Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Departement Informationstechnologie & Elektrotechnik
Professur für Technische Informatik
Professor Dr. Lothar Thiele

Andri Kofmehl
Javier Coedo

Evaluation von Arbitrierungsschemen in Netzwerkprozessoren

Diplomarbeit DA-2003.17
Wintersemester 2002/2003

Betreuer:
Simon Künzli
Samarjit Chakraborty

Verantwortlicher:
Prof. Dr. Lothar Thiele

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Kurzfassung

In den Netzknoten des Internets werden zur Verarbeitung der Paketströme speziell konzipierte Prozessoren eingesetzt. Diese Netzwerkprozessoren sind einerseits auf typische Anwendungen zugeschnitten, bieten aber gleichzeitig ein hohes Mass an Flexibilität. Sie werden aus mehreren Prozessor-, Speicher- und Kommunikationseinheiten kombiniert und als Gesamtsystem auf einem Chip integriert. Den prozessorinternen Datenaustausch unter den Verarbeitungseinheiten kann ein schneller Bus übernehmen. Bei hohem Datenaufkommen wird dieser stark ausgelastet, womit das Scheduling auf Busebene, die sogenannte Arbitrierung, an Bedeutung gewinnt.

In dieser Arbeit wird das Problem der Arbitrierung anhand einer Beispiel-Prozessorarchitektur analysiert und mit Hilfe von Simulationen genauer untersucht. Es zeigt sich, dass die Wahl des Arbiters entscheidend an der resultierenden Prozessorleistung beteiligt ist und überdies auch bei der Einhaltung von Quality-of-Service-Vorgaben eine wichtige Rolle spielt. In diesem Sinne wurden Konzepte ausgearbeitet, welche Paketscheduling und Busarbitrierung kombinieren und als Instrument für den Entwurf und die Konfigurierung moderner Netzwerkprozessoren dienen können. Die erwarteten Vorteile wurden durch Simulationen bestätigt.

Inhaltsverzeichnis

Kurzfassung **3**

1 Einleitung **1**
1.1 Netzwerkprozessoren 3
1.1.1 Geschichtliches 3
1.1.2 Aufgaben eines Netzwerkprozessors 4
1.1.3 Moderne Netzwerkprozessoren 5
1.2 Projektüberblick 5

2 Netzwerkprozessor-Simulator **7**
2.1 Referenz-Architektur 7
2.2 Modellabstraktion 8
2.3 Komponenten-Modell 9
2.3.1 Busarchitektur 9
2.3.2 Ethernet-Block (EMAC) 10
2.3.3 PLB-OPB-Brücke 11
2.3.4 Speicher 13
2.3.5 Prozessor-Block 14
2.3.6 Zusammenfassung 17
2.4 Erweiterte Architektur 19
2.4.1 Multiprozessoreinheit 20
2.4.2 Direct Memory Access Controller 22
2.5 Matlab-Simulation 23

3 Bus-Arbitrierung **25**
3.1 Definition des Arbiters 25
3.2 Konzepte zur Arbitrierung 26
3.2.1 Beispiele von Arbitern 27
3.2.2 Prioritäts- oder Bandbreiten-orientierte Algorithmen . 29
3.2.3 Mehrstufige Arbitrierung 30
3.3 Arbitrierung im Netzwerkprozessor 30
3.3.1 Ausgangslage 30
3.3.2 PLB-Verkehr 31

3.3.3 Systemkapazität und -dimensionierung 33
3.3.4 Warteschlangen und Scheduler 36
3.3.5 Bedeutung von Arbitrierung und Scheduling 37
3.4 Szenario I: Minimierung der Paketverzögerung 42
3.4.1 Ausgangslage 42
3.4.2 Scheduling 42
3.4.3 Arbitrierung 43
3.4.4 Simulationsergebnisse 44
3.4.5 Schlussfolgerungen 47
3.5 Szenario II: Best-effort-Optimierung bei gleichzeitigem Real-time-Verkehr 52
3.5.1 Ausgangslage 52
3.5.2 Scheduling 52
3.5.3 Arbitrierung 55
3.5.4 Simulationsergebnisse 56
3.5.5 Schlussfolgerungen 58

4 Software-Dokumentation **61**
4.1 SystemC-Simulator 61
4.2 Multiprozessormodell 62
4.2.1 Prozessor-Anfragen 66
4.3 Direct Memory Access Controller (DMA) 67
4.3.1 PLB-OPB-Brücke 68
4.3.2 DMA-Modul 71
4.4 Arbitrierung und Scheduling 74
4.4.1 Prozessor-Scheduling 74
4.4.2 Input- und Output-Scheduling 74
4.4.3 PLB-Arbitrierung 76
4.4.4 Verkehrsklassen 77

5 Zusammenfassung und Ausblick **79**

Literaturverzeichnis **81**

Kapitel 1

Einleitung

Das Internet bietet heute eine Vielzahl von Kommunikationsdiensten an. In den letzten Jahren wurden neue Anwendungen eingeführt, wie zum Beispiel E-Commerce, Video-on-Demand oder Voice-over-IP. Solche Anwendungen haben hohe Dienstgüteanforderungen an das Netzwerk. Die Dienstgüte (Quality-of-Service, QoS) eines Netzes wird normalerweise durch unterschiedliche technische Parameter wie Durchsatz, Verzögerung, Verzögerungsschwankungen (Jitter) und Paketverlustrate charakterisiert [1]. Obwohl sich das Internet im ununterbrochenen Wachstum hinsichtlich der Anzahl Hostrechner, der Vielfalt der Anwendungen und der Kapazität der Verbindungen befindet, bleibt seine Architektur gegenüber seinen frühen Tagen weitgehend unverändert. Das Internet funktioniert als Netzwerk, in welchem Daten in Form einzelner Pakete übertragen werden. Eine maximale Verzögerung der Pakete kann nicht garantiert werden und im Falle von Überlastung können Pakete sogar verloren gehen. Man spricht von einem Best-effort-Dienst. Diese Unzuverlässigkeit des Internet steht nicht im Einklang mit neuen Diensten, welche Qualitätsgarantien benötigen. Die Übertragung von Video- oder Audiodatenströmen beispielsweise kann unbefriedigend ausfallen.

Mit der schnellen Wandlung des Internet in eine kommerziell genutzte Infrastruktur hat sich die Nachfrage nach Mechanismen, welche Qualitätsgarantien bieten, erhöht. Es gibt verschiedene Ansätze mit dem Ziel, QoS im Internet zu unterstützen. Die Internet Engineering Task Force (IETF) [2] hat zu diesem Zweck zwei Modelle entwickelt: Integrated Services (IntServ) [3] und Differentiated Services (DiffServ) [4]. Beide Ansätze basieren auf der Annahme, dass die Bandbreiten-Nachfrage in einer oder mehreren Lokalitäten des Netzes das Angebot übersteigt und somit irgendeine Art von Ressourcen-Reservierung zur Sicherung einer bestimmten Dienstqualität durchgeführt werden muss. Es gibt Stimmen, welche die Notwendigkeit solcher Mechanismen anzweifeln. So wird argumentiert, dass dank moderner Technologien,

wie Wellenlängen-Multiplex (DWDM, Dense Wavelength Division Multiplexing), Bandbreite in Zukunft so reichlich und preiswert vorhanden sein werde, dass die benötigte Dienstgüte automatisch gegeben sei [13]. In der Tat bietet ein einziges Glasfaserkabel eine Bandbreite von 25 bis 30 THz an und ermöglicht damit Übertragungsraten von einigen Terabits (10^{12} Bits) pro Sekunde. Theoretisch könnten damit alle Telefongespräche, die weltweit zur selben Zeit geführt werden, problemlos durch ein einziges Glasfaserkabel übertragen werden [5]. Es bleibt aber zu bedenken, dass egal wie viel Bandbreite das Internet jemals zur Verfügung stellt, diese von neuen Anwendungen verwendet werden wird. Folglich werden Mechanismen zur Sicherung der QoS auch in Zukunft erforderlich sein.

Um Dienstgüte im Internet zu garantieren, müssen Betriebsmittel des Netzwerks reserviert werden können. Betriebsmittel sind Netzwerkressourcen wie Bandbreite, Pufferspeicher oder Rechenleistung. Sobald im Netz eine momentane Überlast besteht, werden die Ressourcen gemäss Reservierung den entsprechenden Paketströmen zugewiesen, womit für leistungskritische Anwendungen die verlangte Dienstgüte aufrechterhalten werden kann. Es sind dazu effiziente Scheduling-Mechanismen erforderlich, welche die Abarbeitungsreihenfolge der Pakete in den Netzwerkknoten regeln, und die Betriebsmittel unter den Paketströmen aufteilen.

Scheduling-Algorithmen werden allgemein dazu verwendet, eine Ressource gemäss einer definierten Zielfunktion zu verwalten. Sie treten in verschiedenen Schichten des OSI-Referenzmodells auf. In der Literatur wird das Scheduling-Problem zumeist auf der Netzwerkschicht betrachtet, d.h. es wird untersucht, wie Paketswitches die Verbindungen verwalten sollen (Link Scheduling). Im Sachen Link Scheduling wurde bereits sehr viel Forschung betrieben [7, 8, 9]. So sind am Institut für Technische Informatik der ETH Zürich mehrere Scheduling-Algorithmen entwickelt und publiziert worden, welche eine theoretisch garantierte Performanz aufweisen, wie beispielsweise [6].

Die vorliegende Arbeit beschäftigt sich mit Scheduling-Verfahren für Kommunikationsbusse in Netzwerkprozessoren. Im Falle von Bussen spricht man in der Regel nicht mehr von Scheduling, sondern von *Arbitrierung*. In der Industrie herrscht die Ansicht vor, dass die Arbitrierung nur eine untergeordnete Rolle spielt. Ziel dieser Arbeit ist es den Einfluss der Arbitrierung auf die Leistung des Netzwerkprozessors aufzuzeigen sowie Lösungen zu erarbeiten, welche die Einhaltung von QoS-Vorgaben unterstützen.

1.1 Netzwerkprozessoren

Ein Netzwerkprozessor ist ein ASIP (Application Specific Instruction Set Processor), welcher auf die Klasse von Netzwerkanwendungen zugeschnitten ist. Netzwerkprozessoren können aus mehreren Rechen-, Kommunikations- und Speichereinheiten auf einem einzigen Chip bestehen. Dies wird als System-on-a-Chip (SoC) bezeichnet. Recheneinheiten können Mikroprozessor-Blöcke, Micro-Engines oder dedizierte Hardware-Komponenten für spezielle Paketverarbeitungsfunktionen sein. Als Kommunikationseinheiten werden Busse oder Crossbars eingesetzt.

1.1.1 Geschichtliches

Paketswitches der ersten Generation bestanden aus einem PC mit mehreren Netzwerkkarten. Die Pakete wurden über den Systembus in den Hauptspeicher geladen und danach von einem General Purpose Prozessor (GPP) bearbeitet. Als nach und nach Netzwerkgeschwindigkeit und -dichte zunahmen, konnte diese einfache Architektur den steigenden Anforderungen nicht mehr gerecht werden. Leistungsbegrenzend konnten alle Komponenten sein: General Purpose Prozessoren, die Pakete nicht genug schnell bearbeiteten, Hauptspeicher mit zu langen Zugriffszeiten oder Systembusse mit mangelnder Kapazität.

Um diesen Problemen entgegenzuwirken, wurden in der nächsten Generation von Paketswitches Hochgeschwindigkeitsbusse eingesetzt und die Funktionalität wurde in Hardware, auf Basis von ASICs (Application Specific Integrated Circuits) oder FPGAs (Field Programmable Gate Arrays) implementiert. Um die Leistung noch weiter zu erhöhen, wurden gewisse Aufgaben, die in den ersten Paketswitches zentral vom GPP erledigt worden waren, auf die Netzwerkkarten übertragen. Heute werden intelligente Netzwerkkarten eingesetzt, welche über Speicher verfügen und zusätzliche Aufgaben wie das Routing übernehmen. In einem weiteren Schritt wurde der gemeinsame Bus durch ein Durchschaltnetzwerk (Switch Fabric) ersetzt [10]. Diese Architekturen sind auf Leistungssteigerung hin entwickelt und optimiert worden. Sie haben aber den entscheidenden Nachteil, dass sie sehr unflexibel sind. Wie bereits erwähnt, sehen wir heute im Internet eine Entwicklung in Richtung bandbreitenintensive und verzögerungssensitive Anwendungen. Damit das Internet diese neuen Aufgaben gut erfüllen kann, müssen unter anderem neue Protokolle unterstützt werden.

Während also auf der einen Seite möglichst verzögerungsarme Übertragung und möglichst grosser Durchsatz gefordert sind, wird auf der anderen Seite die Flexibilität und die Möglichkeit schneller Anpassungen immer wich-

tiger. Performanz und die Flexibilität sind gegenläufige Entwurfsziele, sie bilden einen sogenannten Trade-Off. Das bedeutet, dass nie beide zugleich maximiert werden können. Im Bemühen, einen Kompromiss zwischen diesen beiden Entwurfsrichtungen zu finden, wurden die Netzwerkprozessoren geboren. Netzwerkprozessoren vereinigen Vorteile beider oben erwähnten Ansätze, nämlich einerseits die Programmierbarkeit von General Purpose Prozessoren und anderseits die Schnelligkeit von ASICs.

1.1.2 Aufgaben eines Netzwerkprozessors

Da Netzwerkprozessoren speziell für Netzwerke entwickelt sind, liegt es nahe sich zuerst der Aufgaben in einem heterogenen Netzwerk, das aus vielen Teilnehmern besteht, bewusst zu werden. Gemäss [14] lassen sich alle Netzwerkanwendungen in folgende Kernprozesse aufteilen:

- **Pattern Matching.** Unter Pattern Matching versteht man die Aufgabe, bestimmte Bitmuster in Datenpaketen zu finden. Man vergleicht bei diesem Prozess immer das Datenpaket gegen einen regulären Ausdruck. Das Ergebnis einer solchen Operation ist ein Boolescher Wert, der angibt, ob der gesuchte Ausdruck im Datenpaket enthalten ist.

- **Lookup.** Als Lookup bezeichnet man die Aufgabe, in einer Datenbank nach einem bestimmten Schlüssel zu suchen. Diese Funktion wird oft in Verbindung mit Pattern Matching benötigt. Eine Beispielanwendung ist die Suche nach dem richtigen Ausgangsport eines Datenpakets in einer Routing-Tabelle.

- **Computation.** Als Computation bezeichnet man die Aufgabe, für ein Datenpaket einen bestimmten Wert zu berechnen. Dafür gibt es verschiedenste Möglichkeiten. Zum Beispiel muss für die meisten Protokolle eine Prüfsumme bestimmt werden. Diese Werte müssen immer dann neu berechnet werden, wenn sich Werte im Header von Datenpaketen ändern.

- **Data Manipulation.** Unter Data Manipulation versteht man alle möglichen Aufgaben, die in einem Datenpaket Werte verändern. Dies sind zum Beispiel beim Internet-Protokoll die Dekrementierung des Time-to-Live-Felds, das Einfügen von Sequenznummern bei der Fragmentierung von Paketen bzw. das Zusammenfügen dieser Pakete.

- **Queue Management.** Unter Queue Management wird das Verwalten von Warteschlangen verstanden. Dies beinhaltet das Scheduling und die Speicherung von ankommenden und abgehenden Paketen. Dieser Prozess ist auch zuständig für das Einhalten von Dropping- und Traffic-Shaping-Regeln in QoS-bezogenen Applikationen.

- **Control Processing.** Unter Control Processing versteht man verschiedene Aufgaben, welche nicht mit Linkgeschwindigkeit durchgeführt werden müssen. Beispiele dafür sind das Aktualisieren von Tabellen und Statistiken sowie die Behandlung von Ausnahmen.

1.1.3 Moderne Netzwerkprozessoren

Netzwerkprozessoren unterliegen wie alle Systeme gewissen Anforderungen. Um Leistungskriterien zu erfüllen, werden Hochgeschwindigkeitsbusse oder spezielle Kommunikationseinheiten, z.B. Crossbars, eingesetzt, welche die unterschiedlichen Komponenten miteinander verbinden. Ausserdem werden mehrere Recheneinheiten bereitgestellt, die ein Paket parallel oder in einem Pipeline-Modus verarbeiten. Dadurch lässt sich die Leistung eines Netzwerkprozessors erheblich steigern. Zum Beipspiel ist der IBM PowerNP NP2G ein Netzwerkprozessor, der 12 Pikoprozessoren, einen eingebetteten PowerPC und mehrere Hardware-Beschleuniger vereinigt [15]. Um ein weiteres Beispiel zu nennen, sei hier Intels Netzwerkprozessor IXP2850 aufgeführt. Dieser besteht aus 16 Micro-Engines, einem eingebetteten RISC-Prozessor und aus mehreren Hardware-Beschleunigern [16].

Flexibilität ist ebenfalls ein wichtiges Kriterium. Dieses wird durch die Programmierbarkeit von Prozessoren erfüllt. Es ist von grosser Bedeutung, Netzwerkknoten möglichst rasch und kostengünstig an neue Dienste und Protokolle anpassen zu können.

Das System-on-a-Chip-Design bringt weitere Vorteile mit sich, wie die eines geringen Gewichts, eines kleinen Volumens und einer geringen Leistungsaufnahme. Auch die Zuverlässigkeit eines SoC ist sehr hoch. Da es keine schnell getakteten externen Busse gibt, sind weniger elektromagnetische Emissionen zu erwarten. Fortschritte in der Mikroelektronik ermöglichen es heute, mehrere Millionen Transistoren auf einem einzigen Chip zu integrieren. Um diese komplexen Systeme realisieren zu können, hat sich der Entwurf von ICs (Integrated Circuits) gewandelt. Das heutige Vorgehen besteht darin, bereits vorhandene Blöcke auf Systemebene mit selbst entworfenen Blöcken zu einem neuen Gesamtsystem zu kombinieren. Man spricht von einer blockbasierten (core-based) Entwurfsmethode, die hohe Produktivität durch Wiederverwendung bereits vorhandener Blöcke erreicht [17].

1.2 Projektüberblick

Diese Arbeit wurde zusammen mit dem IBM-Forschungslabor in Rüschlikon [18] ausgeführt. Für den Entwurf neuer Generationen von Netzwerkprozes-

soren hat IBM SystemC-Modelle [19] einzelner SoC-Blöcke wie Prozessoren, Speichercontroller und Busse erstellt. SystemC ist eine C++-Bibliothek, welche die nötigen Konstrukte zur Verfügung stellt, um Software- und Hardware-Systeme taktgenau zu modellieren. In zwei kürzlich erstellten Arbeiten [20, 21] werden diese Modelle kombiniert und ein Simulator für den Datenpfad eines einfachen Netzwerkprozessors geschaffen.

Die bestehende Simulationsumgebung wurde mit einem Multiprozessorblock zur Modellierung einer parallelen Paketverarbeitung sowie einer DMA-Einheit als zusätzliche Verkehrsquelle und -senke am Prozessor-Speicher-Bus ergänzt. Das ursprüngliche Vorhaben, die Arbitrierung im Netzwerkprozessor mit Hilfe dieses Simulators zu untersuchen, wurde jedoch aufgrund von bereits in der ursprünglichen Programmversion enthaltenen Fehlern verunmöglicht. Da diese nicht innert nützlicher Frist behoben werden konnten, kam für die Analyse der Arbitrierung ein selbstgeschriebener Matlab-Simulator zum Einsatz, der genau auf die entsprechende Problematik zugeschnitten ist.

Kapitel 2

Netzwerkprozessor-Simulator

In diesem Kapitel wird der uns zur Verfügung gestellte Simulator eines Netzwerkprozessors beschrieben. Zuerst wird ein Überlick der modellierten Architektur gegeben und danach werden die einzelnen Komponenten erklärt. In einem nächsten Abschnitt sind die am Modell getätigten Erweiterungen beschrieben.

2.1 Referenz-Architektur

Figur 2.1 zeigt eine einfache Netzwerkprozessor-Architektur, die mit Blöcken aus existierenden Core-Bibliotheken [22, 23] aufgebaut ist. Sie besteht aus einem Prozessorblock, aus mehreren Speichern bzw. Speichercontrollern sowie Ethernet-Blöcken. Kernstück dieses Netzwerkprozessors ist ein Bus-System, das auf der CoreConnect-Architektur [22] basiert. Diese schliesst einen schnellen prozessornahen Bus (Processor Local Bus, PLB) und einen peripheren Bus (On-Chip Peripheral Bus, OPB) ein, welche mittels einer Brücke miteinander verbunden sind.

Der Datenpfad sieht folgendermassen aus: Ein eintreffendes Paket wird von einem Ethernet Core gelesen. Danach wird der Brücke signalisiert, dass ein Paket zur Übertragung bereitsteht. Die Brücke kümmert sich nun darum, das Paket in kleinen Dateneinheiten über den OPB zu lesen und via PLB in den Speicher zu schreiben. In einem nächsten Schritt wird dem Prozessor signalisiert, dass ein Paket zur Bearbeitung bereitsteht. Der Prozessor liest je nach Applikation das ganze Paket oder bloss seinen Header in seinen Cache ein und bearbeitet es. Danach wird es wieder in den Speicher zurückgeschrieben. Zum Abschluss der Verarbeitung ist wiederum die Brücke dafür zuständig, das Paket vom Speicher in einen Ethernet-Block zu schreiben.

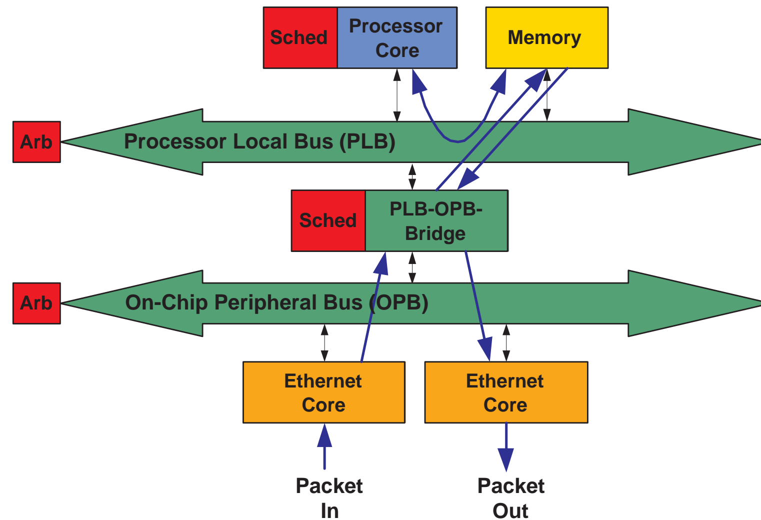

Abbildung 2.1: Einfache Netzwerkprozessor-Architektur.

Arbitrierung ist an den Stellen möglich, die in Abbildung 2.1 mit roter Farbe gekennzeichnet sind. In jedem Bus ist ein Arbiter integriert, der jeweils die entsprechenden Busanfragen verwaltet. Ebenfalls mit roter Farbe markiert, sind die Scheduling-Einheiten. Die Brücke verfügt über zwei Scheduler, die das Input/Output-Scheduling durchführen. Der Scheduler auf Prozessor-Ebene entscheidet, welche Pakete verarbeitet werden.

2.2 Modellabstraktion

Unter einem Modell versteht man die formale Beschreibung eines Systems. Dabei werden von einem zu modellierenden Objekt nur bestimmte Eigenschaften berücksichtigt und unerwünschte Details weggelassen – dies in einer Weise, dass die Aussagen über bestimmte Eigenschaften des Modells sich möglichst gut mit den entsprechenden Eigenschaften des realen Systems decken. Dabei möchte man erreichen, dass die Analyse des Modells wenn möglich einfacher ist als die entsprechenden Untersuchungen am realen System. Diesen Vorgang nennt man Abstraktion.

Der uns zur Verfügung stehende Simulator modelliert den Datenpfad des im vorherigen Abschnitt beschriebenen Netzwerkprozessors. Das Modell ist durch eine Vielzahl an Konfigurationsmöglichkeiten sehr flexibel und ermöglicht eine Entwurfsraumexploration, dadurch dass die Busarchitektur eine standardisierte Schnittstelle für unterschiedliche Einheiten, wie Prozessoren, Speichercontroller und Brücken bereitstellt. Diese Eigenschaft steht auch im

Einklang mit der Philosophie des System-on-a-Chip-Entwurfs, nämlich der Kombination bereits vorhandener Blöcke mit selbst eintworfenen Blöcken zu einem neuen Gesamtsystem.

Das Modell entspricht einer taktgenauen Repräsentation des realen Systems und erlaubt es, verschiedene Architekturentwürfe in Bezug auf ihre Leistung zu analysieren. Die Leistung des Systems wird in diesem Fall durch verschiedene Parameter, wie Durchsatz (Pakete pro Zeiteinheit), Verzögerung der einzelnen Pakete und Speicherbedarf charakterisiert. Über die Busse werden jedoch keine wirklichen Daten übertragen; ein Bus entspricht eher einem Verzögerungsglied, das je nach Umfang der zu übertragenden Daten eine Verzögerung generiert, die der realen Bus-Zugriffszeit und -Besetzungszeit entspricht. Pakete werden in der Simulation durch eine Datenstruktur repräsentiert und stehen nicht in Verbindung mit irgendwelchen Speicheradressen. Auf diese Datenstruktur wird mit Hilfe des C++-Zeigerkonzeptes zugegriffen. Für jeden Zugriff werden jedoch die entsprechenden Busanfragen generiert, die auch im realen System erforderlich wären. Dadurch wird der modellierte Paketfluss entsprechend verzögert.

2.3 Komponenten-Modell

In diesem Abschnitt wird ein Überblick der verschiedenen SoC-Komponenten gegeben. Es wird auf Eigenschaften der realen Blöcke eingegangen sowie auf deren Modellierung. Für genauere Informationen verweisen wir auf die Arbeiten [20, 21].

2.3.1 Busarchitektur

Die Busarchitektur besteht aus zwei Datenbussen, dem PLB und OPB. Zusätzlich ist ein Kontrollbus vorhanden, der in Figur 2.1 nicht eingezeichnet ist und auch nicht modelliert wird, weil er für die vorangehenden Arbeiten wie auch für diese nicht relevant ist. Der PLB ist ein Hochleistungsbus, der eine Schnittstelle für Einheiten bietet, die auf eine hohe Bandbreite und auf eine möglichst kleine Latenz angewiesen sind. Die Kommunikation auf dem PLB ist vollsynchron und es stehen getrennte Leitungen für Lese-, Schreib- und Adresszugriffe zur Verfügung. Anfragen können gepipelined werden. Masters und Slaves teilen sich den Bus, während ein zentraler Arbiter die Zugriffe auf diesen regelt (siehe Abbildung 2.2).

