Exzellenzcluster
Cognitive Interaction Technology
Kognitronik und Sensorik
Prof. Dr.-Ing. U. Rückert

On-Chip-Netzwerk-Architekturen für eingebettete hierarchische Multiprozessoren

zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEUR (Dr.-Ing.)

der Technischen Fakultät
der Universität Bielefeld

genehmigte Dissertation

von

Dipl.-Ing. Johannes Ax

Referent: Prof. Dr.-Ing. Ulrich Rückert
Korreferent: Prof. Dr. Sybille Hellebrand

Tag der mündlichen Prüfung: 25.07.2019

Bielefeld / 2019

Kurzfassung

Das Ziel der vorliegenden Arbeit ist die Realisierung und Analyse einer skalierbaren Verbindungsstruktur für ein Multi-Prozessorsystem auf einem Chip (MPSoC). Durch die zunehmende Digitalisierung werden in immer mehr Geräten des täglichen Lebens und der Industrie mikroelektronische Systeme eingesetzt. Hierbei handelt es sich häufig um energiebeschränkte Systeme, die zusätzlich einen stetig steigenden Bedarf an Rechenleistung aufweisen. Ein Trend, diesen Bedarf zu decken ist die Integration von zunehmend mehr Prozessorkernen auf einem einzelnen Mikrochip. Many-Core-Systeme mit vielen hunderten bis tausenden ressourceneffizienten CPU-Kernen versprechen hierbei eine besonders hohe Energieeffizienz. Im Vergleich zu Systemen mit wenigen leistungsfähigen, jedoch auch komplexeren CPUs, wird bei Many-Cores die Rechenleistung durch massive Parallelität erzielt. In der AG Kognitronik und Sensorik der Universität Bielefeld wird dazu das CoreVA-MPSoC entwickelt. Um hunderte von CPUs auf einen Chip zu integrieren, verfügt das CoreVA-MPSoC über eine hierarchische Verbindungsstruktur. Diese besteht aus einem On-Chip-Netzwerk (NoC), welches eine Vielzahl von CPU-Cluster koppelt. In jedem CPU-Cluster sind mehrere ressourceneffiziente VLIW-Prozessorkerne über eine eng gekoppelte Bus-Struktur verbunden.

Der Fokus dieser Arbeit ist die Entwicklung und Entwurfsraumexploration einer ressourceneffizienten NoC-Architektur für den Einsatz im CoreVA-MPSoC. Die Entwurfsraumexploration findet dazu auf verschiedenen Ebenen statt. Auf der Ebene der Verbindungsstruktur des NoCs werden verschiedene Topologien und Mechanismen der Flusskontrolle untersucht. Des Weiteren wird die Entwicklung und Analyse eines synchronen, mesochronen und asynchronen NoCs vorgestellt, um die Skalierbarkeit und Energieeffizienz dieser Methoden zu untersuchen. Eine weitere Ebene bildet die Schnittstelle zum Prozessorsystem bzw. CPU-Cluster, die einen maßgeblichen Einfluss auf die Softwareentwicklung und Gesamtperformanz des Systems hat. Auf Systemebene wird schließlich die Anbindung verschiedener Speicherarchitekturen an das NoC vorgestellt und deren Auswirkung auf Performanz und Energiebedarf analysiert. Ein abstraktes Modell des CoreVA-MPSoCs mit Fokus auf dem NoC erlaubt die Abschätzung von Fläche, Performanz und Energie des Systems, bzw. der Ausführung von Streaming-Anwendungen. Dieses Modell kann im CoreVA-MPSoC-Compiler für die automatische Abbildung von Anwendungen auf dem MPSoC eingesetzt werden. Zehn Streaming-Anwendungen, vorwiegend aus dem Bereich der Signal- und Bildverarbeitung, zeigen bei der Abbildung auf einem CoreVA-MPSoC mit 32 CPUs eine durchschnittliche Beschleunigung um den Faktor 24 gegenüber der Ausführung auf einer CPU.

Ein CoreVA-MPSoC mit 64 CPUs und insgesamt 3 MB Speicher besitzt bei einer prototypischen Implementierung in einer 28-nm-FD-SOI-Standardzellenbibliothek einen Flächenbedarf von 14,4 mm^2. Bei einer Taktfrequenz von 700 MHz liegt die durchschnittliche Leistungsaufnahme bei 2 W. Eine FPGA-basierte Emulation auf einem FPGA-Cluster aus Xilinx Virtex-5-FPGAs erlaubt zudem eine skalierbare Verifikation eines CoreVA-MPSoCs mit nahezu beliebig vielen CPUs.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Das CoreVA-MPSoC . . . 4
- 1.3 Gliederung der Arbeit . . . 6

2 Stand der Technik von NoC-Architekturen **9**
- 2.1 Terminologie und grundlegende Eigenschaften von NoCs . . . 9
  - 2.1.1 Topologie . . . 10
  - 2.1.2 Routing-Verfahren . . . 16
  - 2.1.3 Flusskontrolle . . . 18
  - 2.1.4 Global Asynchron Lokal Synchron (GALS) . . . 21
- 2.2 NoC-Architekturen in der Forschung . . . 25
  - 2.2.1 Æthereal . . . 25
  - 2.2.2 Spidergon-STNoC . . . 26
  - 2.2.3 DSPIN/ASPIN . . . 26
  - 2.2.4 SpiNNaker . . . 27
  - 2.2.5 Tomahawk2 . . . 28
  - 2.2.6 Argo . . . 28
  - 2.2.7 GigaNetIC . . . 29
  - 2.2.8 STHORM . . . 29
  - 2.2.9 KiloCore . . . 29
- 2.3 Kommerzielle NoC-Architekturen . . . 30
  - 2.3.1 Adapteva's Epiphany . . . 30
  - 2.3.2 Kalray's MPPA . . . 31
  - 2.3.3 Intel Xeon Phi Knights Landing . . . 32
  - 2.3.4 NoC-IP-Kern Anbieter . . . 32
- 2.4 Einordnung des CoreVA-MPSoCs in den Stand der Technik . . . 33

3 Grundlagen und Werkzeugkette des CoreVA-MPSoC **37**
- 3.1 Der VLIW-Prozessor CoreVA . . . 37
- 3.2 Das Cluster im CoreVA-MPSoC . . . 41
- 3.3 Hardware-Entwurfsablauf . . . 43
  - 3.3.1 Synthese . . . 44
  - 3.3.2 Platzieren und Verdrahten . . . 45
  - 3.3.3 Simulation . . . 46

3.3.4 Analyse der Verlustleistung 47
3.3.5 Hierarchischer Entwurfsablauf 47
3.4 Software-Entwicklungsumgebung 51
3.4.1 LLVM-Compiler 51
3.4.2 Software-Basisfunktionen 52
3.4.3 Kommunikationsmodell 53
3.4.4 CoreVA-MPSoC-Compiler für Streaming-Anwendungen 55
3.4.5 Simulator 59

4 Bewertungsmaße von NoC-Architekturen 63
4.1 Analyse des Ressourcenbedarfs 63
4.1.1 Chipfläche 63
4.1.2 Leistungsaufnahme 64
4.1.3 Energiebedarf 65
4.2 Analyse der Performanz 66
4.2.1 Durchsatz 66
4.2.2 Latenz 66
4.2.3 Benchmark-Auswahl 67

5 Verbindungsstrukturen für NoC-Architekturen 73
5.1 Router für eingebettete Multiprozessoren 74
5.1.1 Stand der Technik 74
5.1.2 Implementierung eines Routers für das CoreVA-MPSoC 76
5.2 Topologien für eingebettete Multiprozessoren 80
5.2.1 Stand der Technik 80
5.2.2 Implementierung verschiedener Topologien im CoreVA-MPSoC 82
5.2.3 Vergleich verschiedener Topologien im CoreVA-MPSoC 85
5.3 GALS-Erweiterungen für eingebettete Multiprozessoren 89
5.3.1 Bezug zum Stand der Technik 89
5.3.2 Implementierung Mesochroner Router 92
5.3.3 Implementierung Asynchroner Router 96
5.3.4 Entwurfsraumexploration von GALS-Methoden 102
5.4 Zusammenfassung 110

6 Netzwerk-Schnittstellen für NoC-Architekturen 113
6.1 Stand der Technik von Netzwerk-Schnittstellen 114
6.2 Architektur einer Netzwerk-Schnittstelle für das CoreVA-MPSoC . . . 116
6.2.1 Cluster-Schnittstelle 117
6.2.2 Sendevorgang 118
6.2.3 Empfangsvorgang 119
6.2.4 Router-Schnittstelle 120

6.3 Entwurfsraumexploration von Netzwerk-Schnittstellen 121
6.3.1 Softwarekosten 121
6.3.2 Kommunikationskanäle 125
6.4 Zusammenfassung 130

7 NoC-Architekturen auf Systemebene 131
7.1 Stand der Technik von Speicherarchitekturen in MPSoCs 132
7.2 Integration von CPU-Clustern und Speicherarchitekturen 135
7.2.1 Verbindungsstrukturen im Cluster 135
7.2.2 NoC-Anbindung an eng gekoppelte Speicher 137
7.3 Entwurfsraumexploration auf Systemebene 139
7.3.1 Chipfläche 139
7.3.2 Energiebedarf 141
7.3.3 Performanz 144
7.4 Abstrakte Modellierung des CoreVA-MPSoCs 147
7.4.1 Modell für den Flächenbedarf 147
7.4.2 Ein Modell zur Abschätzung von Performanz und Energie . . . 148
7.5 Das CoreVA-MPSoC als Plattform für Echtzeitanwendungen 152
7.5.1 Stand der Technik 153
7.5.2 Echtzeit im CoreVA-MPSoC 154
7.6 Zusammenfassung 155

8 Prototypische Implementierungen des CoreVA-MPSoCs 157
8.1 ASIC-Prototyp 157
8.1.1 Makro eines Cluster-Knoten 158
8.1.2 Layout des CoreVA-MPSoCs 159
8.1.3 Testchip 161
8.2 FPGA-Prototyp 162
8.2.1 FPGA-Design 162
8.2.2 Multi-FPGA-Lösung 163
8.2.3 Demonstrator 164
8.3 Zusammenfassung 166

9 Zusammenfassung und Ausblick 167

Abbildungsverzeichnis 173

Tabellenverzeichnis 177

Abkürzungsverzeichnis 179

Referenzen 183

Eigene Veröffentlichungen **201**

Betreute Arbeiten **203**

Anhang **205**
A NoC-Grundlagen 205
B Programmbeispiele für das CoreVA-MPSoC 206
C Eigenschaften der betrachteten Beispielanwendungen 206
D Spice Simulationen asynchroner Schaltungen 208
E Ergebnisse für den CPU-Cluster 210

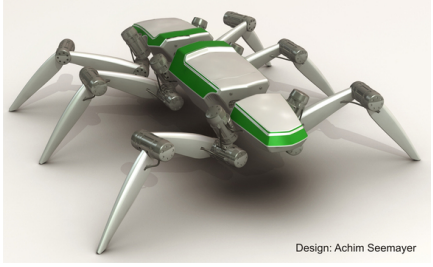
1 Einleitung

Mobile und eingebettete digitale Systeme finden in einer Vielzahl von Lebensbereichen Verwendung. Im Jahr 2020 werden 50 Milliarden vernetze Geräte (Internet der Dinge) erwartet [34]. Anwendungen wie Software-Defined Radio (SDR) [17], Sprach- oder Bilderkennung erfordern eine hohe Rechenleistung bei gleichzeitig beschränktem Energiebudget. Um ein System in verschiedenen Anwendungsfällen flexibel einsetzen zu können, nimmt die Programmierbarkeit von eingebetteten Systemen zu. Beispiele für autonome, mobile und eingebettete Systeme sind die an der Universität Bielefeld entwickelte Roboter-Stabheuschrecke Hector und der Miniroboter AMiRo[1]. Die in Abbildung 1.1a dargestellte Roboter-Stabheuschrecke Hector verfügt über zwei Kameras sowie drei Mikrofone. Der AMiRo besitzt sechs Infrarotsensoren und kann um eine Kamera erweitert werden (siehe Abbildung 1.1b). Für die Vorverarbeitung der Sensordaten sowie zur Realisierung eines autonomen Verhaltens ist eine hohe Rechenleistung notwendig.

1.1 Motivation

Insbesondere mobile Systeme erfordern eine hohe Energie- bzw. Ressourceneffizienz. Die in dieser Arbeit betrachteten Ressourcen sind – neben der Energie – Chipfläche

[1] *Autonomous Mini Robot*



(a) Roboter-Stabheuschrecke Hector

(b) Miniroboter AMiRo

Abbildung 1.1: An der Universität Bielefeld entwickelte Roboter

und Rechenleistung. Die stetigen Fortschritte in der Mikroelektronik erlauben die Integration von Milliarden Transistoren auf einem Chip [75]. Die Anzahl der Transistoren pro Chip steigt exponentiell an [118]. Diese hohe Zahl an Transistoren bedingt jedoch gesteigerte Anforderungen an die Entwurfsmethodik und Systemarchitektur von integrierten Bausteinen [94, S. 669]. Durch eine Verkleinerung der Strukturgrößen sinken zudem die Kosten pro integriertem Transistor seit einigen Jahren nicht mehr automatisch [62]. Der Wechsel eines Chipentwurfs auf kleinere Strukturgrößen ist somit nicht mehr automatisch mit geringeren Kosten pro Chip verbunden. Ein Grund hierfür sind die steigenden Kosten für die Belichtungsschritte (Lithographie) während der Herstellung von Halbleiter-Chips [43].

Anwendungsspezifische Schaltungen (ASICs[2]) sind nach der Herstellung in ihrer Funktion nicht mehr veränderbare Halbleiterbausteine. ASICs benötigen sehr wenig Energie, sind jedoch nicht flexibel für andere Algorithmen einsetzbar. Der Aufwand für den Entwurf von ASICs vergrößert sich mit steigender Transistoranzahl immens [65]. Daher ist der Einsatz von anwendungsspezifischen Schaltungen nur im Massenmarkt oder für spezielle Anwendungen, die eine sehr hohe Rechenleistung benötigen, sinnvoll.

Ein Prozessor (CPU[3]) besitzt durch Programmierbarkeit eine höhere Flexibilität im Vergleich zu ASICs. Die Leistungsfähigkeit einzelner Prozessoren ist jedoch nur bedingt durch die Verwendung einer größeren Zahl an Transistoren steigerbar. So zieht beispielsweise eine Steigerung der Taktfrequenz eine deutlich größere Leistungsaufnahme nach sich. On-Chip-Multiprozessorsysteme (MPSoC[4]) integrieren mehrere Prozessoren auf einem Silizium-Chip. Eine Anwendung kann auf einem MPSoC parallel auf mehreren CPUs ausgeführt werden. Dies erlaubt eine Erhöhung der Leistungsfähigkeit des Gesamtsystems. Alternativ werden die einzelnen CPUs niedriger getaktet und das System besitzt eine höhere Energieeffizienz. Wird eine Anwendung parallel auf mehreren CPUs ausgeführt, ist jedoch eine Synchronisierung erforderlich. Der Einsatz von Multiprozessoren sowohl in Mehrzweck-CPUs (*General-Purpose-CPUs*) als auch in eingebetteten Systemen nimmt stetig zu. Zudem steigt die Anzahl an CPU-Kernen pro System an [87].

Neben den Verarbeitungseinheiten stellen jedoch auch die On-Chip-Verbindungsstrukturen eine zunehmend kritische Größe für die Ressourceneffizienz eines Multiprozessorsystems dar [49]. Klassische Einprozessorsysteme verwenden typischerweise einen geteilten Bus für die Kommunikation der CPU mit dem Speicher und weiterer Peripherie. Es kann nur eine Transaktion gleichzeitig erfolgen. Bei einfachen Multiprozessorsystemen sind die CPU-Kerne ebenfalls an einen geteilten Bus angeschlossen. In größeren Systemen wird dieser Bus jedoch schnell ein begrenzender Faktor für die Leistungsfähigkeit des Gesamtsystems. Schaltmatrizen (*Switching-Matrix, Crossbar*) ermöglichen durch Punkt-zu-Punkt-Verbindungen die gleichzeitige Kommunikation

[2] *Application Specific Integrated Circuit*
[3] *Central Processing Unit*
[4] *Multiprocessor System On a Chip*

zwischen verschiedenen Komponenten im System, haben jedoch einen hohen Ressourcenbedarf [99]. Ein weiterer Ansatz ist die Verwendung von On-Chip-Netzwerken (NoC[5]) [15]. Hierbei versenden die Kommunikationspartner Daten als Pakete, die von Netzwerkknoten (*Routern*) bis zum Ziel weitergeleitet werden. NoCs skalieren sehr gut für große Systeme mit mehreren dutzend oder hundert Kommunikationspartnern. Die Router eines NoCs können jedoch einen deutlichen Mehraufwand in Bezug auf Chipfläche und Verzögerungszeit bedeuten [41]. Aus diesem Grund werden hierarchische Kommunikationsstrukturen eingesetzt, bei denen eine Gruppe von CPUs (*Cluster*) über einen geteilten Bus oder eine Crossbar verbunden ist. Mehrere Cluster kommunizieren über ein NoC. Eine geringe Übertragungslatenz innerhalb eines Clusters bietet zudem Vorteile für Algorithmen, bei denen mehrere CPUs häufig untereinander kommunizieren müssen [41]. Diese Algorithmen werden möglichst auf den CPUs eines Clusters abgebildet, um eine hohe Verarbeitungsgeschwindigkeit sowie eine geringe Kommunikation zwischen den Clustern zu erreichen.

Klassische Multiprozessorsysteme (*Multi-Cores*) verwenden vergleichsweise wenige (bis ca. 24) leistungsstarke CPU-Kerne. Beispiele sind Prozessoren von Intel (Xeon E5 [22]) oder IBM (Power9 [152]). Die Integration von immer mehr dieser leistungsstarken CPUs auf einem Chip ist aufgrund der hohen Leistungsaufnahme nicht für alle Einsatzzwecke sinnvoll, wie Borkar [21] anführt. Im Gegensatz zu Multi-Core-Systemen bestehen *Many-Core*-MPSoCs aus einer sehr großen Anzahl kleiner CPU-Kerne, die im Vergleich Anwendungen ressourceneffizienter ausführen können. Dies wird erreicht, indem mehr Transistoren (bzw. Chipfläche) für die eigentlichen Rechenwerke und weniger für z. B. On-Chip-Caches verwendet werden [140, S. A-10 f.]. Zudem verfügt die Mehrzahl der Many-Core-Architekturen über keine zentralisierte Speicherarchitektur (*Non-Uniform Memory Access*, NUMA) und/oder verwendet explizite Kommunikation. Durch die dynamische Reduzierung des Taktes und der Versorgungsspannung einzelner CPU-Kerne (*Dynamic Voltage and Frequency Scaling*, DVFS) sowie das Abschalten gerade nicht verwendeter CPUs (*Power Gating*) kann die Leistungsfähigkeit des MPSoCs dynamisch an die Anforderungen der jeweils ausgeführten Anwendung angepasst werden. Dies kann bei der Verwendung von kleinen CPU-Kernen wesentlich feingranularer als bei wenigen leistungsstarken Prozessoren erfolgen.

Weitere Unterscheidungsmerkmale zwischen klassischen Multiprozessorsystemen und Many-Cores sind die Speicherhierarchie sowie die Speicherverwaltung. Klassische Multiprozessorsysteme besitzen einen gemeinsamen Adressraum für alle CPUs, mehrere (private und/oder gemeinsame) Caches sowie einen externen DRAM[6]-Speicher. Dies ermöglicht eine einfache Programmierung, bedingt jedoch einen signifikanten Mehrbedarf an Fläche und Leistungsaufnahme durch Cache-Logik und Kohärenzverwaltung. Balfour et al. zeigen in [11], dass ein eingebetteter SPARC V8 Prozessor 70 % seines Energiebedarfs für die Bereitstellung und Behandlung (Dekodierung) von

[5] *Network on a Chip*
[6] *Dynamic Random Access Memory*

Instruktionen und Daten benötigt. Eine selbst entwickelte, optimierte CPU benötigt hierfür immerhin 51 % der Energie. Dieses Problem vergrößert sich, da der Anteil an Speicher an der Gesamtfläche von MPSoCs kontinuierlich ansteigt [75]. Software-gesteuerte Speicher (*Scratchpad-Memories*) können eine ressourceneffiziente Alternative zu Caches darstellen [13]. Die CPUs im Multiprozessorsystem können direkt Daten über einen Scratchpad-Speicher austauschen. Alternativ kann ein DMA[7]-Controller Daten zwischen verschiedenen Scratchpad-Speichern transferieren. Mit steigender Anzahl an CPUs wird ein gemeinsamer Speicher, sei es ein Cache oder ein Scratchpad-Speicher, jedoch zunehmend zu einem Flaschenhals bezüglich der Übertragungsbandbreite [188]. Eine nachrichtenbasierte Kommunikation umgeht diesen Flaschenhals, indem Synchronisation und Daten direkt zwischen einzelnen CPUs ausgetauscht werden [183].

Eine effiziente Partitionierung einer Anwendung auf dutzende oder hunderte Prozessorkerne stellt eine große Herausforderung für Softwareentwickler dar. Scratchpad-basierte Speicherarchitekturen erfordern eine explizite Verwaltung der verschiedenen Speicher durch die Anwendungssoftware. Beides führt dazu, dass die Portabilität von parallelen Anwendungen, die in klassischen Programmiersprachen wie C oder C++ beschrieben sind, stark eingeschränkt ist. Um die Programmierung eines Many-Core-MPSoCs zu unterstützen, ist die Verwendung eines einheitlichen Programmiermodells für Inter- und Intra-CPU-Kommunikation sinnvoll. Eine weitere Vereinfachung der Anwendungsentwicklung ist möglich, indem Anwendungen in speziellen parallelen Programmiersprachen (z. B. OpenCL[8] [72] oder StreamIt [179]) beschrieben werden. Dies ermöglicht eine automatisierte Partitionierung von Anwendungen auf die Prozessoren eines MPSoCs. Parallele Programmiersprachen können den Programmierer auch von der expliziten Verwaltung von Scratchpad-Speichern entlasten [13].

1.2 Das CoreVA-MPSoC

Die Arbeitsgruppe Kognitronik und Sensorik [6] der Universität Bielefeld entwickelt das ressourceneffiziente Many-Core System CoreVA-MPSoC.

Das CoreVA-MPSoC wird insbesondere für die effiziente Verarbeitung von Streaming-Anwendungen in eingebetteten, energiebeschränkten Systemen entwickelt. Beispiele für Streaming-Anwendungen sind in der Signalverarbeitung, wie SDR [205; 211], aber auch in der Bildverarbeitung zu finden. Allgemein können alle Anwendungen, die sich durch eine kontinuierliche Verarbeitung von Datenströmen durch verschiedene Verarbeitungsschritte (Tasks) auszeichnen, als Streaming-Anwendung beschrieben werden.

Das CoreVA-MPSoC verfügt über eine hierarchische Verbindungsstruktur, wie in Abbildung 1.2 ersichtlich ist. In den Dissertationen [78] und [73] wurde bereits die CoreVA-CPU vorgestellt, die im CoreVA-MPSoC als Verarbeitungseinheit eingesetzt wird.

[7] *Direct Memory Access*
[8] *Open Computing Language*

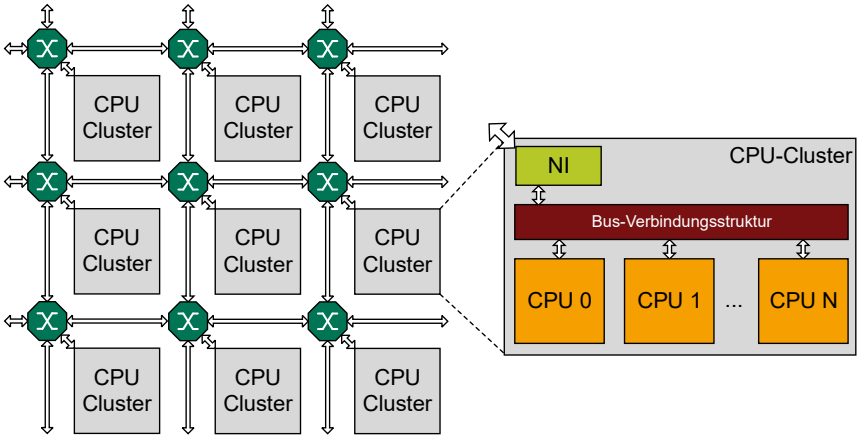


Abbildung 1.2: Blockschaltbild des hierarchischen CoreVA-MPSoCs

Die Verschaltung mehrerer CoreVAs zu einem CPU-Cluster wurde in der Dissertation [162] behandelt. Im Rahmen dieser Arbeit wird nun die Verschaltung von beliebig vielen CPU-Clustern mittels NoC zu einem skalierbaren MPSoC adressiert.

Der CPU-Kern des CoreVA-MPSoC basiert auf der 32 bit VLIW[9]-Architektur CoreVA[10], welche für eine hohe Ressourceneffizienz entworfen ist. Die CoreVA-CPU verfügt über Parallelität auf Daten- und Instruktionsebene (DLP[11] und ILP[12]), durch die Verwendung von mehreren VLIW-Slots sowie die Integration von SIMD[13]-Rechenwerken. Die Anzahl der VLIW-Slots und deren Verarbeitungseinheiten kann zur Entwurfszeit konfiguriert werden [74]. Die CoreVA-CPU besitzt getrennte Speicher für Instruktionen und Daten sowie sechs Pipeline-Stufen. Weitere Details zur CoreVA-CPU sind in Abschnitt 3.1 und den Dissertationen [78] und [73] zu finden.

Parallelität auf Thread-Ebene (TLP[14]) wird im hierarchischem Many-Core CoreVA-MPSoC angewandt. Innerhalb eines CPU-Clusters kommunizieren die einzelnen CPUs untereinander mit geringer Latenz und hoher Übertragungsbandbreite. Zudem ist der Ressourcenbedarf einer busbasierten Kopplung innerhalb eines CPU-Clusters im Vergleich zu einem On-Chip-Netzwerk (NoC) mit einer CPU pro Knoten geringer [41]. Bei der Kommunikationsstruktur im Cluster CoreVA-MPSoC kann zur Entwurfszeit der Bus-Standard (AXI, Wishbone), die Datenbusbreite und die Topologie (geteilter Bus,

[9] *Very Long Instruction Word*
[10] *Configurable Ressource Efficient VLIW Architecture*
[11] *Data Level Parallelism*
[12] *Instruction Level Parallelism*
[13] *Single Instruction Multiple Data*
[14] *Thread Level Parallelism*

Punkt-zu-Punkt) konfiguriert werden [210]. Weitere Details zu dem CPU-Cluster des CoreVA-MPSoC sind in Abschnitt 3.2 und der Dissertation [162] zu finden.

Wie in [210] gezeigt, stößt die Bus-basierte Kopplung jedoch bei der Integration von immer mehr CPUs auf einem Chip an ihre Grenzen. Aus diesem Grund wird im CoreVA-MPSoC ein On-Chip-Netzwerk (NoC) als zweite Verbindungsstrukturebene verwendet, welches der Fokus dieser Arbeit ist. Das NoC verbindet mehrere CPU-Cluster und erlaubt so die Integration von vielen hundert CPU-Kernen auf einem einzelnen Chip. Die Ansprüche, die an das NoC für das CoreVA-MPSoC gestellt werden, sind daher neben einer ressourceneffizienten Kommunikationsstruktur, auch die effiziente Skalierbarkeit des Gesamtsystems. Des Weiteren stellt auch die hierarchische Struktur des CoreVA-MPSoC gesonderte Ansprüche an das NoC.

Zur Programmierung des CoreVA-MPSoCs steht ein LLVM[15]-basierter Übersetzer für die Programmiersprache C zur Verfügung. Um eine einfache Programmierung von Anwendungen für verschiedene MPSoC-Konfigurationen zu ermöglichen, ist im Rahmen dieser Arbeit und in Zusammenarbeit mit der Dissertation [162] ein einheitliches Programmiermodell entstanden, welches in Abschnitt 3.4.3 vorgestellt wird. Dieses Modell erlaubt die Kommunikation zwischen Anwendungsteilen, die auf CPUs im gleichen Cluster oder in verschiedenen Clustern ausgeführt werden. Die manuelle Partitionierung von Anwendungen auf ein Multiprozessorsystem stellt hohe Anforderungen an den Entwickler. Für eine automatisierte Partitionierung von Anwendungen auf die CPUs des CoreVA-MPSoCs stehen Compiler für die Sprachen OpenCL [216] und StreamIt [179] zur Verfügung. Der StreamIt-Compiler ist im Rahmen einer Kooperation mit der Queensland University of Technology (QUT) in Brisbane, Australien entstanden [206]. Ein StreamIt-Programm besteht aus mehreren Tasks (auch Filter genannt), die über unidirektionale Kanäle miteinander kommunizieren. Filter können parallel ausgeführt und daher auf verschiedene CPUs partitioniert werden.

1.3 Gliederung der Arbeit

In Kapitel 2 wird der aktuelle Stand der Technik von NoC-Architekturen vorgestellt. Dazu werden zunächst die Grundlagen und allgemeinen Eigenschaften von NoC-Architekturen vorgestellt, die als Basis für diese Arbeit gelten. Anschließend werden verschiedene NoC-Architekturen für MPSoCs aus der Forschung und Industrie vorgestellt, die der Einordnung des CoreVA-MPSoCs in den Stand der Technik dienen.

Die Systemgrundlagen und die Werkzeugkette des CoreVA-MPSoC werden in Kapitel 3 beschrieben. Zunächst werden die Hardwarearchitektur der CoreVA-CPU und die des CPU-Clusters vorgestellt. Anschließend wird der Hardware-Entwurfsablauf beschrieben, der im Rahmen dieser Arbeit entwickelt und eingesetzt wurde. Als Zieltechnologie für den Hardwareentwurf wird eine 28-nm-FD-SOI[16] Standardzellenbibliothek verwendet.

[15] *Modulare Compilerarchitektur, früher Low Level Virtual Machine*
[16] *Fully-Depleted Silicon-on-Insulator*

Des Weiteren werden in Kapitel 3 die Software-Entwurfsumgebung für die Sprachen C und StreamIt sowie die Simulations- und Emulationsumgebung des CoreVA-MPSoCs vorgestellt.

In Kapitel 4 werden verschiedene Bewertungsmaße für NoC-Architekturen in eingebetteten Multiprozessoren dargestellt und definiert. Diese Bewertungsmaße erlauben eine Entwurfsraumexploration verschiedener Systemkomponenten von NoC-Architekturen, die im weiteren Verlauf der Arbeit betrachtet werden. Bewertungsmaße sind neben der Chipfläche und Energie, die den Ressourcenbedarf von eingebetteten Multiprozessoren beschreiben, auch Bewertungsmaße für die Performanz von NoC-Architekturen. Die Performanz kann durch Größen wie die Latenz und Durchsatz bewertet werden. Zusätzlich werden in Kapitel 4 Benchmarks aus dem Bereich von Streaming-Anwendungen, wie z.B. aus der Signal- und Bildverarbeitung vorgestellt. Diese erlauben eine Bewertung des NoC auf Systemebene für verschiedene Anwendungen.

Allgemein lassen sich NoCs für Multiprozessorsystemen in zwei Hauptkomponenten gliedern. Dies ist zum einen die NoC-Verbindungsstruktur, welche die Netzwerkknoten (*Router*) und die Verbindungsstruktur zwischen ihnen beschreibt. Zum anderen ist es die Netzwerkschnittstelle, die sich zwischen CPU bzw. CPU-Cluster und Router befindet.

In Kapitel 5 werden verschiedene Architekturkonzepte und Konfigurationen von NoC-Verbindungsstrukturen für eingebettete Multiprozessoren, wie das CoreVA-MPSoC diskutiert. Zunächst werden verschiedene Router-Architekturen und -Konzepte betrachtet. Anschließend wird die Anordnung und Verbindung dieser Router durch verschiedene Topologien vorgestellt. Das Kapitel schließt mit der Analyse verschiedener GALS[17]-Methoden, die sich für NoC-basierte Mulitprozessoren ergeben.

Netzwerkschnittstellen für NoC-Architekturen werden in Kapitel 6 untersucht. Diese Schnittstellen, die auch als Netzwerk-Interface (NI) bezeichnet werden, haben einen großen Einfluss auf Effizienz von NoC-Architekturen in MPSoCs. Der Fokus wird in dieser Arbeit auf das Zusammenspiel von Software und Hardware gelegt, welches für die NoC-Kommunikation insbesondere im NI umgesetzt wird. Es wird zunächst Bezug zum Stand der Technik von NI-Architekturen und den dazugehörigen Kommunikationsmodellen genommen. Anschließend werden Architekturkonzepte und Konfigurationen vorgestellt und analysiert, die für den NI des CoreVA-MPSoCs umgesetzt wurden.

In Kapitel 7 wird die NoC-Architektur schließlich auf Systemebene betrachtet. Als Systemebene ist dabei das gesamte System eines MPSoCs zu verstehen, bestehend aus CPUs, Speicher und allen Verbindungsstrukturen, wie die innerhalb eines CPU-Clusters und das NoC (siehe Abbildung 1.2). Das Kapitel beginnt mit der Integration von CPU-Clustern in ein NoC und einer Entwurfsraumexploration verschiedener MPSoC-Konfigurationen und Speicheranbindungen auf Systemebene. Diese Analyse von NoC-Architekturen auf Systemebene ermöglicht eine Aussage über dessen Leistungsfähigkeit im Gesamtsystem. Des Weiteren wird in Kapitel 7 eine abstrakte Modellierung des CoreVA-MPSoCs mit Schwerpunkt NoC vorgestellt. Dieses Modell kann und wird

[17] *Global Asynchron Lokal Synchron*

bereits in Arbeiten am CoreVA-MPSoC-Compiler zur automatischen Partitionierung von StreamIt-Anwendungen auf dem CoreVA-MPSoC eingesetzt. Das Kapitel schließt mit einer Einordnung des CoreVA-MPSoCs als Plattform für Echtzeitanwendungen.

Prototypische Implementierungen ausgewählter Konfigurationen des CoreVA-MPSoCs mit NoC werden in Kapitel 8 vorgestellt. Dies ist zum einen die prototypische Implementierung eines Chip-Layouts in einer 28-nm-FD-SOI Standardzellenbibliothek von STMicroelectronics. Zum anderen wird ein FPGA[18]-basierter Prototyp auf dem System RAPTOR vorgestellt, bei dem ein CoreVA-MPSoC mit Hilfe mehrerer FPGA-Module prototypisch implementiert und skaliert werden kann.

Die Zusammenfassung dieser Arbeit ist in Kapitel 9 zu finden. Neben einem Fazit wird hier auch ein Ausblick auf mögliche zukünftige Arbeiten gegeben.

[18] *Field Programmable Gate Array*

2 Stand der Technik von NoC-Architekturen

Dieses Kapitel präsentiert den aktuellen Stand der Technik von NoC-Architekturen. Als Basis für diese Arbeit werden in Abschnitt 2.1 zunächst die Grundlagen und allgemeinen Eigenschaften von NoC-Architekturen vorgestellt. Anschließend werden verschiedene NoC-Architekturen aus der Forschung betrachtet (siehe Abschnitt 2.2). Dies sind neben MPSoCs auch reine NoC-Architekturen. In Abschnitt 2.3 werden kommerzielle MPSoCs und NoC-Architekturen vorgestellt, die aktuell auf dem Markt verfügbar sind. Das Kapitel schließt mit einer Einordnung des CoreVA-MPSoCs in den Stand der Technik vergleichbarer MPSoC-Chips (siehe Abschnitt 2.4).

2.1 Terminologie und grundlegende Eigenschaften von NoCs

Dieser Abschnitt gibt einen Überblick über die wichtigsten NoC-Eigenschaften, die als Grundlage für die in dieser Arbeit behandelten Themen dienen. Viele dieser NoC-Eigenschaften sind bereits in der Literatur beschrieben [40][77][138][76][4][139] und ausführlich in Arbeiten wie [127] dargestellt. Dieser Abschnitt fasst jedoch noch einmal die wichtigsten Eigenschaften von NoCs für den Einsatz in eingebetteten MPSoCs zusammen.

Alle Kommunikationsnetze und Verbindungsstrukturen haben gemein, dass sie mehreren Teilnehmern erlauben Daten miteinander auszutauschen. Diese Teilnehmer werden bei Kommunikationsnetzen häufig als Endknoten bezeichnet. Endknoten können bei On-Chip-Verbindungsstrukturen sowohl aktive Teilnehmer, wie CPUs oder Hardwarebeschleuniger, als auch passive Teilnehmer wie Speicher sein.

Die klassische Verbindungsart von verschiedenen Endknoten auf einem Chip sind Bussysteme. Klassische Bussysteme sind passive Verbindungsstrukturen, die mehrere aktive Komponenten miteinander verbinden. Zumeist teilen sich alle Endknoten ein Medium, sodass das Bussystem eine Arbitrierung durchführen muss, oder es bestehen Punkt-zu-Punkt-Verbindungen zwischen allen Endknoten. Der Nachteil beider Varianten ist eine schlechte Skalierbarkeit. Werden hunderte von Endknoten auf einem Chip eingesetzt, kann der geteilte Bus zu starken Performanzeinschränkungen führen. Bei Punkt-zu-Punkt-Verbindungen wiederum stellt sich aufgrund der vielen Verbindungs-

leitungen ein hoher Flächen- und Energiebedarf ein. Klassische Bussysteme werden im CoreVA-MPSoC innerhalb eines CPU-Clusters eingesetzt und sind bereits in der Dissertation [162] untersucht.

Ein NoC hingegen bietet eine gute Skalierbarkeit, da es als Kompromiss zwischen den beiden Bussystemen anzusehen ist. Aktive Elemente (sog. Routingknoten oder kurz Router) bieten eine Leitweglenkung (engl. Routing) innerhalb der Kommunikationsstruktur. Dies wiederum ermöglicht es, verschiedene Verbindungsstrukturen zwischen den Endknoten zu realisieren, die einen optimalen Kompromiss zwischen Performanz, Energie- sowie Flächenbedarf bieten können.

Durch Router können verschiedene Verbindungsstrukturen in Form von *Topologien* zwischen Endknoten realisiert werden. Dies erfordert das sogenannte *Routing*, welches den Weg von Datenpaketen durch die Verbindungsstruktur festlegt, die von einem Quellknoten zu einem Zielknoten übertragen werden müssen. Des Weiteren stehen bei der Realisierung von NoCs verschiedene *Switching-Verfahren* zur Verfügung. Als Switching-Verfahren wird die technische Realisierung bezeichnet, auf welche Art und Weise Datenpakete von einem Endknoten zum nächsten übergeben werden.

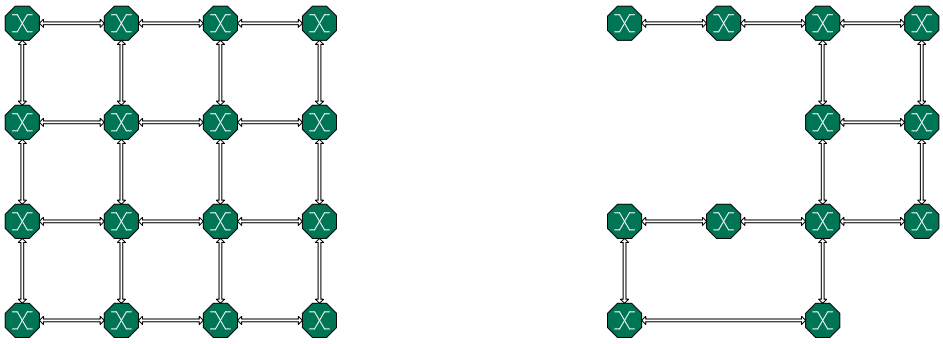
Die grundlegenden Eigenschaften von NoCs lassen sich damit in *Topologie*, *Routing* und die *Flusskontrolle* gliedern. Neben diesen drei grundlegenden Eigenschaften spielt auch die physikalische Umsetzung des NoCs eine wichtige Rolle. Um die Skalierbarkeit der NoCs auch physikalisch umzusetzen, ist häufig eine Implementierung des Systems als GALS nötig. Die Grundlagen und verschiedene Varianten dieser fünf Eigenschaften und Techniken werden in den folgenden Abschnitten aufgeführt.

2.1.1 Topologie

Als Topologie wird der physikalische Verbindungsaufbau eines Netzwerkes bezeichnet [38]. Der Begriff Topologie hat seinen Ursprung im Bereich der Off-Chip-Verbindungsnetzwerke und gibt an, wie Endknoten zueinander angeordnet und miteinander verbunden sind. Aufgrund anderer Anforderungen von On-Chip-Netzwerken, müssen die Ausführungen und Eigenschaften der verschiedenen Topologien für den Einsatz im NoC jedoch differenziert bewertet werden [14]. Insbesondere bei Chips, die in eingebetteten Systemen eingesetzt werden, spielen Fläche und Leistungsaufnahme, die sich durch die entsprechende Topologie ergeben, eine entscheidende Rolle für die Bewertung der Topologie.

Dieser Abschnitt erörtert den Aufbau zur Klassifizierung und Unterscheidung der Topologien. Im Rahmen dieser Arbeit wird eine allgemeine Zusammenfassung über die Klassifizierung von Topologien gegeben.

Es existieren zwei grundlegende Klassifizierungen, die die Grundstruktur der Topologie bestimmen. Zum einen die Klassifizierung in *direkte* und *indirekte* Topologien [139, Kapitel 12]. Dabei charakterisiert sich eine direkte Topologie durch die direkte Verbindung der *Endknoten* mit dem Netzwerk. Das bedeutet, dass der Endknoten und der angebundene Router als eine Einheit angesehen werden. Bei der indirekten Topologie



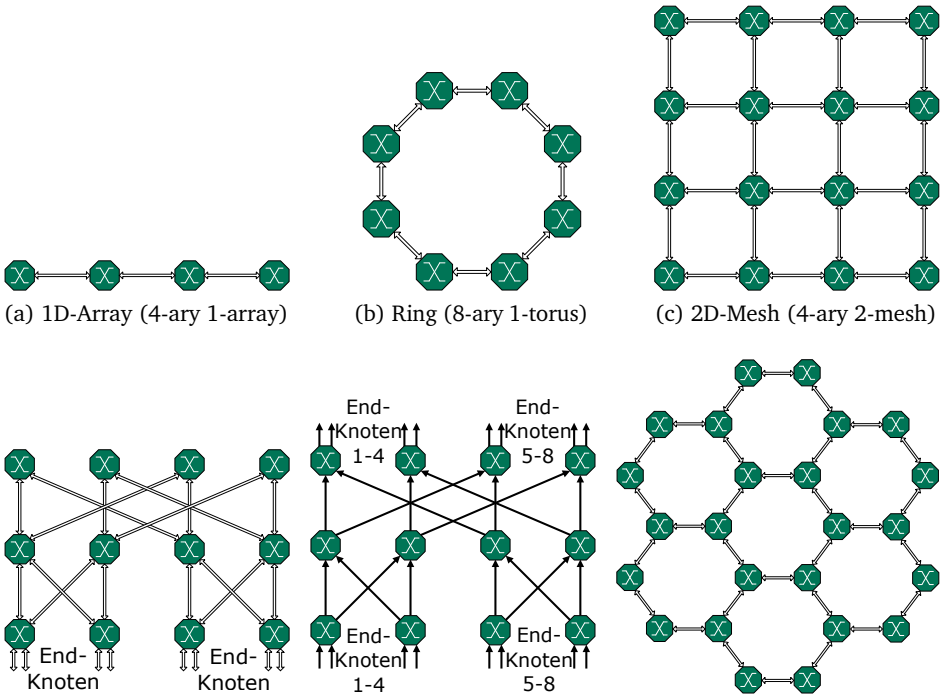
(a) Geordnete Topologie (b) Ungeordnete Topologie

Abbildung 2.1: Klassifizierung am Beispiel einer 2D-Mesh Topologie

hingegen sind die Endknoten über Leitungen miteinander verbunden. Dementsprechend können mehrere Endknoten an einem Router angebunden sein. Zum anderen die Einteilung in die Klassifizierung *geordnete* und *ungeordnete* Topologien. Eine geordnete Topologie besitzt ein vordefiniertes Verbindungsmuster und wird oft aufgrund der guten Skalierbarkeit sowie der Wiederverwendbarkeit gewählt (Abbildung 2.1a). Ein nicht vordefiniertes Verbindungsmuster zeichnet die ungeordnete Topologie aus, die hauptsächlich unter speziellen Anforderungen vorzufinden ist (Abbildung 2.1b). Für homogene MPSoCs bietet sich aufgrund der guten Skalierbarkeit und der Homogenität eine geordnete Topologie an, wohingegen spezialisierte Chips mit vielen verschieden Spezialkomponenten (Intellectual Property (IP) Blöcke) häufig eine ungeordnete Topologie aufweisen.

Insgesamt existiert eine Vielzahl verschiedener Topologietypen, die zusätzlich verschiedene Ausführungen besitzen können. Im Rahmen des EU-Projekts NaNoC wurde bereits eine ausführliche Analyse von Topologien speziell für homogene MPSoCs durchgeführt [121]. Hier findet eine Beschränkung auf direkte und geordnete Topologien statt. Um die verschiedenen Topologien formal beschreiben und voneinander abgrenzen zu können, existiert eine Definition über die drei charakteristischen Parameter, die die Form beziehungsweise Struktur der meisten Topologie bestimmen. Die *Dimension n* beschreibt in wie viele Richtungen sich die Topologie ausdehnt. Im *n-Tupel* $\langle k_1, k_2, ..., k_n \rangle$ definiert k_i die Anzahl der Routingknoten in der jeweiligen Dimension i. Als dritter Parameter legt m die Anzahl der Endknoten mit eigenem Anschluss an jedem Router fest. Es gilt $n, k, i, m \in \mathrm{N} \setminus \{0\}$. Daraus folgt die allgemeine Form *k-ary n-Topologie*, wobei hier zu beachten ist, dass diese Form aufgrund der *k-ary* Struktur nur geordnete Topologien einschließt. [40]

Da sich nicht alle Topologien allein mit den oben genannten Parametern n, k und m formulieren lassen, werden hier weitere Parameter eingeführt, die jedoch zum Teil



(a) 1D-Array (4-ary 1-array) (b) Ring (8-ary 1-torus) (c) 2D-Mesh (4-ary 2-mesh)

(d) Baum (2-ary 3-tree) (e) Butterfly (2-ary 3-fly) (f) Honeycomb (Honeyc.-mesh)

Abbildung 2.2: Beispiele einiger typischer Topologien

nur einmalig vorkommen. In der Vorstellung der Topologien werden die relevanten Parameter in dem jeweiligen Abschnitt zum besseren Verständnis namentlich benannt. Die Gesamtanzahl der Routingknoten wird als Parameter N definiert, der sich bei rechtwinkligen Topologien aus Breite B und Länge L, $B \cdot L$, errechnen lässt. Des Weiteren wird der Parameter e für den Kantengrad der Routingknoten ohne Endknoten und der Parameter l für das Ausdehnungslevel der Topologie verwendet. Bei Topologien basierend auf einer Baumstruktur dient der Parameter h für die Stufenanzahl. Der Parameter t gibt die Anzahl an Ringen aus Hexagone an. Es gilt des Weiteren $N, B, L, e, l, h, t \in \mathrm{N} \setminus \{0\}$ und $h > 1$.

In Abbildung 2.2 sind beispielhaft einige der verbreitetsten Topologien abgebildet. Sind Endknoten hier nicht explizit angegeben, bedeutet dies, dass sich an jedem Routingknoten ein Endknoten befindet. Die Topologie der Form 4-ary 1-array (siehe Abbildung 2.2a) ist mit vier Routingknoten angeordnet in einem linearen 1D-Array die einfachste Form aller dargestellten Topologien. Eine Erweiterung dieser Topologie als

Torus der Form 8-ary 1-torus stellt Abbildung 2.2b dar, welche auch als Ring-Topologie bezeichnet wird. In Abbildung 2.2c ist mit der Form 4-ary 2-mesh eine 2D-Mesh Topologie bestehend aus 16 Knoten zu sehen. Eine Baumtopologie des Typs Fat-Tree ist in Abbildung 2.2d mit der Form 2-ary 3-tree dargestellt. Recht ähnlich, jedoch mit unidirektionalen Verbindungen, ist die Schmetterlings-Topologie (engl. Butterfly) in Abbildung 2.2e der Form e-ary h-fly bestehend aus 12 Knoten. Die Topologie in der Honigwaben-Form (engl. Honeycomb) ist nicht in dem Schema k-ary n-Topologie darstellbar und wird als Mesh mit 24 Knoten beispielhaft in Abbildung 2.2f dargestellt. [40][121]

Eine Topologie zeichnet sich durch neun verschiedene Eigenschaften aus. Je nach Anforderung und Einsatzgebiet des MPSoCs müssen diese Eigenschaften unterschiedlich bewertet werden, um so eine geeignete Topologie zu finden. Bei den Anforderungen handelt es sich um die *Ausfallsicherheit*, die benötigte *Fläche*, die *Leistungsaufnahme*, die *Latenz* und die *Belastungskapazität* [127]. Die im Folgenden aufgeführten neun Eigenschaften haben jeweils Auswirkungen auf die zuvor genannten Anforderungen.

Symmetrie Eine Topologie heißt *symmetrisch*, wenn sie aus der Perspektive jedes Routers exakt gleich aussieht. Symmetrie bietet den Vorteil, dass das Routing-Verfahren in allen Knoten nahezu gleich zu handhaben ist und nicht lokal auf Individualitäten der Struktur reagiert werden muss [127]. Ungeordnete Topologien, das heißt, der Wert von k ist über die Dimensionen unterschiedlich, sind grundlegend asymmetrisch.

Bisektionsbandbreite Der Wert der *Bisektionsbandbreite* verdeutlicht, wie viele Verbindungen in der Topologie getrennt werden müssen, damit das Netz in zwei gleichgroße Netze geteilt werden kann. Je mehr Verbindungen getrennt werden müssen, desto größer ist die Bisektionsbandbreite. Ein großer Wert bedeutet eine bessere Behandlung von hohen Datenverkehrsaufkommen im Netzwerk, da eine Kollision von Datenpaketen unwahrscheinlicher wird. Er kann als Belastungskapazität des Netzwerkes angesehen werden.

Kantengrad Der Kantengrad gibt die maximale Anzahl aller Ports des Routers an. Je höher der *Kantengrad* ist, desto bessere Werte können bei der Bisektionsbandbreite erzielt werden. Jedoch bedeutet dies auch immer eine größere Chipfläche, die für die Router und Verbindungen aufgewendet werden muss. Dies führt häufig auch zu einer niedrigeren maximalen Betriebsfrequenz oder höheren Verlustleistung des Routers. Stark beeinflusst wird der Kantengrad durch die Dimension n.

Homogenität Eine Topologie heißt *homogen*, wenn alle Router den gleichen Kantengrad, das bedeutet die gleiche Anzahl an Ports, vorweisen.

Hop-Anzahl Die *Hop-Anzahl* gibt die Menge der Router an, die auf dem kürzesten Pfad zwischen den zwei maximal entferntesten Endknoten passiert werden müssen (Worst-Case). Je geringer die Hop-Anzahl eines Datenpakets auf dem Weg vom

Quellknoten zum Zielknoten im Netzwerk ist, desto geringer ist die Latenz im Netzwerk. Angegeben wird hier häufig die durchschnittliche Hop-Anzahl aller Datenpakete, wenn jeder Knoten mit jedem anderen Knoten im NoC kommuniziert.

Durchmesser Der *Durchmesser* gibt die kürzeste Länge zwischen den zwei maximal entferntesten Endknoten im Netzwerk an. Er unterscheidet sich zur Eigenschaft Hop-Anzahl in der absoluten Länge der Verbindungen. Ein Ausdehnen einer Topologie führt demnach zu einem erhöhten Durchmesser bei gleichbleibender Hop-Anzahl. Längere Verbindungen beinhalten eine größere Verzögerungszeit, die eventuell mehrere Hops benötigen. Aufgrund dessen sind kleinere Durchmesser und gleichlange Verbindungen bevorzugt.

Konnektivität Hierbei beschreibt der Wert der *Konnektivität*, wie viele Verbindungen im Netzwerk mindestens durchtrennt sein müssen, damit ein Router vom Netzwerk getrennt wird. Folgendermaßen ist ein hoher Konnektivitätswert ein Indikator für die maximale Toleranz defekter Verbindungen und für die Ausfallsicherheit der Topologie.

Gesamtanzahl Router Die *Gesamtanzahl an Routern* gibt an, wie viele Router benötigt werden, sodass alle Endknoten über eine Topologie miteinander verbunden sind. Eine hohe Anzahl erhöht den Ressourcenbedarf, der für ein vollständiges NoC benötigt wird.

Gesamtanzahl Netzwerkverbindungen Die *Gesamtanzahl an Netzwerkverbindungen* beschreibt die minimale Menge an unidirektionalen Verbindungen, damit alle Router über eine Topologie miteinander verbunden sind. Diese Eigenschaft ist daher für die Implementierung ein nützlicher Hinweis, um unmittelbar einen Eindruck von der Gesamtanzahl der benötigten Ports und somit ein Indiz auf die Fläche zu erhalten. Die Portanzahl setzt sich aus der absoluten Menge an unidirektionalen Netzwerkverbindungen und der absoluten Menge an Endknoten zusammen.

Die Eigenschaften der verschiedenen Topologien sind in Tabelle 2.1 zusammengetragen. Für einige Topologien lassen sich keine Werte der durchschnittlichen Hop-Anzahl bestimmen. Diese sind mit *N/A* gekennzeichnet. Sie lassen sich allerdings mittels der allgemeinen Formel $H_{avg} = \frac{1}{N^2} \sum_{x,y \in \mathrm{N}} H(x,y)$ berechnen [127]. N bezeichnet hierbei die Gesamtanzahl der Knoten. Alle zuvor genannten Eigenschaften und Parameter sind im Anhang A in Tabelle A.1 zusammengefasst. Eine konkrete Interpretation der Tabelle für ein NoC mit 64 Endknoten ist in Kapitel 5.2 aufgeführt. Weitere Details und eine Visualisierung der verschiedenen Topologien sind in der betreuten Bachelorarbeit [219] zu finden.

Um Topologien sinnvoll auswählen zu können, ist die Auswirkung der jeweiligen Eigenschaften auf die Anforderung ausschlaggebend. Die Ausfallsicherheit lässt sich

Topologie	Kantengrad	Symmetrie	Homogenität	Bisektions-bandbreite
k-ary 1-array	$2+m$	✗	✗	2
k-ary 1-torus	$2+m$	✓	✓	4
k-ary n-mesh	$2n+m$	✗	✗	$2k^{n-1}$
2-ary n-mesh	$2n+m$	✓	✓	4^{n-1}
k-ary 2-mesh	$4+m$	✗	✗	$2k$
k-ary n-torus	$2n+m$	✓	✓	$4k^{n-1}$
Honeyc.-mesh	$3+m$	✗	✗	$2 \cdot 0.82\sqrt{N}$
Honeyc.-torus	$3+m$	✓	✓	$2 \cdot 2.04\sqrt{N}$
e-ary h-tree	$2e$	✗	✗	e^h
e-ary h-fly	e	✗	✓	$\frac{e^h}{2}$
k-ary h-flat	$(k-1)(h-1)+k$	✓	✓	$\frac{k^h}{2}$

Topologie	Hop-Anzahl	durchschnittl. Hop-Anzahl	Konnektivität	Netzwerk-verbindungen
k-ary 1-array	k	$\frac{2}{3} \cdot k$	1	$2(k-1)$
k-ary 1-torus	$\frac{k}{2}+1$	$\frac{k}{3}$	2	$2k$
k-ary n-mesh	$n(k-1)+1$	$\frac{nk}{3} \mid n(\frac{k}{3}-\frac{1}{3k})$	n	$2n(k-1)k^{n-1}$
2-ary n-mesh	$n+1$	$\frac{2n}{3}$	n	$2n \cdot 2^{n-1}$
k-ary 2-mesh	$2(k-1)+1$	$\frac{2k}{3} \mid \frac{2k}{3}-\frac{2}{3k}$	2	$4(k-1)k$
k-ary n-torus	$n(\frac{k}{2})+1$	$\frac{nk}{4} \mid n(\frac{k}{4}-\frac{1}{4k})$	$2n$	$2n \cdot k^n$
Honeyc.-mesh	$4\sqrt{\frac{N}{6}}$	N/A	2	$2(9t^2-3t)$
Honeyc.-torus	$2\sqrt{\frac{N}{6}}+1$	$0.54\sqrt{N}$	3	$2(9t^2)$
e-ary h-tree	$2h-1$	$\log_2(N)$	e	$2e^h \cdot (h-1)$
e-ary h-fly	h	h	1	$e^h \cdot (h-1)$
k-ary h-flat	h	N/A	$(k-1)(h-1)$	$(k-1)(h-1)k^{h-1}$

Tabelle 2.1: Die Eigenschaften der Topologien im Überblick [219]

anhand der Konnektivität bemessen. Eine Abschätzung der zu benötigten Fläche ergibt sich aus der Gesamtanzahl der Netzwerkverbindungen, da sie Informationen sowohl über die Anzahl der Verbindungen als auch über die dazu benötigten Ports liefert. Die Worst-Case- und durchschnittliche Hop-Anzahl sind gute Indikatoren für die Latenz im Netzwerk, denn sie geben den längsten beziehungsweise durchschnittlichen Pfad an. Als fünfte Anforderung ist die Belastungskapazität zu nennen, da alle Netzwerke unterschiedlich auf ein hohes Verkehrsaufkommen der Daten reagieren. Die Belastungskapazität des Netzwerks lässt sich gut durch die Bisektionsbandbreite wiedergeben. Alle anderen wichtigen Eigenschaften Kantengrad, Symmetrie und Homogenität geben

Auskunft über die Einfachheit des Routing-Verfahrens. Die einzige Anforderung, die nicht oder nur sehr ungenau aus den Eigenschaften extrahiert werden kann, ist die Leistungsaufnahme, die sich durch die entsprechende Topologie ergibt. Diese hängt stark von der eingesetzten Technologie und der technischen Umsetzung ab. Als erste Indikatoren können jedoch die gleichen Eigenschaften herangezogen werden, die auch die Fläche beeinflussen.

Neben den zuvor genannten Eigenschaften spielt es bei einer Topologie ebenfalls eine Rolle, wie sich diese praktisch auf einem Chip umsetzen lässt. Hier spielt neben den Leitungslängen zwischen Routern auch die Anordnung eine Rolle. Eine rechteckige und regelmäßige Anordnungen bietet bei der Skalierbarkeit des MPSoCs aufgrund ihrer verbreiteteren Verwendung in der Chipentwicklung eine bessere Werkzeug-Unterstützung und damit Vorteile gegenüber anderen Topologien. Bei dreidimensionalen Topologien sind außerdem zusätzliche Techniken wie das 3D-Die-Stacking notwendig [131][153]. Dies sind sehr aufwendige und neue Prozesse, die im Rahmen dieser Arbeit nicht zur Verfügung stehen und daher auch nicht näher betrachtet werden. Detaillierte Informationen über die Topologien, welche im Rahmen des CoreVA-MPSoCs betrachtet werden, sind in Kapitel 5 zu finden.

2.1.2 Routing-Verfahren

Das Routing-Verfahren bestimmt die Wegwahl der einzelnen Pakete innerhalb des Netzwerks. Es legt anhand eines Algorithmus den Pfad eines Pakets zwischen Quell- und Zielknoten innerhalb des Netzwerkes fest. Das Routing-Verfahren bestimmt entscheidend über die Effizienz der Topologie und umgekehrt hat jede Topologie auch andere Anforderungen an das Routing-Verfahren.

Grundsätzlich wird bei Routing-Verfahren zwischen *statischen* und *adaptiven* Verfahren unterschieden [40][139, Kapitel 12]. Adaptive Routing-Verfahren sind in der Lage, auf Änderungen innerhalb des Netzwerks zu reagieren. So kann beispielsweise der Ausfall von Knoten oder sehr hohes Verkehrsaufkommen auf bestimmten Kommunikationswegen berücksichtigt werden, indem die Route der Pakete dynamisch angepasst wird. Statische Routing-Verfahren berücksichtigen die aktuellen Verkehrsverhältnisse des Netzwerks nicht, sind jedoch einfacher und mit weniger Hardwareaufwand zu realisieren, als adaptive Routing-Verfahren.

Zu beachten bei der Auswahl von Routingalgorithmen ist das Auftreten von fehlerhaften Situationen, den *Deadlocks*, *Livelocks* und *Starvations*. Beim *Deadlock* blockieren sich zwei Pakete gegenseitig im Router aufgrund der Belegung von Endknoten. Ein *Livelock* bezeichnet die Situation, wenn ein Paket unendlich im Netz zirkuliert, ohne den Zielknoten jemals zu erreichen. Während eines Livelocks bleibt zwar die Funktionalität erhalten, jedoch wird die Leistungsfähigkeit des Netzwerks beeinträchtigt. Gibt es unterschiedlich priorisierte Pakete, kann es zu einer *Starvation* kommen. Muss das Netzwerk dauerhaft Pakete hoher Priorität transportieren, können Pakete geringer Priorität nicht weitergeleitet werden und „verhungern“. Je nach Routing-Verfahren

können die zuvor genannten Probleme eine Rolle spielen und es müssen geeignete Gegenmaßnahmen wie Erweiterungen oder Einschränkungen am NoC vorgenommen werden. Diese können am Algorithmus oder auf anderen Protokollebenen erfolgen.

Insgesamt stehen eine Vielzahl von Routing-Verfahren zur Verfügung [126][51][139, Kapitel 12], auf die in dieser Arbeit nicht im Einzelnen eingegangen werden kann. Als adaptives Routing-Verfahren ist beispielsweise das *Minimal-Adaptive-Routing* zu nennen. Bei diesem Verfahren wird zunächst immer der kürzeste Pfad zum Zielknoten gewählt [139, Kapitel 12]. Gibt es innerhalb des kürzesten Pfades alternative Routen, wird immer die mit der geringsten Auslastung ausgewählt. Im Gegensatz dazu steht das voll adaptive Routing, bei dem immer der Pfad mit der geringsten Auslastung gewählt wird, auch wenn damit ein Umweg in Kauf genommen werden muss. Die adaptiven Routing-Verfahren gelten als anspruchsvoller und beanspruchen zumeist mehr Fläche und Energie für die Hardware-Logik als die statischen Verfahren. Dafür reagieren diese aktiv auf bestimmte Situationen im Netzwerk, wie defekte Verbindungen oder Knoten mit hoher Verkehrsbelastung [40].

Beim statischen Routing wird zwischen deterministischen und stochastischen Verfahren unterschieden. Bei deterministischen Verfahren ist die Wegwahl im Vorfeld immer fest definiert und Pakete folgen zwischen den gleichen Ziel- und Quellknoten immer entlang des gleichen Pfades. Stochastische Routing-Verfahren entscheiden die Wahl des Pfades nach dem Zufallsprinzip. Dies kann zu einer gleichmäßigeren Auslastung des NoCs führen, jedoch können im Vorfeld keine Abschätzungen über Latenz und Durchsatz von Paketen getätigt werden. Zu den verbreitetsten stochastischen Routing-Verfahren gehören die so genannten Flooding-Algorithmen. Dabei wird ein Paket zu allen Verbindungen weitergeleitet (vervielfältigt), wodurch eine „Überflutung“ des Netzwerks stattfindet. Sobald ein Paket den Zielknoten erreicht, werden alle weiteren Kopien beim Eintreffen am Zielknoten gelöscht. Der Nachteil ist, dass dieser Algorithmus durch die Kopien von Paketen das Netzwerk generell mehr auslastet und damit auch den Energiebedarf erhöht. Außerdem ist für diese Algorithmen eine Begrenzung der Lebenszeit von Paketen nötig, da es sonst zu einem Livelock kommen kann.

Das bekannteste statische Routing-Verfahren ist das *Shortest-Path-Routing*, hier wird mit Hilfe einer Routingtabelle der kürzeste Pfad vom Quellknoten bis zum Zielknoten bestimmt. Routingtabellen haben jedoch den Nachteil, dass bei steigender Knotenanzahl die Anzahl von Einträgen in der Tabelle steigt. Dies erfordert größere Speicher, welche wiederum einen höheren Flächen- und Energiebedarf haben. Die Skalierbarkeit ist bei diesem Verfahren daher nur mit Einschränkung gegeben. Das einfachste und ressourceneffizienteste Verfahren ist das *XY-Routing*. Es eignet sich insbesondere für homogene 2D-Mesh Topologien und stellt dort eine Unterklasse des Shortest-Path-Routings dar. Beim XY-Routing wird ein Paket zunächst horizontal (X-Richtung) und anschließend vertikal (Y-Richtung) bis zum Zielknoten weitergeleitet [29]. Neben dem geringen Hardwareaufwand hat dieses Routing-Verfahren den Vorteil, dass keine Deadlocks oder Livelocks von Paketen auftreten können [42]. Ein Nachteil ist jedoch, dass sich bei diesem Routing-Verfahren eine hohe Last in der Mitte des Netzwerks einstellt. Eine

Erweiterung des klassischen XY-Routings bietet das *Surrounding-XY-Routing*. Hierbei können Pakete bei einer Blockierung auf dem eigentlichen XY-Pfad umgeleitet werden. Ist beispielsweise die vertikale Richtung blockiert, wird zunächst in die horizontale Richtung weitergeleitet und umgekehrt [40].

2.1.3 Flusskontrolle

Als Flusskontrolle (engl. *Flow Control*) wird der Mechanismus bezeichnet, wie Datenpakete das NoC durchlaufen und Netzwerkressourcen genutzt werden [139, Kapitel 12][40]. Dabei ist das erfolgreiche Übermitteln eines Datenstroms von einem Quellknoten zu einem Zielknoten die Hauptaufgabe der Flusskontrolle. Zusätzlich werden bei einer guten Flusskontrolle Netzwerkressourcen, wie die Link-Bandbreite und Speicherkapazitäten, effizient ausgenutzt, sodass im NoC nahezu der maximal mögliche Durchsatz bei möglichst geringer Latenz von Paketen erreicht werden kann [40].

Switching-Vefahren

Auf globaler Ebene im NoC entscheidet das Switching-Verfahren über den Transport von Paketen innerhalb des Netzwerks. Als Switching-Verfahren wird die technische Realisierung bezeichnet, wie Datenpakete von einem Quellknoten zum Zielknoten gelangen. Die im On-Chip-Bereich am verbreitetsten Switching-Verfahren sind das *Circuit*- sowie das *Packet-Switching*, wobei es zu letzterem wiederum weitere Unterteilungen gibt [4][139, Kapitel 12].

Beim Circuit-Switching wird die gesamte physikalische Verbindung während der Übertragung so lange gehalten, bis ein Datentransfer abgeschlossen ist. Dieses Switching-Verfahren kann als pufferloses, d.h. ohne Elemente zur Zwischenspeicherung von Daten, umgesetzt werden. Erforderlich hierzu ist allerdings zunächst ein Verbindungsaufbau, in dem geprüft wird, ob der entsprechende Pfad im Netzwerk für diesen Datentransfer frei ist. Dies erfordert in jedem Fall zusätzlichen Kontrollaufwand, der entweder in Software oder zusätzlicher Hardware umgesetzt werden muss. Des Weiteren kann es bei der Übertragung großer Datensätze zu einer langen Blockierung eines Pfades kommen, sodass dies zu Wartezeiten bei anderen Nutzern dieses Pfades kommen kann. Der Vorteil des Circuit-Switching ist jedoch, dass die optimale Übertragungszeit für den Datensatz garantiert werden kann, sobald eine Verbindung aufgebaut und reserviert ist. Diese Garantie wird auch als Guaranteed-Service (GS) bezeichnet, sodass Circuit-Switching in vielen MPSoCs mit Echtzeitanforderungen eingesetzt wird. In [102] konnte gezeigt werden, dass Circuit-Switching im Vergleich zu Packet-Switching mit geringeren Hardwareressourcen und höherer maximaler Taktfrequenz umgesetzt werden kann. Dabei ist die Latenz insbesondere für kleinere NoCs beim Versand von großen Paketen geringer als beim Packet-Switching. Mit wachsenden NoCs muss jedoch die Paketgröße linear mit ansteigen, um weiterhin effizienter als Packet-Switching zu sein.

Beim Packet-Switching wird ein Datensatz als Paket versendet. Durch das Zwischenspeichern von Paketen oder Paketteilen innerhalb von NoC-Knoten (Puffern), können sich so auf einem Verbindungsweg gleichzeitig mehrere Pakete befinden [4][19]. Eine physikalische Verbindung kann somit gleichzeitig durch mehrere Datensätze verwendet werden. Dies erhöht zwar ggf. die Latenz von einzelnen Paketen, jedoch können NoC-Links besser ausgelastet werden, sodass sich im Durchschnitt für alle Pakete ein höherer Datendurchsatz und eine geringere Latenz einstellen. Diese Art von Nachrichtenverkehr wird auch als Best-Effort (BE) bezeichnet. Eine Unterklasse des Packet-Switching ist das *Wormhole-Switching*, bei dem jedes Paket in kleinere Flusssegmente aufgeteilt wird, die auch als Flits (Flow Control Digit) bezeichnet werden. Jedes Flit besteht aus einem kurzen Kopf mit Routing-Informationen und einer Flusskontrolle sowie einem Rumpf mit Nutzdaten. Diese Technik zeichnet sich durch einen geringen Speicherbedarf pro Knoten aus, da nur wenige Flits und nicht ganze Pakete in den Routern zwischengespeichert werden müssen [4].

Als Alternativen zum Wormhole-Switching sind *Store-And-Forward (S&F)*- und *Virtual-Cut-Trough (VCT)*-Switching zu nennen [4][19]. Beim S&F-Switching wird ein Paket nur verschickt, wenn genügend Speicherplatz im nachfolgenden Router zur Verfügung steht, da ein ganzes Paket zwischengespeichert werden muss. Dies hat den Vorteil, dass Pakete nicht in Flits aufgeteilt werden müssen. Hierdurch ist nur noch ein Header pro Paket und nicht pro Flit nötig. Dieser Vorteil wird jedoch dadurch relativiert, dass zur Zwischenspeicherung ganzer Pakete deutlich mehr Speicher in jedem Router benötigt wird und häufig eine höhere Latenz zu verzeichnen ist als z.B. beim Wormhole-Switching. Das VCT-Switching funktioniert ähnlich wie das Wormhole-Switching mit dem Unterschied, dass das erste Flit erst weitergeleitet wird, wenn die Zwischenspeicherung des gesamten Paketes im Ziel-Router garantiert werden kann. Daraus resultieren zwar ähnlich geringe Latenzen wie beim Wormhole-Switching, jedoch ist auch hier der Speicherbedarf pro Knoten deutlich größer.

Lokale Flusskontrolle

Die Verwendung von Puffern zur Zwischenspeicherung von Paketen bzw. Flits erfordert eine weitere lokale Flusskontrolle zwischen zwei benachbarten Knoten. Vor der Übergabe eines Pakets/Flits an den nächsten Knoten muss sichergestellt sein, dass dieses dort zwischengespeichert werden kann. Auch hierzu stehen verschiedene Methoden zu Verfügung, die ebenfalls eine Auswirkung auf Performanz und die benötigten Hardwareressourcen haben. Unterschieden werden die drei Methoden *Credit-Based-Flow-Control*, *ON/OFF-Flow-Control* und *ACK/NACK-Flow-Control* [40][139, Kapitel 12]. Beim *Credit-Based-Flow-Control* hält der sendende Knoten den Zählerstand über die Anzahl freier Speicherplätze im empfangenden Knoten. Solange dieser Zählerstand größer als Null ist, können Flits gesendet werden. Erreicht der Zählerstand jedoch Null, können keine weiteren Flits weitergeleitet werden und müssen weiterhin im sendenden Knoten gehalten werden. Hat der empfangende Knoten ein Flit weitergeleitet, wird dem sendenden

Knoten der dadurch frei gewordene Speicherplatz in Form eines *Credits* signalisiert. Die Zeit auf Seite des Sendeknotens, die sich durch Signallaufzeiten oder Speicherelemente vom Sendezeitpunkt des Flits bis zum Erhalt des *Credit*-Signals ergibt, wird als *Round-Trip-Delay* bezeichnet. Um den maximalen Durchsatz aufgrund des *Round-Trip-Delays* nicht zu limitieren, müssen im Empfangsknoten genügend Speicherplätze vorhanden sein. Dies ermöglicht es, das *Round-Trip-Delay* zu verstecken. Ein Nachteil des *Credit-Based-Flow-Control* ist die Eins-zu-eins-Korrespondenz zwischen Flits und *Credit*-Signalen. Dies führt unter Umständen zu einem erhöhten Signalaufwand vom empfangenden zum sendenden Knoten. Genau dieser Signalaufwand wird beim *ON/OFF-Flow-Control* reduziert, indem nur eine einzige Signalleitung angibt, ob gesendet werden darf (ON) oder nicht (OFF). Um den maximalen Durchsatz nicht zu limitieren, müssen auch bei dieser Methode genügend Speicherplätze im empfangenden Knoten zur Verfügung stehen, um das *Round-Trip-Delay* bis Erhalt des ON/OFF-Signals auszugleichen. Der Empfänger muss das OFF-Signal bereits setzen, wenn nur noch so viele freie Speicherplätze zu Verfügung stehen wie Flits unterwegs sind, bis das Off-Signal im Sender eintrifft.

