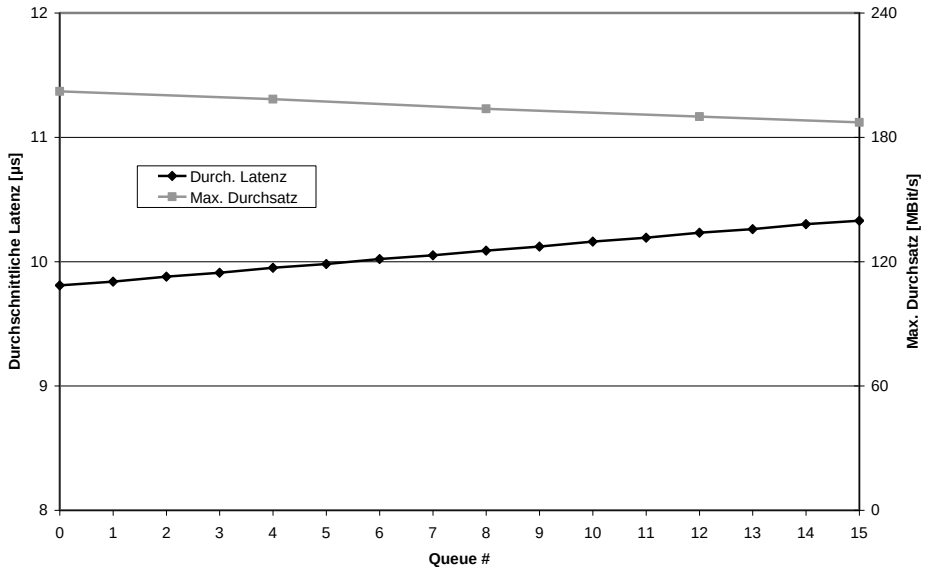

Abbildung 7.5: Durchschnittliche Systemlatenz bei 10 MBit/s Forwarding-Verkehr und maximaler Datendurchsatz jeweils bei 64 Byte großen Paketen mit nicht optimiertem *Interrupt Controller*.

Die Ursache für dieses zunächst seltsame Verhalten findet sich in der Implementierung des *Packet Distributors / Interrupt Controllers* und der Auswertung des *Interrupt Pending Registers* im Treiber (siehe Abschnitt 5.3.6 bzw. Tabelle 5.2). Das *Interrupt Pending Register* enthält die aktiven, zu bearbeitenden Interrupts. Dabei repräsentiert jedes Bit des 32 Bit großen Registers einen Interrupt-Eingang. Der Treiber muss deshalb den höchstprioren, aktiven Interrupt identifizieren und die zugehörige *Service Routine* starten. Die Bestimmung des auszuführenden Interrupts geschieht entsprechend der originalen Implementierung des zugrundeliegenden Xilinx *Single Processor Interrupt Controllers* als lineare Suche mit Hilfe des folgenden Code-Fragments:

```
for (IntrNumber = 0; IntrNumber < 32; IntrNumber++) {
   if (PendingInts & 1) {
      TablePtr = &(CfgPtr->HandlerTable[IntrNumber]);
      TablePtr->Handler(TablePtr->CallBackRef);

      if((CfgPtr->AckBeforeService & IntrMask) == 0) {
         XIntc_mSendAck((CfgPtr->BaseAddress+0x00008000), IntrMask);
      }
      break;
   }
   IntrMask <<= 1;
   PendingInts >>= 1;
}
```

Die Schleife überprüft jeweils das erste (höchstpriore) Bit. Sofern das Bit gesetzt ist, wird die zugehörige *Service Routine* gestartet und – sofern der Interrupt *levelsensitiv* ist – der Interrupt mit einem *Acknowledgement* zurückgesetzt. Sofern ein aktiver Interrupt gefunden wurde, wird die Routine mittels `break` verlassen. Der Registerinhalt wird bei jedem Schleifendurchlauf um eine Stelle verschoben um somit im nächsten Durchlauf die nächste Stelle zu überprüfen. Damit jedoch wird die Schleife umso häufiger durchlaufen, je höher die Interrupt-/Queue-Nummer ist.

Dieses Vorgehen ist bei einem Einprozessorsystem sinnvoll, da dann in einen Durchlauf ohne Nutzung des `break`-Befehls alle aktuell aktiven Interrupts identifiziert und bearbeitet werden können. Im vorliegenden Mehrprozessor-System darf jeweils aber nur der höchstpriore (für diese CPU vorgesehene) Interrupt bearbeitet werden. Nur für diesen liegt schließlich auch ein Paket-Deskriptor bereit. Die Bearbeitung lässt sich also dadurch vereinfachen, dass der *Packet Distributor* anstatt dem gesamten Inhalt des *Interrupt Pending Registers*, nun direkt die binär codierte Nummer des höchstprioren Interrupts übergibt. Damit kann direkt die korrekte *Service Routine* angesprungen werden und die lineare Suche in Software entfällt. Somit wird zum einen die Bearbeitungsdauer unabhängig von der Queue-Nummer, es lässt sich zum anderen aber auch eine nennenswerte Steigerung des Durchsatzes erreichen. Die durchschnittliche Latenz sinkt auf konstant 9,33 μs, der Durchsatz steigt auf rund 215 MBit/s (entsprechend einer Steigerung um 6,3% - 14,9%).

Der optimierte *Packet Distributor* hat dabei einen Ressourcenverbrauch von 1.127 Slices und damit 15 Slices mehr als die ursprüngliche in Abschnitt 5.3.6 vorgestellte Implementierung. Im Folgenden wird der optimierte *Packet Distributor* verwendet.

7.3.3 IP-Forwarding unter Verwendung des Post-Processors

Die erste Messreihe untersucht die Leistungssteigerung bei Verwendung des *Post-Processors*. Dazu werden Pakete auf die zwei verfügbaren PowerPCs mittels *Spraying* verteilt und bearbeitet. Hierbei wird nur IP-Forwarding betrachtet. Mit Hilfe des RFC 2544 Tests wird jeweils der maximale Durchsatz für unterschiedliche Paketgrößen gemessen. Dabei werden drei unterschiedliche Szenarios getestet:

- **Ohne Hardware-Unterstützung:** Die Bearbeitung erfolgt komplett in Software, die Ergebnisse der Hardware-Vorverarbeitung werden nicht genutzt. Ebenso werden sämtliche Paketmodifikation in Software vorgenommen. Dieses Szenario kann somit als Referenz ohne FlexPath-Komponenten betrachtet werden.
- **Nur mit CII:** Der Prozessor verwendet die Ergebnisse der Hardware-Vorverarbeitung (*Pre-Processor*). Dazu wird für jedes Paket der CII ausgelesen und ausgewertet. Modifikationen werden dennoch direkt am Paketheader vorgenommen.
- **Mit CII/CIO:** Neben den CII wird nun zusätzlich der *Post-Processor* zur Paketmodifikation verwendet. Dazu wird für jedes Paket ein passender CIO angelegt.

Es wird zunächst die Paketrate für die reine CPU-Bearbeitung ohne CII oder CIO in Abbildung 7.6 betrachtet. Für 64 Byte große Pakete liegt die Paketrate mit beiden CPUs bei ca. 160.000 Paketen pro Sekunde (160 kpps). Die Rate bleibt zunächst auch für größere Pakete konstant. Dies ist nicht weiter verwunderlich, da beim Forwarding schließlich nur der Header prozessiert wird und die Länge des Pakets für die Verarbeitung unerheblich ist. Mit steigender Paketgröße und konstanter Paketrate steigt demnach die maximale Datenrate im System linear an (siehe Abbildung 7.7). Man erkennt aber auch, dass die Paketrate für größere Paketgrößen langsam auf 140 kpps bei ca. 1.300 Byte großen Paketen und danach schlagartig auf 111 kpps bei 1.518 Byte Paketgröße abnimmt. Die Datenrate scheint hierbei an eine obere Grenze zu stoßen. Tatsächlich zeigt sich, dass die zentrale Anbindung von Speicher und Bus überlastet ist. Mit steigender Paketgröße müssen immer größeren Datenmengen in den Speicher geschrieben und wieder daraus gelesen werden. Der Speicher wird somit zur Engstelle, wohingegen das CPU-Cluster nicht mehr der limitierende Faktor ist.

Unter Verwendung der *Pre-Processor*-Ergebnisse im CII zeigt sich insbesondere bei kleinen Paketen eine deutliche Steigerung der Paketrate von zuvor 160,5 kpps auf nun 286,5 kpps. Der deutliche Anstieg um 78,5% lässt sich mit der Auslagerung der sehr aufwändigen Überprüfung der IP-Checksumme, aber auch des *Address Lookups*, erklären. Der Anstieg fällt dabei sogar um einiges deutlicher aus als in Tabelle 4.1 (S. 66) prognostiziert. Der Grund liegt in der im Vergleich zum *Lightweight IP*-Stack optimierten Implementierung, wodurch die Berechnung der Checksumme relativ einen größeren Anteil an der Gesamtprozessierung besitzt. Man erkennt aber auch, dass durch die von

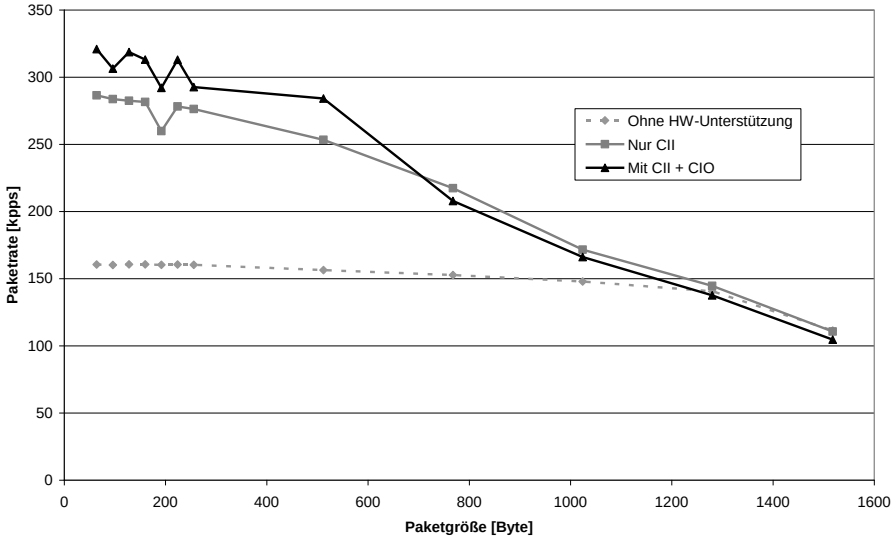

Abbildung 7.6: CPU-Paketdurchsatz für unterschiedliche Paketgrößen mit und ohne Hardware-Unterstützung.

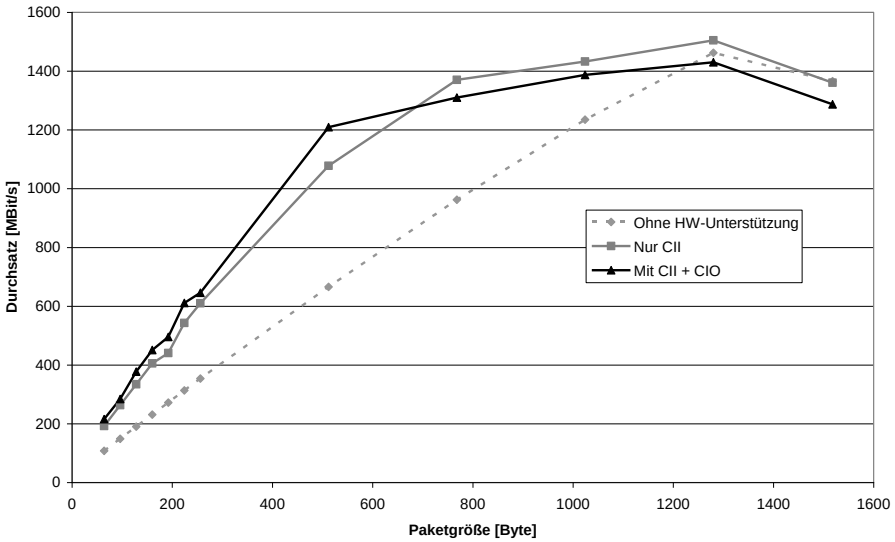

Abbildung 7.7: CPU-Datendurchsatz für unterschiedliche Paketgrößen mit und ohne Hardware-Unterstützung.

Beginn an höhere Datenrate die Limitierung des Speichers früher erreicht wird und sich die Paketrate für große Pakete (>1.200 Byte) der ersten Kurve annähert.