Der OPB ist ein zweiter Bus, der eine Schnittstelle für Eingabe/Ausgabe-Einheiten bietet. In diesem Fall handelt es sich um Ethernet-Blöcke. Der OPB hat im Gegensatz zum PLB keine getrennten Leitungen für Schreib-

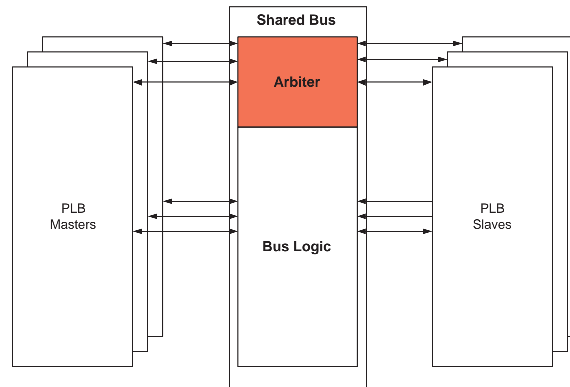

Abbildung 2.2: Gemeinsamer Bus.

und Lesezugriffe.

Im Modell lassen sich beide Busse parametrisieren. Die Frequenz des PLB ist frei wählbar, muss aber einem Vielfachen der OPB Frequenz entsprechen. Die zugehörigen Parameter im Modell sind:

- *PLB_FREQUENCY (plb_params.h)*: PLB-Busfrequenz.
- *OPB_PLB_CLK_RATIO (opb_params.h)*: Verhältnis der PLB- zur OPB-Busfrequenz (≥ 1).
- *NUM_MASTERS (plb_params.h)*: Anzahl PLB Masters.
- *NUM_SLAVES (plb_params.h)*: Anzahl PLB Slaves.

Des weiteren lässt sich die Breite der Daten- und Adressleitungen konfigurieren:

- *PLB_DATA_WIDTH (plb_params.h)*: Breite der Datenleitung in bit.
- *PLB_ADDR_WIDTH (plb_params.h)*: Breite der Adressleitung in bit.
- *OPB_DATA_WIDTH (opb_params.h)*: Breite der Datenleitung in bit.
- *OPB_ADDR_WIDTH (opb_params.h)*: Breite der Adressleitung in bit.

2.3.2 Ethernet-Block (EMAC)

Der Ethernet-Block entspricht einer allgemeinen Implementierung des Medium-Access-Control-Protokolls. Zwei FIFOs puffern jeweils Paketdaten für

den Empfangs- und Sendepfad. Der EMAC hat zwei OPB-Schnittstellen, eine für den Zugang auf Konfigurations- und Statusregister, und eine andere bidirektionale Schnittstelle, um Daten zu senden und zu empfangen. Ein Ethernet-Block kann als Eingangs- und Ausgangsport betrachtet werden.

In der Simulation wird aus einer Datei eine reale Verkehrs-Trace gelesen, die auf eine Sequenz von Paketlängen reduziert ist, da weitere Header-Informationen nicht von Bedeutung sind. Die Verkehrs-Trace stammt aus der NLANR Datenbank [24]. Die wichtigsten Parameter zur EMAC-Konfiguration sind die folgenden:

- *INPUT_RATE_PER_EMAC (emac_params.h):* Entspricht der Linkgeschwindigkeit, mit der Pakete gelesen werden.

- *IFG (emac_params.h):* (Interframe Gap) Entspricht der Zwischenankunftszeit der Pakete.

- *RX_CHANNEL_NBR (Bridge_params.h):* Entspricht der Anzahl Rx-EMACs (Receive EMACs). Ein RxEMAC entspricht repräsentiert den Empfangskanal (Rx-Kanal) eines EMACs.

- *TX_CHANNEL_NBR (Bridge_params.h):* Entspricht der Anzahl Tx-EMACs (Transmit EMACs). Ein TxEMAC repräsentiert den Sendekanal (Tx-Kanal) eines EMACs.

- *PKT_NUMBER_PER_EMAC (emac_params.h):* Entspricht der Anzahl Pakete, die vom EMAC während einer Simulation gelesen werden.

2.3.3 PLB-OPB-Brücke

Die Brücke dient als Verknüpfungsglied zwischen den zwei Bussen und ermöglicht eine direkte Kommunikation zwischen peripheren Einheiten und PLB Masters, indem Brücken- und DMA-Funktionalität geboten werden. Ein EMAC kommuniziert mit dem PLB über die PLB-OPB-Brücke. Aus Sicht des OPBs ist ein EMAC ein Slave und hat somit keine Möglichkeit, einen Transfer zu initiieren. Aus diesem Grund gibt es eine weitere Schnittstelle zwischen EMAC und Brücke, über die sogenannte Sideband-Signale übertragen werden, welche die Brücke für eine Paketübertragung triggern. Die PLB-OPB-Brücke überträgt ein Paket vom EMAC in den Speicher in Form von parametrisierbaren Dateneinheiten, sogenannten Bursts. Die entsprechenden Parameter im Modell sind:

- *BRIDGE_RX_BURST_SIZE (Bridge_params.h).* Burst im Empfangspfad.

- *BRIDGE_TX_BURST_SIZE (Bridge_params.h).* Burst im Sendepfad.

Eine weitere Eigenschaft der Brücke ist, dass diese über zwei Scheduler verfügt. Diese realisieren das Input/Output-Scheduling. Es ist zu beachten, dass für jede zu übertragende Dateneinheit eines Pakets auf OPB-, PLB- und ebenfalls auf Brücken-Ebene arbitriert wird.

Die Brücke kann gleichzeitig einen Empfangs- und einen Sendekanal bedienen (Rx- und Tx-Kanal). Im Empfangskanal wird ein Paket vom Eingangsport in den Speicher geschrieben. Es sind dafür OPB-Lese- und PLB-Schreibzugriffe nötig. Im Sendekanal wird das Paket vom Speicher in einen Ausgangsport geschrieben. Es werden PLB-Lese- und OPB-Schreibzugriffe getätigt. Folgende Kombinationen von gleichzeitigen Busbesetzungen sind aus Sicht der Brücke möglich:

Rx-Kanal		Tx-Kanal
PLB schreiben	$\longleftrightarrow$	PLB lesen
PLB schreiben	$\longleftrightarrow$	OPB schreiben
OPB lesen	$\longleftrightarrow$	PLB lesen

Tabelle 2.1: Gleichzeitig mögliche Busbesetzungen.

2.3.3.1 Pufferdeskriptoren-Verwaltung

Die Brücke ist dafür zuständig, für jedes Paket die nötigen Pufferdeskriptoren zu verwalten. Ein Pufferdeskriptor ist eine Datenstruktur, die unter anderem eine Speicheradresse und ein Statusfeld beinhaltet. Deskriptoren selber befinden sich in einem dedizierten On-Chip-Speicher und müssen über den PLB gelesen werden, bevor der entsprechende Pufferspeicher benutzt werden kann. Die Deskriptoren stellen eine gemeinsame Ressource dar, die von mehreren PLB Masters (Brücke, Prozessor) benutzt wird. Folglich müssen Zugriffs- und Synchronisationsregeln eingehalten werden, um mögliche Konflikte zu vermeiden. Der Prozessor sollte zum Beispiel nicht einen Puffer überschreiben, der noch nicht von der Brücke geschrieben worden ist. Zur Vermeidung solcher Fälle dient das Statusfeld, das nach jedem Lesen eines Deskriptors und der Benutzung des entsprechenden Speicherbereichs aktualisiert werden muss. Bei Beendigung einer Paketübertragung muss die Brücke zusätzlich den benutzten Pufferspeicher wieder freigeben.

Im Modell werden zwei parametrisierbare Puffergrössen unterschieden. Mit Hilfe der Parameter

- *BRIDGE_SBD_DATA_SIZE (Bridge_params.h)*: Speichergrösse der kleinen Puffer.
- *BRIDGE_LBD_DATA_SIZE (Bridge_params.h)*: Speichergrösse der grossen Puffer.

sind diese konfigurierbar. Im Modell werden Puffergrössen von 64 und 1472 Bytes unterschieden. Je nach Paketgrösse sind demzufolge ungleich viele Pufferdeskriptoren nötig. Der Header eines Pakets, dessen Grösse im Modell stets 64 Byte beträgt[1], wird in einen kleinen Puffer geschrieben. Für die Payload des Pakets, werden je nach Grösse einer oder mehrere grosse Puffer benutzt.

Es ist zu erwähnen, dass die Brücke auf zwei verschiedene Arten betrieben werden kann. Der Unterschied liegt in der Art, wie die Pufferdeskriptoren verwaltet werden. In einem Fall wird ein Pufferdeskriptor nur bei Bedarf aus dem Speicher gelesen. Im anderen Fall hat die Brücke einen zusätzlichen Cache-Speicher, in welchem mehrere Pufferdeskriptoren für eine spätere Verwendung zwischengespeichert werden können. In dieser Arbeit wurde jeweils der erste Fall betrachtet.

2.3.4 Speicher

Auf Speichereinheiten wird über den PLB zugegriffen. Sie können entweder On-Chip oder Off-Chip sein. Im Modell sind zwei Speicher vorhanden, ein schneller Speicher (z.B. ein SRAM) für die Pufferdeskriptoren und ein langsamerer für die Paketdaten (z.B. ein DRAM).

Das Speichermodell erlaubt es, Latenzen für erste und sequentielle Schreib- und Lesezugriffe zu parametrisieren. Diese Parameter werden bei der Initialisierung der PLB-Slave-Module eingestellt. Dadurch lassen sich auf einer hohen Abstraktionsebene unterschiedliche Speichereinheiten modellieren. Wie bereits im vorherigen Abschnitt beschrieben, werden zwei Puffergrössen unterschieden. Die Anzahl Puffer wird durch die Parameter

- *BRIDGE_SBD_NBR (Bridge_params.h)*: Entspricht der Anzahl kleiner Puffer.
- *BRIDGE_LBD_NBR (Bridge_params.h)*: Entspricht der Anzahl grosser Puffer.

eingestellt.

[1] Die übliche Grösse eines IPv4-Headers beträgt 20 Bytes; maximal sind 60 Bytes möglich, dies falls der optionale Header-Bereich voll ausgeschöpft wird.

2.3.5 Prozessor-Block

Für die Recheneinheiten stehen zwei unterschiedliche Modelle zur Verfügung. In einem einfachen Modell wird der Prozessor als ein reines Verzögerungsglied modelliert, welches je nach Paketlänge unterschiedlich lange wartet, entsprechend der eigentlichen Bearbeitungszeit. Die einzige PLB-Last, welche vom Prozessor generiert wird, rührt vom Lesen eines Pufferdeskriptors und des Headers sowie dem Schreiben des Pufferdeskriptor-Statusfeldes her.

In einem zweiten Modell, das in [25] detailliert beschrieben ist, werden Speicherzugriffe und somit auch die generierte PLB-Last je nach Prozessor-Anwendung modelliert. Für diesen Zweck werden verschiedene Netzwerkprozessor-Anwendungen bezüglich ihrer Art und Anzahl Instruktionen betrachtet. Die modellierten Anwendungen orientieren sich an den CommBench-Applikationen [26]. CommBench unterscheidet zwei Klassen von Anwendungen, nämlich solche mit Header Processing und solche mit Payload Processing. Je nach Anwendung werden verschiedene Typen von Instruktionen

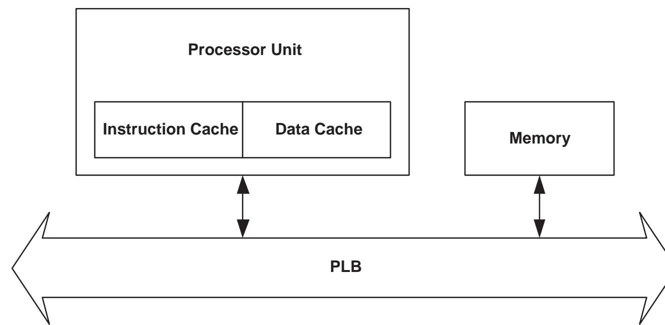

Abbildung 2.3: Prozessor-Modell.

unterschieden:

- Ladeinstruktionen: Instruktionen, die Daten vom Cache oder Hauptspeicher lesen.
- Schreibinstruktionen: Instruktionen, die Daten in den Cache oder Hauptspeicher schreiben.
- Interne Instruktionen: Alle restlichen Instruktionen, die keine externen Zugriffe benötigen.

Die Anzahl Instruktionen hängt von der Anwendung selber und im Falle von Payload Processing zusätzlich von der Paketgrösse ab. Weiter ist für jede Anwendung eine Verteilung der Instruktionen auf drei Phasen gegeben.

Je nachdem ob sich das Programm in der Start-, Mittel- oder Endphase befindet, ist die Wahrscheinlichkeit, dass ein bestimmter Typ von Instruktion auftritt, unterschiedlich. Zum Beispiel werden anfangs allgemein mehr Ladeinstruktionen und gegen Ende mehr Schreibinstruktionen auftreten. Die Speicherzugriffe sind von der Anzahl Lade- und Schreibinstruktionen, sowie von der Cachegrösse abhängig. Da der interne Prozessorcache eine partielle Kopie des Hauptspeicher beinhaltet, muss der Prozessor nicht auf externen Speicher zugreifen, sofern sich die Daten im Cache befinden. Da die totale Datenmenge im Cache und nicht sein eigentlicher Inhalt von Bedeutung ist, wird im Modell ein Zähler verwendet, welcher die momentan freie Cachekapazität repräsentiert. Das Prozessormodell ist sehr flexibel aufgrund der Vielzahl an Konfigurationsmöglichkeiten. Die wichtigsten Parameter sind hier aufgeführt:

- *NO_OF_INSTRUCTIONS_APP (processor_params.h)*: Anzahl Instruktionen je nach Anwendung.
- *LD_FREQUENCY_APP (processor_params.h)*: Anzahl Ladeinstruktionen je nach Anwendung.
- *ST_FREQUENCY_APP (processor_params.h)*: Anzahl Schreibinstruktionen je nach Anwendung.
- *PROCESSOR_SPEED (processor_params.h)*: Prozessor-Geschwindigkeit (Taktfrequenz).
- *CYCLES_PER_INSTRUCTION (processor_params.h)*: Anzahl Zyklen pro Instruktion.

Abbildungen 2.4 und 2.5 zeigen die resultierenden PLB-Zugriffe in Abhängigkeit der einstellbaren Cache-Parameter. Die Parameter

- *ICACHE_MISS_APP (processor_params.h)*: Missrate für den Instruktionscache. Pro Anwendung einstellbar.
- *DCACHE_MISS_APP (processor_params.h)*: Missrate für den Datencache. Pro Anwendung einstellbar.

entsprechen der Wahrscheinlichkeit, dass beim Ausführen einer Ladeinstruktion sich die entsprechenden Daten nicht im Cache vorfinden (Cache Miss). *APP* ist dabei ein Platzhalter für die verschiedenen Anwendungen, die im Prozessormodell simuliert werden. Abhängig von den Parametern:

- *ICACHE_SIZE (cache_params.h)*: Instruktionscache-Grösse
- *DCACHE_SIZE (cache_params.h)*: Datencache-Grösse

- *DC_WRITEBACK (processor_params.h)*

sind bei einem Cache Miss entweder ein oder zwei PLB-Zugriffe nötig. Falls noch Cachespeicher zur Verfügung steht, können die Daten vom Speicher in den Cache geschrieben werden. Falls kein Cachespeicher mehr frei ist, wird anhand des Parameters *DC_WRITEBACK* entschieden, ob vorhandene Daten überschrieben werden können. Falls nicht, muss ein Teil des Caches in den Speicher zurückgeschrieben werden und es sind zwei PLB-Zugriffe erforderlich – ein Lese- und ein Schreibzugriff.
Beim Ausführen einer Schreibinstruktion wird anhand des Parameters

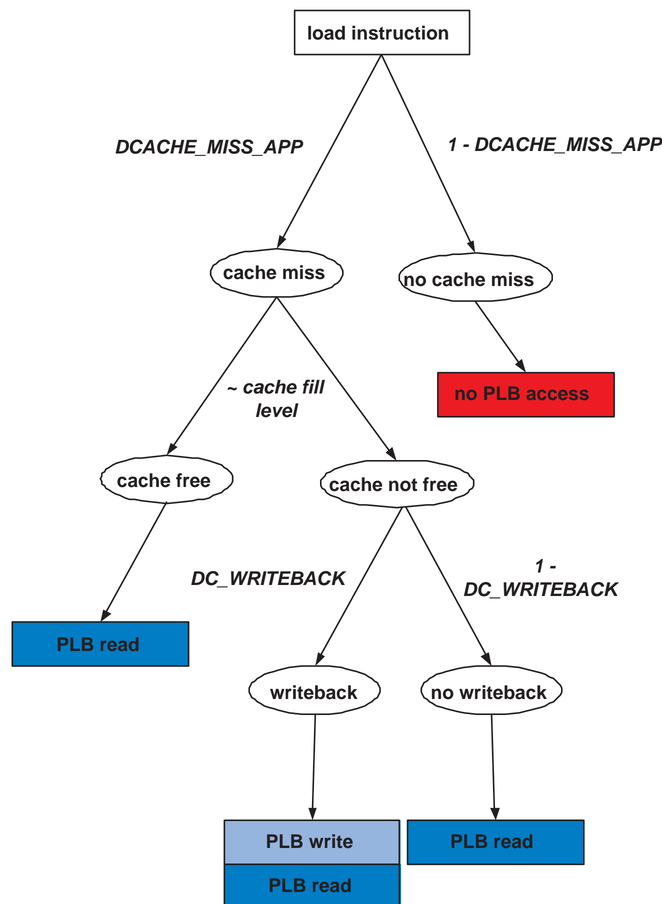

Abbildung 2.4: Ladeinstruktion.

- *WRITE_MEMORY (processor_params.h)*

die Wahrscheinlichkeit definiert, dass die Daten in den Hauptspeicher geschrieben und nicht bloss im Cache zur späteren Verwendung zwischen-

gespeichert werden. Falls die Daten im Cache abgelegt werden, spielt der Parameter

- *WRITE_OVERWRITE (processor_params.h)*

eine Rolle. Sollte nicht genügend Cachespeicher vorhanden sein, so wird gemäss diesem Parameters entschieden, ob vorhandene Daten überschrieben werden können.

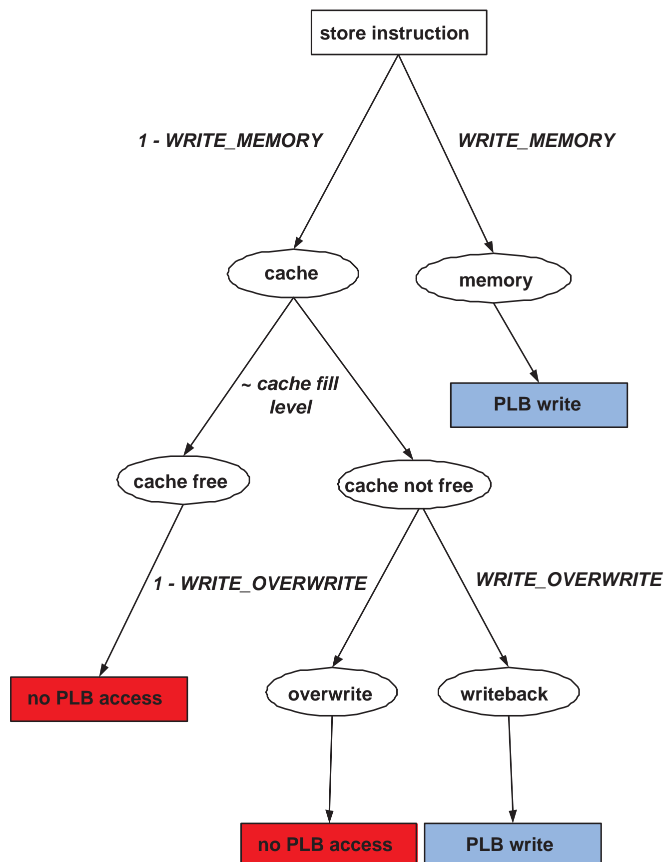

Abbildung 2.5: Schreibinstruktion.

Die Parameter sind derzeit so gewählt, dass ein General Purpose Prozessor simuliert wird.

2.3.6 Zusammenfassung

Anhand der Abbildung 2.6 wird der Datenpfad der modellierten Netzwerkprozessor-Architektur erklärt. Die Beschriftung der Pfeile entspricht der zeit-

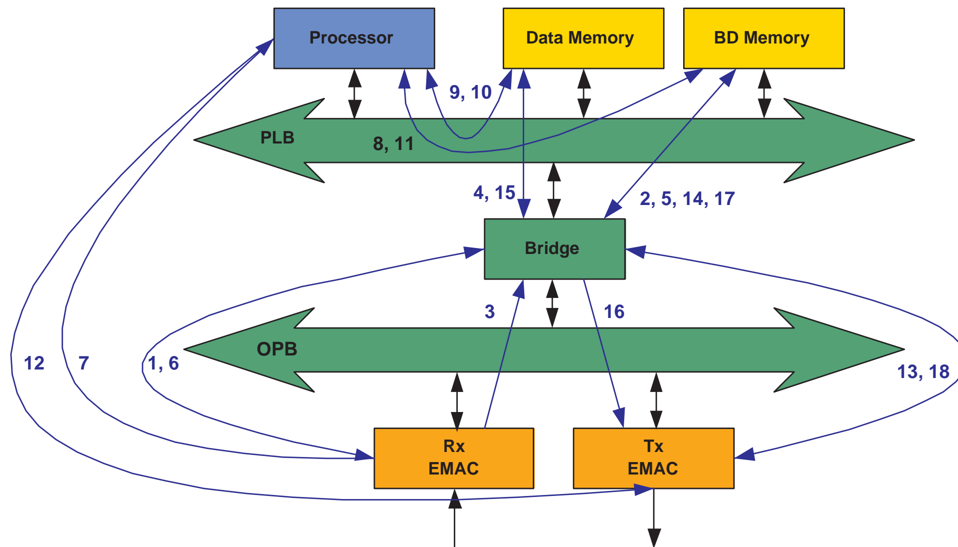

Abbildung 2.6: Datenpfad.

lichen Reihenfolge der Schritte, die bei der Abarbeitung eines Pakets durchlaufen werden.

1. Nach der Ankunft eines Pakets im EMAC wird die Brücke über Sideband-Signale für eine Paketübertragung getriggert.
2. Die Brücke liest einen Pufferdeskriptor. Die Paketübertragung kann beginnen.
3. Daten werden vom EMAC über den OPB gelesen.
4. Daten werden über den PLB in den Speicher geschrieben.
5. Die Schritte 3 und 4 werden solange wiederholt, bis das ganze Paket in den Speicher geschrieben worden ist. Falls nötig, müssen weitere Pufferdeskriptoren gelesen werden. Wenn das ganze Paket übertragen ist, wird das Statusfeld der verwendeten Pufferdeskriptoren aktualisiert.
6. Der EMAC wird über Sideband-Signale darauf hingewiesen, dass die Paketübertragung beendet ist.
7. Daraufhin triggert dieser über Sideband-Signale den Prozessor, um diesen zu informieren, dass ein Paket zur Bearbeitung bereitsteht.
8. Der Prozessor liest die nötigen Pufferdeskriptoren.
9. Prozessor liest über den PLB das Paket bzw. nur dessen Header, falls ausschliesslich Header-Processing-Anwendungen ausgeführt werden.

10. Schritt 9 wird solange ausgeführt bis das ganze Paket gelesen worden ist. Danach wird es bearbeitet und wieder in den Speicher zurückgeschrieben.
11. Statusfelder der Pufferdeskriptoren werden aktualisiert.
12. Über Sideband-Signale wird ein TxEMAC getriggert.
13. Daraufhin triggert dieser über Sideband-Signale die Brücke, um diese zu informieren, dass ein Paket zur Übertragung bereit steht.
14. Die Brücke liest einen Pufferdeskriptor. Die Paketübertragung kann beginnen.
15. Daten werden über den PLB gelesen.
16. Daten werden über den OPB in den TxEMAC geschrieben.
17. Die Schritte 15 und 16 werden so lange wiederholt, bis das ganze Paket in den TxEMAC geschrieben worden ist. Bei Beendigung der Übertragung werden die Pufferdeskriptoren freigegeben.
18. TxEMAC wird über Sideband-Signale darauf hingewiesen, dass die Paketübertragung beendet ist.

2.4 Erweiterte Architektur

Die bestehende Architektur wurde so ergänzt, dass die getätigten Erweiterungen im Einklang mit modernen Netzwerkprozessor-Architekturen stehen. Wie bereits in Abschnitt 1.1.3 erwähnt, werden in heutigen Netzwerkprozessoren mehrere Recheneinheiten eingesetzt, um den Durchsatz zu erhöhen. Die Referenzarchitekur wurde deshalb um eine Multiprozessoreinheit ergänzt, die in Abbildung 2.7 mit blauer Farbe eingezeichnet ist. Des weiteren wurde das Modell um eine DMA-Einheit erweitert, die in Abbildung 2.7 mit blauer Farbe gezeichnet ist. Dadurch besteht die Möglichkeit einen Netzwerkprozessor in Kombination mit einem DMA-Controller als Netzwerk-Terminal einzusetzen.

Durch diese Erweiterungen ist gleichzeitig eine sinnvolle Umgebung geschaffen worden, um die Arbitrierung auf PLB-Ebene zu analysieren. Damit die Arbitrierung einen merklichen Einfluss auf die Leistung haben kann, muss der PLB eine knappe Ressource darstellen. Dies bedeutet, dass zu gewissen Zeiten der PLB nicht alle Anfragen bearbeiten kann und es zu Konflikten zwischen den anfragenden Masters kommt. Sobald das System einen Zustand der Überlast erreicht, wird die Arbitrierung interessant. Durch die Verwendung effizienter Algorithmen, welche nach gewissen Kriterien die Abarbeitungsreihenfolge der Anfragen bestimmen, sind Gewinne hinsichtlich Durchsatz und mittlerer Verzögerung zu erwarten.

Arbitrierungsfragen auf OPB-Ebene werden hier nicht beachtet, da an diesem Bus nur ein einziger Master angeschlossen ist, nämlich die PLB-OPB-

Brücke.

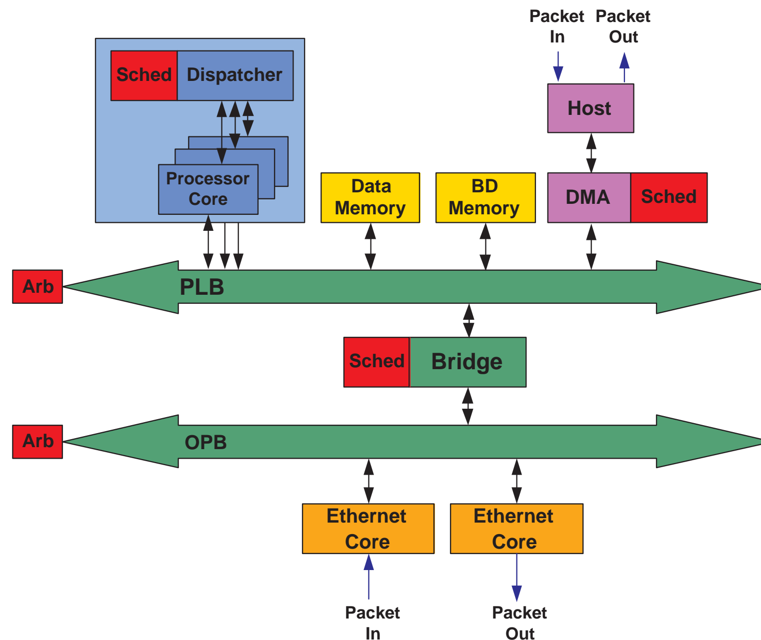

Abbildung 2.7: Erweiterte Netzwerkprozessor-Architektur.