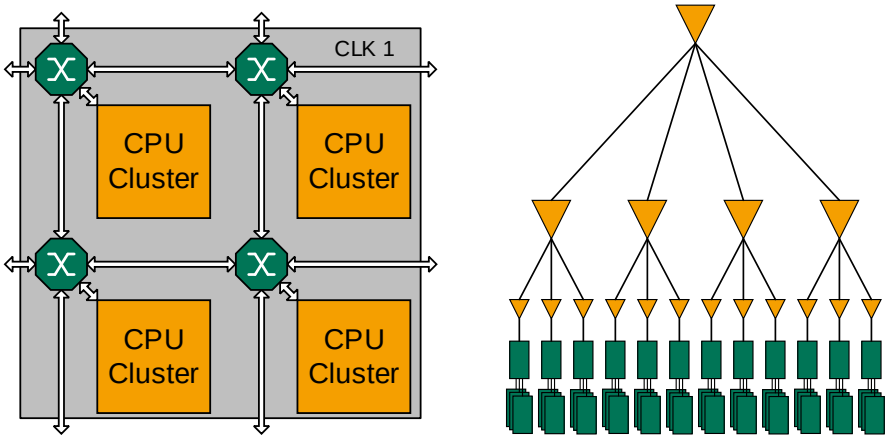
Die Methode *ACK/NACK-Flow-Control* bietet einen Ende-zu-Ende-Quittierungsmechanismus (Handshake-Verfahren), bei dem der empfangende Knoten dem sendenden Knoten mitteilt, ob ein Flit angenommen werden konnte (ACK) oder nicht (NACK). Dies halbiert die durchschnittliche Zeit für das *Round-Trip-Delay* im Vergleich zu den anderen beiden Methoden. Der Nachteil ist jedoch, dass Flits zusätzlich für die gesamte Zeit des *Round-Trip-Delay* im Sendeknoten gehalten werden müssen, da erst dann entschieden werden kann, ob das Flit tatsächlich vom Empfangsknoten angenommen werden konnte. Außerdem werden so auch im Fall einer Stauung ständig Flits übertragen, obwohl diese nie angenommen werden. Dies führt zu einem erhöhten Energiebedarf [40]. Einen Vorteil bietet der *ACK/NACK-Flow-Control* jedoch bei der Verwendung von asynchronen Schaltungen. Asynchrone Schaltungen verwenden ohnehin ein Handshake-Verfahren, um einen Datentransfer zu koordinieren, sodass diese Methode damit auch direkt als Flusskontrolle für das NoC eingesetzt werden kann (vgl. Abschnitt 2.1.4).

Virtuelle Kanäle

Eine Erweiterung für das Puffern von Flits bietet die Methode der virtuellen Kanäle (engl. *Virtual-Channels*) [139, Kapitel 12]. Dazu werden mehrere Puffer für einen physikalischen Link benötigt, sodass dieser Link weiterhin genutzt werden kann, auch wenn sich dort Flits eines anderen Pakets stauen. So werden Flits unterschiedlicher Ziele oder Priorisierung in parallel zur Verfügung stehende Puffer eingereiht, sodass diese andere Flits überholen können. Dies steigert ggf. den gesamten Netzwerkdurchsatz oder kann zur Latenzminimierung einzelner Paketklassen herangezogen werden. Der Nachteil von virtuellen Kanälen ist jedoch ein zum Teil deutlich erhöhter Ressourcenbedarf aufgrund der zusätzlich benötigten Speicherkapazitäten [40].

2.1.4 Global Asynchron Lokal Synchron (GALS)

Bei einer stetig wachsenden Anzahl von Prozessoren auf einem Chip wird es zunehmend schwieriger, diesen als synchrone Schaltung zu implementieren. Synchrone Schaltungen sind zumeist mit Hilfe von Taktflanken gesteuerten Flip-Flops realisiert. Beruht der Aufbau des Systems auf einem überall gleichzeitig gültigen Takt (siehe Abbildung 2.3a), muss sichergestellt werden, dass die Taktflanke an allen Flip-Flops des Systems zur richtigen Zeit ohne Störungen eintrifft. Daraus ergibt sich ein großer und weitverzweigter Taktbaum (siehe Abbildung 2.3b) aus Takttreibern, die eine Phasenverschiebung des Takts über den Chip ausgleichen. Bei einer synchronen Schaltung steigt die Anzahl und Größe der benötigten Takttreiber mit der Anzahl von Schaltungsteilen. Dies führt zu einem Anstieg der Fläche und einem höheren Energieverbrauch. Des Weiteren wird in [143] gezeigt, dass sich bei der Skalierung von synchronen MPSoCs auch unter Verwendung von NoCs eine starke Reduzierung der maximalen Frequenz einstellt.



(a) NoC mit nur einer Taktdomäne. (b) Taktbaum mit einer Taktdomäne

Abbildung 2.3: MPSoC mit einer Taktdomäne. 2.3a stellt die Taktdomäne am MPSoC-Design dar und 2.3b den entsprechenden Taktbaum

Um die Skalierbarkeit von NoCs jedoch weiter effektiv zu nutzen, ist eine Implementierung als GALS-System von Vorteil [69]. Dies bedeutet, dass einzelne Module des Systems weiterhin synchron betrieben werden, die globale Kommunikation zwischen diesen Modulen jedoch asynchron ist. Die Entwicklung eines vollkommen asynchronen Systems ist nur schwer umsetzbar, da sowohl die Entwurfswerkzeuge als auch IP-Bibliotheken stark auf dem synchronen Paradigma basieren. Das GALS-Paradigma bietet daher einen guten Kompromiss zwischen Hardwareeffizienz und Entwicklungszeit.

Die Implementierung als GALS-System teilt den zuvor sehr großen Taktbaum in viele kleine autonome Taktbäume auf (siehe Abbildung 2.4). Dies verringert die Anzahl und Größe von benötigten Takttreibern, da sich eine Phasenverschiebung zwischen den einzelnen Taktbäumen einstellen darf. Ein weiterer Vorteil eines GALS-Systems ist die Verringerung von elektromagnetischen Störeinflüssen (Electromagnetic Interference (EMI)). Störeinflüsse bewirken ein Rauschen zwischen der Versorgungsspannung und Nullpotential, sodass sich bei kleinen Strukturgrößen und geringen Versorgungsspannungen kritische Schaltungsverzögerungen einstellen können [30][155]. Ein synchroner Taktbaum hat einen besonders großen Anteil an der Entstehung von EMI, da sich durch das synchrone Schalten aller Register auf dem Chip während eines kurzen Zeitraumes eine sehr hohe Stromaufnahme einstellt [194]. Durch die Implementierung eines GALS-Systems ist es möglich, eine Phasenverschiebung zwischen den verschiedenen synchronen Modulen einzustellen, sodass der Schaltzeitpunkt der Register über eine gesamte Taktperiode verteilt werden kann. Dies hat zur Folge, dass sich eine gleichmäßige Stromaufnahme einstellt, die eine Reduktion von EMI auf dem Chip selbst bedeutet [171].

Weiterhin wird in [171] eine Möglichkeit zur Reduzierung der Verlustleistung aufgezeigt. Durch die Implementierung eines NoC als GALS-System ist es möglich, einzelne synchrone Module dynamisch an die aktuelle Performanzanforderung anzupassen. So können Knoten innerhalb des NoCs mit einer geringeren Taktfrequenz versorgt werden, die während einer bestimmten Anwendung eine geringere Performanz als andere Knoten liefern müssen. Außerdem kann durch die Reduzierung der Taktfrequenz ggf. auch die Versorgungsspannung angepasst werden, welches auch als DVFS bezeichnet wird [101]. Dabei wird der Chip in mehrere Blöcke mit verschiedener Versorgungsspannung partitioniert, diese Blöcke werden auch als Spannungsinseln (engl. *Voltage Islands*) bezeichnet [130]. Sowohl die Adaption der Taktfrequenz als auch der Versorgungsspannung haben laut [35] eine starke Reduzierung der Verlustleistung zur Folge.

Da in einem GALS-System die verschiedenen lokal synchronen Module asynchron zueinander sind, ist zwischen den verschiedenen Modulen eine Synchronisation erforderlich. Diese Synchronisation ist notwendig, um metastabile[1] Zustände in den Registern eines Moduls zu vermeiden. Kommt ein Signal am Eingangsregister eines Empfangs-Moduls an, kann nicht garantiert werden, dass die Setup- bzw. Hold-Zeiten des Registers eingehalten werden. Dies kann über einen unbestimmten Zeitraum zu einem metastabilen Zustand führen [12].

Bei der Implementierung eines MPSoC als GALS-System lassen sich zwei verschiedene Szenarien unterscheiden. Das erste Szenario ist das gelöst-synchrone System (*loosely synchronous*), bei dem zur Entwurfszeit eine bestimmte Abhängigkeit zwischen den verschieden Takten bekannt ist. Dieses Szenario lässt sich in den *mesochronen*, *plesio-*

[1] Flipflops können für eine gewisse Zeit in undefinierten Zwischenzuständen zwischen den stabilen Zuständen (Spannungsniveaus für logisch-0 und logisch-1) sein.

chronen und *heterochronen* Fall unterteilen. Das zweite Szenario ist ein asynchrones System, bei dem die Module von völlig verschiedenen Takten versorgt werden [175].

Verbesserungen und der Einsatz neuer Methoden in modernen Entwurfswerkzeugen bieten jedoch auch synchronen Architekturen neue Möglichkeiten der Skalierung. Moderne Entwurfswerkzeuge (siehe Kapitel 3.3) bieten den sogenannten Clock Concurrent Optimization (CCOpt)-Entwurfsablauf, in dem Taktbaum und kombinatorische Logik gleichzeitig optimiert werden [39]. Dies ermöglicht es zum Teil sogar die Phasenverschiebung des Takts positiv auszunutzen (useful skew), sodass höhere Taktfrequenzen erreicht werden können. Nichtsdestotrotz ermöglichen einige GALS-Methoden weiterhin eine höhere Ressourceneffizienz, da diese teils vollkommen unabhängig von einer Phasenverschiebung sind. Ein Vergleich des neuen CCOpt-Entwurfsablaufs mit verschiedenen GALS-Methoden wird in Kapitel 5.3.4 durchgeführt.

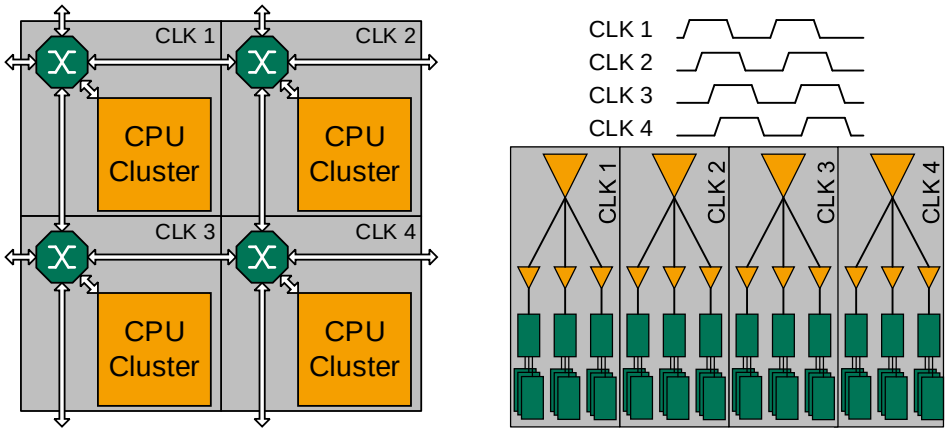
Mesochrone Systeme

Bei einem mesochronen GALS-System wird für jedes synchrone Modul eine eigene Taktdomäne geschaffen (siehe Abbildung 2.4a), sodass der Taktbaum in mehrere Teilbäume aufgeteilt werden kann. Die Bedingung für ein mesochrones System ist jedoch, dass alle Module weiterhin mit derselben Taktfrequenz versorgt werden. Zwischen den Taktsignalen der einzelnen Module darf jedoch eine unbekannte Phasenverschiebung existieren (siehe Abbildung 2.4b). Häufig werden in mesochronen Systemen alle Module von der gleichen Taktquelle versorgt. Da global jedoch keine Takttreiber eingesetzt werden, stellt sich aufgrund der unterschiedlichen Leitungslängen auf dem Chip eine Phasenverschiebung zwischen den einzelnen Modulen ein [175].

Aufgrund der möglichen Phasenverschiebung ist zur Vermeidung von metastabilen Zuständen eine Synchronisation zwischen den Modulen nötig. Hierzu können an den Modulübergängen neben bi-synchrones FIFOs auch spezielle Synchronisationsschaltungen eingesetzt werden, da die Taktfrequenz beim Sender- und Empfänger-Modul identisch ist (siehe Kapitel 5.3).

Plesiochrone Systeme

Bei plesiochronen GALS-Systemen arbeiten alle Module mit der gleichen Taktfrequenz, jedoch bestehen sehr geringe Abweichungen zwischen den Frequenzen. Dies hat zur Folge, dass sich im Laufe der Zeit langsam eine Phasenverschiebung zwischen den Modulen einstellt. Eine Möglichkeit dieses Phänomen zu vermeiden, ist es den instabilen Zeitpunkt zu detektieren und zu einem stabilen Zustand zurückzukehren. Dies ist jedoch nur möglich, falls eine Detektion realisierbar ist. Ein Vorteil dieser Methode ist, dass keine Synchronisation nötig ist, während sich das System im stabilen Zustand befindet. Somit ergeben sich keine zusätzlichen Latenzen beim Datentransfer.



(a) NoC bei mesonchroner Domänenaufteilung.

(b) Taktbäume beim mesonchronen Ansatz.

Abbildung 2.4: Mesochroner MPSoC: 2.4a zeigt Aufteilung innerhalb des NoC,2.4b die dazu gehörigen Taktbäume und mögliche Phasenverschiebung der Takte.

Heterochrone Systeme

Heterochrone Systeme kommen dem asynchronen Fall am nächsten, da es sich hier um vollständig verschiedene Taktfrequenzen handelt. Sie lassen sich noch einmal in ratiochrone und nonratiochrone Systeme unterteilen. Die nonratiochronen Systeme unterscheiden sich zum asynchronen Fall nur dadurch, dass sich die Frequenz nicht dynamisch ändert. Ratiochrone Systeme haben die Besonderheit, dass die verschiedenen Taktfrequenzen jeweils genau ein rationales Vielfaches voneinander sind. Auch bei heterochronen Systemen werden spezielle bi-synchrone FIFOs zur Synchronisation verwendet. Diese lassen sich, je nachdem ob es sich um ein ratio- oder nonratiochrone Systeme handelt, mit geringerem Aufwand implementieren.

Asynchrone Systeme

Ein asynchrones GALS-System zeichnet sich dadurch aus, dass die verschiedenen Module von vollständig unabhängigen Takten versorgt werden, die sich ggf. sogar ändern können. Dies bietet ein hohes Maß an Flexibilität, jedoch ist eine komplexe Synchronisation zwischen den verschiedenen Modulen nötig. Eine Lösung bieten auch hier bi-synchrone FIFOs an den Modulübergängen, welche neben der Latenz auch die Fläche und Verlustleistung des Systems erhöhen. Die Erhöhung der Verlustleistung kann jedoch ggf. durch eine dynamische Verringerung der Frequenz bzw. der Versorgungsspannung

einzelner Knoten wieder ausgeglichen oder sogar verbessert werden. Eine effizientere Methode zur Implementierung eines asynchronen GALS-Systems bieten vollständig asynchron arbeitende Router des NoCs.

Im Gegensatz zu einem synchronen System, wird bei einem vollständig asynchronen System nur auf den Zustand von bestimmten Signalen und nicht auf Taktflanken reagiert. Eine Herausforderung beim Empfangen asynchroner Signale ist es, sicherzustellen, dass das empfangene Signal erst dann ausgewertet wird, wenn es einen gültigen Zustand erreicht hat (Einschwingen). Dies erhöht den Entwicklungsaufwand der Router und erfordert ebenfalls zusätzliche Schaltungselemente beim Wechsel von einer synchronen zu einer asynchronen Domäne. Eine Entscheidung muss sauber und schnell geschehen, ohne auf ungewollte Schwankungen von Signalen auszulösen. Hierzu sind in der Literatur bereits einige Schaltungen beschrieben. In [111] und [132] werden beispielsweise verschiedene Logikelemente für asynchrone Schaltungen vorgestellt. Hervorzuheben sind dabei verschiedene Arbiter, C-Elemente, Handshake-Verfahren und Buffer. Zur Überprüfung von Fehlerzuständen in asynchron gestalteten Schaltungen müssen viele zusätzliche Anstrengungen unternommen werden. Die Veröffentlichungen [115] und [195] nehmen sich diesen Sachverhalt an und stellen Schaltungen eines Scan-Tests, eines Self-Test und das Testen durch externe Testschaltungen vor. Scan-Tests und Self-Test sind durch synchrone Schaltungen hinreichend bekannt. Die Autoren stellen zusätzliche Testschaltungen vor, mit denen asynchrone Schaltungen auf Fehler überprüft werden können.

2.2 NoC-Architekturen in der Forschung

In der Literatur gibt es eine Vielzahl von Arbeiten, die sich mit dem Thema NoC auseinander setzen. Viele dieser Arbeiten befassen sich mit Software-Simulationen von NoCs, Beispiele sind [41; 167]. Diese Arbeiten untersuchen zwar verschiedene Konzepte auf Performanz, können jedoch keine oder nur ungenaue Aussagen über ihre Hardwareressourcen machen. Für einen Vergleich, mit den in dieser Arbeit entstandenen NoC-Architekturen, eignen sich diese daher nur begrenzt. Im Folgenden werden daher NoC-Architekturen vorgestellt, für die bereits Untersuchungen in Fertigungstechnologien vorliegen. Diese zeichnen sich durch einen vollständigen Entwurfsablauf aus und betrachten auch die Hardware-Implementierungen der Komponenten.

2.2.1 Æthereal

Das Æthereal-NoC entstand ab 2000 im Rahmen eines Forschungsprojekts von „Philips Research Laboratories“ (später „NXP Semiconductors“) und wurde über viele Jahre stetig weiterentwickelt [56]. Der Fokus des Æthereal NoCs liegt auf der Audio- und Videoverarbeitung innerhalb eingebetteter Systeme aus der Konsumelektronik. Für Anwendungen mit Echtzeitanforderungen bietet das Æthereal-NoC eine Garantie für

die Latenz und den Durchsatz von Paketen, durch sogenannte GS-Verbindungen [5]. Ermöglicht wird dies durch die Verwendung von Circuit-Switching (siehe Abschnitt 2.1.3) und einem TDM[2]-Verfahren. Dazu befindet sich in jedem Router eine Tabelle, die eine Verschaltung von Ein- und Ausgängen für definierte Zeitschlitze vorgibt. Eine Zuweisung von Zeitschlitzen zu bestimmten Verbindungen muss zu Beginn einer Anwendung konfiguriert und kann als Optimierungsproblem behandelt werden [149]. Im frühen Entwicklungsstadium verfügte das Æthereal-NoC neben den GS-Verbindungen über zusätzliche BE-Verbindungen. Diese konnten zwar keine Schranken für Latenz und Durchsatz garantieren, lieferten im Durchschnitt jedoch bessere Ergebnisse. Der Ansatz dieser Kombination von GS und BE wurde bei der späteren Entwicklung des Æthereal-NoC allerdings verworfen, da die zusätzliche BE-Funktionalität zu kostenintensiv in Form von Fläche und Energie war [56]. Die eigentlich synchrone Verbindungsstruktur des Æthereal-NoCs kann durch die Verwendung spezieller Router aus [66] in ein mesochrones oder auch asynchrones NoC umgewandelt werden. In [67] wird das Æthereal-NoC als Verbindungsstruktur innerhalb eines gesamten MPSoCs vorgestellt, in dem neben einem Host-PC, Peripherie und gemeinsamen Speicher auch eine Vielzahl von VLIW-CPUs über das NoC verbunden sind.

2.2.2 Spidergon-STNoC

Das Spidergon-STNoC ist ein NoC, welches in Zusammenarbeit von STMicroelecronics und der Universität Pisa entstanden ist [36]. Es ist ein NoC, welches für heterogene SoCs mit vielen verschiedenen IPs entwickelt ist. Dazu unterstützt das Spidergon-STnoC eine Vielzahl verschiedener Topologien von einer Ring-Topologie über einfache Spannbäume bis hin zu irregulären Chordal-Ringen. Das NoC bietet zusätzlich spezielle Schaltungen, um ein mesochrones Gesamtsystem zu ermöglichen [158]. Als Switching-Methode wird im Spidergon-STNoC ein Wormhole-Switching verwendet, welches durch zwei virtuelle Kanäle erweitert ist. Dazu unterstützten Router und Netzwerk-Schnittstelle eine Paket-basierte Kommunikation. Das Routing innerhalb des NoCs ist deterministisch und im Paketheader kodiert. Damit verfügt die Paket-basierte Kommunikation über eine Ende-zu-Ende-Kontrolle und bietet damit eine gewisse Dienstgüte (GS). Zyklengenaue Garantien über Latenz und Durchsatz können jedoch nicht gegeben werden, da Verzögerungen aufgrund von Kollisionen in den Routern nicht verhindert bzw. berücksichtigt werden [157].

2.2.3 DSPIN/ASPIN

Das DSPIN-NoC (Distributed Scalable Predictable Interconnect Network) wurde bereits 2006 [117] entwickelt und physikalisch für die FAUST-Architektur in einer CMOS

[2] *Time Division Multiplexing*

130 nm Low-Power-Technologie von STMicroelecronics implementiert [116]. Der FAUST-Chip ist kein klassischer MPSoC, sondern verbindet mit Hilfe des NoC mehrere Hardwareeinheiten für die Basisbandverarbeitung des Mobilfunkstandards Long Term Evolution (LTE). Das DSPIN unterstützt eine 2D-Mesh Topologie mit statischem XY-Routing unter Verwendung von Wormhole-Switching. Zum Versenden von Paketen stehen auch hier mit GS und BE zwei Prioritäten für den Datentransfer zur Verfügung. Kommen GS-Pakete am Router des NoC an, wird der aktuelle BE-Datenverkehr unterbrochen und erst wieder durchgelassen, sobald alle Flits des GS-Paketes übertragen sind. Mit Hilfe von bi-synchronen FIFOs in den I/O-Ports der Router kann das DSPIN als GALS-Design verwendet werden, sodass sich einzelne Endknoten im NoC mit unterschiedlichen Taktfrequenzen und/oder Phasenverschiebungen zwischen den Taktsignalen betreiben lassen. Als Alternative wurde später mit dem ASPIN ein asynchrones NoC entwickelt, welches sich von der reinen Funktionalität, wie Topologie, Routing, etc., genauso wie das DSPIN verhält. Im ASPIN sind die Router jedoch komplett mit asynchronen Schaltungsteilen aufgebaut, lediglich am I/O-Port zum Endknoten befinden sich Asynchron⇔Synchron-Wandler. Ein Vergleich zwischen beiden Systemen ist in [160] aufgeführt, welcher auch in Abschnitt 5.3 für den Vergleich zwischen verschiedenen GALS-Designs herangezogen wird.

2.2.4 SpiNNaker

Die Universität Manchester hat zur Simulation neuronaler Netze den MPSoC SpiNNaker [136][137] entworfen, der 18 ARM968-Prozessoren enthält. Über ein asynchrones Netzwerk können bis zu 65536 dieser MPSoC-Chips verbunden werden. Dazu verfügt jedes MPSoC über einen zentralen Router, der über sechs bidirektionale -Ports mit anderen Chips verbunden ist. An diesen Router sind neben den -Ports auch die internen CPUs des MPSoCs angeschlossen. Dieses NoC dient insbesondere der Kommunikation zwischen Chips, bei der hauptsächlich neuronale Events Aktionen (Spikes) werden [191]. Dazu werden kurze Pakete von 40 bis 72 bit Größe seriell durch 4 bit breite Flits transportiert. Als Switching-Methode wird im Router des SpiNNaker ein S&F-Switching verwendet [124][123]. Zur Entlastung der CPUs beim Versenden von NoC-Paketen verfügt jede CPU über einen Kommunikations-Controller, der einer Netzwerk-Schnittstelle gleichzusetzen ist. Jeder MPSoC des SpiNNaker verfügt zur internen Kommunikation zusätzlich über ein System-NoC von Silistix, einer Ausgründung der Universität Manchester [52]. Über dieses NoC können von jeder CPU beispielsweise 32 kB interner Speicher, 128MB externer Speicher sowie eine Ethernet-Schnittstelle angesprochen werden.

2.2.5 Tomahawk2

Der Tomahawk2 [129] ist ein an der Technischen Universität Dresden entwickeltes heterogenes MPSoC für SDR[3]-Anwendungen. Spezialisiert ist dieser insbesondere für die Basisbandverarbeitung des Mobilfunkstandards LTE. Die 20 heterogenen Kerne des Tomahawk2-MPSoCs können über ein hierarchisches Paket-basiertes NoC miteinander kommunizieren. Zum Erzielen einer hohen Performanz über große Entfernungen zwischen den Kernen sind die Router des NoC teilweise über serielle Hochgeschwindigkeitslinks verbunden. Um Echtzeitanforderungen, die sich durch die Verarbeitung von LTE ergeben, zu unterstützen, verfügt der Tomahawk2 über einen zentralen *NoC-Manager* [189]. Der NoCMangager sucht und alloziert Verbindungen, auf denen ein bestimmter Durchsatz und eine bestimmte Latenz garantiert werden können. Diese GS-Verbindung wird anschließend dem Kommunikationskanal des entsprechenden Anwendungsteils (Task) zugewiesen. Nachteil dieses zentralen NoC-Managers ist, dass eine Verbindung zu jedem Router und jedem Kern erforderlich ist, welches die Skalierbarkeit des Systems begrenzt. Zusätzlich zu den GS-Verbindungen unterstützt das NoC des Tomahawk2 auch einen zweiten virtuellen Kanal für BE-Kommunikation, bei dem ein XY-Routing zum Einsatz kommt. Beim NoC handelt es sich um ein synchrones NoC, welches mit einer Taktfrequenz von 500 MHz betrieben wird. Zur Reduzierung der dynamischen Verlustleistung erlaubt der Tomahawk2 allerdings die dynamische Anpassung von Taktfrequenz und Versorgungsspannung der einzelnen Kerne.

2.2.6 Argo

Argo ist ein ab 2011 an der Dänischen Technischen Universität (DTU) entwickeltes NoC. Es entstand im Rahmen des Projekts „T-CREST - Time-predictable Multi-Core Architecture for Embedded Systems“ und ist Teil der T-CREST-Plattform [159]. Beim Argo-NoC handelt es sich um ein NoC, welches speziell für eingebettete MPSoCs mit harten Echtzeitanforderungen entwickelt ist [91]. Zur Unterstützung echtzeitkritischer Anwendungen bietet Argo Ende-zu-Ende-Verbindungen, die eine Garantie für Latenz und Durchsatz der zu übertragenen Daten geben. Ähnlich wie beim Æthereal NoC wird dies durch die Verwendung von Circuit-Switching und einem TDM-Verfahren ermöglicht. Das TDM-Verfahren wird durch eine spezielle Netzwerk-Schnittstelle (NI) bereitgestellt, indem verschiedenen Kommunikationskanälen Zeitschlitze zugeteilt werden. Die Verbindungsstruktur des Argo-NoC bilden asynchrone Router, die wiederum mit mesochronen NIs verbunden sind [90]. Argo ist daher ein GALS-System und bietet damit eine sehr gute Skalierbarkeit.

[3] Software Defined Radio

2.2.7 GigaNetIC

Am Fachgebiet Schaltungstechnik der Universität Paderborn wurde der hierarchische Multiprozessor GigaNetIC entwickelt [127][128]. Beim GigaNetIC sind mehrere CPU-Cluster über das GigaNoC-On-Chip-Netzwerk miteinander verbunden [145]. Ein CPU-Cluster besteht aus vier skalaren N-Core CPUs, die über einen Wishbone-Bus miteinander verbunden sind. Jede N-Core-CPU verfügt über einen lokalen Speicher, der für Instruktionen und Daten verwendet wird. Zudem besitzt jedes CPU-Cluster eine Netzwerk-Schnittstelle (hier Communication-Controller) zum GigaNoC. Das GigaNoC unterstützt ein Paket-basiertes Wormhole-Switching, für das jedem CPU-Cluster ein dedizierter Paketspeicher zur Verfügung steht. Eine Switch-Box des GigaNoC hat eine Latenz von drei Taktzyklen und unterstützt mehrere virtuelle Kanäle. Die Topologie des NoC ist zur Entwurfszeit konfigurierbar, standardmäßig wird hier ein 2D-Mesh in Kombination mit XY-Routing verwendet.

2.2.8 STHORM

Der STHORM [16][24][112] ist ein MPSoC, welches an der Universität Bologna und dem CEA in Zusammenarbeit mit STMicroelectronics entwickelt wurde. Erste Arbeiten zum STHORM sind ebenfalls unter den Begriffen „Platform 2012“ und P2012 zu finden, die zunächst als Namen für den MPSoC dienten. Der MPSoC STHORM hat eine hierarchische Architektur und besitzt mehrere CPU-Cluster, die jeweils bis zu 17 CPU-Kerne und eng gekoppelten gemeinsamen Speicher (siehe Abschnitt 7.2.2) enthalten. Die CPU-Kerne basieren auf dem STxP70-V4 und können aufgrund ihrer superskalare Architektur bis zu zwei 32-bit-Instruktionen pro Takt parallel ausführen. Jede CPU besitzt einen privaten Instruktionscache mit einer Größe von 16 kB. Alle CPUs innerhalb eines Clusters können auf einen eng gekoppelten gemeinsamen L1-Speicher mit einer Größe von 128 kB zugreifen. In jedem Cluster befindet sich zusätzlich eine CPU für Kontrollaufgaben, die über einen separaten 32 kB großen Datenspeicher verfügt. Mehrere dieser Cluster sind über ein asynchrones 2D-mesh NoC verbunden (ANoC) [180]. Die Router des ANoCs arbeiten vollkommen asynchron und verwenden ein Circuit-Switching als Switching-Methode. Das ANoC transferiert die Datenpakete Flit-basiert. Als Anschluss zu den lokal synchronen Clustern steht eine GALS-Schnittstelle zur Verfügung. Zudem enthält jedes Cluster einen zweikanaligen DMA-Controller der für Datentransfers über das NoC genutzt werden kann.

2.2.9 KiloCore

Die University of California hat in Zusammenarbeit mit IBM den in [20] vorgestellten KiloCore entwickelt. Der in einer 32 nm Partially-Depleted Silicon-on-Insulator (PD-SOI)-Technologie gefertigte Chip verfügt über 1000 16-bit RISC-CPUs, bei einer Gesamtfläche von etwa 62 mm^2. Die CPUs bestehen aus einer 7-stufigen Pipeline und

können mit einer maximalen Taktfrequenz von 1,78 GHz betrieben werden. Je nach eingestellter Taktfrequenz hat der Chip eine Leistungsaufnahme zwischen 13 W und 24 W. Sein Energieoptimum liegt bei einer Versorgungsspannung von etwa 0,9 V, wo bei einer Taktfrequenz von 1 GHz die Leistungsaufnahme 13,1 W beträgt. Der KiloCore verzichtet auf Caches, stattdessen verfügt jede CPU über einen sehr kleinen lokalen Instruktions- (128 bit × 40 bit) und Datenspeicher (zwei 128x16-bit-Bänke). Um die hohe Taktfrequenz zu erreichen, sind beide lokale Speicher mit Registern implementiert. Dies resultiert in einen vergleichsweise hohen Flächen- und Energiebedarf. Zusätzlich zu den lokalen Speichern sind insgesamt zwölf 64 kB Speicher über den gesamten Chip verteilt, auf die alle CPUs zugreifen können. Die On-Chip-Kommunikation wird durch zwei unabhängige 2D-Mesh-NoCs realisiert. Das erste NoC bietet Circuit-Switching und einen maximalen Durchsatz von 28,5 Gbps pro Link. Das zweite NoC unterstützt ein Paket-basiertes Wormhole-Switching, wobei jeder der fünf Ports ein 4 bit × 18 bit Puffer beinhaltet. Jeder dieser Puffer kann je nach Kommunikationsaufkommen zu einem großen Puffer oder zu vier virtuellen Kanäle verschaltet werden [184].

2.3 Kommerzielle NoC-Architekturen

Einige NoC-Architekturen finden bereits Verwendung in kommerziellen Produkten oder werden als kommerzielle IP-Kerne angeboten. Dieser Abschnitt stellt einige dieser kommerziellen NoC-Architekturen vor. Generell sind zu kommerziellen Produkten nur wenig detaillierte Architekturinformationen öffentlich verfügbar. Dennoch kann in Abschnitt 2.4 und Kapitel 6 ein Vergleich mit einigen Eigenschaften des CoreVA-MPSoC durchgeführt werden.

2.3.1 Adapteva's Epiphany

Der Epiphany von Adapteva ist ein kommerzieller Many-Core, bei dem mehrere RISC-CPUs über ein 2D-Mesh-NoC miteinander verbunden sind [3]. Pro Takt können eine Ganzzahl- und eine Fließkomma-Operation ausgeführt werden. Jede CPU verfügt über einen lokalen Scratchpad-Speicher. Die Speicherverwaltung wird vollständig in Software durchgeführt, wobei alle lokalen Scratchpad-Speicher zusammen einen verteilten und gemeinsamen Speicher (engl.: Disributed Shared Memory) bilden. Dazu wird für alle CPUs und Speicher ein global geteilter Adressraum verwendet, sodass jede Speicherstelle von jeder CPU direkt adressiert werden kann.

Als Verbindungsstruktur werden insgesamt drei unabhängige 2D-Mesh-NoCs verwendet. CPU-Cluster sind nicht vorhanden, da jeweils nur eine CPU pro Knoten an die NoCs angeschlossen ist. Ein NoC ist exklusiv für alle Lesezugriffe, die intern auf dem Chip stattfinden, zuständig, ein anderes für alle Schreibzugriffe die intern auf dem Chip stattfinden. Das dritte NoC behandelt alle externen -Zugriffe, die je nach

Ziel an einen der vier -Links geleitet werden. Diese -Links dienen als externe Schnittstelle und ermöglichen es, mehrere Chips zu einem größeren Multiprozessorsystem zusammenzuschalten. Der Epiphany kann in C, C++ sowie OpenCL programmiert werden.

Der 64-CPU-Multiprozessor Adapteva Epiphany E64G401 ist in einer 65-nm-Technologie gefertigt. Die CPUs des E64G401 sind 32-bit-RISC CPUs mit jeweils 32 kbit lokalen Scratchpad-Speicher. Bei einer maximalen Taktfrequenz von 800 MHz wird die Leistungsaufnahme auf unter 2 W angegeben [2].

Zum Zeitpunkt der Entstehung dieser Arbeit hat Adapteva bereits die nächste Generation des Epiphany angekündigt, der in einer 16-nm-FinFet-Technologie gefertigt werden soll [134]. Dieser soll aus 1024 64-bit-CPUs bestehen und abgesehen von der größeren Anzahl CPUs und des größeren Adressraums eine ähnliche Architektur besitzen wie der E64G401.

2.3.2 Kalray's MPPA

Ein weiterer kommerzieller Multiprozessor ist der Kalray MPPA-256 [45][46]. Dieser hat ähnlich wie der CoreVA-MPSoC eine hierarchische Architektur und besteht aus 16 Rechen- sowie vier I/O-Clustern, die über zwei verschiedene NoCs miteinander verbunden sind. Ein Rechencluster besteht aus 16 Rechen-CPUs, die vom Anwender genutzt werden können sowie einer zusätzlichen CPU für die Systemverwaltung. Die 32-bit-CPU-Kerne verfügen über eine VLIW-Architektur, die bis zu fünf verschiedene Instruktionen parallel ausführen kann. Außerdem besitzt eine CPU 8 kB lokale L1-Instruktions- und Datencaches, die auf 2 MB geteilten L2-Speicher eines Rechencluster zugreifen können. Jedes Cluster hat einen eigenen Adressraum und verfügt über einen DMA-Controller, der Daten innerhalb des geteilten L2-Speichers oder Daten von diesem zum NoC transferieren kann [45].

Die einzelnen Cluster kommunizieren über NoC-Pakete miteinander. Für diese Kommunikation wird dem Programmierer eine verteilte Laufzeitumgebung geboten, die auf dem *Inter Process Communication (IPC)*-Modell basiert [46]. Der Kalray MPPA verfügt über zwei seperate NoCs, die verschiedene Services und Typen von Netzwerkverkehr bereitstellen. Dabei ist ein NoC für den Transfer von Nutzdaten vorgesehen (D-NoC). Das D-NoC stellt hohe Übertragungsbandbreiten zur Verfügung und bietet GS-Verbindungen. Im Gegensatz dazu steht das C-NoC, welches ausschließlich für den Transfer von Kontrollinformationen gedacht ist. Pakete sind auf die Größe von 64 bit beschränkt, verfügen über keine Flusskontrolle und können als eine Art Mailbox-System angesehen werden [46]. Beide NoCs sind in einer 2D-Torus-Topologie abgebildet, verbinden die insgesamt 16 Rechen- und 16 I/O-Cluster und bestehen aus jeweils 32 Routern. Die Router sind über bidirektionale Links miteinander verbunden und haben an jeweils einem Port die Sende- und Empfangsschnittstelle zum Cluster. Der MPPA-256 ist in einer 28-nm-Technologie gefertigt und benötigt 16 W bei einer Taktfrequenz von 600 MHz sowie 24 W bei 800 MHz [89].

2.3.3 Intel Xeon Phi Knights Landing

Der Intel Xeon Phi Knights Landing [86] ist ein klassischer Vertreter eines Großrechners (Mainframe), der ebenfalls über eine hohe Anzahl CPUs verfügt, die über ein NoC kommunizieren können. Insgesamt sind beim Knights Landing 76 CPU-Kerne (x86) integriert, von denen jedoch vier standardmäßig abgeschaltet sind, um die Ausbeute zu erhöhen. Jeder CPU-Kern verfügt über 32 kB Instruktions-Cache und 32 kB Daten-Cache. Jeweils zwei dieser CPU-Kerne bilden einen CPU-Tile, in dem sie sich einen 1 MB L2-Cache teilen. Alle L2 Caches können über eine 2D-Mesh NoC-Verbindungsstruktur mit insgesamt 8 MCDRAM-Controllern kommunizieren, die auf bis zu 16 GB DRAM innerhalb des gleichen Chipgehäuses zugreifen können. Außerdem sind zwei DDR4-Controller und eine PCIe-Schnittstelle über das NoC ansprechbar. Das NoC verwendet ein XY-Routing, bei dem zunächst in vertikaler (Y) Richtung geroutet wird. Ein Link zwischen zwei Routern ist bidirektional und hat eine Breite von 32 B. Die Latenz beträgt zwei Taktzyklen in horizontaler Richtung und nur einen Taktzyklus in vertikaler Richtung pro Hop (Router).

In einer 14-nm-FinFET Technologie arbeiten die CPUs mit einer Taktfrequenz von 1,31 GHz und das NoC mit 1,05 GHz. Der Knights Landing erreicht dabei eine Performanz von 3 Tflops (double-precision) bzw. 6 Tflops (single-precision). Insgesamt wird für den gesamten Chip eine Verlustleistung von etwa 200 W angegeben.

Der Intel Xeon Phi Knights Landing ist in dieser Arbeit zum Vergleich aufgeführt. Vergleichbare Architekturen werden in dieser Arbeit nicht betrachtet, da allgemein die Leistungsaufnahme von Mainframe-Prozessoren mehrere hundert Watt beträgt und sie damit nicht für mobile, eingebettete Systeme geeignet sind.

2.3.4 NoC-IP-Kern Anbieter

Neben den bereits vorgestellten kommerziellen Chips gibt es auch Unternehmen, die sich nur auf den NoC-Interconnect spezialisiert haben und entsprechende IP-Kerne anbieten. Zu nennen sind hier die Unternehmen Netspeed, Arteris und Sonsics [63]. Arteris unterstützen mit ihrem FlexNoC Schnittstellen zu gängigen Bussystemen, wie z.B. ARM AMBA AXI, ACE und OCP [7]. Das NoC basiert auf Master-Slave-Transaktionen, die insbesondere für Cache-basierte Systeme ausgelegt sind. Mit Hilfe eines grafischen Architektur-Editors kann eine für benutzerspezifische SoCs geeignete Topologie konfiguriert werden [98]. Auch das Unternehmen Netspeed [125] bietet mit ihrem Orion-NoC ein NoC-Interconnect, der gängige Busstandards wie AXI unterstützt. Des Weiteren bieten die NoCs Gemini und Pegasus optimierte NoCs für Cache-basierte Systeme. Beide NoCs unterstützen dabei zusätzlich ein Cache-Kohärenz-Protokoll [63]. Ähnliche NoCs bietet auch Sonics, die ebenfalls NoC-IP-Kerne für AXI und Cache-basierte SoCs anbieten [166]. Bei allen drei NoC-IP-Kern Anbietern sind, ähnlich wie bei den kommerziellen MPSoCs, keine Implementierungs- und Architekturdetails öffentlich einsehbar.

2.4 Einordnung des CoreVA-MPSoCs in den Stand der Technik

In diesem Abschnitt wird das CoreVA-MPSoC in den Stand der Technik vergleichbarer MPSoC-Chips eingeordnet. Dazu werden die allgemeinen Systemeigenschaften des CoreVA-MPSoCs mit den Systemeigenschaften einiger MPSoCs aus den vorherigen Abschnitten verglichen. Herangezogen werden für den Vergleich in diesem Abschnitt lediglich MPSoCs, bei denen Ergebnisse für Chips in einer möglichst vergleichbaren Technologie (Strukturgrößen) vorliegen, bzw. überhaupt physikalische MPSoCs erstellt wurden. Ein detaillierter Vergleich der verschiedenen NoC-Komponenten auf Architekturebene findet anschließend in den weiteren Kapiteln der Arbeit statt. Dort werden zu den jeweilig relevanten Architektureigenschaften der NoCs und MPSoCs aus den Abschnitten 2.2 und 2.3 Bezug genommen.

Tabelle 2.2 enthält die allgemeinen Architektureigenschaften der MPSoCs. Die dargestellten Werte stammen aus den bei der Vorstellung des jeweiligen MPSoCs angegeben Quellen und sind teilweise hieraus berechnet bzw. abgeschätzt. Die Angabe zur Größe des Speichers beinhaltet die Summe aller On-Chip-Speicher, beispielsweise die Größe der L1-Daten- und Instruktionscaches sowie des L2-Speichers.

Das CoreVA-MPSoC ist als hierarchisches MPSoC mit sechzehn per NoC verbundenen CPU-Clustern und insgesamt 64 CPUs aufgeführt. Die Werte hierfür stammen aus Kapitel 8.1 sowie [200]. Insbesondere die Angaben für Chipfläche und Leistungsaufnahme sind als vorläufig zu betrachten, da zum Zeitpunkt der Entstehung dieser Arbeit die Input/Output (I/O)-Schnittstelle des CoreVA-MPSoCs noch nicht vollständig spezifiziert ist.

Zur besseren Einordnung der vorgestellten MPSoCs werden die ARM-basierten MPSoCs aus dem Exynos-5450 für den Vergleich herangezogen [61]. Der Exynos-5450 besitzt jeweils vier Cortex-A7-CPUs und vier Cortex-A15-CPUs, die sich aufgrund ihrer Fertigung in einer 28-nm-Technologie für einen Vergleich mit dem CoreVA-MPSoC eignen. Außerdem kommen ARM-basierte MPSoCs in vielen mobilen und eingebetteten Systemen zum Einsatz. Für den Vergleich wird das jeweilige CPU-Cluster isoliert betrachtet, sodass weitere Peripherie auf dem Exynos-5450 Chip nicht mit einfließt. Der Flächenbedarf wird inklusive L2-Cache angegeben (Cortex-A15 mit 2 MB, Cortex-A7 mit 512 kB). Die Leistungsaufnahme wird in [61] hingegen nur für die einzelnen CPUs (ohne L2-Cache) angegeben.

Aus dem Bereich der Großrechner ist als weiterer NoC-basierter MPSoC der Intel Xeon Phi Knights Landing [86] aufgeführt, dessen Leistungsaufnahme mit 200 W in einer anderen Dimension liegt.

[4] Abgeschätzt, basierend auf [129]
[5] Abgeschätzt, basierend auf [89]

Tabelle 2.2: Architektureigenschaften von eingebetteten Multiprozessoren

Name	CPU-Typ	CPUs pro Chip	CPUs pro Cluster	Verbindungs-struktur	On-Chip-Speicher [MB]	Technologie [nm]
SpiNNaker	Skalar	18	1	NoC	1,72	130
Tomahawk2	Hetero-gen	20	2 – 8	NoC	2,13	65
KiloCore	Skalar	1024	1	NoC	1,9	32
Kalray MPPA-256	5-fach VLIW	288	4 – 17	NoC + part. Crossbar	44,5	28
Adapteva Epiphany	2-fach	64	1	NoC	2	28
STM STHORM	2-fach	68	17	NoC + gem. L1-Speicher	2,24	28
CoreVA-MPSoC	2-fach VLIW	64	4	NoC + Crossbar + gem. L1-Speicher	3	28
Quad ARM Cortex-A7	5-fach	–	4	Crossbar	0,77	28
Quad ARM Cortex-A15	8-fach	–	4	Crossbar	2,3	28
Intel Xeon Phi Knights Landing	Super-skalar	76	2	NoC + gem. L2-Speicher	42,8	14

Tabelle 2.3: Ressourcenbedarf von eingebetteten Multiprozessoren

Name	F_{max} [MHz]	Chipfläche [mm^2]	Leistungsauf-nahme [W]	Rechenleistung [GOPS]	Rechenleistung [GFLOPS]	Energieeffizienz [GOPS pro W]	Flächeneffizienz [GOPS pro mm^2]
SpiNNaker	180	102	1	3,24	–	3,2	0,03
Tomahawk2	500	36	1,4[4]	105	3,8	75,5	2,9
KiloCore	1000	62	13,1	1000	–	76,3	16,1
Kalray MPPA-256	600	k.A.	16	864[5]	345,6[5]	54	k.A.
Adapteva Epiphany	800	10	2	102	51,2	51	10,2
STM STHORM	600	26	2	76	38	38	2,92
CoreVA-MPSoC	700	14,4	2	89,6	–	44,8	6,2
Quad ARM Cortex-A7	1200	3,8	> 0,85	14,4	9,6	< 16,9	3,8
Quad ARM Cortex-A15	1600	19	> 4,20	57,6	51,2	< 13,7	3,0
Intel Xeon Phi Knights Landing	1320	–	200	6000	3000	30	–

Der Vergleich in Tabelle 2.3 ist lediglich als Einordnung des CoreVA-MPSoCs zu sehen. Ein vollkommen fairer Vergleich der betrachteten MPSoCs ist nicht gegeben, da die MPSoCs unterschiedliche Eigenschaften und damit verschiedene Vor- und Nachteile aufweisen. So unterscheidet sich die Menge an On-Chip-Speicher (1,28 MB bis 44,5 MB), die jedoch in den Effizienzwerten nicht erfasst werden kann. Außerdem besitzen die MPSoCs eine unterschiedliche Anzahl und Konfiguration an I/O-Schnittstellen, sowie weiteren On-Chip Hardwareeinheiten. Zudem sind die MPSoCs in verschiedenen Technologien gefertigt bzw. für diese entworfen. Ein direkter Vergleich bezüglich Flächenbedarf, Leistungsaufnahme und maximaler Taktfrequenz ist daher nur schwer möglich. Zudem sind verschiedene Fertigungsprozesse mit gleicher Strukturgröße auf unterschiedliche Ziele hin optimiert. Beispiele sind hier *High-Performance*-Prozesse, die sich durch hohe Taktfrequenzen auszeichnen, sowie *Low-Power*-Prozesse, die besonders geringe Leckströme aufweisen.

3 Grundlagen und Werkzeugkette des CoreVA-MPSoC

In diesem Kapitel werden die Systemgrundlagen und die Werkzeugkette vorgestellt, die zur Entwicklung der in den folgenden Kapiteln vorgestellten Hard- und Softwarekomponenten eingesetzt werden. In Abschnitt 3.1 wird zunächst die Architektur der CoreVA-CPU beschrieben. Diese hat als Verarbeitungseinheit im CoreVA-MPSoC eine zentrale Rolle im weiteren Verlauf der Arbeit. Anschließend wird in Abschnitt 3.3 ein Hardware-Entwurfsablauf eingeführt, der im Rahmen dieser Arbeit verwendet wird, bzw. entstanden ist. Die Werkzeuge und Bibliotheken, die zur Software-Entwicklung des CoreVA-MPSoC entwickelt und eingesetzt werden, sind in Abschnitt 3.4 aufgeführt. Hervorzuheben ist, dass die hier vorgestellte Werkzeugkette nicht nur zur Entwicklung des CoreVA-MPSoC eingesetzt werden kann, sondern generell für den Entwurf von eingebetteten Multiprozessoren. Große Teile des Entwurfsablaufs und der eingesetzten Systemgrundlagen sind in Zusammenarbeit mit Gregor Sievers entstanden, sodass Teile dieses Kapitels ebenfalls in Kapitel 3 der Dissertation [162] beschrieben werden.

3.1 Der VLIW-Prozessor CoreVA

Die CoreVA-CPU [73][74][78][74] wird als Verarbeitungseinheit im CoreVA-MPSoC eingesetzt. Deren Architektur hat damit nicht nur einen Einfluss auf die in diesem Kapitel vorgestellte Werkzeugkette, sondern auch auf viele andere Systemkomponenten. Die CoreVA-CPU wurde im Rahmen des MxMobile-Projektes [80] an der Universität Paderborn entwickelt und liegt als synthetisierbare Register Transfer Level (RTL)-Beschreibung in der Hardwarebeschreibungssprache Very High Speed Integrated Circuit Hardware Description Language (VHDL) vor [165, S. 379 ff.]. Einsatzgebiete sind energiebeschränkte eingebettete Systeme. Die Architektur der CoreVA-CPU wird im folgenden Abschnitt vorgestellt. Im Anschluss daran wird auf die CPU-Makros eingegangen, die als Basisblöcke für den hier vorgestellten Hardware-Entwurfsablaufs für das CoreVA-MPSoC herangezogen werden.

Die CoreVA-CPU verfügt über eine konfigurierbare 32-bit-Architektur. Die nach dem VLIW-Prinzip [50] parallel ausführbaren 32-bit-Instruktionen werden auch als Slots bezeichnet. Beim der CoreVA-CPU ist die Anzahl dieser Slots zur Entwurfszeit konfigurierbar. Grundsätzlich kann durch mehr Slots die Leistungsfähigkeit bei gleicher Taktfrequenz der CPU gesteigert werden. Wie viele Operationen jedoch tatsächlich

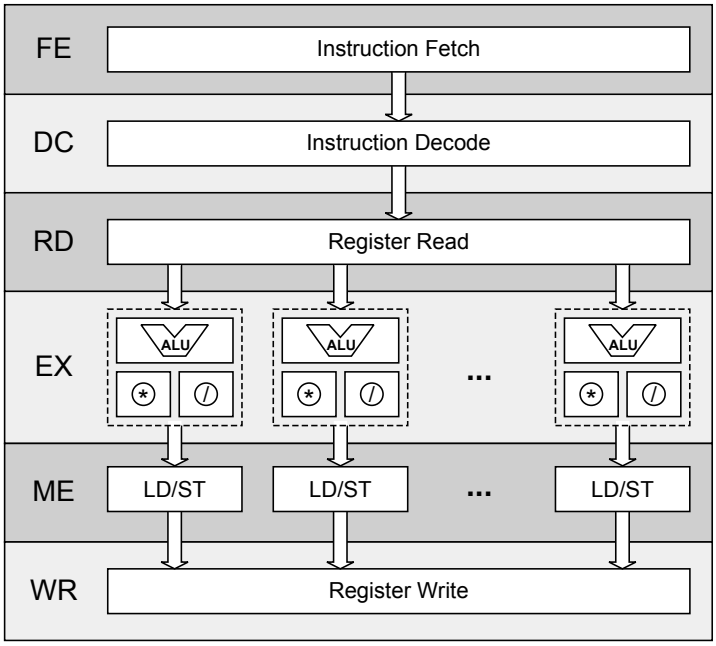


Abbildung 3.1: Blockschaltbild der Pipeline des CoreVA-Prozessors

parallel ausgeführt werden können, hängt von der Parallelität auf Instruktionsebene (ILP) des ausgeführten Softwareanwendung ab. Die Aufteilung des Programmstroms auf die einzelnen Ausführungseinheiten werden vom Compiler vorgenommen (siehe Abschnitt 3.4.1).

Die Registerbank (*Register-File*) der CoreVA-CPU besitzt 31 32-bit-Register. Zusätzlich verfügt die CoreVA-CPU über ein 8-bit-Kontrollregister (*Condition-Register*) für die bedingte Ausführung von Instruktionen.

Zur Steigerung der Taktrate besitzt der Prozessor eine sechsstufige Fließbandverarbeitung (*Pipelining*, siehe Abbildung 3.1). In der *Instruction-Fetch*-Stufe (FE) werden die Instruktionen aus dem Instruktionsspeicher geladen und in einem *Alignment*-Register gespeichert. Durch die Instruktionskompression kann es vorkommen, dass sich eine Instruktionsgruppe über zwei aufeinander folgenden Speicherstellen erstreckt. Da für die Dekodierung eine Instruktionsgruppe vollständig vorliegen muss, wird das Alignment-Register zur Ausrichtung verwendet. Die an den Instruktionsspeicher angelegte Adresse ergibt sich aus dem Befehlszähler (Program Counter (PC)), einem speziellen Register der CPU. Bei der kontinuierlichen Abarbeitung des Programmstroms wird der PC jeden Takt um die Größe der jeweils ausgeführten Instruktionsgruppe erhöht. Enthält eine Instruktionsgruppe einen Sprungbefehl, wird der PC auf das entsprechende Sprungziel

gesetzt. Die *Instruction-Decode*-Stufe (DC) dekodiert die einzelnen Befehle einer Instruktionsgruppe. Zudem besitzt die DC-Stufe eine einfache Sprungvorhersage. Rücksprünge werden spekulativ ausgeführt, Vorwärtssprünge spekulativ nicht ausgeführt. Wurde ein Sprung falsch vorhergesagt, muss er rückabgewickelt werden. Hierzu wird die Pipeline vollständig geleert (*Flush*). In der *Register-Read*-Stufe (RD) werden benötigte Operanden aus dem Register-File gelesen.

In den drei bisher genannten Pipelinestufen sind alle VLIW-Slots gleich aufgebaut. Heterogene VLIW-Slots können durch die Verwendung von verschiedenen Ausführungseinheiten in der *Execute*-Stufe (EX) realisiert werden. Jeder VLIW-Slot besitzt zwingend eine arithmetisch-logische Einheit (Arithmetic Logical Unit (ALU)). Die ALU kann beispielsweise Additionen, Subtraktionen oder Bit-Schiebeoperationen auf ganzen Zahlen ausführen. Optional kann ein Slot einen Multiplikations-Akkumulierer (Multiply Accumulate (MAC)), eine Dividiereinheit[1] (DIV) und eine Einheit für Zugriffe auf den Datenspeicher (Load/Store (LD/ST)) enthalten. Die MAC-Einheit ist aufgrund der hohen Komplexität auf die beiden Pipelinestufen EX und ME aufgeteilt. Durch diese Aufteilung liegt die MAC-Einheit nicht im kritischen Pfad der CPU, die Latenz von MAC-Operationen erhöht sich allerdings auf zwei Takte. Die für Speicherzugriffe benötigte Adresse wird in der EX-Stufe berechnet und an den Datenspeicher angelegt. Der Datenspeicher befindet sich parallel zu den EX-ME-Pipeline-Registern. Resultate von Leseoperationen werden in ein Register der ME-WR-Pipelinestufe gespeichert. Schreibzugriffe auf den Speicher können eine Größe von 8 bit, 16 bit und 32 bit aufweisen. Leseoperationen besitzen immer eine Datenbreite von 32 bit. Die Ergebnisse aller ausgeführten Operationen der Ausführungseinheiten werden in der *Register-Write*-Stufe (WR) in das Register-File geschrieben. Zusätzlich verfügt die CoreVA-CPU im SIMD-Modus über die Möglichkeit, pro VLIW-Slot zwei gleiche 16-bit-ALU- oder MAC-Operationen in einem Zyklus auszuführen. Für die bedingte Ausführung von SIMD-Instruktionen ist ein zweites 8-bit-Condition-Register integriert. Der Instruktionssatz der CoreVA-CPU ist zudem für die Verarbeitung von Signalverarbeitungsalgorithmen optimiert [211], kann aber auch durch Instruktionen zur verbesserten Berechnung von neuronalen Netzen erweitert werden [221].

Durch den Pipeline-*Bypass* werden Rechenergebnisse direkt aus der EX-, ME- und WR-Stufe an die RD-Stufe weitergeleitet. Rechenergebnisse stehen somit für weitere Berechnungen zur Verfügung, bevor sie im Register-File gespeichert worden sind. Eine detaillierte Analyse und Optimierung des Pipeline-Bypasses der CoreVA-CPU ist in [81] zu finden. Durch die Verwendung des Pipeline-Bypasses besitzen fast alle Operationen eine Latenz von einem Takt. Lediglich Lade- und MAC-Operationen weisen eine Latenz von zwei Takten auf. Da der Instruktionssatz der CoreVA-CPU bisher nicht vollständig durch spezifizierte Instruktionen belegt wird, ist es möglich, die CPU um anwendungsspezifische Instruktionen zu erweitern.

[1] Realisiert im Divisions-Schritt-Verfahren

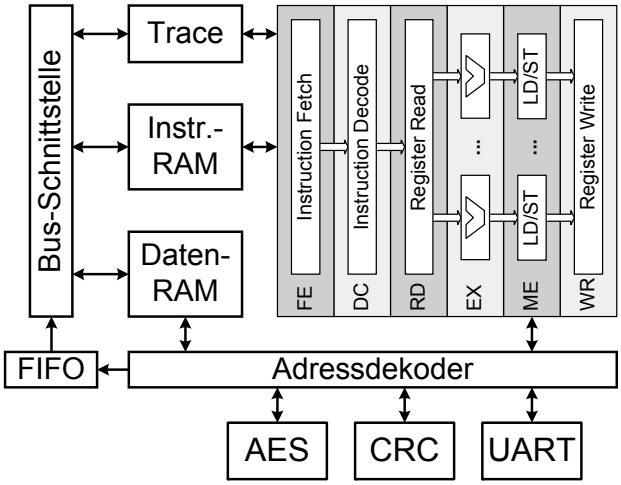


Abbildung 3.2: Blockschaltbild der CoreVA-CPU mit Speicher und Hardwareerweiterungen

Mittels eines Adressdekoders kann die CPU auf Hardwarebeschleuniger und externe Schnittstellen zugreifen (siehe Abbildung 3.2). Dies wird als Memory Mapped I/O (MMIO) bezeichnet, da anhand der Adresse die verschiedenen Komponenten ausgewählt werden. Bis zu 32 Erweiterungen können so an die CoreVA-CPU angebunden werden. Jeder Erweiterung steht ein Adressbereich von bis zu 1 MB Größe zur Verfügung, der in den Adressbereich der CoreVA-CPU eingeblendet wird. Für die CoreVA-CPU stehen unter anderem Beschleuniger für Verschlüsselung (Advanced Encryption Standard (AES) und Elliptic Curve Cryptography (ECC)) sowie Prüfsummenberechnung (Cyclic Redundancy Check (CRC)) zur Verfügung [144][208][205]. Die Integration einer Erweiterung beschleunigt den jeweiligen Algorithmus um bis zu mehreren Größenordnungen, bedingt jedoch einen teils deutlich höheren Ressourcenbedarf [78, S. 165 ff.]. Eine serielle Schnittstelle (Universal Asynchronous Receiver Transmitter (UART)) ermöglicht die Kommunikation der CPU mit der Außenwelt.

Der CoreVA-Prozessor besitzt getrennte Instruktions- und Datenspeicher (Harvard-Architektur). Die lokalen Daten- und Instruktionsspeicher besitzen jeweils eine maximale Größe von 1 MB. Alternativ können Daten- und Instruktionscaches integriert werden [163].

In [162] ist eine Anbindung der CPU an den CPU-Cluster entstanden. Diese in Abbildung 3.2 als First-In First-Out (FIFO) dargestellte Komponente, ermöglicht der CPU über eine *Master*-Schnittstelle den Zugriff auf andere Komponenten innerhalb eines Clusters, z. B. auf die lokalen Speicher anderer CPUs innerhalb eines Clusters oder auch die Netzwerk-Schnittstelle (NI, siehe Kapitel 6). Schreibzugriffe werden über

ein FIFO entkoppelt. Hierdurch muss die CPU bei hohem Kommunikationsaufkommen im Cluster nur angehalten werden, wenn das FIFO vollständig gefüllt ist. Der Speicher des FIFOs kann aus Registern oder Static Random Access Memory (SRAM) bestehen [220]. Ein Lesezugriff auf eine andere Komponente im CPU-Cluster hat eine Latenz von vier oder mehr Takten. Da ein lokaler Lesezugriff eine Latenz von zwei Takten aufweist, muss die CPU bei einem Lesezugriff im Cluster mindestens zwei Takte auf die angeforderten Daten warten. Die CoreVA-CPU unterstützt als VLIW-Architektur keine ausstehenden Speicherzugriffe (*Outstanding-Transactions*), daher muss die CPU in dieser Zeit angehalten werden. Aus diesem Grund wird im Rahmen dieser Arbeit ein Kommunikationsmodell eingesetzt, das (nahezu) vollständig auf Lesezugriffe im MPSoC verzichtet (siehe Abschnitt 3.4.3).

Eine *Slave*-Schnittstelle ermöglicht vom CPU-Cluster aus Zugriffe auf Instruktions- und Datenspeicher einer CPU. Zudem ermöglicht diese Schnittstelle das Starten und Stoppen der CPU sowie das Auslesen des CPU-Status. Eine dedizierte Hardwareeinheit erlaubt die zyklengenaue Speicherung des Zustandes der CPU in einem externen Speicher [78, S. 44 ff.; 79]. Diese *Trace*-Einheit lässt die CPU hierzu einen Taktzyklus ausführen und versendet anschließend den Zustand in mehreren Taktzyklen. Die genaue Anzahl der Taktzyklen hängt von der Konfiguration der CPU ab.

Basierend auf der CoreVA-CPU-Architektur sind in einer 65-nm-Technologie zwei Chip-Prototypen (CoreVA-VLIW und CoreVA-Ultra Low Power (ULP)) entstanden. Beide Prototypen integrieren jeweils 16 kB Daten- und Instruktionscache und besitzen eine Chip-Fläche von 2,64 mm^2. Der CoreVA-VLIW-ASIC [78, S. 198 ff.] enthält vier VLIW-Slots sowie jeweils zwei MAC-, DIV- und LD/ST-Einheiten. Zudem sind Hardwareerweiterungen für die Algorithmen CRC und ECC integriert. Die Leistungsaufnahme beträgt 169 mW bei einer maximalen Taktfrequenz von 300 MHz und einer Versorgungsspannung von 1,2 V.

Der CoreVA-ULP-ASIC integriert drei CoreVA-CPUs mit jeweils einem VLIW-Slot [107][109]. Zwei CPU-Kerne sind unter Verwendung einer für den Subschwell-Betrieb optimierten Standardzellen-Bibliothek entworfen worden. Diese Bibliothek erlaubt einen Betrieb der CPUs bei Spannungen von 200 mV bis 1,2 V. Die geringste Energie pro ausgeführten Takt (9,94 pJ) wird bei einer Versorgungsspannung von 325 mV und einer Taktfrequenz von 133 kHz erreicht. Zudem integriert der CoreVA-ULP-ASIC 2 kB einen für den Subschwell-Betrieb optimierten SRAM-Speicher [108]. Ein weiterer CPU-Kern basiert auf einer konventionellen Standardzellen-Bibliothek. Bei einer maximalen Taktfrequenz von 260 MHz und einer Spannung von 1,2 V beträgt der Energiebedarf 66 pJ pro Takt.

3.2 Das Cluster im CoreVA-MPSoC

Nachdem im vorangegangenen Abschnitt mit der CoreVA-CPU die Komponenten innerhalb eines eng gekoppelten CPU-Clusters beschrieben sind, wird im Folgenden die

Architektur des CPU-Clusters des CoreVA-MPSoCs vorgestellt (siehe Abbildung 3.3). Als zentrale Verarbeitungseinheiten können theoretisch bis zu 128 CoreVA-CPUs integriert werden. Der Adressraum jeder CPU ermöglicht die Integration von bis zu 1 MB lokalen Scratchpad-Speicher (Software-gesteuerte Speicher) bzw. Cache für Instruktionen und Daten.

Die CPUs verfügen über jeweils eine generische Master- und Slave-Schnittstelle für die Kommunikation innerhalb des CPU-Clusters. Die Slave-Schnittstelle erlaubt Zugriff auf die lokalen Speicher sowie bis zu sechs Hardwareerweiterungen und kann für die Steuerung und Überwachung der CPU verwendet werden. Über die Master-Schnittstelle können die LD/ST-Einheiten der CPU, die Caches sowie die Trace-Schnittstelle mit Slave-Komponenten im Cluster kommunizieren. Dadurch ist es möglich, dass CPUs auf die lokalen L1 Speicher anderer CPUs zugreifen können. Durch die höhere Latenz über die Cluster-Verbindungsstruktur ergibt sich so eine NUMA-Speicherarchitektur im CoreVA-MPSoC. NUMA-Systeme weisen im Vergleich zu klassischen Cache-basierten Uniform Memory Access (UMA)-Systemen typischerweise eine deutlich höhere Speicherbandbreite bei gleichzeitig reduzierter Latenz[2] auf [10, S. 260]. Das generische Protokoll der Master- und Slave-Schnittstellen erlaubt eine Übersetzung auf verbreitete Bus-Standards und wird in [225, S. 5 f.] detailliert beschrieben.

Im Rahmen der Dissertation [162] sind Verbindungsstrukturen nach den Standards AMBA AXI und OpenCores Wishbone entstanden. Die Verbindungsstrukturen sind standardmäßig rein kombinatorisch aufgebaut. Um die maximale Taktfrequenz zu steigern, können Registerstufen zwischen den Mastern und der Verbindungsstruktur sowie zwischen der Verbindungsstruktur und den Slaves integriert werden. Die minimale Latenz eines Lesezugriffes von einer CPU auf einen Slave innerhalb des Clusters beträgt vier

[2] bei Zugriffen auf Speicher in der „Nähe“ einer CPU

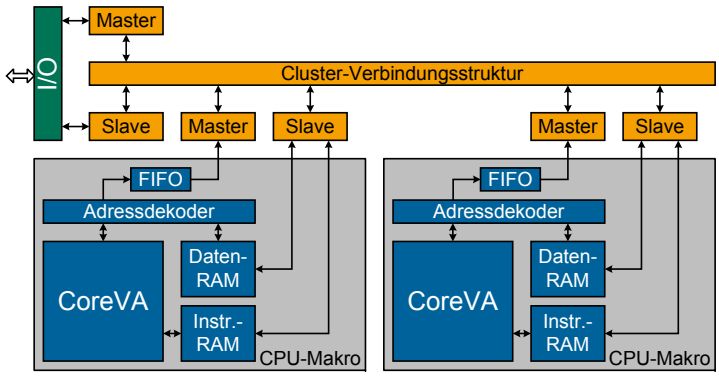


Abbildung 3.3: Blockschaltbild eines CPU-Clusters mit zwei CPUs und I/O-Schnittstelle [162]

Takte. Durch die Verwendung von Registerstufen steigt die Latenz entsprechend an. Alle Komponenten eines Clusters teilen sich einen 32-bit-Adressraum. Die Arbitrierung von Buszugriffen wird durch das Round-Robin-Verfahren [139, S. 26] realisiert. Als Topologie für die Verbindungsstruktur stehen ein geteilter Bus sowie eine partielle und eine vollständige Crossbar zur Verfügung. Bei der Verwendung einer partiellen Crossbar kann die Anzahl an Bus-Slaves pro Bus-Segment frei gewählt werden.

In einem CPU-Cluster kann zudem bis zu 128 MB gemeinsamer L2-Speicher integriert werden. Der gemeinsame Speicher kann aus mehreren Bänken bestehen. Des Weiteren kann der CPU-Cluster um einen DMA-Controller [220], eine Off-Chip-Schnittstelle sowie den NI als Schnittstelle zum On-Chip-Netzwerk des CoreVA-MPSoCs erweitert werden (siehe Kapitel 6).

Als weitere Komponente kann ein eng gekoppelter gemeinsamer Datenspeicher in den CPU-Cluster integriert werden. Als Topologie für die dedizierte Verbindungsstruktur kann eine Crossbar oder ein Mesh of Trees (MoT) zum Einsatz kommen. In beiden Fällen wird eine Round-Robin-Arbitrierung verwendet. Die Anzahl an Speicherbänken ist zur Entwurfszeit konfigurierbar. Die maximale Größe des eng gekoppelten gemeinsamen Speichers beträgt 128 MB. Dieser gemeinsame Speicher innerhalb eines Clusters wird im Rahmen dieser Arbeit an das NoC bzw. den NI gekoppelt (siehe Abschnitt 7.2).

3.3 Hardware-Entwurfsablauf

In modernen Fertigungstechnologien hergestellte Chips enthalten mehrere Milliarden Transistoren [75]. Diese große Zahl an Transistoren erfordert eine hohe Entwurfsproduktivität [94, S. 669]. Im Rahmen dieser Arbeit wird eine *Semi-Custom*-Entwurfsmethode verwendet (siehe Abbildung 3.4) [82, S. 33 f.; 94, S. 673 f.]. Ausgehend von einer (abstrakten) Spezifikation und einer HDL[3]-Beschreibung wird durch mehrere automatisierte oder teilautomatisierte Entwurfsschritte das CoreVA-MPSoC auf eine Zieltechnologie abgebildet. Hierbei kommt eine 28-nm-FD-SOI[4]-Technologie von STMicroelectronics [172] mit zehn Metallisierungsebenen (*Metal-Layer*) zum Einsatz. Des Weiteren werden im Rahmen dieser Arbeit nur reguläre VT-Standardzellen eingesetzt, die im Gegensatz zu den RT-Standardzellen schneller sind, jedoch eine leicht höhere Verlustleistung aufweisen. Bei FD-SOI-Technologien sind die Transistoren durch eine vergrabene, dünne Oxidschicht vom Silizium-Wafer isoliert. Zudem ist der Kanal der Transistoren vollständig verarmt. Dieser Aufbau verringert im Vergleich zur klassischen Fertigungstechnologie (*Bulk*-Prozess) Leckströme und parasitäre Kapazitäten der Transistoren und erlaubt geringere Verzögerungszeiten. STMicroelectronics stellt eine Standardzellenbibliothek sowie verschiedene SRAM-Speicherblöcke in Form von Hardmakros zur Verfügung. Im Rahmen dieser Arbeit werden flächeneffiziente *High-*

[3] *Hardware Description Language*
[4] *Fully-Depleted Silicon-on-Insulator*

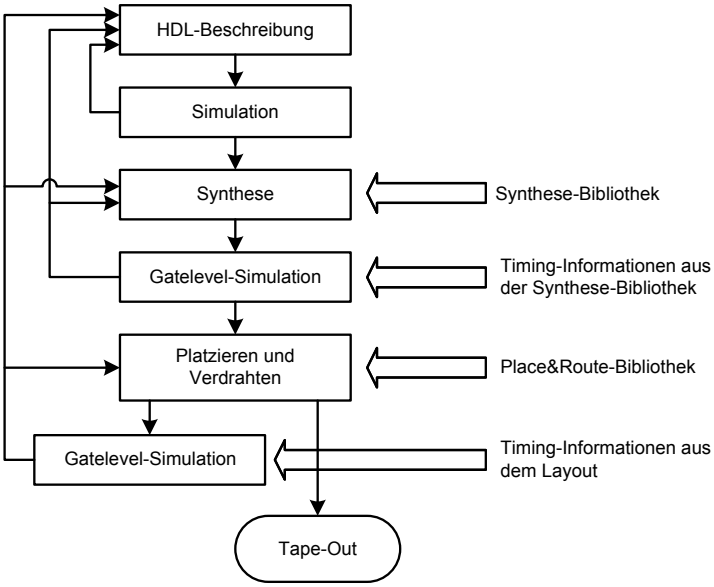


Abbildung 3.4: Hardware-Entwurfsablauf für Standardzellen-Technologien nach [94, S. 673]

Density-Speicher eingesetzt. Es existieren zudem Speicherblöcke mit geringeren Verzögerungszeiten (*High-Speed*-SRAM), die jedoch eine geringere Ressourceneffizienz besitzen.

3.3.1 Synthese

In einem ersten Schritt wird die HDL-Beschreibung in einer RTL-Simulation auf eine korrekte Funktion hin überprüft (siehe Abschnitt 3.3.3). Anschließend wird der als Verhaltensbeschreibung vorliegende HDL-Code in eine Strukturbeschreibung auf Gatter-Ebene überführt. Dieser Schritt wird als (Logik-)**Synthese** bezeichnet. Als Werkzeug für die Synthese wird Genus RTL Compiler 15.20 [25] von Cadence verwendet. Neben der HDL-Beschreibung wird eine Synthesebibliothek von STMicroelectronics verwendet. Die Bibliothek enthält Informationen zu Flächenbedarf, Verlustleistung und Verzögerungszeit der verfügbaren Logikgatter.