Bei Verwendung des *Post-Processors* würde man zunächst eigentlich keine weitere Leistungssteigerung erwarten. Zwar müssen die MAC-Adressen im Header ausgetauscht werden, ob dies nun aber direkt im Header oder über Anlegen eines entsprechenden CIOs im Speicher geschieht, macht aus Prozessorsicht zunächst keinen nennenswerten Unterschied. Selbst die Aktualisierung der IP-Checksumme ist vom Aufwand her überschaubar. Da sich lediglich das TTL-Feld im Header dekrementell ändert, kann die Checksumme sehr einfach inkrementell aus der alten Checksumme berechnet werden. Andererseits muss bei Verwendung des *Post-Processors* hierzu auch erst der passende CIO-Befehl erstellt und in den Speicher geschrieben werden. Dennoch zeigt sich unter Verwendung des CIOs eine nochmalige Steigerung um 12% auf nun 320,9 kpps bei 64 Byte großen Paketen. Der Effekt lässt sich dadurch erklären, dass der Prozessor nun nicht mehr mit zwei unterschiedlichen Datenstrukturen arbeiten muss. Bei der reinen Softwareprozessierung wird nur der Header in den Cache geladen, bearbeitet und abgespeichert. Bei Verwendung des CIIs muss dieser zusätzlich aus dem Speicher gelesen werden. Werden CII und CIO verwendet entfällt dagegen das Laden des Headers. Der CIO wird im gleichen Speicherbereich abgelegt wie der CII und der Datentransfer zum Prozessor dadurch reduziert.

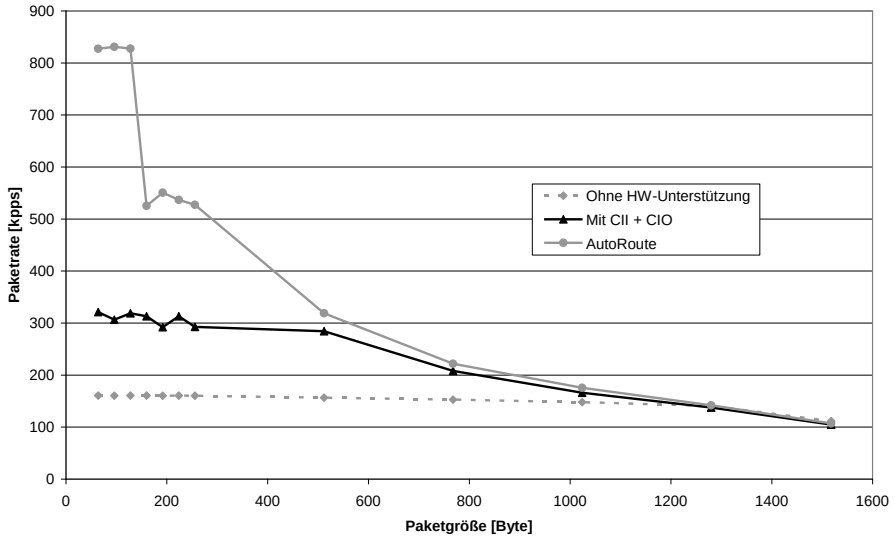

Abbildung 7.8: Paketdurchsatz für unterschiedliche Paketgrößen bei *AutoRoute* im Vergleich zur CPU-Verarbeitung.

Für steigende Paketgrößen verringert sich der Vorteil der Verwendung des *Post-Processors* – die Paketrate sinkt letztlich sogar unter die Rate der Softwarelösung bzw. bei Verwendung von nur CII. Der CIO erzeugt hierbei beim Auslesen durch den *SmartMem* eine zusätzliche Last im Speicher und den Bus – der Speicherdatentransfer pro Paket steigt also an. Als Folge wird das Speicherlimit eher erreicht, die Nutzdatenrate fällt

unter die beiden anderen Fällen.

Insgesamt kann durch die Messungen gezeigt werden, dass der *Post-Processor* die Prozessierung von IP-Paketen beschleunigt. Der Geschwindigkeitsvorteil liegt dabei bei ca. 12%, zumindest solange das CPU-Cluster der limitierende Faktor ist. Der Hauptnutzen des *Post-Processors* liegt jedoch in der Unterstützung von *AutoRoute*. In Abbildung 7.8 erkennt man insbesondere für kleine Pakete mit einer Paketrate von 830 kpps den Vorteil gegenüber der CPU-Prozessierung. Auch hier gilt, dass für ansteigende Paketgrößen und damit steigenden Datenraten sehr schnell die Speicherlimitierung erreicht und der theoretisch erreichbare Vorteil reduziert wird bzw. nicht mehr vorhanden ist.

7.3.4 Einfluss des Softwareeinstiegspunkts auf den Datendurchsatz

Bisher wurden unterschiedliche Pfade innerhalb des FlexPath-NPs in der Regel als physikalische Pfade definiert. Ein Pfad bestimmt danach also, ob ein Paket einer bestimmten CPU, einem Pool von CPUs, einem Hardware-Beschleuniger oder auch nur der Hardware zur Bearbeitung übergeben wird. Zusätzlich zur Bestimmung der Prozessierungseinheit kann es jedoch sinnvoll sein, den korrekten Softwareeinstiegspunkt für ein Paket zu wählen. Der Vorteil soll an einem einfachen Beispiel demonstriert werden: Ankommende Pakete werden hierbei einer CPU zur Bearbeitung übergeben. Es werden zwei verschiedene Szenarien unterschieden:

- **Szenario 1 (ohne Vorklassifizierung):** Pakete werden der CPU zur Bearbeitung übergeben. Die CPU erhält keinerlei Zusatzinformationen zum Paket (siehe Abbildung 7.9a).
- **Szenario 2 (mit Vorklassifizierung):** Es findet bereits eine Vorklassifizierung im *Path Dispatcher* statt. Das Ergebnis wird der CPU, z.B. anhand unterschiedlicher Eingangs-Queues, übergeben (siehe Abbildung 7.9b).

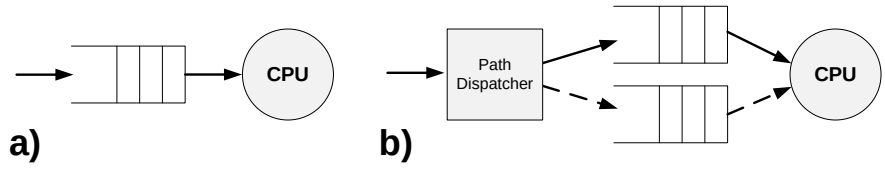

Abbildung 7.9: Paketübermittlung an eine CPU ohne (a) und mit (b) Vorklassifizierung.

Im konkreten Fall wird von einer Umgebung ausgegangen, die IPsec unterstützen soll. In diesem Fall muss die *Security Policy Database* überprüft werden (siehe Abschnitt 2.3.4). Die dort gespeicherten Einträge enthalten in Abhängigkeit vom IP 5-Tupel die Anweisung, ob ein Paket verschlüsselt werden soll, oder ob sich die Bearbeitung auf Forwarding beschränkt. Es ist dabei davon auszugehen, dass – je nach Einsatzumgebung – nur ein kleiner Teil der Pakete verschlüsselt werden muss. Die Überprüfung muss jedoch für alle Pakete von der CPU durchgeführt werden. Mit Hilfe des *Path Dispatchers* kann jedoch eine Vorklassifizierung stattfinden. Da die Adressen der zu verschlüsselnden Pakete bekannt sind, können die ankommenden Pakete (z.B. anhand des Subnetzes) aufgeteilt

werden in Pakete, die sicher nur IP-Forwarding benötigen und Pakete, die möglicherweise zu verschlüsseln sind. Eine endgültige Überprüfung kann dann in Software durchgeführt werden. Eine komplette Überprüfung in Hardware wäre zwar möglich, aber auch sehr aufwändig und wird daher nicht praktiziert. Beide Paketarten werden in unterschiedliche Queues verteilt. Für jede Queue kann direkt eine optimierte *Service Routine* in der CPU verwendet werden.

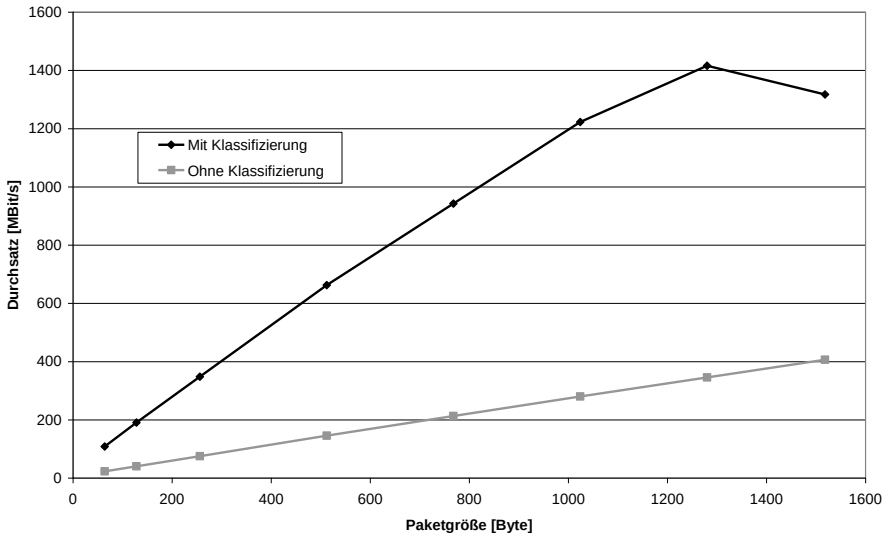

Abbildung 7.10: Einfluss des Software-Einstiegspunkt beim Forwarding auf den Datendurchsatz mit und ohne Vorklassifizierung in Hardware (RFC 2544, eine CPU).

In Abbildung 7.10 sind die gemessenen Durchsatzraten für unterschiedliche Paketgrößen und eine CPU bei Forwarding aufgezeichnet. Bei der Variante ohne Vorklassifizierung führt die CPU die Überprüfung der *Security Policy Database* für alle Pakete durch, bei der Variante mit Vorklassifizierung entfällt diese für Nicht-IPsec-Pakete. Die Durchsatzraten unterscheiden sich dabei deutlich. Während der Durchsatz für 64 Byte große Pakete ohne Überprüfung bei 108,1 MBit/s liegt, sind es mit Überprüfung lediglich 22,8 MBit/s. Es sei hier angemerkt, dass die verwendete IPsec-Implementierung nicht hochoptimiert ist. Der Unterschied der beiden Varianten kann durch eine entsprechende Optimierung sicherlich reduziert werden. Dennoch ist der Aufwand, der für die Überprüfung der IP-Adressen anfällt (auch in Abhängigkeit von der Anzahl der Einträge und dem verwendeten Suchalgorithmus), in jedem Fall signifikant. Es ist daher von einer nennenswerten Leistungssteigerung auszugehen, sofern ein Großteil der Pakete wegen der Vorklassifizierung ohne Überprüfung der *Security Policy Database* bearbeitet werden kann.

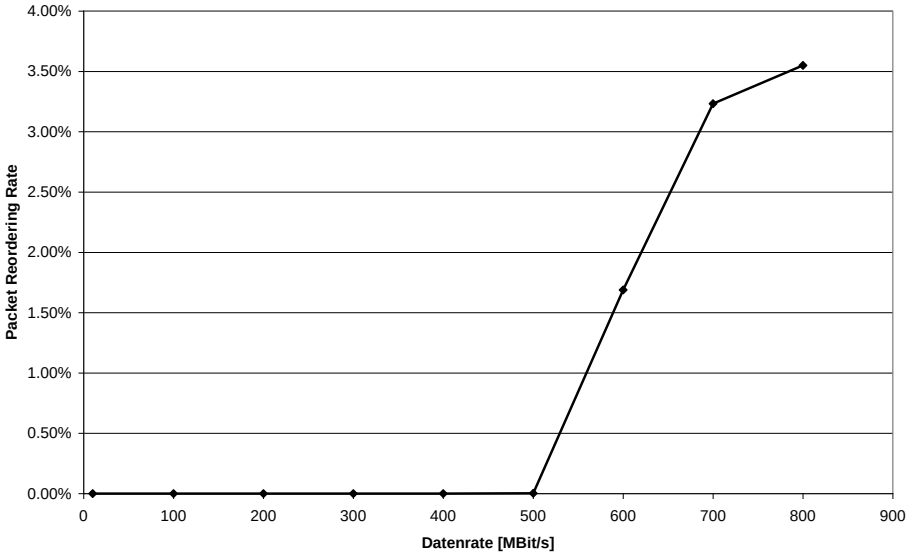

Abbildung 7.11: *Packet Reordering*-Raten mit einem Flow bei unterschiedlichen Eingangsdatenraten **ohne** Resequenzierung (IMIX).