2.4.1 Multiprozessoreinheit

Die Architektur wurde um zusätzliche Prozessoren erweitert, die unter der Kontrolle eines Dispatchers stehen. Dadurch kann die Linkgeschwindigkeit erhöht werden, da genügend Rechenleistung zur Verfügung steht, um die Paketverarbeitung mit der nötigen Geschwindigkeit zu verrichten.
Die Anzahl Prozessoren wird durch den Parameter

- *NUM_COPROCESSORS (MultiProcessor_params.h)*

festgelegt. Das Multiprozessormodell kann auf zwei verschiedene Arten betrieben werden: im Run-to-Completion- oder im Pipelining-Modus. Der entsprechende Parameter im Modell ist:

- *RUN2COMP (MultiProcessor_params.h).*

Im Run-to-Completion-Modus kann man die Recheneinheiten als gleichwertige General Purpose Prozessoren betrachten, die alle die gleichen Anwendungen ausführen können. Ein Paket wird in diesem Modus jeweils nur von

einem Prozessor bearbeitet. Der Dispatcher ist so etwas wie ein Load Balancer, der die anfallende Last unter den vorhanden Prozessoren aufteilt.

Im Pipelining-Modus kann man die Recheneinheiten als dedizierte Hardware-Einheiten betrachten, die jeweils nur eine bestimmte Paketverarbeitungsfunktion ausführen. Ein Paket wird in diesem Modus von mehreren Recheneinheiten nacheinander bearbeitet. In beiden Modi basieren die einzelnen Recheneinheiten auf dem bereits existierenden Prozessormodell, das in Abschnitt 2.3 beschrieben wurde. Im Pipelining-Modus müssen die einzelnen Verarbeitungsfunktionen auf die entsprechenden Prozessoren abgebildet werden. Dies wird mit Hilfe von

- *WHO_DOES_IT (MultiProcessor_params.h).*

festgelegt.

Im Run-to-Completion-Modus wird von einem Prozessor folgende PLB-Last generiert: Es werden, je nach Paketgrösse, die nötigen Pufferdeskriptoren gelesen und nach der Bearbeitung deren Statusfeld aktualisiert. Je nach Anwendung wird nur der Header oder das ganze Paket gelesen. Bei Payload-Anwendungen werden nach der Bearbeitung die Daten in den Speicher zurückgeschrieben. Bei Header-Anwendungen wird ein Zurückschreiben des möglicherweise modifizierten Headers vernachlässigt. Zusätzliche PLB-Last ist abhängig von der Cachegrösse und der Cachemissrate.

Im Pipelining-Modus fällt die gleiche PLB-Last pro Pipeline-Stufe an. Das bedeutet, dass ein Paket beim Durchlaufen von vier Recheneinheiten, die jeweils unterschiedliche Anwendungen ausführen, viermal so viel Last erzeugt, wie wenn nur ein Prozessor alle Anwendungen ausführen würde. Die vom Programmfluss abhängigen PLB-Zugriffe (Cachemisses, Memory Writebacks) werden auf die ganze Pipeline verteilt, nehmen aber in ihrer Zahl nicht zu.

Ein weiteres Merkmal der Multiprozessoreinheit ist, dass der Dispatcher als funktionale Einheit eines bestimmten Prozessors unter den vorhandenen betrachtet werden kann. Mit Hilfe des Parameters

- *CPU_COPROCESSOR (MultiProcessor_params.h).*

wird ein Prozessor bestimmt, der mit dem Dispatcher eine Einheit bildet. Durch diese Einstellung wird dem Dispatcher ermöglicht, über die PLB-Schnittstelle des Prozessors den Bus zu benutzen. Im Modell wird von dieser Eigenschaft nicht Gebrauch gemacht. Der Dispatcher wird jeweils als externe Einheit aufgefasst, die eine reine Kontrollfunktion besitzt und nicht auf

den PLB zugreifen muss. Diese Fähigkeit wurde im Hinblick auf mögliche Erweiterungen des Multiprozessormodells implementiert.

2.4.2 Direct Memory Access Controller

Durch eine DMA-Einheit lassen sich direkt am PLB neue Verkehrsquellen und -senken hinzufügen, um unabhängig vom OPB die PLB-Last zu variieren. Durch diese Erweiterung ändert sich der Datenpfad. Pakete können nun über die DMA-Einheit in das System eingespiesen werden oder das System verlassen.

Das DMA-Modell ist eine vereinfachte Version der PLB-OPB-Brücke. Im Vergleich zur Brücke ist die OPB-Schnittstelle nicht vorhanden. Die DMA-Einheit verfügt des weiteren über eine Schnittstelle zur PLB-OPB-Brücke. Sobald ein Paket beim DMA-Block zur Übertragung bereitsteht, müssen zuerst die erforderlichen Pufferdeskriptoren bezogen werden. Da die PLB-OPB-Brücke diese zentral verwaltet, ist es Aufgabe der Brücke, die Deskriptoren der DMA-Einheit bekanntzugeben. Im realen System sendet die Brücke dem DMA-Block über den Kontrollbus (siehe Abschnitt 2.3) eine Datenstruktur, die auf den nächsten zur Verfügung stehenden Pufferdeskriptor zeigt. Im Modell lässt sich für diesen Vorgang die Latenz konfigurieren, d.h. die Anzahl Taktzyklen, die vom Zeitpunkt der DMA-Anfrage eines Pufferdeskriptors bis zu dessen Verfügbarkeit in der DMA-Einheit vergehen. Der zugehörige Parameter heisst

- *LATENCY_NEXT_BD (DMA_params.h).*

Am DMA-Block ist ein Host-Modul angeschlossen, das eine Verkehrsquelle und -senke darstellt. Der Host entspricht im Modell einem EMAC-Modul und kann ebenfalls als Eingangs- und Ausgangsport betrachtet werden. Die Kommunikation zwischen Host und DMA-Einheit entspricht der Art und Weise wie ein EMAC mit der Brücke kommuniziert. In diesem Fall ist aber kein OPB vorhanden, und die Pakete stehen der DMA-Einheit zur Verfügung, sobald diese vom Host gelesen wurden.

Durch folgende Parameter ist die DMA-Einheit konfigurierbar:

- *HOST_RATE (DMA_params.h):* Enstpricht der Geschwindigkeit, mit der Pakete vom Host gelesen werden.
- *IFG_HOST (DMA_params.h):* Zwischenankunftszeit der Pakete.
- *RX_HOST_NBR (DMA_params.h):* Anzahl Verkehrsquellen.
- *TX_HOST_NBR (DMA_params.h):* Anzahl Verkehrssenken.

- *DMA_RX_BURST_SIZE (DMA_params.h):* Grösse der Dateneinheit, die im Empfangspfad von der DMA übertragen wird.

- *DMA_TX_BURST_SIZE (DMA_params.h):* Grösse der Dateneinheit, die im Sendepfad von der DMA-Einheit übertragen wird.

2.5 Matlab-Simulation

Zur Untersuchung des Datenaustauschs auf dem Processor Local Bus wurde neben der SystemC-Simulation auch eine Matlab-Simulation erstellt. Dieses parallele Vorgehen erlaubte einerseits die Umgehung der Probleme mit der übernommenen SystemC-Simulation, andererseits aber auch eine Fokussierung auf die Umgebung des PLB und damit eine ausschliessliche Simulation der für die Untersuchung der PLB-Arbitrierung relevanten Prozessoreigenschaften. Weiter bietet Matlab den wertvollen Vorteil einer grösstmöglichen Zugänglichkeit zu den Simulationsdaten und dazu eine Vielzahl an spezialisierten Befehlen zur Auswertung und Darstellung der Ergebnisse.

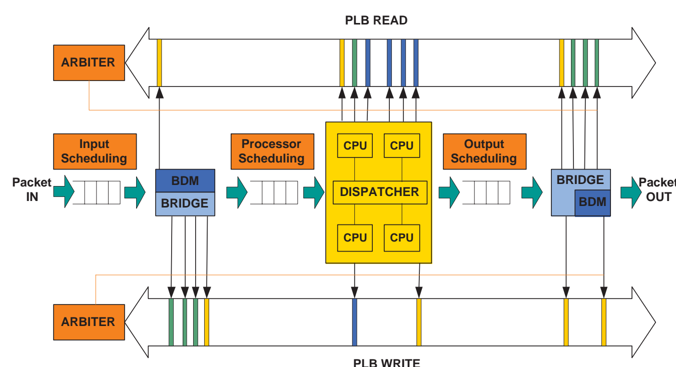

Abbildung 2.8: PLB-Umgebung mit typischem Requestmuster.

Das für die Matlab-Simulation verwendete Modell lässt sich am besten anhand von Abbildung 2.8 erläutern. Dargestellt ist die Umgebung des PLB (dieser ist aufgeteilt in Read- und Write-Bus) mit den involvierten PLB-Masters. Die Pakete durchlaufen den mit grünen Pfeilen markierten Weg von der Bridge (Receive Path) über den Prozessorblock und zurück zur Bridge (Transmit Path). Dieser Prozess verursacht für jedes Paket ein charakteristisches Muster von Requests auf den beiden Komponenten des PLB. Es wurde darauf geachtet, dass dieses Muster so weit als möglich dem im

SystemC-Simulator implementierten Verhalten entspricht[2], indem die Parameter, welche für die Zahl, den Moment des Auftritts und die Länge der Requests verantwortlich sind, miteinbezogen wurden.

Von zentraler Bedeutung sind die drei Scheduler (Input, Processor und Output Scheduling) sowie die Bus-Arbitrierung. Ziel der Simulation ist die Untersuchung des Zusammenspiels der Arbitrierung mit dem Scheduling der Verarbeitungseinheiten des Netzwerkprozessors. Zu diesem Zweck wurde eine Reihe an Arbitern implementiert, die mit verschiedenen Schedulern kombiniert werden können. Die betrachteten Szenarien mit den entsprechenden Resultaten sind Thema im Kapitel 3.

Wie erwähnt sind die nahezu unbegrenzten Möglichkeiten in Sachen Auswertung ein Pluspunkt der Matlab-Simulation. So erlauben die im Anschluss an die Simulation erzeugten Graphiken einen schnellen Ueberblick der Resultate und bieten gleichzeitig Hand für eine genauere Analyse und weiterführende Erkenntnisse.

[2] Es ist hier anzumerken, dass dies nicht in allen Fällen verifiziert wurde, da der SystemC-Simulator nur beschränkt eingesetzt werden konnte. Es kann also nicht mit hundertprozentiger Sicherheit auf eine Übereinstimmung der Resultate gebaut werden.

Kapitel 3

Bus-Arbitrierung

3.1 Definition des Arbiters

Wie in Kapitel 1.1.3 beschrieben, werden in modernen Netzwerkprozessoren mehrere Recheneinheiten gleichzeitig eingesetzt, um die parallele Verarbeitung von Paketen zu ermöglichen. Der erhöhte Aufwand wird mit einem bedeutend gesteigerten Durchsatz belohnt, womit auch hohe Linkkapazitäten von der Elektronik gemeistert werden können. Mit der Anzahl der involvierten Prozessorblöcke nimmt auch die Komplexität ihrer gegenseitigen *Kommunikation* zu. Eine vergleichsweise einfache Lösung, welche die relevanten Komponenten verbindet und gleichzeitig einen hohen Grad an Flexibilität (Ausbaufähigkeit und Konfigurierbarkeit) bietet, ist der Einsatz eines Busses. Im vorliegenden Beispiel dient der so genannte Processor Local Bus (PLB) dazu, den Austausch von Daten unter den verschiedenen Komponenten (Rechen- und Speichereinheiten, die PLB-OPB-Brücke) zu ermöglichen.

Da es sich bei einem Bus um eine geteilte Ressource (shared medium) handelt, sind Konflikte beim Zugriff, insbesondere im Falle einer hohen Auslastung, nicht zu vermeiden. Die Aufgabe, eine gleichzeitige Benutzung durch mehrere Masters zu verhindern, obliegt einer dem Bus angegliederten Instanz – dem Arbiter[1]. Dieser arbitriert die Anfragen der Masters, welche zu einem gegebenen Zeitpunkt Daten über den Bus schicken möchten. Der Arbiter entspricht damit einer Funktion, die aus einer vorliegenden Menge von Busanfragen (Requests) diejenige auswählt, welche als nächste abgearbeitet werden soll. Dies erfolgt in der Regel dadurch, dass dem entsprechenden Master die Vollmacht zur alleinigen Benutzung des Busses während einer bestimmten Zeit oder bis zum Zeitpunkt, da dieser ihn nicht mehr benötigt,

[1]lat. arbiter = Schiedsrichter

erteilt wird. Als Kriterien für den Entscheid des Arbiters kommen z. B. in Frage:

- die Identität des Masters
- die Identität des Slaves
- die Art des Requests (Priorität, Grösse u.a.)
- der Zustand des Systems (Belastungsgrad, Betriebsmodus)
- die vorhergehenden Arbiterentscheide

Basierend auf den genannten Kriterien sind unzählige Arbitrierungsschemen denkbar. In der Praxis basiert der Entscheid des Arbiters jedoch meist nicht auf mehreren Faktoren, womit der damit verbundene Algorithmus verhältnismässig einfach gehalten werden kann.

3.2 Konzepte zur Arbitrierung

In der Vergangenheit wurden zahlreiche Methoden entwickelt, die das Problem der Arbitrierung in unterschiedlicher Weise angehen. Von den existierenden Algorithmen sollen hier einige kurz vorgestellt werden. Zu ihrer Beurteilung sind die folgenden Kriterien von Bedeutung:

- **Fairness.**[2] Ein Arbiter ist dann fair, wenn er allen Masters denselben Anteil an Busbandbreite zugesteht, sofern diese ständig neue Requests anbringen. Fairness kann auch in dem Sinne uminterpretiert werden, als dass jedem Master ein gewisser Mindestanteil garantiert wird, der jedoch von Master zu Master variiert. In diesem Fall spricht man von *Weighted Fairness*.
- **Begrenzte Latenz.** Eine begrenzte Latenz ist dann gewährleistet, wenn eine obere Zeitlimite für die Abarbeitung eines Requests existiert.
- **Busauslastung.** Ein Arbiter sollte dafür sorgen, dass die Auslastung des Busses maximal ist, solange von Seiten der Masters Anfragen zur Abarbeitung bereitstehen.
- **Komplexität der Implementierung.** In der Praxis spielt es eine wichtige Rolle, ob ein Arbitrierungsalgorithmus leicht in schneller Hardware implementiert werden kann. Die entsprechende Komponente

[2]Für den Begriff der *Fairness* existieren in der Literatur verschiedene zum Teil relativ umständliche Definitionen. Hier werden wir uns auf eine einfache Formulierung beschränken, welche für die folgende Betrachtung unterschiedlicher Arbiter ausreicht.

sollte ihre Aufgabe möglichst ohne grossen Speicher- und Rechenaufwand in kürzester Zeit erledigen können.

Zur Implementierung ist noch anzumerken, dass gewisse Schemen nicht unbedingt durch eine separate Hardware-Einheit umgesetzt werden müssen, sondern auch verteilt in den Busmasters realisiert werden können, man spricht dann von *dezentraler Arbitrierung*.

3.2.1 Beispiele von Arbitern

3.2.1.1 Static Priority

Der wohl einfachste Weg, einen Arbitrierungsentscheid zu fällen, besteht darin, sich ausschliesslich auf eine statisch festgelegte Masterpriorität zu konzentrieren. Der Bus wird dem Master überlassen, der in der Rangfolge den obersten Platz belegt unter denjenigen, die den Bus beanspruchen möchten. Ein einziger Master erhält somit die Möglichkeit, den Bus auf alle Ewigkeit zu blockieren, sofern er nicht von sich aus auf weitere Datenübertragung verzichtet. Damit sind Fairness und begrenzte Latenz nicht gegeben. Hingegen kann dieser Arbiter mit einer guten Busauslastung und einer problemlosen Implementierung aufwarten.

3.2.1.2 Round Robin und Weighted Round Robin

Ein Round-Robin-Algorithmus benötigt eine zyklische Liste der involvierten Masters sowie einen Zeiger, der auf einen Listeneintrag zeigt. Bei der Arbitrierung erhält nun derjenige Master Priorität, auf welchen der Round-Robin-Pointer zum aktuellen Zeitpunkt gerade zeigt. Liegt vom entsprechenden Master kein Request vor, so springt der Zeiger zum nächsten Listeneintrag (bzw. Master) bis schliesslich ein Master an der Reihe ist, der auch einen Request bereitliegen hat. Nach der Abarbeitung dieser Anfrage springt der Zeiger für die nächste Arbitrierung auf den darauffolgenden Listeneintrag. Beim einfachen Round-Robin-Algorithmus enthält die zyklische Liste für jeden Master einen Eintrag, während die Gewichtung beim Weighted-Round-Robin-Algorithmus durch mehrfache Einträge gewisser Masters erreicht werden kann. Der (Weighted-)Round-Robin-Arbiter ist gleichzeitig fair und garantiert eine begrenzte Latenz. Seine Implementierung ist nicht besonders schwierig, benötigt er doch nur wenig Speicherplatz (für die zyklische Liste und den Zeiger) und keine komplexen Rechenoperationen.

3.2.1.3 First-come-first-served-Arbiter (FCFS)

Der FCFS-Arbiter arbeitet mit einer First-in-first-out-Warteschlange. Requests werden also in der Reihenfolge ihres Eintreffens abgearbeitet. Wird eine einzige Warteschlange verwendet, so ist die durchschnittliche Wartezeit pro Request für alle Masters gleich (es sei denn, diese treten nach einem zeitlich korrelierten Muster auf). Der einem Master zur Verfügung stehende Bandbreitenanteil richtet sich nach dem Verhältnis seines Requestaufkommens zur gesamten Menge an Requests pro Zeiteinheit. Damit erfüllt das Verfahren das Kriterium der Fairness nicht, da Masters mit hohem Requestaufkommen die den restlichen Masters zur Verfügung stehende Bandbreite schmälern. Die Latenz hängt stark vom gesamten Requestaufkommen ab; es kann jedoch eine obere Grenze angegeben werden, sofern der Bus so ausgelegt ist, dass pro Master nur eine begrenzte Zahl Requests gleichzeitig vorliegen kann. Obwohl der FCFS-Arbiter vom Gedanken her simpel daherkommt, kann sich seine Implementierung durch die Notwendigkeit einer zentral verwalteten Warteschlange relativ aufwändig gestalten.

3.2.1.4 Request Priority Arbiter

Anstelle einer Priorisierung aufgrund der Master-Identität kann auch eine Bevorzugung anhand der Request-Identität in Betracht gezogen werden. Dies ist insbesondere dann von Interesse, wenn die auftretenden Anfragen von der Verarbeitung von Daten unterschiedlicher Dringlichkeit herrühren. Ist also ein bestimmter Master beispielsweise im Begriff, ein zeitkritisches Paket zu verarbeiten, so kann er eine unerwünschte Verzögerung beim Buszugriff durch das Markieren seiner Requests verhindern bzw. eindämmen. Über ein längeres Zeitintervall betrachtet, entsteht ein Szenario ähnlich demjenigen der statisch zugewiesenen Masterpriorität, jedoch mit dem Unterschied, dass die Masterpriorität dynamisch variieren kann. Der zusätzliche Aufwand besteht darin, vom Master eine zusätzliche Leitung zum Arbiter zu führen, welche die Priorität des aktuellen Requests signalisiert. Sofern nicht zahlreiche Prioritätsstufen gleichzeitig zum Einsatz kommen, sind bei hoher Busauslastung Konflikte von Requests gleicher Priorität zu erwarten. In diesem Fall muss der Arbiter für seine Entscheidung auf eine andere Ebene zurückgreifen können. So wird beispielsweise bei einem Patt die Masterpriorität oder ein Round-Robin-Schema zu Rate gezogen. Die Kriterien der Fairness und der begrenzten Latenz müssen dann anhand dieser zweiten Ebene beurteilt werden.

3.2.1.5 Random Arbiter

Als letztes Beispiel soll hier noch der Random Arbiter aufgeführt werden. Ein solcher wählt aus den anstehenden Requests nach dem Zufallsprinzip einen aus und schert sich nicht um weitere Faktoren. Der Algorithmus ist zwar theoretisch gesehen nicht fair und verfügt auch nicht über eine begrenzte Latenz, verhält sich jedoch über längere Zeit betrachtet ähnlich wie ein FCFS-Arbiter. Seine Umsetzung verlangt die Integration eines Zufallsgenerators, was im Hinblick auf den fehlenden speziellen Nutzen unverhältnismässig erscheint. Hingegen kann der Random Arbiter als Referenzalgorithmus beigezogen werden, um in vergleichenden Simulationen die Nützlichkeit massgeschneiderter Algorithmen zu demonstrieren.

3.2.2 Prioritäts- oder Bandbreiten-orientierte Algorithmen

Die vorgestellten Arbitrierungsschemen können in zwei verschiedene Klassen eingeordnet werden:

- **Priority Oriented Algorithms.** Arbiter, die für ihren Entscheid auf einen Prioritätsparameter zurückgreifen, sind dann angebracht, wenn der Buszugriff eines bestimmten Masters von übergeordneter Wichtigkeit ist und damit die Unterdrückung aller anderen Requests rechtfertigt. Wird die Prioritätszuordnung auf Request-Ebene vorgenommen, so macht dies beispielsweise dann Sinn, wenn in einem Netzwerkprozessor gleichzeitig priorisierte und nicht-priorisierte Pakete in Verarbeitung sind.

- **Shared Bandwidth Algorithms.** Soll in einem Prozessor die zur Verfügung stehende Buskapazität unter den Masters gleichmässig oder nach einem bestimmten Schlüssel aufgeteilt werden, so kann dies durch einen geeigneten Arbiter bewerkstelligt werden. An dieser Stelle sollte man zwischen Algorithmen unterscheiden, die bloss mit gewissen Einschränkungen die Kriterien der Fairness und der begrenzten Latenz erfüllen und solchen, die effektive Garantien bezüglich der pro Master zur Verfügung stehenden Bandbreite abgeben. Liegt eine klar begrenzte Latenz vor sowie eine minimale Anzahl garantierter Requests pro Zeit, so haben die Masters die Gewissheit, ihr Datenaufkommen in einer im Voraus abschätzbaren Zeit übermitteln zu können.

Von den im vorhergehenden Abschnitt beschriebenen Arbitern gehören zwei in die Klasse der Prioritäts-orientierten Algorithmen (Static Priority und Request Priority), während die restlichen der Bandbreiten-orientierten Klasse zuzuordnen sind. Unter letzteren erfüllt der (Weighted-)Round-Robin-Arbiter die Garantie-Anforderungen bezüglich minimalem Bandbreitenan-

teil und maximaler Latenz. Der FCFS-Arbiter kann mit vergleichbaren Qualitäten aufwarten, sofern die Anzahl gleichzeitig vorhandener Requests pro Master begrenzt ist.

3.2.3 Mehrstufige Arbitrierung

Wie bereits erwähnt, kann es bei der Arbitrierung nach einem primären Schema zu Situationen kommen, die einen Entscheid anhand eines sekundären Schemas erfordern. Darüber hinaus ist die Verwendung von zusätzlichen Ebenen auch ein mögliches Mittel, einen differenzierteren Arbiterentscheid zu fällen, als es ein einfacher Arbiter erlauben würde. Beispielsweise könnte ein einzelner Master aufgrund seiner Wichtigkeit über alle anderen priorisiert werden, während unter den übrigen Masters ein Round-Robin-Schema für die Arbitrierung sorgt. Eine mehrstufige Praxis ermöglicht damit eine Arbitrierung, die auf die Eigenschaften des Systems und dessen Aufgabe zugeschnitten ist. Abhängigkeiten zwischen den einzelnen Systemkomponenten und Bandbreitenbedürfnissen kann dadurch optimal Rechnung getragen werden. Der Preis einer ausgeklügelten Arbitrierung ist jedoch stets die erhöhte Komplexität. Hier wird es oftmals notwendig, zugunsten einer implementierbaren Lösung Abstriche zu machen.

3.3 Arbitrierung im Netzwerkprozessor

Das Thema der vorliegenden Arbeit ist die Untersuchung der Arbitrierung am konkreten Beispiel eines Netzwerkprozessors. Als Vorlage einer Prozessorarchitektur dient das in Kapitel 2 beschriebene Modell. Nun soll in erster Linie der Einfluss der Bus-Arbitrierung auf die Verzögerung der Pakete bei der Verarbeitung durch den Prozessor nachvollzogen und beschrieben werden, während in einem zweiten Schritt eine optimale Lösung in Bezug auf die Arbitrierung für verschiedene Szenarien erarbeitet werden soll.

3.3.1 Ausgangslage

Für die folgende Untersuchung ist eine Paketverarbeitung im parallel ablaufenden Run-to-completion-Modus vorgesehen, d.h. mehrere Pakete werden gleichzeitig von einer Anzahl (identischer) Prozessoren verarbeitet, wobei jeweils alle auszuführenden Anwendungen vom selben Prozessor vorgenommen werden, ein einzelnes Paket also immer nur einen Prozessor durchläuft. Vor der Verarbeitung wird das Paket über die PLB-OPB-Brücke in den Speicher geschrieben, am Ende wird es wieder von der Brücke gelesen und an einen EMAC weitergeleitet. Um die Situation fürs erste möglichst über-

schaubar zu halten, wurde auf den Einsatz einer DMA-Einheit als zusätzliche Verkehrsquelle und -senke verzichtet. Weiter wird angenommen, dass alle ankommenden Pakete einer einzigen Trace entstammen, somit also keine Konflikte zwischen unterschiedlichen Input- und Output-Strömen auftreten.

Im Zentrum der Untersuchung steht die Verwaltung des Datenverkehrs auf dem PLB und ihr Einfluss auf den Paketfluss. Um sicherzustellen, dass äussere Faktoren eine untergeordnete Rolle spielen, wurde angenommen, dass die Kapazität des OPB keinen begrenzenden Faktor darstellt. Weiter wollen wir uns ausschliesslich auf Header Processing Applications (HPA) beschränken und die durchzuführenden Anwendungen für alle Pakete gleich halten. Tabelle 3.1 enthält die wichtigsten Parameter wie sie weiter unten für die Simulationen gewählt wurden.

Was den eintreffenden Paketfluss angeht, so besteht eine Variabilität in folgenden drei Punkten:

- **Paketgrösse.** Die Paketgrösse variiert gemäss einer vorgegebenen Internet Traffic Trace [2]. Ihre Verteilung ist in Abbildung 3.1 ersichtlich.
- **Interframe Gap.** Der Abstand zweier aufeinanderfolgender Pakete richtet sich nach einer Exponentialverteilung mit gegebenem Mittelwert.
- **Priorität.** Nach dem Zufallsprinzip wird ein gewisser Anteil der Pakete als priorisierte Daten markiert, welche bestimmten Quality-of-Service-Ansprüchen genügen sollen.