Zudem kann der Entwickler Vorgaben für die Synthese (*Design-Constraints*) wie z. B. die gewünschte Taktfrequenz angeben. Als Ausgabe wird eine Netzliste erzeugt, die alle Gatter der Schaltung sowie deren Verschaltung enthält. Diese Netzliste kann für eine erste Gatelevel-Simulation (siehe Abschnitt 3.3.3) verwendet werden.

3.3.2 Platzieren und Verdrahten

Der nächste Schritt ist das **Platzieren und Verdrahten** (*Place and Route*, P&R), bei dem die Netzliste aus der Synthese sowie eine P&R-Bibliothek von STMicroelectronics Verwendung finden. Für das P&R wird Cadence Innovus Digital Implementation System 16.13 [26] verwendet. Die P&R-Bibliothek wird auch als Standardzellen-Bibliothek bezeichnet, da alle Logikzellen eine identische Höhe besitzen [94, S. 674 f.; 165, S. 142 ff.]. Die Logikfunktionen der einzelnen Zellen sind durch zwei oder mehr Transistoren in CMOS[5]-Technik realisiert. Je nach Funktion und Treiberstärke unterscheiden sich die Zellen bezüglich ihrer Breite. Die Standardzellen enthalten Transistoren und verwenden typischerweise die untersten ein bis zwei Metalllagen für die interne Verschaltung. Leitungen für die Versorgungsspannung (z. B. VDD und GND) werden an festgelegten Positionen horizontal durch alle Zellen geführt (siehe [94, S. 677, Abbildung 8.5]).

Der Entwickler muss die Abmessungen des Chips sowie die Position von Makros (z. B. Speicher oder CPU-Kerne, siehe Abschnitt 3.3.5) in einem *Floorplan* festlegen. Es werden Netze für die Versorgungsspannung angelegt, da alle (Logik-)Zellen mit VDD und GND versorgt werden müssen. Diese Netze müssen ausreichend dimensioniert sein, damit in einzelnen Teilen der Schaltung kein übermäßiger Spannungsabfall aufgrund eines erhöhten Leitungswiderstand auftritt. Es werden ein oder mehrere Bereiche festgelegt, in denen Standardzellen platziert werden können. Die Platzierung der einzelnen Zellen erfolgt automatisiert durch das P&R-Werkzeug. Typischerweise wird für die Verdrahtung der Standardzellen eine Fläche benötigt, die größer als die Summe der Fläche aller Standardzellen ist. Um die Verdrahtung aller Standardzellen zu ermöglichen bzw. zu erleichtern, wird zusätzliche Fläche durch die Verwendung von nichtfunktionalen Zellen (*Filler-Cells*) bereitgestellt. Diese Zellen werden von dem P&R-Werkzeug automatisiert in den Entwurf eingefügt. Die *Utilization* eines Entwurfs bezeichnet den Anteil der Fläche von funktionalen Zellen an der Gesamtfläche. Bei einer Utilization von 0,75 sind beispielsweise 25 % der Gesamtfläche durch nichtfunktionale und 75 % durch funktionale Zellen belegt. Die in Abschnitt 3.3.5 vorgestellten Makros der CoreVA-CPU erreichen beispielsweise eine Utilization von bis zu 96 %.

Um eine Kommunikation des Chips mit der (elektronischen) Außenwelt zu ermöglichen, ist die Integration von sogenannten I/O-Zellen notwendig [94, S. 282 ff.]. I/O-Zellen werden ebenfalls vom Chiphersteller zur Verfügung gestellt und besitzen Schutzschaltungen, um Schäden am Chip durch Überspannungen zu vermeiden. Zudem wird der Chip über spezielle I/O-Zellen an die Versorgungsspannung angeschlossen. Typischerweise werden I/O-Zellen ringförmig um die eigentliche Logik des Chips herum angeordnet.

Nach dem Platzieren aller Zellen werden diese in einem *Routing*-Schritt verbunden. Hierbei muss insbesondere der kritische Pfad der Schaltung berücksichtigt und optimiert werden. Um die gewünschte Taktfrequenz der Schaltung zu erhalten, sind ggf. weitere Optimierungsschritte notwendig. Abschließend wird der Entwurf auf Fehler

[5] *Complementary Metal-Oxide-Semiconductor*

überprüft und wiederum auf Gatterebene simuliert. Sind alle Anforderungen der Spezifikation an den Chipentwurf erfüllt, kann dieser in das GDSII[6]-Format überführt, an den Chiphersteller gesendet und von diesem gefertigt werden (*Tape-Out*) [94, S. 675 f.].

3.3.3 Simulation

Die **RTL-Simulation** einer Multiprozessor-Architektur führt Anwendungen unter Verwendung der HDL-Beschreibung des MPSoCs aus[7]. Die RTL-Simulation ist für die Hardware-Entwicklung unverzichtbar, um die Übereinstimmung der HDL-Beschreibung mit der Spezifikation sicherzustellen [94, S. 674]. Typischerweise wird ein Multiprozessorsystem auf verschiedenen Hierarchieebenen simuliert. Hierzu ist jeweils die Implementierung einer eigenen Testumgebung (*Testbench*) notwendig, die in einer nicht synthetisierbaren HDL-Beschreibung realisiert wird. Das CoreVA-MPSoC kann auf den drei Hierarchieebenen CPU-Kern, CPU-Cluster sowie MPSoC simuliert werden. Um möglichst viele Testroutinen in allen Testbenches wiederverwenden zu können, ist im Rahmen des CoreVA-MPSoC-Projektes eine Testbench-Bibliothek entstanden. Diese Bibliothek enthält beispielsweise Funktionen zur Initialisierung und Steuerung einzelner oder mehrere CPU-Kerne. Zudem ist es möglich, Unterschiede zwischen ASIC- und FPGA-Realisierung des MPSoCs zu simulieren (z. B. Verwendung von unterschiedlichen Speichermakros). Es ist eine einheitliche Schnittstelle für die Ausführung von Regressionstests vorhanden. Dies ermöglicht die automatisierte Simulation von unterschiedlichen MPSoC-Konfigurationen mit verschiedenen Testprogrammen. Eine RTL-Simulation ermöglicht zudem die Aufnahme von Schaltaktivitäten, welche für eine Bestimmung der Leistungsaufnahme des MPSoCs verwendet werden können.

Nachdem die HDL-Beschreibung mittels einer Standardzellen-Synthese in eine Gatter-Netzliste überführt worden ist (siehe Abschnitt 3.3.1), kann eine **Gatelevel-Simulation** durchgeführt werden [94, S. 674 f.]. Hierzu sind neben der Netzliste eine angepasste Testbench, Simulationsmodelle der verwendeten Standardzellen-Technologie sowie das Zeitverhalten der Verbindungen zwischen den einzelnen Standardzellen notwendig. Für das CoreVA-MPSoC kann die Testbench der RTL-Simulation mit marginalen Änderungen übernommen werden. Die Laufzeit einer Gatelevel-Simulation ist deutlich höher im Vergleich zu einer RTL-Simulation. Schaltaktivitäten können jedoch genauer bestimmt werden. Zudem wird die Korrektheit der Synthese überprüft und die HDL-Beschreibung mit der Spezifikation verglichen. Eine noch genauere Simulation ist durch eine **Gatelevel-Simulation** auf der Netzliste nach dem Platzieren und Verdrahten möglich (siehe Abschnitt 3.3.2). Dies entspricht einer finalen Simulation des Chip-Layouts, da hier bereits alle Schaltungsteile (inklusive Treiber und Leitungsverzögerungen) und physikalische Einflüsse wie die Signalintegrität berücksichtigt werden können.

[6] *Graphic Database System 2*

[7] Das CoreVA-MPSoC ist in der Sprache VHDL beschrieben

3.3.4 Analyse der Verlustleistung

Um bereits vor der Herstellung des ASICs eine Abschätzung über die Leistungsaufnahme des Chips oder einzelner Schaltungsteile zu erhalten, kann eine Analyse der Verlustleistung durchgeführt werden. Eine erste Abschätzung bieten die Synthese- und P&R-Werkzeuge Genus und Innovus an. Da erst nach dem P&R alle Schaltungsteile (z.B. Treiber, Taktbaum) des finalen Chips im Design enthalten sind, ist die Abschätzung von Innovus im Vergleich zu Genus genauer. Bei der Abschätzung muss zwischen der statischen und dynamischen Verlustleistung unterschieden werden (siehe Abschnitt 4.1). Während die statische Verlustleistung sehr genau abgeschätzt werden kann, muss das Werkzeug für die dynamische Verlustleistung zufällige Schaltaktivitäten annehmen. Eine Schaltaktivität von 5% zeigte für das CoreVA-MPSoC im Durchschnitt die größte Übereinstimmung mit der Gatelevel-Simulation, kann sich je nach Anwendung jedoch davon unterscheiden. Um aussagekräftige Bewertungsmasse verschiedener Architekturvarianten mit Bezug auf Anwendungen zu erhalten, werden in den Kapiteln 5 und 7 dieser Arbeit genauere Analysen der Verlustleistung durchgeführt. Dazu werden während einer Gatelevel-Simulation entsprechende Anwendungen auf der Netzliste des platzierten und verdrahteten Layouts des CoreVA-MPSoCs ausgeführt und die tatsächlichen Schaltaktivitäten aufgezeichnet. Die Verlustleistung kann anschließend mit diesen Schaltaktivitäten und den Informationen aus der Standardzellen-Bibliothek mit Hilfe des Werkzeugs Cadence Voltus IC PowerIntegrity Solution 15.12 [27] sehr genau bestimmt werden.

3.3.5 Hierarchischer Entwurfsablauf

Das Synthetisieren, Platzieren und Verdrahten eines gesamten MPSoC mit dutzenden von CPUs in einem einzigen Schritt (als sog. flaches Design), stellt sich als unpraktikabel und ineffizient dar. Das Erstellen eines gesamten MPSoC als flaches Design führt zum einen zu sehr langen Laufzeiten der Hardwareentwurfswerkzeuge. Zum anderen zu einem hohen Bedarf an Arbeitsspeicher, sodass große MPSoCs mit mehr als 32 CPUs gar nicht erstellt werden können[8].

Im Rahmen dieser Arbeit ist daher ein hierarchischer Entwurfsablauf entstanden, um die Entwicklung eines skalierbaren MPSoCs zu ermöglichen. Dazu wird das Design des CoreVA-MPSoC während des Hardwareentwurfs in kleinere Teile, sogenannte Makros, aufgeteilt. Insgesamt besteht der hierarchische Entwurfsablauf aus drei Ebenen, die in Abbildung 3.5 beispielhaft durch ein Layout des CoreVA-MPSoC dargestellt werden.

Die erste Ebene bilden CPU-Makros der CoreVA-CPU, um die Reproduzierbarkeit und die Geschwindigkeit von Synthesen auf Ebene eines Cluster-Knotens zu steigern. Dieser Cluster-Knoten bildet das Makro der zweiten Ebene des Entwurfsablaufs. Ein solcher Cluster-Knoten besteht aus mehreren CPU-Makros, die über eine Verbindungsstruktur zu einem CPU-Cluster verbunden werden (siehe Abschnitt 3.2). Des Weiteren sind in

[8] Bei den verfügbaren Servern mit 128GB Arbeitspeicher.

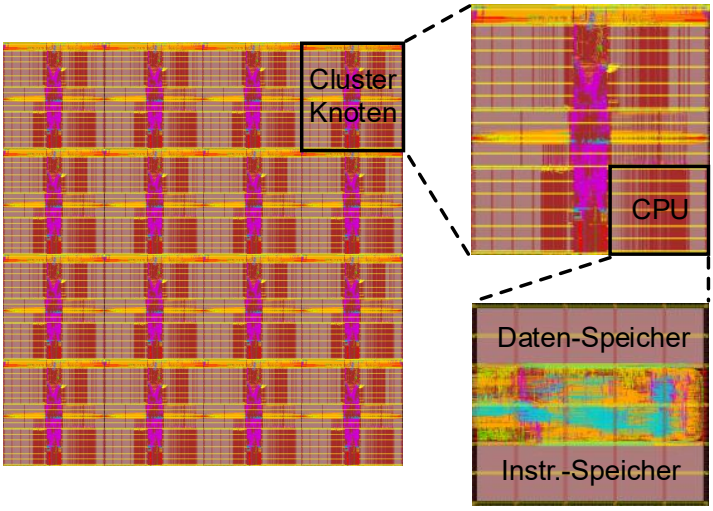


Abbildung 3.5: Layout eines 4x4 2D-Mesh MPSoCs, unter Verwendung des hierarchischen Entwurfsablaufs mit CPU und Cluster-Knoten Makros

einem Cluster-Knoten bereits die NoC-Komponenten wie Router (siehe Kapitel 5) und NI (siehe Kapitel 6) integriert. Auch die Links zwischen den Routern sind bereits bis an die Kanten des Cluster-Knoten Makros geführt. Dies ermöglicht die Verschaltung nahezu beliebig vieler Cluster-Knoten in einem dritten Schritt des Entwurfsablauf, sodass durch Verbindung der NoC-Links ein vollständiges MPSoC - ohne entsprechenden -Ring - erstellt werden kann.

CPU-Makros

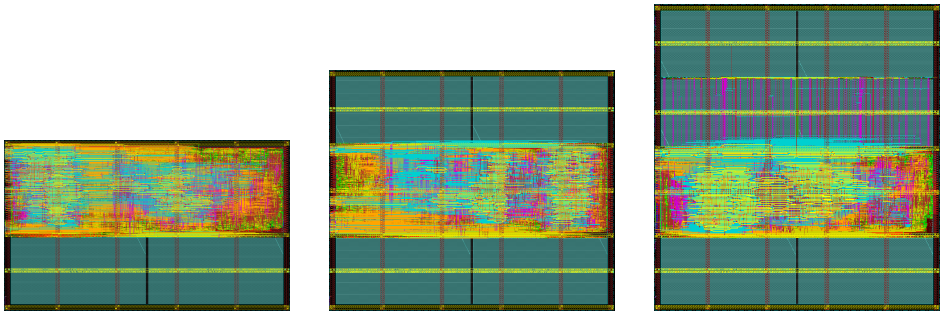
CPU-Makros der CoreVA-CPU werden im Rahmen dieser Arbeit verwendet, um die Reproduzierbarkeit und die Geschwindigkeit von Synthesen auf Ebene des CoreVA-MPSoCs zu steigern. Dazu dienen sie als Basisblock für alle MPSoC-Designs, die im Rahmen dieser Arbeit untersucht werden.

In [73] zeigte Hübener, dass bei der Verarbeitung von Signalverarbeitungsalgorithmen auf der CoreVA-CPU mit aktuellem LLVM-Compiler eine Konfiguration mit 2-VLIW-Slots die höchste Ressourceneffizienz aufweist. Aus diesem Grund wird für den in dieser Arbeit verwendeten Entwurfsablauf des CoreVA-MPSoCs ebenfalls eine 2-Slot Konfiguration für die CPU-Makros verwendet. Grundsätzlich erlaubt die Architektur des CoreVA-MPSoCs die Integration aller Varianten der CoreVA-CPU. Eine Anpassung ist daher möglich, sollte sich in Zukunft eine andere Konfiguration als ressourceneffizienter

erweisen. Grund hierfür könnte z.B. eine Verbesserung des LLVM-Compilers (siehe Abschnitt 3.4.1) sein.

In Abbildung 3.6 sind die Layouts von drei verschiedenen CPU-Makros dargestellt (Makro (a) bis (c)), die in dieser Arbeit eingesetzt werden. Tabelle 3.1 führt die verschiedenen Eigenschaften dieser Makros auf. Die maximale Taktfrequenz beträgt bei jedem Makro 800 MHz. Die Speicherblöcke des lokalen L1-Datenspeichers befinden sich im oberen Bereich der Makros. Speicherblöcke mit höherer Speicherkapazität besitzen eine höhere Flächeneffizienz (kB pro mm^2) im Vergleich zu Speicherblöcken mit geringerer Kapazität. Die Verzögerungszeiten der Speicherblöcke (*Setup* und *Clock-to-Output*) steigen mit höherer Speicherkapazität jedoch an. Solange die geforderte Taktfrequenz erreicht wird, sollten also möglichst wenig Speicherblöcke verwendet werden. Das Makro (a) verfügt über keinen Datenspeicher. Der Datenspeicher des Makros (b) besteht aus jeweils zwei 8-kB-Speicherblöcken (siehe Abbildung 3.6a). Das Makro (c) integriert 32 kB Speicher. Werden hierfür zwei 16-kB-Speicherblöcke verwendet, wird die geforderte Taktfrequenz von 800 MHz nicht erreicht. Daher werden vier 8-kB-Speicherblöcke integriert. Dies führt zu einer Erhöhung des Flächenbedarfs des Datenspeichers um 0,01 mm^2 verglichen mit der Verwendung von zwei Speicherblöcken. Die Fläche des CPU-Makros erhöht sich hierdurch um 7,2 %.

Im unteren Bereich der Markos sind die Speicherblöcke des Instruktionsspeichers angeordnet. Alle Konfigurationen verfügen über 16 kB Instruktionsspeicher, der durch jeweils zwei 8-kB-Speicher aufgebaut ist. Zwischen Instruktions- und Datenspeicher sind die Standardzellen platziert und verdrahtet.



(a) 2 VLIW-Slots, 0 kB DMem, L1-Speicherschnittstelle

(b) 2 VLIW-Slots, 16 kB DMem, L1-Speicherschnittstelle

(c) 2 VLIW-Slots, 32 kB DMem

Abbildung 3.6: Layouts von verschiedenen CoreVA-CPU-Makros mit unterschiedlichen Datenspeicher-Konfigurationen (DMem), jeweils 16 kB Instruktionsspeicher und einer Taktfrequenz von 800 MHz

Tabelle 3.1: Eigenschaften von CPU-Makros der CoreVA-CPU

Makro	VLIW-Slots	MAC	Größe Daten-speicher [kB]	Schnittst. für gem. L1-Datenspeicher	Instr. Speicher [kB]	Datenbreite Instr. Speicher [bit]	Fläche [mm^2]
(a)	2	1	0	ja	16	64	0,082
(b)	2	1	16	ja	16	64	0,116
(c)	2	1	32	nein	16	64	0,148

Umlaufend um die genannten Bereiche herum ist ein Ring von Versorgungsleitungen (*Power-Ring*) für VDD und GND auf den obersten beiden Verdrahtungsebenen (M9 und M10) angeordnet. Zusätzlich besitzen die Makros horizontale und vertikale Versorgungsleitungen auf den Ebenen M9 und M10, um alle Standardzellen und Speicherblöcke niederohmig anzuschließen. Über den Speicherblöcken sind zusätzlich Versorgungsleitungen auf der Ebene M6 integriert. Im Rahmen dieser Arbeit sind die CPU-Makros so beschränkt, sodass nur die unteren acht Verdrahtungsebenen für die Verdrahtung der Schaltungselemente aufgewendet werden dürfen. In der zweiten Ebene des hierarchischen Entwurfsablaufs ermöglicht dies dem P&R-Werkzeug lange Leitungen, wie z.B. die NoC-Links, auf den Verdrahtungsebenen M9 und M10 über die CPU-Makros zu etablieren. Die I/O-Anschlüsse der CPU sind an der linken Seite des Makros platziert. Aufgrund des 64-Bit Datenbus im Cluster eines NoC-Knoten (siehe Kapitel 7.2), besitzen die Makros jeweils 197 I/Os für die Master- sowie 171 I/Os für die Slave-Bus-Schnittstelle. Die Makros (a) und (b) verfügen zudem über 182 I/O-Anschlüsse für den gemeinsamen L1-Datenspeicher auf Clusterebene.

Alle Makros besitzen eine einheitliche Breite von 370,6 µm. Diese ist durch die Breite der Speichermakros vorgegeben. Die Höhe der Makros variiert von 220 µm (Makro (a)) bis 398,4 µm (Makro (c)). Der hieraus resultierende Flächenbedarf liegt zwischen 0,082 mm^2 und 0,148 mm^2. Die Makros (a) und (b) verfügen im Gegensatz zu Makro (c) über eine Schnittstelle zur Integration eines gemeinsamen L1-Datenspeichers auf Cluster-Ebene, die sonstigen Parameter sind identisch. Durch die Integration der L1-Datenspeicher-Schnittstelle erhöht sich der Flächenbedarf des CPU-Kerns um 3,6 %. Das Makro (a) verfügt ebenfalls über diese Schnittstelle, integriert jedoch keinen lokalen L1-Datenspeicher. Die Flächenausnutzung (*Utilization*) der CPU-Makros weist im Standardzellenbereich sehr hohe Werte zwischen 90 % (Makro (a)) und 96 % (Makro (c)) auf.

3.4 Software-Entwicklungsumgebung

Eingebettete Many-Core-Systeme stellen große Herausforderungen an die Entwickler von Softwareanwendungen. Eine effiziente Entwicklung von Software für das CoreVA-MPSoC ermöglicht die in diesem Abschnitt vorgestellte Software-Entwicklungsumgebung.

Um Anwendungsentwicklern von eingebetteten Systemen eine komfortable Entwicklungsumgebung zur Verfügung zu stellen, werden Anwendungen typischerweise nicht direkt auf der Zielplattform übersetzt. Auch die Entwicklungsumgebung des CoreVA-MPSoCs wird auf einem x86-Linux ausgeführt, um dort komfortabel Maschinencode für die Ausführung auf dem CoreVA-MPSoC zu erzeugen. So wird zur Überführung einer Hochsprache in den Maschinencode für eine einzelne CoreVA-CPU ein *Cross-Compiler* eingesetzt (siehe Abschnitt 3.4.1). In Abschnitt 3.4.3 wird ein Kommunikationsmodell vorgestellt, welches die Programmierung vieler CPUs innerhalb des CoreVA-MPSoCs ermöglicht. Des Weiteren besteht die Software-Entwicklungsumgebung aus dem CoreVA-MPSoC-Compiler [206][203], welcher eine automatische und optimierte Abbildung von Streaming-Anwendungen auf das CoreVA-MPSoC ermöglicht (siehe Abschnitt 3.4.4). Zusätzlich steht zur Unterstützung bei Entwicklung von Anwendungen ein zyklenakkurater Simulator (Abschnitt 3.4.5) des CoreVA-MPSoCs zur Verfügung.

3.4.1 LLVM-Compiler

Das *Open-Source*-Projekt LLVM[9] [103] entwickelt eine Compilerinfrastruktur bestehend aus einem modularen Compiler, einer C/C++-Laufzeitumgebung, einem Debugger sowie etlichen weiteren Komponenten. Gegenüber anderen Compilern zeichnet sich LLVM durch eine besonders schnelle Übersetzung von Anwendungen aus [97]. Typischerweise verwenden moderne Compiler einen dreistufigen Übersetzungsprozess [105, S. 50 f.].

Das *Frontend* analysiert und validiert den Quelltext der zu übersetzenden Anwendung. LLVM bietet Unterstützung für eine Vielzahl von Programmiersprachen wie Python, Fortran, Ruby und C#. Clang ist ein LLVM-Frontend für die Sprachen C, C++ sowie Objective-C. Anschließend wird der Quelltext in eine abstrakte, sprachunabhängige Beschreibung überführt. LLVM verwendet hierzu die Zwischensprache LLVM IR[10]. Eine Optimierungsstufe führt Transformationen und Vereinfachungen auf dieser Darstellung durch. In der Codeerzeugungs- oder *Backend*-Stufe wird die abstrakte Beschreibung in die Maschinensprache der Zielplattform übersetzt.

Das LLVM-Backend für die CoreVA-CPU ist im Rahmen von studentischen Arbeiten entstanden [182]. Es unterstützt neben VLIW-Vektorisierung viele Eigenschaften der CoreVA-CPU wie den *Delay*-Slot bei Sprüngen oder das bedingte Ausführen von Instruk-

[9] Modulare Compilerarchitektur, früher *Low Level Virtual Machine*
[10] *LLVM Intermediate Representation*

tionen. Die SIMD-Funktionalität der CoreVA-CPU kann bisher nur rudimentär durch den Einsatz spezieller Datentypen verwendet werden. Die LLVM-basierte Entwurfsumgebung für die CoreVA-CPU bietet Unterstützung für die Einbettung von Assembler-Maschinencode in C-Programme.

3.4.2 Software-Basisfunktionen

Im Rahmen des CoreVA-MPSoC-Projektes ist eine Systembibliothek mit verschiedenen Software-Basisfunktionen entstanden, die über definierte Programmierschnittstellen (Application Programming Interface (API)) Zugriff auf Systemfunktionen sowie eine On-Chip- und Off-Chip-Kommunikation ermöglichen. Hierdurch vereinfacht sich die Programmierung des CoreVA-MPSoCs für Anwendungsentwickler und es ist mit weniger Aufwand möglich, die volle Leistungsfähigkeit des MPSoCs auszunutzen [64, S. 129 ff.]. Die Schnittstellen für den Zugriff auf Systemfunktionen und Kommunikation sind in allen Abstraktionsebenen der Simulations- und Emulationsumgebung des CoreVA-MPSoCs integriert (siehe Abschnitt 3.4.5).

Die Systembibliothek des CoreVA-MPSoCs ermöglicht die Initialisierung von einzelnen CPUs sowie das Starten und Stoppen von Anwendungen. Eine weitere wichtige Komponente ist die `malloc`-Funktion zur Allokation von Speicher. Es ist möglich, Speicher im lokalen L1-Datenspeicher jeder CPU sowie in den gemeinsamen L1-Speichern innerhalb eines CPU-Clusters zu allozieren (siehe Abschnitt 7.2.2 sowie [217, S. 18 ff.]). Zudem stehen Funktionen zur Synchronisierung von CPU-zu-CPU-Kommunikation innerhalb eines Clusters und über das NoC zur Verfügung (siehe Abschnitt 3.4.3).

Eine effiziente Off-Chip-Kommunikation basiert ebenfalls auf dem in Abschnitt 3.4.3 vorgestellten blockbasierten Kommunikationsmodell. Es ist möglich, eine Datei wortweise auf einem Host-System zu lesen bzw. zu beschreiben. Hierbei kann jedes Datenwort die Größe von einem, zwei oder vier Byte besitzen.

Um eine Ausgabe von Zeichenketten (`printf`) aller CPUs des CoreVA-MPSoCs zu ermöglichen, ist im Rahmen dieser Arbeit die Implementierung einer Standardausgabe entstanden [218]. Neben dem auszugebenden Zeichen (*Character*) werden die NoC-Koordinaten des jeweiligen CPU-Clusters sowie die Nummer der CPU an ein FIFO gesendet. Dieses FIFO wird vom Host-PC ausgelesen. Sobald eine komplette Ausgabezeile einer CPU empfangen worden ist, wird diese vom Host-PC ausgegeben. Des Weiteren stehen Bibliotheken für die Kommunikation mit Aktoren und Sensoren des AMiRos [181] sowie den RAPTOR-Modulen DB-ETG[11] und DB-TFT[12] [218] zur Verfügung (siehe Abschnitt 8.2). Das DB-TFT kann zudem für die Darstellung der Standardausgabe verwendet werden.

[11] *Daughterboard Ethernet Gigabit*
[12] *Daughterboard Thin-Film Transistor*, ein Bildschirmmodul

3.4.3 Kommunikationsmodell

Auf einem Multiprozessorsystem können verschiedene Anwendungen bzw. Anwendungsteile parallel ausgeführt werden. Greifen verschiedene Anwendungsteile (Prozesse oder auch *Threads*) auf gemeinsame Ressourcen (z. B. Speicher oder einen Beschleuniger) zu, so ist eine Synchronisierung erforderlich. Eine Synchronisierung ist auch notwendig, wenn ein oder mehrere Prozesse ein Ergebnis produzieren (**Erzeuger**), das von einem oder mehreren anderen Prozessen konsumiert wird (**Konsumenten**). Eine Synchronisierung kann in Hardware oder in Software realisiert werden. Oft wird eine Kombination aus beidem verwendet, um eine hohe Effizienz bei gleichzeitig hoher Flexibilität zu ermöglichen [70, S. 237 f.; 140, S. 137 f.].

Verschiedene Mutex[13]-basierte Mechanismen werden bereits in [162, S. 53] zur Synchronisierung innerhalb eines CPU-Clusters vorgestellt. Ein Mutex (auch *Lock* genannt) basiert auf dem Verfahren des wechselseitigen Ausschlusses. Hierbei kann zur selben Zeit nur ein Prozess den Mutex besitzen bzw. einen **kritischen Abschnitt** betreten [187, S. 97]. Schlägt der Versuch eines Prozesses, einen Mutex zu erwerben, fehl, so blockiert dieser Prozess, bis er den Mutex erhält. Optional kann es möglich sein, den Zustand vom Mutex abzufragen (*Test*), ohne dass der Prozess blockiert [187, S. 97].

Nachrichtenbasierte Kommunikation ist eine Alternative zur Synchronisierung mittels Mutex. Ein Sender verschickt die Nutzdaten zusammen mit Zusatzinformationen wie der Länge der Nachricht an den Empfänger [140, S. 641 f.]. Benötigt der Sender eine Bestätigung, dass die Nachricht richtig empfangen worden ist, kann der Empfänger eine Bestätigungsnachricht an den Sender verschicken. Ein NoC stellt eine Hardwareimplementierung von nachrichtenbasierter Kommunikation dar. Der Sender gibt die Adresse des Empfängers an und die Daten werden als Paket zum Empfänger weitergeleitet. Alle MPSoCs aus der Forschung (siehe Abschnitt 2.2) kommunizieren primär über Nachrichten. Bei den kommerziellen MPSoCs kommuniziert der Kalray MPPA-256 ebenfalls mittels Nachrichten über das NoC. Andere kommerzielle MPSoCs, wie der Epiphany, arbeiten mit einem globalen Adressbereich (siehe Abschnitt 2.3), sodass hier wie im Cluster des CoreVA-MPSoC auch eine rein mutexbasierten Synchronisierung eingesetzt werden kann.

Auch in einem CPU-Cluster mit gemeinsamem Adressraum kann mithilfe von mutexbasierten Verfahren eine nachrichtenbasierte Kommunikation realisiert werden. Diese kann abhängig von der Anwendung effizienter sein als die Synchronisierung und Kommunikation mittels gemeinsamen Speichers. In Rahmen von [162] und dieser Arbeit ist ein neuartiges mutex- und nachrichtenbasiertes Synchronisierungsverfahren entstanden. Das Synchronisierungsverfahren arbeitet auf verschiedenen Blockgrößen oder – je nach Sichtweise – Abstraktionsebenen. Bei einer Synchronisierung auf Wortebene wird (in Software oder Hardware) vor jedem Zugriff auf den Speicher überprüft, ob die entsprechenden Daten gültig sind. Eine blockbasierte Kommunikation fasst mehrere Wörter zusammen. Die Daten eines Datenblocks sind typischerweise hintereinander in

[13] *Mutual Exclusion*

einem Speicher abgelegt. Bei einer Synchronisierung auf Funktions- oder Task-Ebene wird der Zugriff auf mehrere Datenblöcke mithilfe eines Synchronisierungsmechanismus verwaltet. Je mehr Daten zusammen verwaltet werden, desto geringer ist der Mehraufwand für die Synchronisierung. Ist die Blockgröße jedoch zu groß gewählt, kann dies die Latenz der Kommunikation erhöhen. Zudem kann ein Mehraufwand (z.B. erhöhter Speicherbedarf) entstehen, wenn Sender und Empfänger unterschiedlich viele Daten pro Aufruf verarbeiten.

Bei blockbasierter Kommunikation unter Verwendung eines einzelnen Datenblocks kann immer nur ein Kommunikationspartner auf diesen Datenblock zugreifen. Der zweite Kommunikationspartner muss warten, bis der Datenblock freigegeben ist. Um diesen Flaschenhals zu umgehen, können mehrere Datenblöcke verwendet werden (*Multi-Buffering*). Dies ermöglicht einen parallelen Zugriff auf unterschiedliche Blöcke. Werden genau zwei Datenblöcke verwendet, wird die Bezeichnung *Double-Buffering* verwendet.

In der Basisarchitektur des CoreVA-MPSoCs besitzt jede CPU einen lokalen L1-Scratchpad-Datenspeicher, auf den andere CPUs eines Clusters zugreifen können (NUMA, siehe Abschnitt 3.2). Lesezugriffe auf den lokalen Speicher weisen eine Latenz von zwei Takten auf. Lesezugriffe auf Speicher anderer CPUs besitzen eine deutlich höhere Latenz. Schreibzugriffe auf den Speicher anderer CPUs im Cluster werden durch die Verwendung eines FIFOs entkoppelt und verursachen, bei geringer Busauslastung, keine weiteren CPU-Takte bzw. Latenzen als Schreibzugriffe auf den lokalen Datenspeicher. Aus diesem Grund wird auf Lesezugriffe über den Bus verzichtet.

Um dies zu realisieren, wird der Speicherbereich für die Nutzdaten im Speicher des Konsumenten alloziert. Auf diesen Speicher greift der Erzeuger nur schreibend zu. Es wird ein Mutex-Paar verwendet, bei dem sich je ein Mutex im Speicher des Erzeugers (**Sender-Mutex**) und des Konsumenten (**Receiver-Mutex**) befindet. Der Sender-Mutex gibt an, ob der Sender Daten schreiben darf (Eins) oder nicht (Null). Dieser kann durch die Synchronisierungsfunktion **getWriteBuf** vom Sender abgefragt und durch **setReadBuf** vom Empfänger wieder freigegeben werden. Dementsprechend gibt eine Eins im Receiver-Mutex an, dass die Daten gültig sind und vom Konsumenten gelesen werden können (getReadBuf). Dieser kann durch die Synchronisierungsfunktion **getReadBuf** vom Empfänger abgefragt und durch **setWriteBuf** vom Sender wieder frei gegeben werden. In Abbildung 3.7 ist anhand eines Beispiels der Ablauf eines Synchronisierungszyklus dargestellt.

Die Kommunikation über das NoC verhält sich aus Sicht der CPUs exakt nach dem gleichen Schema. Der einzige Unterschied ist, dass hier die nachrichtenbasierte NoC-Kommunikation zwischengeschaltet ist. Eine mutexbasierte Synchronisierung, wie zuvor für das Cluster beschrieben, findet daher jeweils nur lokal innerhalb der Cluster zwischen CPU und NI statt. Der NI überträgt schließlich den Datenblock über die Verbindungsstruktur des NoC (siehe Kapitel 5) zum NI des empfangenden Clusters. Im empfangenden Cluster wird schließlich ebenfalls ein mutexbasierter Kommunikationskanal zwischen NI und CPU verwendet. Die Kommunikation zwischen den CPUs und

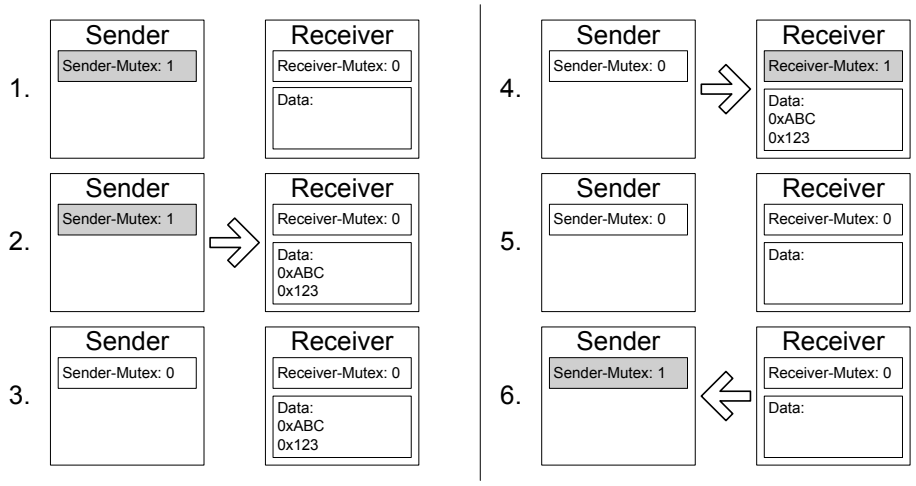


Abbildung 3.7: Blockbasierte Kommunikation im CoreVA-MPSoCs [162]

den NIs und die daraus resultierenden Softwarekosten sind in Abschnitt 6.3.1 näher beschrieben.

Die Anzahl an Kommunikationskanälen innerhalb eines CPU-Clusters ist nur durch die Größe der On-Chip-Speicher begrenzt. Im NoC ist die Anzahl an Kommunikationskanälen zusätzlich durch die Netzwerk-Schnittstelle begrenzt (siehe Abschnitt 6.3.2).

Die vorgestellten Synchronisierungsverfahren sind in der Programmiersprache C implementiert. Es stehen Funktionsaufrufe für die Initialisierung der verschiedenen Kommunikationskanäle (Cluster, NoC) zur Verfügung. Das Synchronisierungsverfahren ist zudem in den Veröffentlichungen [201; 206; 210] beschrieben.

3.4.4 CoreVA-MPSoC-Compiler für Streaming-Anwendungen

Eine effiziente Programmierung von MPSoCs mit dutzenden oder hunderten von CPU-Kernen stellt Programmierer vor eine große Herausforderung. Es ist notwendig, mögliche Parallelität innerhalb einer Anwendung zu erkennen und die Anwendung anschließend auf die CPU-Kerne zu partitionieren. Als Partitionierung wird die Abbildung der verschiedenen Anwendungsteile (Tasks) auf dem MPSoC bezeichnet, d.h. welcher Task konkret auf welcher CPU ausgeführt wird. Hierbei existiert eine Vielzahl an gültigen Partitionierungen. Es ist für den Programmierer nur schwer möglich, eine optimale Partitionierung z. B. in Bezug auf die Anforderungen der Anwendung zu finden. Aus diesem Grund wird in der Arbeitsgruppe Kognitronik und Sensorik der Universität Bielefeld an der automatisierten Partitionierung von Anwendungen geforscht. Eine

automatisierte Partitionierung vereinfacht zudem die Entwurfsraumexploration von verschiedenen Hardware-Konfigurationen eines MPSoCs.

Anwendungen, die kontinuierliche Datenströme verarbeiten, werden als *Streaming*-Anwendungen bezeichnet. Typische Streaming-Anwendungen kommen z. B. aus den Bereichen Signal-, Sensor-, Bildverarbeitung und Verschlüsselung. In Kooperation mit der QUT in Brisbane, Australien ist ein Compiler für die am Massachusetts Institute of Technology (MIT) spezifizierte Sprache StreamIt entstanden [57; 177; 179]. Die StreamIt-Sprache ermöglicht die Beschreibung einer Streaming-Anwendung als Datenflussgraph, der aus sogenannten Filtern besteht. Eine kleine Beispielbeschreibung einer Streaming-Anwendung ist im Anhang B zu finden. Jeder Filter kann als atomarer Block betrachtet werden, der eine definierte Menge an Eingangsdaten einliest, diese verarbeitet und eine definierte Menge an Ausgangsdaten erzeugt. Der eigentliche Arbeitsschritt (Work-Funktion) kann innerhalb eines Filters in einer C ähnlichen Syntax beschrieben werden. Die einzelnen Filter sind mittels Pipeline- und *Split-Join*-Strukturen verbunden. Ein Filter ist zustandslos (*Stateless*), falls seine Ausgabe nur von dem aktuellen Eingabewert abhängig ist. Ist dies nicht der Fall, ist der Filter zustandsbehaftet (*Stateful*). Diese Art der Beschreibung ermöglicht es, die inhärente Parallelität einer Anwendung als Task-, Pipeline- und Daten-Parallelität auszudrücken, ausführliche Informationen dazu sind in [162, S. 86] zu finden.

Der CoreVA-MPSoC-Compiler liest ein StreamIt-Programm ein und erzeugt einen abstrakten Syntaxbaum (Abstract Syntax Tree (AST)). Aus dem AST wird ein hierarchischer Filter-Graph erzeugt. In diesem Graph muss sichergestellt sein, dass jeder Filter genau so viele Daten konsumiert, wie der vorangegangene Filter erzeugt hat. Falls notwendig, wird ein Filter hierzu mehrfach aufgerufen. Dies führt zu einem stabilen Zustand (*Steady-State*), der beliebig oft wiederholt werden kann. Um Verklemmungen aufgrund von Daten-Abhängigkeiten zu vermeiden, konsumiert jeder Filter nur Daten, die in einem vorangegangenen Steady-State produziert worden sind.

Der AST kann verschachtelte Pipeline- und *Split-Join*-Strukturen enthalten. Im *Flattening*-Schritt wird der AST in einen „flachen“ Graphen überführt, der ausschließlich aus Filtern besteht, die direkt mittels Kanten bzw. Kommunikationskanälen verbunden sind. Dieser Graph kann anschließend auf die CPUs des CoreVA-MPSoCs partitioniert werden, welches in Abbildung 3.8 verdeutlicht ist.

Zur Partitionierung stehen drei Algorithmen zur Verfügung [206]. Im Rahmen dieser Arbeit wird als Optimierungsziel der Durchsatz der Anwendung maximiert oder die Latenz minimiert. Aktuelle Forschungsprojekte erweitern den Compiler um weitere Optimierungsziele wie z. B. die Energie der Anwendung. Des Weiteren berücksichtigt der CoreVA-MPSoC-Compiler die Verfügbarkeit von physikalischen Ressourcen, wie Speicher und NoC-Kommunikationskanäle. Diese Funktionalität wird auch im Rahmen dieser Arbeit zur Bewertung verschiedener Hardware-Konfigurationen verwendet (siehe Abschnitt 6.3.2 und 7.3). Der CoreVA-MPSoC-Compiler ermöglicht es, ein Optimierungsziel zu verfolgen, während alle anderen Anforderungen von der Anwendung bzw. der Hardware vorgegeben werden (siehe Abbildung 3.9).

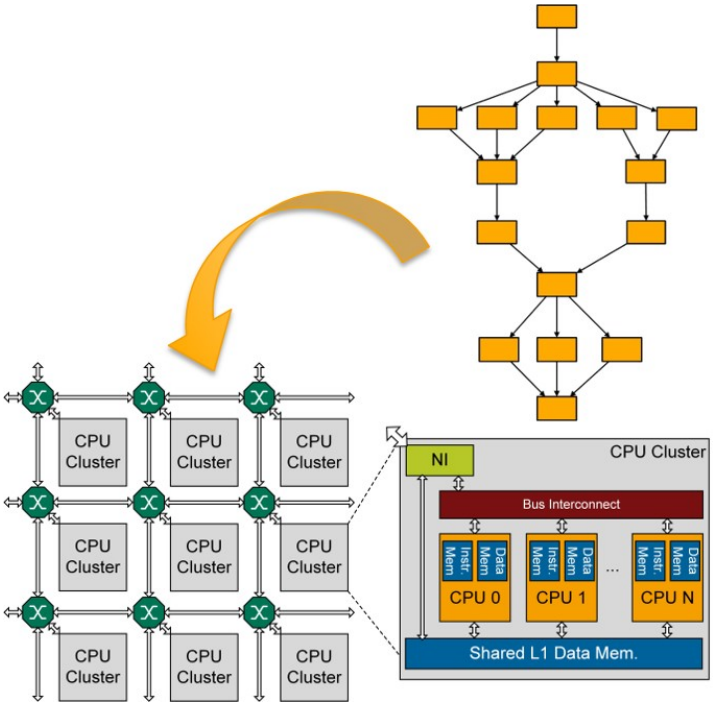


Abbildung 3.8: Der CoreVA-MPSoC-Compiler partitioniert den Graphen einer StreamIt-Anwendung auf das CoreVA-MPSoC

Der *Greedy*[14]-Algorithmus weist in jedem Schritt den Filter mit der längsten Laufzeit der CPU mit der geringsten Auslastung zu [95, S. 8]. Dies wird so lange wiederholt, bis jeder Filter einer CPU zugewiesen worden ist. Mögliche zusätzliche Taktzyklen für die Kommunikation zwischen Filtern werden nicht berücksichtigt. Alternativ kann ein im Rahmen des CoreVA-MPSoC-Projektes entwickelter und in [206] veröffentlichter *Simulated-Annealing*[15]-Optimierungsalgorithmus verwendet werden. Hierbei wird das Ergebnis des Greedy-Algorithmus als Start-Partitionierung verwendet. Anschließend wird die Partitionierung verändert, indem z. B. ein Filter auf eine andere CPU verschoben wird. Zudem ist es möglich, einen zustandslosen Filter auf zwei oder mehr CPUs zu replizieren. Als weiterer Freiheitsgrad kann der Optimierungsalgorithmus die Anzahl der Daten erhöhen, die pro Steady-State-Iteration erzeugt werden (Multiplikator). Um eine bestimmte Partitionierung z. B. bezüglich des erzielbaren Durchsatzes oder der Latenz zu bewerten, kann eine Simulation auf einem Instruktionssatzsimulator

[14] gierig
[15] simulierte Abkühlung

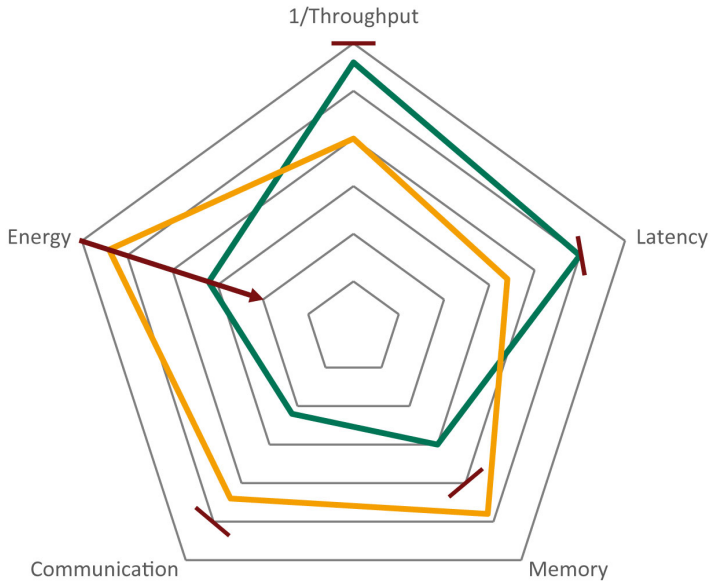


Abbildung 3.9: Optimierungsziele des CoreVA-MPSoC-Compiler

durchgeführt werden. Da der Optimierungsalgorithmus allerdings viele hunderte oder tausende verschiedene Partitionierungen bewertet, würde dies zu einer sehr langen Laufzeit des Compilers führen. Als Alternative wird daher ein Schätzer für die Performanz von Streaming-Anwendungen („*Performance-Estimator*“) entwickelt, der eine Partitionierung wesentlich schneller bewertet (siehe Abschnitt 7.4). Hierzu wird die vorab bestimmte Laufzeit für jeden Filter eingelesen und, zusammen mit Abschätzungen für die Kommunikation zwischen den Filtern, eine Gesamtlaufzeit der Partitionierung bestimmt. Das im Performance-Estimator verwendete Modell des CoreVA-MPSoCs ist in Abschnitt 7.4 beschrieben. Weitere Details zum Performance-Estimators sind in einem technischen Bericht [198] und dem Konferenzbeitrag [203] veröffentlicht. Als dritter Partitionierungs-Algorithmus steht eine Portierung des in [58] von Gordon et al. vorgestellten Algorithmus zur Verfügung.

Die Art des Kommunikationskanals, mit dem zwei Filter verbunden sind, hängt von der Platzierung der Filter ab. Sind beide Filter auf derselben CPU platziert, kann die Kommunikation über den Scratchpad-Speicher dieser CPU erfolgen (*Memory-Channel*). Da beide Filter sequenziell nacheinander ausgeführt werden, ist keine Synchronisierung erforderlich. Werden die Filter auf zwei unterschiedlichen CPUs des gleichen Clusters ausgeführt, wird ein *Cluster-Channel* verwendet. Hierbei wird eines der in Abschnitt 3.4.3 vorgestellten Kommunikationsmodell eingesetzt. Sind beide Filter auf unterschiedlichen Clustern des MPSoCs platziert kommt ein *NoC-Channel* zum Einsatz.

Abschließend erzeugt der CoreVA-MPSoC-Compiler C-Code für jede CPU des MPSoCs. Dieser kann mithilfe des LLVM-basierten C-Compilers in einen Maschinencode übersetzt werden. Weitere Details zur Implementierung des CoreVA-MPSoC-Compilers für Streaming-Anwendungen sind in der Bachelorarbeit [95] und den Veröffentlichungen [203; 206] zu finden. Der CoreVA-MPSoC-Compiler wird in Kombination mit dem *Simulated-Annealing*-Partitionierungsalgorithmus im Rahmen dieser Arbeit für alle Untersuchungen von verschiedener Hardware-Konfigurationen des CoreVA-MPSoCs verwendet.

3.4.5 Simulator

Für eine effiziente Software- und Hardwareentwicklung ist eine Simulation und Emulation des Ziel-MPSoCs auf verschiedenen Abstraktionsebenen unabdingbar [176]. Die bereits vorgestellten RTL- und Gatelevel-Simulationen erweisen sich durch ihre sehr langen Laufzeiten für die Hardwareentwicklung und insbesondere für die Softwareentwicklung als äußerst ineffizient. Eine Effizienzsteigerung verspricht die Simulation des MPSoC auf höheren Abstraktionsebenen.

Je nach Abstraktionsebene sind bei der Simulation Kompromisse zwischen Performanz (Laufzeit der Simulation) und Genauigkeit der Simulation zu machen. Als Genauigkeit wird der Unterschied zwischen der Ausführungszeit der Software auf dem simulierten MPSoC und der späteren MPSoC-Hardware bezeichnet. Die Genauigkeit gibt demnach an, wie zyklenakkurat der Simulator im Vergleich zur Hardware ist. In [114] wird gezeigt, dass sich durch Thread-Parallelisierung der Simulation auf mehreren Host-PCs, die Performanz zwar linear zur Host-Anzahl steigern lässt, dies jedoch zu Einbußen in der Genauigkeit führt. Bei einer höheren Genauigkeit der Simulation ist eine häufige Synchronisation zwischen Threads nötig, welches die Skalierbarkeit und damit die Performanz des Simulators einschränkt.

Aus diesem Grund werden für das CoreVA-MPSoC Simulatoren auf zwei unterschiedlich hohen Abstraktionsebenen entwickelt. Die höchste Abstraktionsebene bietet eine auf POSIX[16]-Threads (**x86-Threads** oder Pthreads) basierende Simulationsumgebung. In einem MPSoC tauschen sich die einzelnen (Software-)Anwendungsteile untereinander und mit Hardwareblöcken Daten aus. Für das CoreVA-MPSoC wird die erforderliche Synchronisierung der Kommunikation durch das in Abschnitt 3.4.3 vorgestellte Kommunikationsmodell in einer Software-Bibliothek gekapselt.

Diese Software-Bibliothek kann neben der Schnittstelle zur Hardware des CoreVA-MPSoC auch eine Schnittstelle zu x86-Threads verwenden. Durch die Verwendung dieser x86-Thread-Schnittstelle ist es möglich, eine Softwareanwendung bereits funktional auf einem oder mehreren x86-PCs zu testen. Dieses Vorgehen bietet zudem den Vorteil, dass Entwicklungsumgebungen wie beispielsweise Eclipse oder Microsoft Visual Studio für die Entwicklung, Codeanalyse und Debugging verwendet werden können.

[16] *Portable Operating System Interface*

Die Schnittstelle (*Wrapper*) steht für die Betriebssysteme Windows und Linux zur Verfügung. Zudem ist die Verwendung mehrerer Betriebssystem-Threads möglich, sodass die Simulation durch Parallelisierung auf mehrere x86-Kerne beschleunigt werden kann. Da die Software jedoch direkt auf den x86-CPUs ausgeführt werden geht jegliche Genauigkeit im Vergleich zur Ausführung auf dem CoreVA-MPSoC verloren, sodass kein Zeitverhalten simuliert werden kann. Durch die einheitliche Schnittstelle für den Softwareentwickler, kann die auf x86-Threads simulierte Software anschließend jedoch direkt auf der Hardware oder genaueren Simulationen des CoreVA-MPSoC ausgeführt werden.

Um dem Softwareentwickler eine schnelle Simulation mit hoher Genauigkeit zur Verfügung zu stellen, ist ein **zyklengenauer Multiprozessor-Simulator** entstanden. Dieser Simulator abstrahiert die eigentliche Hardware des CoreVA-MPSoC, hat jedoch aus Sicht der Software das gleiche Zeitverhalten wie diese. Im Kontext der Simulation eines einzelnen Prozessorkerns wird hierzu der Begriff **Instruktionssatzsimulator** (ISS[17]) verwendet. Ein ISS abstrahiert die Eigenschaften einer Mikroarchitektur [78, S. 42 f.]. Beispielsweise wird die Pipeline-Architektur der CPU nicht simuliert. Es wird im Folgenden jedoch ISS als Bezeichnung für den gesamten Simulator des MPSoCs verwendet, da dieser Begriff in der Fachwelt weit verbreitet ist. Ein ISS wird basierend auf der Hardware-Spezifikation entwickelt und steht zu einem frühen Zeitpunkt im Entwicklungsprozess zur Verfügung.

Der CoreVA-MPSoC-Instruktionssatzsimulator (ISS) erlaubt eine zyklengenaue Simulation des CoreVA-MPSoCs auf einem x86-Linux-PC. Der ISS ist modular aufgebaut und besteht aus Komponenten zur Simulation von einer einzelnen CoreVA-CPU, eines CPU-Clusters und des NoCs. Der Simulator ist in der Sprache C implementiert. Als Eingabe dient dem Simulator eine vom LLVM-Compiler und Linker erzeugte Binärdatei für jede simulierte CPU.

Der **CPU-Simulator** führt jede VLIW-Instruktionsgruppe in zwei Schritten aus [214, S. 6 f.]. Zuerst werden die Maschinenbefehle dekodiert und in eine interne Datenstruktur überführt. In einem zweiten Schritt werden die Ergebnisse der Instruktionen berechnet und auf die Registerbank angewendet. Zudem werden Ereignisse (*Events*) für Sprünge und Speicheroperationen erzeugt. Diese Events werden in einem nächsten Schritt ausgewertet. Hierdurch ist es einfach möglich, beispielsweise Hardwareerweiterungen an die CPU anzubinden. Zudem wird für jeden Zugriff von extern über den Bus des Clusters ein Event auf die Slave-Schnittstelle der CPU erzeugt.

Der **Cluster-Simulator** instantiiert mehrere CPU-Simulatoren und bindet die einzelnen CPUs des Clusters an ein Simulationsmodell der AXI- bzw. Wishbone-Verbindungsstruktur an. Eine weitere Komponente ist ein Simulationsmodell des eng gekoppelten gemeinsamen L1-Datenspeichers (siehe Abschnitt 7.2.2).

Der **NoC-Simulator** erweitert den Cluster-Simulator um den NI, der Netzwerk-Schnittstelle zum NoC. Die Funktionalität des NI wird ausführlich in Kapitel 6 beschrie-

[17] *Instruction Set Simulator*

ben und ist abstrakt im Simulator nachgebildet. Dazu werden Buszugriffe ereignisbasiert entgegengenommen und vom NI abgearbeitet. So können gewünschte NoC-Transfers durchgeführt und Flits an die Simulationskomponente der NoC-Verbindungsstruktur übergeben werden, bzw. von dieser entgegengenommen werden. Die Simulationskomponente der NoC-Verbindungsstruktur abstrahiert ebenfalls die Hardware (siehe Kapitel 5) und verhält sich zu dieser zyklenakkurat.

Für die Synchronisierung der Kommunikation zwischen einzelnen Komponenten des Simulators und deren Parallelisierung werden im Master-Projekt [214] verschiedene Implementierungen verglichen, die ebenfalls in [162] vorgestellt werden. Hier zeigte sich, dass durch die häufige Synchronisation zwischen den Komponenten zu jedem simulierten Taktzyklus, die Performanz nicht von einer Parallelisierung profitieren konnte.

Durch die Verwendung des ISS kann eine verlässliche Aussage über die Laufzeit bzw. Ausführungsgeschwindigkeit der Anwendung getroffen werden. Im Vergleich zum x86-Simulator ist es daher möglich, die Anwendung auf ihre Performanz zu untersuchen und zu optimieren. Auch für die Hardwareentwicklung in einem frühen Entwicklungsstadium bietet der ISS Vorteile. So können durch Änderungen am Simulator bereits Hardwarekonzepte ausprobiert werden, sodass eine erste abstrakte Entwurfsraumexploration der Hardware durchgeführt werden kann.

Neben dem bereits erwähnten Simulationsebenen bietet auch eine Emulation auf einem FPGA die Möglichkeit, den MPSoC vor der Chipfertigung zu testen. Außerdem kann hierdurch bereits Software in einem gesamten eingebetteten System entwickelt werden. Der FPGA-Protoyp des CoreVA-MPSoC wird in Kapitel 8.2 vorgestellt.

Ein Vergleich über Genauigkeit und Performanz aller verschiedener Simulations- und Emulationsebenen ist bereits in [162] vorgestellt und in [203] veröffentlicht. So zeigt der ISS einen Fehler in der Genauigkeit von unter 1% im Vergleich zur RTL-Simulation.

4 Bewertungsmaße von NoC-Architekturen

In diesem Kapitel werden verschiedene Bewertungsmaße für NoC-Architekturen in eingebetteten Multiprozessoren vorgestellt und definiert. Diese Bewertungsmaße erlauben eine Entwurfsraumexploration verschiedener Systemkomponenten von NoC-Architekturen, die im weiteren Verlauf der Arbeit betrachtet werden. Außerdem ist mithilfe dieser Maße eine Bewertung auf Systemebene möglich, um bereits während des Entwurfsprozesses verschiedene Konfigurationen des CoreVA-MPSoCs zu beurteilen.

Zunächst werden die Bewertungsmaße Chipfläche und Energie vorgestellt, die den Ressourcenbedarf von eingebetteten Multiprozessoren beschreiben (siehe Abschnitt 4.1). Anschließend werden in Abschnitt 4.2 Bewertungsmaße für die Performanz von NoC-Architekturen aufgezeigt. Die Performanz von NoC-Systemkomponenten kann durch Größen wie die Latenz und Durchsatz bewertet werden. Zusätzlich erfolgt in dieser Arbeit eine Bewertung auf Systemebene durch Benchmarks aus dem Bereich von Streaming-Anwendungen, wie z.B. aus der Signal- und Bildverarbeitung.

Allgemein erlaubt ein ressourceneffizientes System eine möglichst hohe Performanz bei möglichst kleiner Chipfläche und geringem Energiebedarf. Je nach Anwendung und Einsatzgebiet des MPSoCs müssen die verschiedenen Bewertungsmaße unterschiedlich gewichtet werden, da häufig eine direkte Abhängigkeit zwischen den verschiedenen Größen besteht.

4.1 Analyse des Ressourcenbedarfs

Als Ressourcen werden in mikroelektronischen Systemen die Mittel Chipfläche und Energie bezeichnet. Ressourcen sind Betriebsmittel, die für den Betrieb eines Systems erforderlich sind. Die in diesem Abschnitt definierten Ressourcen Chipfläche und Energiebedarf werden benötigt, um das dritte Bewertungsmaß Performanz bereitzustellen.

4.1.1 Chipfläche

Als **Chipfläche** A wird die Fläche bezeichnet, die eine Schaltung auf einem gefertigten Chip benötigt. Die benötigte Chipfläche hängt dabei neben der Schaltungsarchitektur auch maßgeblich von der Herstellungstechnologie des Chips ab. Es muss unterschieden werden, ob die Schaltung ein in sich geschlossenes Gesamtsystem darstellt, welches so

gefertigt und verwendet werden kann, oder eine Teilkomponente (Makro), welche erst mit weiteren Komponenten zu einem Gesamtsystem verschaltet werden muss [162]. Im ersten Fall wird zusätzliche Chipfläche für die Kontaktierung des gefertigten Chips mit der Außenwelt (I/O-Pads) zum Anschluss von Spannungsversorgung und I/O-Signalen verwendet (siehe Abschnitt 8.1).

Da es sich, bei der in dieser Arbeit entwickelten, NoC=Architektur um eine Teilkomponente eines Gesamtsystems handelt, werden die Angaben zur Chipfläche vorwiegend für Teilkomponenten angegeben. Bei NoC-Architekturen setzt sich die Chipfläche zumeist aus den Standardzellen der Logikgatter und Speicherzellen zusammen. Insbesondere bei der NoC-Verbindungsstruktur kann jedoch auch der Platz für Verbindungsleitungen zwischen den Standardzellen den Flächenbedarf dominieren. Diese befinden sich auf den höheren Metalllagen des Chips, sodass die Chipfläche auch von der Anzahl verfügbarer Metalllagen abhängt.

Von Bedeutung ist die Chipfläche insbesondere für wirtschaftliche Interessen, da sich von ihr die **Herstellungskosten** ableiten lassen. Weitere wichtige Faktoren für die Herstellungskosten sind jedoch auch die Fertigungstechnologie, das benötigte Volumen des eingepassten Chips (*Chip-Package*) und die angeforderten Stückzahlen.

4.1.2 Leistungsaufnahme

Die *Leistungsaufnahme P* von CMOS-Schaltungen setzt sich aus der dynamischen und der statischen Verlustleitung zusammen [127, S. 65 f.; 94, S. 269 f.]:

$$P_{gesamt} = P_{dyn} + P_{stat}. \tag{4.1}$$

Die statische Verlustleistung P_{stat} ist auf Subschwellströme von sperrenden Transistoren zurückführen [107, S. 24 f.] und wird als *Leakage* bezeichnet. Aus diesem Grund spielt die Verlustleistung kontinuierlich eine Rolle, selbst wenn sich Schaltungskomponenten in einem passiven Zustand befinden. Vermieden werden kann die statische Verlustleistung nur durch ein vollständiges Abschalten der Versorgungsspannung, durch sogenanntes Power-Gating.

Die dynamische Verlustleistung P_{dyn} ergibt sich durch die Pegeländerung von Eingangssignalen eines Logikgatters oder Speicherelements. Eine Pegeländerung eines Eingangs führt zu einem Umladen von Lastkapazitäten innerhalb des Gatters sowie an dessen Ausgang. Außerdem tritt ein Querstrom (Kurzschlussstrom) auf, wenn sowohl die p-Kanal- als auch die n-Kanal-Transistoren des Gatters kurzzeitig zeitgleich leitend sind. Die dynamische Verlustleistung besteht zu ca. 90 % aus dem Umladen von Lastkapazitäten [144, S. 56]. Aus diesem Grund kann die dynamische Verlustleistung durch folgende Gleichung approximiert werden [94, S. 271]:

$$P_{dyn} = \frac{1}{2} \cdot C_L \cdot {U_{DD}}^2 \cdot f. \tag{4.2}$$

C_L entspricht der Summe aller umgeladenen Kapazitäten der Schaltung und hängt von der Schaltwahrscheinlichkeit sowie der Gesamtkapazität der Schaltung ab. Die Schaltwahrscheinlichkeit gibt an, wie oft sich ein Signal pro Taktzyklus im Durchschnitt ändert. U_{DD} ist die Versorgungsspannung der Schaltung, f die Taktfrequenz.

Während der Analyse der Verlustleistung (siehe Abschnitt 3.3.4), lässt sich die dynamische Verlustleistung funktional noch einmal in eine interne Verlustleistung (engl. internal power) und eine durch Schaltaktivitäten verursachte Verlustleistung (engl. switching power) unterteilen. Die durch Schaltaktivitäten verursachte Verlustleistung ergibt sich durch Änderungen an einem oder mehreren Eingangssignalen, die eine Auswirkung auf das Ausgangssignal einer Zelle haben. Die interne Verlustleistung ergibt sich hingegen durch Änderung eines Eingangssignals ohne Auswirkung auf den Ausgang. Dies spielt insbesondere bei Taktgesteuerten Speicherelementen (Registern) eine Rolle, da sich das Taktsignal häufig auch während des passiven Zustands eines Schaltungselements schaltet. Eine Minderung dieses Effekts ist durch das zeitweise Abschalten des Taktsignals von Schaltungselementen zu erreichen. Lokal für kleine Schaltungsteile kann dies durch sogenanntes Clock-Gating umgesetzt werden.

Eine hohe Leistungsaufnahme führt unter anderem zu einer verminderten Betriebsdauer bei batteriebetriebenen Systemen bzw. erfordert die Verwendung von einer Batterie (bzw. Akkumulator) mit höherer Kapazität. Zudem entsteht Wärme, die ggf. passiv (durch Kühlkörper) oder aktiv (durch Lüfter) vom Chip abgeführt werden muss, um eine Überhitzung zu vermeiden. Größere Batterien und Kühlkörper steigern die Kosten und das Volumen des Systems, in das der Chip integriert wird.

4.1.3 Energiebedarf

Auf Schaltungsebene kann die umgesetzte **Energie** in Form von Verlustleistung pro Takt abgeleitet werden. Die Energie ist in diesem Fall unabhängig von der Taktfrequenz [107, S. 24 f.]:

$$E_{dyn} = \frac{P_{dyn}}{f} = \frac{1}{2} \cdot C_L \cdot {U_{DD}}^2. \tag{4.3}$$

Als Bewertungsmaß auf Anwendungsebene kann die Energie pro Ausgabewert sowie die Energie für das (einmalige) Ausführen einer Anwendung verwendet werden. Hierzu wird die durchschnittliche Leistungsaufnahme während der Ausführungszeit der Anwendung gemessen und diese durch die Ausführungszeit t geteilt:

$$E = \frac{P}{t}. \tag{4.4}$$

Auf den Energiebedarf auf Anwendungsebene wirken sich daher die Bewertungsmaße Verlustleistung und Performanz aus.

4.2 Analyse der Performanz

Für die Performanz von Verbindungsstrukturen zwischen CPUs lassen sich die Bewertungsmaße Durchsatz und Latenz definieren. Im Folgenden werden diese beiden Bewertungsmaße für die Bewertung verschiedener NoC-Architekturen für eingebettete MPSoCs bestimmt. Des Weiteren wird in diesem Abschnitt eine Auswahl von Streaming-Anwendungen (Benchmarks) vorgestellt, die eine zusätzliche Bewertung der Performanz unter typischen Anwendungsszenarien erlauben.

4.2.1 Durchsatz

Der **Durchsatz** D bezeichnet die Anzahl an Ergebniswerten, die pro Zeiteinheit vom System ausgegeben werden. Typischerweise wird der Durchsatz in Ergebnissen pro Sekunde oder als Datenrate (Bytes pro Sekunde) angegeben.

Der Durchsatz kann in einem MPSoC auf verschiedenen Ebenen betrachtet werden. Für das NoC kann der maximale Durchsatz der Verbindungsstruktur betrachtet werden. Dieser gibt an, wie viele Daten pro Zeiteinheit über einen NoC-Link verschickt werden können. Auf Architekturebene wird der Durchsatz häufig in Flits oder Bytes pro Taktzyklus angegeben. Nach der physikalischen Implementierung mit einer bestimmten Taktfrequenz, kann der Durchsatz auch in Bytes pro Sekunde angegeben werden.

Des Weiteren kann der Durchsatz auf Anwendungsebene betrachtet werden. Dieser gibt an, wie viele Daten eine Anwendung in einer gewissen Zeitspanne verarbeiten oder produzieren kann. Auf den Durchsatz auf Anwendungsebene haben alle Komponenten des Systems einen Einfluss. Neben der NoC-Architektur sind dies auch die Software, sowie die CPU- und Cluster-Architektur. Zwei verschiedene Varianten oder Partitionierungen einer Anwendung können anhand ihres Durchsatzes verglichen werden. Die **Beschleunigung** B ist das Verhältnis des Durchsatzes D_1 der zu untersuchenden Konfiguration zum Durchsatz D_0 einer Basiskonfiguration:

$$B = \frac{D_1}{D_0}. \tag{4.5}$$

Um die Skalierbarkeit und den Einfluss des NoC einer Anwendung aufzuzeigen, wird im Rahmen dieser Arbeit als Basiskonfiguration zumeist die Ausführung der Anwendung auf einer CPU verwendet.

4.2.2 Latenz

Die **Latenz** L kennzeichnet in technischen Systemen die Zeit von der Eingabe bis zu dem Zeitpunkt, an dem die entsprechende Ausgabe das System verlässt.

Auch die Latenz kann in einem MPSoC auf verschiedenen Ebenen betrachtet werden. Für das NoC kann dazu die Latenz auf Ebene der Verbindungsstruktur angegeben

werden. Die Latenz gibt hierbei an, wie lange ein Flit oder auch ein gesamtes Paket benötigt, um von der sendenden CPU bis zur empfangenden CPU zu gelangen.

Besonders in Systemen mit Echtzeitanforderungen stellt die Latenz eine wichtige Kennzahl für die Performanz dar (siehe Kapitel 7.5). Je nach „Härte“ der Echtzeitanforderungen muss die Latenz eine gewisse Zeitverzögerung einhalten und diese nicht überschreiten. Echtzeitanforderungen werden in der Regel durch die Anwendung vorgegeben, weshalb hierzu die Latenz auf Anwendungsebene zu betrachten ist. Die Latenz auf Anwendungsebene ist die Zeit, in der die Anwendung ein gewisses Datum nach Eintritt verarbeitet hat und dieses zur Verfügung stellt. Ähnlich wie beim Durchsatz haben auch hier sämtliche Systemkomponenten einen Einfluss auf die Latenz der Anwendung.

Jitter ist die Varianz der Latenz. Ein Jitter kann z. B. auftreten, wenn die Dauer einer Berechnung variiert und von verschiedenen Faktoren beeinflusst werden kann. So kann beispielsweise die Berechnungszeit von den Eingangsdaten abhängen. Des Weiteren können in Prozessorsystemen auch dynamische Systemkomponenten wie Caches zu einem erhöhten Jitter führen.

4.2.3 Benchmark-Auswahl

In diesem Abschnitt werden Anwendungsbeispiele vorgestellt, die im weiteren Verlauf der Arbeit als Benchmarks zur Bewertung der verschiedenen Hardwarekonfigurationen des CoreVA-MPSoCs verwendet werden.

Für das Benchmarking von MPSoCs stehen verschiedene Benchmark-Bibliotheken zur Verfügung. Die am weitesten verbreitete und wohl auch anerkannteste Benchmark-Bibliothek für eingebettete Prozessoren ist die des *Embedded Microprocessor Benchmark Consortium (EEMBC)* [48]. Zwar sind hier mit CoreMarkPro, MultiBench und AutoBench verschiedene Benchmarks für Mehrkern-Prozessoren vorhanden, jedoch ist der Quellcode nur über eine kostenpflichtige Mitgliedschaft verfügbar. Hinzu kommt, dass die Implementierungen der Benchmarks für Systeme mit geteiltem Speicher ausgelegt sind. Dies erfordert einige Anpassungen des Quellcodes, um diesen für den CoreVA-MPSoC nutzbar zu machen. Eine frei verfügbare Benchmark-Bibliothek für eingebettete Prozessoren ist *MiBench* [60]. Diese wurde bereits im Jahr 2001 von der University of Michigan entwickelt [59], ist jedoch nicht als MPSoC-Benchmark-Bibliothek konzipiert, sondern beschränkt sich auf einzelne CPU-Kerne.

Für MPSoCs mit expliziter Kommunikation – wie dem CoreVA-MPSoC – kommen generell Benchmark-Bibliotheken wie *SPEC MPI 2007* oder *ParkBench* in Frage. Beide nutzen den Message Passing Interface (MPI)-Standard [119], um zwischen CPUs zu kommunizieren. Die frei verfügbare Benchmark-Bibliothek *ParkBench* [18] wurde bereits 1996 entwickelt und ist noch in der heutzutage weniger verbreiteten Programmiersprache Fortran implementiert. Die kommerziell verfügbare Benchmark-Bibliothek *SPEC MPI 2007* [170] ist vorwiegend für skalierbare CPU-Systeme mit mehreren tausend CPUs ausgelegt (insbesondere Off-Chip), eignet sich generell aber auch für On-Chip Many-Cores.

Da das CoreVA-MPSoC insbesondere Streaming-Anwendungen für eingebettete und energiebeschränkte Systeme adressiert, eignet sich in dieser Arbeit die StreamIt-Benchmark-Bibliothek [57; 177–179]. Beispiele für Streaming-Anwendungen sind insbesondere in der Signal- und Bildverarbeitung zu finden. Allgemein können aber alle Anwendungen, die sich durch eine kontinuierliche Verarbeitung von Datenströmen durch verschiedene Verarbeitungsschritte (Tasks) auszeichnen, als Streaming-Anwendung beschrieben werden.

Für alle Streaming-Anwendungen, die in dieser Arbeit als Benchmarks verwendet werden, steht daher eine Implementierung in der Sprache StreamIt zur Verfügung. Der größte Teil der StreamIt-Anwendungen basiert auf der StreamIt-Benchmark-Bibliothek [173]. Diese Anwendungen sind nicht speziell für eine Abbildung auf die Architektur des CoreVA-MPSoCs optimiert worden. Da die CoreVA-CPU zurzeit über keine Fließkomma-Einheit verfügt (siehe Abschnitt 3.1), verwenden die Anwendungen hauptsächlich Ganzzahl- bzw. Festkommaarithmetik. Mit Hilfe des CoreVA-MPSoC-Compilers (siehe Abschnitt 3.4.4) können die im Folgenden vorgestellten StreamIt-Anwendungen automatisch auf verschiedenen Konfigurationen des CoreVA-MPSoCs abgebildet werden.