7.3.5 Packet Reordering und Hardware-Resequenzierung bei Spraying

In diesem Abschnitt werden nun die Auswirkungen von *Packet Reordering* beim *Spraying* untersucht. Dazu werden wiederum beide CPUs genutzt, auf die die ankommenden Pakete per *Spraying* verteilt werden. Zunächst wird untersucht, welcher Anteil an *Packet Reordering* sich ohne Resequenzierung beim *Spraying* ergibt. Diese Ergebnisse werden im zweiten Schritt Messungen mit Resequenzierung gegenübergestellt. Alle Messungen wurden dabei über einen Zeitraum von jeweils 30 Sekunden mit unterschiedlichen Datenraten durchgeführt.

Die erste Messreihe startet mit einem Flow und variierender Eingangsdatenrate. Es wird eine „Simple IMIX“-Verteilung gewählt. Die IMIX-Verteilung ist in [88] definiert und gibt eine einfache Verteilung aus 64-, 594- und 1.518-Ethernet Paketen wieder, die versucht die realen Paketgrößenverhältnisse im Internet näherungsweise widerzuspiegeln. Man erkennt in Abbildung 7.11, wie für kleine Datenraten zunächst kein *Packet Reordering* festgestellt werden kann. Die Erklärung hierfür besteht darin, dass die beiden CPUs sich zunächst in Unterlast befinden. Ankommende Pakete können direkt von einer CPU bearbeitet werden. Der Abstand zwischen den ankommenden Paketen ist dabei groß genug, dass es zu keiner Überholung aufgrund des Prozessierungsjitter kommt. Erst ab ca. 400 MBit/s Datenrate lassen sich erste, vereinzelte *out-of-order*-Pakete messen. Die *Packet Reordering*-Rate bleibt aber noch nahe Null. Erst ab 600 MBit/s steigt die *Reordering* Rate sprunghaft an und scheint bei etwas mehr als 3,5% allmählich zu sättigen.

Es ist naheliegend, dass die Wahrscheinlichkeit von *Packet Reordering* bei steigender Anzahl an Flows bei konstanter Datenrate abnimmt. Der Anteil der Paketvertauschungen im System sollte bei an sich gleichen Rahmenbedingungen (Systemkonfiguration,

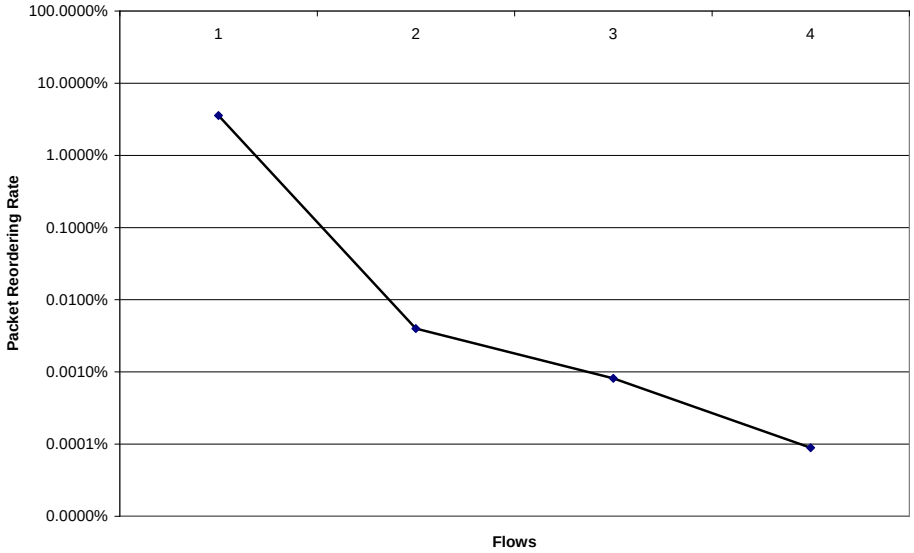

Abbildung 7.12: *Packet Reordering*-Raten bei *Spraying* und ansteigender Anzahl an Flows (IMIX, konstante Datenrate von 800 MBit/s) **ohne** Resequenzierung.

Eingangsdatenrate) einigermaßen konstant bleiben. Allerdings sinkt die Wahrscheinlichkeit bei mehreren Flows, dass eine Vertauschung zwei Pakete des gleichen Flows betrifft, was ja die Grundvoraussetzung für *Packet Reordering* ist. Anders ausgedrückt ist die Flow-Datenrate entscheidend für die *Packet Reordering*-Rate und nicht die aggregierte Datenrate aus allen Flows. In Abbildung 7.12 erkennt man, wie sehr die *Packet Reordering* Rate bei mehreren Flows sinkt. Es wurde eine konstante Eingangsdatenrate von 800 MBit/s gewählt. Diese wurde jedoch mit einer unterschiedlichen Anzahl an Flows generiert (die Flow-Datenrate sinkt also auf (800 MBit/s)/n, mit n gleich der Anzahl der Flows). Die *Reordering*-Rate sinkt bereits mit zwei Flows von 3,55% auf nurmehr 0,004%. Bei vier Flows liegt die Rate lediglich bei 0,0001% und ab fünf Flows sind gar keine *out-of-order*-Pakete mehr messbar.

Der starke Abfall ist dabei sicherlich auch der künstlichen Verkehrserzeugung geschuldet. Diese verteilt die Flows gleichmäßig über die Zeit. In einem realen Verkehr dürften zumindest einzelne Bursts eines Flows auch bei einer großen Anzahl an Flows noch *Packet Reordering* provozieren, wie dies in den Simulationen auch zu sehen war. Es zeigen sich hierbei aber auch die Limitierungen des Demonstrators, der lediglich zwei CPUs zur Verfügung hat. Bei mehr CPUs (und damit mehr Paketen, die parallel bearbeitet werden) würde die Wahrscheinlichkeit von *Packet Reordering* sicherlich wieder ansteigen (siehe hierzu auch die Simulationsergebnisse im Abschnitt 6.4.1). Die Messung mit Resequenzierung wurde aus diesem Grunde als *worst case*-Betrachtung mit nur einem Flow durchgeführt.

Die Messungen nach der Resequenzierung haben ergeben, dass alle Pakete wieder in die

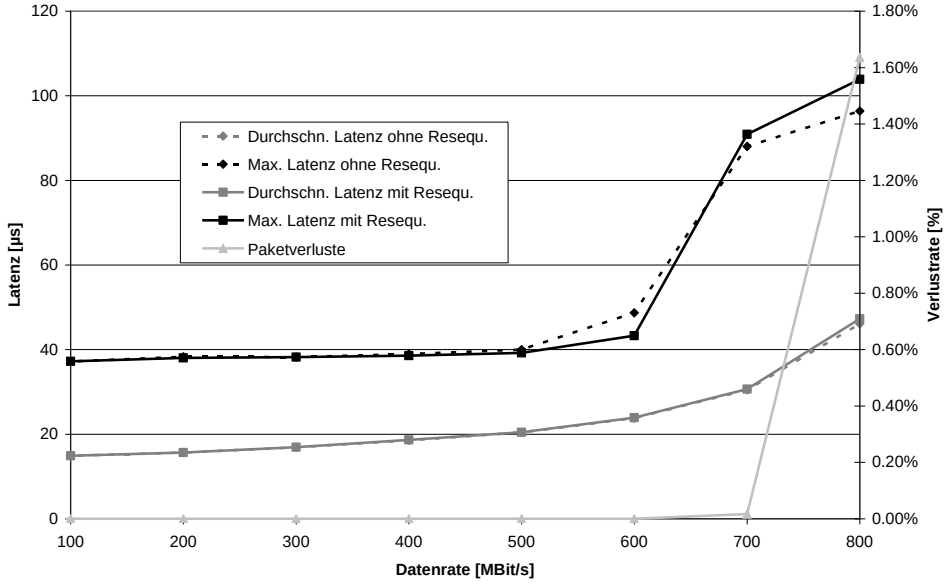

Abbildung 7.13: Durchschnittliche und maximale Paketlatenzen bei einem Flow bei unterschiedlichen Eingangsdatenraten vor und nach Resequenzierung (*Spraying*, IMIX).

korrekte Reihenfolge gebracht werden konnten. Interessant ist dabei, welche Auswirkungen die Resequenzierung auf die Paketlatenz besitzt. Es wurde sowohl die durchschnittliche Paketlatenz aller Pakete als auch die maximal erreichte Paketlatenz gemessen und mit den Werten der Messreihe ohne Resequenzierung verglichen (siehe Abbildung 7.13). Unter 500 MBit/s sind die beiden Messkurven sowohl für die durchschnittliche, als auch maximale Latenz praktisch identisch. Dies verwundert insofern nicht, als in diesem Bereich noch kaum *out-of-order*-Pakete auftreten. Die Latenzen steigen jedoch insgesamt etwas an, da mit steigender Datenrate der durchschnittliche Queue-Füllstand steigt. Eine erste nennenswerte Entwicklung gibt es erst bei 700 MBit/s. Die maximale Latenz steigt dabei von 88,0 μs auf immerhin 90,9 μs. Eine Rolle spielen hierbei die ersten im System verlorengegangenen Pakete (Verlustrate 0,017%), die die folgenden Pakete in die Warteschleife zwingen und auf den Timeout warten müssen. Die durchschnittliche Latenz wird dabei noch kaum verändert (30,45 μs ohne Resequenzierung, 30,64 μs mit Resquenzierung). Dies ändert sich bei 800 MBit/s Datenrate. Die Verlustrate steigt hier auf 1,64% an, was nun auch leichte Auswirkungen auf die durchschnittliche Latenz hat (46,06 μs ohne Resequenzierung, 47,27 μs mit Resequenzierung). Das System ist hierbei also insgesamt bereits in Überlast.

7.3.6 Packet Spraying vs. dedizierte Lastbalancierung

Aufbauend auf den Untersuchungen zur Lastbalancierung im Kapitel 5, werden in diesem Abschnitt Messungen an unterschiedlichen Lastbalancierungsverfahren durchgeführt. Ba-

sierend auf dem Schwerpunkt dieser Arbeit bezüglich der Verteilung von zustandslosem Verkehr interessieren hierbei v.a. die Eigenschaften von *Spraying*. Als Referenz wird das in Abschnitt 5.2.2 vorgestellte dedizierte Verfahren HLU verwendet. Bereits in den Simulationen konnte gezeigt werden, dass HLU bei geringem Implementierungsaufwand im Vergleich zu anderen dedizierten Verfahren gute Ergebnisse liefert (siehe Abschnitt 5.4.3). Als Verkehrsstimuli wird ausschließlich zustandsloser IP-Forwarding Verkehr verwendet.

Es werden die folgenden Konfigurationen untersucht:

- **Spraying:** Alle ankommenden Pakete werden in einer gemeinsamen Eingangsqueue im *Packet Distributor* verwaltet und auf die beiden CPUs mittels *Spraying* verteilt. Die Queuetiefe liegt dabei bei 16 Paketen.
- **HLU 8:** Jede CPU erhält eine eigene Eingangsqueue im *Packet Distributor*. Die Aufteilung in die Queues erfolgt anhand des HLU-Algorithmus. Das Regelintervall beträgt 50 ms. Die Queuetiefe ist auf 8 Pakete beschränkt. Der Speicherbedarf der beiden Queues entspricht daher in Summe dem Speicherbedarf beim *Spraying*.
- **HLU 16:** Die Queuetiefe pro CPU wird zu 16 gewählt und entspricht daher der Queuetiefe beim *Spraying*. Der Speicherbedarf der Queues verdoppelt sich daher gegenüber *Spraying*. Die restlichen Einstellungen entsprechen der Konfiguration von *HLU 8*.

Stimulierung / Verkehrsaufbau

Bei der Bewertung der Lastbalancierungsverfahren spielt der verwendete Verkehr eine entscheidende Rolle. Als ideal erweisen sich hierbei die bei den Simulationen verwendeten Verkehrsmitschnitte aus Abschnitt 5.4.2, die ein realistisches Verhalten bezüglich Anzahl an Flows, Paket-Bursts, zeitlichem Verhalten etc. widerspiegeln. Gerade diese zeitliche Dynamik ist für die Lastbalancierung und die damit verbundenen Umbalancierungen maßgeblich. Die Möglichkeiten zur Erstellung eines realistischen Zeitverhaltens sind bei den Messungen jedoch bei Verwendung des Spirent Netzwerk-Testgeräts eingeschränkt. Insbesondere die Anzahl der unabhängig voneinander definierbaren Flows mit unterschiedlichem zeitlichen Verhalten unterliegt starken Einschränkungen.