Es kann weiter angenommen werden, dass der Paket-Trace durch so genannte *Arrival Curves* Grenzen gesetzt sind in Bezug auf die Menge an Paketen bzw. Daten, die in einem gewissen Zeitintervall maximal ankommen dürfen. Eine solche Begrenzung entspricht beispielsweise den Tspec-Kurven [28].

In den betrachteten Simulations-Szenarien wurde jeweils angenommen, dass nur eine EMAC-Einheit Pakete in den Netzwerkprozessor speist und nach ihrer Verarbeitung weiter überträgt.

3.3.2 PLB-Verkehr

Der zu verarbeitende Paketstrom wird auf dem PLB eine charakteristische Last erzeugen. Anhand von Abbildung 3.2, welche bereits zur Illustration des Matlab-Modells diente, soll hier genauer erläutert werden, welches Verkehrsaufkommen bei der Verarbeitung eines Pakets generiert wird (siehe dazu auch Kapitel 2, Abschnitt 2.3.6).

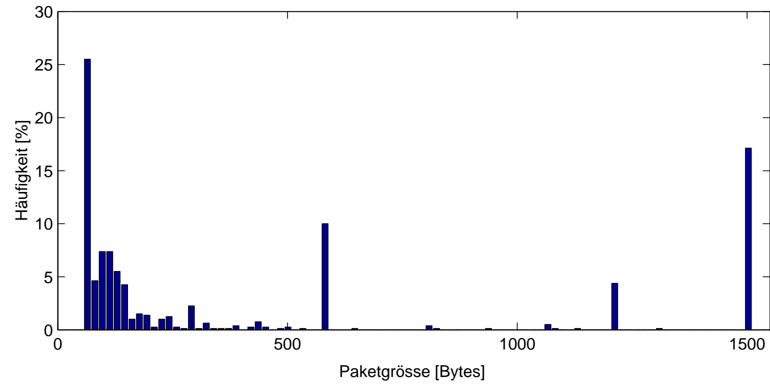

Abbildung 3.1: Verteilung der Paketlänge in der verwendeten Trace.

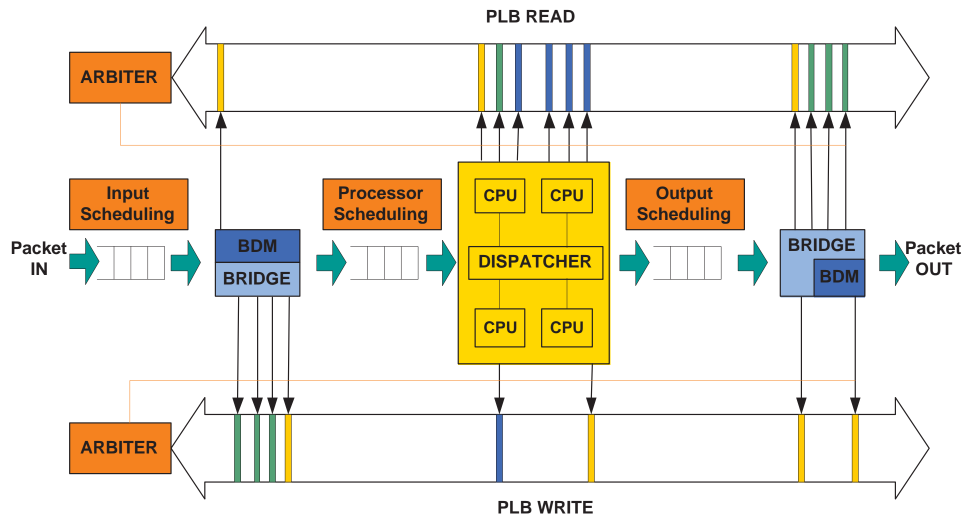

Abbildung 3.2: Verkehr auf dem PLB.

Der Prozess lässt sich in drei Etappen unterteilen: Transfer des Pakets in den Speicher, Verarbeitung des Pakets durch einen Prozessor sowie Transfer des Pakets aus dem Speicher heraus zur weiteren Übertragung.

3.3.2.1 Transfer in den Speicher (Receive Path)

In dieser Etappe beginnt nach einem Zugriff des Pufferdeskriptoren-Verwalters auf den Read-Bus der eigentliche Transfer des Pakets (Header und Payload) in den Speicher. Am Schluss steht ein Status-Write-Zugriff auf den

Pufferdeskriptor-Speicher. Als Masters sind der Pufferdeskriptoren-Verwalter und vor allem die Brücke aktiv. Der überwiegende Teil des Verkehrs wird auf dem Write-Bus verursacht; bei Paketen, deren Grösse den Umfang des Headers deutlich übersteigt, ist die Buslast in etwa proportional zur Paketgrösse.

3.3.2.2 Paketverarbeitung durch den Prozessor

Auch diese Etappe beginnt mit einem Read-Zugriff zwecks Bezug eines Pufferdeskriptors. Darauf folgt der Transfer des Headers in den prozessoreigenen Cachespeicher. Während der Ausführung der Anwendungen finden verschiedentlich Read- und Write-Zugriffe statt aufgrund von Cache Misses (read) und Memory Writebacks (write). Wird der jeweilige Prozessor-Cache knapp dimensioniert (einige Kilobytes), so sind die Read-Zugriffe eindeutig in der Überzahl. Auch diese Etappe wird durch einen Status-Write-Zugriff abgeschlossen. Als Masters fungieren in dieser Etappe ausschliesslich die Prozessoren (bzw. ein einziger Prozessor bei Betrachtung eines einzelnen Pakets). Die PLB-Last ist von der Paketgrösse unabhängig, handelt es sich doch ausschliesslich um HPAs. Die Anzahl Requests pro Paket kann relativ stark variieren, gehorcht aber in unserem Modell in etwa einer Gaussverteilung, von der Mittelwert und Varianz bekannt sind und die praktisch gesehen nach oben begrenzt ist.

3.3.2.3 Transfer aus dem Speicher heraus (Transmit Path)

Nach dem Lesen des Pufferdeskriptors werden über den Read-Bus die Header- und Payload-Daten vom Speicher über die Brücke zum EMAC übertragen. Ein Status-Write schliesst diesen Vorgang ab, womit das Paket den Netzwerkprozessor verlassen kann. Im Anschluss folgt noch ein Write-Zugriff des Pufferdeskriptoren-Verwalters, welcher jedoch für die Paketverzögerung nicht mehr relevant ist. Die letzte Etappe ist weitgehend analog zur ersten, mit dem Unterschied, dass die Busbelastungen auf dem Read- und Write-Bus vertauscht sind.

3.3.3 Systemkapazität und -dimensionierung

In diesem Abschnitt soll kurz auf die Dimensionierung und die zu erwartende Kapazität des vorliegenden Systems eingegangen werden. Als gegebene Grössen werden angenommen:

seitens der Prozessoren:

- die mittlere Rechenzeit t_{appl}, welche zur Verarbeitung eines Pakets benötigt wird,
- die mittlere Anzahl Requests r, die ein Prozessor während der Verarbeitung eines Pakets an den PLB richtet.
- die zur Verarbeitung eines Bus-Requests benötigte Zeit t_{bus},

seitens der Brücke:

- die mittlere PLB-Zeit t_{bridge}, welche die Brücke benötigt, um ein Paket zu übertragen. (Dies entspricht der Reziprokwert des Paket-Durchsatzes der Brücke D_{bridge}.)

Die PLB-spezifischen Grössen hängen ihrerseits von einer Vielzahl an Parametern und Eigenheiten der Bus-Architektur ab. Für die folgende Abschätzung wird vereinfachend angenommen, dass alle Requests dieselbe Grösse aufweisen. Der Bus sei stets voll oder näherungsweise voll ausgelastet. Die Buskapazität widerspiegelt sich in den Zeiten t_{bus} sowie t_{bridge}. Was die Prozessorgrössen angeht, so sind diese eine Konsequenz der gewählten Anwendungen und der Prozessoreigenschaften (Taktfrequenz, Cache-Speicher, Cycles per Instruction etc.). Sind die auszuführenden Anwendungen für alle Pakete gleich und zudem nicht von der Grösse der Pakete abhängig (Header Processing Applications), so kann t_{appl} als konstant angesehen werden. Für die Anzahl Requests r wird zur analytischen Betrachtung ein Mittelwert verwendet. Das Laden des Headers fliesst in r ein.

Ausgehend von den genannten Eigenschaften soll nun näherungsweise eine Abschätzung der zu erwartenden Systemkapazität vorgenommen werden. Wenn man davon ausgeht, dass sich das System in einem über ein längeres Zeitintervall betrachtet stabilen Zustand befindet, so kann ein (mittlerer) Paketdurchsatz D eingeführt werden. Pro Paket steht damit durchschnittlich eine PLB-Zeit

$$t_{PLB} = \frac{1}{D} \tag{3.1}$$

zur Verfügung. Da im besprochenen Prozessor die Read-Komponente des PLB den kapazitätsbegrenzenden Faktor darstellt, soll im folgenden das Augenmerk ausschliesslich auf diesen Teil des Busses gerichtet werden. Im wesentlichen wird der Read-Bus von der Brücke und den Prozessoren genutzt; der kleine Anteil an Buffer-Descriptor-Verkehr kann hier vernachlässigt werden. Sofern der Bus (langfristig gesehen) nicht überlastet ist, muss gelten

$$t_{PLB} \leq t_{bridge} + r\, t_{bus}. \tag{3.2}$$

Der zweite Term der Summe entspricht der vom Prozessor beanspruchten PLB-Zeit pro Paket. Die PLB-Auslastung Γ ergibt sich damit zu

$$\Gamma = \frac{t_{bridge} + r\, t_{bus}}{t_{PLB}}. \tag{3.3}$$

Wird der Bus an seine Grenzen getrieben, so ist er im Mittel (praktisch) voll ausgelastet. Damit wird $\Gamma \approx 1$ und es gilt

$$D_{max} = \frac{1}{t_{bridge} + r\, t_{bus}}. \tag{3.4}$$

Somit erhalten wir mit D_{max} eine Obergrenze für den maximalen (mittleren) Paketdurchsatz. Wird also der Netzwerkprozessor über längere Zeit mit einer Paketrate grösser als D_{max} belastet, so ist früher oder später ein Speicherüberlauf zu erwarten.

Nachdem nun das maximal zu bewältigende Verkehrsaufkommen bekannt ist, stellt sich die Frage, welche Rechenleistung zu seiner Verarbeitung bereitgestellt werden muss, so dass nicht seitens der Prozessoren ein Engpass entsteht. Gleichzeitig ist man aus Kosten- und Platzgründen daran interessiert, nicht unnötig viele Prozessor-Cores auf dem Chip zu allozieren. Gefragt ist also eine möglichst tiefe Anzahl Prozessoren N, die den Bus unter den herrschenden Bedingungen voll auszulasten vermag.

Zur Abschätzung der notwendigen Rechenleistung müssen erst Annahmen zur Arbitrierung gemacht werden. Wird ein Shared Bandwidth Arbiter eingesetzt (wie beispielsweise ein Weighted Round Robin), so kann die Arbitrierung als die Aufteilung eines konstanten Busintervalls aufgefasst werden. Bei Vollauslastung teilen sich Brücke und Prozessoren die verfügbare Bandbreite entsprechend ihrer relativen Buslast. Auf ein Round-Robin-Intervall bezogen bedeutet dies eine Aufteilung in N Abschnitte der Länge t_{bus} zugunsten der Prozessoren sowie in einen einzelnen Abschnitt der Länge τ_{bridge} zugunsten der Brücke. Um dem Bandbreitenbedarf der Masters gerecht zu werden, muss die Aufteilung folgende Bedingung erfüllen:

$$\frac{\tau_{bridge}}{N\, t_{bus}} = \frac{t_{bridge}}{r\, t_{bus}} \qquad \text{bzw.} \qquad \tau_{bridge} = \frac{N\, t_{bridge}}{r}. \tag{3.5}$$

Geht man nun davon aus, dass alle Masters ihren Busanteil ausschöpfen, so kann für die Prozessoren die mittlere Request-Wartezeit t_{wait} angegeben werden mit

$$t_{wait} = \frac{1}{2}(N\, t_{bus} + \tau_{bridge}) = \frac{1}{2}(N\, t_{bus} + \frac{N\, t_{bridge}}{r}). \tag{3.6}$$

Betrachten wir nun den zu erwartenden Paketdurchsatz D_{proc} beim permanenten Einsatz von N Prozessoren. Dieser beträgt

$$D_{proc} = \frac{N}{t_{appl} + r\,(t_{bus} + t_{wait})}. \tag{3.7}$$

Setzt man 3.6 in 3.7 ein, so ergibt sich schliesslich

$$D_{proc} = \frac{N}{(t_{appl} + r\,t_{bus}) + N \cdot \frac{1}{2}(t_{bridge} + r\,t_{bus})}. \tag{3.8}$$

Damit kann für eine gegebene Anzahl Prozessoren deren mittlere Verarbeitungskapazität abgeschätzt werden. Um nun einen andauernden Engpass seitens der Recheneinheiten zu verhindern, muss $D_{proc} \geq D_{max}$ gelten, also (unter Einbezug von 3.4)

$$\frac{N}{(t_{appl} + r\,t_{bus}) + N \cdot \frac{1}{2}(t_{bridge} + r\,t_{bus})} \geq \frac{1}{t_{bridge} + r\,t_{bus}}. \tag{3.9}$$

Aufgelöst nach N erhält man

$$N \geq 2 \cdot \frac{t_{appl} + r\,t_{bus}}{r\,t_{bus} + t_{bridge}}. \tag{3.10}$$

Bei der Dimensionierung des Netzwerkprozessors spielen Faktoren wie die Taktfrequenz oder der Umfang des Cachespeichers der Recheneinheiten unter Umständen eine entscheidende Rolle. Die Wahl der Taktfrequenz hat einen direkten Einfluss auf die Zeit t_{appl}, welche ein Prozessor zur Verarbeitung eines Pakets benötigt. So hängt die Anzahl benötigter Prozessoren linear von t_{appl} ab. Die Cachegrösse hingegen prägt die Anzahl Buszugriffe während der Paketverarbeitung. Aus Gleichung 3.4 geht hervor, dass eine grosse Anzahl Requests - resultierend aus einem knapp dimensionierten Cachespeicher - den maximalen Paketdurchsatz des Netzwerkprozessors erheblich verringern kann. Bezieht man in diesen Evaluationsprozess weitere Faktoren wie die Kostenfrage, die Leistungsaufnahme, die Chipgrösse mit ein, so kommt man zu einer Entwurfsraumexploration im Kleinen.

Die gewonnenen Erkenntnisse erweisen sich als sehr hilfreich für die Wahl geeigneter Einstellungen für die Simulation der betrachteten Netzwerkprozessor-Architektur (siehe Abschnitte 3.4 und 3.5).

3.3.4 Warteschlangen und Scheduler

Die drei im Abschnitt 3.3.2 beschriebenen Etappen entsprechen in ihrer Hardware-Realisierung drei unterschiedlichen Funktionsblöcken (sofern man den Empfangs- und den Sendekanal der Brücke als zwei unabhängige Einheiten betrachtet). Die Kontrolle über ein Paket wird von der Receive-Einheit

der Brücke an den Dispatcher weitergereicht, welcher seinerseits nach Beendigung der Verarbeitung das Paket zur Übertragung an die Transmit-Einheit der Brücke übergibt. Am Eingang der drei Blöcke können sich bei fluktuierendem Paketaufkommen jeweils Warteschlangen bilden, die es zu verwalten gilt. Mit dieser Aufgabe werden die drei Scheduler betraut, welche im folgenden als Input-, Prozessor- und Output-Scheduler bezeichnet werden und auch in Abbildung 3.2 ersichtlich sind.

Der Scheduler beeinflusst das Geschehen insofern, als dass er die Reihenfolge bestimmt, in welcher die in der Warteschlange präsenten Pakete an die nächste Instanz weitergereicht werden. Das Problem des Scheduling ist mit demjenigen der Arbitrierung eng verwandt, handelt es sich doch in beiden Fällen um die Verwaltung des Zugriffs auf eine knappe gemeinsam genutzte Ressource. Entsprechend kommen ähnliche Algorithmen zum Einsatz – beispielsweise Lösungen basierend auf der Paketpriorität oder solche, welche die zur Verfügung stehende Kapazität in fairer Weise unter den Paketströmen aufteilen. Im Zusammenhang mit QoS-Fragen bei der Paketübermittlung über mehrere Netzknoten, wie dies im Internet tagtäglich milliardenfach geschieht, wurden die theoretischen und praktische Aspekte des Schedulings in den vergangenen Jahren ausführlich ergründet [7, 8, 9]. Im folgenden wird sich zeigen, dass das Scheduling im betrachteten Netzwerkprozessor eine eminent wichtige Rolle spielt. Es werden hier jedoch (mit einer Ausnahme) bloss einfache Scheduler eingesetzt, so dass sich eine tiefere Betrachtung des Schedulings nicht aufdrängt.

Oft wird im Sprachgebrauch in den Begriff des Schedulings die Arbitrierung miteingeschlossen, quasi als eine spezielle Art des Schedulings auf der Ebene eines Busses. Der besseren Verständlichkeit zuliebe wird in dieser Arbeit *Scheduling* ausschliesslich zur Bezeichnung der Verwaltung von Paketwarteschlangen benutzt, während das Scheduling auf Busebene immer *Arbitrierung* genannt wird.

3.3.5 Bedeutung von Arbitrierung und Scheduling

In der besprochenen Rechnerarchitektur gehen Arbitrierung und Scheduling Hand in Hand. Es liegt eine Kaskadierung von Prozessen vor, denen jeweils eine Warteschlange vorausgeht und deren Laufzeit vom Arbiter beeinflusst wird. Dies bedeutet, dass sich die verwendeten Scheduling-Disziplinen in der Regel unmittelbar auf die Arbitrierung auswirken, während umgekehrt das Paketaufkommen in den Warteschlangen stark von dem praktizierten Arbitrierungsschema abhängen kann. Die beim Durchlauf des Netzwerkprozessors entstehende Paketverzögerung ist im Falle von momentaner Überlastung weitgehend durch die kumulierten Verzögerungen beim Scheduling

und bei der Arbitrierung begründet.

Die starke gegenseitige Abhängigkeit verlangt nach einem koordinierten Lösungsansatz, der sowohl Scheduling als auch Arbitrierung miteinbezieht. Im Falle einer vorübergehenden Überlastung des Netzwerkprozessors, verursacht durch ungleichmässig eintreffende Pakete, entsteht momentan eine Ressourcenknappheit, die durch adäquates Vorgehen in Sachen Arbitrierung und Scheduling möglichst effizient gemeistert werden sollte. Zu vermeiden sind insbesondere Speicherüberläufe sowie schlechte Nutzung der Rechenleistung. Ersteres kann zum Verlust von Paketen führen, während letzteres ein ungünstiges Kosten-Nutzen-Verhältnis entstehen lässt und der Paketverzögerung abträglich ist. Eine geschickte Koordination von Arbitrierung und Scheduling ermöglicht

1. Verteilung der Last entsprechend der zur Verfügung stehenden Ressourcen

2. Abschätzung der beim Durchlaufen der drei Etappen (siehe Abschnitt 3.3.2) erfahrenen Verzögerungen.

Gelingt dies, so können äusserst nützliche Voraussagen gemacht werden über

1. die sinnvolle Dimensionierung der im Netzwerkprozessor benötigten Ressourcen (Rechenleistung, Speicher, Kommunikationseinheiten)

2. die Leistung des Netzwerkprozessors im Hinblick auf die Verarbeitung von QoS-relevanten Paketströmen.

Wie man sich diesbezügliche Überlegungen zunutze machen kann, zeigen die Simulationen zweier Szenarien weiter unten in diesem Kapitel.

3.3.5.1 Paketfluss ohne QoS-Relevanz

In einem ersten Schritt soll kurz auf die Bedeutung von Arbitrierung und Scheduling eingegangen werden für den Fall, dass sämtliche Pakete der Input-Trace derselben Klasse von Best-effort-Verkehr angehören. Sie unterscheiden sich somit bloss in ihrer Grösse und Ankunftszeit. Hier sind in erster Linie die im vorhergehenden Abschnitt jeweils unter Punkt 1 genannten Tatsachen von Belang. Ziel des Designers muss es also sein, Scheduling und Arbitrierung im Netzwerkprozessor derart aufeinander abzustimmen, dass ein Minimum an Ressourcen (kumulierte Prozessorleistung und Speicher) vonnöten ist, um die Pakete mit minimaler Verzögerung zu verarbeiten. Vorraussetzung für dieses Vorgehen ist eine möglichst exakte Kenntnis des Verhaltens der Komponenten des Netzwerkprozessors bei der Abarbeitung der Pakete. D.h. es sollte bekannt sein:

- der genaue Datenpfad, d.h. welche Teile der Pakete in welcher Abfolge welche Instanzen durchlaufen,
- die Prozessabhängigkeiten, d.h. welche Prozesse in welcher Hinsicht vom Abschluss vorhergehender Prozesse abhängig sind (Taskgraph),
- die PLB-Last, welche durch das Hineinschreiben eines Pakets in den Speicher hervorgerufen wird,
- die PLB-Last, welche durch das Herauslesen eines Pakets aus dem Speicher hervorgerufen wird,
- die ungefähre PLB-Last, welche bei der Verarbeitung des Pakets durch einen Prozessor hervorgerufen wird (Da diese meist nicht deterministisch ist, sollte zumindest ein Mittelwert sowie eine obere Grenze bekannt sein.),
- der zusätzlich entstehende PLB-Verkehr verursacht durch die Übermittlung von Pufferdeskriptoren,
- die Zeit, welche (unabhängig von den Buszugriffen) für die Paketverarbeitung durch den Prozessor benötigt wird,
- die Kapazität der Funktionsblöcke (Übertragungsgeschwindigkeit der Brücke, kumulierte Prozessorleistung),
- die Eigenschaften der zu erwartenden Paketflows in Bezug auf Paketgrösse und Paketabstand.

Bei der Wahl des Arbiters ist es von zentraler Bedeutung, dass es im Falle hoher Busauslastung nicht zur Bildung unvorteilhafter Warteschlangen kommt. Eine angepasste Arbitrierung kann zwar nicht den Bus vor einer Ueberlastung bewahren, jedoch kann sie dafür sorgen, dass in einem solchen Fall die knappe Busbandbreite derart unter den konkurrierenden Masters aufgeteilt wird, dass keine Blockaden entstehen und Speicherüberläufe aufgrund überquellender Warteschlangen möglichst vermieden werden. Wir werden auf diese Problematik bei der Betrachtung von Simulationsresultaten zurückkommen.

3.3.5.2 Paketfluss mit QoS-Relevanz

Da die Arbitrierung bei starker Busauslastung einen nennenswerten Einfluss auf die Verzögerung der Pakete beim Durchlaufen des Netzwerkprozessors haben kann, ist sie im Falle von Paketströmen mit bestimmten Dienstgüteanforderungen ins Design miteinzubeziehen. Soll beispielsweise eine vorgegebene Deadline für die maximale Zeit eines Pakets im Prozessor eingehalten

werden, so muss der Arbiter Garantien hinsichtlich Latenz bzw. Bandbreite abgeben können. Gefragt sind also Algorithmen, die über eine begrenzte Latenz verfügen und die eine einstellbare Aufteilung der Bus-Bandbreite anbieten. Ist dies gegeben, so kann eine Worst-case-Abschätzung bezüglich der durch den Bus induzierten Verzögerung vorgenommen werden, welche in die QoS-Charakteristiken des gesamten Systems einfliesst. Diese Art des *passiven* Beitrags zur Wahrung der QoS-Vorgaben wird durch das weiter unten betrachtete Szenario II illustriert.

Der Einfluss des Arbiters auf die QoS kann aber auch *aktive* Formen annehmen. Allgemein gesagt wird sich eine unterschiedliche Behandlung von Bus-Requests, deren zugehörige Pakete verschiedenen QoS-Klassen angehören, auf die Einhaltung der QoS-Anforderungen auswirken. Voraussetzung für einen Einbezug der Dienstklasse in die Arbitierung ist selbstverständlich die Fähigkeit des Arbiters, Requests verschiedener Klassen zu erkennen und diese im Rahmen seiner Möglichkeiten und Vorgaben entsprechend zu behandeln. Ist dies gegeben, so kann die Differenzierung auf die Ebene der Arbitrierung ausgedehnt werden und bleibt fortan nicht aufs Scheduling beschränkt. Somit entsteht eine Art *verteiltes Scheduling*, indem der Arbiter im Verlaufe der Paketverarbeitung bei jedem Request die Rolle eines kleinen Schedulers spielt. Hier stellt sich die Frage, ob denn die Aufgabe des prozessorinternen Schedulings gänzlich oder zum Teil an den Arbiter delegiert werden kann und soll. Zwei wichtige Feststellungen lassen hier eine klare Antwort zu:

- Beim herkömmlichen Paket-Scheduling wird aus einer Menge von Paketen ein einzelnes Paket zur Verarbeitung selektioniert. Dies geht auf Kosten der übrigen Pakete, welche bis auf weiteres im Speicher verbleiben. Werden hingegen durch den Arbiter gewisse Pakete zugunsten anderer bevorzugt, so resultiert dies in einer Verzögerung der Verarbeitung seitens der benachteiligten Pakete. Das Resultat ist die Blockierung einer Verarbeitungseinheit, was einer Verschwendung von Rechenleistung gleichkommt. Es macht wenig Sinn, Systemkomponenten zu allozieren, um sie im Fall von Überlastung zu blockieren. Dieser Punkt spricht klar gegen ein Scheduling auf Arbiterebene.

- Das sogenannte verteilte Scheduling durch den Arbiter hat es an sich, dass es aus der Sicht eines einzelnen Pakets immer dann stattfindet, wenn über die Behandlung einer seiner Busanfragen entschieden wird. Da nun aber die Anzahl ebendieser Anfragen von Paket zu Paket stark und in unvorhersehbarer Weise variieren kann, ist der Einfluss dieses Schedulings auf die Paketverzögerung stark von unkontrollierbaren Faktoren abhängig. Diese Tatsache macht es äusserst schwierig, wenn nicht unmöglich, die Einhaltung von QoS-Vorgaben im Netz-

werkprozessor sicherzustellen sowie das Scheduling in Bezug auf die verschiedenen durchlaufenen Instanzen zu koordinieren. Eine Lösung, die den Arbiter als intelligentes aktives Element ins Paketscheduling des Netzwerkprozessors miteinbezieht, wäre mit grosser Wahrscheinlichkeit kaum praktisch implementierbar, muss doch ein Arbiter äusserst schnell (verglichen mit einem Scheduler, der Paketwarteschlangen verwaltet) sein und sollte wenn möglich den zusätzlichen Bedarf an Steuerleitungen gering halten.

Die aufgeführten Punkte sprechen klar gegen ein eigentliches Scheduling auf Arbiterebene. Nichtsdestotrotz kann die Wahl eines Arbiters, der Pakete anhand ihrer QoS-Klasse unterschiedlich behandelt, von Interesse sein. Das folgende Szenario I soll dies illustrieren.