In diesem Abschnitt werden zunächst die Benchmarks aus dem Bereich Signalverarbeitung vorgestellt. Dazu zählen neben verschiedenen digitalen Filtern, einer Fast Furier Transformation (FFT) auch ein Algorithmus zur Laufgeschwindigkeitsbestimmung durch die Klassifikation von Sensordaten eines Körpersensors mittels Support-Vector Machine (SVM). Aus anderen Anwendungsklassen kommen ein Verschlüsselungsalgorithmus, eine Matrixmultiplikation und verschiedene Sortierverfahren zum Einsatz.

Weitere detaillierte Informationen über die Eigenschaften der Anwendungen sind im Anhang in Abschnitt C aufgeführt. In Tabelle C.1 werden beispielsweise die Anzahl der StreamIt-Filter, die CPU-zu-CPU-Kommunikation und die theoretisch erreichbare Beschleunigung für 32 und 128 CPUs angegeben. Des Weiteren werden in Tabelle C.2 weitere Informationen aufgeführt, die sich bei einer Partitionierung auf das CoreVA-MPSoC ergeben.

Digitale Filter

Digitale Filter sind in der Signalverarbeitung weit verbreitet und sind ein wichtiger Teil innerhalb der verschiedensten Anwendungen. Typischerweise ändern Filter ein digitales Signal in Abhängigkeit von der Frequenz. Ein Beispiel ist die Veränderung der Amplitude des Signals in einem bestimmten Frequenzbereich.

Einer der am häufigsten eingesetzten Filter ist der Tiefpassfilter Finite Impulse Response (FIR)-Filter. Aus einem diskreten Eingangssignal x und der Übertragungsfunktion (Filterkoeffizienten) h lässt sich die Ausgabe y des Filters bestimmen (siehe

Gleichung 4.6). M entspricht der „Tiefe“ des Filters bzw. der Anzahl der Filterkoeffizienten.

$$y(n) = \sum_{k=0}^{M-1} h(k) \cdot x(n-k) \tag{4.6}$$

Die Anwendung **LowPassFilter** implementiert einen solchen FIR-Filter und stammt aus der StreamIt-Benchmark-Bibliothek [177, S. 30 f.]. Es werden 64 Filterkoeffizienten verwendet. Zudem wird das Signal um den Faktor zwei herunter getaktet (*Downsampling*).

Die Anwendung **Filterbank** zerlegt ein digitales Signal in acht verschiedene Frequenzbänder, filtert diese und kombiniert die einzelnen Bänder anschließend wieder [57, S. 76]. Für jedes Frequenzband wird das Signal verzögert und eine FIR-Filterung durchgeführt. Hierbei werden 32 Filterkoeffizienten verwendet. Als nächstes wird das Signal herunter getaktet (*Downsampling*) und die Abtastrate anschließend wieder erhöht (*Upsampling*). Die Signale werden wiederum verzögert, FIR-gefiltert und in einem letzten Schritt kombiniert.

Als dritte Anwendung aus dem Bereich Digitale Filter wird der Benchmark **AutoCor** eingesetzt. Bei diesem Benchmark handelt es sich um eine Autokorrelation. In der Signalverarbeitung beschreibt die Autokorrelation eine Korrelation eines Signals mit sich selbst zu einem früheren Zeitpunkt τ. Gleichung 4.7 zeigt die Autokorrelation eines zeitdiskreten Signals mit M Abtastpunkten und der diskreten Verschiebung τ.

$$y(\tau) = \sum_{n=0}^{M-1} x(n) \cdot x(n-\tau) \tag{4.7}$$

Die in dieser Arbeit eingesetzte Autokorrelation arbeitet mit Signalen aus 32 Abtastpunkten und einer diskreten Verschiebung von 8 Elementen. Im Gegensatz zu den anderen beiden digitalen Filtern verwendet **AutoCor** weiterhin Fließkommazahlen, um auch einen Benchmark dieser Art bei den Untersuchungen mit einzubeziehen. Hierzu wird die Fließkommaberechnung in Software auf der CoreVA-CPU emuliert.

Alle betrachteten digitalen Filter stammen aus der StreamIt-Benchmark-Bibliothek. Die auf FIR-Filter basierenden Benchmarks besitzen eine Komplexität von $\mathcal{O}(n)$, da die Berechnung pro Eingabeelement nur von der Summe über die Filterkoeffizienten (siehe Gleichung 4.6) und nicht von der Gesamtanzahl der Eingabeelementen abhängt. Bei der Autokorrelation hängt die Berechnung jedes Ausgabeelements für die Verschiebung τ jedoch von der Größe des Eingangssignals (Abtastpunkte M) ab, sodass sich hier eine Komplexität von $\mathcal{O}(\tau * n)$ ergibt (siehe Gleichung 4.7).

Fast Fourier Transformation (FFT)

Die schnelle Fourier-Transformation (*Fast Furier Transformation*, FFT) kommt in zahlreichen Anwendungsfeldern zum Einsatz. Dazu zählen insbesondere die Bereiche die

Signal-, Video- und Audioverarbeitung. Ein populäres Beispiel ist der Mobilfunkstandard LTE, wo eine FFT in der Basisbandverarbeitung verwendet wird. Eine FFT kann durch die parallele Ausführung auf mehreren CPUs beschleunigt werden [23] und eignet sich damit als Benchmark für MPSoCs. Im Rahmen dieser Arbeit wird eine Radix-2-**FFT** mit 64 Abtastpunkten aus der StreamIt-Benchmark-Bibliothek verwendet. Die Komplexität Radix-2-FFT ist $\mathcal{O}(n \log_2(n))$.

Lineare Support Vector Machine (SVM)

Diese Anwendung ist nicht Teil der StreamIt-Benchmark-Bibliothek, sondern ein in der Dissertation [32] und der Veröffentlichung [33] vorgestellter Algorithmus zur Gang- / Laufgeschwindigkeitsbestimmung durch die Klassifikation von Sensordaten eines Körpersensors. Das Ziel dieses Algorithmus ist es, die Gang- / Laufgeschwindigkeit einer Person basierend auf den Signalen eines Beschleunigungssensors zu bestimmen und zu klassifizieren. Das Gehen und Laufen einer einzelnen Person wird durch die Verwendung eines zweidimensionalen Merkmalraumes und einer SVM mit einer linearen Trennfunktion separiert.

Um den in [32, S. 83] vorgestellten Algorithmus **SVM** als Benchmark für diese Arbeit nutzbar zu machen, wurde dieser in die Sprache StreamIt portiert. Dies ermöglicht es den Algorithmus mit Hilfe des CoreVA-MPSoC-Compilers (siehe Abschnitt 3.4.4) auf verschiedene Hardwarekonfigurationen des CoreVA-MPSoC abzubilden. SVMs haben laut [1] eine Komplexität von $\mathcal{O}(n^3)$.

Matrixmultiplikation

Matrixmultiplikation, auch als Matrizenmultiplikation bezeichnet, ist das mathematische multiplizieren zweier Matrizen. Da dieses ein, in zahlreichen Anwendungsgebieten, eingesetztes Verfahren ist [28, S. 551 ff.], gilt die Matrixmultiplikation für viele Prozessorarchitekturen als klassischer Benchmark. Eine Beispielanwendung ist in der Bildverarbeitung zu finden, wo die Multiplikation von Matrizen für Rotation und Skalierung von Bildern verwendet wird. Zwei Matrizen lassen sich nur multiplizieren, wenn die Anzahl der Spalten der ersten Matrix gleich der Anzahl der Zeilen der zweiten Matrix ist. Gleichung 4.8 zeigt die Multiplikation $C = A \cdot B$ der Matrix A mit der Größe $m \times n$ und der Matrix B mit der Größe $n \times p$. Die Ergebnismatrix C besitzt die Größe $m \times p$.

$$C_{ij} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j} \quad (1 \leq i \leq m, 1 \leq j \leq p) \tag{4.8}$$

C_{ij} ist demnach das Skalarprodukt der Zeile i der ersten Matrix und der Spalte j der zweiten Matrix. Die Komplexität einer Matrixmultiplikation ist daher $\mathcal{O}(m \cdot n \cdot p)$. Für quadratische Matrizen der Größe $n \times n$ ist die Komplexität $\mathcal{O}(n^3)$. Die verwendete

Implementierung **MatrixMult** aus der StreamIt-Benchmark-Bibliothek multipliziert zwei Matrizen der Größe 8×8.

Data Encryption Standard (DES)

Aus dem Anwendungsbereich der Verschlüsselung wird als Benchmark der Data Encryption Standard (DES) eingesetzt. Der DES ist ein symmetrischer Verschlüsselungsalgorithmus mit einer Schlüssellänge von 56 bit bei einer Blockgröße von 64 bit. Zwar wird eine Schlüssellänge von 56 bit heute für viele Anwendungen nicht mehr als sicher erachtet, jedoch lässt sich die Sicherheit durch eine einfache Mehrfachanwendung des DES auf einem Datenwort deutlich erhöhen. Ein Beispiel ist der Triple-DES, der z.B. von Banken in Chipkarten eingesetzt wird.

In [57, S. 72 f.] wird die in dieser Arbeit verwendete StreamIt-Implementierung des **DES** vorgestellt. Diese besteht aus fünf Berechnungsschritten, welche insgesamt viermal nacheinander ausgeführt werden. Die Komplexität des DES ist $\mathcal{O}(n)$, da der DES mit einer festen Blockgröße arbeitet, die unabhängig von der zu verschlüsselten Datenmenge ist.

Sortierverfahren

Des Weiteren bietet die StreamIt-Benchmark-Bibliothek einige Sortieralgorithmen. Auch das kontinuierliche Sortieren von Datenblöcken innerhalb eines Datenstroms, kann als Streaming-Anwendung gesehen werden. Im Rahmen dieser Arbeit werden die Sortierverfahren **BubbleSort**, **BitonicSort** und **RadixSort** als Benchmarks eingesetzt, da sich diese besonders gut für eine parallele Implementierung eignen.

Das Sortierverfahren **BubbleSort** sortiert eine Liste, indem ein Element die Liste von links nach rechts durchläuft und jeweils mit dem nächsten Element („rechter Nachbar“) verglichen und je nach Sortierkriterium vertauscht wird [8]. Dieser Durchlauf eines Elements durch die Liste wird schließlich für jedes Element und damit insgesamt n-mal durchgeführt. Daraus ergibt sich für BubbleSort mit $\mathcal{O}(n^2)$ eine nicht optimale Laufzeit. Der Vorteil ist jedoch, dass für BubbleSort eine einfache Parallelisierung möglich ist, da jeder Vergleich von zwei Nachbarn auf einer anderen CPU ausgeführt werden kann.

RadixSort ist ein stabiles Sortierverfahren und sortiert eine Liste nacheinander für jede Stelle eines Zahlensystems [37, S. 197 ff.]. In digitalen Systemen kommt typischerweise das Binärsystem zum Einsatz. Dabei wird die Liste aufsteigend nach jedem Bit sortiert (beginnend mit Bit 0). Ist der Wert des jeweiligen Bits gleich, bleibt die Reihenfolge der Zahlen unverändert. Ist die Maximalgröße aller Zahlen bekannt, ist die Komplexität mit $\mathcal{O}(n)$ linear zur Größe der zu sortierenden Liste. Allgemein mit einer maximalen Größe der Zahlen von w hat RadixSort eine Komplexität von $\mathcal{O}(n \log_2(w))$. Die in dieser Arbeit verwendete StreamIt-Implementierung sortiert Zahlen mit einer Maximalgröße von 2048, welches in 11 Sortierstufen resultiert.

BitonicSort benötigt bei n zu sortierenden Elementen $\mathcal{O}(n \log_2(n)^2)$ Vergleichsoperationen. Grundsätzlich wird bei BitonicSort ein Datensatz in zwei Datensätze aufgeteilt, die jeweils rekursiv für sich sortiert werden. Anschließend werden die Datensätze zusammengefasst (*BitonicMerge*). Durch das Aufteilen in ständig kleinere Datensätze, die wieder für sich sortiert werden, lässt sich BitonicSort sehr gut parallelisieren [122].

Die im Rahmen dieser Arbeit verwendeten Implementierungen der StreamIt-Benchmark-Bibliothek sortieren kontinuierlich Listen mit je 32 Elementen.

5 Verbindungsstrukturen für NoC-Architekturen

Als Verbindungsstruktur wird der Teil von NoC-Architekturen bezeichnet, der die Kommunikation innerhalb des Netzwerks selbst bereitstellt. Für NoC-Architekturen sind dies insbesondere die Router und deren Verbindungen untereinander, wie in Abbildung 5.1 farblich hervorgehoben.

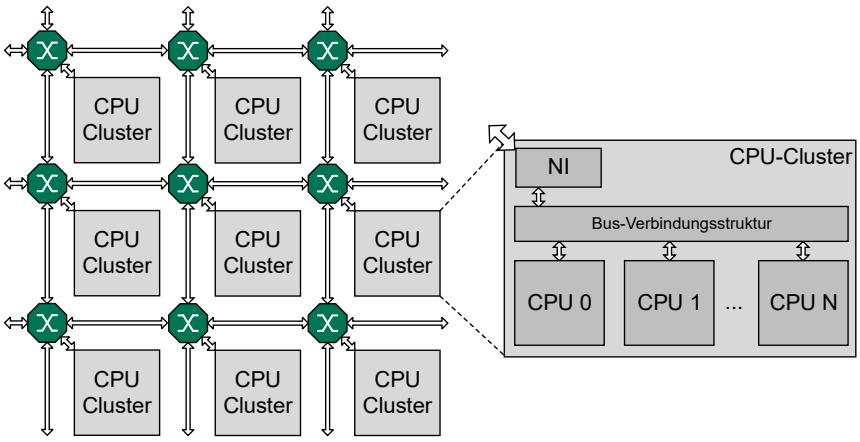


Abbildung 5.1: NoC-Verbindungsstruktur im CoreVA-MPSoC (farblich hervorgehoben)

Dieses Kapitel zeigt verschiedene Architekturkonzepte und Konfigurationen von NoC-Verbindungsstrukturen für eingebettete Multiprozessoren, wie das CoreVA-MPSoC. Thematisch lassen sich NoC-Verbindungsstrukturen in die drei Abschnitte Router (5.1), Topologie (5.2) und GALS[1]-Methoden (5.3) unterteilen. Um eine geeignete NoC-Verbindungsstruktur für eingebettete Many-Core-Systeme wie das CoreVA-MPSoC zu finden, wird in jedem Abschnitt Bezug zum Stand der Technik von NoC-Verbindungsstrukturen genommen. Dies ermöglicht eine Auswahl verschiedener Architekturkonzepte und Konfigurationen, die als NoC-Verbindungsstruktur für eingebettete MPSoCs geeignet sind. Darauf aufbauend wird schließlich die Implementierung und Umsetzung der

[1] Globally Asynchronous Locally Synchronous

geeigneten Konzepte für das CoreVA-MPSoC vorgestellt sowie eine Entwurfraumexploration verschiedener Implementierungsvarianten durchgeführt. Eine Bewertung unterschiedlicher Implementierungsvarianten wird anhand der Bewertungsmaße aus Kapitel 4 vorgenommen.

5.1 Router für eingebettete Multiprozessoren

Router sind die aktiven Kommunikationsknoten und damit die Basis-Komponente innerhalb einer NoC-Verbindungsstruktur. Sie sind für den fehlerfreien Transport von Paketen innerhalb des Netzwerks zuständig und setzen lokal das Routing der Netzwerkpakete und die entsprechende Flusskontrolle um (siehe Kapitel 2.1). Innerhalb dieses Abschnitts wird allein der synchrone Router betrachtet. Die verschiedenen GALS-Erweiterungen werden in Abschnitt 5.3 vorgestellt. Diese basieren bzgl. ihrer Funktionalität (Routing und Flusskontrolle) jedoch auf dem synchronen Router.

5.1.1 Stand der Technik

Als Switching-Methode werden in NoCs aus dem Stand der Technik (siehe Kapitel 2.2 und 2.3) zumeist das Circuit-Switching oder das Wormhole-Switching eingesetzt, deren Eigenschaften bereits in Kapitel 2.1.3 beschrieben sind. In MPSoCs mit Echtzeitanforderungen wird in der Regel das Circuit-Switching verwendet, da hierbei zyklengenaue Übertragungszeiten für bestimmte Datensätze garantiert werden können (GS-Kommunikation). Beispiele für NoCs mit Circuit-Switching sind das NoC des Æthereal [56], das Argo-NoC [91] und das NoC des STHORM [16]. Auf NoCs mit Wormhole-Switching wiederum setzen das Spidergon-STNoC [36], das GigaNetIC [127] und auch Kalray's MPPA [45]. Das Wormhole-Switching bietet zwar keine zyklengenauen Übertragungszeiten für Pakete, jedoch wird ein höherer durchschnittlicher Durchsatz erzielt (BE-Kommunikation), da NoC-Links besser ausgelastet werden können. Die meisten NoCs mit Wormhole-Switching bieten jedoch trotzdem eine gewisse GS-Kommunikation, indem Ressourcen für Bandbreite und eine Ende-zu-Ende-Flußkontrolle auf höheren Ebenen (Netzwerk-Schnittstelle, Software) verwaltet werden. Auch der Epiphany [2] bietet prinzipiell ein Wormhole-Routing, jedoch ist dieses nicht paketbasiert, da jedes Flit die direkte Zielspeicheradresse enthält und somit für sich betrachtet werden kann. Dies erhöht jedoch die Kontrolldaten innerhalb eines Flits, welches den realen Durchsatz von Nutzdaten minimieren kann (siehe Kapitel 6.1). Eine Kombination aus beiden Switching-Methoden bieten der Tomahawk2 [129] und der KiloCore [20], die jeweils ein NoC mit Circuit-Switching und ein weiteres NoC mit Wormhole-Switching auf einem Chip integriert haben. Im Vergleich zu allen anderen NoCs fällt der SpiNNaker mit S&F-Switching etwas heraus, welches jedoch durch die Verwendung von nur sehr kleinen Datenpaketen zu begründen ist.

Beim CoreVA-MPSoC kann trotz möglicher Echtzeitanforderungen auf ein Circuit-Switching verzichtet werden, da durch zusätzliche Methoden eine gewisse GS-Kommunikation geboten werden kann. So wird zum einen durch das Software-Kommunikationsmodell (siehe Kapitel 3.4.3) und der Netzwerk-Schnittstelle (siehe Kapitel 6.2) eine Ende-zu-Ende-Flußkontrolle garantiert. Zum anderen bietet auch der CoreVA-Compiler durch die Berücksichtigung von Latenzschranken eine Unterstützung bei Anwendungen mit Echtzeitanforderungen (siehe Kapitel 7.5). Aus diesem Grund eignet sich für das CoreVA-MPSoC ein Wormhole-Switching, da hierdurch bei moderatem Bedarf von Hardwareressourcen ein höherer durchschnittlicher Durchsatz erzielt werden kann als beim Circuit-Switching.

Eine geeignete Methode zur lokalen Flusskontrolle zwischen zwei Routern (siehe Kapitel 2.1.3) hängt stark von der im NoC verwendeten Switching-Methode und der allgemeinen Architektur des Routers ab. Bei NoCs mit Circuit-Switching gestaltet sich die lokale Flusskontrolle einfacher, da NoC-Pfade für die gesamte Paketübertragung reserviert sind. Asynchrone NoCs (z.B. Æthereal, Argo) nutzen die Methode des *ACK/NACK Flow Control*, um durch das hier verwendete Handshake-Verfahren direkt den korrekten Austausch von Daten ohne einen Takt sicherzustellen. Bei synchronen NoCs mit Wormhole-Switching können grundsätzlich alle drei Methoden zur Flusskontrolle eingesetzt werden. So wird im GigaNetIC ebenfalls die *ACK/NACK* Flusskontrolle eingesetzt, welche in Verbindung mit Wormhole-Switching jedoch einen zusätzlichen Puffer für Flits benötigt. Um ohne Flit-Verlust den maximalen Durchsatz von einem Flit pro Taktzyklus zu erreichen, müssen Flits nach ihrer Weiterleitung noch zusätzlich zwischengespeichert (dupliziert) werden, bis das Bestätigungssignal eintrifft. Um diese Problematik abzuschwächen wird im Spidergon-STNoC die Methode des *Credit-Based Flow Control* eingesetzt. Aufgrund des *Round Trip Delays* ist auch hier zusätzlicher Pufferplatz nötig, um Flits die bereits unterwegs sein können zu speichern. Flits müssen jedoch nicht dupliziert werden, welches die effektive Puffergröße erhöht. Da die *ON/OFF* Flusskontrolle trotz geringerem Signalaufwand die gleichen positiven Eigenschaften besitzt, wird diese im synchronen Router des CoreVA-MPSoCs zur lokalen Flusskontrolle eingesetzt.

Verschiedene Routingverfahren sind bereits in Abschnitt 2.1.2 aufgeführt. Wie in den meisten anderen als Chip realisierten MPSoCs (siehe Abschnitte 2.2 und 2.3) wird auch im CoreVA-MPSoC das klassische XY-Routing eingesetzt. Aufgrund der Einfachheit und gleichzeitigen Effizienz des XY-Routing ist dieses insbesondere in ressourceneffizienten MPSoCs weit verbreitet [29][51]. Gleichzeitig bietet XY-Routing durch das gedächtnislose und deterministische Verfahren eine ideale Voraussetzung für Echtzeitanwendungen und lässt sich zudem gut abschätzen.

Adaptive Routing-Algorithmen können zwar eine höhere Performanz aufweisen, haben jedoch in der Regel einen hohen Ressourcenverbrauch [139, Kapitel 12]. Hinzu kommt, dass diese aufgrund ihrer adaptiven Eigenschaften nicht deterministisch sind und sich somit nicht für die Ausführung echtzeitkritischer Anwendungen eignen.

Die Untersuchungen in Abschnitt 5.2.3 zeigen zudem, dass der Flaschenhals in der NoC-Kommunikation für die untersuchten Streaming-Anwendungen zumeist nicht im Router und auf den Links liegt, sondern an der Netzwerk-Schnittstelle zwischen Router und CPUs bzw. Softwareansteuerung (siehe Kapitel 6). Aus diesem Grund gilt es den Router des CoreVA-MPSoC so ressourceneffizient wie möglich zu gestalten, ohne die Performanz negativ zu beeinflussen.

5.1.2 Implementierung eines Routers für das CoreVA-MPSoC

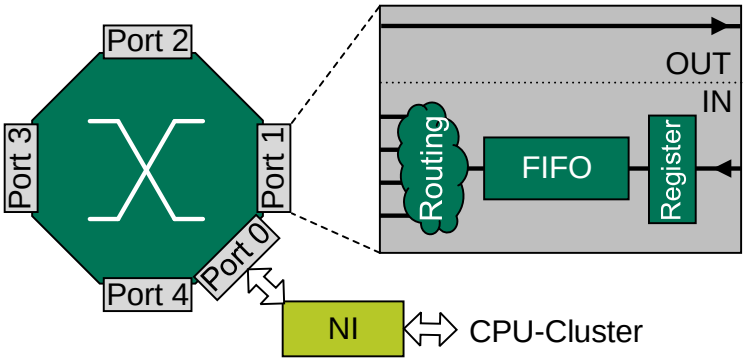


Abbildung 5.2: Aufbau des Routers im CoreVA-MPSoC für eine 2D-mesh Topologie

Der Router im CoreVA-MPSoC setzt sich aus mehrere Teilkomponenten zusammen. Abstrakt gesehen sind dies zunächst die *Input/Output-Ports* (*I/O-Port*) und eine Crossbar (siehe Abbildung 5.2).

Implementierung I/O-Port

Ein I/O-Port ist die Schnittstelle des Routers und befähigt die Verbindung mit den I/O-Ports weiterer Routingknoten. Die NoC-Links zwischen den Routern sind im CoreVA-MPSoC als bi-direktionale Verbindungen umgesetzt, die sich aus zwei unabhängigen uni-direktionalen Links in entgegengesetzter Richtung zusammensetzen. Durch eine generische Implementierung des Routers kann die Anzahl von I/O-Ports zur Entwurfszeit beliebig festgelegt werden. Dies ermöglicht im CoreVA-MPSoC die physikalische Umsetzung verschiedener Topologien, welche im nächsten Abschnitt vorgestellt werden. An jeweils einem I/O-Port ist die Netzwerk-Schnittstelle (siehe Kapitel 6) angebunden, welche als Schnittstelle zum jeweils lokalen CPU-Cluster fungiert.

Innerhalb des I/O-Ports werden die Entscheidungen getroffen was mit ankommenden Paketen passiert. Dies ist neben der Entscheidung über die Weiterleitung des Pakets auch die Art und Weise wie Pakete zwischengespeichert werden. Wie im vorherigen

Abschnitt beschrieben, setzt der Router im CoreVA-MPSoC ein Wormhole-Switching um. Da Pakete bereits in der Netzwerkschnittstelle in mehrere Flits segmentiert werden, kann der Router jedes ankommende Flit einzeln verarbeiten. Der Aufbau eines solchen Flits ist in Abbildung 5.3 zu sehen. Standardmäßig besteht im CoreVA-MPSoC ein Flit aus einem 26-bit-Flitkopf und einem 64-bit-Flitrumpf für Nutzdaten. Im I/O-Port werden die Routinginformationen des Flits (XY-Koordinate) ausgewertet. Alle anderen Informationen des Flitkopfs dienen der übergeordneten Flusskontrolle, die von der Netzwerk-Schnittstelle ausgewertet werden.

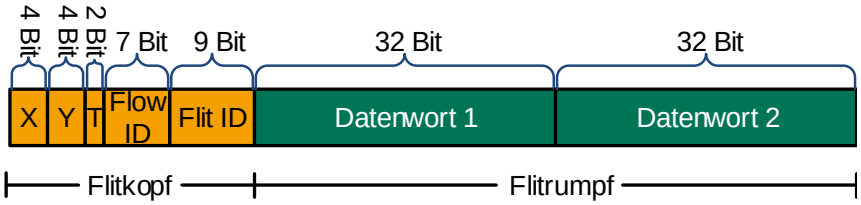


Abbildung 5.3: Beispielhafter Aufbau eines Flit im CoreVA-MPSoC

Der innere Aufbau des I/O-Ports ist im rechten Teil von Abbildung 5.2 zu sehen. Flits, die einen Router verlassen, werden vom I/O-Port direkt weitergeleitet (OUT). Die eigentliche Logik für Routing, Flusskontrolle und Zwischenspeicherung wird im I/O-Port für ankommende Flits angewendet. Am Eingang des I/O-Port werden die ankommenden Flits zunächst in einem Register zwischengespeichert. Diese Registrierung ist notwendig, um den kritischen Pfad der Schaltung vom vorherigen Router zu entkoppeln.

Anschließend findet die eigentliche Zwischenspeicherung von Flits in einem FIFO statt. Zum einen dient das FIFO zur Pufferung von Flits, die nicht direkt weitergeleitet werden können. Dies kann auftreten, wenn mehrere Eingangsports um denselben Ausgangsport konkurrieren. Zum anderen dient das FIFO der lokalen Flusskontrolle (vgl. Kapitel 2.1.3). Im CoreVA-MPSoC wird hierzu der *ON/OFF Flow Control* verwendet, hierbei wird dem sendenden Router auf einer 1-bit-Leitung (ON=1, OFF=0) mitgeteilt, ob noch Flits angenommen werden können oder nicht. Auch diese ON/OFF-Leitung wird aufgrund des kritischen Pfades vor Verlassen des I/O-Ports registriert. Dies führt zusammen mit der Registrierung der Flits am Eingang dazu, dass bereits zwei weitere Flits unterwegs sein können, bevor das OFF-Signal am sendenden Router ankommt. Hierzu muss das FIFO im Eingangsport immer Platz für zwei zusätzliche Flits haben nachdem das OFF-Signal gesetzt wird. Als FIFO im CoreVA-MPSoC wird dazu ein IP-Block aus der DesignWare-Bibliothek von Synopsys [174] mit sogenannter Almost-Full-Funktionalität eingesetzt [44]. Das negierte Almost-Full-Signal entspricht dabei dem OFF-Signal und wird aktiv, sobald nur noch zwei freie Einträge im FIFO vorhanden sind. Insgesamt benötigt das FIFO den Platz für mindestens vier Flits, da auch eine interne Verzögerung zur Erzeugung des Almost-Full-Signals entsteht. Nur so kann im Idealfall – wenn keine Überlastung am Ausgangsport besteht – der maximale Durchsatz von

einem Flit pro Taktzyklus erreicht werden. Ist das FIFO leer, kann ein ankommendes Flit dieses direkt im nächsten Taktzyklus verlassen, sodass die minimale Latenz eines Flits nicht durch die Größe des FIFOs ansteigt. Durch das Register am Eingang und dem nachgeschalteten FIFO ergibt sich für ein Flit im Idealfall eine Latenz von zwei Taktzyklen pro Router.

Für Flits, die am Ausgang des FIFOs anliegen, kann die Routingentscheidung getroffen werden, zu welchem Ausgangsport das Flit transferiert werden soll. Zur Adresszuordnung der CPU-Cluster werden im CoreVA-MPSoC XY-Koordinaten und ein dazu passendes XY-Routing verwendet. Sowohl die Routingentscheidung als auch die Modifikation des Flitkopfes stehen in Abhängigkeit zum verwendeten Routingverfahren, welches auch von der Topologie abhängt (siehe Abschnitt 5.2). Für die klassische 2D-Mesh-Topologie bedeutet dies, dass die Pakete zuerst horizontal und anschließend vertikal durchs NoC weitergeleitet. Zur einfacheren Logikauswertung innerhalb der Router werden relative Koordinaten verwendet. Jeder Router verringert demnach eine Koordinate nach Verlassen des Routers. Ein Vergleich gegen Null signalisiert daher ein Eintreffen im Zielknoten. Zur Richtungsbestimmung in der jeweiligen Koordinate ist jeweils das höchste Bit reserviert. Die in Abbildung 5.3 beispielhaft verwendeten 4 bit erlauben daher maximal eine 8x8-Mesh-Topologie. Je nachdem welchen weiteren Weg das Flit einschlagen soll, wird anschließend bereits die X- oder Y-Koordinate innerhalb des Flitkopfes dekrementiert.

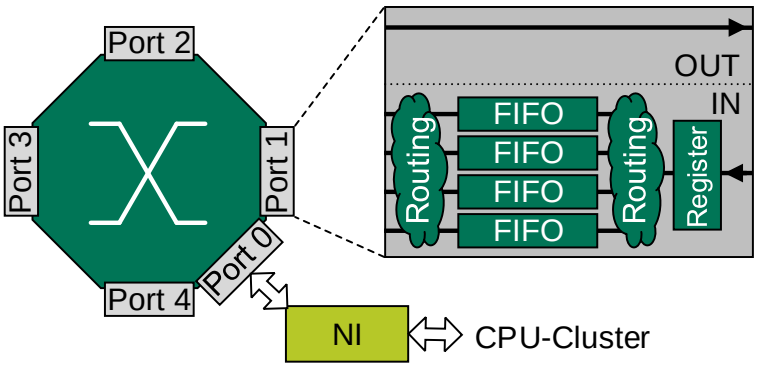


Abbildung 5.4: Aufbau des Routers im CoreVA-MPSoC mit virtuellen Kanälen

Zur Entwurfszeit besteht die Möglichkeit den Router des CoreVA-MPSoCs mit virtuellen Kanälen zu konfigurieren. Der Aufbau des Routers mit virtuellen Kanälen ist in Abbildung 5.4 zu sehen. Die Verwendung von virtuellen Kanälen zur Prioritätssteuerung von bestimmten Paketen ist in der aktuellen Implementierung des CoreVA-MPSoCs nicht vorgesehen, da alle Pakete mit der gleichen Priorität behandelt werden. Dennoch bieten virtuelle Kanäle die Möglichkeit, den durchschnittlichen Durchsatz im NoC zu erhöhen. Ohne virtuelle Kanäle besteht die Gefahr, dass Flits eine längere Zeit im FIFO

gehalten werden müssen als nötig. Liegt am Ausgang des FIFOs ein Flit an, welches über einen zur Zeit blockierenden I/O-Port versendet werden soll, können auch alle weiteren Flits im FIFO nicht gesendet werden, obwohl diese eventuell für einen anderen freien I/O-Port bestimmt sind. Um dieses Problem zu vermeiden, wird die Anzahl der FIFOs im I/O-Port erhöht. In jedem I/O-Port werden nun exklusive FIFOs für jeden weiteren Ausgangsport am Router verwendet. Dies ermöglicht es, ankommende Flits direkt in das passende FIFO zu sortieren, sodass sich in jedem FIFO nur Flits für einen bestimmten Ausgangsport befinden. Aus diesem Grund kann ein physikalischer Kanal zwischen zwei Routern trotz Blockade verwendet werden, solange Flits im weiteren Verlauf unterschiedliche Wege nehmen. Somit beinhaltet ein physikalischer Kanal mehrere virtuelle Kanäle. Für eine effiziente Flusskontrolle besitzt nun jeder virtuelle Kanal eine separate ON/OFF-Leitung. Hierbei ist bereits vor den FIFOs eine Routingentscheidung zu treffen, damit das Flit in das passende FIFO eingereiht wird. Zusätzlich ist nach Durchlauf der FIFOs eine weitere Routing-Entscheidung zu treffen. Hier muss bereits die Routingentscheidung ermittelt werden, die im nächsten Router angewendet wird. Dies ist nötig, um die ON/OFF-Leitung des richtigen virtuellen Kanals des nächsten Routers auszuwerten. Würde dies nicht geschehen, würde der Vorteil von virtuellen Kanälen bereits im nächsten Router verpuffen. Der Nachteil von virtuellen Kanälen ist der erhöhte Flächenbedarf, der sich durch die zusätzlichen FIFOs ergibt.

Implementierung Crossbar

Bei der Verbindungsart zwischen den I/O-Ports innerhalb des Routers sind grundsätzlich verschiedene Lösungen wie Bussysteme oder Punkt-zu-Punkt-Verbindungen denkbar. Ein Bus hat einen vergleichsweise geringen Verdrahtungsaufwand, doch kann nur ein I/O-Port gleichzeitig Daten übertragen. Punkt-zu-Punkt-Verbindungen bieten zwar eine absolute Blockierungsfreiheit, sind im Router streng genommen jedoch nicht umsetzbar, da an jedem Ausgangsport mehrere Anfragen parallel verarbeitet werden müssten. Da der Ausgangsport jedoch maximal ein Flit pro Taktzyklus versenden kann, bietet sich als Verbindungsart eine Crossbar an, welche auch als Koppelfeld (früher Kreuzschienenverteiler) bezeichnet wird. Hierbei werden die Signale eines Eingangsports vollkommen transparent auf die eines Ausgangsports geschaltet. Am Ausgangsport findet nun eine Arbitrierung durch eine Prioritätssteuerung statt, bei der jeweils ein anfragender I/O-Port den Vorzug erhält (siehe Abbildung 5.5). Im Vergleich zum Bus hat dies den Vorteil, dass gewisse Pfade im Router ohne Blockierung durchgeschaltet werden können. Beispielsweise können Flits von Port 1 zu Port 2 parallel mit Flits von Port 3 zu Port 4 übertragen werden.

Sowohl für die Bus-, als auch die Crossbar-Implementierung können verschiedene Verfahren zur Arbitrierung eingesetzt werden. Verschiedene Arbitrierungsverfahren werden bereits in den Arbeiten [127] und [162] betrachtet. Die im Router des CoreVA-MPSoC eingesetzte Crossbar basiert auf der Crossbar des GigaNetIC [127], in dem ein Rundlauf-Verfahren (*Round-Robin*) eingesetzt wird. Der *Round-Robin-Arbiter* sorgt

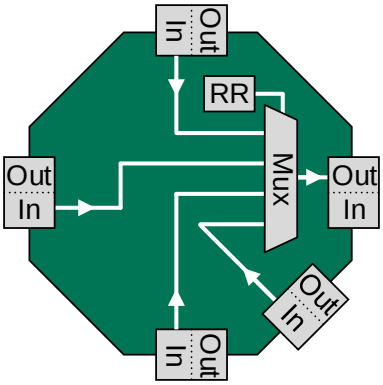


Abbildung 5.5: Aufbau der Crossbar im CoreVA-MPSoC am Beispiel für Ausgangsport 1, mit Round-Robin-Arbitrierung (RR).

dafür, dass der Reihe nach jeder Eingangsport die höchste Priorität erhält und ein Flit weitergeleitet werden kann. Dies vermeidet das „Verhungern“ von Flits mancher Eingangsports und begrenzt die maximale Wartezeit eines Flits auf vier Taktzyklen bei einem Router mit fünf I/O-Ports. Alle I/O-Ports verfügen daher über die gleiche Priorität. Es konkurrieren allerdings immer nur die I/O-Ports miteinander, die auch tatsächlich Anfragen an einen Ausgangsport stellen. Stellt bis auf ein Eingangsport kein anderer Anfragen, erhält dieser ununterbrochen Zugriff auf den Ausgangsport.

5.2 Topologien für eingebettete Multiprozessoren

Mehrere Router können zu einer globalen Topologie im Many-Core-System zusammengeschaltet werden. Durch die Verwendung von CPU-Clustern ist die Topologie des CoreVA-MPSoCs in jedem Fall bereits eine hierarchische Topologie. Im Folgenden steht jedoch allein die Topologie des NoCs im Vordergrund. Hierzu wird die Topologie des NoCs separat betrachtet und ein gesamtes CPU-Cluster als Endknoten angesehen.

5.2.1 Stand der Technik

Abstrakt sind die Eigenschaften unterschiedlicher Topologien bereits in Kapitel 2.1.1 bzw. Tabelle 2.1 diskutiert. Um einen besseren Eindruck über das Verhalten der Topologien zu erlangen, ist im Folgenden ein explizites Szenario eines Multiprozessorsystemes mit 64 CPU-Clustern gegeben. Für die standardmäßig genutzte 2D-Mesh-Topologie des CoreVA-MPSoC entspricht dies den Parametern $m = 1$, $k = 4$, $n = 2$ und der daraus folgenden Form: *4-ary 2-mesh*. In Tabelle 5.1 sind die Eigenschaften (vgl. Abschnitt 2.1.1)

aller betrachteten Topologien aufgeführt. Die Angabe von 64 CPU-Clustern beinhaltet die Konfiguration von maximal einem CPU-Cluster ($m = 1$) pro Router. Genau diese Konfiguration ist allerdings nicht für alle Topologien realisierbar, sodass durchgängig eine Vergleichsbasis mit der Bedingung der abstrakt rechtwinkligen Darstellung existiert. Aufgrund dessen sind Honeycomb-Mesh- und Honeycomb-Torus-Topologien in den Tabellen mit 72 CPU-Cluster aufgeführt. Topologien, die auf einer Baumstruktur basieren, sind logisch mit einem CPU-Cluster pro Router nicht realisierbar. Die Tabellenergebnisse der untersten drei Topologien werden daher mit zwei CPU-Cluster ($m = 2$) Konfiguration errechnet.

Topologie	CPU-Cluster	Router	Kantengrad	Symmetrie	Homogenität	Bisektionsbandbreite
64-ary 1-torus	64	64	3	✓	✓	4
8-ary 2-mesh	64	64	5	✗	✗	16
8-ary 2-torus	64	64	5	✓	✓	32
Honeyc. Mesh	72	72	4	✗	✗	$\sim 13,92$
Honeyc. Torus	72	72	4	✓	✓	$\sim 34,62$
2-ary 6-tree	64	192	4	✗	✗	64
2-ary 6-fly	64	192	2	✗	✓	32
2-ary 6-flat	64	32	7	✓	✓	32

Topologie	Hop-Anzahl	durchschnittl. Hop-Anzahl	Konnektivität	Netzwerkverbindungen
64-ary 1-cube	33	$21,\overline{3}$	2	128
8-ary 2-mesh	15	$5,\overline{3}$	2	224
8-ary 2-cube	9	4	4	256
Honeyc. Mesh	~13,86	N/A	2	~192
Honeyc. Torus	~7,93	4,58	3	~240
2-ary 6-tree	11	7,58	2	640
2-ary 6-fly	6	6	1	320
2-ary 6-flat	6	N/A	5	160

Tabelle 5.1: Konkrete Topologieeigenschaften für ein MPSoC mit 64 beziehungsweise 72 CPU-Cluster

In Tabelle 5.1 sind die Vor- und Nachteile der Topologien zu sehen. Bei den meisten Eigenschaften unterscheiden sich die anderen Topologien zur klassischen 2D-Mesh-Topologie. Da das CoreVA-MPSoC den Aspekt der Ressourceneffizienz verfolgt und die optimale Flächenausnutzung dabei mit einhergeht, sind durch den geringeren Kantengrad Topologien wie das Honeycomb interessante Alternativen für das NoC. Die auf Baumstrukturen basierenden Topologien wirken sich hingegen in großen MPSoCs

nachteilig aus. Ihre benötigte Fläche ist durch die große Anzahl an Routern und Verbindungen in der Regel sehr hoch. Eine Ausnahme bietet der Flattened Butterfly (2-ary 6-fly), eine Weiterentwicklung der Butterfly-Topologie. Diese verfügt über ein kosteneffizientes indirektes Netzwerk, welches den Flächenbedarf stark reduziert. Hierbei werden die Knoten eines Butterfly-Netzwerks in jeder Spalte abgeflacht beziehungsweise zusammengeführt und die gleichen Verbindungen beibehalten [93]. Negativ kann sich hier jedoch die Länge von einzelnen Verbindungen auswirken, welches zu Schwierigkeiten beim Platzieren und Verdrahten (P&R, siehe Kapitel 3.3) führen kann. Da die Länge auch von der Größe der Endknoten (CPU-Cluster) bzw. vom spezifischen Design abhängt, kann diese Eigenschaft jedoch nicht in der Tabelle berücksichtigt werden. Gleiches gilt für Torus-Topologien, bei denen zumeist lange Verbindungen an den Topologie-Grenzen vorliegen. Beim 2D-Torus besteht zwar die Möglichkeit, dieses durch eine gefaltete (Folded-Torus) Darstellung abzuschwächen (siehe Abbildung 5.6b), jedoch sind viele Verbindungen zwischen den Routern auch hier noch doppelt so lang wie bei der klassischen 2D-Mesh-Topologie.

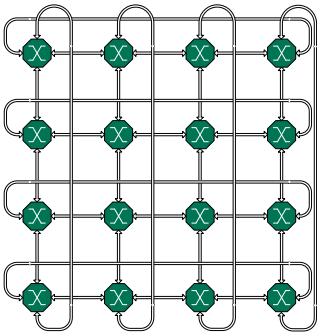
Betrachtet man die Realisierung anderer MPSoC-Chips aus dem Stand der Technik (siehe Kapitel 2.2 und 2.3) zeigt sich, dass zumeist die klassische 2D-Mesh-Topologie verwendet wird. Ausnahmen bilden Kalray's MPPA, der die Folded-Torus-Topologie umsetzt sowie der SpiNNaker, welcher auf einem einzelnen Chip eine Art Sternentopologie verwendet, da alle CPUs an einem einzigen Router angebunden sind.

In Zukunft können auch dreidimensionale Topologien eine effiziente Alternative bieten. Dies ist mehr eine Frage der Technologie als die der Architektur. Aktuelle Verfahren des 3D-Stacking, bei denen mehrere Chips aufeinander gestapelt werden [131][153], sind noch sehr aufwendig und würden erst bei sehr großen MPSoCs einen Vorteil bieten.

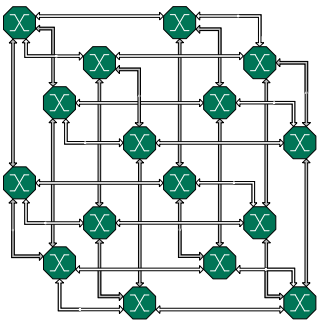
5.2.2 Implementierung verschiedener Topologien im CoreVA-MPSoC

Beim CoreVA-MPSoC ist die klassische 2D-Mesh-Topologie die Standardkonfiguration des Routers und ist in Abschnitt 5.1 beschrieben. Obwohl andere Topologien zum Teil effizientere Eigenschaften besitzen, bleibt ein 2D-Mesh bei der physikalischen Implementierung eine der praktikabelsten. Die rechteckige Anordnung eines 2D-Mesh bietet auf Siliziumchips eine optimale Flächennutzung. Dies erleichtert insbesondere das ohnehin schon sehr aufwendige Erstellen des Layouts (P&R) und verkürzt damit die Entwicklungszeit des MPSoCs.

Das 2D-Mesh wird beim CoreVA-MPSoC als zweidimensionales Koordinatensystem interpretiert, indem jeder Router mit seinem CPU-Cluster durch eindeutige Koordinaten mit $X, Y \in \mathrm{N}_0$ adressiert ist. Dabei nummeriert X die Router in horizontaler Richtung und Y die Router in vertikaler Richtung. Im CoreVA-MPSoC basiert sowohl die Software-Werkzeugkette (siehe Abschnitt 3.4) als auch die Netzwerkschnittstelle (siehe Abschnitt 6) auf diesen X-, Y-Koordinaten. Um diese Komponenten nicht für jede



(a) 2D-Torus (4-ary 2-torus)



(b) Folded 2D-Torus (4-ary 2-torus)

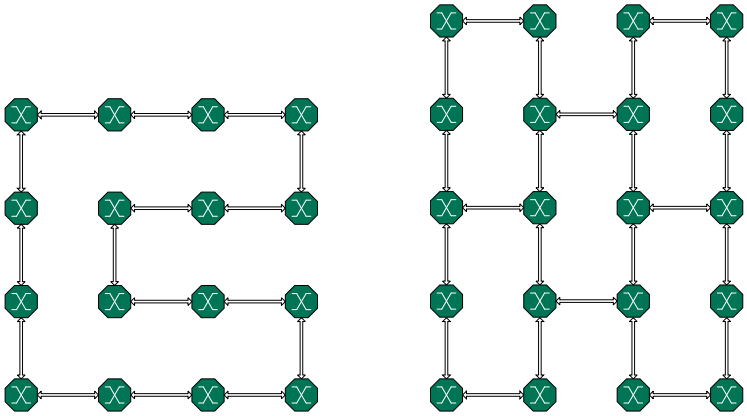
Abbildung 5.6: 2D-Torus-Topologie im CoreVA-MPSoC

Topologie neu entwickeln zu müssen, wird die Vorgehensweise mit diesen Koordinaten auch bei der Umsetzung der im Folgenden vorgestellten Topologien berücksichtigt. Hierzu werden alle Topologien abstrakt als rechteckige Struktur interpretiert, welches je nach Topologie auch Einfluss auf das eingesetzte XY-Routing hat.

Insgesamt ist die Hardwarebeschreibung des Routers für das CoreVA-MPSoC so erweitert, dass zur Entwurfszeit vier verschiedene Topologien konfiguriert werden können. Neben dem klassischen 2D-Mesh sind das der 2D-Torus sowie eine Ring- und Honeycomb-Topologie.

Die 2D-Torus-Topologie ist die Erweiterung des 2D-Mesh und wird abstrakt mit k-ary n-torus beschrieben. Durch die bereits vorhandene regelmäßige und rechteckige Struktur ist eine abstrakte Anpassung für das XY-Koordinatensystem nicht notwendig (Abbildung 5.6a). Eine Anpassung der Verbindungsstruktur erfolgt hauptsächlich an den äußeren Bereichen, denn diese werden miteinander verbunden. Der Routing-Block des Routers (Abbildung 5.2) enthält die Logik für die Wegwahl des Flits im Netzwerk. Wie beim Routing im 2D-Mesh werden X- und Y-Koordinate getrennt betrachtet und die X-Richtung priorisiert. Das Ziel in dieser Routing-Logik ist die Ermittlung des kürzesten Pfades und bedeutet hierbei die Entscheidung zwischen rechten oder linken Ausgangsport. Durch die neuen Verbindungen des 2D-Torus im Vergleich zum 2D-Mesh, können sich hier andere kürzeste Pfade ergeben. Zur Ermittlung des kürzesten Pfads muss daher in jedem Router die Gesamtgröße des NoCs bekannt sein, um Flits ggf. über Pfade mit den äußeren Verbindungen zu schicken.

Ein signifikanter Bestandteil der Torus-Topologie sind die langen Außenverbindungen, die eine physikalische Umsetzung erschweren. Eine Anordnung auf dem Chip wie in Abbildung 5.6a lässt sich aufgrund von Leitungsverzögerungen nur schwer implementieren. Eine Alternative bietet eine gefaltete Anordnung wie in Abbildung 5.6b.



(a) Ring (8-ary 1-torus) (b) Honeycomb (Honeyc.-mesh)

Abbildung 5.7: Ring- und Honeycomb-Topologie im CoreVA-MPSoC

Diese Anordnung ist logisch gesehen identisch, bietet jedoch den Vorteil, dass alle Leitungen vergleichsweise kurz sind. Dennoch stellen die komplexeren Verbindungen und die immer noch doppelt so langen Leitungen im Vergleich zum 2D-Mesh, eine hohe Herausforderung an die Erstellung des physikalischen Designs.

Die Ring-Topologie ist genau genommen eine Sonderform des 2D-Torus und wird abstrakt als *k-ary 1-torus* bezeichnet. Um auch beim der Ring-Topologie das XY-Koordinatensystem zu verwenden, wird bei der Verbindungsstruktur auf eine abstrakte Anordnung der Knoten wie in Abbildung 5.7a zurückgegriffen. Für das Routing wird eine eindeutige globale ID per Knoten verwendet. Diese wird vor Übergabe eines Flits an den Router aus der abstrakten X- und Y-Koordinate von Quell- und Zielknoten und mit Hilfe der Gesamtgröße des NoCs bestimmt. Abhängig davon in welcher Richtung (links- oder rechtsseitig) der kürzere Pfad liegt, wird lediglich ein Informationsbit gesetzt und dem Flit angehängt. Die benötigte Fläche der Routing-Logik im Router ist in dieser Variante minimal. Es wird lediglich auf ein Bit verglichen und entschieden, welcher Ausgangsport damit gewählt wird. Der Vorteil hierbei ist der minimale Flächenverbrauch an jedem Eingangsport des Routers durch die einmalige Berechnung des Informationsbits am Quellknoten. Zu berücksichtigen ist jedoch dort der zusätzliche Logikaufwand in der Netzwerkschnittstelle oder Softwareaufwand auf der CPU.

Die Honeycomb-Mesh-Topologie ist eine der vielversprechenden Topologien, die aus der Theorie hervorgeht. Da eine Honigwabe von Natur aus keine rechteckige Form vorweist, wird im ersten Schritt die Form abstrakt angepasst. Die rechteckige Darstellung in Abbildung 5.7b ermöglicht die Verwendung des XY-Koordinatensystems. Anders als bei den meisten geordneten Topologien gibt die Honeycomb-Mesh-Topologie

von sich aus Einschränkungen vor. Hierbei müssen vollständige Hexagone eingehalten werden, wodurch für die Anzahl Knoten in einer Richtung nur gerade Werte und für die andere nur ungerade Werte eingesetzt werden dürfen. Die vertikalen Verbindungen erfolgen nach dem gleichen Schema wie im 2D-Mesh des CoreVA-MPSoC.

Das Routing basiert auf dem Vergleich der absoluten Koordinaten und nicht auf relativen, wie bei den anderen implementierten Topologien. In diesem Fall wird in der Vergleichslogik zusätzlich auf die einzelne Verbindung in horizontaler Ebene überprüft. Ob eine Verbindung in entsprechender Richtung vorliegt, wird durch die Abfrage des niedrigsten Bits der Router-Position realisiert. Diese Strategie ist dabei eine sehr kostengünstige. Weiterhin wird horizontal geroutet, wenn eine Kante vorhanden ist. Falls nicht, wird stattdessen vertikal geroutet in Abhängigkeit der Position des Zielknotens. Ist die passende X-Koordinate erreicht, folgt das vertikale Routing, sofern notwendig. Somit wird mit möglichst wenigen Vergleichen der kürzeste Pfad detektiert.

5.2.3 Vergleich verschiedener Topologien im CoreVA-MPSoC

In diesem Abschnitt wird eine Entwurfsraumexploration über den Ressourcenbedarf der verschiedenen Topologien durchgeführt, die für das CoreVA-MPSoC implementiert sind. Teile der hier vorgestellten Ergebnisse sind im Rahmen der betreuten Bachelorarbeit [219] entstanden.

Flächenbedarf

In Abbildung 5.8 wird der Flächenbedarf der verschiedenen Router-Implementierungen für die Topologien 2D-Mesh, 2D-Torus, Honeycomb-Mesh und die Ring-Topologie dargestellt. Angegeben ist der Flächenbedarf dabei jeweils für einen Router, der in einem MPSoC aus mehreren CPUs synthetisiert wurde. Um den Fokus auf das NoC bzw. die Topologie zu legen und den Rechenaufwand für die Synthesen zu verringern, wurden in den synthetisierten MPSoCs nur CPU-Cluster mit einem einzelnen CPU-Makro (siehe Abbildung 3.6b) verwendet. Der Flächenbedarf eines NI (0,02 mm^2) und eines CPU-Cluster (0,15 mm^2) ist bei allen Synthesen konstant.

Der Flächenbedarf der Router variiert zwischen 0,012 mm^2 (Ring) und 0,023 mm^2 (Torus) und ist in erster Linie durch die unterschiedliche Anzahl I/O-Ports zu begründen, da ein I/O-Port in jeder Topologie immer etwa eine Chipfläche 0,0038 mm^2 aufweist. Die Anzahl I/O-Ports wirkt sich zusätzlich auf die Größe der Crossbar innerhalb des Routers aus. Zwischen Torus und klassischem 2D-Mesh beträgt der Unterschied der Router-Fläche lediglich 1,88%, da in beiden Topologien fünf I/O-Ports pro Router nötig sind. Das Honeycomb-Mesh liegt mit vier I/O-Ports pro Router mit einer Fläche von 0,017 mm^2 genau zwischen 2D-Mesh und der Ring-Topologie (3 I/O-Ports).

Angegeben ist der Flächenbedarf in Abbildung 5.8 für einen „inneren“ Router mit der maximalen Anzahl von I/O-Ports. Die Router am Rand der Topologien 2D-Mesh und

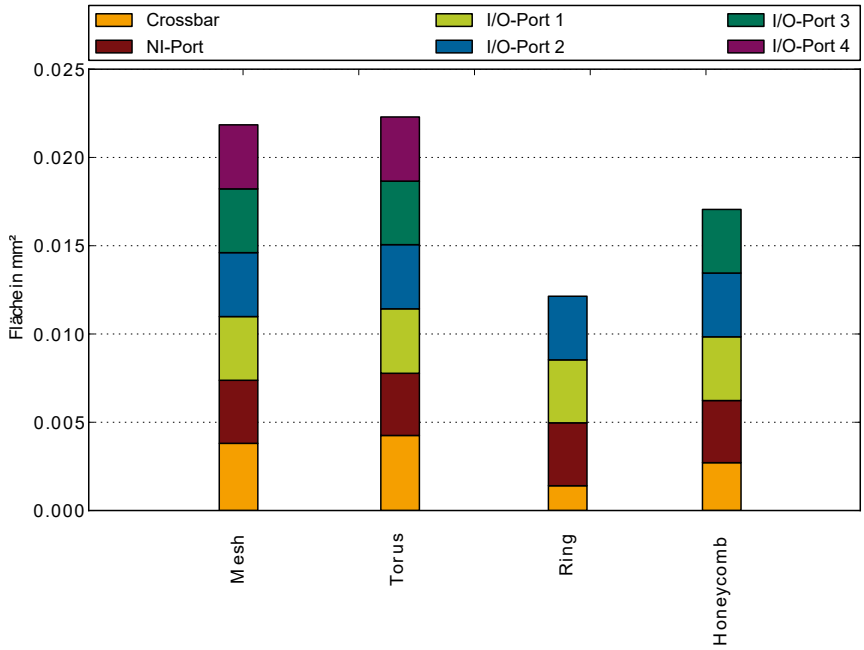


Abbildung 5.8: Flächenbedarf eines Routers für verschiedene Topologien im CoreVA-MPSoC

Honeycomb-Mesh benötigen jedoch weniger Ports und haben einen entsprechend geringeren Flächenbedarf. Der Anteil von Routern am Rand verringert sich bei zunehmender Skalierung des NoCs.

Da Topologien nicht der Hauptfokus dieser Arbeit sind, wird auf die sehr zeitaufwändige Implementierung von platzierten und verdrahteten MPSoC-Layouts mit allen verschiedenen Topologien verzichtet. Prototypisch wird hierzu nur das 2D-Mesh implementiert (siehe Abschnitt 5.3.4 und Kapitel 8.1). Die in dieser Arbeit vorgestellten Ergebnisse dienen daher nur als Abschätzung, da sich die tatsächliche Leitungslänge der Links zwischen den Routern erst durch ein platziertes und verdrahtetes Layout ergibt. Dies kann insbesondere beim Torus dazu führen, dass zusätzliche Treiber oder gar Registerstufen auf den Links nötig sind, welches sowohl Flächenbedarf als auch die Leistungsaufnahme steigert.

Leistungsaufnahme

Die in Abbildung 5.4 dargestellte Leistungsaufnahme wurde ebenfalls mit Hilfe den im vorherigen Abschnitt vorgestellten Synthesen ermittelt. Es handelt sich dabei um eine

Abschätzung der Synthese-Werkzeuge, bei der die Schaltaktivität an den I/O-Ports auf 10% eingestellt ist.

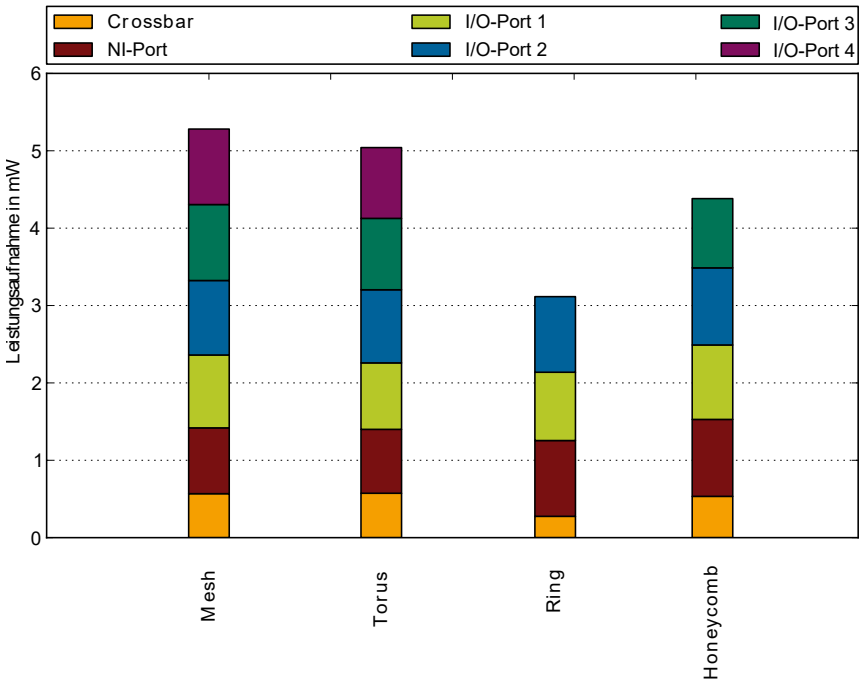


Abbildung 5.9: Leistungsaufnahme eines Routers für verschiedene Topologien im CoreVA-MPSoC

Es zeigt sich, dass die Leistungsaufnahme beim Vergleich der Topologien ähnliche Tendenzen aufzeigt wie beim Flächenbedarf. Auch hier haben Ring- und Honeycomb-Topologie eine insgesamt geringere Leistungsaufnahme pro Router als 2D-Mesh und Torus, was auch hier durch die Anzahl I/O-Ports zu begründen ist. Die Router im 2D-Mesh (5,3 mW) und Torus (5,1 mW) haben etwa die gleiche Leistungsaufnahme. Der Router im Honeycomb hat mit 4,4 mW eine um 17% und die Ring-Topologie (3,1 mW) eine um 41,5% geringere Leistungsaufnahme als das 2D-Mesh.

Performanz

In der Theorie sind durch die unterschiedlichen Eigenschaften der Topologien (siehe Tabelle 5.1) auch Unterschiede für den Durchsatz und die Latenz zu erwarten. Für den Verkehr von NoC-Paketen trifft dies auch zu, da Pakete z.B. in der Ring-Topologie weniger Links zur Verfügung haben, welches sowohl den Durchsatz verringert als auch

die Latenz erhöht. Zahlreiche Untersuchung mit Hilfe von Paketgeneratoren, die anstelle von CPUs direkt an das NoC angebunden werden, haben dies bestätigt [100][138][168].

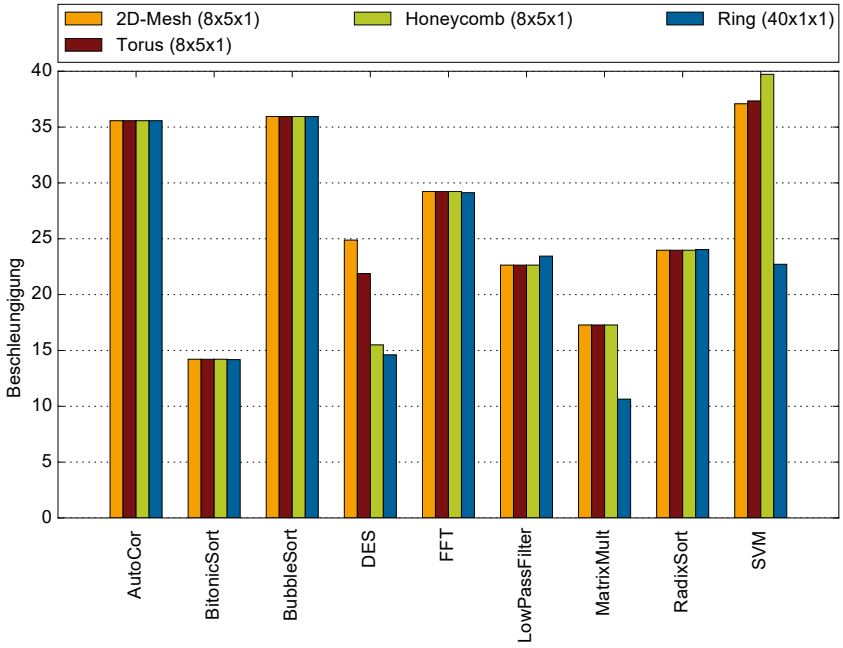


Abbildung 5.10: Beschleunigung von 40 CPUs gegenüber einer CPU, bei Verwendung verschiedener Topologien im CoreVA-MPSoC

Die Ergebnisse in diesem Abschnitt zeigen das Verhalten der verschiedenen Topologien im CoreVA-MPSoC. In Abbildung 5.10 ist die Beschleunigung von Streaming-Anwendungen (siehe Abschnitt 4.2.3) für die Topologien 2D-Mesh, 2D-Torus, Honeycomb und Ring im Vergleich zu einem Einprozessorsystem zu sehen. Die Konfiguration des CoreVA-MPSoC unterscheidet sich dabei nur durch die entsprechende Toplogogie. Alle anderen Systemeigenschaften, wie z.B. die Anzahl CPUs und Speicher, sind bei allen Toplogien gleich. Simuliert wurde ein CoreVA-MPSoC mit je 40 CPUs, da sich bei dieser Anzahl erste Unterschiede zwischen den Topologien darstellen. Außerdem lassen sich bei dieser Anzahl von CPUs alle Topologien vollständig abbilden. Um den Fokus auf die Topologie des NoC zu legen, wird pro CPU-Cluster nur eine einzelne CPU verwendet. Die Simulationen werden mit Hilfe des zyklengenauen Multiprozessor-Simulators (siehe Abschnitt 3.4.5) durchgeführt.

Unterschiede in der Performanz sind bei den Anwendungen DES, MatrixMult und SVM zu erkennen. Bei MatrixMult und SVM zeigt allein der Ring eine um 38,2% (MatrixMult) bzw. 37,8% (SVM) geringere Beschleunigung im Vergleich zum 2D-Mesh. Die Topologie Honeycomb stellt sich nur bei der Anwendung DES als nachteilig heraus und verzeichnet Performanz-Einbußen von 39,2%. Auch beim DES zeigt der Ring eine schlechtere Performanz. Die Unterschiede zwischen 2D-Torus und 2D-Mesh sind minimal und im Rahmen der Schwankungen des CoreVA-MPSoC-Compilers durch das Simulated-Annealing zu erklären (siehe Abschnitt 3.4.4).

Bei allen anderen Anwendungen sind die Unterschiede zwischen den Topologien sehr gering bzw. gar nicht vorhanden. In diesen Fällen befindet sich der Engpass des Systems nicht in der Kommunikation des NoC. Durch das Kommunikationsmodell (siehe Abschnitt 3.4.3) finden die NoC-Kommunikation und das Rechnen auf den CPUs parallel statt. Zumeist sind in diesen Anwendungen daher eine oder mehrere CPUs der Engpass des Systems, da Anwendungsteile nicht weiter parallelisiert werden können. Bei vielen StreamIt-Anwendungen ist z.B. der Eingangsfilter der begrenzende Faktor, da diese den ankommenden Datenstrom verteilen muss. Weiterführende Optimierungen am CoreVA-MPSoC-Compiler und der I/O-Schnittstelle des CoreVA-MPSoC können solche Einflüsse in Zukunft jedoch abschwächen.

5.3 GALS-Erweiterungen für eingebettete Multiprozessoren

Bei sehr großen mikroelektronischen Schaltungen führt eine vollständig synchrone Umsetzung häufig zu einem höheren Flächen- und Energiebedarf sowie einer geringeren Performanz [143]. Wie bereits in Abschnitt 2.1.4 aufgeführt, ist es notwendig große Schaltungen als GALS-Systeme zu entwerfen. In MPSoCs ist es daher äußerst verbreitet, die vielen synchronen CPUs über eine asynchrone Kommunikationsstruktur zu verbinden, um somit das GALS-Paradigma zu erfüllen.

In diesem Abschnitt werden verschiedene GALS-Methoden für NoC-Verbindungsstrukturen vorgestellt, die ein global-asynchrones Many-Core-System ermöglichen. Dazu wird zunächst Bezug zum Stand der Technik genommen, indem verschiedene NoCs und ihre GALS-Methoden vorgestellt werden. Anschließend wird die Implementierung eines mesochronen und eines asynchronen Routers für das CoreVA-MPSoC vorgestellt. Ein Vergleich dieser Implementierungen mit dem synchronen NoC ist in Abschnitt 5.3.4 aufgeführt.

5.3.1 Bezug zum Stand der Technik

Zur Implementierung eines GALS-basierten MPSoCs sind in der Literatur viele verschiedene NoC-Architekturen beschrieben. Grundsätzlich lässt sich dabei zwischen zwei

verschiedenen Paradigmen unterscheiden: Einem vollkommen asynchronen NoC und einem Multi-synchronen NoC.

In Multi-synchronen NoCs arbeiten die Schaltungsteile in den Routern weiterhin synchron auf einem internen Taktsignal. Die direkte Kommunikation zwischen Routern erfolgt jedoch asynchron. Der verbreitetste Ansatz in Multi-synchrones NoCs ist die Verwendung von mesochronen Links. Hierbei arbeiten die Router zwar noch mit derselben Taktfrequenz, jedoch darf sich zwischen den Routern eine unbekannte Phasenverschiebung einstellen [175]. Dies vereinfacht den globalen Taktbaum zu den lokal synchronen Routern und CPUs, sodass hier kleinere Takttreiber eingesetzt werden können. Um Verletzungen von Setup- und Hold-Zeiten zu verhindern, muss die Phasenverschiebung zwischen den Routern jedoch durch entsprechende Maßnahmen ausgeglichen werden. Hierzu können auf den Links zwischen den Routern neben bisynchronen FIFOs [185] auch spezielle Synchronisationsschaltungen eingesetzt werden, da die Taktfrequenz beim Sender- und Empfänger-Modul identisch ist. Letztere weisen sowohl einen geringeren Flächenbedarf, als auch eine geringere Latenz auf [204]. Ein Beispiel eines mesochronen Synchronisierers wird in [113] vorgestellt. Dieser stellt mit Hilfe von drei Datenregistern, einem DLL-basiertem Frequenz-Verdoppler sowie einem Entscheider die Synchronisation sicher. In [120] wird die Synchronisation ebenfalls nach einer Detektion der Phasenverschiebung und durch eine dynamische Veränderung der Signalverzögerung auf den Links sichergestellt. Ein ähnlicher Ansatz wird im Spidergon-STNoC verwendet [158]. Dort wird die Komponente SIML (Skew Insensitive Mesochronous Link) für die Synchronisation eingesetzt. Beim SIML werden insgesamt vier Datenregister auf der Empfängerseite verwendet. Zusätzlich muss im Sender ein Abtastimpuls erzeugt werden. Die vier Register werden hier als Puffer mit zwei Stufen verwendet, sodass sich beim SIML eine Latenz von zwei Taktzyklen einstellt. Eine flächeneffizientere Umsetzung bietet der Tightly Coupled Mesochronous Synchronizer (TCMS), welcher in [106] vorgestellt wird. Diese Architektur benötigt lediglich drei Latches und zwei 2-bit-Zähler zur Phasensynchronisierung. Zusätzlich stellt sich beim TCMS eine Latenz von ein bis zwei Taktzyklen ein, welche aufgrund der unbekannten Phasenverschiebung als minimal anzusehen ist. Aus diesem Grund wird auch im mesochronen NoC des CoreVA-MPSoCs der TCMS eingesetzt.

Bei asynchronen Systemen übernehmen Handshake-Verfahren die explizite Synchronisation zwischen zwei Teilnehmern. In der Literatur werden dazu zwei verschiedene Protokolle beschrieben, ein vierphasiges *Level-Signaling-Protokoll* und ein zweiphasiges *Transition-Signaling-Protokoll* [180]. Beim vierphasigen *Level-Signaling-Protokoll* signalisiert ein gesetztes *Request*-Signal ein neues Datenwort. Nachdem der Empfänger dieses mit seinem *Acknowledge*-Signal bestätigt hat, kehrt der Sender zum Ausgangspegel zurück. Sobald auch der Empfänger wieder zum Ausgangspegel zurückgekehrt ist, kann ein neuer Zugriff erfolgen. Es handelt sich also um ein Verfahren mit Rückkehr zum Nullzustand. Das zweiphasige Transition-Signaling-Protokoll vermeidet diese Pegelwechsel. Neue Datenwörter werden direkt durch einen Wechsel des Zustandes des Request-Signals angezeigt. Beispielhafte Umsetzungen dieser beider Methoden werden

im Folgenden vorgestellt. Die Veröffentlichung von Thonnart et. al. [180] befasst sich mit einem Framework für ein asynchrones NoC. Die asynchrone Kommunikation basiert auf dem Level-Signaling-Protokoll. Über vier I/O-Ports werden die einzelnen Router zu einem 2D-Mesh mit Wormhole-Routing verschaltet und ein fünfter Port führt zu den lokalen IPs. Um die Begrenzungen des CAD-Werkzeugs zu überwinden, ist ein neuer Entwurfsablauf entstanden und an die Bedürfnisse asynchroner Schaltungen angepasst. Ein gefertigtes System in einer 65-nm-CMOS-Technologie weist dabei eine Datenrate von 550 MFlit/s, bei einer Leistungsaufnahme von 86% eines vergleichbaren synchronen Systems, auf. In [135] wird ein asynchrones Routerdesign vorgestellt, das auf einem Level-encoded-dual-rail-Protokoll arbeitet. Die Methodik hinter diesem Protokoll ist nah am zweiphasigen Level-Signaling-Protokoll, verwendet allerdings einen größeren Satz an Kontrollsignalen. Die I/O-Ports eines Routers werden in diesem Design für die gesamte Dauer einer Übertragung reserviert (Circuit-Switching) und erst mit Abschluss dieser für andere Anfragen freigegeben. Zur Analyse wird eine Fertigung in einer 0.13-nm-Technologie vorgestellt, bei der ein 4x4 2D-Mesh mit 16 Spidergon-Kernen abgebildet ist. Es wird eine Latenz von 2,74 ns und ein Durchsatz von 526 MFlits/s erreicht. Auch im Argo-NoC wird ein asynchrones NoC verwendet [90][91]. Die Besonderheit innerhalb des asynchronen Routers stellen die Crossbars dar, mit denen die Eingangs- und Ausgangsports miteinander verbunden sind. In dieser werden sogenannte *Handshake-Takte* durchgeführt, in denen von jedem Eingangsport ein Flit konsumiert und an allen Ausgangsport ein Flit produziert wird. Ein Zustandsbit signalisiert dabei ungültige Flits, welche nur für die Synchronität erzeugt werden. Daraus folgt allerdings auch, dass alle eingehenden Flit auf dieselbe Phase verschoben werden müssen, da es sich um ein Level-Signaling-Protokoll handelt. Dies geschieht zwischen Router und dem davor implementierten FIFO, welche durch eine Mousetrap-Pipeline realisiert werden. Auch im Argo-NoC nutzt das Circuit-Switching, welches die Arbitrierung innerhalb der Router vereinfacht. Für den Übergang zwischen synchroner und asynchroner Domäne werden jedoch keine Vorkehrungen für das asynchrone Request- und Acknowledge-Signal getroffen. Dies kann zu metastabilen Zuständen in den synchronen Registern führen.

Grundsätzlich bietet das zweiphasige Transition-Signaling-Protokoll einige Vorteile gegenüber dem vierphasiges Level-Signaling-Protokoll. Durch das Zurückkehren zum Nullzustand im Level-Signaling-Protokoll sind die Signale zum häufigen Wechsel des Signalpegel gezwungen. Beim zweiphasigen Transition-Signaling-Protokoll wird ein Pegelwechsel nur bei Bedarf vollzogen, sodass die gesamte Kommunikation kompakter, effizienter und vor allem auch energieeffizienter geschehen kann, da insgesamt deutlich weniger Schaltvorgänge benötigt werden.