Auch innerhalb des Systems gibt es Einschränkungen. Für große Pakete wird mit steigender Paketrate die Speicherlimitierung vor Auslastung der CPUs erreicht (vgl. Abbildung 7.7). Um die Effektivität der Lastbalancierung beurteilen zu können, interessiert jedoch v.a. die Auslastung der CPUs. Um Nebeneffekte durch die Speicherlimitierung weitgehend auszuschließen, werden die folgenden Messungen deshalb ausschließlich mit kleinen Paketgrößen durchgeführt.

In Tabelle 7.2 sind die für die Messungen definierten Flows mit Burstlänge und Paketabstand innerhalb des Bursts angegeben. Es wurden insgesamt 16 Flowbündel definiert. Jedes Flowbündel besteht aus je zwei Flows, womit sich insgesamt 32 unterschiedliche Flows ergeben. Alle Flows unterscheiden sich untereinander mindestens in IP-Adresse und/oder TCP-Portnummer. Je die Hälfte der Flows erreicht das Testsy-

#	Flows	TX Port	RX Port	Paketlänge [Byte]	Burst-länge	IFG [μs]	Anteil [%]
0	2	1	2	78	6.000	40	2,54
1	2	1	2	78	5.000	40	3,39
2	2	1	2	78	4.000	40	4,24
3	2	1	2	78	3.000	40	5,08
4	2	1	2	78	2.000	40	2,20
5	2	1	2	78	1.000	40	3,14
6	2	1	2	78	1	–	13,72
7	2	1	2	78	1	–	13,73
8	2	2	1	90	6.000	40	2,54
9	2	2	1	90	5.000	40	3,39
10	2	2	1	90	4.000	40	4,14
11	2	2	1	90	3.000	40	5,35
12	2	2	1	90	2.000	40	4,06
13	2	2	1	90	1.000	40	5,00
14	2	2	1	90	1	–	13,74
15	2	2	1	90	1	–	13,75

Tabelle 7.2: Anteil der verschiedenen Flowbündel (charakterisiert durch Burstlänge und Paketabstand innerhalb der Bursts (= *Inter Frame Gap*, IFG)) an der Gesamtdatenrate.

stem über den ersten bzw. zweiten Ethernet-Port. Als Paketgröße wurde die für den Netzwerktester minimale Paketgröße von 78 Byte (enthält neben den TCP/IP-Header eine eindeutige Identifikationsnummer innerhalb der Payload) bzw. als leichte Variation 90 Byte gewählt. Etwa 55% der Pakete (bezogen auf die Datenrate) werden mit gleichbleibendem Paketabstand versendet und besitzen keinen bursthaften Charakter (Burstlänge eins). Die restlichen 45% der Datenrate teilen sich auf zwölf Flowbündel mit Bursts auf. Der Abstand zwischen zwei Paketen innerhalb des Bursts liegt für alle diese Flowbündel bei konstant 40μs, die Länge der Bursts bleibt für alle Datenraten jeweils konstant, und variiert je nach Flowbündel zwischen 1.000 und 6.000 Paketen. Eine höhere Datenrate verändert damit nicht den Charakter eines einzelnen Bursts, sondern verkürzt demnach nur die Zeit zwischen zwei aufeinanderfolgenden Bursts. Beide Flows innerhalb eines Flowbündels wechseln sich mit jedem Burst (bzw. bei Burstlänge eins: mit jedem Paket) ab.

Für die Messungen wird die durchschnittliche Datenrate zwischen 10 und 275 MBit/s variiert. Nachdem sich dabei der Charakter eines einzelnen Bursts nicht verändert, wechseln sich Zeiten hoher Aktivität mit Zeiten geringer Aktivität ab. Durch die Überlagerung der einzelnen Flowbündel ergeben sich zufällige Lastspitzen, die teilweise kurzfristig weit über den durchschnittlichen Datenraten liegen. Die Messdauer beträgt für jede Datenrate und jede Balancierungsart jeweils 60 Sekunden. Die Hardware-Resequenzierung ist

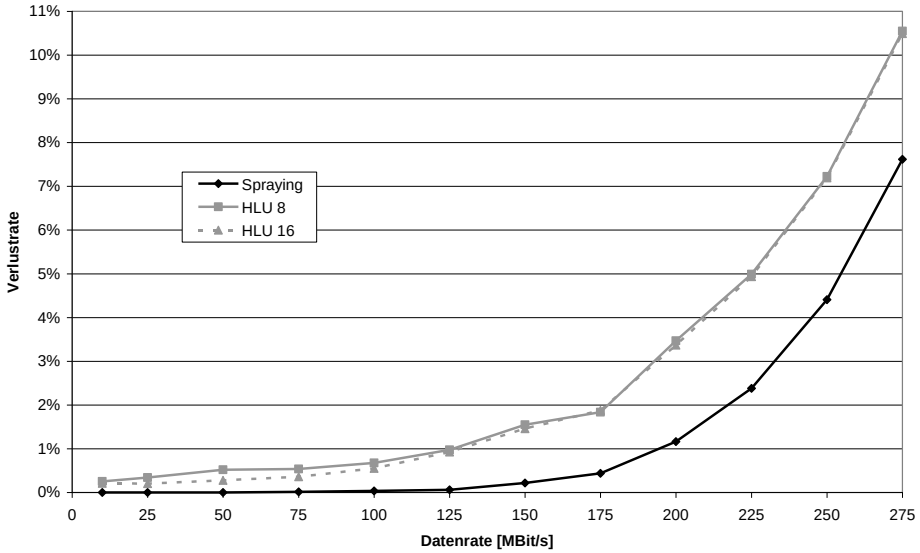

Abbildung 7.14: Paketverlustraten bei unterschiedlichen Eingangsdatenraten mit bursthaftem Verkehr für *Spraying* und HLU.

aktiviert, die Timeout-Zeit liegt bei 10 μs.

Ergebnisse

Zunächst wird die Paketverlustrate für die unterschiedlichen Strategien betrachtet. Es fällt auf, dass HLU unabhängig von der Queuegröße selbst für geringe Datenraten verlustbehaftet ist (siehe Abbildung 7.14). Dies stimmt mit den Simulationen überein (siehe Abbildung 5.14), bei denen selbst bei genügend Gesamtrechenkapazität im System ein geringer Paketverlust auftritt. Nachdem jeder Flow einer CPU fest zugewiesen ist, kann ein Paket-Burst selbst bei im Schnitt niedrigen Datenraten zur temporären Überlastung einer CPU führen. Dabei kann die zweite CPU durchaus in Unterlast oder vollständig frei bleiben. Eine Umbalancierung findet aufgrund des relativ langen Adaptionsintervalls von 50 ms nicht unmittelbar statt. Bei *Spraying* dagegen werden beide CPUs gleichmäßig beansprucht. Ein einzelner Burst kann viel besser abgearbeitet werden. Unterhalb einer Datenrate von 50 MBit/s kommt es daher auch zu keinen Paketverlusten.

Es zeigt sich dabei auch, dass eine längere Queue gerade im Unterlastbereich durchaus hilfreich sein kann, um kurzfristige Lastspitzen abzufedern. Gerade bei 50 MBit/s wird für *HLU 16* eine Verlustrate von 0,28% erreicht, für *HLU 8* sind es dagegen schon 0,52%. Der Vorteil der längeren Queue schwindet jedoch mit steigender Datenrate oberhalb von 150 MBit/s. Hierbei dürften die Queues für beide HLU-Varianten aufgrund der hohen Last die meiste Zeit gefüllt sein. Ein neuer Burst findet daher in beiden Fällen in etwa die gleiche Anzahl an freien Plätzen vor. Der Vorteil von *HLU 16* ist dabei nur noch marginal. In etwa ab diesen Bereich steigt auch die Verlustrate beim *Spraying*

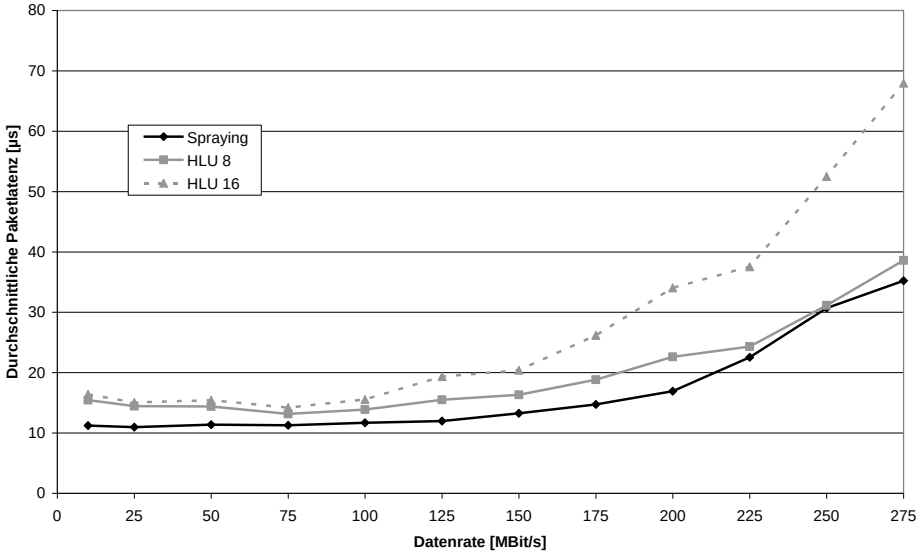

Abbildung 7.15: Durchschnittliche Paketlatenzen bei unterschiedlichen Eingangsdatenraten mit bursthaftem Verkehr für *Spraying* und HLU.

an, wenngleich sie auch hier aufgrund der besseren Balancierung merklich unterhalb der HLU-Varianten bleibt.

Die größere Queue und die damit einhergehende geringere Verlustrate bei *HLU 16* wird auf der anderen Seite durch eine höhere durchschnittliche Paketlatenz erkauft (siehe Abbildung 7.15). Beide HLU-Varianten liegen bei geringen Datenraten deutlich über der Latenz von *Spraying* (bei 10 MBit/s; *Spraying*: 11,2 μs; HLU 8: 15,4 μs; HLU 16: 16,4 μs). Auch dies lässt sich durch die nicht perfekte Ausbalancierung erklären. Während *Spraying* einen Burst mit beiden CPUs effizient bearbeitet, wird bei HLU ein Burst nur von einer CPU bearbeitet. Die Verweilzeit innerhalb der Queue steigt an. Während diese bei *HLU 8* auf acht Pakete reduziert ist, kann sie bei *HLU 16* auf über acht ansteigen. Andererseits werden genau diese Pakete bei *HLU 8* bereits verworfen.

Erstaunlich erscheint zunächst, dass sich die Latenzzeiten für hohe Datenraten zwischen *HLU 8* und *Spraying* angleichen, während *HLU 16* deutlich darüber liegt. Bei 275 MBit/s liegt sie für *Spraying* bei 35,2 μs, bei *HLU 8* bei 38,6 μs und bei *HLU 16* bei 68 μs. Andererseits erkennt man bereits an den Verlustraten, dass sich das System an diesem Punkt bei allen Verfahren bereits massiv im Überlastbereich befindet. Ein ankommendes Paket findet daher praktisch immer eine volle Queue vor. Dadurch wird die Latenz aber auch mehr oder weniger direkt von der Queue-Länge bestimmt. Diese ist bei *Spraying* mit 16 Paketen zwar doppelt so lang wie bei *HLU 8*. Allerdings wird diese auch von zwei CPUs bearbeitet und bei *HLU 8* lediglich von einer. Daher verwundert es nicht, dass sich die Latenzen der beiden Verfahren angleichen. Die Latenz von *HLU 16* dagegen steigt aufgrund der doppelten Queue-Länge auf nahezu den doppelten Wert.