3.4 Szenario I: Minimierung der Paketverzögerung

3.4.1 Ausgangslage

In diesem Szenario wird angenommen, dass eine Trace vorliegt, die aus Paketen zweier verschiedener Serviceklassen besteht. Zum einen handelt es sich hierbei um Best-effort-Pakete, welche keine besonderen Ansprüche an die Quality of Service stellen, zum anderen um Pakete höherer Priorität, welche bevorzugt behandelt werden sollen. Letztere haben den Anspruch, mit einer möglichst niedrigen Verzögerung den Netzwerkprozessor passieren zu können, wobei in keiner Weise auf die ersteren Rücksicht genommen werden muss. Die Differenzierung kann mittels geeigneter Einstellung der drei Scheduler im Netzwerkprozessor erreicht werden. Kommt es dabei nicht nur zu einer gelegentlichen Überlastung der Verarbeitungseinheiten, sondern auch zu einer ebensolchen des PLB, so ist zu erwarten, dass dessen Arbitrierung einen spürbaren Einfluss auf das Geschehen nimmt. Es soll nun untersucht werden, inwiefern sich verschiedene Arbitrierungsschemen auf das Leistungsverhalten des Netzwerkprozessors auswirken. Weiter soll festgestellt werden, ob sich die Verwendung eines QoS-orientierten Arbiters lohnt.

3.4.2 Scheduling

Auf der Ebene der Scheduler sieht die Situation wie folgt aus:

- **Input Scheduling.** Als Input Scheduler wird ein gewöhnlicher FIFO-Mechanismus verwendet. Damit erübrigt sich ein vorzeitiges Header-Parsing im EMAC bzw. in der Brücke, welches einen zusätzlichen Aufwand bedeuten würde. Da ausserdem im vorliegenden Fall der Weg der Paketdaten über den PLB in den Speicher aufgrund der geringeren Anzahl Write-Requests keinen Engpass darstellt, ist nicht mit der Entstehung einer Warteschlange zu rechnen. Ein Prioritäts-basierter Scheduler würde also keinen Gewinn bringen – im Gegenteil: er würde unnötig zusätzliche Kosten verursachen.

- **Processor Scheduling.** Im Falle erhöhten Verkehrsaufkommens ist zu erwarten, dass die Prozessorleistung kurzfristig gesehen nicht immer ausreicht, um eine Warteschlange im Dispatcher zu verhindern. Folglich ist dafür zu sorgen, dass priorisierte Pakete eine möglichst geringe Verzögerung erfahren, indem sie stets den nächsten freiwerdenden Prozessor zugeteilt erhalten. Der Scheduler kann also nach dem Prinzip zweier getrennter FIFO-Warteschlangen aufgebaut werden – eine für die priorisierten Pakete, eine für den Best-effort-Verkehr. Best-effort-Pakete kriegen nur dann einen Prozessor, wenn keine priorisierten Pakete warten.

- **Output Scheduling.** Da die Prozessoren und der Sendepfad der Brücke miteinander in Konkurrenz um den PLB-Read-Bus stehen, kann nicht immer davon ausgegangen werden, dass genügend Bandbreite für das Auslesen der Pakete aus der PLB-Umgebung zur Verfügung steht. Entsprechend können sich Pakete aufstauen, deren Verarbeitung an sich abgeschlossen ist, die jedoch noch von der Brücke an den Ausgangs-EMAC übermittelt werden müssen. Auch hier macht also das Konzept Sinn, welches für das Processor Scheduling aufgezeigt wurde.

3.4.3 Arbitrierung

Was die Arbitrierung betrifft, so soll hier ein Vergleich verschiedener Ansätze stattfinden. Dabei werden die folgenden Schemen berücksichtigt:

- **Random Arbiter.** Als erstes soll eine Referenz für die nachfolgenden Arbiter geschaffen werden, indem ein fiktiver Zufallsarbiter über die Verwendung des Busses entscheidet. Dieser wählt bei freiem Bus aus den vorliegenden Anfragen eine beliebige aus, ohne auf die Master- oder Paketidentität Rücksicht zu nehmen. Die resultierende Referenzsimulation wird als Massstab für zwei besser angepasste Arbitrierungen dienen.

- **Master Priority Arbiter.** Als zweites soll ein Arbiter in Erwägung gezogen werden, der auf die gegenseitigen Abhängigkeiten zwischen den Funktionsblöcken eingeht, wie sie vom Ablauf der Verarbeitung vorgegeben sind. Dies kann in relativ einfacher Form über die Abstimmung der Master-Prioritäten beim Buszugriff vollzogen werden. Bei der Wahl der Prioritäten wird darauf geachtet, dass die Ausgangsseite favorisiert wird, so dass sich allfällige Warteschlangen bereits vor der Verarbeitung bilden. Die Ausgangsverzögerung kann dadurch minim gehalten werden, während die vom Transmit-Pfad nicht genutzte Bus-Bandbreite den Prozessoren zur Verfügung steht. Das System reguliert sich damit selbst und verhindert Ineffizienz aufgrund von Blockaden. Im konkreten Beispiel unserer Netzwerkprozessor-Architektur wirkt sich die Arbitrierung vorwiegend auf den Verkehr auf dem Read-Bus aus – dies aufgrund des geringeren Request-Aufkommens auf dem Write-Bus. Letzterer wird hauptsächlich durch die Brücke beim Transfer der Pakete in den Speicher beansprucht. Da dieser mit einem Pufferdeskriptor-Read beginnt, sollte die Priorität des Pufferdeskriptoren-Verwalters auf dem Read-Bus ausnahmsweise erhöht werden, um nicht den Empfang des Pakets unnötig zu verzögern. Man erhält also folgende Prioritätsabfolgen:

Paket-Trace:	
Paketgrösse [bytes] (Verteilung siehe Abb. 3.1)	64 ... 1 500
Mittlerer Interframe Gap [bytes] (exp. verteilt)	1 000
Empfangsrate [Mbyte/s]	1 250
Anteil priorisierter Pakete	30 %
Processor Local Bus:	
Taktfrequenz [MHz]	400
Busbreite [bit]	128
Prozessoren:	
Anzahl Prozessoren	6
Taktfrequenz [MHz]	400
Zyklen pro Instruktion	0.7
Grösse des Instruktionscache [kbytes]	2
Grösse des Datencache [kbytes]	2
Ausgeführte Anwendungen (siehe [26]): RTR, FRAG, DRR, TCP	

Tabelle 3.1: Einstellungen für die Simulationen.

Read-Bus: Pufferdeskriptoren-Verwalter → Brücke → Prozessoren
Write-Bus: Pufferdeskriptoren-Verwalter → Brücke → Prozessoren

Die Reihenfolgen können anhand der Abbildung 3.2 aus dem Request-Muster abgeleitet werden.

- **Packet Priority Arbiter.** Als drittes geht es um die Beantwortung der Frage, ob eine Ausdehnung der QoS-bezogenen Behandlung auf die Arbiterebene Sinn macht, indem sie zu einer relevanten Verkürzung der Paketverzögerung führt. Dafür wird ein Arbiter eingesetzt, der die Requests anhand der Priorität der mit ihnen verbundenen Pakete arbitriert. Die Priorität wird dem Arbiter vom jeweiligen Master entsprechend des gerade in Verarbeitung befindlichen Pakets signalisiert. So gesehen kann man auch von einer dynamisch festgelegten Masterpriorität sprechen. Liegen gleichzeitig mehrere Requests der gleichen Priorität vor, so fällt der Arbiter seinen Entscheid basierend auf der statischen Masterpriorität, wie sie der zweite Arbiter anwendet.

3.4.4 Simulationsergebnisse

Das vorgestellte Szenario wurde mittels der in Abschnitt 2.5 beschriebenen Matlab-Simulation untersucht. Die Resultate werden anhand eines Ausschnitts bestehend aus 500 aufeinanderfolgenden Paketen erläutert. Die Parameter sind in Tabelle 3.1 ersichtlich.

	Arbiter:	**Random**	**Master Prio.**	**Packet Prio.**
Mittlere	RT	12.8	13.6	7.5
Verzögerung [μs]	BE	156.7	60.3	67.6
Maximale	RT	20.4	27.1	12.3
Verzögerung [μs]	BE	227.4	76.9	84.8
Summe aller	RT	2 204.6	2 335.4	1297.4
Verzögerungen [μs]	BE	51 381.7	19 784.5	22 177.0
	total	53 586.2	22 119.9	23 474.4

Tabelle 3.2: Zusammenfassung der Resultate für Szenario I (RT = Real-time, BE = Best-effort).

Die Abbildungen 3.3 bis 3.5 zeigen oben die Verzögerungen der einzelnen Pakete sowie die jeweils resultierenden Histogramme. Der dunkel hinterlegte Bereich am unteren Rand der Graphik oben links repräsentiert die jeweiligen Zeiten, welche die Pakete im Prozessor verbringen würden, wenn sie die Ressourcen konkurrenzlos nutzen könnten und es folglich zu keinen Verzögerungen bei Scheduling und Arbitrierung käme. Die Skalen der Histogramme sind identisch gewählt, so dass ein direkter Vergleich leicht fällt. Die zwei unteren Graphiken illustrieren den Zeitverlust gegenüber der erwähnten Minimalzeit, welchen die priorisierten respektive die Best-effort-Pakete erfahren. Daraus lassen sich wertvolle Schlüsse im Hinblick auf die für die Paketverzögerung verantwortlichen Engpässe ziehen.

Der Paketstrom ist für alle Simulationen identisch, womit die Anzahl verarbeiteter Pakete pro Zeiteinheit für alle betrachteten Fälle gleich ist. Da die Busauslastung infolge der hohen Empfangsrate sehr hoch wird, war zu erwarten, dass sich unterschiedliche Arbitrierung wesentlich auf die gemessenen Paketverzögerungen auswirken. Die Paketverzögerung soll hier als die Zeitspanne vom Eintreffen des Pakets in der Brücke (Receive-Pfad) bis zum letzten Status-Write-Buszugriff der Brücke (Transmit-Pfad) verstanden werden.

In Tabelle 3.2 sind die wichtigsten Kennziffern der Simulationsresultate für die drei Arbiter eingetragen.

3.4.4.1 Random Arbiter

Der Random Arbiter hat die Eigenschaft, dass er alle anfallenden Busanfragen gleich behandelt, sprich keine Rücksicht auf Master- oder Paket-Priorität nimmt. In Abbildung 3.3 widerspiegelt sich dies in der gleichmässigen Ver-

teilung der Zeitverluste über alle Stufen, in denen der Arbiter für allfällige Verzögerungen verantworlich ist (Receive, Processing, Transmit). Die Bus-Wartezeiten des Empfangspfades fallen vergleichsweise gering aus, da sie den schwach ausgelasteten Write-Bus betreffen.

Für die Best-effort-Pakete fällt die Bevorzugung durch Input-, Prozessor- und Output-Scheduler weg, womit lange Wartezeiten vor den entsprechenden Einheiten entstehen. Als ungünstig stellt sich heraus, dass die zur Übertragung bereiten, verarbeiteten Best-effort-Pakete lange im Speicher ausharren müssen, da die Brücke kein Mittel hat, sich auf Kosten der Prozessoren die erwünschte Bandbreite zu sichern. Dieser Effekt führt zu beachtlichen Zeitverlusten in der Output-Warteschlange.

Was die priorisierten Pakete angeht, so kann der Arbiter als erfolgreich bezeichnet werden, ist er doch imstande, eine Blockierung einzelner Pakete zu verhindern. Dies verdankt er einer quasi-begrenzten Latenz[3] sowie seiner Eigenschaft, die Busbandbreite im Mittel proportional zum Bedarf der einzelnen Masters aufzuteilen.

3.4.4.2 Master Priority Arbiter

Die Feststellung, dass Pakete beim Einsatz eines Random Arbiters viel Zeit damit verbringen, darauf zu warten, dass sie aus dem Speicher gelesen werden, legt die Verwendung einer Arbitrierung nahe, die diesem Engpass abhilft. Gefragt ist ein Schema, welches der Brücke auf dem Read-Bus die Priorität über die Prozessoren erteilt. Mit dem oben beschriebenen Master Priority Arbiter werden Pakete nach ihrer Verarbeitung umgehend aus dem Speicher gelesen, ohne zusätzlichen Zeitverlust in einer Output-Warteschlange. Da jedoch nicht mehr Busbandbreite vorhanden ist als im vorhergehend besprochenen Fall, muss dies auf Kosten der Prozessoren geschehen. In Abbildung 3.4 (unten) ist ersichtlich, dass der vom Processing herrührende Zeitverlust zugenommen hat, was sich (vor allem für den Best-effort-Verkehr) in einer grösseren Prozessor-Wartezeit niederschlägt. Die Zunahme der Wartezeit für Best-effort-Pakete erreicht aber bei weitem nicht das Ausmass der Zeitverluste in der Output-Warteschlange des vorigen Beispiels.

Vergleicht man die Ergebnisse der priorisierten Pakete, so sind keine wesentlichen Unterschiede in Bezug auf die Verzögerung und deren Variabilität auszumachen. Wie der Random Arbiter hält also auch der Master Priority Arbiter die Differenzierung der Verkehrsklassen aufrecht, obwohl er davon keine Kenntnis nimmt. Die Frage, inwiefern ein geeigneter Arbiter die

[3]Streng genommen ist die Latenz des Random Arbiters nicht begrenzt, praktisch sind jedoch sehr hohe Wartezeiten äusserst unwahrscheinlich.

Verzögerung der Pakete zusätzlich reduzieren kann, soll mit der nächsten Simulation beantwortet werden.

3.4.4.3 Packet Priority Arbiter

Beim Betrachten der Abbildung 3.5 stellt man fest, dass mit einer auf die Paketpriorität Rücksicht nehmenden Arbitrierung eine signifikant niedrigere Paketverzögerung seitens der priorisierten Pakete zu erreichen ist. Trotz der hohen Last kommt ein grosser Teil der davon nahe an die theoretische Mindestverzögerung heran. Dadurch dass die favorisierte Paketklasse nun auf Scheduler- wie auch auf Arbiter-Ebene den Best-effort-Paketen vorgezogen wird, kann sie (praktisch[4]) unbeeinflusst von diesen verarbeitet werden. Nur für sich betrachtet, lasten die priorisierten Pakete das System bei weitem nicht aus, folglich führt ihre gegenseitige Interferenz nicht zu namhaften Engpässen.

Die Folgen für den Best-effort-Verkehr sind gering, ändert sich doch dessen Paketverzögerung relativ gesehen nicht wesentlich. Da der Arbiter bei gleicher Paketpriorität auf die Masterpriorität nach dem Muster des vorhergehend besprochenen Arbiters zurückgreift, spielt unter den Best-effort-Paketen derselbe Mechanismus wie zuvor. Dementsprechend ist der Löwenanteil am Zeitverlust dieser Paketklasse ebenfalls in der Prozessor-Warteschlange vorzufinden.

3.4.5 Schlussfolgerungen

In diesem Szenario, das die Minimierung der Paketverzögerung für eine priorisierte Paketklasse bei gleichzeitig vorhandenem Best-effort-Verkehr zum Ziel hatte, wurde der Einfluss der Arbitrierung auf die Leistung des Netzwerkprozessors anhand der resultierenden Paketverzögerung untersucht. Es zeigte sich dabei, dass ein Arbiter, der auf die Abhängigkeiten im Ablauf der Paketverarbeitung abgestimmt ist, für die Best-effort-Pakete deutlich bessere Resultate erreicht als ein nach dem Zufallsprinzip agierender Arbiter. Durch Favorisieren des am Sendepfad beteiligten Masters kann eine Ausgangswarteschlange vermieden werden, womit dieses potentielle zusätzliche Verzögerungsglied entfällt. Die den Prozessoren zur Verfügung stehende Busbandbreite wird durch die Bevorzugung der Brücke nur wenig geschmälert und führt nicht zu einem Anstieg der Verzögerung im Prozessor, der vergleichbar wäre mit den Auswirkungen einer Ausgangswarteschlange. Priori-

[4] Der Vollständigkeit halber muss hier angemerkt werden, dass die Anwesenheit von Best-effort-Verkehr von den priorisierten Paketen nicht gänzlich unbemerkt bleibt; so kann beispielsweise ein Best-effort-Paket, das gerade aus dem Speicher gelesen wird, die Übertragung eines soeben fertig verarbeiteten priorisierten Pakets verzögern.

sierte Pakete erfahren keine wesentliche Verbesserung bzw. Verschlechterung in Bezug auf ihre Verzögerung. Ihretwegen ist eine Anpassung der Arbitrierung nicht zwingend.

Eine Beschleunigung der priorisierten Pakete ist hingegen dadurch realisierbar, dass die Ebene der Arbiters in die Differenzierung der Verkehrsklassen miteinbezogen wird. Es zeigte sich, dass ein auf die Paketpriorität achtender Arbiter eine deutliche Leistungssteigerung bzw. Minimierung der Verzögerung für die priorisierten Pakete ermöglicht. Somit kann die Arbitrierungspraxis einen merklichen Beitrag zur Einhaltung respektive zur Verbesserung der Quality of Service leisten.

Vergleicht man die Summen der Paketverzögerungen in Tabelle 3.2, so stellt man fest, dass die Totalwerte für Simulationen mit Prioritäts-orientierter Arbitrierung sehr nahe liegen. Der Random Arbiter liefert jedoch ein stark abweichendes Resultat, begründet durch das Entstehen einer respektablen Output-Warteschlange und der damit verbundenen zusätzlichen Paketverzögerung. Wird hingegen statt eines Master Priority Arbiters ein Packet Priority Arbiter eingesetzt, so findet eine Art Umverteilung der Paketverzögerung zugunsten der priorisierten Pakete statt. Dies steht im Einklang mit einem aus der Scheduling-Theorie stammenden Erhaltungssatz [10], der besagt, dass für Scheduling-Methoden, die work-conserving sind, die aufsummierten Paketverzögerungen einen konstanten Wert annehmen. Konkret heisst dies, dass ein konstanter Wert resultiert, wenn die von n einzelnen Paketflüssen herrührenden Verkehrsanteile ρ_i jeweils mit ihren Wartezeiten q_i multipliziert und danach addiert werden:

$$\sum_{i=1}^{n} \rho_i \; q_i = \text{konst.} \tag{3.11}$$

Die ρ_i sind das Produkt aus der Ankunftsrate des Paketflusses und der Verarbeitungszeit eines Pakets. Der Satz kann auch auf die Arbiterebene übertragen werden, wo die Request-Wartezeiten priorisierter Pakete verkürzt werden und daraus eine zusätzliche Verzögerung seitens der Best-effort-Pakete resultiert. Dieser Effekt schlägt sich dann in der gesamten Verzögerung der betroffenen Pakete nieder. (Die Summen über alle Pakete sind hier nicht exakt gleich, da die von den Prozessoren generierten Requestmuster von Simulation zu Simulation leicht variieren.)

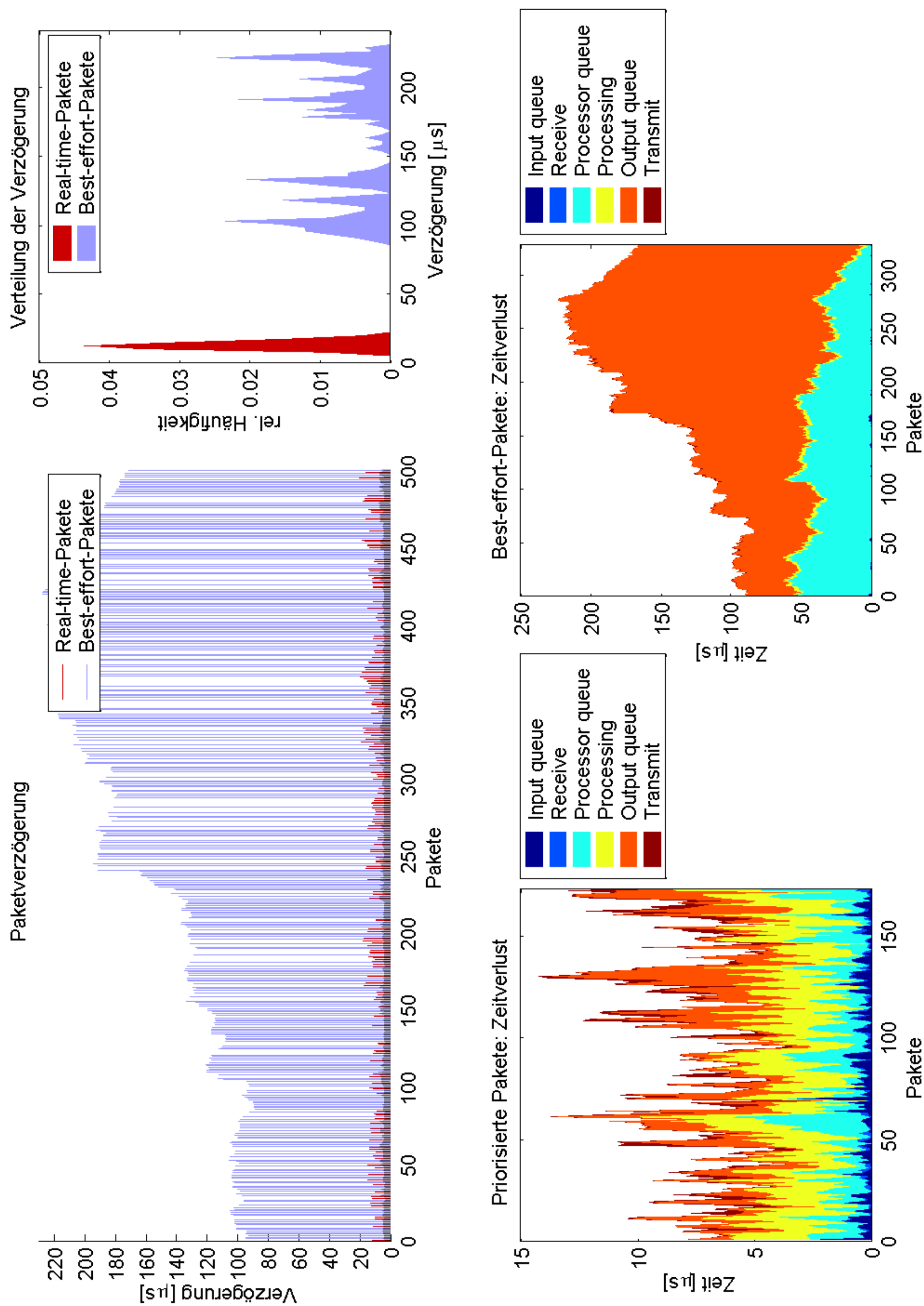

Abbildung 3.3: Simulationsresultate für den Random Arbiter.

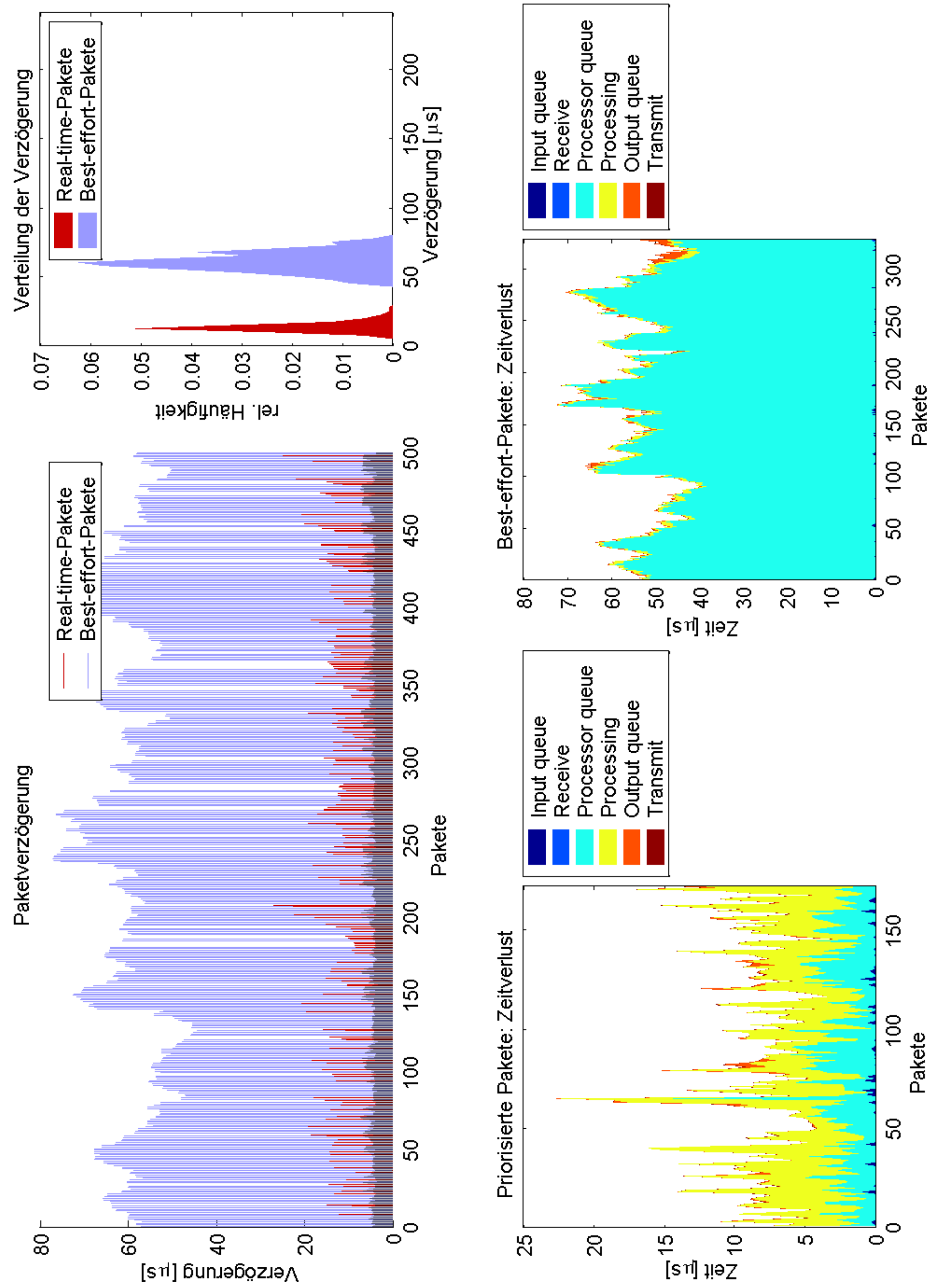

Abbildung 3.4: Simulationsresultate für den Master Priority Arbiter.