Vergleiche zwischen verschiedenen GALS-Methoden und vollständig synchronen Systemen sind bereits in vielen Veröffentlichungen durchgeführt worden [193][54]. Ein Vergleich zwischen mesochronen und vollständig asynchronen NoCs im Einsatz hierarchischer MPSoCs ist bisher jedoch nur selten untersucht worden. Soweit bekannt hat dieses Thema nur die Arbeit von Sheibanyrad et al. [160] aufgegriffen, in dem sie ein

mesochrones NoC (DSPIN) und ein vollständig asynchrones NoC (ASPIN) miteinander vergleichen. Im DSPIN werden bi-synchrone FIFOs zwischen den Routern verwendet, um ein mesochrones NoC aufzubauen. Das ASPIN arbeitet hingegen mit vollständig asynchronen Routern, die das vierphasige Level-Signaling-Protokoll verwenden. Des Weiteren wird ein Dual-Rail-Protokoll verwendet, bei dem jeweils zwei Leitungen ein Bit repräsentieren. Dieses Dual-Rail-Protokoll wird in der ASPIN-Architektur benötigt, um die Bits eines Flits miteinander zu synchronisieren. Der Vergleich zwischen DSPIN und ASPIN ist für den maximalen Durchsatz, minimale Latenz, Chipfläche und Leistungsaufnahme durchgeführt. Hervorzuheben ist, dass die Arbeit auch die Leitungsverzögerung zwischen den Routern betrachtet, die insbesondere bei hierarchischen MPSoCs nicht zu vernachlässigen ist. Beide NoC-Designs zeigen sehr ähnliche Ergebnisse für den maximalen Durchsatz und Chipfläche. Bei einer um 2,5 mal geringeren minimalen Latenz übertrifft das asynchrone ASPIN das DSPIN. Hinzu kommt, dass die Leistungsaufnahme im Idle-Modus im ASPIN dreimal geringer ist, dieses jedoch während der aktiven Datenübertragung eine etwas höhere Leistungsaufnahme aufweist. Insbesondere bei langen Leitungen zwischen den Routern verliert das Dual-Rail-Protokoll die Vorteile des asynchronen NoCs, da doppelt so viele Leitungen benötigt werden wie im synchronen NoC.

Im asynchronen NoC des CoreVA-MPSoC – dessen genaue Implementierung im Folgenden vorgestellt wird – wird aus den zuvor genannten Gründen das zweiphasige Transition-Signaling-Protokoll in Verbindung mit der sogenannten Mousetrap-Schaltung eingesetzt. Eine zusätzliche Verzögerung des Request-Signals in Bezug auf die Datenbits eines Flits, erlaubt einen Verzicht auf das Dual-Rail-Protokoll. Dies führt dazu, dass die Anzahl der Leitungen zwischen Routern für alle drei Router-Designs (synchron, mesochron und asynchron) nahezu identisch ist.

5.3.2 Implementierung Mesochroner Router

Zur Umsetzung eines mesochronen NoCs im CoreVA-MPSoC wird das Eingangsregister innerhalb des synchronen Routers (siehe Abschnitt 5.1) durch einen mesochronen Synchronisierer ersetzt. Im CoreVA-MPSoC wird hierzu das Konzept des zuvor genannten TCMS [106] umgesetzt und implementiert. Dieser zeichnet sich durch eine sehr enge Kopplung an den Router aus, welches sich auch im Namen *Tightly Coupled Mesochronous Synchronizer* wieder spiegelt. Die Aufgabe des TCMS ist es, die mögliche Phasenverschiebung des Taktes zwischen zwei Routern auszugleichen, sodass alle Datenbits eines Flits in einem stabilen Zustand im FIFO der Router gespeichert werden können.

Der TCMS ist in das sogenannte *Front-end* und das *Back-end* unterteilt (siehe Abbildung 5.11). Das Front-end befindet sich am Eingang des TCMS und erhält den Takt *CLK_EXT* zusammen mit dem Flit vom externen Router. Das Back-end bildet den Ausgang des TCMS und gehört zur Taktdomäne des lokalen Routers. Zwischen Front- und Back-end findet die Synchronisation statt.

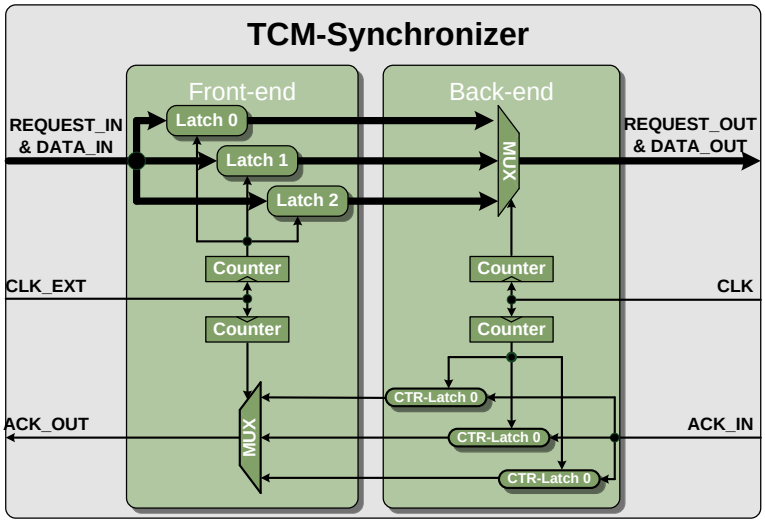


Abbildung 5.11: Architektur des Tightly Coupled Mesochronous Synchronizer (TCMS) wie sie im CoreVA-MPSoC umgesetzt ist.

Im Front-end befinden sich die drei Latches mit einer Datenbreite eines gesamten Flits, inklusive des Request-Signals. Welches der drei Latches die Daten speichert, wird durch einen Zähler bestimmt. Dieser 2-Bit-Zähler steuert die Enable-Signale der Latches. Der Zähler zählt von Null bis Zwei, sodass der Wert auch das Latch bestimmt, in dem die ankommenden Daten abgespeichert werden. Damit es beim Schreiben der Daten in ein Latch nicht zu einem metastabilen Zustand kommt, schaltet der Zähler mit dem Takt des externen Routers. Des Weiteren muss verhindert werden, dass ein Enable-Signal eines Latches zu lange aktiv ist und schon das nächste Datenwort speichert. Dies wird sichergestellt, indem ein Enable-Signal nur während des High-Pegels des externen Taktes aktiv ist. Durch eine Simulation auf Gatterebene, bei der die Verzögerungszeiten der Schaltelemente mitberücksichtigt werden, kann jedoch ein fehlerhaftes Verhalten bei der in [106] vorgestellten Schaltung, verifiziert werden. So bewirken Glitches bei den Enable-Signalen der Latches, dass Daten in einem falschen Latch gespeichert werden. Dieses Fehlverhalten ist darauf zurückzuführen, dass ein Enable-Signal während des High-Pegels schaltet, sich der Wert des Zählers aufgrund von Verzögerungszeiten aber erst kurze Zeit später ändert. Um diesem Problem aus dem Weg zu gehen, wird der Zähler im TCMS des CoreVA-MPSoCs so implementiert, dass dieser zur negativen Flanke des externen Taktes zählt. Dies hat zur Folge, dass der Wert des Zählers bereits eine halbe Taktperiode stabil ist, bevor der nächste High-Pegel des Taktes kommt, der das entsprechende Enable-Signal aktiviert. Da die Verzögerungszeit

des Zählers aufgrund des längeren Pfades immer größer ist als die des externen Taktes, kann auch am Ende des High-Pegels ein korrektes Verhalten sichergestellt werden.

Das Back-end des TCMS entscheidet aus welchem Latch die Daten weitergeleitet werden. Dort befindet sich ein Multiplexer, der durch einen weiteren 2-bit-Zähler gesteuert wird. Dieser Zähler schaltet zur positiven Flanke des Taktes des lokalen Routers (*CLK*) und zählt ebenfalls von Null bis Zwei. Der Zähler muss genau so eingestellt sein, dass er die Daten eines Latches weiterleitet, die schon über einen längeren Zeitraum dort gespeichert und damit stabil sind. Der in Abbildung 5.12 dargestellte Signalverlauf gibt ein Beispiel des Schaltverhaltens innerhalb des TCMS.

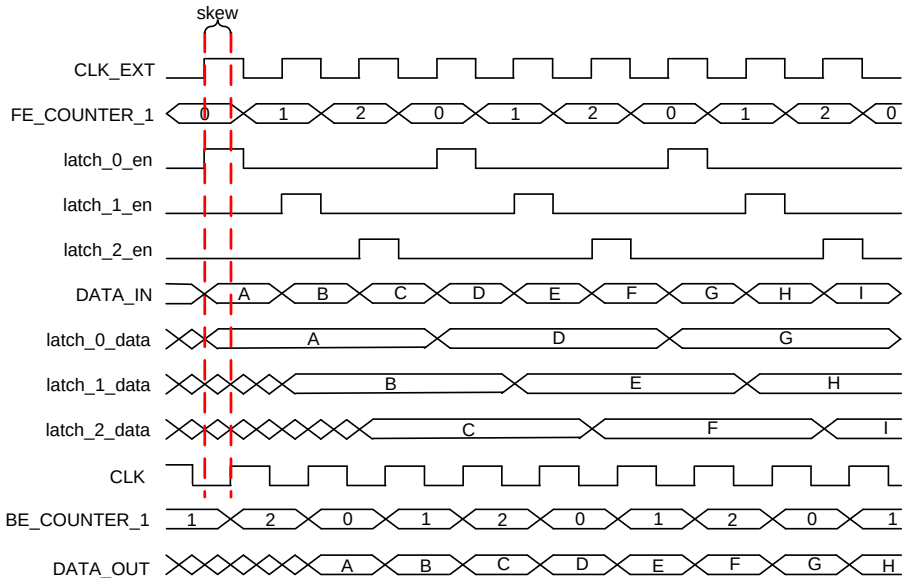


Abbildung 5.12: Beispiel für das Schaltverhalten im TCMS.

Damit der TCMS überhaupt metastabile Zustände vermeidet, müssen die Zähler korrekt zueinander eingestellt sein. So muss die Einstellung des Zählers im Front-end genau zur Einstellung des Zählers im Back-end passen. Es ist eine Konstellation zu finden, die bei allen möglichen Phasenverschiebungen zwischen -180° und +180° nur Daten weiterleitet, die einen stabilen Zustand im Latch erreicht haben. Die korrekte Einstellung der Zähler wird durch ein geeignetes Reset der Zähler sichergestellt. Das globale Reset des CoreVA-MPSoC ist asynchron. Dies bedeutet, dass nicht feststeht, wie das Reset in Bezug zur Taktflanke steht. Somit ist die Konstellation der Zähler nicht deterministisch und damit kein geeigneter Startwert der Zähler zu bestimmen. Um einen definierten Anfangszustand zu erzwingen, muss zunächst das Reset mit dem Takt des lokalen Routers synchronisiert werden. Zur Synchronisierung für das Reset-Signal wird

ein *Brute-Force*-Synchronisierer verwendet, der durch zwei hintereinandergeschaltete Flip-Flops realisiert ist. Mit diesem, zum lokalen Router synchronisierten Reset-Signal, werden alle Zähler im TCMS zurückgesetzt. Dies kann jedoch in den Zählern des Front-ends zu metastabilen Zuständen führen, da das Reset-Signal sich zu nah an der Taktflanke des externen Takts befinden könnte. Um das zu vermeiden, wird der Takt des Zählers im Front-end erst gestartet, wenn das Reset bereits erfolgt ist. Die Verzögerung des Taktes wird mit Hilfe von Latch-basiertem Clock-Gating realisiert, um Glitches innerhalb des Taktsignals zu vermeiden. Damit im verwendeten Latch ebenfalls kein metastabiler Zustand entsteht, wird das zum internen Takt synchrone Reset wiederum mit einem *Brute-Force* Synchronisierer zum externen Takt synchronisiert. Es ist wichtig, an dieser Stelle das bereits zum internen Takt synchronisierte Reset noch einmal zu synchroniseren, um einen definierten Anfangszustand zu erhalten.

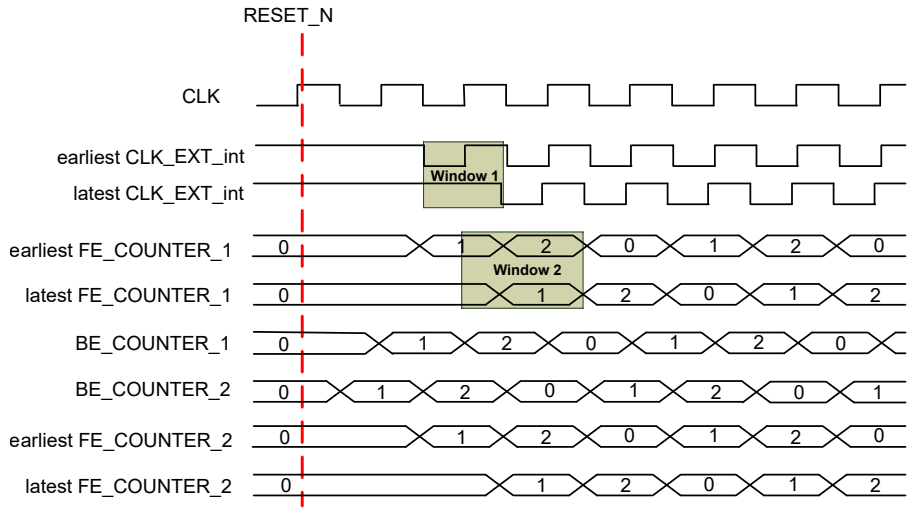


Abbildung 5.13: Signalverlauf zur Einstellung der Zähler im TCMS.

Nach Einstellen des Resets ist ein geeigneter Startwert der Zähler zu definieren. Wie in Abbildung 5.13 zu sehen ist, gibt es nach dem Reset aufgrund der Phasenverschiebung ein Fenster (*Window 1*), in dem die erste negative Flanke des externen Takts (CLK_EXT_int) auftreten kann. Dieses Fenster erstreckt sich aufgrund einer möglichen Phasenverschiebung von bis zu 360° über eine Taktperiode. Auch für den Zähler im Front-end existiert ein Fenster, in dem er einen festen Wert hat. Das Fenster *Window 2* gibt den Zeitraum an, in dem Daten in das Latch mit der aktuellen Nummer des Zählers übernommen werden können. Durch Aufbau dieses Szenarios lässt sich nun der Startwert für den Zähler im Back-end bestimmen, der ein stabiles Datenwort aus dem Latch weiterleitet.

Bisher wurde nur der Datenpfad des ankommenden Flits beschrieben. Im Kontrollpfad zum externen Router muss jedoch auch das ON/OFF-Signal, welches sich aus dem Füllstand der FIFOs ergibt, in die Taktdomäne des externen Routers übertragen werden. Hierzu wird das gleiche Prinzip wie beim Datenpfad verwendet. So stehen im Back-end des TCMS drei weitere Latches zur Verfügung die das ON/OFF-Signal speichern. Da es sich nur um ein einzelnes Signal handelt, werden an dieser Stelle 1 Bit breite Latches verwendet. Die Enable-Signale werden ebenfalls durch einen 2-bit-Zähler gesteuert, der jedoch zur negativen Flanke des Taktes des lokalen Routers schaltet. Aus demselben Grund wie im Datenpfad sind auch hier die Enable-Signale nur während des High-Pegels des lokalen Taktes aktiv. Im Front-end befindet sich ein Multiplexer, der durch einen weiteren Zähler gesteuert wird und das ON/OFF-Signal aus dem entsprechenden Latch weiterleitet. Diese Erweiterung innerhalb des TCMS sorgt dafür, dass im externen Router keine weitere Synchronisation der Bestätigung notwendig ist.

Aus dem in Abbildung 5.13 dargestelltem Signalverlauf kann auch die Latenz des TCMS abgeleitet werden. Die Latenz ist abhängig von der Phasenverschiebung zwischen den beiden Taktsignalen und kann zwischen ein und zwei Taktzyklen betragen. Gleiches gilt für die Latenz des Bestätigungssignals, welches zum externen Router zurückgesendet wird. Da es sich hier jedoch um den entgegengesetzten Fall handelt, ergänzen sich beide Verzögerungszeiten immer zu genau drei Taktzyklen Gesamtverzögerung bis das Bestätigungssignal am externen Router ankommt. Da dies im Vergleich zum synchronen Fall ein Taktzyklus länger dauert, muss das Almost-Full-Signal der FIFOs bereits bei nur drei freien Plätzen gesetzt werden (vgl. Abschnitt 5.1). Dies ist bei der Größe der FIFOs zu berücksichtigen und führt zu einer Mindestgröße von fünf Flits.

5.3.3 Implementierung Asynchroner Router

Zur Implementierung des asynchronen Routers werden die synchronen Schaltungsteile des synchronen Router (siehe Abschnitt 5.1) durch asynchrone Schaltungsteile ersetzt. Ansonsten setzt der asynchrone Router genau die gleiche Funktionalität wie der synchrone und mesochrone Router um. Die Implementierung des asynchronen Routers für das CoreVA-MPSoC ist insbesondere in der betreuten Masterdarbeit [224] entstanden. Der Aufbau des asynchronen Routers wird in Abbildung 5.14 dargestellt.

Eine Kernkomponente für das implementierte Routerdesign bildet das *Mousetrap* FIFO. Die Mousetrap (MT), vorgestellt in [164], ist ein simples aber effektives asynchrones Pipelinestufen-Design. An allen Ein- und Ausgängen eines Routers speichern diese über das NoC verschickte Flits zwischen und sorgen gleichzeitig für die Umsetzung des zweiphasigen Transition-Signaling-Protokoll. In Abbildung 5.15a ist der Aufbau des Mousetrap FIFO dargestellt. Eine Mousetrap-Stufe besteht aus Latches für Daten- und Request-Signal und einem XNOR für die Transparentschaltung der Latches. Die Funktionsweise ähnelt dabei dem Namen getreu einer Mausefalle. Alle Latches sind im Ausgangspunkt transparent und lassen Daten durch. Wechselt das Request-Signal seinen Wert, schnappt die Mausefalle zu und hält den Wert, bis das eingehende Acknowledge-

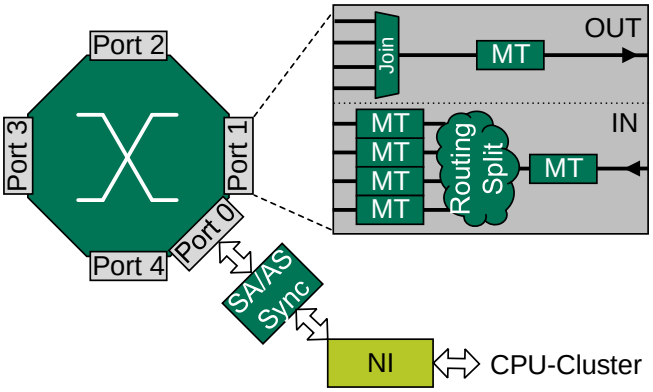
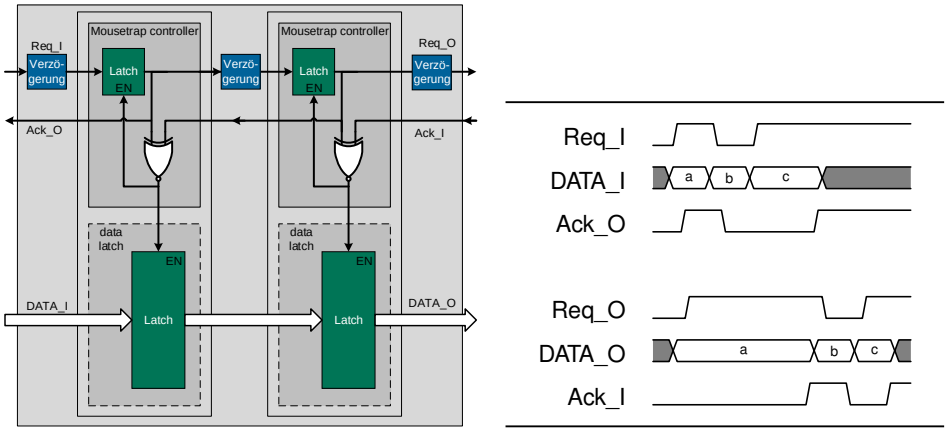


Abbildung 5.14: Architektur des asynchronen Routers.

Signal die Abnahme bestätigt. Das ausgehende Request-Signal zur nächsten Stufe wird ebenfalls als zurückführendes Acknowledge-Signal verwendet. Bei diesem Design muss auf die Signalreihnenfolge geachtet werden. Um metastabile Zustände an den Datenlatches zu vermeiden, müssen die Daten vor dem Eintreffen des Request-Signals anliegen. Aus diesem Grund wird das Request-Signal vor einer Mousetrap-Stufe stets durch eine Inverterkette leicht verzögert. Für eine FIFO-Komponente sind mehrere dieser Mousetrap-Stufen hintereinander geschaltet. Die Tiefe des implementierten FIFO ist konfigurierbar.

Der Signalverlauf in Abbildung 5.15b verdeutlicht das Schaltverhalten für ein Mousetrap FIFO der Tiefe Zwei. Das eingehende Request-Signal (*Req_I*) durchläuft bei diesem Aufbau zwei Latches und eine Inverterkette mit 6 Invertern. Im Gegensatz dazu durchlaufen die Datenbits nur zwei Latches und besitzen somit eine geringere Verzögerung. Durch diesen Laufzeitunterschied wird die richtige Signalreihenfolge sichergestellt. Damit verbunden ist auch das Einhalten des angestrebten zweiphasigen Transition-Signaling-Protokolls. Bei dem Aufbau eines FIFO aus Mousetrap-Elementen ist darauf zu achten, dass das zurückführende Acknowledge-Signal der vorhergehenden Stufe ebenfalls eine gewisse Latenz aufweist. Dies ist notwendig, um die Hold-Zeit der Latches sicherzustellen und jegliche Metastabilität auf den Signalen zu vermeiden. Hierfür sind keine besonderen Vorkehrungen zu treffen, allerdings sind Verzögerungen durch Schaltungsgatter und Leitungen zu berücksichtigen. Je nach tatsächlichem Layout-Design und finalen Verzögerungszeiten muss die Länge der Inverterkette ggf. angepasst werden. Dies ist die einzige manuell anpassbare Größe, die jedoch einen gewissen Mehraufwand beim Erstellen des Chiplayouts darstellt. Durch den hierachischen Entwurfsablauf beschränkt sich dieser Aufwand jedoch auf die Erstellung eines einzigen Cluster-Knotens (siehe Kapitel 8.1).



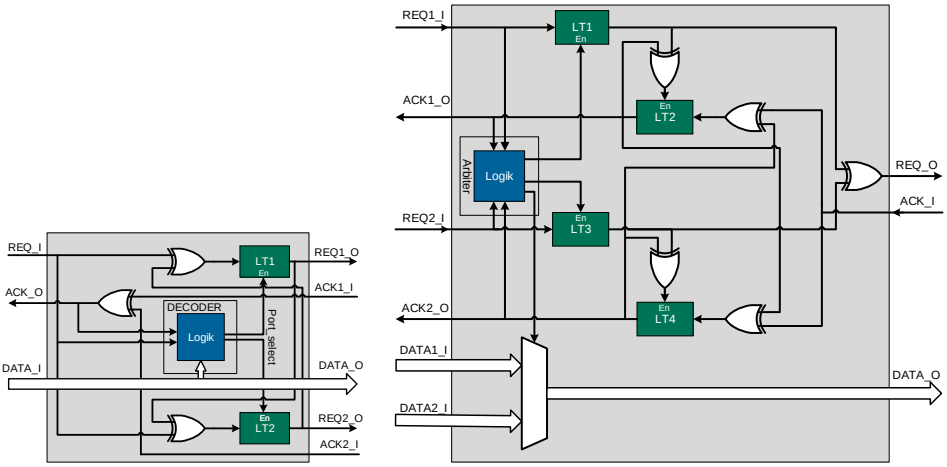
(a) Verschaltung zweier Mousetrap-Schaltungen zu einem FIFO.

(b) Beispielhaftes FIFO-Schaltverhalten. Das erste Datenwort wird verzögert abgenommen.

Abbildung 5.15: Aufbau und Schaltverlauf für ein Mousetrap-FIFO der Tiefe Zwei.

Wie in Abbildung 5.14 zu sehen, muss das Routing-Element das ankommende Flit je nach Zielport an die entsprechende Mousetrap weiterleiten. Die Routingentscheidung selbst ist analog zum synchronen Router durch ein XY-Routing umgesetzt. Um im Transition-Signaling-Protokoll ein Abzweigen der Daten zu ermöglichen, wird ein Split mit wechselseitigem Ausschluss benötigt. Die in [55] vorgestellte Schaltung ermöglicht ein Abzweigen der Daten auf zwei Ausgangsports (siehe Abbildung 5.16a). Die Daten können ohne Änderung durch dieses Element geführt werden, da es nur einen Sender gibt und es somit zu keiner Kollision kommen kann. Ausgehende Kontrollsignale sind dagegen eindeutig an einen Empfänger zu übertragen. Es darf immer nur ein Port bedient werden, da ein wechselseitiger Ausschluss bestehen soll. Zwei Latches halten dazu die Ausgangswerte der Request-Signale, wenn diese vom Dekodierer – in diesem Fall der Routing-Komponente – nicht transparent geschaltet sind. Ein *XOR*-Gatter vor jedem dieser Latches sorgt für den richtigen Ausgangspegel, da immer nur einer der Pegel geändert wird. Dies ist einzuhalten, da im zweiphasigen Transition-Signaling-Protokoll ein Pegelwechsel immer ein neues Datenwort signalisiert. Für die Aufteilung auf die insgesamt vier I/O-Ports bei der standardmäßigen 2D-Mesh-Topologie des CoreVA-MPSoC werden mehrere dieser Split-Instanzen hintereinander kaskadiert.

Analog zum Split müssen Datenpfade beim Transition-Signaling-Protokoll auch wieder zusammengeführt werden können. Im Router wird dies in der Crossbar bzw. Arbitrierung vor den Ausgangsports umgesetzt (siehe Abbildung 5.14). Das Join ist der Implementierung aus der Arbeit [55] nachempfunden und in Abbildung 5.16b zu sehen.



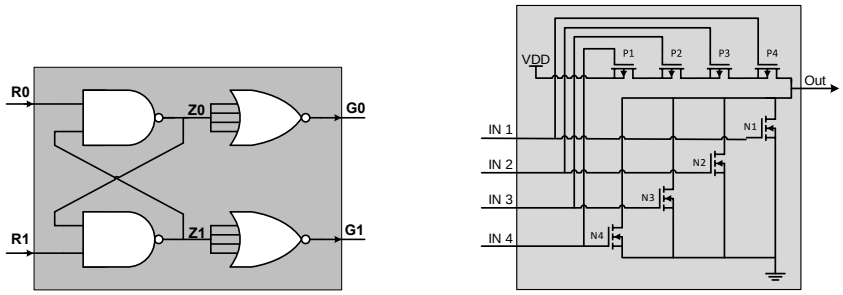
(a) Der Aufbau des Splits mit wechselseitigem Ausschluss.

(b) Der Aufbau des Joins.

Abbildung 5.16: Split- und Join-Elemente zum Abzweigen und Zusammenführen von asynchronen Datenpfaden

In diesem Beispiel verbinden sich zwei Eingangsports zu einem Ausgangsport. Vier Latches halten die ausgehenden Signale auf ihren Pegeln und mehrere *XOR*-Gatter sorgen für die richtigen Pegel. Da beide Eingangsports gleichzeitig neue Daten liefern können, muss ein spezieller Arbiter für den wechselseitigen Ausschluss sorgen. Ein sicherer wechselseitiger Ausschluss stellt eine besondere Herausforderung in einem asynchronen Design dar [133][141][196]. Durch das Fehlen eines Taktes muss auf andere Weise sichergestellt werden, dass zu jedem Zeitpunkt und Zustand eine gültige Entscheidung zwischen Anfragen von Teilnehmern getroffen wird. Schwankungen auf den Request-Signalen und besonders metastabile Zustände müssen um jeden Preis vermieden werden. Mögliche Implementierungen, die ein solches Schaltbild aufweisen, bestehen in der Regel aus einem Teil für den wechselseitigen Ausschluss und einem Filter für metastabile Zustände. Der kombinatorische Teil für den Ausschluss besteht dabei immer aus zwei kreuzverbundenen *NAND*-Gatter, von denen nur ein Ausgang logisch 0 sein darf.

Die Schaltpläne in Abbildung 5.17a veranschaulichen das für diese Arbeit verwendete Design. Es besteht aus zwei *NAND*- und zwei *NOR*-Gattern. Für den wechselseitigen Ausschluss sind die Ausgänge der beiden NANDs mit einem Eingang des jeweils anderen NANDs verbunden. Die weiteren Eingänge sind für die externen Eingangssignale. Da ein NAND-Gatter nur logisch 0 als Ausgang besitzt sofern beide Eingänge logisch



(a) Kombinatorischer Aufbau des Arbiters.

(b) Aufbau eines NOR4 in CMOS-Technologie.

Abbildung 5.17: Aufbau des Arbiters zur Filterung von metastabilen Zuständen innerhalb des Join-Elements.

1 aufweisen, sind im Regelfall die beiden Ausgänge logisch 1. Wechselt einer der Eingänge auf logisch 1, so wechselt der Ausgang des dazu gehörigen NAND-Gatter auf logisch 0. Dies verhindert den Wechsel des anderen NAND-Gatter zu logisch 0 und der gegenseitige Ausschluss ist gegeben. Beim möglichen Fall des gleichzeitigen Wechsels der Eingangssignale zu logisch 1 können allerdings metastabile Zustände entstehen. Zu begründen ist dies durch die Abhängigkeit zum jeweils anderen NAND-Gatter. Es kann in diesem Fall einige Zeit dauern, bis einer der Ausgänge eine stabile logische 1 ausgibt. Durch leichte Varianz der elektrischen Bauteile der NAND-Gatter ist aber das Erreichen eines stabilen Zustandes zu erwarten. Um diese Metastabilität herauszufiltern, können zwei NOR-Gatter mit jeweils vier Eingängen verwendet werden. Sie sind durch ihren Aufbau für diese Filteraufgabe geeignet. Bei einem NOR-Gatter mit vier Eingängen sind vier PMOS-Transistoren in Reihe geschaltet. Diese schalten langsamer als NMOS-Transistoren und sind generell hochohmiger. Der Aufbau ist in Abbildung 5.17b dargestellt. Um eine Aussage über das Verhalten des Arbiters treffen zu können, ist die Schaltung in der Simulationsumgebung Virtuoso in einer 28-nm-Standardzellentechnologie implementiert und simuliert und damit charakterisiert. Ergebnisse der Spice-Simulationen dieser Schaltung sind in Anhang D zu finden. Im Join-Element hängt die Entscheidung zwischen den eingehenden Daten nur vom Arbiter ab. Die Join-Elemente werden ebenfalls kaskadiert, um die vier Eingangsports auf einen Ausgangsport abzubilden.

Eine weitere kritische Stelle bei der Verwendung eines asynchronen NoCs ist die Schnittstelle zwischen Router und NI. Da der NI weiterhin synchron betrieben wird, besteht die Notwendigkeit einen stabilen Signalübergang zwischen diesem und dem asynchronen Port0 herzustellen (Komponente *SA/AS-Sync* aus Abbildung 5.14). Be-

trachtet man zunächst den Wechsel eines einzelnen Signals zwischen einer asynchronen und synchronen Schaltung, ist hier nur der Übergang von asynchron zu synchron problematisch. Bei asynchronen Systemen ist der Zeitpunkt des Setzens eines Signals an keine Bedingungen geknüpft und daher unproblematisch. Synchrone Systeme beruhen hingegen auf einem Datenfluss von einem Register zum nächsten. Register übernehmen ihren Wert zu jeder steigenden oder fallenden Taktflanke, wobei das ankommende Signal eine gewisse Zeit vor diesem Zeitpunkt bereits einen stabilen Zustand angenommen haben muss (Setup-Zeit). Im Falle eines ankommenden asynchronen Signals kann diese Annahme allerdings nicht getroffen werden. Um metastabile Zustände bei Missachtung der Schaltzeiten von Registern zu verhindern, wird ein Synchronisierer benötigt. Im CoreVA-MPSoC wird hierzu der bereits im mesochronen Router eingesetzte *Brute-Force*-Synchronisierer verwendet. Um eine Aussage über das erwartete Schaltverhalten dieser Schaltung treffen zu können, wurde hierzu eine SPICE-Simulation auf Transistorebene durchgeführt (siehe Abbildung A.1 in Anhang D).

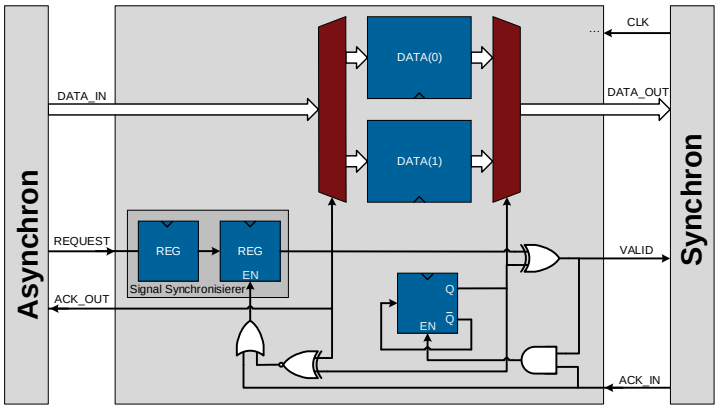


Abbildung 5.18: Aufbau des Synchronisierers von Asynchron zu Synchron.

Zwischen Netzwerk-Schnittstelle und Router müssen jedoch nicht nur einzelne Bits sondern ganze Datenworte (Flits) ausgetauscht werden. Beim Übergang von Flits vom Router zur Netzwerk-Schnittstelle müssen die Datenbits eines Flits und das Request-Signal synchronisiert werden. Das Blockschaltbild in Abbildung 5.18 stellt die Architektur dieser Synchronisationsschaltung dar. Ähnlich wie bei der Mousetrap-Schaltung wird auch hier das Request-Signal so verzögert, um die Abhängigkeit zum Datenwort sicher zu stellen. Das Datenwort trifft damit immer vor dem Request-Signal ein und kann damit stabil in einem Register gespeichert werden. Damit muss allein das Request-Signal den Brute-Force-Synchronisierer passieren. Gültige Daten liegen mindestens eine halbe Taktlänge eher am nächsten Datenregister an. In einer Zielfrequenz von 700 MHz entspricht dies einer Zeit von über 700 ps, was im Vergleich zur Setup-Zeit

der Register ausreicht, um die Daten als valide annehmen zu können. Sollte eine Metastabilität auf den Datenleitungen auftreten, trifft das Request-Signal einen ganzen Takt später ein und das Datenwort wird erneut erfasst. Somit wird erst mit einem garantiert gültigen Wort fortgefahren. Um einen möglichst hohen Durchsatz zu erzielen, werden eingehende Datenwörter in einem von zwei Registern gespeichert. Es existiert je ein Register für den Fall einer logischen 0 und für den Fall einer logischen 1 des Request-Signals im zweiphasigen Transition-Signaling-Protokoll.

Beim Übergang von der synchronen zur asynchronen Domäne muss das Acknowledge-Signal durch den Brute-Force-Synchronisierer synchronisiert werden. Der Zustand des registerten Acknowledge-Signals steuert, aus welchen Registern das Datenwort auf den Ausgang gelegt wird. Eingehende Flits auf der synchronen Seite werden in eins von zwei Registern geschrieben. Auch hier werden zwei Register dazu verwendet, um in das zweiphasigen Transition-Signaling-Protokoll des asynchronen Routers zu überführen. Ein Flit hat beim Passieren der Synchronisierungsschaltungen sowohl vom NI zum Router als auch vom Router zum NI eine Latenz von einem Takt.

5.3.4 Entwurfsraumexploration von GALS-Methoden

Zur Bewertung der im vorherigen Abschnitt vorgestellten GALS-Methoden wird in diesem Abschnitt eine Entwurfsraumexploration durchgeführt. Dazu wird das asynchrone NoC-Design mit dem synchronen und mesonchronen Design verglichen. Die Entwurfsraumexploration wird auch hier für die Bewertungsmaße Flächenbedarf, Leistungsaufnahme sowie Latenz und Durchsatz durchgeführt (vergl. Kapitel 4).

Insbesondere mit Bezug auf das asynchrone Design sind hier nur aussagekräftige Ergebnisse auf einem platzierten und verdrahteten Layout zu erzeugen. Dies ist erforderlich, da Leitungsverzögerungen beim Entwurfsablauf eines asynchronen Designs mitberücksichtigt werden müssen. Insbesondere bei hierarchischen Many-Core-Systemen, wie dem CoreVA-MPSoC, sind diese aufgrund der größeren CPU-Cluster nicht zu vernachlässigen. Basis der folgenden Ergebnisse bildete daher das platzierte und verdrahtete Design eines Cluster-Knotens. Mehrere dieser Cluster-Knoten können dann nach dem in Kapitel 3.3.5 vorgestellten hierarchischen Entwurfsablauf zusammengeschaltet werden. Ein Cluster-Knoten besteht aus vier CPU-Makros (siehe Kapitel 3.1), der Cluster-Verbindungsstruktur und 64 kB geteiltem L1-Datenspeicher (siehe Kapitel 7.2). Hinzu kommen die NoC-Komponenten, wie die Netzwerk-Schnittstelle und der Router mit seinen I/O-Port in alle vier Richtungen eines 2D-Mesh. Die maximale Taktfrequenz für die synchronen Teile des Knotens wird durch den geteilten L1-Speicher vorgegeben und liegt bei etwa 700 MHz.

Um die Auswirkungen der verschiedenen GALS-Methoden auf das gesamte System zu untersuchen, wird im Abschluss dieses Abschnitts der globale Taktbaum analysiert, der sich durch Zusammenschalten der verschiedenen Knoten ergibt. Die in diesem Abschnitt vorgestellten Ergebnisse sind in [199] veröffentlicht.

Flächenbedarf

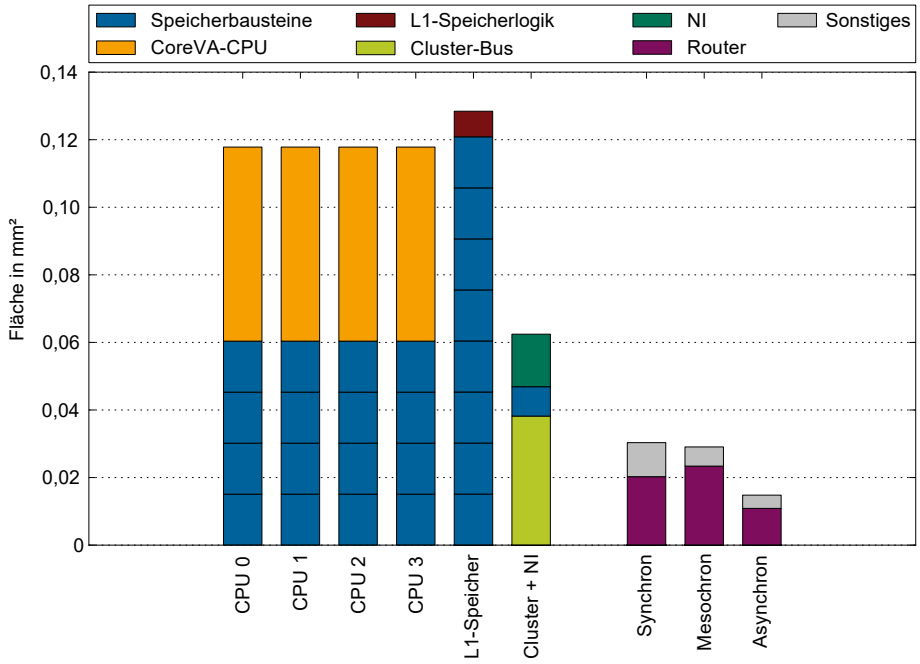
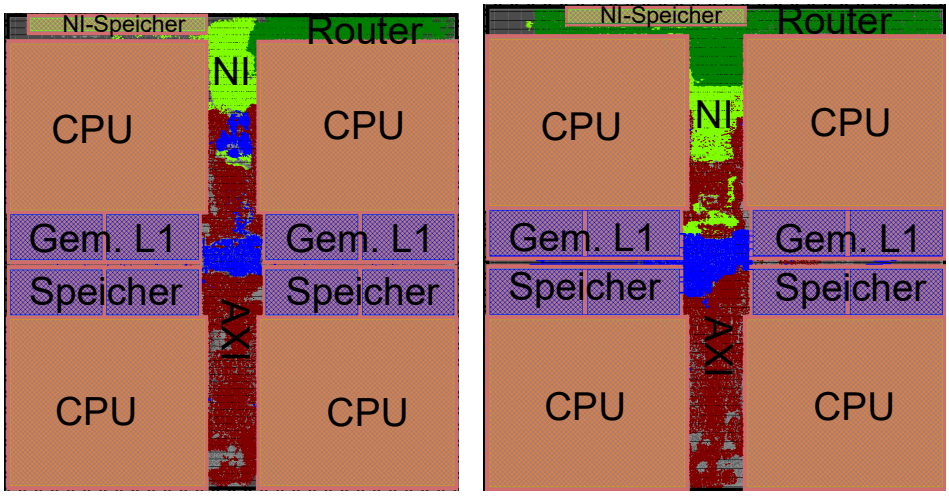


Abbildung 5.19: Flächenbedarf der CPUs, des Speichers, Cluster-Verbindungsstruktur, Netzwerk-Schnittstelle (NI) und der verschiedenen Router-Implementierungen.

In Diagramm 5.19 ist der Flächenbedarf eines Cluster-Knotens dargestellt. Der Flächenbedarf für CPUs, Speicher, Cluster-Verbindungsstruktur und Netzwerk-Schnittstelle ist für alle drei GALS-Implementierungen identisch und auf der linken Seite des Diagramms zu sehen. Auf der rechten Seite des Diagramms ist der Flächenbedarf der drei verschiedenen Router-Implementierungen zu sehen. In „Sonstiges“ sind alle Schaltungsteile enthalten, die nicht direkt einer Instanz zugeordnet werden können, wie beispielsweise der Taktbaum. Aufgrund des Fehlens eines Taktbaums und der Verwendung von Latches anstelle von Flip-Flops zur Zwischenspeicherung von Flits, benötigt der asynchrone Router nur 42 % der Fläche eines synchronen Routers. Wie erwartet, weisen der synchrone und der mesochrone Router nahezu die gleiche Fläche auf. Auf der einen Seite steigt der Flächebedarf des mesochronen Router durch den TCMS zwar leicht an. Auf der anderen Seite kann dieser Anstieg, durch die erlaubte Phasen-

verschiebung zwischen den NoC-Links und dem damit entspannteren Zeitverhalten, kompensiert werden.



(a) Asynchrones NoC. (b) Synchrones NoC.

Abbildung 5.20: Platziert und verdrahtetes Layout des asynchronen und synchronen Cluster-Knotens

Die Layouts für das synchrone und das asynchrone Design sind in Abbildung 5.20 zu sehen. Der Router ist im jeweiligen Design dunkel-grün markiert und zeigt visuell den Unterschied des Flächenbedarfs, der sich nach dem Platzieren und Verdrahten ergibt. Um die vier NoC-Links aufzubauen, die bei einem Cluster-Knoten an die vier Seiten des Makros geführt werden, ist es dem P&R-Werkzeug erlaubt in den oberen beiden Metalllagen über die CPU-Makros zu routen. Im Fall des synchronen und mesochronen Design hat das gesamte Layout eines Cluster-Knotens eine Fläche von 0,817 mm^2. Beim asynchronen Design verringert sich der Flächenbedarf eines gesamten Cluster-Knotens um 3,1 % auf 0,792 mm^2. Für größere Cluster (z.B. 8 oder 16 CPUs) ist kein großer Flächenzuwachs der NoC-Komponenten zu erwarten. Allein die Verzögerung des Request-Signals durch eine längere Inverterkette (siehe Abschnitt 5.3.3) und zusätzliche Treiber aufgrund längerer Leitungen für die NoC-Links, können zu einem Anstieg des Flächenbedarfs führen.

Leistungsaufnahme

Die Leistungsaufnahme der verschiedenen Router-Implementierungen wird mit Hilfe von Simulationen von zwei verbundenen Cluster-Knoten bestimmt. Zur Analyse der dynamischen Verlustleistung werden die Schaltaktivitäten während eines Pakettransfers über die verschiedene Router-Implementierungen aufgenommen. Die Schaltaktivitäten werden durch Simulationen auf Gatterebene der platzierten und verdrahteten Layouts durchgeführt. Mit Hilfe dieser Schaltaktivitäten können schließlich Power-Simulationen durch das Werkzeug Voltus von Cadence durchgeführt werden (siehe Kapitel 3.3), um die jeweilige Leistungsaufnahme zu bestimmen.

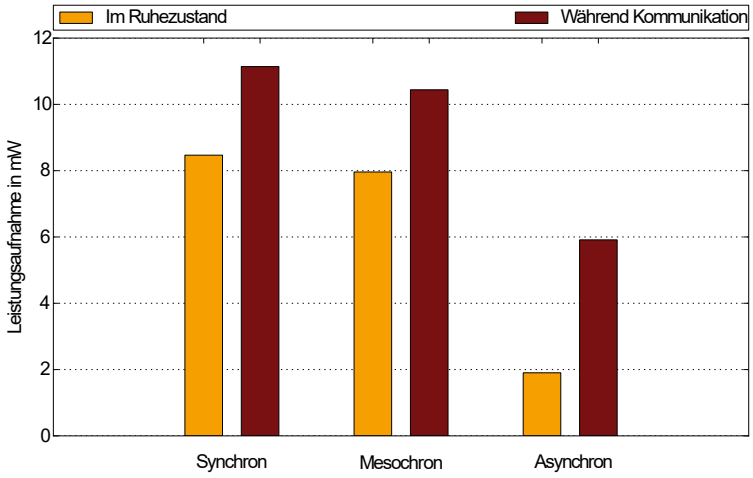


Abbildung 5.21: Vergleich der Leistungsaufnahme für die verschiedenen Router-Implementierungen

In Abbildung 5.21 ist die Leistungsaufnahme eines einzelnen Routers während des Ruhezustands und während einer aktiven Kommunikation dargestellt. Der synchrone Router verbraucht 4,23 mW im Ruhezustand und 5,57 mW während einer aktiven Paketübertragung. Beim mesochronen Design verringert sich die Leistungsaufnahme geringfügig auf 3,98 mW bzw. 5,21 mW. Aufgrund des fehlenden Taktsignals hat der asynchrone Router im Idle-Modus mit 0,94 mW nur noch 22,4 % der Leistungsaufnahme eines synchronen Routers. Den größten Teil dieser Leistung nehmen die beiden Synchronisierer zwischen Router und Netzwerk-Schnittstelle auf, die weiterhin Takt-basierte Schaltungselemente enthalten. Ein solcher Synchronisierer verbraucht 0,467 mW, wohingegen alle anderen Komponenten des Routers nur 0,056 mW benötigen. Während einer aktiven Kommunikation steigt die Leistungsaufnahme des asynchronen Routers

auf 2,94 mW an, welches 53 % der Leistungsaufnahme eines synchronen Routers entspricht. In diesem Fall verbraucht der aktive Synchronisierer zur Netzwerk-Schnittstelle 0,7 mW und alle anderen Router Komponenten 1,75 mW.

Latenz und Durchsatz

Bei einem synchronen Design sind sowohl die minimale Latenz als auch der maximale Durchsatz allein von der Taktfrequenz des Systems abhängig. Die maximale Taktfrequenz muss im synchronen Design vom gesamten System eingehalten werden und liegt, wie bereits oben erwähnt, bei etwa 700 MHz. Da bei allen Synthesen 1,42 ns für eine Taktperiode vorgegeben sind, liegt die Taktfrequenz jedoch bei 704 MHz, welches für einen fairen Vergleich zum asynchronen Design zu berücksichtigen ist. Im synchronen Router hat ein Flit eine Latenz von zwei Taktzyklen, um von einem Eingangsport zu einem anderen Ausgansport weitergeleitet zu werden. Somit ergibt sich eine minimale Latenz von 2,84 ns. Dies gilt allerdings nur, wenn keine Kollisionen mit konkurrierenden Flits auftreten. Der maximale Durchsatz eines uni-direktionalen NoC-Links liegt bei 704 MFlits/s, da während eines Taktzyklus über einen Link immer nur ein Flit verschickt werden kann. Das mesochrone Design ermöglicht den gleichen maximalen Durchsatz und weißt im besten Fall auch die gleiche minimale Latenz wie im synchronen Fall auf. Da die Takte im mesochronen System jedoch zueinander phasenverschoben sein können, kann die minimale Latenz eines Flits im schlimmsten Fall nahezu einen gesamten Taktzyklus länger dauern. Die minimale Latenz im mesochronen Router kann daher zwischen 2,84 ns und 4,26 ns variieren.

Im Vergleich zum synchronen und mesochronen Design ist der asynchrone Router vollkommen unabhängig von einem Taktsignal. Dies bedeutet, dass die minimale Latenz und der maximale Durchsatz nur von den Schaltungsverzögerungen der Logik- und Speicherelemente, Leitungsverzögerungen und der lokalen Handshake-Methode abhängen. Aufgrund von Variationen bei den Leitungs- und Gatterverzögerungen unterscheiden sich die Ergebnisse zwischen den verschiedenen I/O-Ports. Aus diesem Grund variiert auch die minimale Latenz zwischen 1,79 ns und 2,43 ns und der maximale Durchsatz zwischen 704 MFlits/s und 840,0 MFlits/s, je nach dem über welche I/O-Ports Flits gesendet werden. Die minimale Latenz und der maximale Durchsatz, der bei einem Pakettransfer über die verschiedenen I/O-Port des asynchronen Routers erreicht werden kann, ist in Tabelle 5.22 aufgeführt.

Eine Besonderheit bildet I/O-Port 0, bei dem der maximale Durchsatz durch die synchrone Netzwerk-Schnittstelle bestimmt wird und damit ebenfalls 704 MFlits/s aufweist. Insgesamt ist zu beobachten, dass vom asynchronen Router sowohl der maximale Durchsatz als auch die minimale Latenz des synchronen und mesochronen Systems in allen Fällen erreicht wird. Im Durchschnitt zeigt der Router des asynchronen NoCs einen um 15 % höheren maximalen Durchsatz und eine um 25 % geringere Latenz im Vergleich zu den NoCs die auf einem Takt basieren.

	Min. Latenz in ns				Max. Durchsatz in MFlits/s			
	Port 1	Port 2	Port 3	Port 4	Port 1	Port 2	Port 3	Port 4
Port 0	1.81	1.79	2.0	1.97	704	704	704	704
Port 1	-	1.93	2.17	2.20	-	816.4	818.0	797.3
Port 2	1.99	-	2.19	2.21	792.0	-	808.2	801.4
Port 3	2.13	2.11	-	2.34	820.9	817.8	-	825.1
Port 4	2.23	2.21	2.43	-	818.7	821.5	840.0	-

Abbildung 5.22: Minimale Latenz und maximaler Durchsatz, der über die verschiedenen I/O-Ports des asynchronen Routers erreicht werden kann.

Der asynchrone Router ist damit grundsätzlich performanter als die getakteten Systeme und kann theoretisch einen gleichwertigen bis höheren Durchsatz erreichen. Der maximale Durchsatz wird allerdings im idealen Fall, bei dem keine Paketkollisionen auftreten, vom synchronen Grundsystem (NI, CPUs) ausgebremst. Insbesondere unter voller Auslastung des NoCs kann jedoch vom asynchronen NoC profitiert werden. Um dieses zu untersuchen, werden von drei Eingangsport jeweils hundert Flits auf den verbliebenen vierten I/O-Port geschrieben. Der I/O-Port 0 wird bei diesem Stresstest aufgrund der Begrenzung durch den Takt außen vor gelassen. Durch die hiermit erzwungene Paketkollision konnte dem asynchronen NoC auch gerade bei hoher Auslastung ein höherer durchschnittlicher Durchsatz nachgewiesen werden. Sowohl der synchrone als auch der mesochrone Router benötigt zur Übertragung der 3x100 Flits 300 Taktzyklen und damit 426 ns, da nur ein Flit pro Taktzyklus vom Ausgangsport angenommen werden kann. Der asynchrone Router benötigt zur Übertragung der 3x100 Flits jedoch nur 230,95 ns - welches etwa 163 Taktzyklen entspricht - und damit nur 54,3 % der Zeit.

Globaler Taktbaum

In diesem Abschnitt wird der globale Taktbaum des gesamten MPSoC betrachtet. Dieser ergibt sich durch Verschaltung vieler Cluster-Knoten und gibt ein Maß über die Skalierbarkeit des Systems. Bei einer traditionell erstellten synchronen Schaltung steigt die Anzahl und Größe der benötigten Takttreiber mit der Anzahl von Schaltungsteilen, welches zu einem Anstieg der Fläche und einem höheren Energieverbrauch führt. Pullini et al. [143] haben außerdem gezeigt, dass sich bei steigender Skalierung von synchronen NoCs zusätzlich eine starke Reduzierung der maximalen Frequenz einstellt.

Verbesserungen und der Einsatz neuer Methoden in modernen Design-Werkzeugen bieten jedoch auch bei synchronen Architekturen neue Möglichkeiten der Skalierung. Moderne Design-Werkzeuge (siehe Kapitel 3.3) bieten den sogenannten "clock con-

# CPUs (NoC-Mesh)	Synchron		Mesochron	Asynchron
	trad.-CTS	CCOpt-flow		
16 (2x2)	0,39 mW	0,28 mW	0,27 mW	0,27 mW
64 (4x4)	2,55 mW	1,79 mW	1,47 mW	1,35 mW
256 (8x8)	-	7,83 mW	7,55 mW	5,78 mW

Abbildung 5.23: Leistungsaufnahme des globalen Taktbaumes verschiedener MPSoC-Größen und GALS-Methoden.

current optimization"(CCOpt) Entwurfsablauf, in dem Taktbaum und kombinatorische Logik gleichzeitig optimiert werden [39]. Dies ermöglicht es auch die Phasenverschiebung des Takts in den Griff zu bekommen, bzw. kann dieser teils sogar positiv genutzt werden (useful skew). Nichtsdestotrotz ermöglichen GALS-Methoden weiterhin eine bessere Energieeffizienz und weitere Vorteile, da diese teils vollkommen unabhängig von einer Phasenverschiebung sind. In diesem Abschnitt wird daher ein Vergleich des traditionellen synchronen Entwurfsablaufs (Clock Tree Synthesis (CTS)), dem neuen CCOpt-Entwurfsablauf und den verschiedenen GALS-Methoden durchgeführt.

Auf globaler Ebene werden viele Cluster-Knoten zusammengeschaltet, um ein 2D-Mesh NoC aufzubauen. Zusätzlich muss das Taktsignal durch den globalen Taktbaum zu all diesen Cluster-Knoten geführt werden. Im Falle des synchronen Designs muss dieser Taktbaum so angelegt werden, dass auf den NoC-Links zwischen den Cluster-Knoten ein korrektes Zeitverhalten für Setup- und Hold-Zeiten eingehalten werden. Im mesochronen und asynchronen Design ist die Erstellung des Taktbaumes hingegen entspannter, da diese vollkommenen unabhängig von der Phasenverschiebung des Taktsignals sind. Nichtsdestotrotz werden auch hier Takttreiber benötigt, um ein stabiles Taktsignal zu allen Cluster-Knoten zu führen.

Um die Skalierbarkeit der verschiedenen Entwurfsmethoden aufzuzeigen, wird ein fertiges Layout auf Top-Level eingesetzt. Verwendung findet hier der zuvor erwähnte Cluster-Knoten mit vier CPUs. In Tabelle 5.23 ist die Leistungsaufnahme des globalen Taktbaums für verschiedene MPSoC Größen und Entwurfsmethoden dargestellt. Die Ergebnisse beziehen sich ausschließlich auf die Leistungsaufnahme des globalen Taktbaums exklusive der internen Taktbäume innerhalb der Cluster-Knoten. Für das synchrone Design wird zum einen der traditionelle Entwurfsablauf (trad.-CTS), zum anderen der neue CCOpt-Flow eingesetzt. Die Ergebnisse zeigen eine relativ ähnliche Leistungsaufnahme für das Design unter Verwendung des CCOpt-Flow, und dem mesochronen Design. Der globale Taktbaum des asynchronen Designs weißt hingegen eine deutlich geringere Leistungsaufnahme auf (25 % geringer bei einem MPSoC mit 256 CPUs). Dies ist darauf zurückzuführen, dass hier bereits die internen Taktbäume der Cluster-Knoten kleiner sind, da die synchronen Schaltungsteile des NoC durch

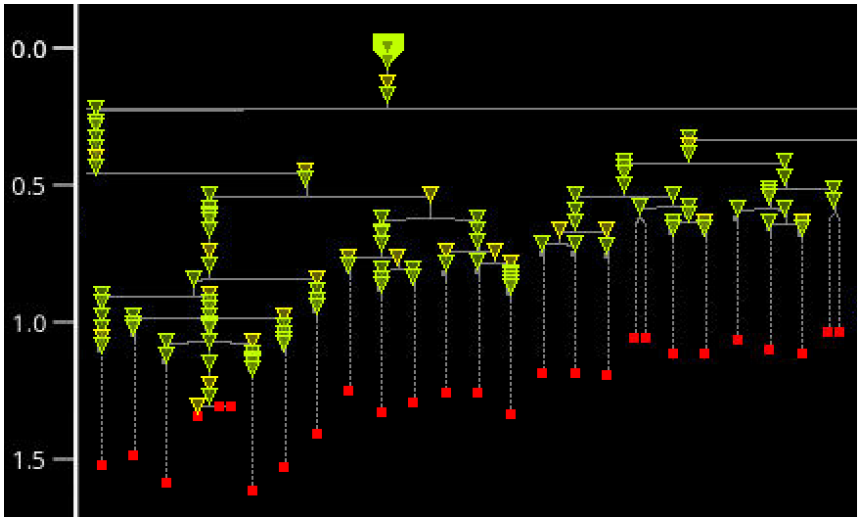


Abbildung 5.24: Ausschnitt des globalen Taktbaumes für ein 8x8 2D-Mesh CoreVA-MPSoC mit 256 CPUs

asynchrone ersetzt sind. Durch die kleineren internen Taktbäume muss der Taktbaum auf globaler Ebene weniger treiben.

Wird beim synchronen Design hingegen der traditionelle CTS-Flow verwendet, führt dies zu einer signifikant größeren Leistungsaufnahme des Taktbaumes. Hinzu kommt, dass hier auch die maximale Taktfrequenz immer schwerer zu erreichen ist, sodass MPSoCs mit über 256 CPUs nicht mehr erstellt werden können. Mit dem CCOpt-Flow ist hingegen eine weitere Skalierung des MPSoCs möglich, da die Phasenverschiebung des globalen Taktbaumes, die sich bei großen Chips einstellt, positiv ausgenutzt werden kann („useful clock skew“). Ein Ausschnitt des globalen Taktbaumes, unter Verwendung des CCOpt-Flow, ist in Abbildung 5.24 für ein MPSoC mit 256 CPUs und einem 8x8-Mesh-NoC (64 Cluster-Knoten) zu sehen. Neben allen Takttreibern (Gelb) und den angeschlossenen Cluster-Knoten (Rot), kann hier an der Y-Achse auch die maximale Phasenverschiebung des Taktes (in ns) gesehen werden, die sich über den Chip einstellt. Es zeigt sich, dass der Taktbaum auch beim synchronen Design nicht mehr vollkommen ausbalanciert sein muss und sich in diesem Beispiel eine Phasenverschiebung von 0,5 ns einstellen kann, die sich zwischen den am weitesten entfernten Cluster-Knoten ergibt. Dies bedeutet, dass auch unter Verwendung des CCOpt-Flow, ähnlich wie beim mesochronen und asynchronen Design, eine gewisse Phasenverschiebung über den Chip erlaubt ist.

Abgesehen vom synchronen Design im traditionellen Entwurfsablauf (trad.-CTS), konnte bei allen untersuchten Designs die maximale Taktfrequenz von 704 MHz der

Cluster-Knoten erreicht werden. Einzig beim synchronen Design unter Verwendung des CCOpt-Flow verringerte sich die maximale Taktfrequenz geringfügig um etwa 1 %

Abschließend lässt sich sagen, dass die Verwendung des CCOpt-Flow eine Skalierung eines synchronen MPSoC erlaubt, wobei geringfügige Einbußen bei der maximalen Taktfrequenz zu erwarten sind. Aus diesem Grund eignen sich das mesochrone und besonders das asynchrone NoC für ein effizientere Skalierung, erfordern jedoch einen höheren Designaufwand.

5.4 Zusammenfassung

In diesem Kapitel wurden verschiedene Verbindungsstrukturen für eingebettete NoC-Architekturen vorgestellt und diskutiert. Verbindungsstrukturen stellen die Kommunikation innerhalb des Netzwerks bereit und bestehen im Fall von NoCs insbesondere aus Routern und deren Verbindungen untereinander.

Zunächst wurden Architekturkonzepte von aktuellen Routern diskutiert, auf deren Basis die Architektur und Implementierung des im CoreVA-MPSoC eingesetzten Routers vorgestellt wurde. Beim Router des CoreVA-MPSoCs wird Wormhole-Switching in Verbindung mit einer *ON/OFF*-Flusskontrolle eingesetzt, da hierdurch bei geringem Ressourcenaufwand ein höherer durchschnittlicher Durchsatz erzielt werden kann als beim Circuit-Switching. Durch zusätzliche Methoden im CoreVA-MPSoC, wie dem Software-Kommunikationsmodell (siehe Kapitel 6.2) und der Netzwerk-Schnittstelle (siehe Kapitel 6.2) kann dennoch eine Ende-zu-Ende-Flußkontrolle garantiert werden. Zusätzlich bietet auch der CoreVA-Compiler durch die Berücksichtigung von Latenzschranken eine Unterstützung bei Anwendungen mit Echtzeitanforderungen (siehe Kapitel 7.5). Aufgrund der Einfachheit und gleichzeitigen Effizienz wird im CoreVA-MPSoC wie in vielen anderen ressourceneffizienten MPSoCs das XY-Routing eingesetzt. Gleichzeitig bietet XY-Routing durch das gedächtnislose und deterministische Verfahren eine ideale Voraussetzung für Echtzeitanwendungen und lässt sich zudem gut vom CoreVA-MPSoC-Compiler abschätzen. Der Router im CoreVA-MPSoC hat eine minimale Latenz von zwei Taktzyklen pro Flit.

Im Anschluss an die Router-Architektur wurden verschiedene Topologien zur Verschaltung der Router vorgestellt. Für den Router des CoreVA-MPSoC wurden die Topologien 2D-Mesh, Torus, Honeycomb und eine Ring-Topologie implementiert und analysiert. Es zeigte sich, dass sowohl die Chipfläche als auch die Leistungsaufnahme insbesondere durch die Anzahl der I/O-Ports pro Router bestimmt sind. Hierdurch haben Topologien wie das 2D-Mesh und Torus einen höheren Ressourcenbedarf als Honeycomb (74% vom 2D-Mesh) und Ring (52% vom 2D-Mesh). Bei der Performanz stellt sich bei drei von zehn Anwendungen die Ring-Topologie als nachteilig heraus und bei einer Anwendung die Honeycomb. Zum klassischen 2D-Mesh stellt Honeycomb, auch wegen des geringeren Ressourcenbedarfs, daher eine interessante Alternative dar.

Um den Ressourcenbedarf weiter zu verringern und auch die Skalierbarkeit des NoCs zu erhöhen, ohne auf Performanz zu verzichten, wurden im letzten Teil dieses Kapitels verschiedene GALS-Methoden untersucht. Verglichen wurden dazu die drei Ansätze eines synchronen, mesochronen und asynchronen NoCs. Für das mesochrone NoC wurden spezielle Synchronisierer zwischen den Links implementiert und eingesetzt. Beim asynchronen NoC wurden die Router komplett durch asynchrone Schaltungselemente realisiert. Die Ergebnisse haben gezeigt, dass durch moderne Funktionalitäten der Entwicklungswerkzeuge (CCOpt-Flow) auch für synchrone NoCs weiterhin eine gute Skalierung von MPSoCs möglich ist. Dennoch zeigte das asynchrone NoC gegenüber den anderen beiden Implementierungen einen geringeren Flächen- und Energiebedarf, bei vergleichbarer Performanz. Beim Vergleich eines platzierten und verdrahteten MPSoCs hat das asynchrone NoC einen um 3,1% geringeren Flächenbedarf für das Gesamtsystem. Die Leistungsaufnahme eines asynchronen Routers beträgt nur 22,4% (0,94 mW im Ruhezustand) bzw. 53% (3,94 mW während Kommunikation) von der Leistungsaufnahme eines Takt-basierten Routers. Im letzten Abschnitt des Kapitels wurde der globale Taktbaum für ein MPSoC mit 256 CPUs untersucht. Dieser zeigte für das synchrone und mesochrone NoC in etwa die gleiche Leistungsaufnahme von etwa 7,7 mW und beim asynchronen NoC mit 5,78 mW eine um 25% geringere Leistungsaufnahme. Hinzu kommt, dass beim synchronen NoC im Vergleich zu den anderen beiden Varianten eine um 2,7% geringere maximale Taktfrequenz erreicht wird.

6 Netzwerk-Schnittstellen für NoC-Architekturen

Neben der eigentlichen Verbindungsstruktur des NoCs (siehe Kapitel 5) hat auch die Schnittstelle vom Prozessorsystem zum NoC einen großen Einfluss auf die Ressourceneffizienz von NoC-Architekturen in MPSoCs. Die Netzwerk-Schnittstelle, welche in der Literatur überwiegend als Network-Interface (NI) bezeichnet wird, ist in Abbildung 6.1 farblich hervorgehoben. Im Falle des CoreVA-MPSoCs wird hier die adressbasierte Kommunikation innerhalb des CPU-Clusters auf die paketbasierte Kommunikation des NoCs umgesetzt.

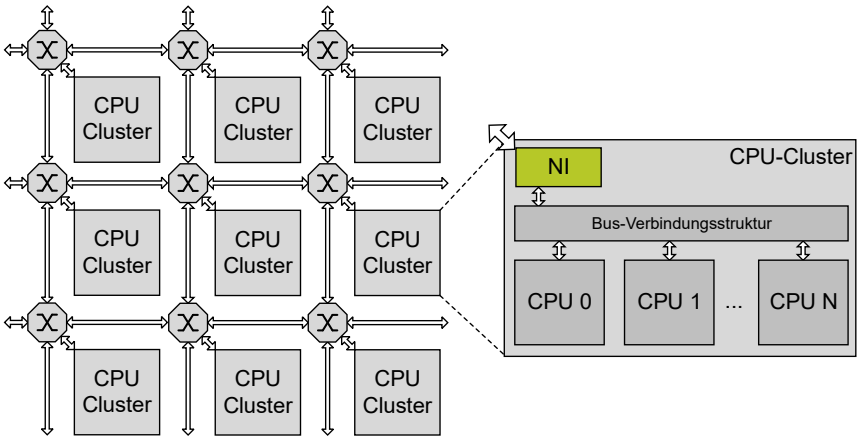


Abbildung 6.1: Netzwerk-Schnittstelle (NI) im CoreVA-MPSoC (farblich hervorgehoben)

In diesem Kapitel wird zunächst Bezug zum Stand der Technik von NI-Architekturen und den dazugehörigen Kommunikationsmodellen genommen (siehe Abschnitt 6.1). Anschließend werden in Abschnitt 6.2 die Architekturkonzepte und Konfigurationen vorgestellt, die für den NI des CoreVA-MPSoCs umgesetzt sind. Das Kapitel schließt mit einer Entwurfraumexploration verschiedener NI-Architekturen und -Konfigurationen (siehe Abschnitt 6.3). Teile der Ergebnisse dieses Kapitels sind in [201] veröffentlicht.

6.1 Stand der Technik von Netzwerk-Schnittstellen

Beim aktuellen Stand der Technik lassen sich NIs allgemein in zwei verschiedene Klassen unterteilen. Diese unterscheiden sich grundsätzlich durch verschiedene Konzepte bezüglich der Architektur bzw. dem zugrundeliegenden Kommunikationsmodell.

In der ersten Klasse von NIs wird eine paketbasierte Streaming-Schnittstelle zu den Routern eingesetzt. Dieser Ansatz ist insbesondere in den klassischen NoCs aus der Forschung verbreitet (siehe Abschnitt 2.2). In dieser Klasse besteht die Aufgabe des NIs darin, die adressbasierte Kommunikation der CPUs in die paketbasierte Kommunikation des NoCs zu überführen. Dazu werden sowohl die Paketköpfe als auch die Nutzdaten direkt in FIFOs oder Ringspeicher innerhalb des NIs abgelegt. Der NI teilt Pakete anschließend typischerweise in kleinere Segmente – sogenannte Flits – auf und versendet diese über das NoC. Flits des gleichen Pakets können auf der Empfangsseite durch eine Flusskontrolle identifiziert werden. Der Vorteil einer paketbasierten Kommunikation über das NoC ist das Erreichen einer hohen Nettodatenrate innerhalb der NoC-Verbindungsstruktur, da vergleichsweise wenige Kontrolldaten übertragen werden müssen. Beispiele dieser Klasse sind die NIs des AEthereal NoC [146], des Spidergon-STNoC [156] und des GigaNetIC [127]. Im AEthereal NoC bietet der NI eine FIFO-basierte Schnittstelle, um Anfragen und Nutzdaten über das NoC zu verschicken. Dazu hat jede CPU, bzw. der Cache einer CPU, eine Master-/Slave-Schnittstelle, sodass Datenanfragen an einen gemeinsamen Speicher durchgeführt werden können. Diese Anfragen werden schließlich vom gemeinsamen Speicher beantwortet, indem Nutzdaten im Rahmen von Paketen zurückgeschickt werden. Einen ähnlichen Ansatz bietet der NI des Spidergon-STNoC. Auch hier ist der NI als Master-/Slave-Schnittstelle zu sehen.

Generell haben die FIFO-basierten Ansätze einen Nachteil, wenn ein wahlfreier Speicherzugriff (engl. Random Memory Access) von CPUs auf einen lokalen Scratchpad-Speicher gewünscht ist. Ist ein wahlfreier Speicherzugriff auf die Nutzdaten eines Pakets nötig, müssen ankommende Paketdaten zunächst von den FIFOs in die lokalen Speicher der CPUs kopiert werden bzw. beim Versenden aus dem lokalen Speicher in die FIFOs des NIs übertragen werden. Dies führt auf den CPUs zu Laufzeitkosten in Software oder erfordern spezielle Hardwareeinheiten wie DMA[1]-Steuergeräte (DMA-Controller). Die zuvor genannten NIs eignen sich daher nur für Cache-basierte MPSoCs gut, da hier ganze Cachezeilen direkt an die passenden Stellen gespeichert werden können und keine zusätzlichen Softwarekosten entstehen. Untersuchungen in [110] und [92] haben jedoch gezeigt, dass in NoCs die Kombination aus DMA-Controllern und lokalen Scratchpad-Speichern eine Cache-basierte Kommunikation leistungsmäßig übertreffen. Der NI des Argo-NoC [169] integriert einen DMA-Controller, in dem Paketinformationen (z.B. Routing-Koordinaten, Schreib-/Lese-Zeiger und die Datenmenge) in einer DMA-Tabelle abgelegt werden können. Mit Hilfe dieser Tabelle unterstützt der NI mehrere

[1] *Direct Memory Access*

DMA-Kanäle. Ein NI mit zwei unabhängigen DMA-Kanälen wird im STHORM-NoC eingesetzt [16].

Die zweite Klasse von NI-Architekturen verwendet einen gemeinsamen globalen Adressraum im gesamten MPSoC. Hier werden Daten nicht als Pakete durch das NoC versendet, sondern CPU-Zugriffe werden zusammen mit der Adresse direkt an das NoC übergeben. Dies verringert jedoch den Durchsatz reiner Nutzdaten über das NoC, da in jedem Flit die Adresse enthalten sein muss. In der Adresse des Zugriffs sind entsprechende Informationen, wie z.B. die Routing-Koordinaten, hinterlegt. Diese Klasse von NI-Architekturen ist insbesondere in kommerziellen NoCs verbreitet, wie z.B. in Adaptevas Epiphany [2]. Der globale Adressraum ermöglicht den CPUs einen wahlfreien Zugriff auf alle Speicher im MPSoC. Dabei können Speicherzugriffe über das NoC direkt durch einzelne Speicheroperation der CPU durchgeführt werden. Zusätzlich können DMA-Controller die CPUs von Speichertransfers entlasten. Ein Nachteil dieses Ansatzes ist jedoch die Beschränkung der Skalierbarkeit des MPSoCs durch den globalen Adressraum. Dies kann jedoch durch die Verwendung von 64-bit-CPUs umgangen werden, wie es z.B. beim Epiphany mit 1024 CPUs [134] umgesetzt ist. Des Weiteren verringert sich durch das Versenden der gesamten Speicheradresse der Anteil von Nutzdaten innerhalb eines Flits, welches wiederum den realen Durchsatz des NoCs verringert.