Interessant ist auch die Betrachtung der Anzahl der *out-of-order*-Pakete nach der

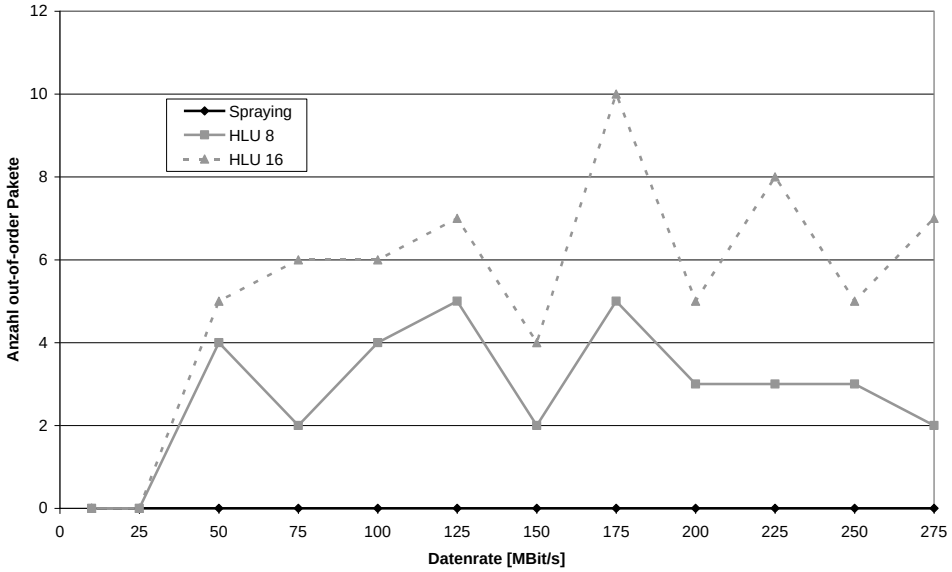

Abbildung 7.16: *Packet Reordering* nach Resequenzierung bei unterschiedlichen Eingangsdatenraten mit bursthaftem Verkehr für *Spraying* und HLU.

Resequenzierung (siehe Abbildung 7.16). Für *Spraying* konnten alle Pakete wieder in die korrekte Reihenfolge gebracht werden. Bei HLU konnten dagegen einzelne *out-of-order*-Pakete festgestellt werden. Bereits in Abschnitt 6.3.4 wurde auf die Auswirkungen des Lastbalancierungsverfahrens auf die Timeout-Zeit eingegangen. Es hat sich gezeigt, dass bei der Paketverteilung nach der Queue gemäß Formel 6.1 eine wesentlich geringere Timeout-Zeit zur sicheren Resequenzierung ausreichend ist als bei der Paketverteilung vor der Queue. Gemäß Formel 6.2 spielt hierbei die Queue-Tiefe eine entscheidende Rolle und liegt aus diesem Grunde für die dedizierten Verfahren wesentlich höher als bei *Spraying*. Möglicherweise findet eine Umbalancierung eines aktiven Bursts statt. Dabei wechselt der Flow von einer vollen CPU-Queue auf die andere, deutlich leerere CPU-Queue. Die gewählte Timeout-Zeit von 10 μs reicht in diesen Fällen dann u.U. nicht aus. Möglich ist aber auch, dass bei einer Umbalancierung die implementierte Resequenzierungs-Queue-Tiefe von acht Paketen nicht ausreicht, sofern die volle Eingangs-Queue überwiegend mit Paketen desselben Flows besetzt ist. Nichtsdestotrotz steigt die *Packet Reordering*-Rate auch in diesem Falle nie höher als $1,5 \cdot 10^{-4}\%$.

7.4 Zusammenfassung

In diesem Kapitel wurden die Vorteile der in den Kapiteln 4 bis 6 vorgestellten Konzepte mittels Messungen am FPGA-Demonstrator gezeigt und validiert.

Die Ergebnisse wiesen dabei den Vorteil des Einsatzes des *Post-Processors* sowohl bei der Prozessierung durch das CPU-Cluster, v.a. aber auch bei Verwendung von *AutoRoute*

nach. Es wurde ferner gezeigt, dass die Verwendung eines optimierten Softwareeinstiegpunkts die Prozessierung deutlich verbessern kann.

Weitere Schwerpunkte der Messungen waren gemäß dem Inhalt dieser Arbeit insbesondere die Paketverteilung von zustandlosem Verkehr mittels *Spraying* und die Resequenzierung der dadurch vertauschten Paketreihenfolge. Die Verteilung der Pakete mittels *Spraying* wird dabei durch den *Packet Distributor* ausgeführt. Es konnte gezeigt werden, dass insbesondere *Spraying* gegenüber einem dedizierten Lastbalancierungsverfahren deutliche Vorteile besitzt. So konnten durch eine effiziente Ausnutzung der Rechenressourcen sowohl Verlustraten als auch Paketlatenzen z.T. deutlich unter dem als Referenz herangezogenen HLU-Algorithmus gehalten werden. Der eigentliche und größte Nachteil von *Spraying* – nämlich das Fördern von *Packet Reordering* – wurde durch die Hardware-Resequenzierung erfolgreich unterbunden.

Trotz der Einschränkungen des Demonstrators (Beschränkung auf zwei CPUs, künstlicher Verkehr, Speicheranbindung), die im Wesentlichen auf die FPGA-Plattform und den zur Verfügung stehenden Messmöglichkeiten zurückzuführen sind, konnten alle wesentlichen Erkenntnisse bestätigt werden.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

In dieser Arbeit wurde auf die Paketverteilung und Resequenzierung im FlexPath-Netzwerkprozessor eingegangen. Die Hauptbeiträge dieser Arbeit lassen sich dabei wie folgt zusammenfassen:

- Im Rahmen dieser Arbeit wurden einige Kernaspekte des **FlexPath-Netzwerkprozessors** (FlexPath-NP) entwickelt. Der FlexPath-NP zeichnet sich durch eine paketabhängige Pfadwahl innerhalb des Systems aus. Hierbei wird versucht, je nach Pakettyp den jeweils günstigsten Pfad zu finden. Neben einer CPU-Bearbeitung sind hierbei auch Pfade mit teilweiser oder vollständiger Hardware-Bearbeitung möglich.
- Im Rahmen dieser Arbeit stand insbesondere die **ausgangsseitige Paketmodifikation** (*Post-Processor*) im Vordergrund. Mit dem *Post-Processor* wurde eine Hardware-Einheit konzipiert, die am laufenden Paketdatenstrom unterschiedliche Modifikationen vornehmen kann. Die genaue Bearbeitung, sowie die notwendigen Daten werden dem *Post-Processor* anhand eines Ausgangs-Kontexts pro Paket übergeben. Der *Post-Processer* hat in erster Linie die Aufgabe, einen **vollständigen Hardware-Bearbeitungspfad** (*AutoRoute*) innerhalb des NPs zu ermöglichen. Er kann aber auch als Hardware-Beschleuniger durch die CPU genutzt werden. Der Aufbau des *Post-Processors* ist modular gehalten und kann damit den jeweiligen Anforderungen an den Funktionsumfang sehr gut angepasst werden.
- Ferner wurde die **Paketverteilung** innerhalb eines Multiprozessor-Clusters untersucht. Mit dem *Packet Distributor* konnte eine Einheit entwickelt werden, die unterschiedliche, konfigurierbare Zuweisungen zwischen Eingangs-Queues und CPUs unterstützt und ausführt. Mit Hilfe des *Packet Distributors* kann somit sowohl eine dedizierte Zuweisung an eine CPU als auch ein Zuweisung innerhalb eines CPU-Pools ausgeführt werden. Kernstück des *Packet Distributor* ist dabei der **Multiprocessor Interrupt Controller** zur Benachrichtigung der Prozessoren. Der MPIntC ist primär für interruptbasierte Systeme konzipiert, er kann jedoch ohne weiterer Anpassungen auch für Polling-Systeme verwendet werden.
- Mit Hilfe des FlexPath-Systems können unterschiedliche Lastbalancierungsverfahren implementiert werden. Im Rahmen dieser Arbeit stand die Lastbalancierung von zustandslosem Verkehr im Vordergrund. Es wurde gezeigt, dass mit **Spraying** ein leicht umsetzbares, aber auch sehr effizientes Verfahren eingesetzt werden kann. *Spraying* verteilt dabei ankommende Pakete aus einer Queue an die jeweils

nächste freie CPU eines Clusters bzw. Teil eines Clusters. Im Gegensatz zu dedizierten Verfahren, bei denen Flows einer festen CPU zugeordnet werden, kommt es daher nicht zu einer teilweisen Überlastung einzelner CPUs bei insgesamt freien Rechenressourcen. Es konnte gezeigt werden, dass *Spraying* Vorteile bezüglich Paketverlustraten und -latenzen gegenüber anderen Verfahren (wie HABS oder HLU) besitzt. *Spraying* kann dabei sehr gut mit dedizierten Verfahren, die für zustandsbehafteten Verkehr Vorteile besitzen, kombiniert werden, sofern der Verkehr zuvor (wie es im FlexPath-NP möglich ist) aufgetrennt werden kann.

- Im Gegensatz zu dedizierten Verfahren kommt es bei *Spraying* durch die wahllose Verteilung von Paketen eines Flows auf unterschiedliche CPUs in gewissem Maße zu Paketüberholungen (*Packet Reordering*). *Packet Reordering* kann aber auch (wenn auch im weit geringerem Maß) bei Umbalancierungen der dedizierten Verfahren auftreten. *Packet Reordering* hat aber nachweislich einen negativen Effekt auf die Netzwerkgeschwindigkeit. Insbesondere potenziert sich der Effekt schon bei relativ kleinen *Packet Reordering*-Raten, da ein Paket zwischen Quelle und Ziel in der Regel eine Vielzahl von Routern durchlaufen muss. Als Lösung wurde eine **Hardware-Resequenzierungseinheit** präsentiert. Durch ein hashbasiertes Verfahren konnte eine effiziente Architektur vorgestellt werden, die *Packet Reordering* zuverlässig und nahezu vollständig verhindert. Durch das Zulassen von einzelnen *out-of-order*-Paketen in Ausnahmesituationen kann zudem der Hardware-Aufwand auf ein Minimum reduziert werden.

Die vorgestellten Verfahren und Konzepte wurden anhand eines **SystemC-Simulationsmodells** evaluiert. Neben der einfachen Konfigurierbarkeit des Systems (z.B. CPU-Clustergröße) war hierbei v.a. die Verwendbarkeit von echten Verkehrsmitschnitten (*pcap*-Dateien) von entscheidendem Vorteil. Gerade bei der Beurteilung der Lastbalancierung und auch der Resequenzierung ist ein realistischer Verkehr von großer Bedeutung für die Aussagekraft der Ergebnisse. In den Simulationen konnten die Vorteile der vorgestellten Konzepte bestätigt werden. Zudem konnten wertvolle Hinweise hinsichtlich der Dimensionierung einzelner Komponenten erhalten werden.

Es konnte gezeigt werden, dass *Spraying* eine für zustandslosen Verkehr optimale Lastverteilung erzeugt. In den Simulationen wurden dabei jedoch *Packet Reordering*-Raten von bis zu 1,28% erreicht. Durch Einsatz der Hardware-Resequenzierungseinheit konnte der Anteil jedoch auf unter 0,000023% reduziert werden. Die Hardware-Resequenzierung ist dabei besonders für *Spraying*-Pakete effizient. Durch Verwendung des hashbasierten Verfahrens besteht die Gefahr von Kollisionen innerhalb der Resequenzierungs-Verwaltung. Kollisionen zwischen Pakete unterschiedlicher Verkehrsklassen (z.B. Forwarding - IPsec) wurden dabei in der Flow-ID-Berechnung ausgeschlossen. Damit konnten die maximalen Latenzen bei Verkehrs-Mix z.B. für hochpriore Pakete von 71,26 μs auf 26,25 μs gesenkt werden.

Durch den Aufbau des FlexPath-Systems mit den beschriebenen Komponenten als **FPGA-Demonstrator** konnte die Implementierbarkeit der vorgestellten Konzepte nachgewiesen werden. Trotz gewisser Einschränkungen insbesondere im Hinblick auf CPU-

Anzahl und *Interconnect*-Geschwindigkeit konnten hierbei diverse Messungen durchgeführt werden, die zum einen als Eingangs-Informationen für die Simulationen verwendet werden konnten. Zum anderen bestätigten die Messungen im Wesentlichen die Ergebnisse der durchgeführten Simulationen. Es hat sich aber auch gezeigt, dass die im Demonstrator verwendete Speicherstruktur und Anbindung an den gemeinsamen Bus unzureichend für die mit der Paketgröße anwachsenden Datenraten ist.

So führte die Hardware-Unterstützung durch den *Post-Processor* zu einer Steigerung der Forwarding-Rate um 12% von 286,5 kpps auf 320,9 kpps. Einen wesentlich größeren Paketdurchsatz erreicht jedoch der *AutoRoute*-Pfad, bei dem ein Durchsatz von 827 kpps gemessen wurde. Mithilfe des Demonstrators konnte zudem die Wirksamkeit von *Spraying* für zustandslosen Verkehr nachgewiesen werden. Die Implementierung der Hardware-Resequenzierungseinheit konnte dabei sämtliche *out-of-order*-Pakete resequenzieren.