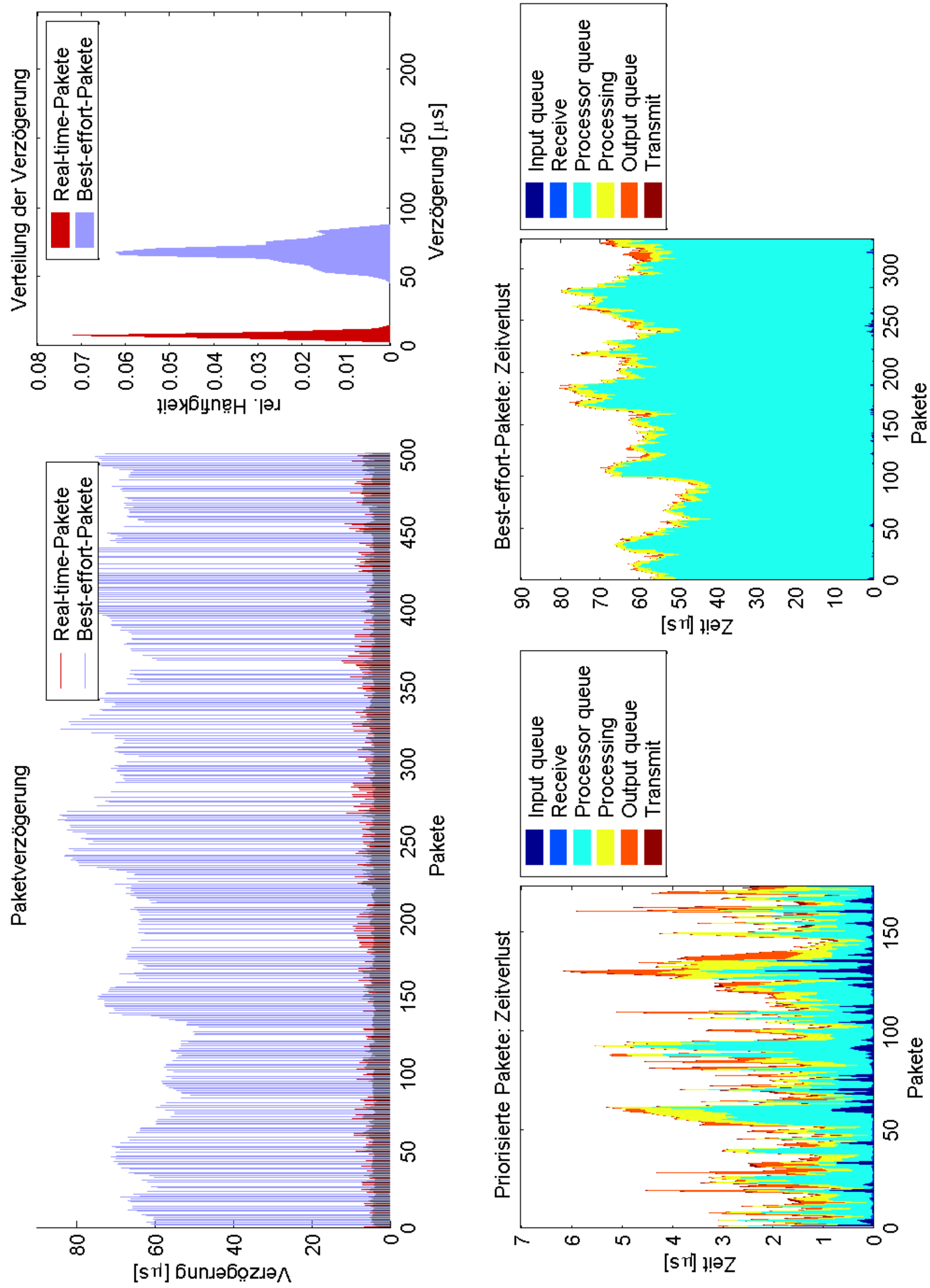

Abbildung 3.5: Simulationsresultate für den Packet Priority Arbiter.

3.5 Szenario II: Best-effort-Optimierung bei gleichzeitigem Real-time-Verkehr

3.5.1 Ausgangslage

Wenn es um Dienstgüte im Netzwerk geht, ist die Einhaltung von Deadlines, also die Vermeidung des Überschreitens einer vorgegebenen Maximalverzögerung, ein oft geäusserter Anspruch. In diesem zweiten Szenario wollen wir deshalb davon ausgehen, dass die priorisierten Pakete, wie sie im ersten Szenario vorkamen, mit einer fixen Deadline versehen sind. Diese besagt nach welcher Zeit ein Paket den Prozessor verlassen haben muss, um nicht die QoS-Vorgaben zu verletzen. Neben dieser Real-time-Paketklasse existiert weiterhin ein erheblicher Verkehr bestehend aus Best-effort-Paketen, die keine besonderen Anforderungen an die QoS stellen.

Im ersten Szenario wurde gezeigt, dass mittels geeigneter Massnahmen im Bereich Scheduling und Arbitrierung eine vergleichsweise sehr tiefe Verzögerung der priorisierten Pakete erreicht werden kann. Eine derartige Differenzierung gegenüber dem Best-effort-Verkehr ist jedoch in der Realität oftmals nicht vonnöten – vielmehr sind Garantien gefragt, die strikte eingehalten werden. Wenn diese nun übertroffen werden, ist dies zwar begrüssenswert, entspricht aber nicht den Bedürfnissen des Nutzers. So gesehen macht es wenig Sinn, Erwartungen bezüglich einer bestimmten Paketklasse zu übertreffen, wenn dies zuungunsten einer anderen geschieht. Im konkreten Fall bedeutet dies, dass ein wiederholtes deutliches Unterschreiten der Deadline der Real-time-Pakete im überlasteten Netzwerkprozessor die ohnehin grössere Verzögerung der Best-effort-Pakete zusätzlich und vor allem unnötig erhöht. Um diesem Problem abzuhelfen, wird im folgenden eine Kombination aus Scheduling und Arbitrierung vorgestellt, welche die gegebenen Deadlines bestmöglich auszuschöpfen sucht, um damit die Paketverzögerung für Best-effort-Verkehr zu reduzieren. Das gleiche Ziel verfolgt ein Verfahren, welches in [6] vorgeschlagen wird.

3.5.2 Scheduling

Kernstück des vorgeschlagenen Konzepts ist ein Scheduler, der darüber wacht, dass sämtliche Real-time-Pakete ihre Deadline einhalten, gleichzeitig aber darum besorgt ist, den Best-effort-Verkehr wenn immer möglich zu fördern. Ein solcher Scheduler – man könnte ihn einen *Deadline Observing Scheduler* nennen – kennt den Zeitpunkt, an welchem ein sich in seiner Warteschlange befindliches Real-time-Paket spätestens verarbeitet werden muss, um nicht seine Deadline zu verletzen. Zwischen der aktuellen Zeit und diesem Zeitpunkt existiert nun ein Zeitfenster, welches den Best-effort-Paketen überlassen werden kann, um deren Verzögerung zu verringern. Um garantieren

zu können, dass das Zeitfenster nicht überschritten wird, muss eine Abschätzung der Zeit möglich sein, während welcher die Ressource (Prozessor, Brücke) besetzt sein wird. Ist dies der Fall, so wird der Scheduler beim Freiwerden seiner zugehörigen Ressource aus den wartenden Real-time-Paketen die Grösse des Zeitfensters bestimmen, um dann aus den wartenden Best-effort-Paketen eines auszuwählen, dessen Verarbeitung im Rahmen der zur Verfügung stehenden Zeit möglich ist. Findet sich kein geeignetes Best-effort-Paket, so wird das dringendste Real-time-Paket selektioniert.

Zu seiner Realisierung benötigt der Deadline Observing Scheduler eine Funktion, welche ihm erlaubt, für ein Real-time-Paket die Zeit Δt_{RT} abzuschätzen, die es bis zum Verlassen des Netzwerkprozessors maximal verbraucht. Zusammen mit der gegebenen Deadline d leitet sich daraus der späteste Zeitpunkt t_{start} für den Beginn der Verarbeitung des Pakets ab. Weiter muss eine Obergrenze Δt_{res} für die Zeit existieren, für welche die dem Scheduler angegliederte Ressource durch ein Paket (Real-time oder Best-effort) reserviert wird. Diese kann variabel sein, indem sie beispielsweise vom letzten ausgewählten Paket abhängt. Es kann nun ein Algorithmus angegeben werden, der auf eine Warteschlange aus n_{RT} Real-time-Paketen und n_{BE} Best-effort-Paketen angewendet werden kann (Der Buchstabe p und der Index i werden zur Bezeichnung von Real-time-Paketen verwendet, q respektive j entsprechend für Best-effort-Pakete):

- Ordne die n_{RT} Real-time-Pakete p_i $(i = 1 \ldots n_{RT})$ nach absteigender Deadline[5] d_i, so dass $d_1 \geq d_i \geq d_{n_{RT}}$.
- Setze einen Zeitmarker τ auf die Deadline d_1.
- Für jedes i von 1 bis n_{RT}:
  - Bestimme $\Delta t_{RT,i}$ sowie $\Delta t_{res,i}$
  - $t_{start,i} = \min(\tau - \Delta t_{res,i}, d_i - \Delta t_{RT,i})$
  - $\tau = t_{start,i}$
- Berechne das Zeitfenster T zur aktuellen Zeit t: $T = \tau - t$
- Bestimme $\Delta t_{res,j}$ für alle Best-effort-Pakete q_j
- Wähle ein Best-effort-Paket q_j, so dass gilt: $\Delta t_{res,j} \leq T$
- Falls $\Delta t_{res,j} > T$ für alle j: (kein Best-effort-Paket passt ins Zeitfenster)
  - Verarbeite das Real-time-Paket $p_{n_{RT}}$ (früheste Deadline).
- Sonst:
  - Verarbeite das gewählte Best-effort-Paket [6].

Der Algorithmus stellt sicher, dass die in der Warteschlange anwesenden Real-time-Pakete ihre Deadline nicht verpassen, selbst wenn sie aufgrund gegenseitiger Beeinflussung eventuell verfrüht bearbeitet werden müssen. Die

[5] Als Deadline ist hier der absolute Zeitpunkt gemeint, an welchem das Paket den Netzwerkprozessor verlassen haben muss.

[6] Bei der Auswahl des eingeschobenen Best-effort-Pakets kommt quasi ein zweiter Scheduler zum Zug. Hier wird aus den in Frage kommenden Paketen der Einfachheit halber stets dasjenige gewählt, welches den Netzwerkprozessor als erstes erreichte.

obige Befehlsfolge stellt selbstverständlich keinen auf Leistung optimierten Algorithmus dar, diesbezügliche Verbesserungen sowie eine Einschätzung der Implementierbarkeit sind wichtig, konnten jedoch im Rahmen dieser Arbeit nicht mehr unternommen werden.

Wie erwähnt baut der beschriebene Algorithmus auf der Möglichkeit einer Abschätzung von maximal benötigten Verarbeitungszeiten auf. Einerseits muss für die Real-time-Pakete bekannt sein, wieviel Zeit maximal bis zum Verlassen des Netzwerkprozessors vergehen kann. Andererseits muss bekannt sein, nach welcher Zeit die von Scheduler bediente Ressource spätestens wieder frei wird. Ersteres erfordert Kenntnisse über sämtliche nachfolgenden Stufen, die ein Real-time-Paket im Netzwerkprozessor durchläuft. Wird also in unserem konkreten Fall der Deadline Observing Scheduler in den Sendepfad der Brücke integriert, so entspricht Δt_{RT} der Zeit, die für den Transfer der Pakets maximal erforderlich ist. Wird er hingegen vor die Prozessoren geschaltet, so muss zusätzlich zur Transferzeit eine maximale Verarbeitungszeit angegeben werden. Die Zeit Δt_{res}, während der die Ressource besetzt sein wird, entspricht beim Einsatz als Output-Scheduler ebenfalls der Auslesezeit. Vor den N Prozessoren ist t_{res} im schlimmsten Fall gleich der Verarbeitungszeit eines einzelnen Pakets (wenn gerade alle Prozessoren gleichzeitig mit der Bearbeitung eines Pakets begonnen haben), im Mittel allerdings bloss ein Bruchteil davon.

Es stellt sich nun die Frage, inwiefern die genannten Zeiten für das vorliegende System bekannt sind und wovon sie abhängen. Steht der Brücke auf dem Read-Bus eine garantierte Bandbreite zur Verfügung, so ist deren zur Übertragung eines Pakets maximal benötigte Zeit eine lineare Funktion der Paketgrösse. Die Abschätzung wird damit zwar abhängig von der Paketgrösse, ist aber ansonsten leicht und relativ präzis zu bewerkstelligen. Die während der Verarbeitung in einem der Prozessoren verstrichene Zeit ist bei reinen Header-Processing-Anwendungen unabhängig von der Paketgrösse, unterliegt aber aufgrund der nicht deterministischen Anzahl Prozessor-Busanfragen unter Umständen erheblichen Schwankungen. Eine Obergrenze anzugeben, wird damit theoretisch unmöglich, praktisch kann jedoch ein genügend hoher Wert angegeben werden, dessen Überschreiten sehr unwahrscheinlich ist. Hier muss ein Kompromiss eingegangen werden zwischen einem hohen Wert, der kaum jemals überschritten wird, und einem tiefen Wert, welcher der mittleren Verzögerung im Prozessor möglichst nahe kommt. Falls die auszuführenden Anwendungen nicht für alle Pakete identisch sind, muss vor dem Scheduling womöglich ein verfrühtes Header Parsing stattfinden und die Abschätzung entsprechend vorgenommen werden. Dies ist mit einem erheblichen Mehraufwand verbunden.

Bei der Besprechung von Szenario I (Abschnitt 3.4.4) wurde festgehalten,

dass sich eine Output-Warteschlange in der vorliegenden Netzwerkprozessor-Architektur grundsätzlich negativ auf die Paketverzögerung auswirkt. Da nun aber für das Wirken des beschriebenen Schedulers eine Warteschlange vorhanden sein muss (ansonsten könnte er durch einen FIFO-Scheduler ersetzt werden), erweist sich sein Einsatz im Sendepfad als inadäquat. Experimente mit einer Warteschlange, deren Länge mittels eines Kontrollmechanismus auf einer für das Scheduling ausreichenden Länge gehalten wurden, haben sich als wenig erfolgversprechend erwiesen. Der vom Scheduler erreichte Zeitgewinn zugunsten der Best-effort-Pakete vermag den Zeitverlust aufgrund der zusätzlichen Warteschlange nicht aufzuwiegen. Die Verzögerung der Best-effort-Pakete bleibt damit gleich oder erfährt gar eine Erhöhung. Der Scheduler an sich leistet jedoch einen hervorragenden Dienst, schöpft er doch die Deadline praktisch voll aus, ohne sie jemals zu überschreiten. In einem anderen Zusammenhang könnte er daher sicherlich erfolgreich eingesetzt werden. Der mangelhafte Nutzen als Output-Scheduler im vorliegenden Szenario stellt sein Prinzip nicht in Frage, sondern ist vielmehr auf andere Faktoren zurückzuführen.

Als Konsequenz folgt die Einsicht, dass der Deadline Observing Scheduler besser als Prozessor-Scheduler im Dispatcher eingesetzt werden sollte. Damit erschwert sich zwar die notwendige Zeitabschätzung für die Real-time-Pakete, dafür erhält der neue Scheduler nun die Kontrolle über die wichtigste Warteschlange im betrachteten Prozessor. Wird eine Ausgangswarteschlange durch geeignete Arbitrierung verhindert, so kann am Ausgang ein einfacher FIFO-Scheduler eingesetzt werden. Für die Simulation dieses Szenarios wurde jedoch wie bereits im Szenario I auf einen Packet Priority Scheduler zurückgegriffen (für eine bessere Vergleichbarkeit der Resultate). Der Eingangsscheduler bleibt ebenfalls unverändert.

3.5.3 Arbitrierung

Eine Grundvoraussetzung für das Funktionieren des vorgestellten Deadline Observing Schedulers ist die Abschätzbarkeit der Paketverzögerung sowohl bei der Verarbeitung in den Prozessoren als auch beim Transfer über die Brücke. Beide Prozesse generieren eine gewisse Menge von PLB-Requests, deren allfällige Verzögerung sich unmittelbar in einer Verlangsamung der entsprechenden Prozesse niederschlägt. Soll nun eine Abschätzung der maximalen Prozessdauer vorgenommen werden, so bedingt dies, dass eine obere Grenze für die von den Buszugriffen herrührende Verzögerung gegeben werden kann. Ein brauchbarer Arbiter muss also zwingend eine begrenzte Latenz anbieten und damit eine Minimalbandbreite garantieren.

Des weiteren wurde festgestellt, dass eine Ausgangswarteschlange möglichst

vermieden werden sollte. Dies kann erreicht werden, wenn die der Brücke zur Verfügung stehende Busbandbreite ausreichend ist, um ein Aufstauen von Paketen am Ausgang zu verhindern. In Szenario I konnte dies durch statische Master-Priorisierung einfach umgesetzt werden, doch hier ist als Folge der Forderung nach garantierter Busbandbreite seitens der Prozessoren ein anderes Vorgehen gefragt. Ein Arbiter mit begrenzter Latenz, der fähig ist, den Buszugriff unter den Masters nach einem bestimmten Schlüssel aufzuteilen, ist der Weighted Round Robin Arbiter (siehe Abschnitt 3.2.1.2). Wird seine Sequenz so gestaltet, dass sie die für die Brücke reservierte Bandbreite auf Kosten der Prozessoren im Verhältnis zum realen Request-Aufkommen klar übergewichtet, so resultiert dies in einer Priorisierung des Ausgangs, der gleichzeitig die prozessorseitige Latenz-Garantie aufrechterhält. Da die überschüssige Busbandbreite der Brücke von den Prozessoren genutzt werden kann, kommt es zu keinen Ineffizienzen.

3.5.4 Simulationsergebnisse

Die für die Simulation von Szenario II geltenden Parameter und Paketströme decken sich mit denjenigen, welche schon im Szenario I verwendet wurden (siehe Tabelle 3.1). Damit beschränkt sich der Unterschied der Simulationsresultate auf die von Arbitrierung und Scheduling hervorgerufenen Effekte. Die analogen Graphiken sind in Abbildung 3.6 dargestellt, die entsprechenden numerischen Werte in Tabelle 3.3 (zum Vergleich sind hier nochmals die Ergebnisse aufgeführt, welche sich in Szenario I aus der Simulation mit dem Master Priority Arbiter ergaben).

Anmerkungen zur Simulation:
Aus zeitlichen Gründen konnte der Deadline Observing Scheduler in seiner oben beschriebenen Form nicht mehr vollständig als Prozessor-Scheduler implementiert werden. Wie erwähnt wurde aber seine Funktionstüchtigkeit als Output-Scheduler überprüft und bestätigt. Der Einsatz als Prozessor-Scheduler wird durch die beschriebene Abschätzungs-Problematik sowie durch das Vorhandensein multipler Verarbeitungseinheiten komplizierter. Um dennoch seine Vorteile für den Best-effort-Verkehr verifizieren zu können, wurde er durch eine vereinfachte Scheduling-Disziplin approximiert. Aus Sicht eines Real-time-Pakets entspricht der Deadline Observing Scheduler im wesentlichen einem Verzögerungsglied, dessen Verzögerung je nach gegenwärtigem Verkehrsaufkommen variiert. Liegt nun die Deadline deutlich über der eigentlich notwendigen Verarbeitungszeit für ein Real-time-Paket, so wird die aufgezwungene Verzögerung zu einem grossen Teil aus einem konstanten Offset bestehen. Es ist zu erwarten, dass ein Scheduler, der Real-time-Pakete bloss um diesen konstanten Wert verzögert und keine Abschätzung bezüglich Verarbei-

Arbiter:		**WRR**	**Master Prio.**
Prozessor-Scheduler:		**DL Observing**	**Packet Prio.**
RT Deadline [μs]		50.0	–
Mittlere	RT	35.5	13.6
Verzögerung [μs]	BE	51.4	60.3
Maximale	RT	42.5	27.1
Verzögerung [μs]	BE	66.3	76.9
Summe aller	RT	6 137.1	2 335.4
Verzögerungen [μs]	BE	16 865.0	19 784.5
	total	23 002.1	22 119.9

Tabelle 3.3: Zusammenfassung der Resultate für Szenario II (WRR = Weighted Round Robin, DL = Deadline; RT = Real-time, BE = Best-effort). Der Deadline Observing Scheduler wurde durch eine Verzögerungsdisziplin (konstante Verzögerung von 25.0 μs) und einen Packet Priority Scheduler approximiert.

tungszeiten vornimmt, ähnliche – wenn auch weniger ausgeprägte – Vorteile für die Best-effort-Pakete zu Tage bringen wird. Zu beachten ist bei dieser Annäherung, dass der Verzögerungs-Scheduler work-conserving sein muss, ansonsten verliert der Erhaltungssatz (3.11) seine Gültigkeit. Pakete dürfen folglich nur dann zurückgehalten werden, wenn ausreichend Best-effort-Pakete vorhanden sind, die davon profitieren können. Real-time-Pakete, die ihre Verzögerung abgesessen haben, werden danach von einem herkömmlichen Packet Priority Scheduler einem Prozessor zugewiesen. Der Verzögerungsparameter wurde auf einen konservativen Wert gesetzt, der die Einhaltung der Deadline nicht gefährdet.

Die Resultate sollen nun im Vergleich zur Simulation mit dem Master Priority Arbiter aus Szenario I erläutert werden. Wird Tabelle 3.3 bzw. Abbildung 3.6 konsultiert, so stellt man fest, dass sich die eingeführte Verzögerungsdisziplin der Real-time-Pakete direkt in ihrer Gesamtverzögerung niederschlägt. Von diesem Verlust profitieren die Best-effort-Pakete, welche gegenüber der Vergleichssimulation eine signifikant reduzierte Verzögerung aufweisen. Ihre Varianz bleibt vergleichbar gering. Die beobachtete Verlagerung der Paketverzögerung bestätigt die Gültigkeit des Erhaltungssatzes 3.11, was auch in der nahezu unveränderten Summe aller Verzögerungen zum Ausdruck kommt. In der Histogramm-Darstellung äussert sich der beobachtete Effekt durch ein Zusammenrücken beiden Verteilungskurven.

Die gesetzte Deadline wird von keinem Real-time-Paket verletzt, womit die geforderte Quality of Service gewährleistet ist. Der zur Verfügung stehende Zeitrahmen für diese Paketklasse könnte sogar noch besser ausgeschöpft wer-

den – ein Ziel, welches höchstwahrscheinlich mit Hilfe eines vollumfänglich implementierten Deadline Observing Schedulers erreicht werden könnte.

3.5.5 Schlussfolgerungen

Zu Beginn dieses Szenarios wurde das Ziel gesetzt, die Dienstgüte des Best-effort-Verkehrs zu verbessern bei gleichzeitigem Einhalten einer strengen Real-time-Vorgabe. Zur Lösung des Problems wurde eine gezielte Schedulingdisziplin vorgeschlagen, welche im betrachteten Netzwerkprozessor eingesetzt werden sollte. Von ihrer Eigenschaft, die Best-effort-Pakete zu bevorzugen, ohne dass dabei Deadlines von Real-time-Paketen verpasst werden, wurde eine merkliche Umverteilung der Paketverzögerung erwartet. Bei der Simulation eines ähnlich operierenden (aber einfacher implementierbaren) Schedulers traten denn auch die angestrebten Effekte auf, ohne dass dies auf Kosten der Effizienz ging. Es stellt sich heraus, dass sich durch geeignete Massnahmen die Verzögerungen verschiedener Paketströme kontrolliert umverteilt und damit den Bedürfnissen des Netzwerks angepasst werden können.

Es muss hier unterstrichen werden, dass obwohl in diesem zweiten Szenario mehrheitlich von Scheduling die Rede war, die Arbitrierung keineswegs als nebensächlich angesehen werden darf. Solange die Buskapazität begrenzender Faktor im Netzwerkprozessor ist, kommt dem Arbiter die zentrale Aufgabe zu, über die Entstehung von Warteschlangen vor Verarbeitungseinheiten zu entscheiden. Diese wiederum machen den Hauptanteil der Paketverzögerung aus und können bei ungeschickter Verteilung zu beträchtlichen Leistungseinbussen führen. Gleichzeitig bietet das Vorhandensein von Warteschlangen die Möglichkeit, durch gezieltes Scheduling Paketströme entsprechend ihrer QoS-Vorgaben zu verwalten. Schedulingdisziplinen wie der vorgestellte Scheduler benötigen Garantien bezüglich Buslatenz bzw. -bandbreite, die ihnen nur durch eine adäquate Arbitrierung zugesichert werden können. Hier leistet der in diesem Szenario verwendete Weighted Round Robin Arbiter gute Dienste, dies bei einer hohen Flexibilität bezüglich Bandbreitenaufteilung und gleichzeitig einer unproblematischen Realisierung.

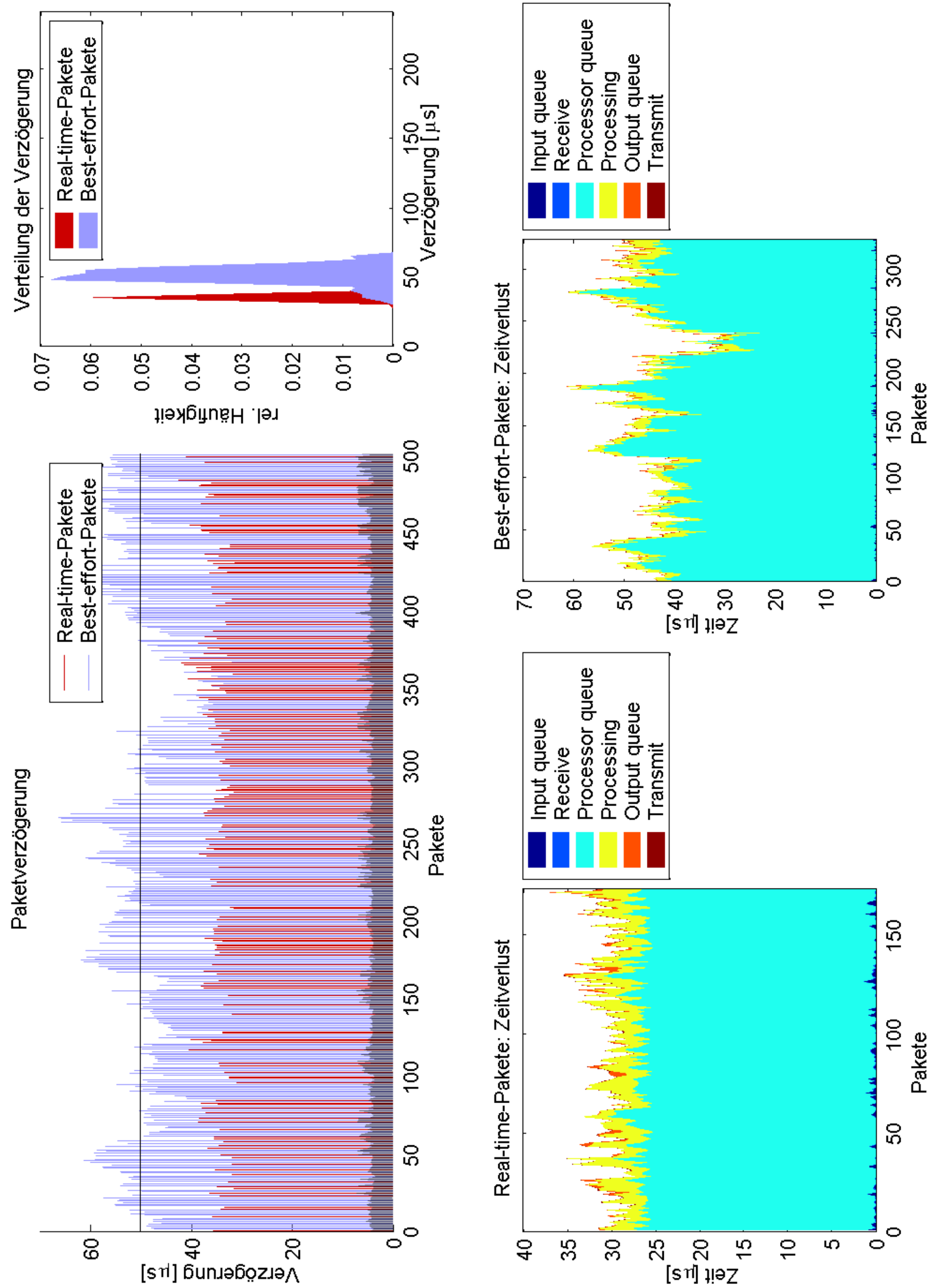

Abbildung 3.6: Simulationsresultate für den Deadline Observing Scheduler.