Im hierarchischen MPSoC von Kalray (MPPA [45]) wird hingegen kein globaler Adressraum auf NoC-Level unterstützt. Hier wird lediglich innerhalb eines CPU-Clusters ein gemeinsamer Adressraum verwendet. Tasks eines Threads können daher nur innerhalb eines Clusters miteinander kommunizieren. Ein Austausch von Daten über das NoC wird hingegen nur zwischen Prozessen unterstützt (IPC). Detaillierte Informationen wie dieses umgesetzt wird, sind allerdings nicht verfügbar.

Eine Kombination beider Klassen von NI-Architekturen bieten die kommerziellen NoC-Anbieter wie Arteris (FlexNoC), Sonics (SonicsGN) oder NetSpeed (Orion) [63]. Die NIs dieser NoCs bieten Schnittstellen zu konventionellen Bus-Standards wie AXI. Unabhängige Transaktionen dieser Bus-Standards werden in Pakete umgewandelt und über das NoC verschickt. Leider sind hier keine detaillierten Informationen über die Architektur verfügbar.

Der NI des CoreVA-MPSoC, dessen Architektur im folgenden Abschnitt im Detail vorgestellt wird, kombiniert ebenfalls beide Ansätze. Auch hier findet eine paketbasierte Kommunikation über das NoC statt, während der NI eine Adress- und Speicher-basierte DMA-Funktionalität bietet. Um eine hohe Skalierbarkeit und ein effizientes MPSoC zu gewährleisten, unterstützt der NI des CoreVA-MPSoC zur Laufzeit eine flexible Verwaltung unabhängiger Transaktionskanäle.

6.2 Architektur einer Netzwerk-Schnittstelle für das CoreVA-MPSoC

Die Netzwerk-Schnittstelle innerhalb des CoreVA-MPSoCs ermöglicht die Kommunikation zwischen zwei CPUs, die sich in verschiedenen CPU-Clustern befinden. Jeder CPU-Cluster kann durch eine eindeutige X- und Y-Koordinate in der entsprechenden Topologie des NoC identifiziert werden. Innerhalb eines CPU-Clusters wird ein gemeinsamer Adressraum verwendet, um alle Komponenten (CPUs, Speicher etc.) anzusprechen. Die Aufgabe des NIs ist es, die adressbasierte Kommunikation innerhalb des CPU-Clusters in die Fluß- und paketbasierte Kommunikation des NoCs zu überführen.

Die Architektur von Netzwerk-Schnittstellen und das Programmier- bzw. Kommunikationsmodell von MPSoCs hängen sehr stark zusammen. Für eine hohe Effizienz der Kommunikation im Gesamtsystem muss daher beides aufeinander abgestimmt sein. Um die Programmierbarkeit innerhalb des CoreVA-MPSoCs sowohl einfach als auch effizient zu gestalten, basiert die Funktionalität des NIs auf dem CoreVA-MPSoC-Kommunikationsmodell, welches bereits in Kapitel 3.4.3 vorgestellt ist. Aus Sicht des Programmierers verhält sich die Kommunikation über das NoC sehr ähnlich wie die Kommunikation innerhalb eines CPU-Clusters. Des Weiteren führt die Synchronisation der Datenblöcke, wie sie im Kommunikationsmodell verwendet wird, automatisch zu einer Ende-zu-Ende-Flußkontrolle, sodass auf Software- und NI-Ebene keine Verklemmungen (engl. Deadlocks) im NoC herbeigeführt werden kann. Diese Ende-zu-Ende-Flußkontrolle wird sichergestellt, indem ein Paket erst versendet wird, wenn der Speicherbereich für dieses Paket im Ziel-Cluster freigegeben wird.

Um die Task-Parallelität einer Anwendung effizient auszunutzen, ist es wichtig die Kosten für eine Kommunikation zwischen CPUs möglichst gering zu halten. Dies betrifft nicht nur die Kosten des eigentlichen Datentransfers innerhalb der Hardware-Verbindungsstruktur, sondern auch die Softwarekosten, welche von den beteiligten CPUs aufgewendet werden müssen. Die Aufgabe des NIs ist es daher die CPUs vom eigentlichen Versenden von Paketen zu entlasten, um so die Softwarekosten für Kommunikation zu minimieren. Um dies zu erreichen legt der NI des CoreVA-MPSoCs die Paketdaten direkt an die Stelle im lokalen Datenspeicher der CPU ab, an der die Daten direkt ohne Kopiervorgang von der Anwendung verwendet werden können. Paketdaten müssen von der Ziel-CPU daher nicht aufwendig in Empfang genommen und zugeordnet werden, wie es beispielsweise bei den NI-Architekturen des AEthereal NoC, Spidergon-STNoC und GigaNetIC der Fall ist (vgl. Abschnitt 6.1). Des Weiteren profitiert die CPU von der geringen Lese-Latenz des lokalen Speichers. Ein ähnlicher Mechanismus wird auch zum Versenden von Paketdaten verwendet. Der NI verhält sich daher wie ein DMA-Controller, der Pakete nebenläufig zur CPU-Verarbeitung versendet und empfängt.

Das Blockschaltbild in Abbildung 6.2 veranschaulicht die Architektur des NIs mit den enthaltenen Teilkomponenten. Dessen Funktionalität und Zusammenspiel wird im Folgenden genauer beschrieben.

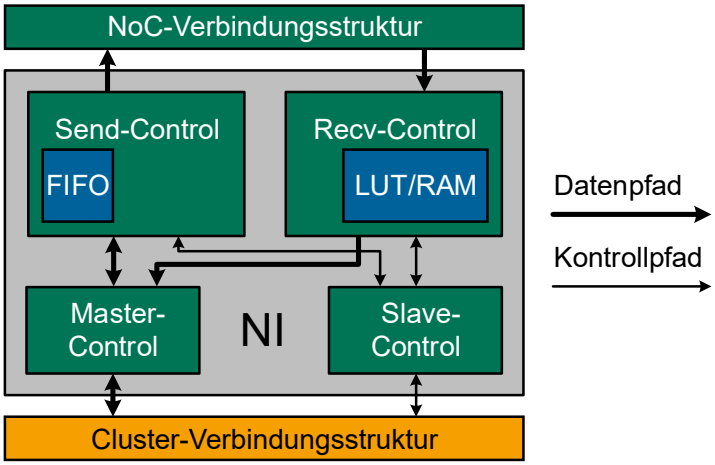


Abbildung 6.2: Architektur der Netzwerk-Schnittstelle (NI) im CoreVA-MPSoC

6.2.1 Cluster-Schnittstelle

Der NI ist über eine Master- und Slave-Schnittstelle direkt mit der Verbindungsstruktur des Clusters verbunden. Als Cluster-Verbindungsstruktur wird im Zusammenhang mit dem NoC der AMBA-AXI-Standard eingesetzt (siehe Abschnitt 7.2) und kann sowohl als geteilter Bus oder als Crossbar konfiguriert werden. Eine weitere Konfigurationsmöglichkeit ist die Datenbreite der AXI-Verbindungsstruktur, die entweder auf 32 bit oder 64 bit gesetzt werden kann.

Die Master-Schnittstelle (Master-Control) ermöglicht dem NI einen direkten Zugriff auf alle Komponenten (wie CPUs, Speicher, etc.) innerhalb des Clusters. Verwendung findet die Master-Schnittstelle sowohl beim Versenden als auch beim Empfangen von Paketen. Kommt ein Paket über das NoC an, werden die Daten ankommender Flits direkt in die lokalen Speicher der CPUs geschrieben. Beim Senden wiederum werden die Paketdaten per Burst-Zugriff direkt aus dem lokalen Speicher der CPUs gelesen. Bei einem Burst-Zugriff hat ein Master für mehrere Taktzyklen Zugriff auf den Bus und kann kontinuierlich Daten von aufeinander folgenden Adressen lesen, ohne unterbrochen zu werden. Da der Schreib- und Lesekanal im AXI-Standard getrennt ist, kann der Empfangs- und Sendevorgang parallel durchgeführt werden (siehe Abschnitt 7.2.1).

Über die Slave-Schnittstelle (Slave-Control) kann der NI von jeder CPU im Cluster angesprochen und konfiguriert werden. Dazu entscheidet die Slave-Control anhand der Adresse des Bus-Zugriffs, um welchen Zugriffstyp es sich handelt. Anschließend wird der Zugriff an die entsprechende Komponente des NIs zu übergeben.

Tabelle 6.1: Kanalinformationen für die Netzwerk-Schnittstelle (NI) zum Versenden eines Pakets

Name	Typ. Größe	Beschreibung
X	3 bit	X-Koordinate des Ziel-Clusters (hier für ein 8x8 NoC)
Y	3 bit	Y-Koordinate des Ziel-Clusters (hier für ein 8x8 NoC)
NUMBER_OF_FLITS	8 bit	Die Anzahl zu versendender Flits, um den gesamten Datenblock des Kanals zu versenden (hier maximal 256, 2 kB)
DATA_POINTER	32 bit	Zeiger auf die Speicherstelle des Datenblocks, der versendet werden soll
MUTEX_POINTER	32 bit	Zeiger auf die Speicherstelle des Mutex, um ein erfolgreiches Versenden des Kanals zu signalisieren (Synchronisation)

6.2.2 Sendevorgang

Für den Sendevorgang von Paketen ist insbesondere die Komponente Send-Control verantwortlich (siehe Abbildung 6.2). Ein Paket ist mit dem Datenblock eines Kanals des Kommunikationsmodells aus Kapitel 3.4.3 gleichzusetzen. Sobald die zu sendende CPU einen Datenblock produziert hat, übergibt diese einen Sendebefehl über die Slave-Schnittstelle an den NI. Für die CPU besteht dieser Sendebefehl aus einem einzelnen 32-bit-Schreibzugriff, welcher die Softwarekosten zum Versenden von Paketen minimal hält. Innerhalb des NIs wird jeder Sendebefehl im FIFO der Komponente Send-Control abgespeichert. Durch die Verwendung eines FIFOs ist es möglich, parallele Sendebefehle verschiedener CPUs anzunehmen, ohne ein Blockieren von CPUs herbeizuführen. Die Sendebefehle innerhalb des FIFOs werden sukzessiv von der Send-Control abgearbeitet. Ein Sendebefehl besteht aus einem 32-bit-Zeiger (engl. Pointer) auf eine bestimmte Stelle innerhalb des lokalen Speichers der CPU. An dieser Stelle sind weitere Informationen des Kanals gespeichert, die während des Kanalaufbaus von der CPU dort abgelegt werden. Der nur aus 32 bit bestehende Sendebefehl minimiert damit sowohl die Softwarekosten der CPU als auch den Flächenbedarf des NI. Ressourcen im NI können eingespart werden, da sich die eigentlichen Kanalinformation der verschiedenen Kanäle weiterhin im lokalen Speicher der CPU befinden. In Tabelle 6.1 sind die Informationen aufgeführt, die zum Versenden eines Pakets bzw. Kanals nötig sind.

Zu Beginn eines Sendevorgangs liest der NI über die Master-Schnittstelle diese Kanalinformationen und speichert diese zwischen. Nach Erhalt dieser Informationen beginnt die Send-Control die Daten aus der entsprechenden Speicherstelle zu lesen, um

diese anschließend Flit für Flit über die NoC-Verbindungsstruktur zu versenden. Einem Flit werden dazu neben den Daten auch die entsprechenden Header-Informationen wie z.B. die X- und Y-Zielkoordinate angefügt. Bei einem erfolgreichem Burst-Lese-Transfer über die Cluster-Verbindungsstruktur kann in jedem Takt ein Flit mit 64-bit-Nutzdaten versendet werden, sofern der 64-bit-AXI-Datenbus verwendet wird. Ist hingegen der 32-bit-AXI-Datenbus konfiguriert, ist ein Versenden eines Flits mit 64-bit-Nutzdaten nur in jedem zweiten Takt möglich, sodass sich dadurch die maximal mögliche Bandbreite des NIs halbiert. Sobald alle Daten des Kanals versendet sind, ist der Sendevorgang abgeschlossen und die Send-Control setzt per Schreibzugriff über die Master-Schnittstelle den Mutex des Kanals. Anschließend kann der nächste Sendebefehl innerhalb des FIFOs abgearbeitet werden.

6.2.3 Empfangsvorgang

Beim Empfangsvorgang werden Flits behandelt, die über die NoC-Verbindungsstruktur am NI eintreffen. Hauptkomponente dieses Vorgangs ist die Recv.-Control (siehe Abbildung 6.2), welche eingehende Flits entgegennimmt und diese ihrer Paket- bzw. Kanalzugehörigkeit zuordnet.

Im Gegensatz zum Senden stellt sich das Empfangen von Flits als komplexer dar, da sich aufgrund des im NoC des CoreVA-MPSoCs verwendeten Wormhole-Switching die Flits verschiedener Pakete vermischen können (siehe Kapitel 5). Dies ist auch der große Unterschied anderer DMA-basierter NIs, die nur das Circuit-Switching unterstützen, wie z.B. beim Argo-NoC. Beim Wormhole-Switching müssen ankommende Flits nicht nur einmal zu Beginn für das gesamte Paket zugeordnet werden, sondern jedes Flit einzeln. Um die Flits einem Paket bzw. Kanal zuzuordnen, besitzt jedes Flit einen Identifikator (ID) zur Flußkontrolle, der für alle Flits eines Pakets gleich sein muss. Des Weiteren muss diese sogenannte FLOW_ID innerhalb eines Clusters eindeutig einem Kanal zuzuordnen sein. Mit dieser eindeutigen FLOW_ID kann die Recv.-Control nun weitere Informationen über den Kanal anfordern, um die Daten des ankommenden Flits an der passenden Stelle im lokalen Speicher abzulegen.

Ein Ansatz zur Anforderung der Kanalinformationen ist es, diese aus dem lokalen Speicher der entsprechenden CPU zu lesen. Dieser Ansatz ist analog zum Sendevorgang und hat neben der Einsparung von Ressourcen (Speicher), auch den Vorteil einer höheren Flexibilität bei der Anzahl möglicher Kanäle. Als entscheidender Nachteil dieses Ansatzes ist jedoch zu beachten, dass diese Informationen für jedes einzelne Flit erneut angefordert werden müssten. Im Gegensatz zum Sendevorgang kann aufgrund des Wormhole-Switchings nicht davon ausgegangen werden, dass Filts eines Pakets kontinuierlich ankommen. Ein Anfordern von Kanalinformationen kann dadurch in jedem Takt nötig werden, wodurch die Performanz des gesamten NoCs stark eingeschränkt würde.

Ein erfolgversprechenderer Ansatz ist es daher, die Kanalinformationen direkt im NI vorzuhalten. Dazu wird der NI um einen Speicher erweitert, in dem jede CPU zum

Tabelle 6.2: Kanalinformationen für die Netzwerk-Schnittstelle (NI) zum Versenden eines Pakets

Name	Typ. Größe	Beschreibung
NUMBER_OF_FLITS	8 bit	Die Anzahl an Flits, die angenommen werden müssen bis das Paket bzw. der Datenblock des Kanals vollständig ist
DATA_POINTER	32 bit	Zeiger auf die Speicherstelle, an die der ankommende Datenblock des Kanals abgelegt werden soll
MUTEX_POINTER	32 bit	Zeiger auf die Speicherstelle des Mutex, um den vollständigen Empfang des gesamten Datenblocks zu signalisieren (Synchronisation)

Kanalaufbau die entsprechenden Kanalinformationen ablegt. Die Recv-Control nutzt diesen Speicher als Umsetzungstabelle, welche auch als Lookup-Tabelle (LUT) bezeichnet wird. Als Schlüssel für einen Eintrag dieser LUT dient die eindeutige FLOW_ID, um dynamisch die für ein ankommendes Flit passenden Informationen aus der LUT zu erhalten. In Tabelle 6.2 ist der Inhalt eines einzelnen LUT-Eintrags zu sehen, der jeweils einer eindeutigen FLOW_ID zugeordnet ist.

Die LUT kann entweder als SRAM-Speicher oder mit Hilfe von Registern implementiert werden. Register bieten den Vorteil, die Informationen mit einer geringeren Latenz zu erhalten, da diese noch im selben Takt des ankommenden Flits zur Verfügung stehen. Bei einer hohen Anzahl von Kanälen, die eine LUT parallel bereitstellen kann, müssen entsprechend viele Daten in Form von Kanalinformationen gespeichert werden. Dies führt bei einer Implementierung mit Registern jedoch zu einem sehr hohen Ressourcenbedarf an Chipfläche und Energie. SRAM-Speicher ist im Vergleich zu Registern sehr kompakt und energiesparsam, erhöht jedoch die Latenz der ankommenden Flits um einen Takt, da Informationen des zugehörigen Kanals erst einen Takt später zur Verfügung stehen. Zu beachten ist jedoch, dass die Anzahl von LUT-Einträgen der Anzahl unabhängiger NI-Empfangskanäle entspricht, die von Tasks nebenläufig genutzt werden können. Eine Diskussion, welche Anzahl von Kanälen sich am besten für das CoreVA-MPSoC eignet, wird in den Abschnitten 6.3.1 und 6.3.2 durchgeführt.

6.2.4 Router-Schnittstelle

Die Schnittstelle zum angebundenen Router besteht aus zwei Ports. Der Input-Port ist für die Entgegennahme ankommender Flits verantwortlich und der Output-Port übergibt zu sendende Flits vom NI an den Router. Da der Input-Port das gleiche Protokoll wie die Ports der Router umsetzen muss, besteht auch dieser aus einem FIFO mit 5

Einträgen und Allmost-Full-Funktionalität (vgl. Kapitel 5.1). Durch das FIFO erhält ein ankommendes Flit eine zusätzliche Latenz von einem Takt, bis es vom Input-Port an die Komponente Recv-Contol übergeben wird.

Auch der Output-Port muss ausgehende Flits in das Protokoll der Router überführen. Wie bei der Routing-Komponente des Routers muss hier bereits das Routing analysiert werden, um zu prüfen ob das Flit vom nächsten Router entgegengenommen werden kann. Zur Entkopplung von NI und Router bezüglich des kritischen Pfades der Schaltungsteile, wird das Flit vor Übergabe an den Router in einem Register zwischengespeichert.

6.3 Entwurfsraumexploration von Netzwerk-Schnittstellen

Um die im vorherigen Abschnitt vorgestellten Architekturkonzepte und Konfigurationen von NIs für das CoreVA-MPSoC zu bewerten, wird in diesem Abschnitt eine Entwurfsraumexploration durchgeführt. Die Bewertung findet anhand der in Kapitel 4 vorgestellten Bewertungsmaße statt.

Zunächst werden die für das Kommunikationsmodell notwendigen Softwarekosten betrachtet, die sich durch NoC-Kommunikation und der entsprechenden NI-Architektur ergeben. Als Softwarekosten wird die Laufzeit von CPUs bezeichnet, die diese explizit für die Kommunikation aufbringen müssen.

In den letzten beiden Abschnitten dieses Kapitels wird eine Entwurfsraumexploration durchgeführt, um eine optimale Dimensionierung für den NI des CoreVA-MPSoC zu finden. Dies betrifft insbesondere die Dimensionierung der physikalischen Kommunikationskanäle in der Empfangsstruktur.

6.3.1 Softwarekosten

In der Literatur beschränkt sich die Betrachtung der Performanz von NoCs häufig auf die Kosten eines Datentransfers, die sich innerhalb der Hardware der NoC-Kommunikationsstruktur ergeben [146][154][180]. Diese Kosten beinhaltet typischerweise die Latenz von Paketen, die durch Hardwarekomponenten wie Router und NI hervorgerufen wird. Die tatsächlichen Kosten eines Datentransfers zwischen CPUs müssen jedoch auf Systemlevel betrachtet werden, da auch auf den CPUs zusätzliche Kosten in Form von CPU-Taktzyklen für den Datenaustausch aufgewendet werden müssen. Generell entstehen in Multiprozessorsystemen Softwarekosten durch die Synchronisierung und der Verwaltung von Datenblöcken. Bei nachrichtenbasierter Kommunikation ergeben sich weitere Softwarekosten. So muss die sendende CPU den Nachrichtentransfer initiieren und die empfangende CPU diesen terminieren.

Um diese Softwarekosten zu bestimmen, wird ein synthetischer Benchmark eingesetzt. Dieser bestimmt neben den Hardwarelatenzen auch die Softwarekosten, die sich durch eine CPU-zu-CPU-Kommunikation ergeben. Dazu werden zwei Tasks auf zwei CPUs verschiedener Cluster ausgeführt. Der eine Task produziert einen Datenblock und sendet diesen periodisch per NoC-Kanal zum anderen Task. In Abbildung 6.3 ist eine abstrakte Sequenz für das Senden und Empfangen eines Datenblocks dargestellt. Zusätzlich sind in dem Sequenzdiagramm die Softwarekosten der Funktionen für die Synchronisierung und Verwaltung der Datenblöcke dargestellt. Dies sind die Funktionen getWriteBuf, setWriteBuf, getReadBuf und setReadBuf des Kommunikationsmodells, welche bereits in Abschnitt 3.4.3 vorgestellt sind. Die Kosten dieser Funktionen sind von der Größe des zu versendenden Datenblocks unabhängig. In diesen Funktionen werden die Datenblöcke verwaltet und die Kommunikation mit dem NI für das Senden und Empfangen des Datenblocks durchgeführt. Es ist von entscheidender Bedeutung die Kosten dieser Funktionen möglichst gering zu halten, da während ihrer Ausführung die CPU nicht ihre eigentliche Arbeit verrichten kann. Im Gegensatz dazu können die Latenzen, welche sich während des Datentransfers durch die Hardwarekomponenten ergeben, typischerweise durch Methoden wie Double- oder Multi-Buffering versteckt werden. Diese Kommunikationslatenzen werden in Abbildung 6.3 durch den Datentransfer-Pfeil dargestellt.

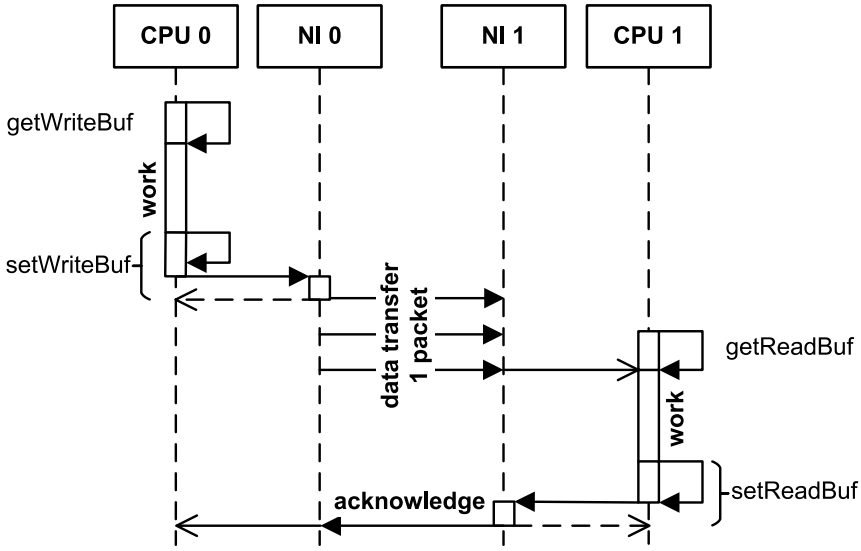


Abbildung 6.3: Sequenzdiagramm des NI.

Die Funktionsweise des NIs hat eine Auswirkung auf die Laufzeit der Funktionen des Kommunikationsmodells. Bei NoCs, in denen die CPU die Paketverwaltung übernimmt,

	Semi-Static	dynamic	cluster
getWriteBuf	15	17	15
setWriteBuf	28	126	19
getReadBuf	15	15	15
setReadBuf	19	19	19
data transfer (16 B)	26	148	20
data transfer (1 kB)	177	285	20

Tabelle 6.3: Softwarekosten und Latenz in Taktzyklen für die Verwendung eines semi-statischen und dynamischen NoC-Kanals sowie die des Cluster-Kanals.

sind die Laufzeiten dieser Funktionen entsprechend hoch, da Daten ständig aus FIFOs entgegengenommen oder aus Paketspeichern kopiert werden müssen. Bei NIs mit DMA-Funktionalität, wie z.B. beim NI des CoreVA-MPSoCs, wird die Verwaltung der Paketdaten durch die Hardware übernommen. Für die DMA-basierte Kommunikation gibt es zwei verschiedene Ansätze mit der Limitierung von physikalisch vorhandenen Kanälen (DMA-Kanäle bzw. NoC-Kanäle) umzugehen. Beide Ansätze haben unterschiedliche Auswirkungen auf die Kosten für Kommunikation.

Der erste Ansatz ist eine Limitierung der Anwendung bzw. der Verteilung der Tasks, um nur so viele Kommunikationskanäle zu nutzen wie physikalisch vorhanden sind. Der Nachteil dieses Ansatzes ist ein geringerer Freiheitsgrad bei der Verteilung von Tasks, was zu einer geringeren Performanz führen kann. Der Vorteil besteht jedoch darin, dass Kanäle nur einmal während einer Initialisierungsphase konfiguriert werden müssen. Anschließend können diese vorkonfigurierten Kanäle periodisch von der Anwendung für ein und demselben Kommunikationskanal zwischen Tasks verwendet werden. Dieser Ansatz wird als semi-statisch bezeichnet, da Kanäle über einen längeren Zeitraum statisch sind, bei Anwendungswechsel durch Neuinitialisierung jedoch wieder umkonfiguriert werden können.

Der zweite dynamischere Ansatz limitiert die Anzahl Kommunikationskanäle zwischen Tasks nicht. In dieser Methode konkurrieren Tasks um einen physikalischen Kanal. Bekommt ein Task den Zugriff, kann dieser den Kanal für die Dauer eines Datentransfers nutzen. Im Anschluss daran muss der physikalische Kanal jedoch wieder frei gegeben werden, um für andere Tasks nutzbar zu sein. Somit besteht prinzipiell keine Einschränkung bei der Verteilung von Tasks, da theoretisch unendlich viele logische Kommunikationskanäle zwischen Tasks möglich sind. Der Nachteil bei diesem Ansatz ist jedoch, dass ein physikalischer Kanal vor jedem Datentransfer erneut aufgebaut werden muss. Beim CoreVA-MPSoC muss z.B. der entsprechende Kanal im empfangenden NI konfiguriert werden. Um die Synchronisation zu garantieren, muss diese Rekonfiguration durch den sendenden Task vorgenommen werden.

Um den Unterschied der beiden Ansätze aufzuzeigen, wird der zuvor beschriebene synthetische Benchmark eingesetzt, der die Softwarekosten und die Latenz für eine CPU-zu-CPU-Kommunikation bestimmt. In Tabelle 6.3 sind die Ergebnisse dieses Benchmarks aufgeführt. Dabei handelt es sich um die verschiedenen Softwarekosten, die bei dem Transfer eines Datenblocks über einen NoC-Kanal entstehen. Zusätzlich zu den beiden Ansätzen eines NoC-Transfers werden die Kosten für eine CPU-zu-CPU-Kommunikation innerhalb eines Clusters gezeigt. Denn genau wie bei einem NoC-Transfer muss auch bei der Kommunikation innerhalb des Clusters der Transfer von Datenblöcken verwaltet und synchronisiert werden. Der größte Unterschied zwischen den Transfer-Typen besteht in der Funktion setWriteBuf, welche das Senden eines Datenblock initiiert. Da im dynamischen Ansatz des NoC-Transfers der NI für den Transfer jedes neuen Datenblocks rekonfiguriert werden muss, erhöht sich die Laufzeit der Funktion setWriteBuf um einen Faktor von 4,5 im Vergleich zum semi-statischem Ansatz.

Neben den Softwarekosten wird in Tabelle 6.3 auch die Latenz des eigentlichen Datentransfers aufgeführt. Angegeben ist die Latenz aus Sicht der Anwendung und enthält neben Softwarekosten auch die Latenz, welche sich durch die Hardware ergibt. Gemessen wird die Latenz ab dem Zeitpunkt, wenn die sendende CPU den Datenblock produziert hat (vor setWriteBuf), bis zu dem Zeitpunkt, wenn die Daten bei der empfangenden CPU zur Verfügung stehen (nach getReadBuf). Die Größe von Datenblöcken spielt bei der Latenz durch die Hardware eine Rolle, sodass in Tabelle 6.3 Ergebnisse für zwei verschieden große Datenblöcke (16 B und 1 kB) angegeben sind. Aufgrund des vom NI durchgeführten Blocktransfers von 1 kB Daten über das NoC ist die Latenz bei einem NoC-Transfer deutlich höher (157 Taktzyklen beim semi-statischen Ansatz) als bei der Kommunikation innerhalb eines Clusters. Dies ist darauf zurückzuführen, dass CPUs im Cluster die Daten bereits während der Work-Funktion eines Tasks direkt in den lokalen Speicher der anderen CPU schreiben. Bei einem NoC-Transfer muss hingegen der gesamte Datenblock bereits im lokalen Speicher der CPU abgelegt sein, bevor der NI mit dem Versenden beginnen kann.

Zu beachten ist, dass die Ergebnisse dieses Abschnitts lediglich den besten Fall darstellen, da beim synthetischen Benchmark bereits alle Datenblöcke produziert sind. Des Weiteren gibt es keine Konflikte auf der Kommunikationsinfrastruktur durch zusätzlichen Datenverkehr. In einer realen Anwendung kann es zu einem weiteren Nachteil des dynamischen Ansatzes kommen. Sollten alle physikalischen Kanäle des NIs bereits aktiv sein, wird eine CPU, die einen weiteren Kanal aufbauen will, so lange blockiert bis ein Kanal wieder zur Verfügung steht. Aus diesem Grund ist insbesondere für Echtzeitanwendungen der semi-statische Ansatz zu bevorzugen, da dieser deterministischer und damit besser vorhersagbar ist. Insbesondere Streaming-Anwendungen können vom semi-statischen Ansatz profitieren, da hier die Verarbeitung eines kontinuierlichen Datenstroms ein sich immer wiederholender Task ist. Sobald beim semi-statischem Ansatz ein physikalischer Kanal konfiguriert ist, kann dieser periodisch für die gesamte Laufzeit der Anwendung verwendet werden.

6.3.2 Kommunikationskanäle

Die Untersuchungen aus Abschnitt 6.3.1 haben gezeigt, dass sich der semi-statische Ansatz bei der Nutzung von Kommunikationskanälen besonders gut eignet. Der semi-statische Ansatz verursacht im Vergleich zum dynamischen Ansatz geringere Softwarekosten auf der CPU und zeichnet sich zudem durch ein deterministisches Verhalten aus. Ein Nachteil besteht jedoch darin, dass die Partitionierung einer Anwendung auf mehrere CPUs durch die Limitierung von Kanälen anspruchsvoller wird. Um den Einfluss der Anzahl physikalisch konfigurierbarer Kanäle auf die Performanz und den Flächenbedarf zu bestimmen, wird im Folgenden eine Entwurfsraumexploration durchgeführt.

Chipfläche
Zur Bestimmung der Chipfläche des NIs wird der Hardwareentwurfsablauf auf Basis einer 28-nm-Technologie verwendet (siehe Kapitel 3.3). Ergebnisse des Flächenbedarfs für verschiedene NI-Konfigurationen bei der Anzahl von Kanälen sind in Abbildung 6.4 dargestellt.

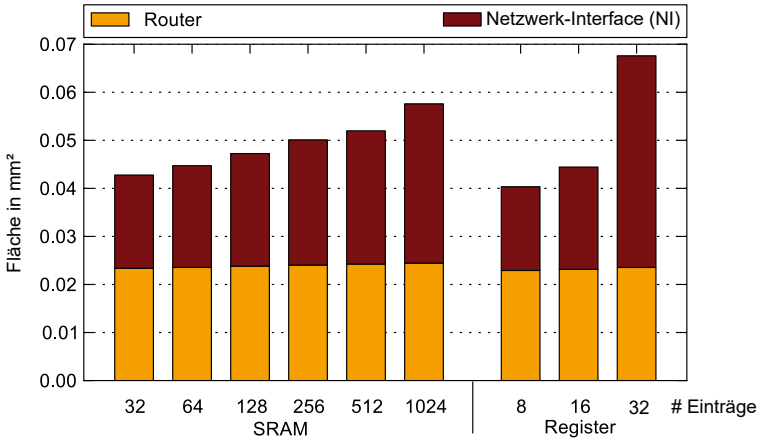


Abbildung 6.4: Flächenbedarf des synchr. Router und verschiedener NI-Konfigurationen

Die Anzahl physikalischer Kanäle, die der NI zur Verfügung stellt, hat insbesondere eine Auswirkung auf den Speicher, der für die LUT in der Empfangsstruktur verwendet wird (vgl. Abschnitt 6.2). Je mehr unabhängige Kommunikationskanäle vom NI unterstützt werden, desto mehr Einträge müssen in der LUT gespeichert werden können. Ein NI mit 32 Kanälen benötigt unter der Verwendung von SRAM-Speicher eine Chipfläche von 0,019 mm^2. Um bis zu 1024 Kanäle zu unterstützen, wird ein größerer SRAM-Speicher verwendet, welcher den Flächenbedarf des NIs auf 0,033 mm^2 erhöht. Wie in der Abbildung 6.4 zu sehen ist, steigt der Flächenbedarf des NIs nicht linear mit der Anzahl Kanäle, sondern hat vielmehr einen logarithmischen Verlauf. Dies ist auf die flächenef-

fizientere Implementierung von größeren Speicher-Makros zurückzuführen, da der Anteil kombinatorischer Logik im Vergleich zu den eigentlichen Speicherzellen immer geringer wird. Zu beachten ist jedoch, dass die Latenz bei großen Speicher-Makros ansteigt, welches Auswirkungen auf den kritischen Pfad und damit die Taktfrequenz des Systems haben kann. Eine generell geringere Latenz bietet eine Registerimplementierung der LUT. Wie erwartet, steigt hierbei jedoch der Flächenbedarf bei vielen Kanälen deutlich an. So benötigt ein NI bereits mit 32 Kanälen eine Chipfläche von 0,035 mm^2 und ist damit um 84 % größer als die Variante mit SRAM-Speicher. Sinnvoll ist die Verwendung von Registern jedoch bei einer Anzahl kleiner als 8 Kanäle, da hier der Flächenbedarf vergleichsweise gering ist (0,016 mm^2) und SRAM-Speicher dieser Größe nicht zur Verfügung stehen. Wie bei den Untersuchungen im weiteren Verlauf gezeigt wird, eignet sich eine Anzahl von acht Kanälen jedoch nicht für den semi-statischen Ansatz. Bei Verwendung von dynamischen Kommunikationskanälen kann eine Registerimplementierung jedoch sinnvoll sein.

Die Auswirkungen der Anzahl der Kommunikationskanäle auf den Flächenbedarf des Routers ist vergleichsweise gering. Der Flächenbedarf steigt hier lediglich durch den Bedarf einer größeren Bit-Weite für die Flow-ID, welche als Teil des Headers eines Flits durch den Router geführt wird. Da die Flow-ID die ID eines Kanals widerspiegelt, müssen hier bei der Verwendung vieler unabhängiger Kanäle größere Zahlen abgebildet werden können. So beträgt beispielsweise die Bit-Weite der Flow-ID bei acht Kanälen 3 bit und bei 1024 Kanälen 10 bit. Insgesamt ist der hierdurch steigende Flächenbedarf des Routers jedoch vernachlässigbar.

Die zuvor beschriebene Begrenzung von Kommunikationskanälen innerhalb des NIs betrifft allein die Empfangskanäle. Sendekanäle werden hingegen im lokalen Speicher der CPU konfiguriert und haben daher keine direkte Auswirkung auf den NI. Ihre Anzahl kann daher flexibel auf den Bedarf der Anwendung angepasst werden und ist nur durch den verfügbaren Datenspeicher begrenzt.

Performanz

Um für den semi-statischen Ansatz den Einfluss von der Anzahl von NoC-Kommunikationskanälen auf die Performanz einer Anwendung zu bestimmen, werden die Streaming-Benchmark-Suite und der CoreVA-MPSoC-Compiler aus Kapitel 3.4.4 verwendet. Dabei wird die Anzahl der NoC-Kanäle limitiert, die dem CoreVA-MPSoC-Compiler pro NI zur Verfügung stehen. Dieses Limit wird dabei für mehrere Testdurchläufe variiert, um so den Einfluss der Anzahl der Kommunikationskanäle auf die Performanz verschiedener Anwendungen zu bestimmen. Eine geringe Anzahl physikalischer NoC-Kommunikationskanäle schränkt dabei die Anzahl gültiger Partitionierungen ein, die vom Optimierungsalgorithmus des CoreVA-MPSoC-Compilers betrachtet werden. Dies kann, je nach Art der Anwendung, einen Einfluss auf die Performanz des Benchmarks haben.

Untersucht werden drei verschiedene Hardware-Konfigurationen des CoreVA-MPSoCs. Zwei dieser MPSoC-Konfigurationen sind hierarchisch mit NoC und CPU-

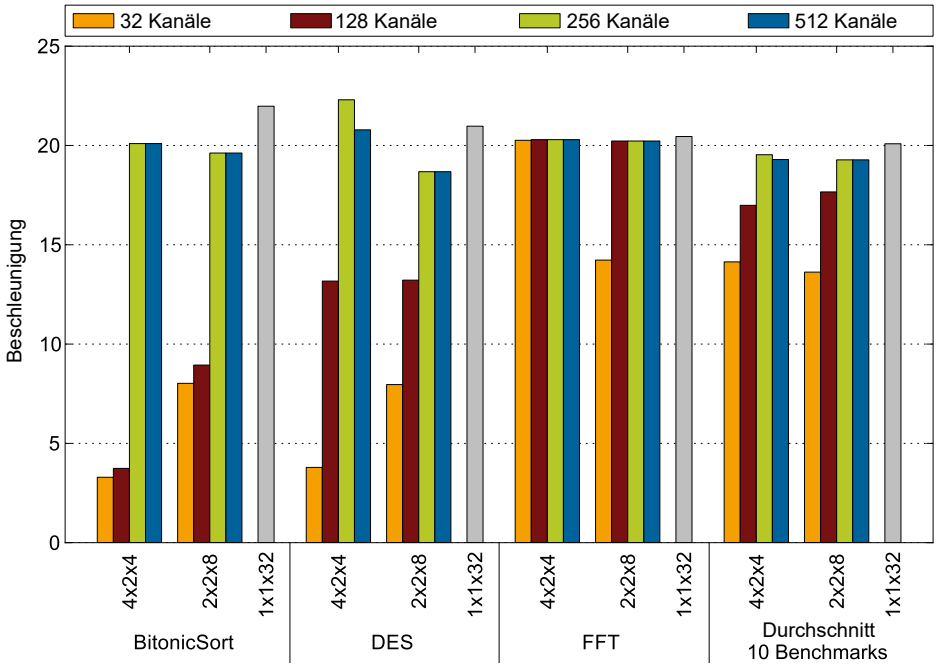


Abbildung 6.5: Beschleunigung verschiedener MPSoC-Konfigurationen mit unterschiedlicher Anzahl von Empfangskanälen per NI im Vergleich zur Lösung auf einer einzelnen CPU

Cluster und verfügen insgesamt über 32 CPUs. Eine mit einem 4x2 2D-Mesh NoC und jeweils 4 CPUs pro Cluster (4x2x4) und die andere mit einem 2x2 2D-Mesh NoC und jeweils 8 CPUs pro Cluster (2x2x8). Die CPUs innerhalb der Cluster sind dabei jeweils über eine volle AXI-Crossbar (Punkt-zu-Punkt-Verbindungen) miteinander verbunden. Als Referenz wird die Analyse zusätzlich für ein reines Cluster mit 32 CPUs durchgeführt, bei dem ebenfalls eine volle AXI-Crossbar verwendet wird. Diese Konfiguration ist als Referenz zu sehen, da ein solcher Entwurf in einem platzierten und verdrahteten Design nur schwer ressourceneffizient umsetzbar ist [162]. Außerdem gibt es bei der reinen Cluster-Konfiguration keine direkte Limitierung von Kommunikationskanälen. Indirekt kann jedoch auch hier die Menge an Datenspeicher im System zu einer Limitierung führen, welches dann eine Auswirkung auf die Performanz haben kann. Ergebnisse für die Beschleunigung der verschiedenen Konfigurationen sind im Balkendiagramm in Abbildung 6.5 zu sehen. Die Werte beziehen sich dabei auf die Ausführung der jeweiligen Anwendung auf dem Ein-CPU-System. Als Optimierungsziel für den CoreVA-MPSoC-Compiler wird hierbei die Maximierung des Durchsatzes eingesetzt. Das Limit

für die Anzahl physikalischer NoC-Kommunikationskanäle wird dabei zwischen 32 und 512 variiert.

Die rechte Gruppe des Balkendiagramms veranschaulicht die durchschnittliche Beschleunigung von 10 Anwendungen der Streaming-Benchmark-Suite (siehe Abschnitt 4.2.3). Die Referenzkonfiguration mit 32 CPUs in einem Cluster zeigt die besten Ergebnisse mit einer durchschnittlichen Beschleunigung von 20 im Vergleich zur Ausführung auf einer einzelnen CPU. Beide hierarchischen MPSoC-Konfigurationen zeigen bei der Verwendung von 256 oder mehr NoC-Kanälen mit einer durchschnittlichen Beschleunigung von 19 ähnlich gute Ergebnisse. Im Durchschnitt profitiert kein Benchmark von mehr als 256 physikalischen NoC-Kommunikationskanälen. Die Performanz steigt um 27,6 % im Vergleich zu einem Limit von 32 NoC-Kanälen, bzw. 13,1 % bei 128 NoC-Kanälen. Insgesamt konnte kein einziger Benchmark von mehr als 256 NoC-Kommunikationskanälen profitieren, obwohl der CoreVA-MPSoC-Compiler teilweise Partitionen mit mehr als 256 NoC-Kanälen für einen NI gefunden hat.

Bei den hierarchischen MPSoCs zeigen Benchmarks wie BitonicSort oder DES den größten Einfluss auf die Performanz, falls nur wenige NoC-Kommunikationskanäle pro NI zur Verfügung stehen. Beide Anwendungen beinhalten eine sehr hohe Anzahl kleiner Tasks, die ein entsprechend hohes Kommunikationsaufkommen verursachen. Dies führt am Beispiel BitonicSort dazu, dass bei 128 oder weniger verfügbaren NoC-Kanälen, die Beschleunigung bei einem 2x2x8-MPSoC nicht über neun und bei einem 4x2x4-MPSoC sogar nicht über vier liegt. Aufgrund mangelnder Anzahl von NoC-Kanälen können vom CoreVA-MPSoC-Compiler nur wenige Tasks auf andere Cluster verteilt werden, sodass die Mehrzahl an Tasks innerhalb eines Clusters verbleiben. DES zeigt eine sehr ähnliche Performanz wie BitonicSort, wobei hier der Unterschied zwischen 32 und 128 NoC-Kanälen größer ist. Bei einem 4x2x4-MPSoC zeigt die NI-Konfiguration mit 128 NoC-Kanälen mit 13,1 bereits eine höhere Beschleunigung als bei 32 NoC-Kanälen mit einer Beschleunigung von 3,8. Auch hier kann die Beschleunigung durch die Verwendung von 256 Kanälen auf 22,4 gesteigert werden. Auffällig ist, dass der DES auf einem 2x2x8 sogar eine etwas höhere Beschleunigung zeigt als bei einem reinen Cluster mit 32 CPUs. In diesem Fall scheint die Anwendung sehr stark von der DMA-Funktionalität des NIs zu profitieren. Andere Anwendungen – z.B. FFT (siehe Abbildung 6.5) – kommen mit einer geringen Anzahl von NoC-Kommunikationskanälen aus und zeigen bereits bei nur 32 NoC-Kanälen pro NI eine sehr gute Performanz durch die parallele Ausführung.

Betrachtet man Performanz und Chipfläche der verschiedenen NI-Konfigurationen gemeinsam, zeigt sich die ressourceneffizienteste NI-Konfiguration in Bezug auf die Anzahl NoC-Kanäle. Das Diagramm in Abbildung 6.6 veranschaulicht das Verhältnis von Beschleunigung und Chipfläche für verschiedene MPSoC-Konfigurationen, mit einer unterschiedlichen Anzahl von NoC-Kommunikationskanäle pro NI. Im Durchschnitt zeigt das 4x2x4-MPSoC mit 256 NoC-Kommunikationskanälen pro NI mit 4,45 das beste Verhältnis von Beschleunigung und Chipfläche. Da der steigende Flächenbedarf insbesondere auf die verwendeten Speicher zurückzuführen ist, kann für diese Analy-

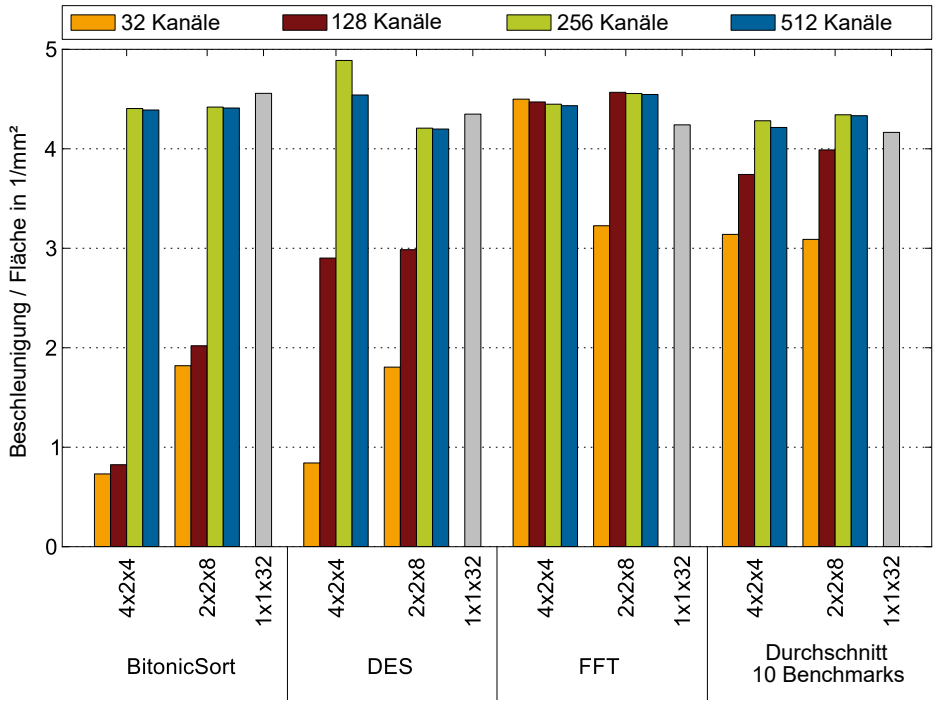


Abbildung 6.6: Verhältnis Beschleunigung und Chipfläche für verschiedene MPSoC-Konfigurationen, mit einer unterschiedlichen Anzahl von NoC-Kommunikationskanälen pro NI

se das Bewertungsmaß Chipfläche auch auf das Bewertungsmaß Energie übertragen werden. Mit der Größe der Speicher steigt die Leistungsaufnahme und somit auch der Energiebedarf des Gesamtsystems. Die Ergebnisse aus Abbildung 6.6 zeigen außerdem, dass im Durchschnitt die NoC-Konfigurationen mit 256 Kanälen eine höhere Ressourceneffizienz als die reine Cluster-Konfiguration 1x1x32 aufzeigen. Da die Ergebnisse des Flächenbedarfs bereits nach dem Synthese-Schritt bestimmt worden sind, ist zu erwarten, dass sich dieser Effekt nach dem Platzieren und Verdrahten weiter verstärkt. Erst beim Platzieren und Verdrahten werden die Leitungsverzögerungen betrachtet, sodass sich bei Cluster-Konfigurationen mit 32 oder mehr CPUs entweder die maximale Taktfrequenz stark verringert oder weitere Registerstufen eingefügt werden müssen [210].

6.4 Zusammenfassung

Die Darstellung mit dem Stand der Technik und die Untersuchungen in diesem Kapitel haben gezeigt, dass neben der eigentlichen Verbindungsstruktur des NoCs insbesondere auch die Schnittstelle vom Prozessorsystem zum NoC einen großen Einfluss auf die Effizienz von NoC-Architekturen in MPSoCs hat. Die Architektur von Netzwerk-Schnittstellen und das Programmier- bzw. Kommunikationsmodell von MPSoCs hängen sehr stark zusammen. Für eine hohe Effizienz der Kommunikation im Gesamtsystem muss daher beides aufeinander abgestimmt sein.

Der aktuelle Stand der Technik (siehe Abschnitt 6.1) zeigte, dass sich NIs in zwei verschiedene Klassen unterteilen lassen. In der ersten Klasse von NIs findet eine paketbasierte Streaming-Schnittstelle zu den Routern Verwendung. Dieser Ansatz ist insbesondere in der Forschung verbreitet und bietet eine hohe Performanz im NoC, insbesondere wenn man das NoC isoliert betrachtet. In dieser Klasse besteht die Aufgabe des NIs darin, die adressbasierte Kommunikation der CPUs direkt in die paketbasierte Kommunikation des NoCs zu überführen, z.B. durch FIFOs oder dedizierte Paketspeicher. Dies erhöht bei vielen Anwendungen jedoch die Softwarekosten der Kommunikation, da kein wahlfreier Speicherzugriff möglich ist. Die zweite Klasse von NI-Architekturen verwendet einen gemeinsamen globalen Adressraum im gesamten MPSoC und ist insbesondere in kommerziellen MPSoCs vertreten. Hier werden Daten nicht als Pakete durch das NoC geschickt, sondern CPU-Zugriffe werden direkt zusammen mit der Adresse an das NoC übergeben. Dies senkt zwar die Softwarekosten, schränkt jedoch die Skalierbarkeit oder den maximalen Datendurchsatz des NoCs ein.

Der in diesem Kapitel vorgestellte NI des CoreVA-MPSoCs kombiniert beide Ansätze. Im NoC wird eine hochperformante Paket- oder Stream-basierte Kommunikation mit Best-Effort-Datenübertragung eingesetzt. Der NI wiederum bietet den CPUs eine Adress- und Speicher-basierte DMA-Funktionalität, welches die Kommunikationskosten der Software minimiert. Dazu unterstützt der NI des CoreVA-MPSoCs zur Laufzeit eine flexible Verwaltung unabhängiger Transaktionskanäle, um so eine hohe Skalierbarkeit und ein effizientes MPSoC zu gewährleisten. Die Entwurfsraumexploration in Abschnitt 6.3 hat gezeigt, dass sich dazu der semi-statische Ansatz bei der Nutzung von Kommunikationskanälen besonders gut eignet. Der semi-statische Ansatz verursacht im Vergleich zum dynamischen Ansatz geringere Softwarekosten auf der CPU und zeichnet sich zudem durch ein deterministisches Verhalten aus. Ein NI mit 256 Kanälen hat einen Flächenbedarf von 0,027 mm^2 bei einer Leistungsaufnahme von etwa 5 mW (siehe Tabelle 7.2 aus Kapitel 7). Mit dieser Konfiguration zeigt ein hierarchisches MPSoC mit NoC (4x2x4 und 2x2x8) im Durchschnitt bessere Ergebnisse bei der Ressourceneffizienz als ein reines CPU-Cluster mit 32 CPUs.

7 NoC-Architekturen auf Systemebene

Der Entwurf effizienter NoC-Architekturen für eingebettete MPSoCs erfordert eine Betrachtung auf Systemebene. Die Systemebene bezeichnet das gesamte System des MPSoCs bestehend aus CPUs, Speicher und allen Verbindungsstrukturen, wie die innerhalb eines CPU-Clusters und das NoC. Erst eine Analyse von NoC-Architekturen auf Systemebene ermöglicht eine Aussage über dessen Leistungsfähigkeit im Gesamtsystem. Das Blockschaltbild in Abbildung 7.1 veranschaulicht ein mögliches Gesamtsystem des CoreVA-MPSoCs.

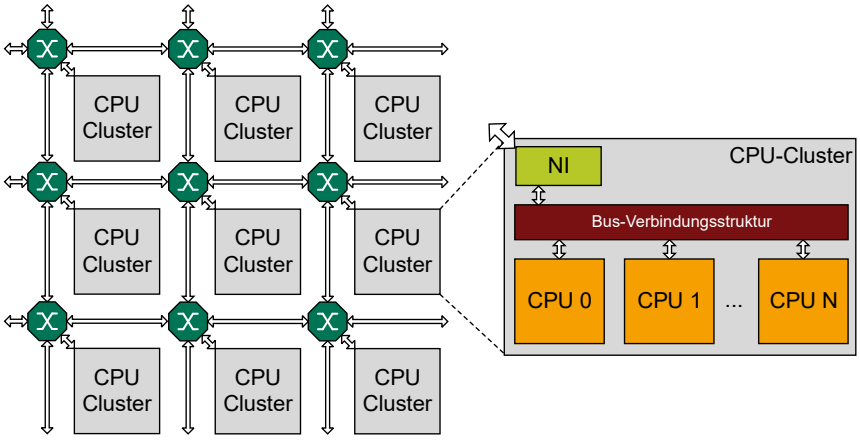


Abbildung 7.1: Blockschaltbild des hierarchischen CoreVA-MPSoCs

In diesem Kapitel werden die in den vorherigen Kapiteln vorgestellten NoC-Architekturen auf Systemebene des CoreVA-MPSoCs betrachtet. Dazu zählt insbesondere die Kombination von NoCs mit verschiedenen Speicherarchitekturen im System. Eine Einordnung in den Stand der Technik von Speicherarchitekturen für MPSoCs findet in Abschnitt 7.1 statt. Beim CoreVA-MPSoC handelt es sich um ein hierarchisches Multiprozessorsystem, welches eine Kommunikationsstruktur aus zwei Ebenen verwenden kann. Zur Entwurfszeit kann die Anzahl von CPUs erhöht werden, die an einen Knoten im Netzwerk über die Netzwerk-Schnittstelle (siehe Kapitel 6) angebunden sind. CPUs eines solchen Clusters sind durch eine eng gekoppelte Verbindungsstruktur verbunden [209]. Die Integration eines CPU-Clusters in das NoC und die Verwendung

verschiedener Speicherarchitekturen ist in Abschnitt 7.2 beschrieben. Anschließend wird in Abschnitt 7.3 eine Entwurfsraumexploration verschiedener MPSoC-Konfigurationen auf Systemebene durchgeführt. Ein Teil der hier vorgestellten Ergebnisse ist in [200] veröffentlicht.

Eine abstrakte Modellierung für das CoreVA-MPSoC mit Fokus auf die NoC-Architektur und -Kommunikation ist in Abschnitt 7.4 aufgeführt. Das Kapitel schließt mit einer Untersuchung des CoreVA-MPSoCs als Plattform für Echtzeitanwendungen (siehe Abschnitt 7.5).

7.1 Stand der Technik von Speicherarchitekturen in MPSoCs

In diesem Abschnitt wird die Verwendung verschiedener Speicherarchitekturen in Verbindung mit NoCs diskutiert und vorgestellt. Insgesamt gibt es zum Zeitpunkt dieser Arbeit vergleichsweise wenig Veröffentlichungen, die sich mit der Analyse verschiedener Speicherarchitekturen für NoC-basierte MPSoCs beschäftigt haben. Unterschiede bestehen hier in der Architektur und Adressierung der Speicher. Speicherarchitekturen unterscheiden sich in ihrer Hierarchie sowie darin ob sie geteilt, privat oder verteilt sind. Einige der hier aufgeführten MPSoCs sind bereits in den Abschnitten 2.2 und 2.3 vorgestellt. In diesem Kapitel liegt der Schwerpunkt jedoch auf der Speicherarchitektur.

Das Unternehmen Kalray präsentiert in [45] ihren hierarchischen MPSoC MPPA-256. Der MPPA-256 verwendet einen gemeinsamen Adressraum nur für Speicher innerhalb eines CPU-Clusters und nicht auf NoC-Ebene. Ein CPU-Cluster besteht aus mehreren CPU-Kernen mit lokalen Datenspeichern, die auf einem gemeinsamen L2-Speicher (Level 2) innerhalb des Clusters arbeiten. Dieser Cache-basierte Ansatz ermöglicht eine einfache Programmierung innerhalb eines Clusters, verringert jedoch die Ressourceneffizienz des Systems [13]. CPUs verschiedener Cluster kommunizieren mittels einer Nachrichten-basierten Interprozesskommunikation über das NoC.

Adaptevas Epiphany [2] ist ein weiterer kommerzieller MPSoC und ein typisches Beispiel einer Many-Core-Architektur. In ihrem 64-Kerne-Chip E64G401 ist jeweils eine 32-Bit-RISC-CPU mit einem lokalen 32 kB Speicher mit dem NoC verbunden. Durch einen globalen Adressraumraum können die CPUs über das NoC mit wahlfreiem Zugriff auf alle Speicher im MPSoC zugreifen. Um dieses System weiter zu skalieren, werden im angekündigten Chip mit 1024 Kernen 64-Bit-RISC CPUs verwendet [134].

Abgesehen von diesen NoC-basierten MPSoCs gibt es andere MPSoC-Architekturen, die eine eng gekoppelte Speicherarchitektur (Level 1) in CPU-Clustern haben und somit sehr geringe Latenzen aufweisen. Rahimi et al. [147] verbindet mehrere CPUs mit einem gemeinsamen Speicher aus mehreren Speicherbänken über eine logarithmische Verbindungsstruktur. In seiner Arbeit vergleicht er verschiedene CPU-Taktfrequenzen und Cluster-Konfigurationen. Ein Lesezugriff auf diesen Speicher hat im besten Fall

(Best-Case) nur eine Latenz von einem Taktzyklus. Die Arbeit von Rahimi et al. ist in [83] um eine kontrollierbare Pipelinstufe zwischen CPUs und Speicherbänken erweitert worden, um so zuverlässiger und fehlertoleranter bei der Herstellung eines Chips zu werden.

In [84] wird ein gemeinsamer L1-Datencache präsentiert. Dieser verwendet ebenfalls die von Rahimi et al. vorgestellte logarithmische Verbindungsstruktur zwischen CPUs und dem Cache, welche auch hier im besten Fall eine Latenz von einem Taktzyklus aufweist. Dieser Cache führt zu einem Mehraufwand in Fläche und Leistungsaufnahme (5 % bis 30 %) im Vergleich zu einem gemeinsamen L1-Speicher ohne Cache-Funktionalität, ermöglicht jedoch eine einfachere Programmierung des Gesamtsystems. Gautschi et al. [53] präsentiert ebenfalls ein Cluster mit vier OpenRISC-CPUs und einem gemeinsamen L1-Speicher mit acht Bänken. Dogan et al. [47] stellt einen Multiprozessor für biomedizinische Algorithmen vor, bei dem acht CPU-Kerne mit einem gemeinsamen L1-Datenspeicher (16 Bänke) und gemeinsamen Instruktionsspeicher (Acht Bänke) verbunden sind. Der gemeinsame Speicher kann so konfiguriert werden, dass Bänke exklusiv von einer einzelnen CPU verwendet werden können. Um private und geteilte Bänke zu segmentieren, ist jedoch eine Memory Management Unit (MMU) nötig. Die Plurality-HyperCore-Architektur [186] integriert 64 RISC-Kerne und einer Hardware-Steuereinheit (Scheduler). Die CPU-Kerne sind dabei mit einem L1-Daten- und L1-Instruktionscache verbunden, der auf einem externen Hauptspeicher arbeiten kann. Eingesetzt wird dazu ein sogenanntes „Smart-Interconnect“, über das keine detaillierten Informationen vorliegen. Der vorgestellte Instruktionscache hat eine Größe von 128 kB, der Datencache eine Größe von 2 MB.

Der STHORM-MPSoC [16] von STMicoelectronics verbindet mehrere CPU-Cluster über ein NoC. Innerhalb eines CPU-Clusters können 16 CPUs auf einen eng gekoppelten 256 kB Datenspeicher zugreifen. Dieser verfügt über 32 Bänke, die über einen logarithmischen Datenspeicher miteinander verbunden sind. Jeder CPU-Kern verfügt zusätzlich über einen privaten Instruktionscache. Jeder CPU-Cluster enthält einen DMA-Controller mit zwei Kanälen, der auf den gemeinsamen Speicher und das NoC über eine zusätzliche Bus-Verbindungsstruktur zugreifen kann. Der MPSoC kann mit OpenCL oder einem nativen Programmiermodell programmiert werden.

Rossi et al. [151] haben eine ähnliche Architektur auf Energie optimiert. Ein CPU-Cluster verfügt über vier 32-bit-CPUs, acht 2 kB L1-Speicherbänke und einem 16 kB L2-Speicher. Der DMA-Controller dieser Architektur ist eng an den gemeinsamen L1-Speicher gekoppelt und verfügt über zwei zusätzliche Ports auf einen externen 64-bit-Bus.

Loi und Benini präsentieren in [104] einen L2-Cache. Dieser erlaubt mehreren CPU-Clustern des STHORM einen Zugriff auf einen externen DRAM. Da der L2-Cache nur einmal im System vorkommt, ist in diesem System keine Kohärenz-Behandlung notwendig. Auf den L2-Cache kann über eine konfigurierbare Busstruktur (STBus) mit 9 bis 16 Taktzyklen Latenz zugegriffen werden. Der Cache erlaubt eine maximale

Taktfrequenz von 1 GHz in einer 28-nm-FD-SOI Technologie und benötigt 20 % bis 30 % mehr Chipfläche im Vergleich zu einem Scatchpad-Speicher derselben Größe.

Gemeinsame L1-Speicher werden jedoch nicht nur in MPSoC-Architekturen, sondern auch in Grafikprozessoren (Graphics Processing Units (GPUs)) eingesetzt. NVIDIA's Pascal GPUs weisen mehrere Partitionen auf, die jeweils einen Pacal-Shader-Kern mit 32 Shader-ALUs enthalten. Zwei dieser Partitionen teilen sich einen L1-Datenspeicher mit 32 Bänken und 64 kB [88]. Zusätzlich teilen sich zwei Partitionen einen L1-Texture- und L1-Instruktionscache. Die High-End Tesla P100 GPU enthält 56 dieser Paskal-Kerne und erlaubt eine Performanz von 21.1 TFlops bei einer Leistungsaufnahme von 300 W.

	Topologie		L1-Speicher			L2-Speicher		DMA
	NoC	Cl.	L.	Sh.	Cache	Sh.	Cache	
MPPA-256 [45]	✓	✓	✓	-	✓	✓	-	✓
Epiphany [2]	✓	-	✓	-	-	-	-	✓
Rahimi et al. [147]	-	✓	-	✓	-	-	-	-
Kakoee et al. [84]	-	✓	-	✓	✓	-	-	-
Gautschi et al. [53]	-	✓	-	✓	-	-	-	-
Dogan et al. [47]	-	✓	-	✓	-	-	-	-
Plurality HyperCore [186]	-	✓	-	✓	✓	-	-	-
STHORM [16]	✓	✓	-	✓	-	-	-	✓
Rossi et al. [151]	-	✓	-	✓	-	-	-	✓
Loi and Benini [104]	-	✓	-	✓	-	✓	✓	✓
NVIDIA's Pascal GPU [88]	-	✓	-	✓	✓	✓	✓	-
CoreVA-MPSoC	✓	✓	✓	✓	-	-	-	✓

Tabelle 7.1: Vergleich verschiedener MPSoCs mit Fokus auf der Speicherarchitektur [9]

In Tabelle 7.1 werden verschiedene Speicherarchitekturen und das CoreVA-MPSoC mit Bezug auf die Datenspeicherarchitektur klassifiziert. Aufgezeigt sind verschiedene Architektureigenschaften, wie NoC, Cluster (Cl.), Caches, gemeinsamen/shared (Sh.) oder lokalen (L.) Speicher und DMA-Controller.

Das CoreVA-MPSoC adressiert Streaming-Anwendungen in energiebeschränkten Systemen, die sich durch eine kontinuierliche Verarbeitung eines Datenstroms durch mehrere Tasks auszeichnen. Durch diesen statischen Datenfluss verwendet das CoreVA-MPSoC durch Software verwaltete Scratchpad-Speicher und keine Caches, wie sie in [45], [84], [88], [104] und [186] verwendet werden. Im Vergleich zum Epiphany [2] verfügt das CoreVA-MPSoC über eine hierarchische Kommunikationsstruktur mit mehreren CPU-Clustern. Tasks, die Daten untereinander mit sehr geringen Latenzen austauschen müssen, können von der Verwendung eines CPU-Clusters profitieren. Das NoC erlaubt zusätzlich die Skalierung des Systems.

Ähnlich wie die Speicherarchitekturen in [53] und [147] kann auch in den CPU-Clustern des CoreVA-MPSoCs ein eng gekoppelter gemeinsamer Speicher eingesetzt werden. Zusätzlich kann das NoC eng an den gemeinsamen Speicher gekoppelt werden, mit einem ähnlichen Konzept wie dem von Rossi et al. [151]. Als neues Konzept erlaubt die Architektur des CoreVA-MPSoC die Koexistenz von eng gekoppelten lokalen und gemeinsamen Datenspeichern. Das CoreVA-MPSoC kann zur Entwurfszeit als System mit nur lokalen Speichern, nur gemeinsamen Speichern oder einem Hybrid-System aus beiden konfiguriert werden. Im Folgenden werden diese Speicherarchitekturen und deren Anbindung an das NoC für das CoreVA-MPSoC im Detail vorgestellt. Eine Analyse über die Ressourceneffizienz für Streaming-Anwendungen der verschiedenen Architekturen ist in Abschnitt 7.3 aufgeführt.

7.2 Integration von CPU-Clustern und Speicherarchitekturen

Die Integration eines CPU-Clusters stellt, im Gegensatz zur Anbindung einer einzelnen CPU, gesonderte Anforderungen an das NoC und insbesondere an das Netzwerk-Interface. Im Folgenden werden Anforderungen und Lösungen zur Integration eines CPU-Clusters in das NoC diskutiert. Des Weiteren werden verschiedene Architekturen von Datenspeichern innerhalb eines CPU-Clusters und deren Anbindung an das NoC vorgestellt.

7.2.1 Verbindungsstrukturen im Cluster

Der allgemeine Aufbau eines CPU-Clusters im CoreVA-MPSoC ist bereits in Abschnitt 3.2 beschrieben. So entstanden im Rahmen der Dissertation [162] zwei Bus-basierte Verbindungsstrukturen zwischen den CPUs. Diese sind zur Entwurfszeit konfigurierbar und basieren auf den Standards AMBA AXI und OpenCores Wishbone. Als Topologie für die Verbindungsstruktur stehen jeweils ein geteilter Bus sowie eine partielle und eine vollständige Crossbar zur Verfügung. Durch die Cluster-Verbindungsstruktur ist es möglich, dass CPUs auf die lokalen L1-Speicher anderer CPUs zugreifen können. Durch die höhere Latenz über die Cluster-Verbindungsstruktur ergibt sich so eine NUMA[1]-Speicherarchitektur im CoreVA-MPSoC.

Grundsätzlich können im CoreVA-MPSoC in Kombination mit dem NoC alle Varianten der Cluster-Verbindungsstruktur eingesetzt werden. Besonders eignet sich jedoch die AXI-Crossbar, da diese einige Vorteile bietet bzw. die anderen Varianten Nachteile aufweisen. Durch die Anbindung des NIs (siehe Kapitel 6) an das CPU-Cluster ergeben sich neue Anforderungen an die Cluster-Verbindungsstruktur. Da der NI über eine DMA-Funktionalität verfügt, werden größere Mengen an Daten (Pakete) als ununterbrochene

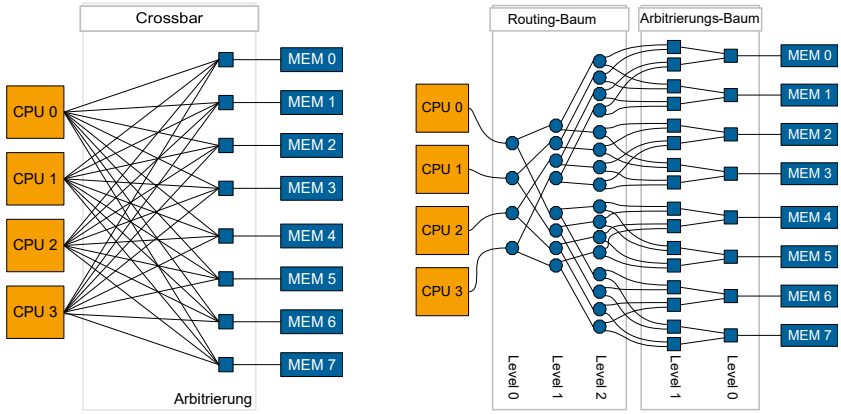
[1] *Non-Uniform Memory Access*

Übertragung (sogenannte Burst-Transfers) über die Cluster-Verbindungsstruktur aus den Speichern der CPUs gelesen bzw. in diese geschrieben. Hier kann es beim Wishbone-Bus zu Verklemmungen (Deadlocks) kommen, sobald sich zwei CPUs aus verschiedenen Clustern gleichzeitig große Pakete zusenden. In diesem Fall können die beiden Sendevorgänge die Cluster-Verbindungsstruktur blockieren, sodass die ankommenden Daten nicht in die Speicher der CPUs abgespeichert werden können und so am Eingang des NIs blockieren. Beim Wishbone-Bus muss dieser Deadlock verhindert werden, indem Burst-Transfers nicht zugelassen werden. Dies verhindert den Deadlock, verringert jedoch die Performanz des NIs erheblich. Der AXI-Standard bietet für dieses Problem eine Lösung, da Schreib- und Lesezurgriffe voneinander getrennt sind und nebenläufig stattfinden können. Dadurch kann der Lese-Burst beim Versenden von Paketen parallel zu den Schreibzugriffen beim Entgegennehmen von Paketen stattfinden. Aufgrund der Nebenläufigkeit des NIs bietet auch die Crossbar-Topologie einen Vorteil zum geteilten Bus, da hier CPUs innerhalb des Clusters weiter kommunizieren können, während der NI Paketdaten aus dem Speicher liest oder in diesen ablegt.

Um die Flexibilität und Bandbreite im CPU-Cluster weiter zu erhöhen, kann ein eng gekoppelter gemeinsamer Datenspeicher (gemeinsamer L1-Speicher) zusätzlich zu den lokalen Speichern im Cluster integriert werden [162]. Dieser Speicher verfügt über mehrere Speicherbänke und ist direkt mit der MMIO-Schnittstelle jeder CPU verbunden. Ein Konflikt tritt nur auf, wenn zwei Konsumenten (z.B. CPUs) im gleichen Taktzyklus auf dieselbe Speicherbank zugreifen wollen. In diesem Fall wird eine Round-Robin-Arbitrierung durchgeführt und ein Konsument wird für einen Taktzyklus angehalten. Um die Anzahl von Konflikten zu verringern, muss also das richtige Verhältnis zwischen der Anzahl von Konsumenten und Speicherbänken gewählt werden. Aktuelle Implementierungen von gemeinsamen L1-Speichern nutzen hier einen Bankfaktor von 2 zur Anzahl CPUs, welches sich als gute Wahl darstellt [16][47][53]. In [209] ist bereits gezeigt, dass sich diese Anzahl durch die zusätzliche Verwendung von lokalen Datenspeichern auch verringern lässt. Je geringer die Anzahl von Speicherbänken, desto ressourceneffizienter fällt die Verbindungsstruktur zwischen Bänken und Konsumenten aus. Durch die zusätzliche Anbindung des NIs als weiteren Konsumenten (siehe Abschnitt 7.2.2), wird im Rahmen dieser Arbeit dennoch ein Bankfaktor von 2 zur Anzahl CPUs verwendet.

Die CoreVA-CPU kann mit einer Latenz von zwei Taktzyklen aus dem lokalen Speicher lesen. Um nun eine Erweiterung oder ein Anhalten der CPU-Pipeline zu vermeiden, gilt diese Bedingung auch für die Verwendung des gemeinsamen L1-Speichers. Damit dies realisierbar ist, wird eine rein kombinatorische Verbindungsstruktur benötigt, damit Leseanfragen im ersten Taktzyklus an den Speicher gestellt werden und im zweiten die Daten erhalten werden können. Im Fall eines Konflikts muss daher auch das Signal zum Anhalten der Pipeline rechtzeitig zur CPU zurückkommen. Dieses Signal stellt sich häufig als der kritische Pfad im gesamten System heraus.

In [209] werden bereits zwei verschiedene Topologien für die Verbindungsstruktur des gemeinsamen L1-Speichers im CoreVA-MPSoC vorgestellt und untersucht. Die



(a) Crossbar. (b) Mesh-of-Trees [147].

Abbildung 7.2: Verbindungsstrukturen für eng gekoppelte gemeinsame Speicher für 4 CPUs und 8 Speicherbänke [162].

erste Topologie ist eine vollständige Crossbar, bei der pro Speicherbank ein einzelner Arbiter verwendet wird (siehe Abbildung 7.2a). Der Arbiter speichert dazu die ID des Konsumenten, um die Antwort eines Lesezugriffst an das korrekte Ziel zu leiten.

Als zweite Topologie kann die von Rahimi et al. [147] vorgestellte logarithmische Baumstruktur (MoT) verwendet werden. Diese MoT-Struktur besteht aus zwei gegensätzlichen logarithmischen Bäumen und ist in Abbildung 7.2b zu sehen. Der erste Baum (Routing) wird verwendet, um die Anfrage der CPU an die entsprechende Bank zu überführen. Der zweite Baum arbitriert die entsprechende Anfrage mit den Anfragen anderer CPUs.