8.2 Ausblick

Die Implementierung des FlexPath-NPs wurde auf Basis einer FPGA-Plattform durchgeführt. Dabei konnten lediglich zwei CPUs mit 200 MHz bei 100 MHz Systemtakt eingesetzt werden. Die absoluten Zahlen der Messungen sind aus diesem Grunde nicht mit denen kommerzieller Produkte vergleichbar. Da die vorgestellten Konzepte allerdings ausnahmslos auf ASIC-Technologie übertragbar sind, sind die relativen Steigerungen der Prozessierungsgeschwindigkeit entscheidend. Die Simulationen haben dabei bewiesen, dass die vorgestellten Konzepte auch in einer echten Multicore-Umgebung Bestand haben. Dabei können viele NP-Architekturen – auch kommerzielle – von den FlexPath-Konzepten profitieren. Es ist dabei keine Grundvoraussetzung, dass das FlexPath-Konzept als Ganzes übernommen wird. Vielmehr besteht FlexPath aus einer Reihe von Konzepten, die auch in Teilen integriert werden können.

Im Hinblick auf eine Verbesserung der Demonstrationsumgebung zeigt sich, dass eine Optimierung des *Interconnects* dringend geboten ist. Auch hierbei kann die Klassifikation am Eingangsdatenpfad wertvolle Dienste leisten. Insbesondere die *AutoRoute*-Pakete müssen vom Prozessor-Cluster aus nicht zugänglich sein, da sie auf direktem Weg per Hardware bearbeitet werden. Das Ausnützen der Pfadentscheidung bereits beim Speichern würde für diese Pakete einen direkten Bypass vom Ein- zum Ausgang ermöglichen. Das würde zum einen *Interconnect* und Speicher merklich entlasten, es würde aber auch den *AutoRoute*-Pfad weiter beschleunigen. Für das *Interconnect* selbst wiederum sind eine Reihe von Alternativen bis hin zum *Network-on-Chip* (NoC) denkbar.

Mit *Spraying* wurde eine effektive Methode zur Lastbalancierung von zustandslosem Verkehr vorgestellt. *Spraying* eignet sich daher v.a. in Bereichen eines Netzes, in denen sich die Bearbeitung weitestgehend auf Forwarding beschränkt, also z.B. im *Backbone*-Bereich. Im FlexPath wird *Spraying* dagegen nicht für zustandsbehafteten Verkehr eingesetzt, da es einfacher und effektiver ist, Zustandsinformationen lokal bei einer CPU zu speichern, weil ein zentraler Speicher Probleme bezüglich Datenkonsistenz mit sich bringt. Es gibt jedoch auch NP-Implementierungen wie den Cisco QuantumFlow Prozes-

sor (siehe S. 41), die genau solch einen zentralen Zustandsspeicher zur Verfügung stellen. Hierbei gilt es zu untersuchen, inwieweit somit auch ein *Spraying* zustandsbehafteter Pakete möglich wäre. Damit würde sich die Lastbalancierung selbst weiter vereinfachen, ein kombiniertes Verfahren aus *Spraying* und HLU und damit eine aufwändige Vorklassifizierung im *Path Dispatcher* wäre für die Lastbalancierung selbst nicht mehr notwendig.

In diesem Szenario wäre die vorgestellte Hardware-Resequenzierung umso wichtiger. Sie ist Grundvoraussetzung beim Einsatz des *Sprayings*. Die vorgestellte Resequenzierung bezog sich lediglich auf Paketvertauschungen innerhalb des NPs. *Packet Reordering* ist jedoch auch allgemein ein Problem in Netzwerken, da verschiedene Pakete unterschiedliche Wege zum Erreichen desselben Ziels verwenden können. Weiterführende Untersuchungen könnten zeigen, inwieweit sich die vorgestellte Hardware-Resequenzierung ganz allgemein zur Resequenzierung von Paketen am Zielort einsetzen lässt. Anstatt eines expliziten *Taggings* beim Start (siehe *Ingress Tagger*), könnte hierzu die Sequenznummer z.B. des TCP-Protokolls verwendet werden. Kollisionen wären dabei vermutlich strengstens zu vermeiden, da eine Kollision zweier unabhängiger TCP-Flows zu nicht konsekutiven (TCP-)Sequenznummern führen würde. Damit stellt sich die Frage, ob das vollständige IP 5-Tupel als Flow-ID verwendet werden müsste, oder ob eine Kollisions-Auflösung andere Möglichkeiten bietet.

Auch beim *Multiprocessor Interrupt Controller* ist noch Entwicklungspotential vorhanden. Insbesondere eine konfigurierbare Interrupt-Priorität u.U. auf CPU-Basis ist denkbar. Dies würde die Flexibilität und Einsatzmöglichkeiten des Controllers erweitern, da gleiche Interrupt-Eingänge von unterschiedlichen CPUs mit unterschiedlicher Priorität bearbeitet werden könnte. Andererseits würde damit auch wiederum die Komplexität und damit der Ressourcenverbrauch des Controllers merklich steigen.

Beim *Post-Processor* gibt es schließlich – wie bereits in Abschnitt 4.5 angedeutet – Optimierungsmöglichkeiten bei der lokalen Speicherung häufig benötigte Daten bzw. der Verwendung von Befehlstemplates. Dies würde den Kommunikationsaufwand insbesondere im Fall von *AutoRoute* zwischen Ein- und Ausgangsdatenpfad des NPs reduzieren.

Begriffserklärungen & Abkürzungen

AutoRoute, AR Hardware-Verarbeitungspfad im FlexPath-Netzwerkprozessor.

ARP *Address Resolution Protocol.* Protokoll zur Ermittlung der physikalischen Adresse (z.B. Ethernet-Adresse) auf Basis der IP-Adresse.

ASIC *Application Specific Integrated Circuit.* Anwendungsspezifische Integrierte Schaltung. Eine Schaltung, deren Funktionalität direkt in Hardware übertragen wurde und daher eine hohe Rechenleistungsdichte besitzt. Die Funktionalität lässt sich jedoch nachträglich nicht mehr ändern.

CAM *Content Addressable Memory.* Assoziativspeicher. Eine spezielle Speicherstrukur, die zu einem vorgegeben Datenwort die passende Adresse ausgibt.

CII *Context Information Input.* Eingangs-Kontext in FlexPath, der relevante Daten (z.B. IP-Adressen, Ports) über das Paket enthält. Dient der CPU zur Prozessierung.

CIO *Context Information Output.* Ausgangs-Kontext in FlexPath mit Anweisungen für den *Post-Processor*.

CPI *Cycles per Instruction.* Anzahl der Taktzyklen zur Ausführung einer Instruktion in einem Prozessor.

DCR *Device Control Register.* 32-Bit *Single Master* Bus am *PowerPC* zur Konfiguration der Peripherie.

DMA *Direct Memory Access.* Speicherzugriffsart in einem System, bei der der Peripherie der direkte Zugriff auf den Systemspeicher zum Lesen und/oder Schreiben erlaubt wird.

Flow Zusammengehörige Pakete mit gleichen Eigenschaften, z.B. identischem IP 5-Tupel (z.B. Pakete derselben TCP-Verbindung).

FPGA *Field Programmable Gate Array.* Programmierbarer Logikbaustein.

FSM *Finite State Machine.* Endlicher Zustandsautomat.

GPP *General Purpose Processor.* Programmierbarer Prozessor, der für keinen konkreten Anwendungszweck optimiert ist.

HABS *Hashing Adapted by Burst Shifting.* Lastbalancierungs-Algorithmus für dedizierten Verkehr nach Kencl und Shi (siehe [55]).

HLU *Hash-Lookup.* Lastbalancierungs-Algorithmus für dedizierten Verkehr im FlexPath-NP (siehe [3]).

IMIX *Internet Mix.* Paketgrößenverteilung, die die reale Verteilung im Internet zu Testzwecken bzw. Durchsatzmessungen abstrahiert. Beim sogenannten *Simple IMIX* besteht diese zu 7/12 aus 64-Byte großen Paketen, zu 4/12 aus 594-Byte großen

Paketen und zu 1/12 aus 1518-Byte großen Paketen.

IPR *Interrupt Pending Register*. Register im →*MPIntC*, das die jeweils aktiven Interrupts einer CPU enthält.

ISR *Interrupt Service Routine*. Programmcode, der in Abhängigkeit von der Interruptnummer beim Interrupt ausgeführt wird.

MAC *Media Access Control/Media Access Controller*. Einheit zum Umsetzen der Sicherungsschicht (*Media Access Control*) bei Ethernet

MPIntC *Multiprocessor Interrupt Controller*. Interrupt Controller zum Ansteuern mehrerer paralleler Prozessoren.

NP *Netzwerkprozessor*. Zur Paketverarbeitung optimierter Chip, oftmals bestehend aus einem Prozessor-Cluster mit für den Anwendungsfall optimierter Hardwareunterstützung.

OPB *On-Chip Peripheral Bus*. Busstandard der Firma IBM mit 32 Bit Datenbreite.

Out-of-order →*Packet Reordering*.

Packet Reordering Man spricht von *Packet Reordering*, wenn Pakete eines Flows nicht mehr die originale Paketreihenfolge besitzen. Die Pakete sind somit *out-of-order*.

Pcap *Packet Capture*. Programmierschnittstelle und Dateiformat zum Mitschneiden und Abspeichern von Paketen. Enthält neben der zeitlichen Relation (Zeitstempel) zudem Paketdaten bzw. Teile der Paketdaten. Bibliotheken sind verfügbar für Linux (*libpcap*) und Windows (*WinPcap*).

PLB *Processor Local Bus*. Busstandard der Firma IBM mit separatem Lese- und Schreibbus. Die verwendete Datenbreite im Rahmen dieser Arbeit lag bei 64 Bit. Standard-Schnittstelle zu den verwendeten *PowerPC*-CPUs.

Resequenzierung Wiederherstellung der originalen Paketreihenfolge eines Flows.

RISC *Reduced Instruction Set Computer*. CPU mit reduziertem Befehlssatz (im Kontrast zu *Complex Instruction Set Computer*, *CISC*)

Spraying Lastbalancierungsverfahren im FlexPath-NP für zustandslosen Verkehr. Dabei werden die Pakete eines Flows zur Bearbeitung auf mehrere CPUs verteilt.

SystemC Simulationsumgebung (basierend auf C++) zur Simulation komplexer Systeme auf hoher Abstraktionsebene.

TCAM *Ternary Content Addressable Memory*. Spezielle Form von →*CAM*, der *don't cares* innerhalb der Suche erlaubt.

VHDL ***V****ery High Speed Integrated Circuit* ***H****ardware* ***D****escription* ***L****anguage*. Eine Hardwarebeschreibungssprache.