Kapitel 4

Software-Dokumentation

Der Netzwerkprozessor-Simulator ist in SystemC geschrieben. Eine kurze Einführung in SystemC bietet [19]. Da eine Dokumentation der Software weitgehend fehlt, werden in diesem Kapitel im Hinblick auf zukünftige Arbeiten die wichtigsten Datenstrukturen, Module sowie deren Kommunikation beschrieben.

4.1 SystemC-Simulator

Module sind SystemC-Datenstrukturen und entsprechen den Grundblöcken im HW/SW-Design. Mit Hilfe der Module lässt sich ein komplexes System in kleinere und überschaubare Einheiten aufteilen. Module werden durch das Schlüsselwort *SC_MODULE* deklariert. Sie können selber weitere Module oder Prozesse beinhalten. Prozesse werden gebraucht, um die Funktionalität eines Systems zu beschreiben. SystemC unterstützt drei verschiedene Typen von Prozessen:

- **Methods:** *Methods* sind ereignisgetrieben. Sobald ein Ereignis auftritt, d.h. ein Signal seinen Wert ändert, wird der Prozess aktiviert und vollständig ausgeführt.
- **Threads:** *Threads* sind ebenfalls ereignisgetrieben. Ein *Thread* kann *wait()*-Funktionen beinhalten, die den Prozess unterbrechen und wieder aktivieren, sobald ein Ereignis auftritt.
- **Clocked Threads:** *Clocked Threads* sind zeitgetrieben. Diese Prozesse werden bei einer der zwei Taktflanken jeweils aktiviert.

Die Prozesse werden im Konstruktur eines Moduls, durch die Schlüsselwörter *SC_METHOD, SC_THREAD* und *SC_CTHREAD* deklariert.

In diesem Kapitel werden *PacketRecord* und Paket gleichbedeutend verwendet. Ein *PacketRecord* ist eine C++-Datenstruktur, welche paketspezifische Informationen beinhaltet, wie zum Beispiel die Paket-ID, die Paketlänge, die Anzahl Pufferdeskriptoren, die EMAC-ID etc. Abbildung 4.1 zeigt die Modularchitektur des Simulators. Alle top-level Module werden in der main()-Funktion (*top.cpp*) instanziert. In diesem Beispiel sind fünf Masters vorhanden, nämlich zwei Prozessoren, der DMA-Block und die PLB-OPB-Brücke, welche zwei Masters repräsentiert. Einer entspricht der eigentlichen Brücke, der andere ist für die Verwaltung von Pufferdeskriptoren zuständig. Die mit gelber Farbe gekennzeichneten Module dienen als Hilfsmodule und entsprechen keinen funktionalen Einheiten im realen System.

4.2 Multiprozessormodell

Abbildung 4.2 zeigt den Datenfluss eines Pakets aus Sicht des Dispatchers. Pakete werden aus dem Hauptspeicher aus einer logischen Empfangswarteschlange gelesen, die durch die C++-Datenstruktur *rx_prplp* repräsentiert wird. Je nach Anwendung wird nur der Header oder das ganze Paket geladen. Der Dispatcher wählt einen Prozessor für die Verarbeitung des Pakets aus und reiht dieses in die entsprechende Warteschlange. Nachdem ein Paket verarbeitet wurde, wird es in die Sendewarteschlange, die durch das Objekt *tx_prplp* dargestellt wird, zurückgeschrieben. Im Dispatcher sind drei Threads aktiv:

- *Entry()*
- *Next_Processing_Step()*
- *Transmit()*

***Entry*-Thread.** Dieser Thread wird von zwei Signalen getriggert:

- *rx_ch_done*
- *rx_host_done*

Diese Signale werden vom RxEMAC oder RxHost gesetzt, sobald das entsprechende Paket vom EMAC oder Host in den Speicher übertragen worden ist und somit zur Bearbeitung bereit steht. Abbildung 4.3 zeigt die Modularchitektur des Multiprozessormodells. Die Pfeile entsprechen den Signalen, durch welche die Module miteinander kommunizieren. Je nach Pfeilrichtung handelt es sich um eine bi- oder unidirektionale Kommunikation. Sobald

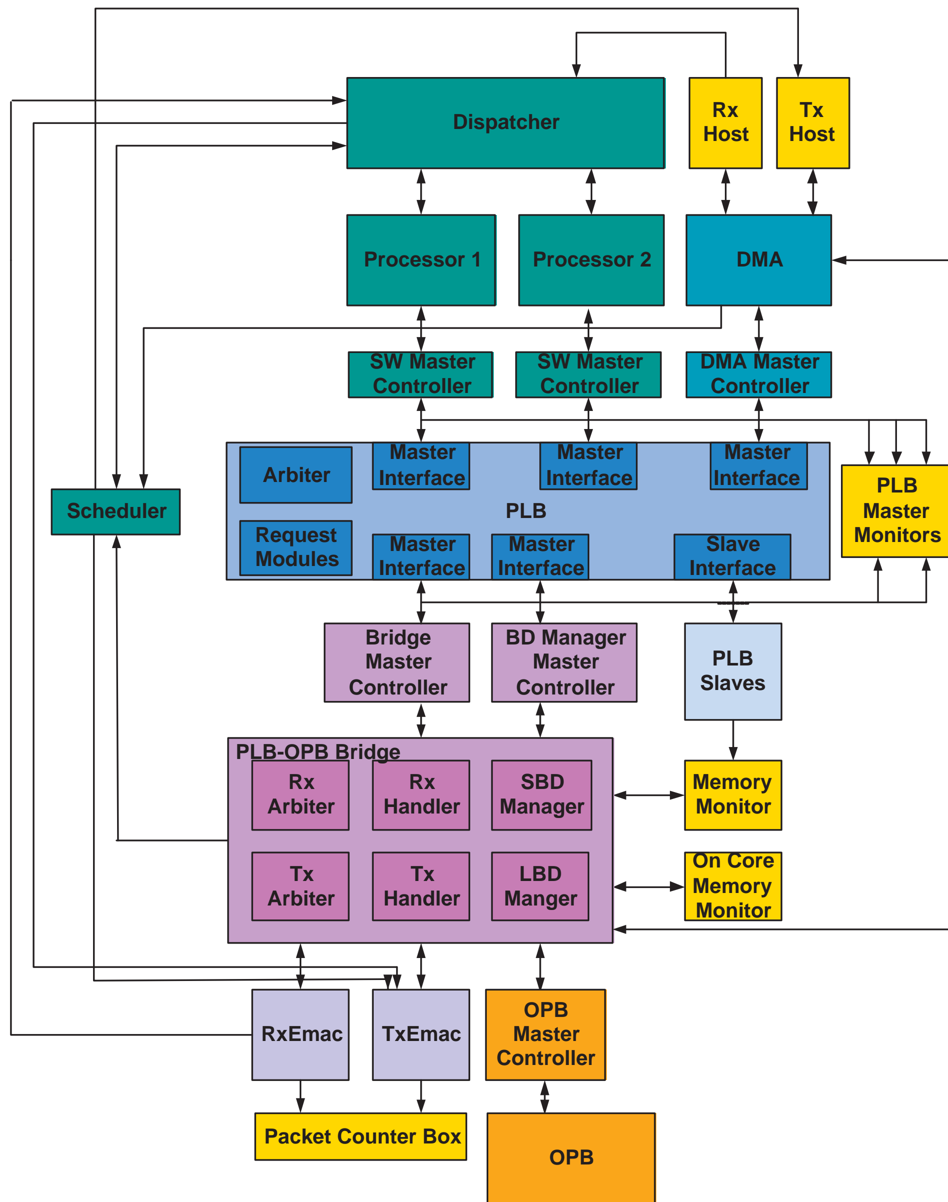

Abbildung 4.1: Netzwerkprozessor-Modell.

eines der oben erwähnten Signale gesetzt ist, wird aus der Empfangswarteschlange (*rx_prplp*) der nächste zu bearbeitende *PacketRecord* gelesen. Die dabei verwendete Funktion *next()* ist eine Scheduling-Funktion, die unter allen aktiven Empfangskanälen (Rx-Kanäle) einen *PacketRecord* auswählt. In Abschnitt 4.4 wird das CPU-Scheduling genauer beschrieben.
In einem nächsten Schritt wird bestimmt, welche Anwendungen auf das Pa-

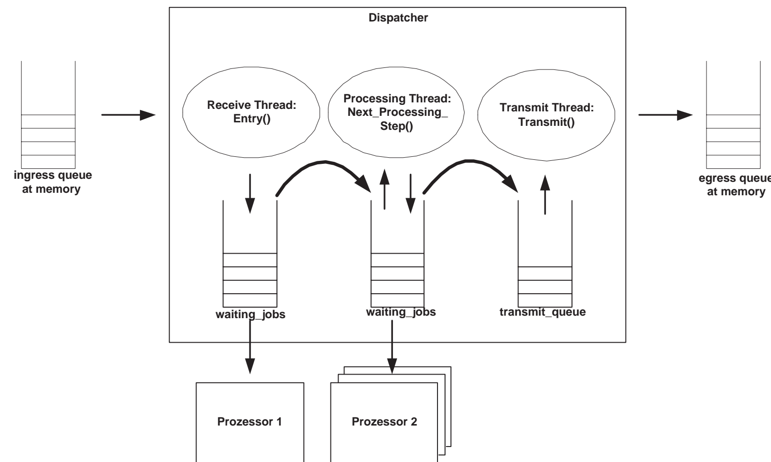

Abbildung 4.2: Dispatcher.

ket auszuführen sind. Die im Prozessormodell simulierten Anwendungen basieren auf den CommBench-Applikationen. CommBench unterscheidet acht Anwendungen, vier beinhalten nur Header Processing, während die restlichen vier Payload Processing durchführen. Die CommBench-Anwendungen decken ein breites Spektrum von Netzwerkanwendungen ab. Das Modell erlaubt es, für jede Anwendung eine Wahrscheinlichkeit zu definieren, mit welcher die entsprechende Applikation auf ein Paket ausgeführt wird oder eben nicht. Dies geschieht mit Hilfe des Paramters

- *MTP_APP_OCCURENCE (MultiProcessor_params.h).*

Danach wird der *PacketRecord* in eine Taskliste eingetragen, deren Elemente durch die Datenstruktur *PPtasklist_entry* repräsentiert werden. Die Taskliste ist eine verkettete Liste. Die einzelnen Listenelemente enthalten neben dem *PacketRecord* einen Array, in welchem die auf ein Paket auszuführenden Anwendungen festgehalten sind.

In einem nächsten Schritt wird anhand der Taskliste und der Funktion *enqueue_packet()*, der *PacketRecord* in die FIFO-Warteschlange *waiting_jobs[]* eines Prozessors geschrieben. Dabei sind zwei Fälle zu unterscheiden, je nachdem ob die Multiprozessoreinheit im Run-to-Completion- oder im Pipelining-Modus läuft. Im Run-to-Completion-Modus wird das Paket dem Prozessor mit der kürzesten Warteschlange zugewiesen. Dies ist ein simpler Load-Balancing-Algorithmus, der nur die Anzahl wartender Pakete betrachtet. Der Algorithmus kann verbessert werden, indem weitere Kriterien einbezogen werden, wie zum Beispiel die Qualitätsklasse des Pakets oder die Länge der Warteschlangen in Bytes. Das Load-Balancing ist in der Funktion *sche-*

dule_packet() implementiert. Im Pipelining-Modus wird das Paket in die FIFO-Warteschlange des Prozessors geschrieben, der die erste auszuführende Anwendung implementiert. Die Methode *send_job()* triggert den entspre-

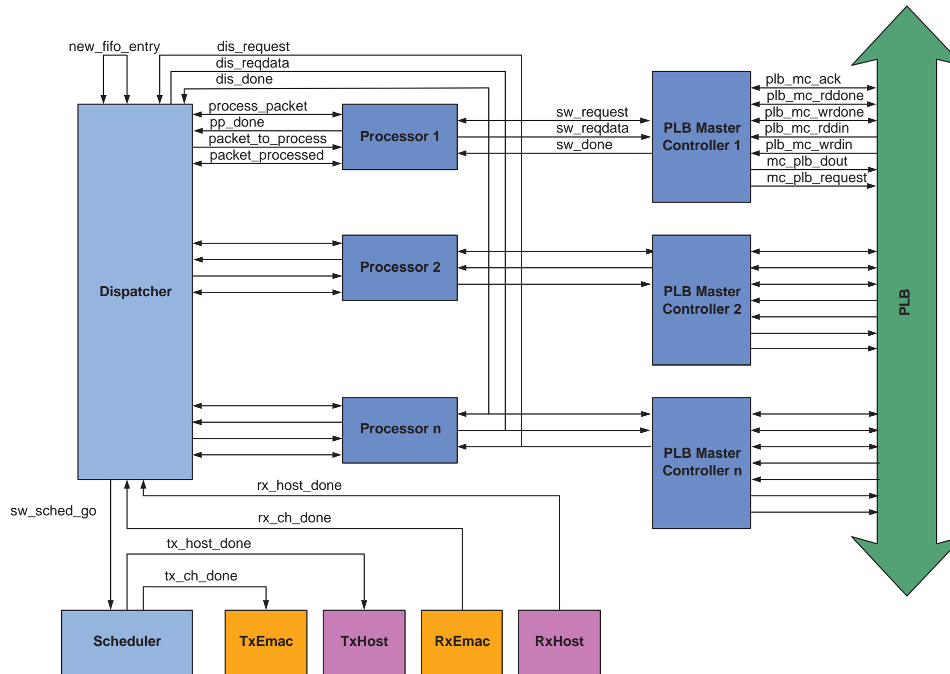

Abbildung 4.3: Multiprozessor-Modell.

chenden Prozessor, indem das Signal *process_packet* gesetzt wird. Mit Hilfe des Signals *packet_to_process* wird dem Processor ein Zeiger auf das zu bearbeitende Paket übergeben. Danach begibt sich der *Entry*-Thread in den Ruhezustand bis ein neues Paket zur Bearbeitung bereitsteht.

***Next_Processing_Step*-Thread.** Sobald ein Processor ein Paket bearbeitet hat, wird dieser Thread aktiviert, indem das Signal *pp_done* gesetzt wird. Über das Signal *packet_to_process* wird dem Dispatcher ein Zeiger auf das entsprechende Paket übergeben. Im Pipelininig-Modus wird nun die nächste auf dieses Paket auszuführende Applikation bestimmt. Falls keine weiteren Verarbeitungsaufgaben mehr auszuführen sind, wird das Paket in eine Ausgangswarteschlange (*transmit_queue*) geschrieben, ansonsten wird das Paket in die entsprechende FIFO-Warteschlange desjenigen Prozessors geschrieben, der für die nächste Applikation zuständig ist. Im Run-to-Completion-Modus tritt dieser Fall jeweils gleich nach der Bearbeitung durch den ersten Prozessor auf, da dieser alle möglichen Anwendungen ausführt.

***Transmit*-Thread.** Dieser Thread ist aktiv, solange seine Warteschlange

transmit_queue nicht geleert ist. Pakete werden in die Sendewarteschlange *tx_prplp* eingereiht, der entsprechende Listeneintrag wird gelöscht und der Scheduler durch das Signal *sw_sched_go* getriggert.

4.2.0.1 Dispatcher-PLB

In Abschnitt 2.4.1 wurde erwähnt, dass der Dispatcher als interne Einheit eines Prozessors aufgefasst werden kann. Über die PLB-Schnittstelle des Prozessors ist es dem Dispatcher erlaubt, den PLB zu benutzen. Über die Signale

- *mt_request*
- *mt_reqdata*
- *mt_done*

wird die Kommunikation mit dem PLB abgewickelt. Der Prozessor verfügt über einen Thread namens

- *handle_Dispatcher_plb_access()*,

der die Anfragen des Dispatchers entgegennimmt und diese über die Signale

- *sw_request*
- *sw_reqdata*
- *sw_done*

weiterleitet.

4.2.1 Prozessor-Anfragen

Jeder Master verfügt über eine Datenstruktur, mit welcher die PLB-Anfragen modelliert werden. Die Prozessoren generieren Anfragen des Typs *SWPlbReq*, die PLB-OPB-Brücke solche des Typs *BRIDGE_PLB_Req*, während DMA-Anfragen durch die Datenstruktur *DMA_PLB_Req* repräsentiert werden. Aufgabe der entsprechenden Master Controllers ist es, ankommende Anfragen in PLB-konforme Anfragen des Typs *PLB_Request* umzuwandeln. Diese werden wiederum durch das Modul *PLB_Slave_Interface* in Anfragen

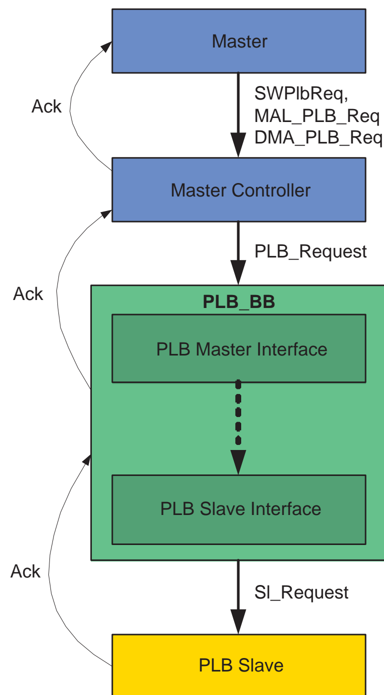

Abbildung 4.4: Anfragenfluss.

des Typs *Sl_Request* umgewandelt. Dieser Anfragenfluss ist in Figur 4.4 abgebildet. Eine Anfrage des Typs *SWPlbReq* kann mehrere PLB-Anfragen beinhalten, maximal jedoch *MAX_SW_PLB_REQ_NBR* Anfragen. Im Modell werden zwei PLB-Anfragen unterschieden: Single- und Burst-Anfragen. Bei Single-Anfragen werden vier Wörter (1 Wort entspricht 32 bit), bei Burst-Anfragen 16 Wörter übertragen. Für eine längere Zeit, als für die Übertragung eines Bursts benötigt wird, kann der PLB-Bus nicht reserviert werden. Das Laden des Headers (64 Bytes) wird durch eine einzige Burst-Übertragung bewerkstelligt. Das Laden der Payload benötigt je nach Grösse unterschiedlich viele Anfragen. Der Master Controller des Prozessors liest die ankommenden Anfragen und generiert die entsprechenden PLB-Anfragen.

4.3 Direct Memory Access Controller (DMA)

Bei der Implementierung des DMA-Moduls konnte bereits vorhandener Code benutzt werden. Das DMA-Modul basiert auf der Modellierung der PLB-OPB-Brücke. Zunächst wird deshalb das Brücken-Modell betrachtet.

4.3.1 PLB-OPB-Brücke

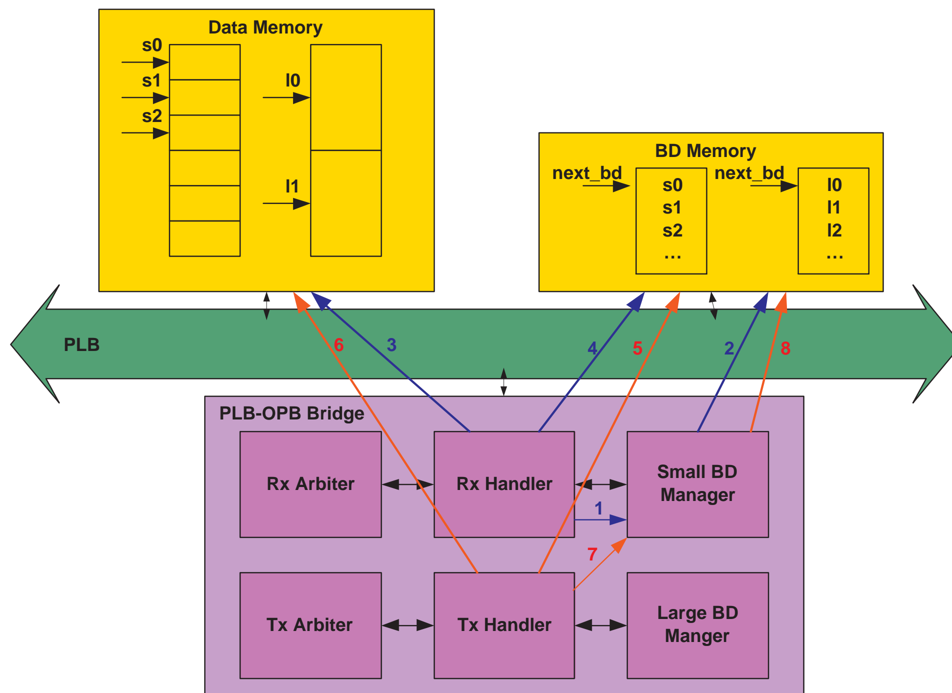

Abbildung 4.5: Pufferdeskriptoren-Verwaltung.

In Abbildung 4.5 sind die PLB-Zugriffe aus Sicht der PLB-OPB-Brücke eingezeichnet, die zur Übertragung eines Pakets notwendig sind. Aus Gründen der Übersichtlichkeit wird dabei der Fall eines Pakets minimaler Länge betrachtet. Mit Hilfe dieser Abbildung wird ein Einblick in die Kommunikation der verschiedenen internen Brücken-Module in Bezug auf die Verwaltung der Pufferdeskriptoren gegeben. Aus der Abbildung wird ersichtlich, dass das Brücken-Modul sechs interne Module beinhaltet:

- *RxArbiter*
- *RxHandler*
- *TxArbiter*
- *TxHandler*
- *Small BD Manager*
- *Large BD Manger*

Die PLB-OPB-Brücke wird vom RxEMAC über zwei Sideband-Signale getriggert:

- *com_rx_arb_level*
- *com_rx_frame*

Durch diese Signale wird das RxArbiter-Modul aktiviert, dessen Aufgabe es ist, unter den anfragenden RxEMACs zu arbitrieren. Nachdem ein RxEMAC ausgewählt worden ist, wird durch interne Signale der RxHandler aktiviert. Dieser schreibt ein Paket in kleinen Dateineinheiten vom RxEMAC in den Speicher.

Bevor ein Paket in den Speicher geschrieben werden kann, muss natürlich freier Speicherplatz zur Verfügung stehen und die betreffenden Speicheradressen müssen bekannt sein. Die Speicheradressen werden im realen System in Pufferdeskriptoren festgehalten, die von der Brücke zentral verwaltet werden. Im Modell werden die Deskriptoren durch die Datenstruktur *BD_entry* repräsentiert. *BD_entry* beinhaltet lediglich einen Zeiger auf den nächsten Pufferdeskriptor und eine Variable (*length*), die dem tatsächlich benutzten Pufferspeicher entspricht. *BD_entry* enthält jedoch keine Speicheradresse und auch kein Statusfeld. Wie bereits in Abschnitt 2.1 erwähnt, sind die Speicher- und Bus-Komponenten auf einer hohen Abstraktionsebene modelliert. Es werden keine Daten über die Busse übertragen und in die Speicher geschrieben, sondern bloss die entsprechenden Übertragungs- und Speicherzugriffszeiten abgewartet. Trotzdem sind im Modell im Hinblick auf eine mögliche Verfeinerung die nötigen Datenstrukturen zur Verwaltung der Deskriptoren bereits vorhanden.

Die Pufferdeskriptoren werden mit Hilfe zweier Listen verwaltet, *small_bd_list*, für die kleinen Puffer und *large_bd_list* für die grossen Puffer. Auf diese Listen hat nur die PLB-OPB-Brücke Zugriff. Die Module namens Small und Large BD Manager verwalten die unterschiedlichen Pufferbereiche. Für die Verwaltung der Deskriptoren sind unter anderem folgende C++-Datenstrukturen von Bedeutung: *sb_cache* und *lb_cache*, die je nach Vorhandensein eines Deskriptorencache, vom Typ *NoBDCache* oder *BDCached* sind. Diese Datenstrukturen beinhalten unter anderem einen Zeiger (*next_bd*) auf den nächsten freien Puffer, eine Liste aller Pufferdeskriptoren (*small_bd_list* oder *large_bd_list*) sowie die folgenden Funktionen mit deren Hilfe die eigentliche Deskriptoren-Verwaltung stattfindet.

- *request()*: Rückgabewert dieser Funktion ist der Zeiger *next_bd*. Falls kein Deskriptor vorhanden ist, wird der Null-Zeiger zurückgegeben.

- *release()*: Die entsprechende Deskriptorenliste (*small_bd_list* oder *large_bd_list*) wird um einen Eintrag erweitert.
- *fetch()*: Die entsprechende Deskriptorenliste (*small_bd_list* oder *large_bd_list*) wird um einen Eintrag reduziert und der Zeiger *next_bd* wird auf einen Listeneintrag gesetzt.

Die Anzahl Einträge einer Deskriptorenliste entspricht der Anzahl freier Puffer. Zu jedem Simulationszeitpunkt kann somit der momentan freie Speicherplatz bestimmt werden. Die Variable *length* der Datenstruktur *BD_entry* repräsentiert den tatsächlich benutzten Speicher pro Puffer. Die Summe der *length*-Variablen über alle *BD_entry*-Elemente entspricht der gesamten Datenmenge im Speicher.

Im nächsten Abschnitt ist die Abbildung 4.5 beschrieben. Die Nummerierung der Pfeile entspricht der logischen Reihenfolge der Schritte, die aus Sicht der Brücke bei der Abarbeitung eines Pakets minimaler Länge durchlaufen werden.

1. Wie bereits erwähnt wird der RxHandler durch interne Signale aktiviert. Dieser ruft die Methode *request()* auf. Da kein Dekriptorencache vorhanden ist und somit auch kein Deskriptor bekannt sein kann, wird der Small BD Manager getriggert, damit dieser aus den Speicher einen Deskriptor liest.
2. Der Small BD Manager ruft die Funktion *fetch()* auf und generiert einen PLB-Zugriff. Der ganze Vorgang entspricht im realen System dem eigentlichen Lesen eines Deskriptors aus den Speicher.
3. Der RxHandler ruft wiederum die Methode *request()* auf. Der Pointer*next_bd* zeigt nun auf einen freien Puffer. Daten werden über den PLB in den Hauptspeicher geschrieben.
4. Danach wird das Statusfeld des entsprechenden Deskriptors aktualisiert. Ein PLB-Zugriff ist nötig. Der Empfangspfad ist mit diesem Schritt abgeschlossen.
5. Nachdem der Prozessor das Paket bearbeitet hat, wird das Paket in einen TxEMAC geschrieben. Der TxEMAC wird durch die Routing-Funktion *fwrd_pkt()* des enstprechenden Prozessormodells gewählt. Der Prozessor triggert einen TxEMAC, sobald ein Paket zum Senden bereitsteht. Dieser wiederum triggert die Brücke ,um die Übertragung zu starten. Der TxHandler liest über den PLB einen Pufferdeskriptor.