Zur Entwurfszeit kann die Anzahl von Konsumenten und Speicherbänken bei beiden Verbindungsarten individuell konfiguriert werden. Im folgenden Abschnitt wird die Anbindung des NIs an diese Verbindungsstruktur im Detail vorgestellt.

7.2.2 NoC-Anbindung an eng gekoppelte Speicher

Um den gemeinsamen L1-Speicher auch für die Kommunikation zwischen CPU-Clustern nutzbar zu machen, muss neben den CPUs auch der NI an die Verbindungsstruktur angeschlossen werden. Dazu wird an den Verbindungsstrukturen aus Abbildung 7.2 nicht nur eine Verbindung zu den CPUs, sondern auch eine Verbindung zum NI benötigt. Das Blockdiagramm in Abbildung 7.3 veranschaulicht die Anbindung des NIs an die verschiedenen Cluster-Verbindungsstrukturen. Der NI des CoreVA-MPSoC erweitert dabei den Ansatz von zusätzlichen Ports an die Verbindungsstruktur des gemeinsamen

L1-Speichers, wie beim DMA-Controller in [151] vorgestellt. Hierdurch kann der NI Lese- und Schreib-Operationen direkt auf dem gemeinsamen L1-Speicher durchführen, welches eine schnelle Kommunikation über das NoC erlaubt. Im Vergleich zu [151] wird in der Architektur des CoreVA-MPSoCs eine hybride Speicherarchitektur verwendet, da nicht nur auf gemeinsamen L1-Speicher, sondern auch auf die lokalen Speicher der CPUs vom NI zugegriffen werden kann. Ob ein Zugriff auf den gemeinsamen L1-Speicher oder auf die Bus-Verbindungsstruktur erfolgt, wird anhand der Adresse des Speicherzugriffs festgelegt.

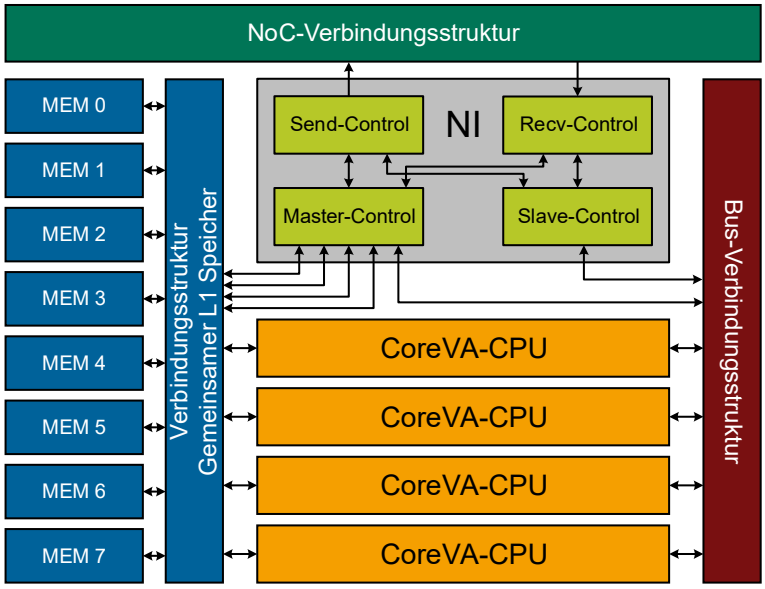


Abbildung 7.3: Der NI ist über vier Ports mit der Verbindungsstruktur des gemeinsamen L1-Speichers verbunden.

Der NI kann das Senden und Empfangen von Daten parallel durchzuführen. Um diese Parallelität zu erhalten, wird das Schreiben und Lesen über unterschiedliche Ports durchgeführt. Des Weiteren ist ein Datenwort bei der NoC-basierten Kommunikation 64 bit, beim gemeinsamen L1-Speicher jedoch nur 32 bit breit. Damit pro Taktzyklus weiterhin 64 bit Nutzdaten in den Speicher geschrieben bzw von ihm gelesen werden können, ist jeweils ein weiterer 32-bit-Port notwendig. Dies führt zu einer Anzahl von insgesamt vier Ports vom NI zur Verbindungsstruktur des gemeinsamen L1-Speichers. Ein 64-bit-Port ist am L1-Speicher zwar grundsätzlich möglich, erfordert durch die 32-bit-Architektur der CoreVA-CPU jedoch Aufwände in Software, um das benötigte Alignment zu gewährleisten.

7.3 Entwurfsraumexploration auf Systemebene

In diesem Abschnitt wird eine Entwurfsraumexploration für das CoreVA-MPSoC auf Systemebene durchgeführt. Auch hier wird eine Bewertung anhand der Bewertungsmaße aus Kapitel 4 vorgenommen. Untersucht werden verschiedene MPSoC-Konfigurationen. So wird zum einen die Anzahl von CPUs pro Cluster variiert und zum anderen verschiedene Speicherarchitekturen eingesetzt. Bei den Speicherarchitekturen wird dabei zwischen drei Varianten unterschieden:

- *Lokal:* Ausschließlich lokale Speicher.
- *Gemeinsam:* Ausschließlich gemeinsame L1-Speicher.
- *Hybrid:* Lokale und gemeinsame L1-Speicher.

In Abschnitt 7.3.1 werden zunächst die Ergebnisse für die Chipfläche vorgestellt. Die Variation der Architektur beim gemeinsamen L1-Speicher wirkt sich jedoch nicht nur auf den Flächenbedarf, sondern auch auf die maximal mögliche Taktfrequenz des Systems aus. Untersuchungen dazu sind bereits in [162], [200] und [209] zu finden. Der Vollständigkeit halber sind die Ergebnisse aber auch noch einmal im Anhang dieser Arbeit aufgeführt (siehe Abbildung A.7). Im Anschluss an den Flächenbedarf wird der Energiebedarf für einen NoC-Transfer diskutiert (siehe Abschnitt 7.3.2). Zum Schluss wird in Abschnitt 7.3.3 eine Performanzanalyse der verschiedenen MPSoC-Konfigurationen durchgeführt.

7.3.1 Chipfläche

Zur Bestimmung der Chipfläche wird der in Abschnitt 3.3 vorgestellte Entwurfsablauf basierend auf der 28-nm-FD-SOI-Technologie von STMicroelectronics verwendet. Der Flächenbedarf ist nach der Synthese bestimmt. Ergebnisse für ein platziertes und verdrahtetes MPSoC-Layout sind für eine ausgewählte Hardwarekonfiguration in Abschnitt 8.1.1 zu finden.

Für die Analysen werden als Basisblöcke die CoreVA-CPU Makros aus Abschnitt 3.3.5 verwendet. Je nach Konfiguration der Speicherarchitektur wird entweder eine CPU mit 32 kB, 16 kB oder ohne lokalen Datenspeicher verwendet. Das Makro mit 16 kB und das ohne lokalen Datenspeicher verfügt zusätzlich über eine Schnittstelle zum gemeinsamen L1-Speicher.

Für den Vergleich der unterschiedlichen Hardwarekonfigurationen wird das gesamte MPSoC auf 32 CPUs und 1 MB Datenspeicher limitiert. Variiert wird die Anzahl der CPUs pro Cluster und die Speicherarchitektur. Die Anzahl der CPUs pro Cluster wird auf 4, 8 und 16 gesetzt, die über die Bus-Verbindungstruktur im Cluster verbunden sind. Diese unterschiedlichen MPSoC-Konfigurationen werden mit 4x2x4, 2x2x8 und 2x1x16 bezeichnet, wobei die ersten zwei Zahlen die NoC-Topologie (z.B. 4x2 2D-Mesh) abbilden und die dritte Zahl die Anzahl CPUs pro Cluster. Zusätzlich werden die drei verschiedenen Speicherarchitekturen (Lokal, Gemeinsam und Hybrid) miteinander verglichen. Die Anzahl der Speicherbänke für den gemeinsamen L1-Speicher

wird jeweils auf die doppelte Anzahl con CPUs gesetzt, also z.B. 32 Bänke bei der Konfiguration mit 16 CPUs pro Cluster. Da der kritische Pfad des Gesamtsystems ab acht Speicherbänken im gemeinsamen L1-Speicher liegt (siehe Anhang A.7), wird aus Gründen der Vergleichbarkeit die Zielfrequenz bei der Synthese auf 650 MHz gesetzt. Dies erlaubt dem Synthesewerkzeug für die verschiedenen MPSoC-Konfigurationen die gleichen flächen- und energieeffizienten Standardzellen zu verwenden.

In Abbildung 7.4 zeigen die Balken für *Lokal* die MPSoC-Konfiguration ohne gemeinsamen L1-Speicher und 32 kB lokalen Datenspeicher pro CPU. Diese Konfiguration verfügt über eine volle AXI-Crossbar (0,22 mm^2 für 16 CPUs) als Bus-Verbindungsstruktur, um so die nebenläufige Kommunikation mehrerer CPUs im Cluster zu ermöglichen.

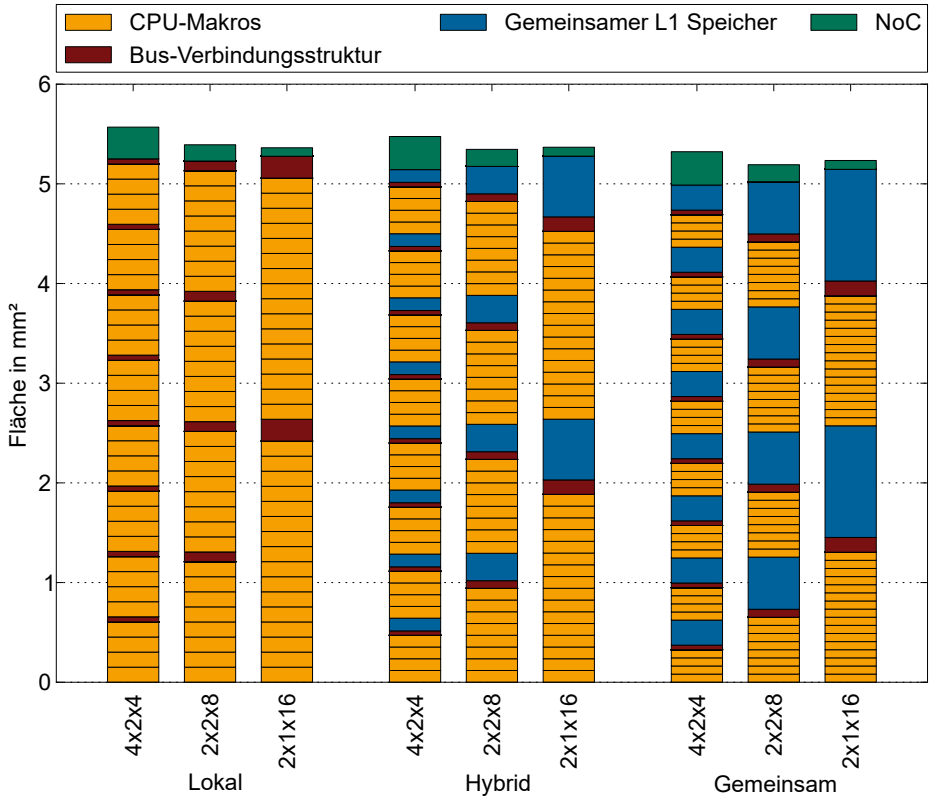


Abbildung 7.4: Flächenbedarf verschiedener CoreVA-MPSoC-Konfigurationen mit insgesamt 32 CPUs and 1 MB Datenspeicher und 650 MHz.

Alle anderen Konfigurationen verwenden einen geteilten AXI-Bus (0,15 mm^2), weil der gemeinsame L1-Speicher zur Kommunikation zwischen den CPUs verwendet wird. In diesem Fall wird der AXI-Bus allein zur Initialisierung, Synchronisation und zum Austausch von Kontrollinformationen zwischen CPUs und NI verwendet. Insbesondere für größere CPU-Cluster ist der Flächenbedarf einer Crossbar signifikant höher im Vergleich zum geteilten Bus [209]. Die Fläche für das NoC verringert sich von 0,33 mm^2 (4 CPUs pro Cluster) auf 0,09 mm^2 (16 CPUs pro Cluster), durch die abnehmende Anzahl von CPU-Clustern und damit NoC-Komponenten. Im Vergleich dazu steigt die gesamte Fläche für den Bus nur leicht an und liegt bei etwa 0,44 mm^2. In [209] ist gezeigt, dass größere CPU-Cluster mit 32 oder mehr CPUs nur mit deutlicher Verringerung der maximalen Taktfrequenz oder großer Flächenzunahme durch Pipeline-Register zu realisieren sind (siehe Anhang A.9).

Alle MPSoC-Konfigurationen *Hybrid* verwenden das CPU-Makro mit 16 kB lokalen Datenspeicher pro CPU und insgesamt 512 kB gemeinsamen L1-Speicher. Die Anzahl von Speicherbänken des gemeinsamen L1-Speichers variiert von 8 bis 32 in Abhängigkeit von der Anzahl CPUs pro Cluster. Der gesamte Flächenbedarf für den gemeinsamen L1-Speicher steigt von 1,02 mm^2 (4 CPUs pro Cluster) auf 1,22 mm^2 (16 CPUs pro Cluster). Der Flächenbedarf des NoC ist ähnlich zur Konfiguration *Lokal*. Die Fläche der Router bleibt dabei konstant, nur die Fläche des NIs steigt leicht von 0,20 mm^2 auf 0,22 mm^2 an. Dies ist durch die die zusätzlichen Ports zum gemeinsamen L1-Speicher zu begründen.

Die MPSoC-Konfigurationen *Gemeinsam* verwenden das CPU-Makro ohne lokale Datenspeicher. Zur Realisierung des größeren gemeinsamen L1-Speichers wird die Größe einer Speicherbank von 8 kB auf 16 kB erhöht, um die gleiche Anzahl von Bänken wie im *Hybrid*-Fall zu ermöglichen. Der Flächenbedarf für den gemeinsamen L1-Speicher in einem einzelnen Cluster variiert hier von 0,25 mm^2 (bei 4 CPUs im Cluster) auf 1,12 mm^2 (bei 16 CPUs im Cluster). Der Grund hierfür ist die stetig ansteigende Anzahl Bänke und damit überproportional ansteigenden Flächenbedarf für die Verbindungsstruktur.

Den geringsten Flächenbedarf hat die MPSoC-Konfiguration mit acht CPUs pro Cluster und ausschließlich gemeinsamen L1-Speicher. Insgesamt sind die Unterschiede zwischen allen MPSoC-Konfigurationen jedoch sehr gering. Aus diesem Grund wird im Folgenden der Energiebedarf und anschließend die Performanz betrachtet.

7.3.2 Energiebedarf

Da das Einsatzgebiet des CoreVA-MPSoCs eingebettete Systeme mit geringem Energiebudget sind, wird in diesem Abschnitt der Energieverbrauch für den Datentransfer zwischen CPUs betrachtet. Der Hauptfokus liegt dabei auf den Energiekosten für den Datentransfer unter Verwendung der verschiedenen Speicherarchitekturen. Die Energiekosten einzelner CPU-Zugriffe auf die verschiedenen Speicher im Cluster sind bereits

in [162, S. 157 f.] vorgestellt. Dieser Abschnitt zeigt daher die Energiekosten für den Datentransfer zwischen CPU-Clustern über das NoC.

Für die meisten Prozesstechnologien von 28 nm oder kleiner werden keine Modelle zur Leitungsverzögerung während der Synthese verwendet. Dies führt gerade für Verbindungsstrukturen (NoC, AXI) zu ungenauen Ergebnissen beim Energiebedarf. Aus diesem Grund wird für die folgenden Untersuchungen ein platziertes und verdrahtetes Layout (P&R-Layout) des CoreVA-MPSoCs verwendet. Hierzu wird das in Abschnitt 8.1.1 vorgestellte Makro eines Cluster-Knoten verwendet. Es besteht aus vier CPUs mit je 16 kB lokalen Datenspeicher und 64 kB gemeinsamen L1-Speicher pro Cluster. Alle Ergebnisse basieren auf der maximalen Taktfrequenz von 700 MHz. Zur Bestimmung der Leistungsaufnahme ist der in Abschnitt 3.3.4 vorgestellte Entwurfsablauf verwendet. Die dynamische Verlustleistung ist unter Verwendung der jeweiligen Schaltaktivitäten bestimmt, aufgenommen durch Simulation der entsprechenden Anwendung auf Gate-Ebene.

Um die Energiekosten für die Kommunikation von CPU zu CPU inklusive NoC-Komponenten zu bestimmen, wird ein synthetischer Benchmark verwendet. Dieser Benchmark besteht aus zwei einfachen Tasks, die auf zwei CPUs verschiedener Cluster abgebildet sind. Ein Task produziert zufällige Daten und sendet diese über einen NoC-Kanal zum konsumierenden Task. Der Sendevorgang wird nur ein Mal durchgeführt, um einen einzigen NoC-Transfer zu isolieren. Die zu sendenden Daten liegen bereits fertig im Speicher nahe der produzierenden CPU, sodass diese direkt gesendet werden können. Das Speichern dieser Daten findet unter realen Bedingungen bereits während der eigentlichen Anwendung (Work-Funktion) statt, sodass dieses nicht für die Kommunikationskosten berücksichtigt werden muss. Sobald die CPUs ihre Aufgaben zur Synchronisation und dem Einleiten der Kommunikation abgeschlossen haben, werden diese in den Schlaf-Modus (Idle-Modus) gesetzt. Im Idle-Modus wird der Takt der CPUs abgeschaltet (Clock-Gating), sodass hier eine deutlich geringere Verlustleistung entsteht. Die Sequenz des synthetischen Benchmarks ist sehr ähnlich zur Sequenz in Abbildung 6.3 aus Kapitel 6, wobei hier nur die Kosten zur Synchronisation und nicht die der Work-Funktion berücksichtigt werden. Das Datenvolumen des Transfers wird zwischen 8 B und 4 kB variiert. Für die Simulation auf Gate-Ebene sind zwei Makros des Cluster-Knoten verbunden. Das Szenario ist in Abbildung A.8 im Anhang zu sehen, in dem alle Komponenten hervorgehoben sind, die bei der Kommunikation über das NoC mit gemeinsamem L1-Speicher involviert sind.

In Tabelle 7.2 ist beispielhaft die Leistungsaufnahme und der Energieverbrauch für eine NoC-Kommunikation mit gemeinsamem L1-Speicher dargestellt. Die Leistungsaufnahme der einzelnen Komponenten ist für den passiven (Idle) und aktiven Zustand während einem Transfer von 1 kB Daten angegeben. Im Idle-Modus besitzen CPUs und Speicherblöcke eine sehr geringe Leistungsaufnahme, da hier der Takt abgeschaltet werden kann. Dies ist jedoch nicht für die NoC- und Bus-Komponenten möglich, weil die Kommunikationsinfrastruktur auch von anderen CPUs verwendet wird. Der Einfluss der Bus-Verbindungsstruktur ist bei der Kommunikation mit gemeinsamem L1-Speicher

Tabelle 7.2: Leistungsaufnahme und Energieverbrauch für den NoC-Transfer unter Verwendung des gemeinsamen L1-Speicher.

		Sendender Cluster				Empfangender Cluster			
		CPU	Gem. L1	NI	Router	Router	NI	Gem. L1	CPU
Leistung in mW	Idle	4.4	1.0	2.4	4.4	4.4	2.4	1.0	4.4
	Aktiv	25.5	13.6	3.8	6.0	5.6	5.3	15.4	25.3
Energie in nJ	Abs.	1.7	3.0	0.8	1.3	1.2	1.2	3.4	1.6
	Rel.	1.5	2.7	0.3	0.4	0.3	0.6	3.1	1.3

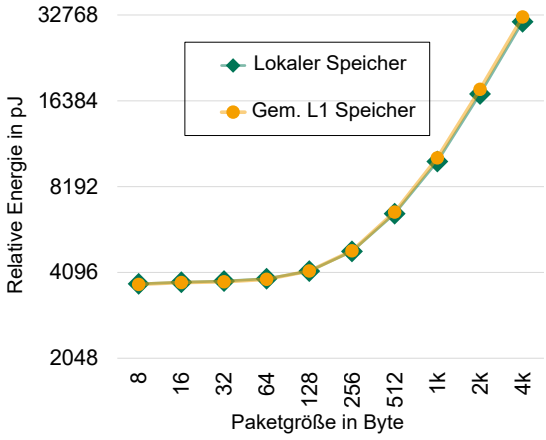


Abbildung 7.5: Energiebedarf für NoC-Transfer Lokal vs. Gem. L1-Speicher.

jedoch vernachlässigbar, da dieser nur für einen einzelnen Schreibzugriff an den NI verwendet wird, um den Sendevorgang einzuleiten. Während der Kommunikation steigt die Leistungsaufnahme der CPUs von 4 mW auf 26 mW und die des gemeinsamen L1-Speichers von 1,1 mW auf 14 mW an. Die Leistungsaufnahme der Router und NIs steigt jedoch nur leicht von 4,5 mW auf 6 mW bzw. von 2,4 mW auf 5 mW. Hierbei handelt es sich um die Leistungsaufnahme des synchronen Routers, beim asynchronen Router ist diese entsprechend geringer (siehe Abschnitt 5.3.4). Je nach dem ob der NI sendet oder empfängt, variiert die Leistungsaufnahme um 0,4 mW (siehe Tabelle 7.2).

Neben der Leistungsaufnahme wird in Tabelle 7.2 auch der Energiebedarf der einzelnen Komponenten dargestellt. Angegeben ist hier zum einen der absolute Energiebedarf während der Kommunikation. Zum anderen der relative Energiebedarf, der den Mehraufwand für Kommunikation angibt. Dort ist der Energiebedarf subtrahiert, der im Idle-Modus ohnehin nötig ist. Diese relativen Energiekosten werden mit Hilfe des Modells

in Abschnitt 7.4.2 für die Optimierung einer Anwendung auf minimalen Energiebedarf berücksichtigt.

Die Unterschiede beim Energiebedarf für die Verwendung von gemeinsamem L1-Speicher und lokalen Speicher sind in Abbildung 7.5 für verschiedene Paketgrößen dargestellt. Generell sind die Ergebnisse für beide Varianten nahezu identisch. Allein für sehr große Pakete zeigt sich ein kleiner Vorteil bei der Verwendung des lokalen Speichers. Der gemeinsame L1-Speicher benötigt 32,2 µJ für einen Transfer von 4 kB Daten, welches im Vergleich zum lokalen Speicher ein um 4 % höherer Energiebedarf ist.

7.3.3 Performanz

Auf Systemebene kann die Performanz der NoC-Kommunikation auf verschiedenen Ebenen gemessen werden. Zunächst wird die minimale Latenz eines einzelnen NoC-Transfers für die Verwendung des gemeinsamen und des lokalen Speichers angegeben. Anschließend wird die Performanz auf Anwendungsebene betrachtet.

Minimale Latenz auf NoC-Ebene

Zur Bestimmung der minimalen Latenz wird der gleiche synthetische Benchmark wie bei der Energieanalyse aus dem vorherigen Abschnitt 7.3.2 eingesetzt. Ein Task produziert Daten und sendet diese über das NoC zum konsumierenden Task. Die Messung der minimalen Latenz startet beim Schreiben des Sendebefehls an den NI und endet, sobald der konsumierende Task vom empfangenden NI das Signal über den Empfang erhält. Dieses Empfangssignal wird erst gesetzt, wenn alle Daten des Pakets angekommen sind und an der vorgesehenen Stelle im Speicher abgelegt sind.

Die minimale Latenz eines NoC-Transfers ist in Abbildung 7.6 für verschiedene Paketgrößen dargestellt. Zu sehen ist der Unterschied zwischen den NoC-Transfers mit gemeinsamen L1-Speichern und denen mit lokalen Speichern. Es zeigt sich, dass die minimale Latenz unter Verwendung des gemeinsamen L1-Speichers immer um genau sechs Taktzyklen geringer ist. Dies ist durch die größere Leselatenz vom NI bei der Verwendung von lokalen Speichern zu begründen, da diese über den Bus des Clusters ausgelesen werden müssen. Aufgrund der Burst-Fähigkeit des NIs spielt diese Leselatenz jedoch nur einmal pro Transfer eine Rolle, sodass bei größeren Paketen dieser Anteil an der gesamten Latenz des Transfers immer geringer wird.

Performanz auf Anwendungsebene

Um den Einfluss des gemeinsamen L1-Speichers auf Anwendungsebene zu bestimmen, werden die StreamIt-Benchmarks aus Abschnitt 4.2.3 verwendet. Auch hier werden, wie bei der Bestimmung des Flächenbedarfs in Abschnitt 7.3.1, die drei MPSoC-Konfigurationen 4x2x4, 2x2x8 und 2x1x16 mit je 32 CPUs und 1 MB Datenspeicher untersucht.

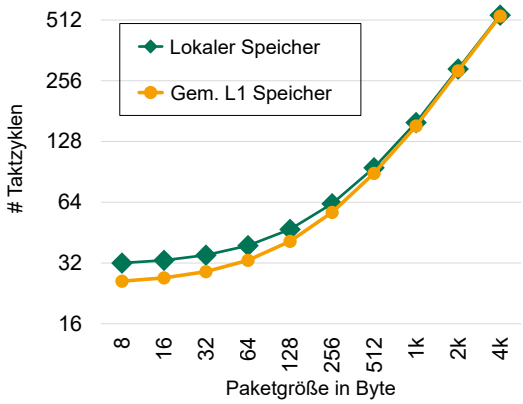


Abbildung 7.6: Minimale Latenz in Anzahl von Taktzyklen, die für NoC-Transfers verschiedener Paketgrößen mit gemeinsamem oder lokalen Speichern benötigt wird.

Hinzu kommt die Unterscheidung der drei Varianten der Speicherarchitektur *Lokal*, *Gemeinsam* und *Hybrid*.

Bei der Konfiguration *Hybrid* wird der Heap und Stack jedes Programms in den lokalen Speicher der CPUs abgelegt. Der gemeinsame L1-Speicher wird hingegen für die Datenblöcke der Kommunikation zwischen den Tasks (Filtern) verwendet. Bei der Konfiguration *Gemeinsam*, bei der kein lokaler Speicher existiert, wird der Heap und Stack jedes Programms ebenfalls im gemeinsamem L1-Speicher abgelegt.

In [162] und [209] ist bereits für ein einzelnes CPU-Cluster gezeigt worden, dass die Konfiguration *Hybrid* alle anderen Speicherkonfigurationen in der Performanz übertrifft. Dort ist eine feste Partitionierung für jede Anwendung verwendet, sodass identische Speicherzugriffsmuster entstehen. In dieser Arbeit ist es dem CoreVA-MPSoC-Compiler erlaubt, die entsprechende Speicherarchitektur zu berücksichtigen, um so von der Flexibilität des gemeinsamen L1-Speichers zu profitieren.

Das Diagramm in Abbildung 7.7 stellt die Beschleunigung des Datendurchsatz ausgewählter Benchmarks für die untersuchten MPSoC-Konfigurationen im Vergleich zur Ausführung auf einer einzelnen CPU dar. Bei vielen Algorithmen mit einer homogenen Struktur, wie z.B. der FFT, haben die verschiedene Speicherarchitekturen so gut wie keinen Effekt. Andere Anwendungen profitieren jedoch vom gemeinsamem L1-Speicher. Dies ist insbesondere der Fall, wenn einige CPUs deutlich mehr Speicher als andere benötigen, wie z.B beim LowPassFilter. In diesem Fall kann der Compiler für die kritischen CPUs größere Mengen an Speicher im gemeinsamem L1-Speicher allozieren, während andere CPUs weniger erhalten. Aus diesem Grund profitieren diese Benchmarks auch

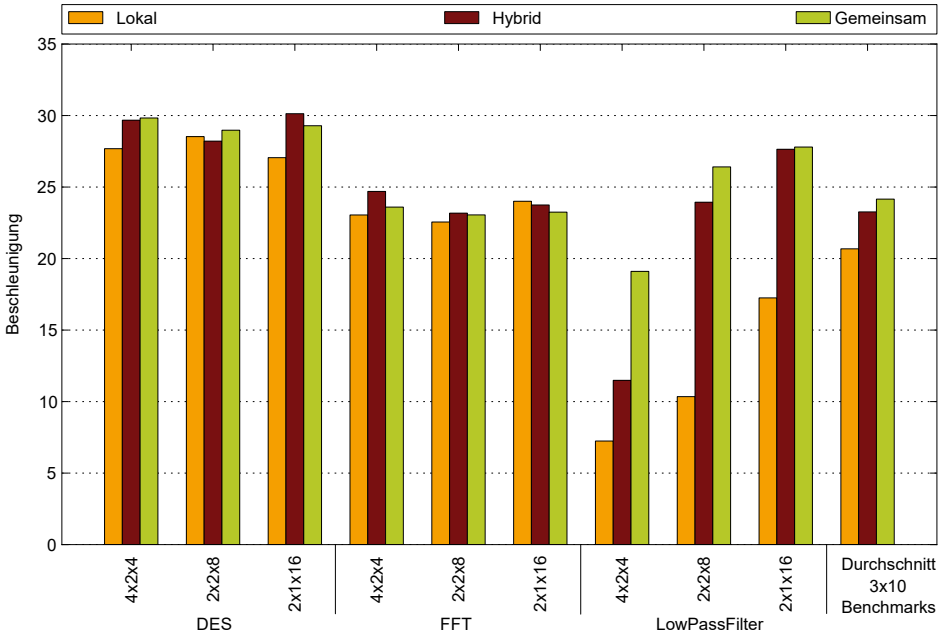


Abbildung 7.7: Beschleunigung für den Durchsatz von StreamIt-Anwendungen für verschiedene MPSoC-Konfigurationen im Vergleich zu einer einzelnen CPU.

von der größeren Menge an CPUs im Cluster, da dort größere gemeinsame L1-Speicher vorhanden sind. Die Balken unter *Durchschnitt* zeigen die durchschnittliche Beschleunigung aller zehn Anwendungen der StreamIt-Benchmark-Bibliothek für alle drei MPSoC-Konfigurationen (also 3x10 Anwendungsvarianten insgesamt). Im Durchschnitt zeigt die Verwendung des gemeinsamen L1-Speichers (*Gemeinsam*) eine um 17,2 % höhere Beschleunigung als bei der Verwendung lokaler Speicher (*Lokal*). Die Konfiguration *Hybrid* zeigt ebenfalls deutlich bessere Ergebnisse als *Lokal*, jedoch im Durchschnitt eine um 4 % geringere Beschleunigung als die Konfiguration *Gemeinsam*, da weniger gemeinsamer L1-Speicher pro Cluster zur Verfügung steht. Bei speziellen Zugriffsmustern zeigt jedoch die *Hybrid*-Konfiguration eine bessere Performanz als *Gemeinsam*, da weniger Bankkonflikte am gemeinsamem L1-Speicher entstehen. In diesem Fall profitiert die *Hybrid*-Konfiguration von weniger Zugriffen auf den gemeinsamen L1-Speicher, da Heap und Stack jeder CPU in den lokalen Speichern abgelegt sind. Allgemein ist festzustellen, dass die Vorteile der Hybriden oder Gemeinsamen Speicherarchitektur anwendungsabhängig sind.

7.4 Abstrakte Modellierung des CoreVA-MPSoCs

Im Folgenden werden abstrakte Modelle zur Abschätzung der in Kapitel 4 vorgestellten Bewertungsmaße für das CoreVA-MPSoCs vorgestellt. Das erste Modell dient der Abschätzung des Flächenbedarfs, welcher sich für verschiedene Konfigurationen des CoreVA-MPSoCs ergibt. Dies erlaubt bereits in einer frühen Entwurfsphase eine Bewertung von MPSoC-Konfigurationen.

Das zweite Modell bewertet die Performanz für die Ausführung von Streaming-Anwendungen auf dem CoreVA-MPSoC. Betrachtet werden hier zum einen die Bewertungsmaße Durchsatz und Latenz. Zum anderen lässt sich aus diesem Modell auch der Energiebedarf der Anwendung abschätzen. Das Modell ist Bestandteil des in Kapitel 3.4.4 vorgestellten MPSoC-Compilers. Weitere Informationen zum Modell sind in der Veröffentlichung [203] sowie im technischen Bericht [198] zu finden.

7.4.1 Modell für den Flächenbedarf

In diesem Abschnitt wird ein Modell für den Flächenbedarf des CoreVA-MPSoCs vorgestellt. Durch den hierarchischen Entwurfsablauf und die Skalierbarkeit des NoCs (siehe Abschnitt 3.3.5) lässt sich die **Chipfläche** A eines homogenen CoreVA-MPSoCs durch folgende Gleichung bestimmen:

$$A = n \cdot (A_{NoC} + A_{CPU\_Cluster}) \tag{7.1}$$

Mit n wird die Anzahl der Cluster-Knoten im gesamten MPSoC bezeichnet. Ein Cluster-Knoten setzt sich zum einen aus dem CPU-Cluster mit allen CPUs, Speichern und der Cluster-Verbindungsstruktur zusammen, zum anderen aus den Komponenten des NoCs. Das detaillierte Modell für den Flächenbedarf eines CPU-Cluster ist bereits in der Dissertation [162, S. 102 f.] beschrieben und wird in dieser Arbeit mit $A_{CPU\_Cluster}$ abgekürzt. Der Fokus dieser Arbeit liegt daher auf dem Modell des NoCs, dessen Flächenbedarf A_{NoC} sich wiederum aus Router und NI zusammensetzt:

$$A_{NoC} = A_{Router} + A_{NI} \tag{7.2}$$

Wie in Kapitel 5 gezeigt, setzt sich die Chipfläche des Routers in erster Linie durch die Anzahl von Ports zusammen, die von der jeweiligen Topologie abhängt. Des Weiteren steigt mit Anzahl der Ports p auch der Flächenbedarf der Crossbar. Bei der Crossbar-Verbindungsstruktur muss p quadratisch berücksichtigt werden, da hier jeder Port mit jedem anderen verbunden ist. Daraus ergibt sich folgende Gleichung für den Flächenbedarf des Routers:

$$A_{Router} = p^2 \cdot A_{Crossbar} + p \cdot A_{Port} \tag{7.3}$$

Im CoreVA-MPSoC beträgt A_{Port} am Beispiel des synchronen Routers 0,0038 mm^2 und $A_{Crossbar}$ etwa 0,000 16 mm^2 (siehe Abschnitt 5.2.3).

Der Flächenbedarf des NIs setzt sich aus einem statischen Teil A_{NI_stat} und einem dynamischen Teil A_{NI_dyn} zusammen:

$$A_{NI} = A_{NI\_stat} + A_{NI\_dyn} \tag{7.4}$$

Der dynamische Anteil wird insbesondere durch die Anzahl der unabhängigen Kommunikationskanäle k bestimmt (siehe Abschnitt 6.3.2). Diese werden im Modell nur für die Verwendung mit SRAM-Speicher angeben und müssen daher logarithmisch berücksichtigt werden. Dies ist auf die flächeneffizientere Implementierung von größeren Speicher-Makros zurückzuführen, da der Anteil kombinatorischer Logik im Vergleich zu den eigentlichen Speicherzellen immer geringer wird. Im Fall von Registern stellt sich ein quadratisches Verhalten ein. Daraus ergibt sich für den dynamischen Anteil des NIs:

$$A_{NI\_dyn} = \log_2 k \cdot A_{NI\_kanal} \tag{7.5}$$

Im CoreVA-MPSoC beträgt A_{NI_stat} 0,01 mm^2 und A_{NI_kanal} kann mit 0,002 mm^2 approximiert werden (siehe Abschnitt 6.3.2). Approximieren lässt sich der Flächenbedarf des gesamten NoCs A_{NoC_ges} im CoreVA-MPSoC daher mit der Formel:

$$A_{NoC\_ges} = n \cdot (p^2 \cdot 0,00016 + p \cdot 0,0038 + 0,01 + \log_2 k \cdot 0,002) \tag{7.6}$$

Dabei ist A_{NoC_ges} in mm^2 angegeben, wobei n die Anzahl CPU-Cluster, p die Anzahl Ports eines Routers und k die Anzahl von Kommunikationskanälen im NI ist.

7.4.2 Ein Modell zur Abschätzung von Performanz und Energie

Dieser Abschnitt beschreibt ein Modell zur Abschätzung der Performanz von Streaming-Anwendungen, die auf dem CoreVA-MPSoC ausgeführt werden. Eingesetzt wird dieses Modell im CoreVA-MPSoC-Compiler (siehe Abschnitt 3.4.4), um die Performanz verschiedener Partitionierungen einer StreamIt-Anwendung zu bewerten. Der Schätzer (*Performance-Estimator*) vereint dabei analytische und simulationsbasierte Ansätze, was auch als simulationsbasierte Abschätzung (*Simulation Based Estimation*, SBE) bezeichnet wird [203].

Das Verfahren bestimmt durch Simulationen die Ausführungszeit von einzelnen StreamIt-Filtern (Tasks), Kommunikationsfunktionen (siehe Abschnitt 3.4.3 und Abschnitt 6.3.1) sowie den Aufwand für die Aufteilung von Daten. Die Bestimmung dieser Größen ist nur einmal pro Anwendung bzw. MPSoC-Konfiguration notwendig. Die Größen werden in einer einheitlichen Datenstruktur abgelegt und vom CoreVA-MPSoC-Compiler eingelesen. Aus diesen Größen berechnet der *Performance-Estimator* die Laufzeit einer konkreten Partitionierung einer Anwendung mit Hilfe des im Folgenden vorgestellten Modells. Dies ermöglicht es dem CoreVA-MPSoC-Compiler in kurzer Zeit viele verschiedene Partitionierungen einer Anwendung zu bewerten. Anschließend kann sich der Compiler für die effizienteste Partitionierung entscheiden. Grundsätzlich lässt

sich dieses Modell für jedes hierarchische MPSoC mit Cluster- und NoC-Kommunikation anwenden.

Modell des CoreVA-MPSoC

Das CoreVA-MPSoC besteht aus einer Menge von Prozessoren P. Der CoreVA-MPSoC-Compiler partitioniert (engl. Mapping) jeden StreamIt-Filter F auf einen bestimmten Prozessor: $M : F \mapsto P$. Das MPSoC hat eine Menge von CPU-Clustern C und jeder Prozessor gehört zu einem Cluster: $C : P \mapsto C$. Zusätzlich besteht das MPSoC aus einer Menge Kommunikations- oder Netzwerklinks N. Jedes $n \in N$ hat eine maximale Bandbreite $B(n)$ $\frac{bytes}{cycle}$. Ein Netzwerklink kann ein NoC-Link, ein Netzwerk-Interface (NI) oder auch ein Bus-Link im Cluster sein. $N(p_a, p_b)$ ist eine Liste aller involvierten Netzwerklinks, wenn eine Nachricht von Prozessor $p_a \in P$ zu Prozessor $p_b \in P$ gesendet wird (Abhängig vom Routing-Algorithmus): $N : (p_a, p_b) \rightarrow [N]$

Modell eines StreamIt-Programms

Eine Streaming-Anwendung zeichnet sich durch die kontinuierliche Verarbeitung eines Datenstroms aus. Bei StreamIt-Anwendungen geschieht dies durch die periodische Verarbeitung von immer gleich großen Datenblöcken. Dabei stellt sich nach einer gewissen Initialisierungsphase ein Gleichgewichtszustand ein, in dem der sich wiederholende Verarbeitungsschritt immer gleich lange dauert. Dieser Verarbeitungsschritt wird im Gleichgewichtszustand auch als Steady-State bezeichnet [177]. Das hier vorgestellte Modell abstrahiert auf der Ebene eines solchen Steady-State.

Ein StreamIt-Programm kann als strukturierter Graph $G = (F, E)$ dargestellt werden (siehe Anhang B), wobei F eine Menge von Filtern und E eine Menge von Kanten (engl. Edges) ist. Jedes $e \in E$ hat dabei die Form (a, b), welches den Kommunikationskanal zwischen Filter $a \in F$ und $b \in F$ repräsentiert. Übertragen wird dabei eine Nachricht mit der Größe $|e|$ (in $\frac{\text{Bytes}}{\text{Work-Funktion von a}}$) von Filter a zu Filter b. Jeder Filter $f \in F$ hat eine Arbeitsfunktion (Work-Funktion) mit einer Ausführungszeit $W(f)$ in Taktzyklen. Die Ausführungszeit $W(f)$ enthält dabei ggf. mehrere Ausführungen der Work-Funktion, sodass genug Daten von angrenzenden Filtern konsumiert bzw. produziert werden können.
Es existiert ein einziger Filter Q ohne eingehende Kanten, welcher der erste Filter und damit die Quelle der Anwendung ist: $\nexists b \in F \; s.t. \; (b, Q) \in E$.
Es existiert ein einziger Filter S ohne ausgehende Kanten, welcher der letzte Filter und damit die Senke der Anwendung ist: $\nexists b \in F \; s.t. \; (S, b) \in E$.
$\mathscr{M}$: ist der Multiplikator, der angibt wie oft die Work-Funktion aller Filter während eines Steady-State aufgerufen wird ($\frac{\text{Aufrufe Work-Funktion}}{\text{Steady-State}}$). Mit $\mathscr{M}$ kann die Größe eines Datenblocks in einem Verarbeitungsschritt erhöht werden.
Ein Filter $f \in F$ hat Eingangskanten: $I(f) = \{(a, f) | (a, f) \in E\}$.
Ein Filter $f \in F$ hat Ausgangskanten: $O(f) = \{(f, b) | (f, b) \in E\}$.

Ausführungszeit eines Prozessors

Für jeden Filter $f \in F$ wird ein Programmcode in folgender Form generiert (vgl. Abschnitt 3.4.3):

```
foreach (Kanal i in I(f))
        i.getReadBuf
foreach (Kanal o in O(f))
        o.getWriteBuf
Work_f()
foreach (Kanal i in I(f))
        i.setReadBuf
foreach (Kanal o in O(f))
        o.setWriteBuf
```

Vor Ausführung der Work-Funktion $Work_f$ von Filter $f \in F$, ist es notwendig auf die Verfügbarkeit aller Kommunikationskanäle/Datenblöcke (Eingangskanten $I(f)$ und Ausgangskanten $O(f)$) zu warten. Nach $Work_f$ können alle Kommunikationskanäle ($I(f)$ und $O(f)$) frei gegeben werden. Die Ausführungszeit der Synchronisierungsfunktionen wird durch den Kanaltyp der Kante $(a,b) \in E$ bestimmt, welcher durch die Platzierung M (Mapping) der Filter a und b bestimmt ist (gleicher Prozessor, verschiedene Prozessoren aber gleicher Cluster oder verschiedene Cluster):
$M(a) = M(b) \rightarrow$ Speicher-Kanal
$M(a) \neq M(b) \ \wedge C(M(a)) = C(M(b)) \rightarrow$ Cluster-Kanal
$C(M(a)) \neq C(M(b)) \rightarrow$ NoC-Kanal
Die Ausführungszeit für das Warten der Eingangskanäle (getReadBuf) von Kante $e \in E$ wird durch $I_g(e)$ repräsentiert und $O_g(e)$ für die Ausgangskanäle (getWriteBuf). Die Ausführungszeit für die Freigabe der Eingangskanäle (setReadBuf) wird durch $I_s(e)$ repräsentiert und $O_s(e)$ für die Ausgangskanäle (setWriteBuf).

Die Ausführungszeit $T(f)$ (in Taktzyklen pro Steady-State) von Filter $f \in F$ ist die Summe aus der Ausführungszeit der Work-Funktion $W(f)$ des Filters, multipliziert mit dem Multiplikator $\mathscr{M}$ und der Summe aller Softwarekosten für die Synchronisationsfunktionen der Kommunikationskanäle aller Eingangs- und Ausgangskanten:

$$T(f) = \mathscr{M} \cdot W(f) + \sum_{e \in I(f)} (I_g(e) + I_s(e)) + \sum_{e \in O(f)} (O_g(e) + O_s(e)) \ \frac{Taktzyklen}{Steady-State} \tag{7.7}$$

Die Ausführungszeit eines Prozessors $p \in P$ in Taktzyklen pro Steady-State ist damit:

$$T(p) = \sum_{f \in M'(p)} T(f) \ \frac{Taktzyklen}{Steady-State} \tag{7.8}$$

Dabei sind $M'(p)$ alle Filter die auf Prozessor $p \in P$ platziert sind: $M'(p) = \{f \in F | M(f) = p\}$

Ausführungszeit eines Kommunikationslinks

Das Datenvolumen $V(n)$ (in Bytes pro Steady-State) gibt die Menge an Daten an, die Kommunikationslink $n \in N$ überqueren. Das Datenvolumen hängt dabei vom Multiplikator $\mathscr{M}$ und der Nachrichtengröße aller Kanten ab, die durch Kommunikationslink n gehen:

$$V(n) = \mathscr{M} \cdot \sum_{e \in \{(a,b) \in E | n \in N(M(a),M(b))\}} |e| \ \tfrac{Bytes}{Steady-State} \quad (7.9)$$

Die aktive Zeit $T(n)$ eines Kommunikationslinks $n \in N$ wird dann durch die maximale Bandbreite $B(n)$ (in Bytes pro Taktzyklus) ermittelt, die ein Kommunikationslink bereitstellt.

$$T(n) = \frac{V(n)}{B(n)} \ \tfrac{Taktzyklen}{Steady-State} \quad (7.10)$$

Durch die Trennung von der Work-Funktion und der eigentlichen Kommunikation sowie durch den Einsatz von *Multi-* oder *Double-Buffering* im Kommunikationsmodell (siehe Abschnitt 3.4.3), kann die Prozessorzeit und die Kommunikationszeit getrennt betrachtet werden. Die gesamte Ausführungszeit eines Steady-States T stellt sich daher auf die Bottleneck-Komponente aller Prozessoren $p \in P$ und Kommunikationslinks $n \in N$ ein:

$$T = \{max(T(p) \cup T(n)) \mid \forall p \in P \ \forall n \in N\} \ \tfrac{Taktzyklen}{Steady-State} \quad (7.11)$$

Durchsatz

Der maximale Durchsatz D (in Bytes pro Taktzyklus) der Anwendung ist dann die Inverse der Ausführungszeit des Steady-State T multipliziert mit dem produzierten Datenvolumen des letzten Filters S:

$$D = V(S) \cdot \frac{1}{T} \ \tfrac{Bytes}{Taktzyklus} \quad (7.12)$$

Je nach Definition kann auch das produzierte Datenvolumen des ersten Filters Q verwendet werden:

$$D = V(Q) \cdot \frac{1}{T} \ \tfrac{Bytes}{Taktzyklus} \quad (7.13)$$

Latenz

In einem StreamIt-Programm stellt jede horizontale Ebene des Graphen eine Piplinestufe der Anwendung dar. Durch das Füllen der Pipeline während der Initialisierungsphase (Prime-Pump, [177]), ist während der Steady-State-Phase eine kontinuierliche Abarbeitung von Datenblöcken möglich. Durch diesen Prime-Pump ist die Latenz L einer Anwendung, durch die Pipelinetiefe (Tiefe des Graphen) $d(G)$ und der Ausführungszeit T des Steady-States vorgegeben:

$$L = d(G) \cdot T \ Taktzyklen \quad (7.14)$$

Energie

Auch zur Abschätzung des Energiebedarfs einer konkreten Anwendung kann das in diesem Abschnitt vorgestellte Modell verwendet werden. Durch das Modell ist für jede einzelne Komponente (*Prozessor*, *Kommunikations-Link*) die aktive Zeit innerhalb eines Steady-States bekannt. Informationen über die Leistungsaufnahme der einzelnen Komponenten im aktiven und passiven Zustand sind in einer Datenbank abgelegt. Mit Hilfe von Modell und Datenbank kann der CoreVA-MPSoC-Compiler schließlich eine Bewertung des Energiebedarfs berechnen.

Zur Bestimmung des Energiebedarfs im aktiven Zustand sind verschiedene Abstraktionsebenen möglich, die sich auf die Genauigkeit der Energieabschätzung auswirken [207]. Für die Komponente *Prozessor* haben beispielsweise die auszuführenden Instruktionen einen Einfluss auf die Leistungsaufnahme der CPU. Untersuchungen hierzu fanden in der betreuten Abschlussarbeit [223] statt und sind in [207] veröffentlicht. Bei den Kommunikationslinks beeinflusst die zu übertragende Datenmenge den Energiebedarf. Außerdem hängt es hier vom genauen Typ des Kommunikationslinks ab. So sind in einer NoC-Kommunikation mehrere Hardwarekomponenten involviert als bei einer Kommunikation innerhalb eines CPU-Clusters. Ergebnisse für den Energiebedarf der NoC-Kommunikation sind in Abschnitt 7.3.2 vorgestellt und ebenfalls Teil der Datenbank des CoreVA-MPSoC-Compilers.

7.5 Das CoreVA-MPSoC als Plattform für Echtzeitanwendungen

Eingebettete mikroelektronische Systeme finden zahlreichen Einsatz innerhalb technischer Systeme mit Echtzeitanforderungen. Als echtzeitfähig wird die deterministische Ausführung einer Aufgabe bezeichnet, bzw. eine Verarbeitung der Aufgabe innerhalb einer bestimmten endlichen Zeit [96]. Dieser Determinismus kann in harte und weiche Echtzeitfähigkeit aufgeteilt werden [190]. Bei harter Echtzeitanforderung ist eine verspätete Verarbeitung aus Sicht der Anwendung nicht akzeptabel. Ist hingegen eine verspätete Verarbeitung tolerierbar und das Gesamtsystem weiterhin funktionsfähig, spricht man von weicher Echtzeitanforderung. Typische Einsatzgebiete eingebetteter Systeme mit Echtzeitanforderungen sind beispielsweise in industriellen Anlagen oder im Automobilbereich zu finden.

Dieses Kapitel beschäftigt sich mit dem CoreVA-MPSoC als Plattform für Echtzeitanwendungen. Betrachtet wird im Rahmen dieser Arbeit insbesondere die Echtzeitfähigkeit mit Bezug auf die Kommunikation im NoC. In Abschnitt 7.5.1 werden zunächst Konzepte zur Realisierung einer echtzeitfähigen NoC-Architektur aus der Literatur vorgestellt. Anschließend wird in Abschnitt 7.5.2 die Betrachtung von Echtzeit im CoreVA-MPSoC diskutiert.

7.5.1 Stand der Technik

In der Metastudie von Hesham et al. [71] werden die verschiedenen Methoden zur Behandlung von Echtzeitbedingungen in NoCs betrachtet. Diese Studie gibt einen detaillierten Überblick der verschiedenen Konzepte und vergleicht diese miteinander. Die entscheidende Eigenschaft von NoC-Architekturen für die Bereitstellung von Echtzeitkommunikation ist die Flusskontrolle, genau genommen das Switching-Verfahren (siehe Abschnitt 2.1.3).

Das klassische Verfahren für Echtzeitkommunikation ist das Circuit-Switching, bei dem eine physikalische Verbindung nur von einem Datentransfer exklusiv genutzt werden kann. Dies garantiert die Übertragungszeit eines Datentransfers (*Guaranteed-Service*, GS) und eignet sich daher als Methode für Echtzeitkommunikation. Als Beispiel für Circuit-Switching ist das NoC des STHORM [180] zu nennen (siehe Abschnitt 2.2). Die NoCs des Æthereal [56] und Argo [91] bieten mit dem TDM[2]-Verfahren eine Erweiterung zum Circuit-Switching. Durch das Senden von Paketen zu definierten Zeitpunkten, kann eine Echtzeitanwendung leichter entwickelt werden.

Um auch für Durchsatz-optimierte NoCs (*Best-Effort*, BE) mit Packet-Switching und Wormhole-Switching eine Echtzeitkommunikation zu erreichen, kann ein prioritätenbasiertes Packet-Switching eingesetzt werden. Grundsätzlich kann die Kommunikation in NoCs mit Wormhole-Switching nur schwer zyklenakkurat vorhergesagt werden [161]. Bei prioritätenbasiertem Packet-Switching können echtzeitkritische Pakete eine höhere Priorität erhalten und Flits anderer Pakete überholen. Dies ermöglicht für diese speziellen Pakete eine GS-Kommunikation. Beispiele für NoCs mit Packet-Switching und prioritätenbasierter Kommunikation sind der Tomahawk2 [189] und das DSPIN [116]. Eine andere Methode für die Mischung von BE- und GS-Kommunikation ist der Einsatz von zwei unabhängigen NoCs, welches in [71] als Hybrid-Technik bezeichnet wird. Typischerweise wird bei der Hybrid-Technik ein NoC mit Circuit-Switching für Echtzeitkommunikation eingesetzt und ein weiteres NoC mit Packet-Switching für die Datenübertragungen mit hohem Durchsatz. Der Kilocore [20] verfügt beispielsweise über eine solche Hybrid-Technik mit zwei NoCs.

Die Studie von Hesham et al. [71] vergleicht die Verfahren Circuit-Switching, TDM, priorisiertes Packet-Switching und die Hybrid-Technik miteinander. Es zeigt sich, dass insbesondere die Hyprid-Technik aufgrund der zwei NoCs den größten Flächenbedarf hat. Circuit-Switching und TDM haben etwa den gleichen Flächenbedarf und die Paket-basierten NoCs mit prioritätenbasierter Kommunikation den geringsten. Für die Unterstützung von Echtzeitanwendungen eignen sich jedoch insbesondere die Circuit-Switching und TDM-Verfahren, da für Echtzeitgarantien die Kommunikation nur auf Ebene von ganzen Datenblöcken betrachtet werden muss. Bei NoCs mit prioritätenbasierten Packet-Switching muss die Kommunikation genau genommen zu jedem Zeitpunkt eindeutig analysiert werden, um Echtzeitgarantien auf Zyklen-Ebene zu

[2] *Time Division Multiplexing*

geben [71]. Hesham et al. [71] kommen daher zu dem Schluss, dass NoCs mit TDM die größte Effizienz für Echtzeitsysteme aufweisen.

Eine Methode zur Behandlung von Echtzeitanwendungen für NoCs ohne spezielle GS-Mechanismen wird hingegen in [148] vorgestellt. Durch die Berechnungen von Kommunikationslatenzen für den schlimmsten Fall (Worst-Case) können Echtzeitschranken berechnet und garantiert werden. Dies ermöglicht auch den Einsatz von BE[3]-NoCs [148], die z.B. durch Wormhole-Routing einen höheren Durchsatz erreichen können (siehe Abschnitt 2.1.3). Ein ähnlicher Ansatz wird auch im CoreVA-MPSoC verfolgt (siehe Abschnitt 7.5.2).

Für alle Methoden muss bei der Umsetzung von harter Echtzeit eine statische Analyse zur Entwicklungszeit durchgeführt werden [71]. Generell eignen sich MPSoCs für Echtzeitanwendungen mit sehr geringen Latenzen und Zykluszeiten nur bedingt, da durch den Softwareanteil Latenzen im Nanosekunden-Bereich nur schwer umzusetzen sind. Eine Alternative für Echtzeitanwendungen mit sehr geringen Latenzen bieten neben klassischen ASICs auch FPGAs [197][202][212].

7.5.2 Echtzeit im CoreVA-MPSoC

Im CoreVA-MPSoC wird Echtzeit ähnlich zum Ansatz von [148] behandelt, da auch beim NoC des CoreVA-MPSoCs mit Hilfe von Wormhole-Switching ein auf Fläche/Energie und Durchsatz optimiertes BE-NoC verwendet wird (siehe Kapitel 5). Obwohl die NoC-Verbindungsstruktur des CoreVA-MPSoCs eine BE-Kommunikation aufweist, erlauben weitere Systemeigenschaften des CoreVA-MPSoCs eine gute Abschätzung der Latenz von Anwendungen. Die Tatsache, dass die CPUs zur Kommunikation untereinander eine explizite Kommunikation mit Scatchpad-Speicher verwenden und auf Caches verzichtet wird, gibt der Anwendung bzw. der Software mehr Kontrolle und ermöglicht eine genauere Abschätzung [68][13]. Des Weiteren kann die Kommunikation nebenläufig zur Ausführungszeit auf den CPUs betrachtet werden. Grund hierfür sind das Kommunikationsmodell (siehe Abschnitt 3.4.3) und die DMA-Funktionalität des NIs (siehe Kapitel 6). Zusätzlich erlaubt das statische Verhalten von StreamIt-Anwendungen eine statische Analyse der Anwendung zur Entwicklungszeit.

Mit Hilfe des in Kapitel 7.4 vorgestellten Modells zur Latenzabschätzung kann der CoreVA-MPSoC-Compiler (siehe Abschnitt 3.4.4) Latenzschranken bei der Partitionierung der einzelnen Tasks/Filter berücksichtigen. Wie in [203] gezeigt, liegt der durchschnittliche Fehler bei der Abschätzung des CoreVA-MPSoC-Compiler von StreamIt-Anwendungen bei 2,6% im Vergleich zu den zyklenakkuraten Ausführungen auf dem Simulator. Je nach Anwendung kann dieser Fehler jedoch auch bis zu 10% betragen. Für Systeme mit harten Echtzeitanforderungen ist daher stets ein finaler Test mit der Hardware nötig. Durch die deterministische Hardware-Architektur des CoreVA-MPSoCs – z.B. durch den Verzicht von Caches – kann bei erfolgreichen Tests die Echtzeit

[3] *Best-Effort*

garantiert werden. Sollten Latenzschranken bei den Tests nicht eingehalten werden, besteht die Möglichkeit, dem CoreVA-MPSoC-Compiler härtere Latenzschranken als eigentlich benötigt vorzugeben, um so ggf. den finalen Test zu bestehen.

7.6 Zusammenfassung

In diesem Kapitel wurde das NoC auf Systemebene betrachtet und dessen Zusammenspiel mit anderen Systemkomponenten wie CPUs, Speicher und Cluster-Verbindungsstrukturen untersucht. Als Bus-Verbindungsstruktur im CPU-Cluster stellt sich die AXI-Crossbar in Verbindung mit dem NoC als vorteilhaft dar. Der AXI-Standard ermöglicht unabhängige Schreib- und Lesekanäle, sodass Pakete vom NI nebenläufig empfangen und versendet werden können.

Bei der Untersuchung verschiedener Speicherarchitekturen hat sich die Anbindung eines gemeinsamen L1-Speichers innerhalb eines CPU-Clusters an das NoC gegenüber einer Architektur mit ausschließlich lokalen L1-Speichern als effizienter erwiesen. Untersucht wurden drei verschiedene Konfigurationen: Nur lokale L1-Speicher an jeder CPU, nur gemeinsame L1-Speicher innerhalb eines CPU-Clusters und eine Hybrid-Variante aus beiden. Sowohl beim Flächen- als auch beim Energiebedarf zeigen alle drei Konfigurationen nur marginale Unterschiede. Insbesondere bei der Performanz auf Anwendungsebene zeigen die beiden Konfigurationen mit gemeinsamen L1-Speichern bessere Ergebnisse. Durch die größere Flexibilität bei der Speicherverwaltung profitieren insbesondere Streaming-Anwendungen mit einem inhomogenen Speicherbedarf pro CPU, wie z.B. LowPassFilter. Bei der Ausführung aller 10-Benchmarks auf drei verschiedenen MPSoC-Konfigurationen (2x1x16, 2x2x8 und 4x2x4) zeigt die Verwendung des gemeinsamen L1-Speichers im Durchschnitt eine um 17,2% höhere Performanz als die Verwendung von ausschließlich lokalen Speichern.

Neben den Untersuchungen auf Systemebene wurde in diesem Kapitel auch eine abstrakte Modellierung für das CoreVA-MPSoC mit Fokus auf die NoC-Architektur und -Kommunikation aufgezeigt. Zum einen ermöglicht das vorgestellte Modell zur Abschätzung des Flächenbedarfs bereits zu einem frühen Entwicklungszeitpunkt eine Prognose über die Chipfläche verschiedener MPSoC-Konfigurationen. Zum anderen unterstützt das Modell zur Bestimmung der Performanz und des Energiebedarfs von Streaming-Anwendungen die Arbeit des CoreVA-MPSoC-Compilers. Mit Hilfe des Modells können Anwendungen in kurzer Zeit anhand ihres Durchsatzes, der Latenz oder des Energiebedarfs bewertet werden und so optimal auf dem CoreVA-MPSoC platziert werden.

Insbesondere die Latenzbestimmungen des CoreVA-MPSoC-Compilers werden auch für die Ausführung echtzeitkritischer Anwendungen auf dem CoreVA-MPSoC eingesetzt. Im Gegensatz zu vielen anderen NoCs, die spezielle Mechanismen zur Behandlung von Echtzeit auf Ebene der Verbindungsstruktur verwenden, kann im CoreVA-MPSoC somit trotzdem die auf Durchsatz optimierte Best-Effort-Kommunikation verwendet werden.

Neben den Performanz-Abschätzungen des CoreVA-MPSoC-Compilers ist hierzu auch die explizite Kommunikation der CPUs (keine Caches) und die DMA-Funktionalität des NIs (nebenläufige Kommunikation) hilfreich.

8 Prototypische Implementierungen des CoreVA-MPSoCs

Zur funktionalen Verifikation von mikroelektronischen Systemen in realen Systemumgebungen ist eine prototypische Implementierung erforderlich. In Abschnitt 8.1 wird dazu zunächst die im Rahmen dieser Arbeit entstandene prototypische ASIC[1]-Implementierung des CoreVA-MPSoCs vorgestellt. Bei diesem ASIC-Prototyp wird die finale Zieltechnologie des CoreVA-MPSoCs eingesetzt, sodass eine genaue Verifikation der Schaltungsteile und des Gesamtsystems möglich ist. Außerdem können genaue Aussagen über die Verlustleistung und maximale Taktfrequenz eines gefertigten Chips getroffen werden. Aus diesem Grund ist der ASIC-Prototyp auch Bestandteil der Evaluierungen in den Kapiteln 5, 6 und 7.

Eine prototypische Abbildung des CoreVA-MPSoCs auf FPGAs wird in Abschnitt 8.2 vorgestellt. Der FPGA-Prototyp erlaubt zum einen eine erste Verifikation der Schaltung in einem realen System. Zum anderen bietet der FPGA-Prototyp bereits eine hohe Ausführungsgeschwindigkeit und Genauigkeit für die Softwareentwicklung, bevor der ASIC des MPSoCs zur Verfügung steht. Der im Rahmen dieser Arbeit entstandene Multi-FPGA-Prototyp des CoreVA-MPSoC ist außerdem Bestandteil einer Demoanwendung zur Ausführung eines Objekterkennungsalgorithmus auf dem CoreVA-MPSoC.

8.1 ASIC-Prototyp

Die Zielarchitektur des CoreVA-MPSoCs ist der ASIC. Zur Erstellung des ASIC-Prototypen für das CoreVA-MPSoC wird der bereits in Kapitel 3.3 vorgestellte hierarchische Hardwareentwurf auf Basis einer 28-nm-FD-SOI-Standardzellentechnologie[2] eingesetzt. In diesem Abschnitt wird zunächst die Implementierung eines Hardmakros für einen Cluster-Knoten vorgestellt. Dieser Cluster-Knoten besteht aus einem CPU-Cluster, dem Netzwerk-Interface und dem Router mit Verbindungen zu benachbarten Cluster-Knoten. Eine Vielzahl dieser Cluster-Knoten werden anschließend zum Gesamtsystem als ASIC-Prototyp des CoreVA-MPSoCs zusammengeschaltet. Das Layout dieses Gesamtsystems und mögliche I/O-Anbindungen sind in Abschnitt 8.1.2 beschrieben.

[1] *Application Specific Integrated Circuit*

[2] STMicroelectronics, 10 Metalllagen und Regular-VT Standardzellen

8.1.1 Makro eines Cluster-Knoten

Teilergebnisse des in diesem Abschnitt beschriebenen Cluster-Knotens sind bereits in den Kapiteln 5 und 7 aufgeführt. Im Folgenden werden Details für den synchronen Cluster-Knotens vorgestellt. Unterschiede zwischen synchronen und asynchronen Router sind anhand des Cluster-Knoten in Abschnitt 5.3.4 aufgezeigt.

Basisblock für das platzierte und verdrahtete Layout (P&R-Layout) ist das CPU-Makro mit zwei VLIW-Slots sowie je 16 kB Instruktions- und Datenspeicher (siehe Abschnitt 3.3.5). Das Cluster besteht aus vier dieser CPU-Makros, die über eine AXI-Crossbar miteinander verbunden sind. Zusätzlich werden acht Speicherbänke als gemeinsamer L1-Speicher integriert, sodass sich eine hybride Speicherarchitektur aus lokalen und gemeinsamen Datenspeichern ergibt (siehe Abschnitt 7.2). Neben den Cluster-Komponenten sind auch die NoC-Komponenten NI und Router enthalten. Das Layout korrespondiert daher mit dem Blockschaltbild aus Abbildung 7.3.

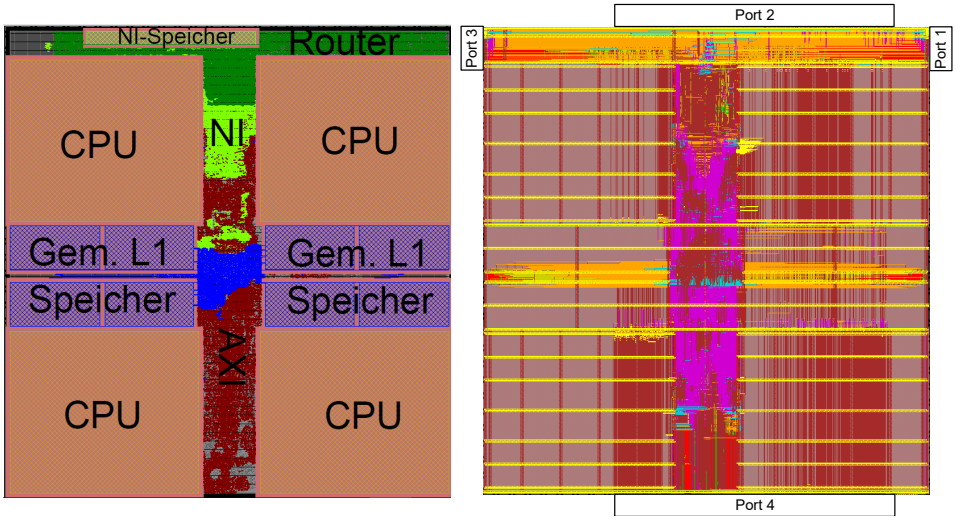


Abbildung 8.1: Physikalisches Layout eines Cluster-Knotens mit vier CPUs, gemeinsamem L1-Speicher und NoC-Komponenten. Links: Verdrahtetes Layout, Rechts: Platziertes Layout mit markierten NoC-Ports.

Auf der linken Seite in Abbildung 8.1 ist die platzierte Schaltung des Cluster-Knoten abgebildet. Die CoreVA-CPUs sind als Makros eingebunden. Vom CPU-Cluster sind der gemeinsame Speicher und dessen Verbindungsstruktur in blau und die AXI-Crossbar in rot markiert. Alle NoC-Komponenten sind in grün hervorgehoben, wobei der NI hellgrün und der Router dunkelgrün dargestellt ist. Das Schaubild des verdrahteten Layouts auf der rechten Seite in Abbildung 8.1 zeigt neben den einzelnen Leitungen auf den Metall-

lagen auch die vier externen Ports des Routers (2D-Mesh). Zur Realisierung der NoC-Links kann das P&R-Werkzeug Innovus über die CPU-Makros verdrahten (engl. routen), da die obersten zwei der zehn verfügbaren Metalllagen nicht innerhalb der CPU-Makros benötigt werden. Dies wird vom P&R-Werkzeug insbesondere für den südlichen NoC-Port ausgenutzt. Nach der Synthese hat ein CPU-Cluster eine maximale Taktfrequenz von 750 MHz im Worst-Case-Szenario[3]. Nach dem Platzieren und Verdrahten verringert sich die maximale Taktfrequenz auf 704 MHz, da nun genauere Informationen über Leitungsverzögerungen und zusätzliche Treiberzellen mit einbezogen werden können. Der kritische Pfad liegt hier in der Verbindungsstruktur des gemeinsamen L1-Speichers. Die Chipfläche des gesamten Cluster-Knotens beträgt 0,817 mm^2. Die Abschätzung der Leistungsaufnahme liegt bei etwa 125 mW für einen Cluster-Knoten. Detaillierte Ergebnisse über die Leistungsaufnahme und den Energiebedarf sind nach der Aufnahme von Schaltaktivitäten bereits in Abschnitt 7.3.2 aufgeführt.

8.1.2 Layout des CoreVA-MPSoCs

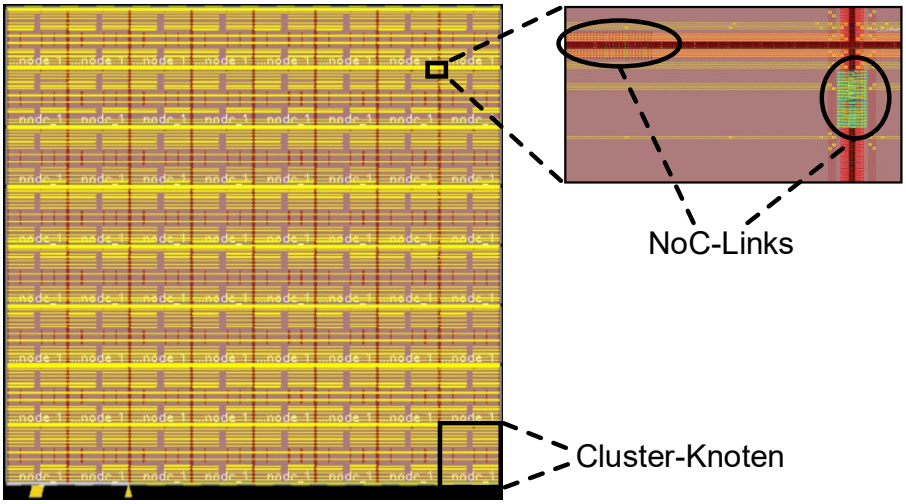


Abbildung 8.2: Physikalisches Layout eines CoreVA-MPSoC mit 8x8 Cluster-Knoten und insgesamt 256 CoreVA-CPUs.

Da die Links zwischen den Routern des NoCs in den Makros der Cluster-Knoten bereits bis an die Kanten geführt sind, kann in einem weiteren P&R-Schritt eine Verschaltung nahezu beliebig vieler Cluster-Knoten erstellt werden. Der hierarchische Entwurfsverlauf zur Verschaltung eines gesamten CoreVA-MPSoCs ist bereits in Abschnitt 3.3.5

[3] Worst-Case-Corner: 1.0 V, 125°C

beschrieben und in Abbildung 3.5 visualisiert. In Abbildung 8.2 ist ein Layout nach dem dritten Entwurfsschritt mit Verschaltung der Cluster-Knoten-Makros aus dem vorherigen Abschnitt zu sehen. Beispielhaft ist ein MPSoC mit 64 Cluster-Knoten (8x8-NoC) und damit insgesamt 256 CoreVA-CPUs abgebildet, um die Skalierbarkeit des Systems aufzuzeigen. Wie in Abschnitt 5.3.4 gezeigt, wird unter Verwendung des asynchronen oder mesochronen NoCs auch bei einem Design mit sehr vielen Cluster-Knoten die maximale Taktfrequenz eines CPU-Clusters von 704 MHz erreicht. Der vergrößerte Ausschnitt in Abbildung 8.2 zeigt die Leitungen zwischen zwei Cluster-Knoten für einen gesamten NoC-Link der horizontalen Verbindung und einen Ausschnitt der vertikalen Verbindung. Letzterer erstreckt sich in einem Cluster-Knoten über einen weiteren Bereich (siehe Abbildung 8.1). Zur Platzierung weiterer Treiberzellen für die Leitungen eines NoC-Links ist ein kleiner Abstand zwischen den Makros notwendig. Hinzu kommen weitere Leitungen für die Versorgungsspannung, um den Power-Ring der Makros auf oberster Ebene anzuschließen. Daraus ergibt sich ein Abstand von 21 µm zwischen zwei Cluster-Knoten. Dies führt zu einem Anstieg des benötigten Flächenbedarfs von 2%, sodass der Flächenbedarf eines Cluster-Knotens auf oberster Ebene 0,836 mm^2 beträgt.

Als I/O-Schnittstelle ist in diesem prototypischen Layout der AXI-Bus eines CPU-Clusters (unten links) direkt nach außen geführt. Um jedoch einen Chip zu realisieren, ist ein I/O-Ring für die Pins des Chips notwendig. Hierzu sind sowohl I/O-Zellen für die I/O-Schnittstelle als auch für die Versorgungsspannung erforderlich. Zum Zeitpunkt dieser Arbeit waren jedoch nur langsame Standard-I/O-Zellen (bis zu 200 MHz) verfügbar, sodass sich die Implementierung einer konkreten I/O-Schnittstelle stets als Flaschenhals des Systems herausstellt. Trotzdem kann bereits eine erste Abschätzung über die Anzahl von I/O-Zellen angegeben werden, die um einen CoreVA-MPSoC platziert werden können. Mit einer Breite von 40 µm für eine I/O-Zelle können pro Cluster-Knoten am Rand des MPSoC 22 I/O-Zellen platziert werden. Dies gilt für alle vier Seiten des MPSoCs. Bei einem 2D-Mesh mit n CPU-Clustern ergibt sich damit folgende Formel für die Anzahl I/O-Zellen:

$$\#IO = 88 \cdot \sqrt{n} \tag{8.1}$$

Für einen ASIC-Chip sind daher weiterführende Arbeiten an der I/O-Schnittstelle des CoreVA-MPSoCs erforderlich, sobald neue I/O-Zellen zur Verfügung stehen. Hierzu bieten sich z.B. differenzielle I/O-Zellen wie LVDS[4]- oder MGT[5]-Zellen an, um Datenraten im Gigabit-Bereich zu ermöglichen. Als I/O-Schnittstelle ist dann z.B. die Integration von hochperformanten IP-Kernen wie Speichercontrollern (z.B. DDR4) denkbar. Eine Alternative dazu ist der Einsatz einer proprietären Schnittstelle durch das Herausführen mehrerer Links von Routern am Rand des NoCs. Dies ermöglicht eine flexible Anpassung an anwendungsspezifische I/O-Standards auf einem angebundenen FPGA. Techniken wie Flip-Chip ermöglichen weitere Optimierungen, sodass I/O-Zellen

[4] *Low Voltage Differential Signaling*
[5] *Multi Giga-bit Tranceiver*

über den gesamten Chip verteilt werden können [85]. Dies kann die Anzahl der I/O-Zellen erhöhen, sodass in Zukunft eine Steigerung der I/O-Bandbreite des CoreVA-MPSoCs möglich ist. Bei sehr großen MPSoCs eignet sich die Flip-Chip-Technik ebenfalls für Power-Zellen, um so die Versorgungsspannung besser über den Chip zu verteilen.