Abbildungsverzeichnis

2.1 OSI Referenzmodell . . . 18
2.2 Ethernet Header nach IEEE 802.3 . . . 20
2.3 Nutzdatenrate für Ethernet, IP und TCP in Abh. der Framegröße . . . 21
2.4 IPv4-Header . . . 22
2.5 IP-Routing Beispiel . . . 23
2.6 IPv6-Header . . . 24
2.7 TCP/UDP Headerstruktur . . . 25
2.8 IP-Paket mit ESP . . . 27
2.9 Exemplarische Netzwerk-Infrastruktur . . . 32
2.10 Triple Play im DSLAM . . . 34

3.1 Design Space bei NP Implementierungen . . . 37
3.2 Paketverarbeitung im Cluster und Pipeline . . . 38
3.3 Xelerated Synchrone Datenfluss Architektur . . . 40
3.4 Architektur des Cisco QuantumFlow Prozessors . . . 41
3.5 Architektur des EZchip TOPcore . . . 42
3.6 Architektur der SafeNet Inline Security Engine . . . 43
3.7 Architektur des Reprogrammable Pipeline Modules (PRO3) . . . 44
3.8 Table-Based Hashing nach Cao . . . 47
3.9 Lastbalancierung nach Dittmann . . . 48
3.10 Lastbalancierung nach Kencl und Shi (HABS) . . . 50
3.11 Load Balancing Switch nach Chang . . . 52
3.12 Per-flow Resequencing in Switches nach Cheng . . . 53
3.13 Resequenzierung nach Wu . . . 56
3.14 Flow-Tabelle in der Resequenzierung nach Wu . . . 58
3.15 IBM Multiprocessor Interrupt Controllers im MP-System . . . 60
3.16 Architektur des IBM Multipr. Interrupt Controllers . . . 61

4.1 Gegenüberstellung herkömmlicher NP zu FlexPath-NP . . . 65
4.2 Prioritäts-Klassifikation bei herkömmlichen NP Architekturen und bei FlexPath . . . 68
4.3 Konzept des FlexPath-Netzwerkprozessors . . . 69
4.4 Einfügen von IPsec-Headern bei fester Paketspeichergröße . . . 72
4.5 Einfügen von IPsec-Headern bei Linux Socket Buffers . . . 72
4.6 Verarbeitung als Paketdatenstrom im Post-Processor . . . 74
4.7 Post-Processor Architektur . . . 76

4.8 Anordnung der Verarbeitungseinheiten im Post-Processor 76
4.9 Datenpfad beim Post-Processor 77
4.10 Beispiel zum Löschen von Paketdaten 77
4.11 Schrittweises Vorgehen beim Löschen im Post-Processor 78
4.12 Schrittweises Vorgehen beim Einfügen im Post-Processor 79
4.13 CIO Beispiel für ein AutoRoute-Paket 80
4.14 Architektur des Kontrollpfads beim Post-Processor 81
4.15 Datensortierung beim Post-Processor 82
4.16 TAPES SystemC-Modell 86
4.17 Systemdurchsatz in Abhängigkeit des AutoRoute-Anteils 87
4.18 Paketlatenzen in Abhängigkeit des AutoRoute-Anteils 88

5.1 Packet Spraying nach Dittmann 94
5.2 Packet Spraying im FlexPath-NP 95
5.3 Zustandsspeicher im Multiprozessor-System 96
5.4 Prioritätsbasierte Lastbalancierung im FlexPath-NP 98
5.5 Multiprocessor Interrupt Controller im Multiprozessor-System 101
5.6 Interne Registerstruktur des MPIntC 103
5.7 Architektur des Packet Distributors 105
5.8 Head of Line Blocking im Multiprozessor-System 107
5.9 Anschluss der Interrupt Eingängen im FlexPath-NP 109
5.10 Adresstabelle des Packet Distributors 110
5.11 Skalierbarkeit des MPIntC in Abhängigkeit der Interrupt-Eingänge 112
5.12 Skalierbarkeit des MPIntC in Abhängigkeit der CPUs 113
5.13 Funktionales SystemC Simulationsmodell 115
5.14 Verlustraten für HABS, HLU und Spraying in Abh. der Clustergröße . . . 118
5.15 CPU-Auslastung für HABS, HLU und Spraying in Abh. der Clustergröße 119
5.16 Zeitlicher Verlauf der CPU-Auslastung für HABS, HLU und Spraying . . 121
5.17 Paketlatenzen für HABS, HLU und Spraying in Abh. der Clustergröße . . 122
5.18 Verlustraten für HABS, HLU und Spraying in Abh. der Clustergröße . . . 122
5.19 Paketlatenzen für HABS, HLU und Spraying in Abh. der Queuegröße . . 123
5.20 Packet Reordering bei Spraying in Abhängigkeit der Clustergröße 124
5.21 Paketlatenzen für HABS und S&H in Abhängigkeit der Clustergröße . . . 125
5.22 Zeitlicher Verlauf der CPU-Auslastung für HABS und S&H 126

6.1 Paket Resequenzierung im FlexPath-NP 131
6.2 Jitterverteilung im FlexPath Demonstrator 133
6.3 Resequenzierungs-Beispiel mit dynamischen Queues 134
6.4 Resequenzierungs-Beispiel mit statischen Queues 135
6.5 Ringpuffer-Struktur und Resequenzierungs-Beispiel 138
6.6 Sequenznummer-Zahlenkreis 140
6.7 Resequenzierungs-Beispiel bei Überschreiten der Queue-Länge 141
6.8 Hash-Kollisionsrate in Abh. der Flow-ID Bitbreite 143
6.9 Speicherbedarf in Abh. der Flow-ID Bitbreite 144

6.10 Offset-Verteilung bei zu resequenzierenden Paketen 145
6.11 Einfluss der Lastbalancierung auf den Prozessierungsjitter 149
6.12 Packet Reordering vor und nach Resequenzierung in Abh. der Clustergröße152
6.13 Packet Reordering in Abhängigkeit der Timeout-Zeit bei IPsec 154
6.14 Zeitlicher Verlauf der maximalen Latenz mit und ohne Resequenzierung . 156
6.15 Paketlatenzen in Abh. der Clustergröße bei Verkehrs-Mix 157
6.16 Paket-Deskriptor im FlexPath 157
6.17 Blockdiagramm der Aggregation Unit 158
6.18 Zeitl. Verlauf der min. & max. Latenz bei adaptiver Timeout-Anpassung . 163
6.19 Zeitl. Verlauf der Timeout-Zeit bei adaptiver Timeout-Anpassung 164
6.20 Zeitl. Verlauf von Latenzen und Timeout-Zeit beim adaptiven Verfahren . 165

7.1 ML410 Development Board Bild 170
7.2 FlexPath-NP Demonstrator Blockschaltbild 171
7.3 Speicherorganisation im FlexPath-Demonstrator 172
7.4 Demonstrator Messaufbau 175
7.5 Paketlatenzen bei Optimierung des Packet Distributors 176
7.6 CPU-Paketdurchsatz mit und ohne Hardware-Unterstützung 179
7.7 CPU-Datendurchsatz mit und ohne Hardware-Unterstützung 179
7.8 Paketdurchsatz bei AutoRoute 180
7.9 Paketübermittlung an eine CPU ohne und mit Vorklassifizierung 181
7.10 Durchsatz in Abhängigkeit des Software-Einstiegspunkts 182
7.11 Packet Reordering bei Spraying ohne Resequenzierung bei unterschiedliche Datenraten 183
7.12 Packet Reordering bei Spraying ohne Resequenzierung bei unterschiedlicher Anzahl Flows 184
7.13 Paketlatenzen bei Spraying mit und ohne Resequenzierung 185
7.14 Paketverlust bei Spraying und HLU 188
7.15 Paketlatenzen bei Spraying und HLU 189
7.16 Packet Reordering nach Resequenzierung bei Spraying und HLU 190

Tabellenverzeichnis

3.1 Simulations-Parameter zur Resequenzierung nach Wu 58

4.1 Anzahl der Assembler Instruktionen IP-Forwarding (lwIP) 66
4.2 Ressourcenverbrauch des Post-Processors 83
4.3 Geschätzte Reduktion der Prozessierungslatenz zur HW-Offload 87

5.1 Ressourcenverbrauch des Packet Distributors 111
5.2 Struktur des Interrupt Pending Registers und Paket-Deskriptors im Packet Distributor 113
5.3 Kenndaten der pcap-Verkehrsmitschnitte 117
5.4 Verarbeitungslatenzen und Verlustraten bei HABS und FlexPath 127

6.1 Puffergrößen zur Resequenzierung für 16 Bit Flow-ID 136
6.2 Max. Anzahl an Paketen im System 146
6.3 Paketlatenzen in Abh. der Flow-ID Struktur 146
6.4 Packet Reordering nach Resequenzierung in Abh. der Clustergröße 153
6.5 Paketlatenzen in Abh. des IPsec-Timouts 155
6.6 Ressourcenverbrauch bei der Hardware Resequenzierung 159
6.7 Speicherverbrauch in der Aggregation Unit 159
6.8 Maximale Paketraten der Aggregation Unit in Abhängigkeit von Packet Reordering bzw. Paketverlusten. 161

7.1 Ressourcenverbrauch im FlexPath Demonstrator 173
7.2 Flowverteilung zur Messung der Lastbalancierungsverfahren 187

Literaturverzeichnis

[1] Bundesverband Informationswirtschaft Telekommunikation und neue Medien e.V., "Fast jeder fünfte Mensch ist online." `http://www.bitkom.org/de/presse/49919_46069.aspx`, 2007. [Online; accessed 04.11.2009].

[2] Cisco, "Ab 2008 nutzen private Anwender stärker das Internet als Unternehmen." `http://www.cisco.com/web/AT/assets/docs/Cisco_IPTraffic_060907.pdf`, 2007. [Online; accessed 04.11.2009].

[3] R. Ohlendorf, *A Network Processor Architecture with Application-Optimized Reconfigurable Processing Paths (FlexPath NP)*. Dissertation, Technische Universität München, München, Deutschland, 2011.

[4] ISO, *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. ISO, Feb. 1995. ISO/IEC 7498-1:1994.

[5] A. Badach and E. Hoffmann, *Technik der IP-Netze. TCP/IP incl. IPv6.* Carl Hanser Verlag, München, 2001.

[6] A. Morton, L. Ciavattone, G. Ramachandran, S. Shalunov, and J. Perser, "RFC4737: Packet Reordering Metrics." `http://tools.ietf.org/html/rfc4737`, Nov. 2006.

[7] A. Meddeb, "Why ethernet WAN transport?," *Communications Magazine, IEEE*, vol. 43, pp. 136 – 141, Nov. 2005.

[8] "IEEE Std 802.3 - 2005 Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications," *IEEE Std 802.3-2005 (Revision of IEEE Std 802.3-2002 including all approved amendments)*, vol. Section1, 2005.

[9] J. Postel, "RFC 791: Internet Protocol." `http://www.ietf.org/rfc/rfc0791.txt`, Sept. 1981.

[10] D. Plummer, "RFC826: Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware." `http://tools.ietf.org/html/rfc826`, Nov. 1982.

[11] S. Deering and R. Hinden, "RFC2460: Internet Protocol, Version 6 (IPv6)." `http://tools.ietf.org/html/rfc2460`, Dec. 1998.

[12] W. John and S. Tafvelin, "Analysis of internet backbone traffic and header anomalies observed," in *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, (New York, NY, USA), pp. 111–116, ACM, 2007.

[13] Defense Advanced Research Projects Agency, "RFC793: Transmission Control Protocol." `http://tools.ietf.org/html/rfc793`, Sept. 1981.

[14] M. Allman, V. Paxson, and W. Stevens, "RFC2581: TCP Congestion Control." `http://tools.ietf.org/html/rfc2581`, Apr. 1999.

[15] M. Laor and L. Gendel, "The Effect of Packet Reordering in a Backbone Link on Application Throughput," *IEEE Network*, no. 16, pp. 28–36, 2002.

[16] S. Govind, R. Govindarajan, and J. Kuri, "Packet Reordering in Network Processors," IPDPS 2007, 2007.

[17] J. Postel, "RFC 768: User Datagram Protocol." `http://tools.ietf.org/html/rfc768`, Aug. 1980.

[18] S. Kent and K. Seo, "RFC 4301: Security Architecture for the Internet Protocol." `http://tools.ietf.org/html/rfc4301`, Dec. 2005.

[19] S. Kent, "RFC 4302: IP Authentication Header." `http://tools.ietf.org/html/rfc4302`, Dec. 2005.

[20] S. Kent, "RFC 4303: IP Encapsulating Security Payload (ESP)." `http://tools.ietf.org/html/rfc4303`, Dec. 2005.

[21] N. Chen, "Software Implementation of IPsec in FlexPath," Master's thesis, Technische Universität München, Munich, Germany, December 2007.

[22] C. Kaufman, "RFC 4306: Internet Key Exchange (IKEv2) Protocol." `http://tools.ietf.org/html/rfc4306`, Dec. 2005.

[23] J. Wroclawski, "RFC 2210: The Use of RSVP with IETF Integrated Services." `http://tools.ietf.org/html/rfc2210`, Sept. 1997.

[24] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "RFC 2475: An Architecture for Differentiated Services." `http://tools.ietf.org/html/rfc2475`, Dec. 1998.

[25] H. Rauchfuß, "Ausarbeitung zum Hauptseminar: Integrated Services and Differentiated Services," tech. rep., Technische Universität München, Germany, 2005.

[26] K. Mochalski and H. Schulze, "Ipoque Whitepaper: Deep Packet Inspection." `http://www.ipoque.com/userfiles/file/DPI-Whitepaper.pdf`, 2009.

[27] M. Meitinger, R. Ohlendorf, T. Wild, and A. Herkersdorf, "Application Scenarios for FlexPath NP," tech. rep., Technische Universität München, Germany, 2005.

[28] ITWissen, "Access-Router." `http://www.itwissen.info/definition/lexikon/Access-Router-access-router.html`, 2010. [Online; accessed 08.01.2010].

[29] Agere Systems, "Implementing Demanding Traffic Processing Requirements for Next-Generation DSLAM Network Architectures," 2005. [Online; accessed 16.12.2005].