6. Die Daten werden vom Speicher in den TxEMAC geschrieben.
7. Danach ruft der TxHandler die Methode *release()* auf und triggert den BD Manager, damit dieser den soeben verwendeten Puffer freigibt. Der Funktionsaufruf bewirkt, dass der Deskriptorenliste (*small_bd_list*) ein Deskriptor hinzugefügt wird.
8. Der zugehörige PLB-Zugriff wird generiert.

4.3.2 DMA-Modul

Abbildung 4.6 zeigt die Modul-Architektur des DMA-Modells, während in Figur 4.7 die internen Module der DMA-Einheit abgebildet sind. Der DMA-Block umfasst folgende Module:

- *DMA_RxArbiter*
- *DMA_TxArbiter*
- *DMA_RxHandler*
- *DMA_TxHandler*

Diese Module entsprechen weitgehend den Modulen der PLB-OPB-Brücke.

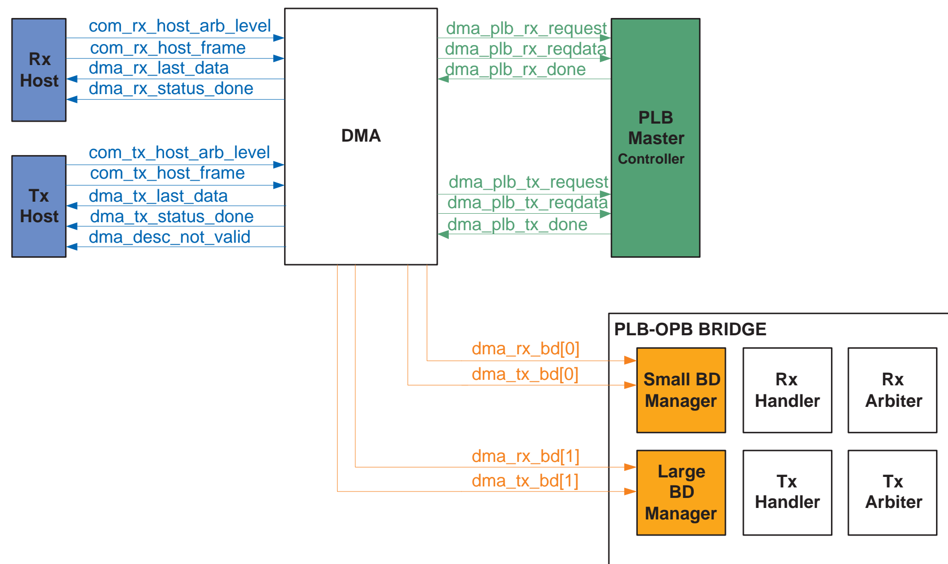

Abbildung 4.6: DMA-Modell.

Die DMA-Einheit verfügt über drei Schnittstellen, die jeweils in den Abbildungen durch unterschiedliche Farben gekennzeichnet sind:

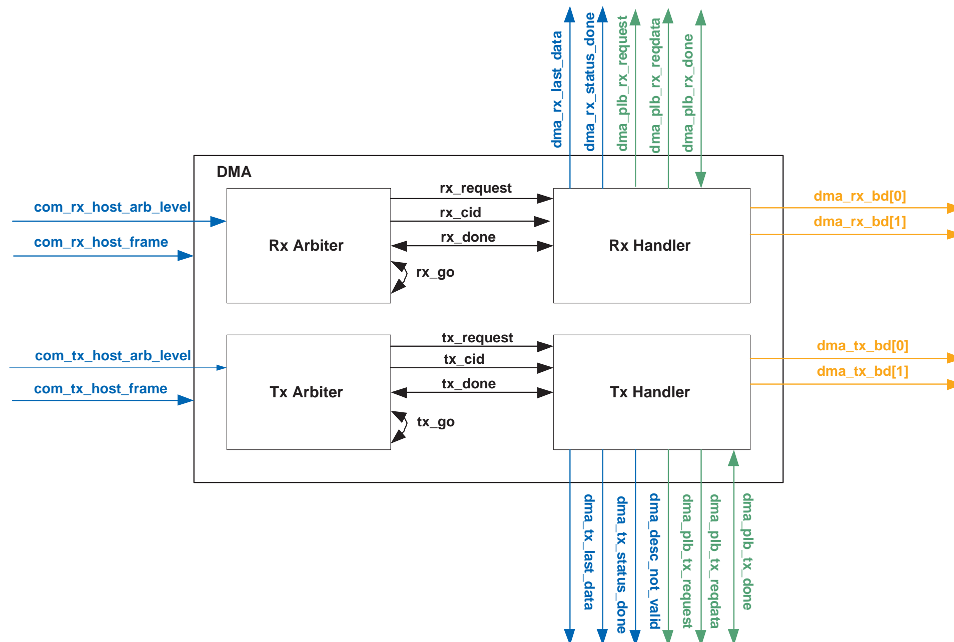

Abbildung 4.7: DMA-Modul.

- DMA – Host
- DMA – PLB
- DMA – Pufferdeskriptoren-Manager

DMA – Host. Das Kommunikationsprotokoll zwischen EMAC und PLB-OPB-Brücke ist ebenfalls für die Kommunikation zwischen Host und DMA verwendet worden. Der Host repräsentiert eine Verkehrsquelle und -senke und ist implementiert worden, um das DMA-Modell in Simulationen einsetzen zu können. Im RxHost wie auch im RxEMAC sind zwei Threads aktiv. Der erste (*producer*) liest reale Verkehrs-Traces aus einer Datei und erstellt einen *PacketRecord* für jedes Paket. Dieser *PacketRecord* wird in eine Liste eingetragen, die dem Host-Empfangs-FIFO entspricht. Der zweite Thread (*rx_channel*) implementiert das Kommunikationsprotokoll mit dem DMA-Block und überträgt Daten zum Hauptspeicher, solange der Empfangs-FIFO nicht leer ist. Da im Vergleich zur Brücke die OPB-Schnittstelle entfernt wurde, stehen die Pakete der DMA-Einheit zur Verfügung, sobald sie in den Empfangs-FIFO geschrieben worden sind. Man kann die Host-Einheit auch als internes Modul der DMA-Einheit betrachten, die jeweils der DMA-Einheit nach einer parametrisierbaren Zeit Pakete zur Übertragung bereitstellt.

DMA – PLB. Über diese Schnittstelle findet die Datenübertragung über den PLB statt. Der PLB-Master-Controller nimmt Anfragen der DMA-Einheit (*DMA_PLB_Req*) entgegen und generiert PLB-konforme Anfragen (*PLB_Request*).

DMA – Pufferdeskriptoren-Verwalter. Über diese Schnittstelle werden die Pufferdeskriptoren-Verwalter getriggert. Falls ein Paket beim Host zur Übertragung bereitsteht, müssen zuerst die Pufferdeskriptoren bezogen werden. Durch das Signal

- *dma_rx_bd[]*

wird je nach Bedarf der Small oder der Large BD Manager getriggert. Dieser aktualisiert die entsprechende Deskriptorenliste, indem er die Funktion *fetch()* aufruft. Gleichzeitig wartet die DMA-Einheit gemäss dem Parameter

- *LATENCY_NEXT_BD (DMA_params.h)*

eine gewisse Anzahl Taktzyklen. Diese Wartezeit entpricht der Zeit, die benötigt wird, um über den Kontrollbus dem DMA-Block einen Pufferdeskriptor zu senden. Nach Ablauf dieser Zeit wird ein PLB-Zugriff generiert, der dem Lesen eines Pufferdeskriptors aus dem Speicher entspricht.

Falls ein Paket das System über einen TxHost verlässt, muss der nicht mehr benötigte Puffer freigegeben werden. Dazu ist ein PLB-Zugriff nötig. Die Deskriptorenliste, die aber nur dem BD Manager bekannt ist, muss ebenfalls angepasst werden. Über das Signal *dma_rx_bd[]* wird der BD Manager getriggert, der darauf die Funktion *release()* aufruft.

4.3.2.1 DMA-Konfigurierung

In Abschnitt 2.4.2 sind bereits mehrere Parameter zur Konfigurierung des DMA- und Host-Moduls erwähnt. Mit Hilfe des Parameters *SELECT_DMA (top_params.h)* kann der Eingangskanal des Host-Moduls deaktiviert werden. Zusätzlich kann je nach Wahl der Forwarding-Funktion des Dispatchers der Ausgangskanal ausgeschaltet werden. Durch diese zwei Einstellungen lässt sich die DMA-Einheit komplett deaktivieren.

Da der Simulator nur beschränkt funktionierte konnte die Implementierung des DMA-Blocks nicht auf Korrektheit verifiziert werden. Derzeit funktioniert der Anschluss der DMA-Einheit am PLB nicht. Deshalb solte diese gemäss der obigen Methode deaktiviert werden.

4.4 Arbitrierung und Scheduling

In diesem Abschnitt werden die Arbiter- und Scheduling-Einheiten betrachtet. Diese sind zu erweitern, falls der SystemC-Simulator dazu benutzt werden soll, die gleichen Szenarien zu simulieren wie es in Kapitel 3 mit dem Matlab-Modell getan wurde.

4.4.1 Prozessor-Scheduling

Beim Prozessor-Scheduling wird entschieden, in welcher Reihenfolge Pakete für eine Verarbeitung aus dem Speicher gelesen werden. Der Dispatcher führt das Prozessor-Scheduling durch den Aufruf der Methode *next()* des Objekts *rx_prplp* aus. *rx_prplp* repräsentiert die Empfangswarteschlange im Speicher. Für jeden Eingangskanal (Rx-Kanal) werden im Speicher drei Warteschalangen unterschieden. In der ersten Warteschlange (*High Priority)* werden priorisierte Pakete, in der zweiten Warteschlange werden Best-Effort-Pakete eingereiht. Die dritte Warteschlange beinhaltet Pakete, die nicht von einer Recheneinheit bearbeitet werden müssen. Solche Pakete werden erst gar nicht in den Hauptspeicher geschrieben, sondern von der PLB-OPB-Brücke direkt in einen Ausgangsport geschrieben. Beim Paket-Scheduling durch den Dispatcher, werden demzufolge keine Pakete in dieser Warteschlange vorhanden sein. Die Zugehörigkeit eines Pakets zu einer dieser Warteschlangen wird bei der Erstellung der Datenstruktur *PacketRecords* durch den RxEMAC nach dem Zufallsprinzip festgelegt.

Die Scheduling-Funktion *next()* funktioniert folgendermassen: Rx-Kanäle untereinander werden nach einem Round-Robin-Schema bearbeitet. In einem Kanal selber werden Pakete aus der High-Priority-Warteschlange bevorzugt behandelt.

4.4.2 Input- und Output-Scheduling

Alle aktiven Rx- und Tx-Kanäle bilden zwei Gruppen. Im Modell wird von jeder Ethernet-Einheit ein Rx- und ein Tx-Kanal bedient. Jeder Kanal hat drei Prioritätsstufen (urgent, high und normal). Die Prioriät wird je nach Gruppe durch die Sideband-Signale

- *com_rx_arb_level*
- *com_tx_arb_level*

festgelegt. Die Gruppe wird nach einer statischen Priorität abgearbeitet, wobei Urgent-Pakete die höchste Priorität haben. Innerhalb derselben Priori-

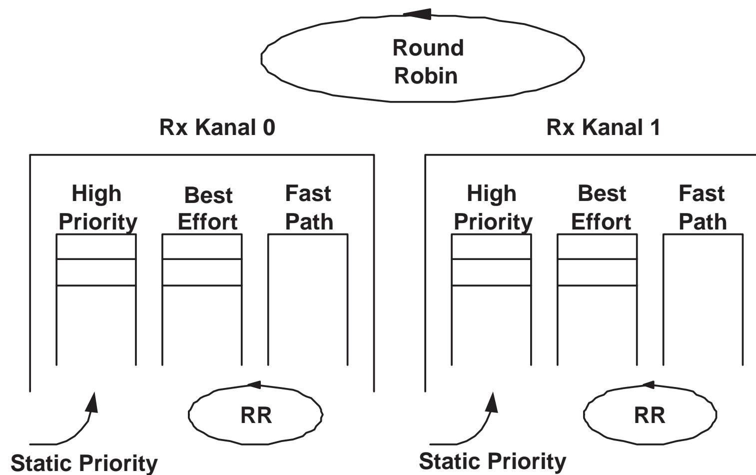

Abbildung 4.8: CPU- und Output-Scheduling.

tätsstufe einer Gruppe wird nach einem Round-Robin-Schema vorgegangen. Die Gruppen untereinander werden ebenfalls im Round-Robin-Verfahren bearbeitet. In Abbildung 4.9 ist das Scheduling bildlich dargestellt.

Die Idee der Differenzierung der Kanäle ist, dass proportional zur momentanen FIFO-Speicherkapazität eines Ethernet-Blocks eine Paketübertragung unterschiedlich behandelt wird. Bevor es zu einem Pufferüberlauf kommt, werden Pakete der entsprechenden Ethernet-Einheit bevorzugt behandelt.
Es ist dabei zu beachten, dass das Scheduling nicht pro Paket stattfindet, sondern für jede übertragene Dateneinheit eines Pakets. Deshalb kann eigentlich auch von einer Art Arbitrierung gesprochen werden.
Falls man davon ausgeht, dass die Zugehörigkeit eines Pakets zu einer bestimmten Verkehrsklasse bereits durch die Ethernet-Blöcke bestimmt wird, lässt sich das Input-Scheduling erweitern, indem zusätzlich die Verkehrsklasse als Kriterium für Scheduling-Entscheidungen beigezogen wird. Das Input-Scheduling ist im Modul *RxArbiter* der PLB-OPB-Brücke mit Hilfe der Methoden *sort()* und *arbitrate()* implementiert.

Das Output-Scheduling findet auf verschiedenen Ebenen statt. Nach Bearbeitung eines Pakets durch die Recheneinheit, wird es in die Ausgangswarteschlange *tx_prplp* geschrieben und das Scheduler-Modul (siehe Abbildung 4.1) wird durch das Signal *sw_sched_go* getriggert. Aufgabe des Schedulers ist es in diesem Fall, einen TxKanal auszuwählen, der Pakete in einer seiner Warteschlangen beinhaltet. Das Scheduler-Modul ruft für diesen Zweck die Methode *channel_go()* des Objekts *tx_prplp* auf. Dabei wird jeweils der TxEMAC mit dem niedrigsten Index ausgewählt. Über Sideband-

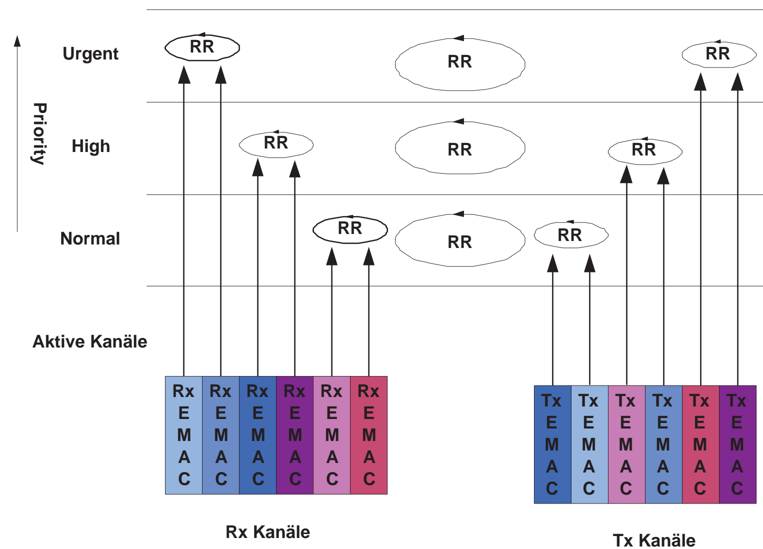

Abbildung 4.9: Input- und Output-Scheduling.

Signale wird der entsprechende TxEMAC informiert, welcher darauf selber die Brücke für eine Paketübertragung triggert. Nach Aktivierung des TxHandler-Moduls der PLB-OPB-Brücke ruft dieser die Methode *dequeue()* des Objekts *tx_prplp* auf. Diese Methode ist eine Scheduling-Funktion für einen Kanal. Analog zum Prozessor-Scheduling werden Pakete aus der High-Priority-Warteschlange jeweils zuerst übertragen. Die Best-Effort- und die Fast-Path-Warteschlange werden nach einem Round-Robin-Verfahren bearbeitet. Durch die Methoden *arbitrate()* und *sort()* des Moduls TxArbiter werden identisch zum Input-Scheduling zusätzlich noch die TxKanäle arbitriert. Wiederum ist zu beachten, dass diese Scheduler-Entscheide nicht das ganze Paket betreffen, sondern jeweils nur die zu übertragenden Dateneinheiten eines Pakets.

4.4.3 PLB-Arbitrierung

Beide Busse, OPB und PLB, verfügen über einen Arbiter. Wie bereits in Abschnitt 2.4 erwähnt, wird hier nicht auf die Arbitrierung auf OPB-Ebene eingegangen. Die Arbitrierung der PLB-Anfragen ist im internen Modul *PLB_arbiter* der PLB-OPB-Brücke implementiert. Es sind vier unterschiedliche Arbiter implementiert. Mit dem Parameter *PLB_ARBITER_POLICY (plb_params.h)* wird der Arbitrierungs-Algorithmus eingestellt.

4.4.4 Verkehrsklassen

Die *PacketRecord*-Datenstruktur wurde um ein Feld erweitert, mit Hilfe welchem die Zugehörigkeit eines Pakets zu einer Verkehrsklasse definiert wird. Es werden drei Verkehrsklassen unterschieden. Die Namensgebung basiert auf den QoS-Ansatz DiffServ [4], der drei Qualitätsklassen umfasst: Assured Forwarding (AF), Expedited Forwarding (EF) und Best Effort (BE). Bei der Erstellung eines *PacketRecords* in den jeweiligen Modulen (RxEMAC und RxHost) wird ein Paket nach einem Zufallsprinzip einer Verkehrsklasse zugeordnet. Die Verkehrklasse eines Pakets wird über alle zugehörigen Anfragen (siehe *BRIDGE_PLB_Req*, SWPlbReq, DMA_PLB_Req und *PLB_Request*) durch die jeweiligen Masters weitergereicht. Die Master-Controllers fügen diese Information ebenfalls in die PLB-Anfragen (*PLB_Request*) ein. Auf diese Weise besteht die Möglichkeit Dienstgüte-Vorgaben auf Bus-Ebene miteinzubeziehen.

Kapitel 5

Zusammenfassung und Ausblick

Im Zentrum dieser Diplomarbeit stand die Bus-Arbitrierung in einer Netzwerkprozessor-Architektur. Das Projekt beinhaltete zum einen die Schaffung einer geeigneten Simulationsumgebung basierend auf einem bestehenden IBM-Prozessormodell, zum anderen die Ergründung der Bedeutung des Arbiters im Netzwerkprozessor.

In einem ersten Schritt wurde die bereits existierende SystemC-Simulation dahingehend erweitert, dass sie den Anforderungen an einen modernen Netzwerkprozessor mit paralleler Paketverarbeitung gerecht wird. Ausserdem wurde mit einer DMA-Komponente eine zusätzliche Verkehrsquelle bzw. -senke am Prozessor-Speicher-Bus, dem PLB, eingeführt. Das mit der Erweiterung verbundene zusätzliche Verkehrsaufkommen lässt die Frage der Bus-Arbitrierung besonders im Falle hoher Belastung an Bedeutung zunehmen. Zur Untersuchung dieser Problematik wurde eine Matlab-Simulation erstellt, mit deren Hilfe der Busverkehr auf dem PLB detailliert betrachtet werden konnte. Im Speziellen wurde Frage der Arbitrierung für zwei unterschiedliche Situationen genauer analysiert und simuliert. Für ein erstes Szenario galt es nachzuweisen, dass die Wahl des Arbiters bei einer erheblichen Busauslastung eine entscheidende Rolle im Hinblick auf die zu erwartende Paketverzögerung spielt. Ziel des zweiten Szenarios war es, eine Strategie zu entwickeln, welche den prozessorinternen Datenaustausch sowohl den Gegebenheiten der Architektur als auch den Forderungen bezüglich Quality of Service optimal anpasst. Zu diesem Zweck wurde gezielt eine Deadline-Schedulingdisziplin entwickelt, welche in Verbindung mit einer zweckmässigen Arbitrierung die Möglichkeit bietet, die Verzögerungen von Real-time- und Best-effort-Verkehr gegeneinander abzuwiegen. Mittels Simulation konnte bestätigt werden, dass die vorgeschlagene Methode zu

den erwünschten Resultaten führt.

Die Erkenntnisse öffnen interessante Perspektiven im Hinblick auf das Paketscheduling im Netzwerkprozessor. Während in dieser Arbeit zwecks besserer Übersichtlichkeit mit relativ einfachen Voraussetzungen bezüglich der Paket-Traces, der Paketverarbeitung und der Busarchitektur operiert wurde, sind nun weitergehende Untersuchungen angebracht. So könnten seitens der Pakete kompliziertere Verkehrsmuster mit identifizierbaren Flows, mehr als zwei Dienstklassen und variierenden Anwendungen ins Auge gefasst werden. Der Betrieb einer DMA-Einheit würde zusätzlichen PLB-Verkehr generieren und damit Scheduling und Arbitrierung vor eine neue Aufgabe stellen. Prozessorseitig wäre es interessant, den Einbezug von dedizierten Hardware-Einheiten, die neben den General-Purpose-Prozessoren spezifische Verarbeitungsschritte übernehmen könnten, zu untersuchen. Während hier nur der Run-to-completion-Modus genauer betrachtet wurde, wäre es ebenso spannend, eine Pipeline-Architektur einer Analyse zu unterwerfen.

Eine Frage, die im Rahmen dieser Diplomarbeit noch nicht ausreichend beantwortet werden konnte, ist diejenige nach der Realisierbarkeit des vorgeschlagenen Deadline-Schedulers. Hier ist eine Optimierung des vorgestellten Algorithmus anzustreben und die Möglichkeit seiner Implementierung in schneller Hardware zu prüfen.

Als Fortsetzung der getätigten Untersuchungen und zur Untermauerung der Erkenntnisse aus den Simulationen und den zum Teil heuristischen Überlegungen wäre eine analytische Betrachtung interessant. So könnten die vorgeschlagenen Lösungen mittels *Real-time Calculus* [27] erfasst werden und Voraussagen zur Schedulability und zu den Leistungsparametern (Kapazität, Speicherbedarf, Auslastung) gemacht werden. Ein analytisches Framework, das sowohl Paketscheduling als auch Arbitrierungsaspekte miteinbezieht, könnte sich als wertvolles Hilfsmittel für die Entwurfsraumexploration im Bereich der Netzwerkprozessoren erweisen.

Literaturverzeichnis

[1] B. Stiller. *Skript zur Vorlesung Protokolle für Multimedia-Kommunikation*, ETH Zürich, Departement ITET.

[2] Internet Engineering Task Force
www.ietf.org

[3] R. Barden, D. Clark und S. Schenker. *Integrated Services in the Internet Architecture*, IETF RFC 1633, 1994.

[4] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang und W. Weiss. *An Architecture for Differentiated Services*, IETF RFC 2475, 1998.

[5] T. Erlebach. *Skript zur Vorlesung Algorithmen für Kommunikationsnetze*, ETH Zürich, Departement ITET.

[6] S. Chakraborty, M. Gries und L. Thiele. *Supporting a Low Delay Best-Effort Class in the Presence of Real-Time Traffic*, 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2002, San Jose, California.

[7] M. Pinedo. *Scheduling: Theory, Algorithms and Systems*, Prentice Hall, 1995.

[8] G. C. Buttazzo. *Hard Real-Time Computing Systems*, Kluwer Academic Publishers, 1997.

[9] J. A. Stankovic, Marco Spuri und Krithi Ramamritham and Giorgio C. Buttazzo. *Deadline Scheduling for Real-Time Systems*, Kluwer Academic Publishers, 1998.

[10] S. Keshav. *An Engineering approach to computer networking*, Addison-Wesley, 1997.

[11] D. A. Patterson und J. L. Hennessy. *Computer Organization and Design*, Morgan Kaufmann Publishers, 1998.

[12] A. S. Tanenbaum und James Goodman. *Structured Computer Organization*, Prentice Hall, 1999.

[13] X. Xiao. *Providing Quality-of-Service in the Internet*, PhD thesis, Department of Computer Science and Engineering, Michigan State University, 2000.

[14] N. Shah. *Understanding Network Processors*, Technical paper, 2001.

[15] IBM PowerNP NP2G.
www-3.ibm.com/chips/techlib/techlib.nsf/products/PowerNP_NP2G

[16] Intel IXP2850.
www.intel.com/design/network/products/npfamily/ixp2850.htm

[17] Marco Platzner. *Skript zur Vorlesung Hardware Software Codesign*, ETH Zürich, Departement ITET.

[18] IBM Zurich Research Laboratory.
www.zurich.ibm.com

[19] Open SystemC Initiative, Functional Specification for SystemC 2.0.
www.systemc.org/projects/sitedocs/document/v201_Func_Spec/en/1

[20] F. Worm. *A Performance Evaluation of Memory Organizations in the Context of Core Based Network Processor Designs*, Master thesis, Institut Eurécom, Sophia-Antipolis, France, 2001 (die Arbeit wurde ausgeführt am IBM Research Laboratory Zürich).

[21] F. Pouget. *Model Design and Performance Evaluation of a Network Processor Subsystem with SystemC*, Master thesis, Institut Eurécom, Sophia-Antipolis, France, 2002 (die Arbeit wurde ausgeführt am IBM Research Laboratory Zürich).

[22] CoreConnect Bus Architecture, IBM.
www.chips.ibm.com/products/coreconnect

[23] Blue Logic Technology, IBM.
www.chips.ibm.com/bluelogic

[24] National Laboratory for Applied Network Research. Traces collected in June 2000
www.pma.nlanr.net/PMA

[25] S. Narasimhan. *Performance Analysis of Network Processor Components*, Master thesis, Department of Computer Science, University of Stuttgart, Germany, 2001 (die Arbeit wurde ausgeführt am IBM Research Laboratory Zürich).

[26] T. Wolf and M. Franklin. *CommBench – A Telecommunications Benchmark for Network Processors*, IEEE International Symposium on Performance Analysis of Systems and Software, 2000.

[27] L. Thiele, S. Chakraborty and M. Naedele . *Real-time Calculus for Scheduling Hard Real-time Systems*, Int. Symposium on Circuits and Systems ISCAS 2000, Geneva, Switzerland.

[28] M. Gries. *Algorithm-Architecture Trade-offs in Network Processor Design*, PhD thesis, ETH Zürich, Insititut für Technische Informatik und Kommunikationsnetze, 2001.