[30] Intel, "Whitepaper: DSL Access Multiplexer, Intel Network & Communications Design Solutions, Chapter 12," 2005.

[31] Juniper Networks, "Whitepaper: Optimizing Architecture for IPTV and Video on Demand," 2005.

[32] UTStarcom, "Whitepaper: Deploying IP-Base xDSL Service with AN-2000^{TM}IP-DSLAM," 2004.

[33] E. Rosen, A. Viswanathan, and R. Callon, "RFC3031: Multiprotocol Label Switching Architecture." `http://tools.ietf.org/html/rfc3031`, Jan. 2001.

[34] N. Shah, "Understanding Network Processors," Master's thesis, University of California, Berkeley, USA, September 2001.

[35] J. Carlström and T. Bodén, "Synchronous dataflow architecture for network processors," vol. 24, pp. 10–18, 2004.

[36] Xelerated, "Product Brief: Xelerator^{TM}X10q Network Processor." `http://www.xelerated.com/uploads/files/62.pdf`, 2009. [Online; accessed 21.10.2009].

[37] Xelerated, "Product Brief: Xelerator^{TM}X11 Network Processor." `http://www.xelerated.com/uploads/files/5.pdf`, 2009. [Online; accessed 21.10.2009].

[38] Xelerated, "Product Brief: HX320 Series Network Processor." `http://www.xelerated.com/uploads/files/95.pdf`, 2009. [Online; accessed 21.10.2009].

[39] Xelerated, "Product Brief: HX330 Series Network Processor with Traffic Manager." `http://www.xelerated.com/uploads/files/97.pdf`, 2009. [Online; accessed 21.10.2009].

[40] Netronome, "Product Brief: NFP-3200 Network Flow Processor." `http://www.netronome.com/files/file/Netronome%20NFP%20Product%20Brief%20%283-09%29.pdf`, 2009. [Online; accessed 23.10.2009].

[41] Cisco Systems Inc., "The Cisco QuantumFlow Processor: Cisco's Next Generation Network Processor," 2008.

[42] Cavium Networks, "Product Brief: OCTEON^{TM}II CN63XX Mulit-Core MIPS64 Processors." `http://www.caviumnetworks.com/pdfFiles/CN63XX_PB%20%20Rev%201.pdf`, 2009. [Online; accessed 18.12.2009].

[43] EZchip Technologies, "White Paper: Network Processor Designs for Next-Generation Networking Equipment." `http://www.ezchip.com/Images/pdf/ezchip_white_paper.pdf`, 1999. [Online; accessed 18.12.2009].

[44] SafeNet Inc., "SafeXcel IP Inline Security Engine: Product Brief." `http://www.safenet-inc.com/library/emb/SafeNet_Product_Brief_SafeXcel_IP_ISE.pdf`.

[45] I. Papaefstathiou, S. Perissakis, T. G. Orphanoudakis, N. A. Nikolaou, G. Komaros, N. A. Zervos, G. Konstantoulakis, D. N. Pnevmatikatos, and K. Vlachos, "PR03: a hybrid NPU architecture," vol. 24, pp. 20–33, 2004.

[46] G. Lykakis, N. Mouratidis, K. Vlachos, N. Nikolaou, S. Perissakis, G. Sourdis, G. Konstantoulakis, D. Pnevmatikatos, and D. Reisis, "Efficient field processing cores in an innovative protocol processor system-on-chip," in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pp. 14–19 suppl., 2003.

[47] S. Hauger, "A novel architecture for a high-performance network processing unit: Flexibility at multiple levels of abstraction," in *High Performance Switching and Routing, 2009. HPSR 2009. International Conference on*, pp. 1–7, June 2009.

[48] C. Albrecht, J. Foag, R. Koch, and E. Maehle, "DynaCORE — A Dynamical-

ly Reconfigurable Coprocessor Architecture for Network Processors," in *Parallel, Distributed, and Network-Based Processing, 2006. PDP 2006. 14th Euromicro International Conference on*, pp. 101–108, Feb. 2006.

[49] T. Pionteck, R. Koch, C. Albrecht, E. Maehle, M. Meitinger, R. Ohlendorf, T. Wild, and A. Herkersdorf, "SPP1148 Booth: Network Processors," FPL 2008, (Heidelberg, Germany), 2008.

[50] Z. Cao, Z. Wang, and E. Zegura, "Performance of hashing-based schemes for Internet load balancing ," in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1, pp. 332–341 vol.1, 2000.

[51] G. Dittmann and A. Herkersdorf, "Network Processor Load Balancing for High-Speed Links," SPECTS 2002, 2002.

[52] L. Kencl and J.-Y. Le Boudec, "Adaptive load sharing for network processors," in *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, pp. 545–554 vol.2, 2002.

[53] N. Brownlee and K. Claffy, "Understanding Internet traffic streams: dragonflies and tortoises," *Communications Magazine, IEEE*, vol. 40, pp. 110 – 117, Oct. 2002.

[54] W. Shi, M. H. MacGregor, and P. Gburzynski, "Load Balancing for Parallel Forwarding," vol. 13, pp. 790–801, 2005.

[55] W. Shi and L. Kencl, "Sequence-Preserving Adaptive Load Balancers," ANCS'06, (San Jose, CA, USA), 2006.

[56] W. Shi, M. MacGregor, and P. Gburzynski, "A scalable load balancer for forwarding internet traffic," in *Architecture for networking and communications systems, 2005. ANCS 2005. Symposium on*, pp. 145–152, Oct. 2005.

[57] C.-S. Chang, D.-S. Lee, and Y.-S. Jou, "Load balanced Birkhoff-von Neumann switches, part I: one-stage buffering," *Computer Communications*, vol. 25, no. 6, pp. 611–622, 2002.

[58] C.-S. Chang, D.-S. Lee, and C.-M. Lien, "Load balanced Birkhoff-von Neumann switches, part II: multi-stage buffering," *Computer Communications*, vol. 25, no. 6, pp. 623–634, 2002.

[59] H. Cheng, Y. Jin, Y. Gao, Y. Yu, W. Hu, and N. Ansari, "Per-flow Re-sequencing in Load balancing Switches by Using Dynamic Mailbox Sharing," ICC 2009, (Beijing, China), 2009.

[60] Intel, "Intel® IXP2400 Network Processor: Flexible, High-Performance Solution for Access and Edge Applications White Paper." `http://www.intel.com/design/network/papers/ixp2400.htm`, 2002. [Online; accessed 11.12.2009].

[61] B. Wu, Y. Xu, H. Lu, and B. Liu, "A Practical Packet Reordering Mechanism with Flow Granularity for Parallelism Exploiting in Network Processors," IPDPS 2005, 2005.

[62] International Business Machines Corporation, *Multiprocessor Interrupt Controller*

- *Data Book*, Mar. 2006. Preliminary.

[63] D. Llorente, *SmartMem - An Advanced Memory Subsystem for Networking Applications*. Dissertation, Technische Universität München, München, Deutschland, 2011.

[64] R. Ohlendorf, A. Herkersdorf, and T. Wild, "FlexPath NP - A Network Processor Concept with Application-Driven Flexible Processing Paths," CODES+ISSS 2005, (Jersey City, NJ, USA), 2005.

[65] "lwIP - A Lightweight TCP/IP stack." `http://savannah.nongnu.org/projects/lwip/`. [Online; last accessed 26.01.2010].

[66] R. Ohlendorf, T. Wild, M. Meitinger, H. Rauchfuss, and A. Herkersdorf, "Simulated and measured performance evaluation of RISC-based SoC platforms in network processing applications," *Journal of Systems Architecture*, vol. 53, no. 10, pp. 703–718, 2007.

[67] R. Ohlendorf, M. Meitinger, T. Wild, and A. Herkersdorf, "A Packet Classification Technique for On-Chip Processing Path Selection," WASP 2007, (Salzburg, Austria), 2007.

[68] R. Ohlendorf, M. Meitinger, T. Wild, and A. Herkersdorf, "A Processing Path Dispatcher in Network Processor MPSoCs," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 10, pp. 1335 –1345, 2008.

[69] "How SKBs work." `http://vger.kernel.org/~davem/skb_data.html`. [Online; last accessed 05.03.2010].

[70] M. Meitinger, R. Ohlendorf, T. Wild, and A. Herkersdorf, "A Programmable Stream Processing Engine for Packet Manipulation in Network Processors," ISVLSI 2007, 2007.

[71] R. Ohlendorf, T. Wild, M. Meitinger, H. Rauchfuss, and A. Herkersdorf, "Performance Evaluation of RISC-based SoC Platforms in Network Processing Applications," in *Embedded Computer Systems: Architectures, Modeling and Simulation, 2006. IC-SAMOS 2006. International Conference on*, pp. 152–159, July 2006.

[72] Open SystemC Initiative (OSCI), "SystemC Homepage." `http://www.systemc.org`.

[73] T. Wild, A. Herkersdorf, and G.-Y. Lee, "TAPES - Trace-based architecture performance evaluation with SystemC," *Design Automation for Embedded Systems*, vol. 10, pp. 157–179, September 2005.

[74] T. Wild, A. Herkersdorf, and R. Ohlendorf, "Performance Evaluation for System-on-Chip Architectures using Trace-based Transaction Level Simulation," in *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 1, pp. 1–6, March 2006.

[75] "TCPDUMP/LIBPCAP Homepage." `http://www.tcpdump.org/`.

[76] R. Ohlendorf, M. Meitinger, T. Wild, and A. Herkersdorf, "An Application-Aware Load Balancing Strategy for Network Processors.," in *HiPEAC* (Y. N. Patt, P. Fo-

glia, E. Duesterwald, P. Faraboschi, and X. Martorell, eds.), vol. 5952 of *Lecture Notes in Computer Science*, pp. 156–170, Springer, 2010.

[77] Xilinx, "OPB Interrupt Controller Data Sheet." `http://www.xilinx.com/support/documentation/ip_documentation/opb_intc.pdf`, Jan. 2005.

[78] A. Bauer, "Implementation of an Interrupt Controller for Multiprocessor Systems," Bachelor's Thesis, Technische Universität München, Germany, Januar 2007.

[79] CAIDA OC48 Trace Project, "CAIDA OC48 Traces 2002-08-14 (collection)." `http://imdc.datcat.org/collection/1-0016-F=CAIDA+OC48+Traces+2002-08-14`, files used: 20020814-090000-1-anon.pcap, 20020814-091500-1-anon.pcap, 20020814-093000-1-anon.pcap, 20020814-094500-1-anon.pcap.

[80] C. Shannon, E. Aben, K. C. Claffy, and D. E. Andersen, "CAIDA Anonymized 2008 Internet Traces Dataset (collection)." `http://imdc.datcat.org/collection/1-06BX-2=CAIDA+Anonymized+2008+Internet+Traces+Dataset`, 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, eq-chic.dirA.20080717-132000.UTC.anon.pcap.

[81] M. Meitinger, R. Ohlendorf, T. Wild, and A. Herkersdorf, "A Hardware Packet Re-Sequencer Unit for Network Processors," in *Architecture of Computing Systems – ARCS 2008*, no. 4934 in Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 85–97, Springer, 2008. Dresden, 25.02.2008-28.02.2008.

[82] S. Traboulsi, "Design and Implementation of a Network Processor Path Control," Master's thesis, Technische Universität München, Munich, Germany, October 2007.

[83] S. Traboulsi, M. Meitinger, R. Ohlendorf, and A. Herkersdorf, "An Efficient Hardware Architecture for Packet Re-sequencing in Network Processors MPSoCs," DSD 2009, (Patras, Greece), 2009.

[84] M. Meitinger, R. Ohlendorf, T. Wild, and A. Herkersdorf, "FlexPath NP - A Network Processor Architecture with Flexible Processing Paths," SoC 2008, (Tampere, Finland), 2008.

[85] Xilinx, "Xilinx ML410 Documentation and Tutorials." `http://www.xilinx.com/products/boards/ml410/index.html`. [Online; accessed 18.05.2010].

[86] E. Scheiber, "Development of a network stack for the NP3 platform," Master's thesis, Technische Universität München, Munich, Germany, May 2007.

[87] S. Bradner and J. McQuaid, "RFC 2544: Benchmarking Methodology for Network Interconnect Devices." `http://tools.ietf.org/html/rfc2544`, Mar. 1999.

[88] Agilent, "Mixed Packet Size Throughput." `http://advanced.comms.agilent.com/n2x/docs/insight/2001-08/TestingTips/1MxdPktSzThroughput.pdf`, 2001.