8.1.3 Testchip

Ein Testchip mit zwei CoreVA-CPUs – der als Verarbeitungseinheit für einen Sensorknoten konzipiert ist – enthält mit dem asynchronen Router und dem NI zwei wesentliche Elemente dieser Arbeit. In diesem Chip verfügt jede CoreVA-CPU über 2-VLIW-Slots, 32 kB Instruktionsspeicher und 16 kB lokalen Datenspeicher. Des Weiteren verfügt dieser Chip über 32 kB gemeinsamen L1-Speicher. Hinzu kommen I/O-Schnittstellen wie SPI[6] und I^2C[7], sodass bei diesem Chip ein vollständiger I/O-Ring integriert ist. Der NI und asynchrone Router sind als Testschaltungen auf dem Chip integriert. Der asynchrone Router wird dabei als Makro konzipiert, dessen Links auf oberster Ebene miteinander kurzgeschlossen sind. Dies ermöglicht den Austausch von NoC-Paketen zwischen den CPUs, sodass alle Schaltungsteile des NoCs getestet werden können. Abbildung 8.3 zeigt das Layout dieses Testchips. Eine Herstellung des Chips ist für das Jahr 2019 geplant.

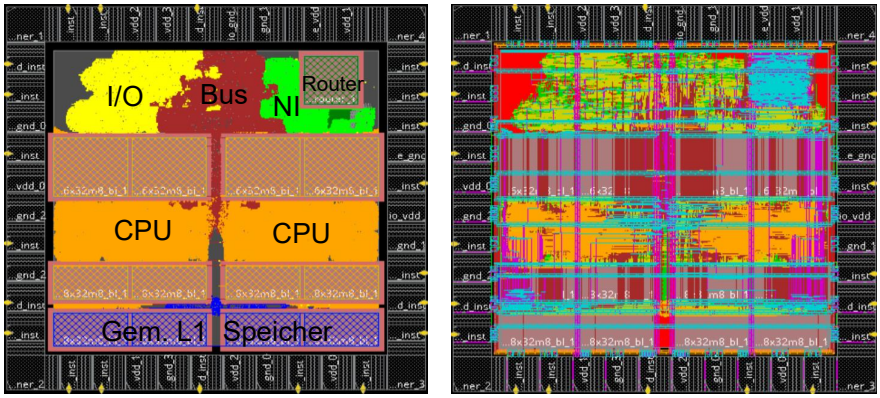


Abbildung 8.3: Physikalisches Layout des Testchips mit zwei CoreVA-CPUs, NI, asynchronem Router und I/O. Links: Platziertes Layout. Rechts: Verdrahtetes Layout.

[6] *Serial Peripheral Interface*
[7] *Inter Integrated Circuit*

8.2 FPGA-Prototyp

In diesem Abschnitt wird der FPGA-Prototyp des CoreVA-MPSoCs beschrieben. Als Prototyping-System wird die FPGA-basierte Plattform RAPTOR-XPress eingesetzt [142]. Die Plattform verfügt über eine hohe Modularität, da die Anzahl von FPGA-Modulen flexibel erweitert werden kann. Dies erlaubt die Abbildung verschiedenster CoreVA-MPSoC-Konfigurationen, da die Größe des CoreVA-MPSoCs zusammen mit der Verfügbarkeit von FPGA-Ressourcen im RAPTOR-System variiert werden kann. Im Folgenden wird zunächst die Abbildung eines Cluster-Knotens auf einem einzelnen FPGA vorgestellt. Anschließend wird auf die Implementierung größerer CoreVA-MPSoC-Konfigurationen auf mehreren FPGAs eingegangen. Hierzu wird das CoreVA-MPSoC verteilt auf einem FPGA-Cluster abgebildet, in dem die Kommunikationsstruktur des RAPTOR-Systems zwischen FPGA-Modulen verwendet wird. Ein solcher FPGA-Prototyp bietet bereits vor Fertigstellung eines ASICs die Möglichkeit, synthetisierte Schaltungsteile zu testen. Außerdem können schnelle Simulationen von Anwendungen auf dem CoreVA-MPSoC durchgeführt werden [31]. Durch die Anbindung von Peripheriegeräten ist eine Systemintegration des CoreVA-MPSoCs möglich. Im letzten Abschnitt wird daher eine Demo-Anwendung auf Basis des FPGA-Prototypen vorgestellt. In diesem Szenario wird der Videostrom einer angebundenen Kamera zum CoreVA-MPSoC geschickt, auf dem anschließend ein Objekterkennungsalgorithmus durchgeführt wird.

8.2.1 FPGA-Design

Ein einzelner Cluster-Knoten inklusive NoC-Komponenten ist auf einem Virtex-5-FPGA von Xilinx abgebildet. Dieser FPGA befindet sich auf dem DB-V5-Modul eines RAPTOR-XPress-Boards. Das RAPTOR-Modul DB-V5 integriert einen Virtex-5 LX100T- sowie einen LX30T-FPGA, welche in einer 65-nm-Technologie gefertigt sind [192]. Die HDL-Beschreibung des CoreVA-MPSoCs unterscheidet sich von der ASIC-Implementierung dabei lediglich durch die Verwendung anderer SRAM-Speicherblöcke, sodass alle anderen Schaltungsteile prototypisch getestet werden können.

Der Cluster-Knoten des CoreVA-MPSoC ist auf dem LX100T-FPGA des DB-V5 abgebildet. Der LX30T-FPGA wird als „Schnittstellen-FPGA“ eingesetzt und bietet die Anbindung von Peripherie-Schnittstellen wie PCI[8]-Express zum Host-PC. Die Anbindung an den „Schnittstellen-FPGA“ erfolgt beim CoreVA-MPSoC über ein XPS[9]-basiertes Peripherie-Subsystem und ermöglicht so die Kommunikation mit dem Host-PC. Das XPS-Subsystem ist dazu direkt an den AXI-Bus des CPU-Clusters angebunden. Zusätzlich ist ein 2 MB großer gemeinsamer Speicher im XPS-Subsystem integriert, der zur Kommunikation mit dem Host-PC verwendet werden kann.

[8] *Peripheral Component Interconnect*
[9] *Xilinx Platform Studio*

Die Konfigurationen des CPU-Clusters ist mit Ausnahme der Anzahl integrierter CoreVA-CPUs identisch zum Cluster-Knoten aus Abschnitt 8.1.1. Auf Grund limitierter Ressourcen auf dem V5-FPGA werden jedoch anstatt vier CoreVA-CPUs lediglich drei CPUs in einem CPU-Cluster integriert. Zur Emulation größerer CPU-Cluster sind neuere bzw. größere FPGAs notwendig. In [162] wird z.B. gezeigt, dass auf dem Virtex-7-FPGA des DB-V7 insgesamt 24 CoreVA-CPUs abgebildet werden können[10]. Zum Zeitpunkt dieser Arbeit standen diese DB-V7-Module jedoch nicht in größeren Stückzahlen zur Verfügung, sodass zur Erstellung der im nächsten Abschnitt vorgestellten Multi-FPGA-Lösung auf das DB-V5 zurückgegriffen wird.

Tabelle 8.1: Ressourcenbedarf des CoreVA-MPSoCs bei der Abbildung auf dem Virtex-5 LX100T-FPGA des DB-V5

Ressource	Slice-Register		Slice-LUT		BRAM	
Einzelne CPU	4506	7,0 %	12986	20,3 %	23	10,1 %
CPU-Cluster	18173	28,4 %	43564	68,1 %	69	30,3 %
Router	3379	5,3 %	2721	4,3 %	0	0,0 %
NI	1218	1,9 %	1743	2,7 %	2	0,9 %
XPS-Subsystem	5766	9,0 %	5778	9,0 %	101	44,3 %
MGT-Interface	3064	4,8 %	1503	2,3 %	36	15,8 %
Gesamt	32396	50,6 %	57690	90,1 %	208	91,2 %

In Tabelle 8.1 ist der Ressourcenbedarf eines Cluster-Knotens des CoreVA-MPSoCs auf einem Virtex-5-FPGA dargestellt. Der Ressourcenbedarf des CoreVA-MPSoCs ist nach der Synthese sowie dem Platzieren und Verdrahten mit den Werkzeugen Xilinx ISE 14.7 und Synopsys Synplify Premier bei einer maximalen Taktfrequenz von 100 MHz aufgenommen. Die rekonfigurierbaren Logikressourcen des Virtex-5 sind in sogenannte Slices aufgeteilt. Jedes Slice enthält vier LUTs (Lookup Table) sowie vier Register [192]. Ein BRAM-Block des FPGAs kann bis zu 36 kbit an Daten speichern. Es zeigt sich, dass das System durch die Logikressourcen limitiert ist, da 90,1 % aller verfügbaren LUTs verwendet werden. Dominiert wird der Ressourcenbedarf durch die CoreVA-CPUs, bei der eine einzelne CPU insgesamt 20,3 % aller LUTs benötigt. Neben dem XPS-Subsystem befindet sich auch das MGT-Interface im Design eines Cluster-Knotens, welches zur Kommunikation mit weiteren FPGA-Modulen über das NoC verwendet wird (siehe nächsten Abschnitt).

8.2.2 Multi-FPGA-Lösung

Bereits auf einem einzelnen RAPTOR-XPress-Board können vier DB-V5 Module integriert werden. Eine zusätzliche Skalierung ist durch das an der Universität Bielefeld

[10] Ohne NoC-Komponenten und Peripherie-Schnittstellen

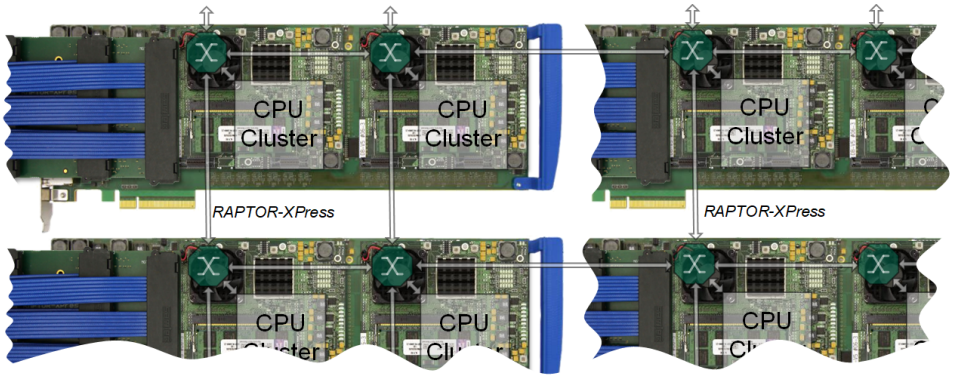


Abbildung 8.4: Schematische Abbildung des CoreVA-MPSoC auf dem FPGA-Cluster, bestehend aus RAPTOR-XPress Boards, bestückt mit DB-V5 Modulen.

entwickelte FPGA-Cluster möglich. Bei diesem FPGA-Cluster können mehrere RAPTOR-XPress-Boards über serielle Hochgeschwindigkeitsverbindungen (MGTs) miteinander verbunden werden [150]. Hierzu sind beim CoreVA-MPSoC die vier NoC-Links des integrierten Routers über Aurora-IP-Kerne von Xilinx an die MGT-Schnittstellen des FPGAs angebunden. Hierdurch kann die NoC-Kommunikation über Modul-Grenzen hinweg stattfinden, sodass ein großes CoreVA-MPSoC über mehrere Module und RAPTOR-Boards verteilt werden kann. Schematisch ist dies in Abbildung 8.4 dargestellt. Zu berücksichtigen ist jedoch, dass sich durch die Latenz der Aurora-Schnittstelle eine zusätzliche Latenz von 45 Taktzyklen bei einer Paketübertragung pro NoC-Link einstellt (bei der maximalen Taktfrequenz von 100 MHz). Für eine Emulation des CoreVA-MPSoC mit höherer Zyklengenauigkeit zum ASIC kann jedoch die Taktfrequenz CoreVA-MPSoC auf dem FPGA verringert werden, sodass die Latenz der NoC-Links auf zwei bis drei Taktzyklen minimiert werden kann. Der maximale Durchsatz des NoCs kann hingegen bei allen Konfigurationen voll ausgeschöpft werden.

8.2.3 Demonstrator

Zur funktionalen Verifikation des Multi-FPGA-Prototypen ist im Rahmen dieser Arbeit ein Demosystem auf Basis eines RAPTOR-XPress-Boards entstanden, das mit vier DB-V5 Modulen bestückt ist. Dazu ist in einem Studentenprojekt im Rahmen der Vorlesung Bildverarbeitung der Universität Bielefeld eine Anwendung zur Objekterkennung in der Sprache StreamIt für das CoreVA-MPSoC entstanden. Mit Hilfe einer Kamera am Host-PC kann über die PCI-Schnittstelle zum RAPTOR-Board ein Videodatenstrom zum CoreVA-MPSoC übertragen werden. Dort wird die Anwendung zur Objektdetektion auf einem verteilten CoreVA-MPSoC-System mit insgesamt zwölf CoreVA-CPUs ausgeführt.

Die Anwendung kombiniert dabei verschiedene Basistechniken der Bildverarbeitung zur Erstellung eines Referenzbildes. Dazu wird jedes Bild in partielle Ausschnitte aufgeteilt, um so eine parallele Verarbeitung zu ermöglichen. Anschließend wird eine morphologische Dilatation und Erosion auf jedem Ausschnitt durchgeführt. Um die Maße des Bildes nicht zu verändern, werden dazu die Ränder jedes Ausschnitts gespiegelt. Zum Schluss werden die Regionen gelabelt und nur die größten Regionen mit Unterschieden zum Referenzbild zurückgegeben. Der Aufbau des Demonstrators mit RAPTOR-Board und visualisiertem Ergebnis der Objekterkennung, die auf dem CoreVA-MPSoC ausgeführt wird, ist in Abbildung 8.5 zu sehen. Das System zeigt im Vordergrund das RAPTOR-XPress-Board mit vier DB-V5 Modulen, wobei hier nur die zwei vorderen sichtbar sind. Im Hintergrund ist das Ergebnis der Objektdetektion zu sehen, bei dem beispielhaft eine Hand in fünf Regionen unten rechts erkannt wird.

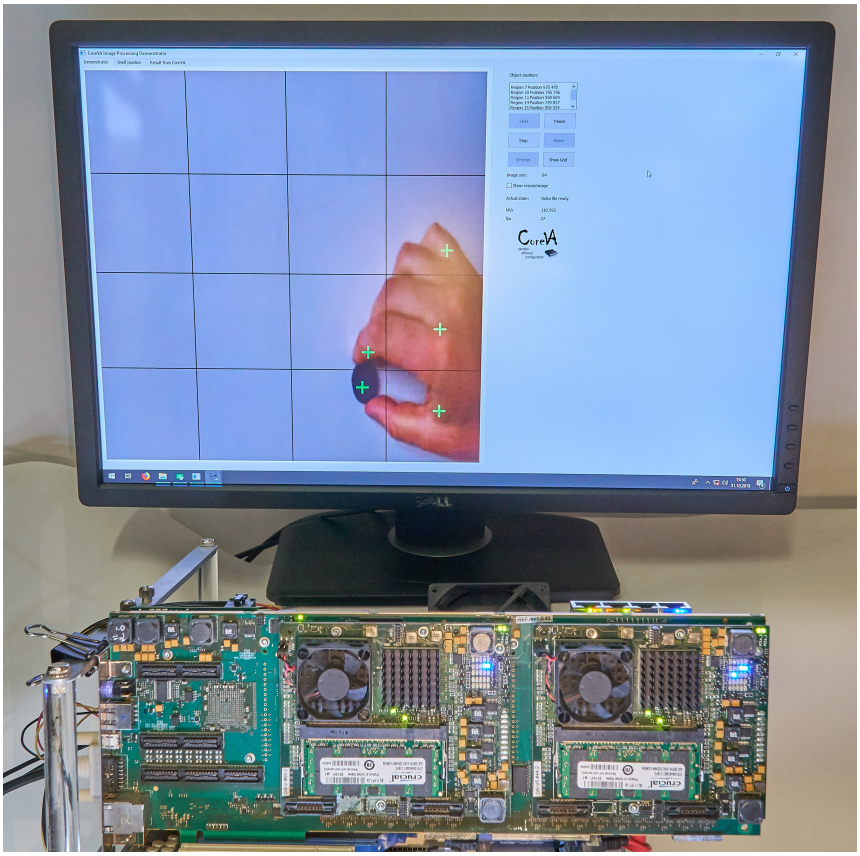


Abbildung 8.5: Demoaufbau zur Objektdetektion.

8.3 Zusammenfassung

In diesem Kapitel wurden die prototypischen Implementierungen des CoreVA-MPSoCs auf FPGA- und ASIC-Basis beschrieben. Dazu wurde ein platziertes und verdrahtetes Hardmakro eines Cluster-Knotens in einer 28-nm-FD-SOI-Standardzellentechnologie erstellt. Dieses Makro enthält ein CPU-Cluster bestehend aus vier CoreVA-CPUs verbunden über eine AXI-Crossbar und optionalem gemeinsamen L1-Speicher sowie jeweils einen NI und Router. Ein solcher Cluster-Knoten hat mit synchronem NoC eine Fläche von 0,817 mm^2 (0,792 mm^2 mit asynchronem Router). Die maximale Taktfrequenz ist durch den gemeinsamen L1-Speicher vorgegeben und liegt im Worst-Case-Szenarion bei 704 MHz. Für einen vollständigen Knoten liegt die Leistungsaufnahme bei etwa 125 mW. Alle Links zwischen den Routern sind bereits bis an die Kanten des Cluster-Knoten-Makros geführt, sodass in einem weiteren Entwurfsschritt nahezu beliebig viele dieser Cluster-Knoten zu einem MPSoC verbunden werden können. Die Ergebnisse dieser Arbeit ermöglichen die Fertigung eines CoreVA-MPSoC-Chips mit einer skalierbaren Anzahl von CoreVA-CPUs

Im zweiten Teil dieses Kapitels wurde der in dieser Arbeit entwickelte Prototyp auf Basis eines FPGA-Clusters vorgestellt. Jeweils ein Cluster-Knoten mit drei CoreVA-CPUs wurde dazu auf einem Virtex-5 FX100T-FPGA von Xilinx abgebildet. Der FPGA-Cluster koppelt über serielle Hochgeschwindigkeitsverbindungen 16 RAPTOR-XPress-Basisboards mit insgesamt 64 Virtex-5 FX100T-FPGAs. Durch diese Hochgeschwindigkeitsverbindungen wurden die NoC-Links zwischen den Routern des CoreVA-MPSoCs realisiert. Dies ermöglicht eine Emulation eines CoreVA-MPSoCs mit insgesamt 192 CoreVA-CPUs.

Des Weiteren entstand im Rahmen dieser Arbeit ein Demosystem, bestehend aus einem RAPTOR-XPress mit vier FPGA-Modulen. Im Demosystem wird auf dem Videostrom einer angebundenen Kamera eine Objekterkennung durchgeführt, die auf dem emulierten CoreVA-MPSoC ausgeführt wird.

9 Zusammenfassung und Ausblick

Aufgrund ihrer Flexibilität und Programmierbarkeit finden Multi-Prozessorsysteme auf einem einzelnen Chip (MPSoCs) in eingebetteten Systemen zunehmend Verwendung. Many-Core-Systeme mit vielen hunderten bis tausenden ressourceneffizienten CPU-Kernen versprechen dazu eine Kombination aus hoher Rechenleistung und geringem Energiebedarf. Mögliche Einsatzszenarios für Many-Core-Systeme sind z.B. mobile Roboter, die durch eine ressourceneffiziente Datenverarbeitung über einen langen Zeitraum autonom betrieben werden können. Typische Anwendungen kommen in diesem Bereich aus der Signalverarbeitung sowie Sprach- oder Bilderkennung. Innerhalb der AG Kognitronik und Sensorik der Universität Bielefeld wird dazu das Many-Core-System CoreVA-MPSoC entwickelt. Das CoreVA-MPSoC verfügt über eine hierarchische Verbindungsstruktur, die eine Integration von hunderten ressourceneffizienten CoreVA-CPUs auf einem Chip ermöglicht. In einem CPU-Cluster sind zunächst mehrere dieser CoreVA-CPUs über eine eng gekoppelte Bus-Struktur verbunden. Ein On-Chip-Netzwerk (NoC) verbindet schließlich mehrere dieser CPU-Cluster zu einem MPSoC mit mehreren hundert oder tausend CPUs. In dieser Arbeit wurde eine ressourceneffiziente NoC-Architektur für den Einsatz im CoreVA-MPSoC entworfen und analysiert. Die Entwicklung und Entwurfsraumexploration wurde mit Hilfe einer 28-nm-FD-SOI-Standardzellenbibliothek auf verschiedenen Ebenen durchgeführt. Im Folgenden wird die vorliegende Arbeit auf Basis der einzelnen Kapitel zusammengefasst.

Stand der Technik von NoC-Architekturen

In Kapitel 2 fand die Einführung in den Stand der Technik von NoC-Architekturen statt. Zunächst sind dazu die allgemeinen Eigenschaften wie Topologie, Routing und Flusskontrolle dargestellt worden. Neben diesen drei grundlegenden Eigenschaften wurde mit dem GALS[1]-Paradigma auch ein Ansatz zur physikalischen Umsetzung der Skalierbarkeit von NoCs vorgestellt. Im zweiten Teil des Kapitels sind aktuelle NoC-Architekturen und NoC-basierte MPSoCs aus Forschung und Industrie aufgeführt worden. Beim Vergleich mit diesen MPSoCs aus dem Stand der Technik zeigte das CoreVA-MPSoC einen guten Kompromiss aus Flächen- und Energieeffizienz und damit eine hohe Ressourceneffizienz. So erreicht ein CoreVA-MPSoC mit 64 CPUs, ohne I/O-Zellen, 89,6 GOPS bei einer Fläche von 14,4 mm^2 und einer Leistungsaufnahme von etwa 2 W bei 700 MHz.

[1] *Global Asynchron Lokal Synchron*

Grundlagen und Werkzeugkette des CoreVA-MPSoC
Die Vorstellung der Grundlagen und der Werkzeugkette des CoreVA-MPSoC fand in Kapitel 3 statt. Dazu wurde zunächst die Hardwarearchitektur der CoreVA-CPU und die des CPU-Clusters vorgestellt. Die CoreVA-CPU basiert auf einer ressourceneffizienten VLIW-Architektur und eignet sich daher gut für den Einsatz in MPSoCs energiebeschränkter eingebetteter Systeme. Das CPU-Cluster verfügt über eine Bus-Struktur, die zur Entwurfszeit als geteilter Bus oder Crossbar des AXI- oder Wishbone-Standards konfiguriert werden kann. Des Weiteren wurde der hochautomatisierte Hardware- und Software-Entwurfsablauf des CoreVA-MPSoCs beschrieben, der im Rahmen dieser Arbeit mitentwickelt und eingesetzt wurde. Der Hardware-Entwurfsablauf ermöglichte die Realisierung und Entwurfsraumexploration eines Chip-Prototypen des CoreVA-MPSoCs in einer 28-nm-FD-SOI[2]-Standardzellentechnologie. Die Software-Entwurfsumgebung basiert auf Compilern für die Sprachen C und StreamIt. Für Letztere ermöglicht der CoreVA-MPSoC-Compiler eine automatisierte Partitionierung von Streaming-Anwendungen. Grundlage hierfür ist ein neuartiges Software-Kommunikationsmodell mit blockbasiertem Synchronisierungsverfahren. Dies erlaubt eine effiziente Synchronisierung sowohl im CPU-Cluster als auch im NoC des CoreVA-MPSoCs. Eine Simulation bzw. Emulation des CoreVA-MPSoCs ist auf verschiedenen Abstraktionsebenen möglich, die ebenfalls in Kapitel 3 beschrieben sind.

Bewertungsmaße zur Entwurfsraumexploration von NoC-Architekturen
In Kapitel 4 wurden die Bewertungsmaße für NoC-Architekturen in eingebetteten Multiprozessoren eingeführt. Bewertungsmaße sind neben der Chipfläche und Energie auch die Performanz in Form von Latenz und Durchsatz. Auf Basis dieser Bewertungsmaße erfolgte die Entwurfsraumexploration für die verschiedenen Systemkomponenten und NoC-Architekturen dieser Arbeit. Zusätzlich wurden Benchmarks aus dem Bereich von Streaming-Anwendungen, wie z.B. aus der Signal- und Bildverarbeitung vorgestellt. Diese erlauben eine Bewertung des NoCs auf Systemebene für verschiedene Anwendungen.

Verbindungsstrukturen für NoC-Architekturen
Verschiedene Verbindungsstrukturen für eingebettete NoC-Architekturen wurden in Kapitel 5 vorgestellt und diskutiert. Verbindungsstrukturen stellen die Kommunikation innerhalb des Netzwerks bereit und bestehen im Fall von NoCs insbesondere aus Routern und deren Verbindungen untereinander. Zunächst erfolgte eine Diskussion der Architekturkonzepte von aktuellen Routern, auf dessen Basis die Architektur und Implementierung des im CoreVA-MPSoC eingesetzten Routers vorgestellt wurde. Beim Router des CoreVA-MPSoCs wird Wormhole-Switching in Verbindung mit einer ON/OFF-Flusskontrolle eingesetzt, da hierdurch bei geringem Ressourcenaufwand ein höherer durchschnittlicher Durchsatz erzielt werden kann als beim Circuit-Switching. Das ressourceneffiziente XY-Routing bietet durch das gedächtnislose und determinis-

[2] *Fully-Depleted Silicon-on-Insulator*

tische Verfahren eine ideale Voraussetzung für Echtzeitanwendungen und lässt sich zudem gut vom CoreVA-MPSoC-Compiler abschätzen. Der Router im CoreVA-MPSoC hat eine minimale Latenz von zwei Taktzyklen pro Paketsegment (Flit). Im Anschluss an die Router-Architektur wurden mit 2D-Mesh, Torus, Honeycomb und Ring verschiedene Topologien zur Verschaltung der Router vorgestellt. Es zeigte sich, dass sowohl die Chipfläche als auch die Leistungsaufnahme insbesondere durch die Anzahl der I/O-Ports pro Router bestimmt sind. Hierdurch haben Topologien wie das 2D-Mesh und Torus einen höheren Ressourcenbedarf als Honeycomb (74% vom 2D-Mesh) und Ring (52% vom 2D-Mesh). Bei der Performanz stellt sich bei drei von zehn Anwendungen die Ring-Topologie sowie bei einer Anwendung Honeycomb als nachteilig heraus. Zum klassischen 2D-Mesh – welches im weiteren Verlauf der Arbeit eingesetzt wurde – stellt Honeycomb aufgrund des geringeren Ressourcenbedarfs jedoch eine interessante Alternative dar. Um den Ressourcenbedarf weiter zu verringern und auch die Skalierbarkeit des NoCs zu erhöhen, erfolgte die Untersuchung verschiedener GALS-Methoden. Verglichen wurden dazu die drei Ansätze eines synchronen, mesochronen und asynchronen NoCs. Für das mesochrone NoC wurden spezielle Synchronisierer zwischen den Links implementiert und eingesetzt. Bei dem im Rahmen dieser Arbeit entworfenen asynchronen NoC erfolgte die Realisierung der Router vollständig durch asynchrone Schaltungselemente. Die Ergebnisse haben gezeigt, dass durch moderne Funktionalitäten der Entwicklungswerkzeuge (z.B. Cadence CCOpt-Flow) auch für synchrone NoCs weiterhin eine gute Skalierung von MPSoCs möglich ist. Dennoch zeigte das asynchrone NoC gegenüber den anderen beiden Implementierungen einen geringeren Flächen- und Energiebedarf, bei vergleichbarer Performanz. Beim Vergleich eines platzierten und verdrahteten MPSoCs hat das asynchrone NoC einen um 3,1% geringeren Flächenbedarf für das Gesamtsystem. Die Leistungsaufnahme eines einzelnen asynchronen Routers beträgt 22,4% (0,94 mW im Ruhezustand) bzw. 53% (3,94 mW während Kommunikation) von der Leistungsaufnahme eines Takt-basierten Routers. Bei den Untersuchungen des globalen Taktbaums für ein MPSoC mit 256 CPUs zeigte das asynchrone NoC mit 5,78 mW eine um 25% geringere Leistungsaufnahme als das synchrone und mesochrone NoC (ca. 7,7 mW). Das asynchrone NoC eignet sich daher besonders gut für den Einsatz in energiebeschränkten Systemen.

Netzwerk-Schnittstellen für NoC-Architekturen

Neben der eigentlichen Verbindungsstruktur des NoCs, die im vorangegangenen Kapitel vorgestellt worden ist, hat auch die Netzwerk-Schnittstelle (*Network Interface*, NI) vom Prozessorsystem zum NoC einen großen Einfluss auf die Effizienz von NoC-Architekturen in MPSoCs (siehe Kapitel 6). Für eine hohe Effizienz der Kommunikation im Gesamtsystem musste die Netzwerk-Schnittstelle und das Programmier- bzw. Kommunikationsmodell gut aufeinander abgestimmt werden. Aktuelle NIs sind entweder paketbasierte Streaming-Schnittstellen oder auf einem globalen Adressraum basierende NIs. Eine paketbasierte Streaming-Schnittstelle bietet eine hohe Performanz im NoC, insbesondere wenn man das NoC isoliert betrachtet. Die Aufgabe des NIs ist es, die

adressbasierte Kommunikation der CPUs direkt in die paketbasierte Kommunikation des NoCs zu überführen, z.B. durch FIFOs oder dedizierte Paketspeicher. Hierbei besteht häufig kein wahlfreier Speicherzugriff auf Paketdaten. Dies erhöht bei vielen Anwendungen die Softwarekosten, die für Kommunikation aufgewendet werden müssen. Um die Softwarekosten zu senken, sind diese NIs häufig für Cache-basierte Systeme ausgelegt. Bei NI-Architekturen mit gemeinsamem globalen Adressraum im gesamten MPSoC besteht dieser Nachteil nicht. Hier werden Daten nicht als Pakete durch das NoC geschickt, sondern CPU-Zugriffe direkt zusammen mit der Adresse an das NoC übergeben. Dies senkt zwar die Softwarekosten, schränkt jedoch die Skalierbarkeit oder den maximalen Datendurchsatz des NoCs ein. Der in dieser Arbeit neu entwickelte NI des CoreVA-MPSoCs kombiniert beide Ansätze. Im NoC wird eine hochperformante Paket- oder Stream-basierte Kommunikation mit Best-Effort-Datenübertragung eingesetzt. Den CPUs wiederum bietet der NI eine Adress- und Speicher-basierte DMA-Funktionalität, welches die Kommunikationskosten der Software minimiert. Dazu unterstützt der NI eine flexible Verwaltung unabhängiger Transaktionskanäle, um so eine hohe Skalierbarkeit und eine schnelle Kommunikation zu gewährleisten. Die Entwurfsraumexploration in Kapitel 6 hat gezeigt, dass sich ein neuartiger semi-statischer Ansatz mit sehr vielen physikalisch vorhandenen Kommunikationskanälen besonders gut eignet (256 Kanäle für die untersuchten Streaming-Anwendungen). Dieser Ansatz verursacht im Vergleich zu einem dynamischen Ansatz mit wenigen Kanälen geringere Softwarekosten auf den CPUs und zeichnet sich zudem durch ein deterministisches Verhalten aus. Der NI des CoreVA-MPSoCs hat einen Flächenbedarf von 0,027 mm^2 bei einer Leistungsaufnahme von etwa 5 mW.

NoC-Architekturen auf Systemebene

Aufbauend auf den Ergebnissen der vorangegangenen Kapiteln, wurde in Kapitel 7 das NoC auf Systemebene betrachtet und dessen Zusammenspiel mit anderen Systemkomponenten wie CPUs, Speicher und Cluster-Verbindungsstrukturen untersucht. Als Bus-Verbindungsstruktur im CPU-Cluster stellt sich die AXI-Crossbar in Verbindung mit dem NoC als vorteilhaft dar. Der AXI-Standard ermöglicht unabhängige Schreib- und Lesekanäle, sodass Pakete vom NI nebenläufig empfangen und versendet werden können. Bei der Untersuchung verschiedener Speicherarchitekturen hat sich die Anbindung eines gemeinsamen L1-Speichers innerhalb eines CPU-Clusters an das NoC gegenüber einer Architektur mit ausschließlich lokalen L1-Speichern als effizienter erwiesen. Untersucht wurden drei verschiedene Konfigurationen: Nur lokale L1-Speicher an jeder CPU, nur gemeinsame L1-Speicher innerhalb eines CPU-Clusters und eine Hybrid-Variante aus beiden. Sowohl beim Flächen- als auch beim Energiebedarf zeigen alle drei Konfigurationen nur marginale Unterschiede. Insbesondere bei der Performanz auf Anwendungsebene zeigen die beiden Konfigurationen mit gemeinsamen L1-Speichern jedoch bessere Ergebnisse. Durch die größere Flexibilität bei der Speicherverwaltung profitieren insbesondere Streaming-Anwendungen mit einem inhomogenen Speicherbedarf pro CPU, wie z.B. LowPassFilter. Bei der Ausführung aller zehn Bench-

marks auf drei verschiedenen MPSoC-Konfigurationen mit 32 CPUs (2x1x16, 2x2x8 und 4x2x4) zeigt die Verwendung des gemeinsamen L1-Speichers im Durchschnitt eine um 17,2% höhere Performanz als die Verwendung von ausschließlich lokalen Speichern. Neben den Untersuchungen auf Systemebene wurde in diesem Kapitel auch eine abstrakte Modellierung für das CoreVA-MPSoC mit Fokus auf die NoC-Architektur und -Kommunikation vorgestellt. Zum einen ermöglicht das vorgestellte Modell zur Abschätzung des Flächenbedarfs bereits zu einem frühen Entwicklungszeitpunkt eine Prognose über die Chipfläche verschiedener MPSoC-Konfigurationen. Zum anderen unterstützt das Modell zur Bestimmung der Performanz und des Energiebedarfs von Streaming-Anwendungen die Arbeit des CoreVA-MPSoC-Compilers. Mit Hilfe des Modells können Anwendungen in kurzer Zeit anhand ihres Durchsatzes, der Latenz oder des Energiebedarfs bewertet und so optimal auf dem CoreVA-MPSoC partitioniert werden. Insbesondere die Latenzbestimmungen des CoreVA-MPSoC-Compilers werden auch für die Ausführung echtzeitkritischer Anwendungen auf dem CoreVA-MPSoC eingesetzt. Im Gegensatz zu vielen anderen NoCs, die spezielle Mechanismen zur Sicherstellung der Echtzeitanforderungen auf Ebene der Verbindungsstruktur verwenden, kann im CoreVA-MPSoC somit trotzdem die auf Durchsatz optimierte Best-Effort-Kommunikation verwendet werden. Neben den Performanz-Abschätzungen des CoreVA-MPSoC-Compilers ist hierzu auch die explizite Kommunikation der CPUs (keine Caches) und die DMA-Funktionalität des NIs (nebenläufige Kommunikation) hilfreich.

Prototypische Implementierungen des CoreVA-MPSoCs

Die Vorstellung der prototypischen Implementierungen des CoreVA-MPSoCs auf FPGA- und ASIC-Basis erfolgte in Kapitel 8. Für eine mögliche Fertigung in einer 28-nm-FD-SOI-Standardzellentechnologie wurden platzierte und verdrahtete Hardmakros eines Cluster-Knotens erstellt. Diese Makros enthalten ein CPU-Cluster bestehend aus vier CoreVA-CPUs verbunden über eine AXI-Crossbar und optionalem gemeinsamen L1-Speicher sowie jeweils einen NI und Router. Ein solcher Cluster-Knoten hat mit synchronem NoC eine Fläche von 0,817 mm^2 (0,792 mm^2 mit asynchronem Router). Die maximale Taktfrequenz ist durch den gemeinsamen L1-Speicher vorgegeben und liegt im Worst-Case-Szenarion bei 704 MHz. Für einen vollständigen Knoten liegt die Leistungsaufnahme bei etwa 125 mW. Die Links zwischen den Routern sind bereits bis an die Kanten des Cluster-Knoten-Makros geführt, sodass in einem weiteren Entwurfsschritt nahezu beliebig viele dieser Cluster-Knoten zu einem MPSoC verbunden werden können. Des Weiteren entstand im Rahmen dieser Arbeit ein Prototyp auf Basis eines FPGA-Clusters. Jeweils ein Cluster-Knoten mit drei CoreVA-CPUs wurde dazu auf einem Virtex-5 FX100T-FPGA von Xilinx abgebildet. Der FPGA-Cluster koppelt über serielle Hochgeschwindigkeitsverbindungen 16 RAPTOR-XPress-Basisboards mit insgesamt 64 Virtex-5 FX100T-FPGAs. Durch diese Hochgeschwindigkeitsverbindungen wurden die NoC-Links zwischen den Routern des CoreVA-MPSoCs realisiert. Dies ermöglicht eine Emulation eines CoreVA-MPSoCs mit insgesamt 192 CoreVA-CPUs.

Verwertbarkeit und weiterführende Arbeiten
Die in dieser Arbeit entwickelten und evaluierten Konzepte von NoC-Architekturen eignen sich allgemein für den Einsatz in Many-Cores eingebetteter energiebeschränkter Systeme. Zudem ermöglichen die praktischen Ergebnisse in einer 28-nm-FD-SOI-Standardzellentechnologie eine zeitnahe Fertigstellung eines Many-Core-Chips mit einer skalierbaren Zahl von CoreVA-CPUs. Der in dieser Arbeit entwickelte CoreVA-MPSoC-Chip bietet dabei einen bestmöglichen Kompromiss zwischen Leistungsfähigkeit und Energiebedarf unter Berücksichtigung der Chipfläche. Neben einer Integration eines ASIC-Moduls in das Prototyping-System RAPTOR ist auch ein Einsatz eines ASICs in den mobilen Robotern AMiRo und Hector denkbar, die an der Universität Bielefeld entwickelt werden. Dies ermöglicht eine effiziente Verarbeitung von Signal- und Bildverarbeitungsalgorithmen auf diesen Systemen.

Für einen ASIC-Chip sind jedoch noch weiterführende Arbeiten an der I/O-Schnittstelle des CoreVA-MPSoCs nötig. Um die hohe Rechenleistung eines CoreVA-MPSoCs mit vielen CPUs auch ausnutzen zu können, ist eine hohe I/O-Bandbreite erforderlich. Denkbar ist hier z.B. die Integration von IP-Kernen wie Speichercontroller (z.B. DDR4) oder Multi-Gigabit-Transceiver, falls diese verfügbar und finanzierbar werden. Eine andere Option ist die Verwendung hochperformanter paralleler I/O-Zellen (z.B. LVDS-Zellen), um so die Links von Routern am Rand des NoCs proprietär nach außen zu führen. Dies ermöglicht eine flexible Anpassung an anwendungsspezifische I/O-Standards auf einem angebundenen FPGA.

Weitere Optimierungen für die Ressourceneffizienz des CoreVA-MPSoCs sind durch zusätzliche Stromspartechniken zu erreichen. So bietet das in dieser Arbeit entwickelte asynchrone NoC eine leichte Integration von zusätzlichen Mechanismen wie *Dynamic Voltage and Frequency Scaling*, DVFS. Das asynchrone NoC entkoppelt die CPU-Cluster von einem gemeinsamen Takt, sodass deren Taktfrequenzen in Zukunft dynamisch angepasst werden können. Zusätzlich dazu kann mit Hilfe zusätzlicher Level-Shifter in diesem Fall auch die Versorgungsspannung einzelner Cluster angepasst werden, um so die Leistungsaufnahme dynamisch zu verringern. Eine weitere Möglichkeit zur Steigerung der Energieeffizienz ist der Einsatz von Power-Switches zur Reduzierung der statischen Verlustleistung. Damit können z.B. ganze CPU-Cluster abgeschaltet werden, die für eine bestimmte Anwendung nicht erforderlich sind. Mit Hilfe spezieller Steuer-Flits über das NoC können eine Taktanpassungen und das An- und Abschalten von CPU-Clustern durch andere CPUs im System angestoßen werden. All diese Mechanismen können die Energieeffizienz des CoreVA-MPSoCs weiter steigern, um so z.B. längere Batterielaufzeiten für mobile Roboter wie AMiRo und Hector zu ermöglichen.

Abbildungsverzeichnis

1.1 An der Universität Bielefeld entwickelte Roboter 1
1.2 Blockschaltbild des hierarchischen CoreVA-MPSoCs 5

2.1 Klassifizierung am Beispiel einer 2D-Mesh Topologie 11
2.2 Beispiele Topologien 12
2.3 Taktverteilung bei MPSoCs mit einer Taktdomäne 21
2.4 Taktverteilung bei einem mesochronen MPSoC 24

3.1 Blockschaltbild der Pipeline des CoreVA-Prozessors 38
3.2 Blockschaltbild der CoreVA-CPU mit Speicher und Hardwareerweiterungen 40
3.3 Blockschaltbild eines CPU-Clusters mit zwei CPUs und I/O-Schnittstelle [162] 42
3.4 Hardware-Entwurfsablauf für Standardzellen-Technologien nach [94, S. 673] 44
3.5 Layout eines 4x4 2D-Mesh MPSoCs, unter Verwendung des hierarchischen Entwurfsablaufs mit CPU und Cluster-Knoten Makros 48
3.6 Layouts von verschiedenen Makros der CoreVA-CPU 49
3.7 Blockbasierte Kommunikation im CoreVA-MPSoCs [162] 55
3.8 Der CoreVA-MPSoC-Compiler partitioniert den Graphen einer StreamIt-Anwendung auf das CoreVA-MPSoC 57
3.9 Optimierungsziele des CoreVA-MPSoC-Compiler 58

5.1 NoC-Verbindungsstruktur im CoreVA-MPSoC (farblich hervorgehoben) 73
5.2 Aufbau des Routers im CoreVA-MPSoC für eine 2D-mesh Topologie . 76
5.3 Beispielhafter Aufbau eines Flit im CoreVA-MPSoC 77
5.4 Aufbau des Routers im CoreVA-MPSoC mit virtuellen Kanälen 78
5.5 Aufbau der Crossbar im CoreVA-MPSoC am Beispiel für Ausgangsport 1, mit Round-Robin-Arbitrierung (RR). 80
5.6 2D-Torus-Topologie im CoreVA-MPSoC 83
5.7 Ring- und Honeycomb-Topologie im CoreVA-MPSoC 84
5.8 Flächenbedarf eines Routers für verschiedene Topologien im CoreVA-MPSoC 86
5.9 Leistungsaufnahme eines Routers für verschiedene Topologien im CoreVA-MPSoC 87

5.10 Beschleunigung von 40 CPUs gegenüber einer CPU, bei Verwendung verschiedener Topologien im CoreVA-MPSoC 88
5.11 Architektur des Tightly Coupled Mesochronous Synchronizer (TCMS) wie sie im CoreVA-MPSoC umgesetzt ist. 93
5.12 Beispiel für das Schaltverhalten im TCMS. 94
5.13 Signalverlauf zur Einstellung der Zähler im TCMS. 95
5.14 Architektur des asynchronen Routers. 97
5.15 Aufbau und Schaltverlauf für ein Mousetrap-FIFO der Tiefe Zwei. . . 98
5.16 Split- und Join-Elemente zum Abzweigen und Zusammenführen von asynchronen Datenpfaden 99
5.17 Aufbau des Arbiters zur Filterung von metastabilen Zuständen innerhalb des Join-Elements. 100
5.18 Aufbau des Synchronisierers von Asynchron zu Synchron. 101
5.19 Flächenbedarf der CPUs, des Speichers, Cluster-Verbindungsstruktur, Netzwerk-Schnittstelle (NI) und der verschiedenen Router-Implementierungen. 103
5.20 Platziert und verdrahtetes Layout des asynchronen und synchronen Cluster-Knotens 104
5.21 Vergleich der Leistungsaufnahme für die verschiedenen Router-Implementierungen 105
5.22 Minimale Latenz und maximaler Durchsatz, der über die verschiedenen I/O-Ports des asynchronen Routers erreicht werden kann. 107
5.23 Leistungsaufnahme des globalen Taktbaumes verschiedener MPSoC-Größen und GALS-Methoden. 108
5.24 Ausschnitt des globalen Taktbaumes für ein 8x8 2D-Mesh CoreVA-MPSoC mit 256 CPUs 109

6.1 Netzwerk-Schnittstelle (NI) im CoreVA-MPSoC (farblich hervorgehoben) 113
6.2 Architektur der Netzwerk-Schnittstelle (NI) im CoreVA-MPSoC 117
6.3 Sequenzdiagramm des NI. 122
6.4 Flächenbedarf des synchr. Router und verschiedener NI-Konfigurationen 125
6.5 Beschleunigung verschiedener MPSoC-Konfigurationen mit unterschiedlicher Anzahl von Empfangskanälen per NI im Vergleich zur Lösung auf einer einzelnen CPU 127
6.6 Verhältnis Beschleunigung und Chipfläche für verschiedene MPSoC-Konfigurationen, mit einer unterschiedlichen Anzahl von NoC-Kommunikationskanälen pro NI 129

7.1 Blockschaltbild des hierarchischen CoreVA-MPSoCs 131
7.2 Verbindungsstrukturen für eng gekoppelte gemeinsame Speicher für 4 CPUs und 8 Speicherbänke [162]. 137

7.3 Der NI ist über vier Ports mit der Verbindungsstruktur des gemeinsamen L1-Speichers verbunden. 138
7.4 Flächenbedarf verschiedener CoreVA-MPSoC-Konfigurationen mit insgesamt 32 CPUs and 1 MB Datenspeicher und 650 MHz. 140
7.5 Energiebedarf für NoC-Transfer Lokal vs. Gem. L1-Speicher. 143
7.6 Minimale Latenz in Anzahl von Taktzyklen, die für NoC-Transfers verschiedener Paketgrößen mit gemeinsamem oder lokalen Speichern benötigt wird. 145
7.7 Beschleunigung für den Durchsatz von StreamIt-Anwendungen für verschiedene MPSoC-Konfigurationen im Vergleich zu einer einzelnen CPU. 146

8.1 Physikalisches Layout eines Cluster-Knotens mit vier CPUs, gemeinsamem L1-Speicher und NoC-Komponenten. Links: Verdrahtetes Layout, Rechts: Platziertes Layout mit markierten NoC-Ports. 158
8.2 Physikalisches Layout eines CoreVA-MPSoC mit 8x8 Cluster-Knoten und insgesamt 256 CoreVA-CPUs. 159
8.3 Physikalisches Layout des Testchips mit zwei CoreVA-CPUs, NI, asynchronem Router und I/O. Links: Platziertes Layout. Rechts: Verdrahtetes Layout. 161
8.4 Schematische Abbildung des CoreVA-MPSoC auf dem FPGA-Cluster, bestehend aus RAPTOR-XPress Boards, bestückt mit DB-V5 Modulen. 164
8.5 Demoaufbau zur Objektdetektion. 165

A.1 SPICE-Simulation des Synchronisierers für ein Bit. Dargestellt ist der Wechsel von logisch 0 zu logisch 1 im ungünstigsten Fall. 208
A.2 SPICE-Simulation des Arbiterverhaltens (Arbiter aus Abbildung 5.17a) bei gleichzeitiger Anfrage an beiden Eingängen. Die Metastabilität wird dabei von den NORs erfolgreich herausgefiltert. 209
A.3 SPICE-Simulation des Arbiterverhaltens (Arbiter aus Abbildung 5.17a) bei nur einer eingehenden Anfrage. 209
A.4 SPICE-Simulation des Arbiterverhaltens (Arbiter aus Abbildung 5.17a) einer zusätzlichen Anfrage bei einer bereits bestehenden Gewährung. 209
A.5 SPICE-Simulation des Arbiterverhaltens (Arbiter aus Abbildung 5.17a) beim Entfernen einer gewährten Anfrage mit bestehender zusätzlichen Anfrage am anderen Eingang. 210
A.6 SPICE-Simulation des Arbiterverhaltens (Arbiter aus Abbildung 5.17a) beim Entfernen einer gewährten Anfrage ohne bestehender Anfrage am anderen Eingang. 210
A.7 Maximale Taktfrequenz beim gemeinsamen L1 Speicher für eine verschiedene Anzahl von Speicherbänken (16 kB pro Bank).[162] 211
A.8 Block-Diagramm zwei verbundener Cluster-Knoten. Die bei der Kommunikation über das NoC involvierten Komponenten sind hervorgehoben 211

A.9 Vergleich verschiedener CoreVA-MPSoC Konfigurationen). [209] . . . 212

Tabellenverzeichnis

2.1 Die Eigenschaften der Topologien im Überblick [219] 15
2.2 Architektureigenschaften von eingebetteten Multiprozessoren 34
2.3 Ressourcenbedarf von eingebetteten Multiprozessoren 34

3.1 Eigenschaften von CPU-Makros der CoreVA-CPU 50

5.1 Konkrete Topologieeigenschaften für ein MPSoC mit 64 beziehungsweise 72 CPU-Cluster 81

6.1 Kanalinformationen für die Netzwerk-Schnittstelle (NI) zum Versenden eines Pakets 118
6.2 Kanalinformationen für die Netzwerk-Schnittstelle (NI) zum Versenden eines Pakets 120
6.3 Softwarekosten und Latenz in Taktzyklen für die Verwendung eines semistatischen und dynamischen NoC-Kanals sowie die des Cluster-Kanals. 123

7.1 Vergleich verschiedener MPSoCs mit Fokus auf der Speicherarchitektur [9] 134
7.2 Leistungsaufnahme und Energieverbrauch für den NoC-Transfer unter Verwendung des gemeinsamen L1-Speicher. 143

8.1 Ressourcenbedarf des CoreVA-MPSoCs bei der Abbildung auf dem Virtex-5 LX100T-FPGA des DB-V5 163

A.1 Legende zur Topologie Übersicht und deren Eigenschaften, verwendete Variablen und Besonderheiten 205
C.1 Allgemeine Eigenschaften der betrachteten Beispielanwendungen . . 206
C.2 Eigenschaften der betrachteten Beispielanwendungen bei einer Abbildung auf das CoreVA-MPSoC 207

Abkürzungsverzeichnis

I^2C	Inter Integrated Circuit
AES	Advanced Encryption Standard
ALU	Arithmetic Logical Unit
AMiRo	Autonomous Mini Robot
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AST	Abstract Syntax Tree
BE	Best-Effort
CCOpt	Clock Concurrent Optimization
CMOS	Complementary Metal-Oxide-Semiconductor
CoreVA	Configurable Ressource Efficient VLIW Architecture
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CTS	Clock Tree Synthesis
DES	Data Encryption Standard
DLP	Data Level Parallelism
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DVFS	Dynamic Voltage and Frequency Scaling
ECC	Elliptic Curve Cryptography
EMI	Electromagnetic Interference
FD-SOI	Fully-Depleted Silicon-on-Insulator
FFT	Fast Furier Transformation
FIFO	First-In First-Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GALS	Global Asynchron Lokal Synchron

GDSII	Graphic Database System 2
GPU	Graphics Processing Unit
GS	Guaranteed-Service
HDL	Hardware Description Language
I/O	Input/Output
ILP	Instruction Level Parallelism
IP	Intellectual Property
IPC	Inter Process Communication
ISS	Instruction Set Simulator
LD/ST	Load/Store
LLVM	Modulare Compilerarchitektur, früher Low Level Virtual Machine
LLVM IR	LLVM Intermediate Representation
LTE	Long Term Evolution
LVDS	Low Voltage Differential Signaling
MAC	Multiply Accumulate
MGT	Multi Giga-bit Tranceiver
MIT	Massachusetts Institute of Technology
MMIO	Memory Mapped I/O
MMU	Memory Management Unit
MoT	Mesh of Trees
MPI	Message Passing Interface
MPSoC	Multiprocessor System On a Chip
Mutex	Mutual Exclusion
NI	Network Interface
NoC	Network on a Chip
NUMA	Non-Uniform Memory Access
OpenCL	Open Computing Language
P&R	Place and Route
PC	Program Counter
PCI	Peripheral Component Interconnect
PD-SOI	Partially-Depleted Silicon-on-Insulator
POSIX	Portable Operating System Interface
QUT	Queensland University of Technology

RTL	Register Transfer Level
S&F	Store-And-Forward
SBE	Simulation Based Estimation
SDR	Software-Defined Radio
SIMD	Single Instruction Multiple Data
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SVM	Support-Vector Machine
TCMS	Tightly Coupled Mesochronous Synchronizer
TDM	Time Division Multiplexing
TLP	Thread Level Parallelism
UART	Universal Asynchronous Receiver Transmitter
ULP	Ultra Low Power
UMA	Uniform Memory Access
VCT	Virtual-Cut-Trough
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
XPS	Xilinx Platform Studio

Referenzen

[1] A. Abdiansah und R. Wardoyo. „Time Complexity Analysis of Support Vector Machines (SVM) in LibSVM“. In: *International Journal of Computer Applications* 128.3 (2015), S. 975–8887. DOI: 10.5120/ijca2015906480.

[2] Adapteva. „E64G401 Epiphany 64-Core Microprocessor Datasheet“. In: (2014), S. 1–57.

[3] Adapteva Inc. *Adapteva*. URL: http://www.adapteva.com/ (besucht am 04. 12. 2016).

[4] A. Agarwal, C. Iskander und R. Shankar. „Survey of network on chip (noc) architectures & contributions“. In: *Journal of engineering, Computing and Architecture* 3.1 (2009), S. 21–27.

[5] K. G. Andrei Radulescu. „Communication Services for Networks on Chip“. In: (2002). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.6786.

[6] *Arbeitsgruppe Kognitronik und Sensorik, CITEC, Universität Bielefeld*. URL: http://www.ks.cit-ec.uni-bielefeld.de.

[7] Arteris. *Arteris - FlexNoC*. 2017. URL: http://www.arteris.com/.

[8] O. Astrachan, Owen, Astrachan und Owen. „Bubble sort: an archaeological algorithmic analysis“. In: *Proceedings of the 34th SIGCSE technical symposium on Computer science education - SIGCSE '03*. Bd. 35. 1. ACM Press, 2003, S. 1. DOI: 10.1145/611892.611918.

[9] J. Ax, G. Sievers, J. Daberkow, M. Flasskamp, M. Vohrmann, T. Jungeblut, W. Kelly, M. Porrmann und U. Ruckert. „CoreVA-MPSoC: A Many-Core Architecture with Tightly Coupled Shared and Local Data Memories“. In: *IEEE Transactions on Parallel and Distributed Systems* (2017). DOI: 10.1109/TPDS.2017.2785799.

[10] J.-L. Baer. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. 1. Aufl. Cambridge University Press, 2009. ISBN: 0521769922.

[11] J. Balfour, W. J. Dally, D. Black-Schaffer und V. Parikh. „An Energy-Efficient Processor Architecture for Embedded Systems“. In: *IEEE Computer Architecture Letters* 7.1 (2008), S. 29–32. DOI: 10.1109/L-CA.2008.1.

[12] P. Balog. *Metastabilität (the anomalous behavior of synchronizer circuits)*. 1996. ISBN: 3851330102.

[13] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan und P. Marwedel. „Scratchpad memory: design alternative for cache on-chip memory in embedded systems“. In: *Proceedings of the tenth international symposium on Hardware/software codesign - CODES '02*. ACM Press, 2002, S. 73–78. DOI: 10.1145/774789.774805.

[14] Bei Yu, Sheqin Dong, Song Chen und S. Goto. „Floorplanning and topology generation for application-specific Network-on-Chip“. In: *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, Jan. 2010, S. 535–540. DOI: 10.1109/ASPDAC.2010.5419825.

[15] L. Benini und G. De Micheli. „Networks on chips: A new SoC paradigm“. In: *computer* 35.1 (2002), S. 70–78.

[16] L. Benini, E. Flamand, D. Fuin und D. Melpignano. „P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator“. In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, S. 983–987. DOI: 10.1109/DATE.2012.6176639.

[17] C. H. van Berkel. „Multi-core for mobile phones“. In: *Design, Automation and Test in Europe*. European Design und Automation Association, Apr. 2009, S. 1260–1265. ISBN: 978-3-9810801-5-5.

[18] M. Berry. „Public International Benchmarks for Parallel Computers: PARKBENCH Committee: Report-1“. In: *Sci. Program.* 3.2 (1994), S. 100–146. ISSN: 1058-9244.

[19] T. Bjerregaard und S. Mahadevan. „A survey of research and practices of Network-on-chip“. In: *ACM Computing Surveys* 38.1 (Juni 2006), 1–es. DOI: 10.1145/1132952.1132953.

[20] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo und B. Baas. *A 5.8 pJ/Op 115 Billion Ops/sec, to 1.78 Trillion Ops/sec 32 nm 1000 -Processor Array*. Techn. Ber. Table 1. University of California, 2016, S. 3–5. URL: http://vcl.ece.ucdavis.edu/pubs/2016.06.vlsi.symp.kiloCore/2016.vlsi.symp.kiloCore.pdf.

[21] S. Borkar. „Thousand core chips“. In: *Proceedings of the 44th annual conference on Design automation - DAC '07*. ACM Press, 2007, S. 746. DOI: 10.1145/1278480.1278667.

[22] B. Bowhill, B. Stackhouse, N. Nassif, Z. Yang, A. Raghavan, O. Mendoza und C. Morganti. „The Xeon® Processor E5-2600 v3: a 22 nm 18-Core Product Family“. In: *IEEE Journal of Solid-State Circuits* 51.1 (Jan. 2016), S. 92–104. DOI: 10.1109/JSSC.2015.2472598.

[23] C. Brunelli, R. Airoldi und J. Nurmi. „Implementation and benchmarking of FFT algorithms on multicore platforms“. In: *2010 International Symposium on System on Chip*. IEEE, Sep. 2010, S. 59–62. DOI: 10.1109/ISSOC.2010.5625561.

[24] P. Burgio, A. Marongiu, D. Heller, C. Chavet, P. Coussy und L. Benini. „OpenMP-based Synergistic Parallelization and HW Acceleration for On-Chip Shared-Memory Clusters“. In: *2012 15th Euromicro Conference on Digital System Design*. IEEE, Sep. 2012, S. 751–758. DOI: 10.1109/DSD.2012.97.

[25] Cadence. *Genus Synthesis Solution*. URL: https://www.cadence.com/content/cadence-www/global/en%7B%5C_%7DUS/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html?CMP=Overview%7B%5C_%7DFFlow%7B%5C_%7DDigital (besucht am 09. 06. 2017).

[26] Cadence. *Innovus Imlementation System*. URL: https://www.cadence.com/content/cadence-www/global/en%7B%5C_%7DUS/home/tools/digital-design-and-signoff/hierarchical-design-and-floorplanning/innovus-implementation-system.html?CMP=Overview%7B%5C_%7DFFlow%7B%5C_%7DDigital (besucht am 09. 06. 2017).

[27] Cadence. *Voltus IC PowerIntegrity Solution*. URL: https://www.cadence.com/content/cadence-www/global/en%7B%5C_%7DUS/home/tools/digital-design-and-signoff/silicon-signoff/voltus-ic-power-integrity-solution.html?CMP=Overview%7B%5C_%7DFFlow%7B%5C_%7DDigital (besucht am 09. 06. 2017).

[28] J. D. Carpinelli. *Computer systems organization and Architecture*. Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201612534.

[29] S. D. Chawade, M. A. Gaikwad und R. M. Patrikar. „REVIEW OF XY ROUTING ALGORITHM FOR NETWORK-ON-CHIP ARCHITECTURE“. In: *International Journal of Internet Computing* 1.4 (2012), S. 48–52.

[30] L. Chen, M. Marek-Sadowska und F. Brewer. „Buffer delay change in the presence of power and ground noise“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 11.3 (Juni 2003), S. 461–473. DOI: 10.1109/TVLSI.2003.812310.

[31] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe und H. Angepat. „FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators“. In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007, S. 249–261. DOI: 10.1109/MICRO.2007.36.

[32] P. Christ. „Detektion und Analyse physiologischer und biokinematischer Parameter mit Körpersensoren“. Diss. 2016, S. 260.

[33] P. Christ, F. Werner, U. Rückert und J. Mielebacher. „An approach for determining linear velocities of athletes from acceleration measurements using a neural network“. In: *Proc. of the 6th IASTED Int. Conf. on Biomechanics*. Hrsg. von B. Morrison und M. H. Hamza. ACTA Press, 2011, S. 105–112. DOI: 10.2316/P.2011.751-009.

[34] *Cisco Systems, Inc.: The Internet of Things*. URL: http://share.cisco.com/internet-of-things.html.

[35] F. Clermidy, S. Miermont und P. Vivet. „Dynamic Voltage and Frequency Scaling Architecture for Units Integration within a GALS NoC“. In: *Second ACM/IEEE International Symposium on Networks-on-Chip*. 2008, S. 129–138. DOI: 10.1109/NOCS.2008.26.

[36] M. Coppola, M. D. Grammatikakis, R. Locatelli, G. Maruccia und L. Pieralisi. „Design of Cost-Efficient Interconnect Processing Units: Spidergon STNoC“. In: (Sep. 2008).

[37] T. Cormen und R. Leiserson, Charles Eric Rivest. *Introduction to algorithms*. 2009, S. 1028. ISBN: 9780262033848.

[38] D. E. Culler, J. P. Singh und A. Gupta. *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999. ISBN: 1558603433.

[39] P. Cunningham, M. Swinn und S. Wilcox. „Clock Concurrent Optimization: Rethinking Timing Optimization to Target Clocks and Logic at the Same Time“. In: (2010), S. 1–20. URL: https://www10.edacafe.com/link/Clock-Concurrent-Optimization-Timing-Clocks-Logic-Same-Time/34504/link%7B%5C_%7Ddownload/No/Clock%7B%5C_%7DConcurrent%7B%5C_%7DOpt%7B%5C_%7DWP%7B%5C_%7DV2.pdf.

[40] W. J. Dally und B. Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann Publishers, 2004, S. 550. ISBN: 0122007514.

[41] R. Das, S. Eachempati, A. K. Mishra, V. Narayanan und C. R. Das. „Design and evaluation of a hierarchical on-chip interconnect for next-generation CMPs“. In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 2009, S. 175–186. DOI: 10.1109/HPCA.2009.4798252.

[42] M. Dehyadgari, M. Nickray, A. Afzali-kusha und Z. Navabi. „Evaluation of Pseudo Adaptive XY Routing Using an Object Oriented Model for NOC“. In: *2005 International Conference on Microelectronics*. IEEE, 2005, S. 204–208. DOI: 10.1109/ICM.2005.1590068.

[43] M. Demler. „Lithography Woes Raise Chip Costs“. In: *Microprocessor Report* August (2015). ISSN: 0899-9341.

[44] *DesignWare DW_fifo_s1_sf Datasheet*. Techn. Ber. Synopsys, Inc., 2015.

[45] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss und T. Strudel. „A clustered manycore processor architecture for embedded and accelerated applications“. In: *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2013, S. 1–6. DOI: 10.1109/HPEC.2013.6670342.

[46] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert und T. Strudel. „A Distributed Run-Time Environment for the Kalray MPPA®-256 Integrated Manycore Processor“. In: *Procedia Computer Science* 18 (2013), S. 1654–1663. DOI: 10.1016/j.procs.2013.05.333.

[47] A. Y. Dogan, J. Constantin, M. Ruggiero, A. Burg und D. Atienza. „Multi-core architecture design for ultra-low-power wearable health monitoring systems“. In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, S. 988–993. DOI: 10.1109/DATE.2012.6176640.

[48] EEMBC. *EMBEDDED MICROPROCESSOR BENCHMARK CONSORTIUM*. 2018. URL: http://www.eembc.org (besucht am 08. 03. 2018).

[49] D. Fick, R. G. Dreslinski, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, M. Wieckowski, G. Chen u. a. „Centip3De: A Cluster-Based NTC Architecture With 64 ARM Cortex-M3 Cores in 3D Stacked 130 nm CMOS“. In: *IEEE Journal of Solid-State Circuits* 48.1 (2013), S. 104–117. DOI: 10.1109/JSSC.2012.2222814.

[50] J. A. Fisher. „Very Long Instruction Word architectures and the ELI-512“. In: *Proceedings of the 10th annual international symposium on Computer architecture - ISCA '83*. Bd. 11. 3. ACM Press, 1983, S. 140–150. DOI: 10.1145/800046.801649.

[51] J. Flich, T. Skeie, A. Mejía, O. Lysne, P. López, A. Robles, J. Duato, M. Koibuchi, T. Rokicki und J. C. Sancho. „A survey and evaluation of topology-agnostic deterministic routing algorithms“. In: *IEEE Transactions on Parallel and Distributed Systems* 23.3 (2012), S. 405–425. DOI: 10.1109/TPDS.2011.190.

[52] S. Furber. *Silistix Limited*. URL: http://www.cs.manchester.ac.uk/industry/spin-offs/silistixlimited/ (besucht am 17. 06. 2018).

[53] M. Gautschi, D. Rossi und L. Benini. „Customizing an open source processor to fit in an ultra-low power cluster with a shared L1 memory“. In: *Proceedings of the 24th edition of the great lakes symposium on VLSI - GLSVLSI '14*. ACM Press, 2014, S. 87–88. DOI: 10.1145/2591513.2591569.

[54] D. Gebhardt, J. You und K. S. Stevens. „Comparing Energy and Latency of Asynchronous and Synchronous NoCs for Embedded SoCs“. In: *2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip*. IEEE, 2010, S. 115–122. DOI: 10.1109/NOCS.2010.21.

[55] M. Gibiluka. „Design and implementation of an asynchronous noc router using a transition-signaling bundled-data protocol“. Diss. PONTIF{Í}CIA UNIVERSIDADE CAT{Ó}LICA DO RIO GRANDE DO SUL, 2013.

[56] K. Goossens und A. Hansson. „The aethereal network on chip after ten years“. In: *Proceedings of the 47th Design Automation Conference on - DAC '10*. ACM Press, Juni 2010, S. 306. DOI: 10.1145/1837274.1837353.

[57] M. Gordon. „Compiler techniques for scalable performance of stream programs on multicore architectures“. In: (2010). URL: http://dl.acm.org/citation.cfm?id=2048965.

[58] M. I. Gordon, W. Thies und S. Amarasinghe. „Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs“. In: *ACM SIGPLAN Notices*. Bd. 41. 11. ACM, 2006, S. 151. DOI: 10.1145/1168918.1168877.

[59] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge und R. Brown. „MiBench: A free, commercially representative embedded benchmark suite“. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. IEEE, 2001, S. 3–14. DOI: 10.1109/WWC.2001.990739.

[60] M. Guthaus, J. Ringenberg, T. M. Austin, T. Mudge und R. B. Brown. *MiBench*. 2001. URL: http://vhosts.eecs.umich.edu/mibench/ (besucht am 10. 03. 2018).

[61] L. Gwennap. „How Cortex-A15 Measures Up“. In: *Microprocessor Report* (2013). ISSN: 0899-9341.

[62] L. Gwennap. „Samueli Sees New Focus on Design Skill“. In: *Microprocessor Report* (2015).

[63] T. R. Halfhill. „Opportunity NoCs, NetSpeed Answers“. In: *Microprocessor Report* December (2014). ISSN: 0899-9341.

[64] C. Hamacher, Z. Vranesic, S. Zaky und N. Manjikian. *Computer Organization and Embedded Systems*. 6. Aufl. Bd. 3. McGraw-Hill, 2011. ISBN: 0073380652.

[65] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis und M. Horowitz. „Understanding sources of inefficiency in general-purpose chips“. In: *ISCA '10 Proceedings of the 37th annual international symposium on Computer architecture* 38.3 (2010), S. 37. DOI: 10.1145/1816038.1815968.

[66] A. Hansson, M. Subburaman und K. Goossens. „Aelite: A flit-synchronous Network on Chip with composable and predictable services“. In: *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, Apr. 2009, S. 250–255. DOI: 10.1109/DATE.2009.5090666.

[67] A. Hansson und K. Goossens. „CoMPSoC: A template for composable and predictable multi-processor system on chips“. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 14.1 (2009), S. 1–24. DOI: 10.1145/1455229.1455231..

[68] R. Heckmann, M. Langenbach, S. Thesing und R. Wilhelm. „The influence of processor architecture on the design and the results of WCET tools“. In: *Proceedings of the IEEE* 91.7 (2003), S. 1038–1054. DOI: 10.1109/JPROC.2003.814618.

[69] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Oberg, P. Ellervee und D. Lundqvist. „Lowering power consumption in clock by using globally asynchronous locally synchronous design style“. In: *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*. IEEE, S. 873–878. DOI: 10.1109/DAC.1999.782202.

[70] J. L. Hennessy und D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 4. Aufl. Morgan Kaufmann, 2006. ISBN: 0123704901.

[71] S. Hesham, J. Rettkowski, D. Goehringer und M. A. Abd El Ghany. „Survey on Real-Time Networks-on-Chip“. In: *IEEE Transactions on Parallel and Distributed Systems* (2016), S. 1–1. DOI: 10.1109/TPDS.2016.2623619.

[72] L. Howes und A. Munshi. *The OpenCL Specification, Version 2.2*. Spezifikation. Khronos OpenCL Working Group, 2018. URL: http://www.khronos.org/registry/cl/.

[73] B. Hübener. „Analyse verschiedener Architekturvarianten des CoreVA-VLIW-Prozessors“. Diss. 2016, S. 190.

[74] B. Hübener, G. Sievers, T. Jungeblut, M. Porrmann und U. Rückert. „CoreVA: A Configurable Resource-Efficient VLIW Processor Architecture“. In: *2014 12th IEEE International Conference on Embedded and Ubiquitous Computing*. IEEE, Aug. 2014, S. 9–16. DOI: 10.1109/EUC.2014.11.

[75] *International Technology Roadmap for Semiconductors (ITRS 2.0)*. URL: http://www.itrs2.net/.

[76] A. Ivanov und G. D. Micheli. „Guest Editors' Introduction: The Network-on-Chip Paradigm in Practice and Research“. In: *IEEE Design Test of Computers* 22.5 (Sep. 2005), S. 399–403. DOI: 10.1109/MDT.2005.111.

[77] A. Jantsch und H. Tenhunen, Hrsg. *Networks on Chip*. Kluwer Academic Publishers, 2003. ISBN: 1-4020-7392-5.

[78] T. Jungeblut. „Entwurfsraumexploration ressourceneffizienter VLIW-Prozessoren“. Diss. Bielefeld University, 2011, S. 280.

[79] T. Jungeblut, R. Dreesen, M. Porrmann, U. Rückert und U. Hachmann. „Design Space Exploration for Resource Efficient VLIW-Processors“. In: *University Booth of the Design, Automation and Test in Europe (DATE) conference*. 2008.

[80] T. Jungeblut, R. Dreesen, M. Porrmann, M. Thies, U. Rückert und U. Kastens. *"Netz der Zukunft" - MxMobile - Multi-Standard Mobile Plattform - Schlussbericht*. 2009.

[81] T. Jungeblut, B. Hübener, M. Porrmann und U. Rückert. „A systematic approach for optimized bypass configurations for application-specific embedded processors". In: *ACM Transactions on Embedded Computing Systems* 13.2 (Sep. 2013). DOI: 10.1145/2514641.2514645.

[82] T. Jungeblut, S. Lütkemeier, G. Sievers, M. Porrmann und U. Rückert. „A modular design flow for very large design space explorations". In: *Proceedings of the CDNLive! EMEA*. 2010. URL: http://www.cadence.com/cdnlive/eu/2010/pages/proceedings.aspx.

[83] M. R. Kakoee, I. Loi und L. Benini. „A resilient architecture for low latency communication in shared-L1 processor clusters". In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, S. 887–892. DOI: 10.1109/DATE.2012.6176623.

[84] M. R. Kakoee, V. Petrovic und L. Benini. „A multi-banked shared-l1 cache architecture for tightly coupled processor clusters". In: *2012 International Symposium on System on Chip (SoC)*. IEEE, 2012, S. 1–5. DOI: 10.1109/ISSoC.2012.6376362.

[85] S. Kanazawa, T. Fujisawa, K. Takahata, T. Ito, Y. Ueda, W. Kobayashi, H. Ishii und H. Sanjoh. „Flip-Chip Interconnection Lumped-Electrode EADFB Laser for 100-Gb/s Transmitter". In: *IEEE Photonics Technology Letters* 27.16 (Aug. 2015), S. 1699–1701. DOI: 10.1109/LPT.2015.2438076.

[86] D. Kanter. „KNIGHTS LANDING RESHAPES HPC". In: *Microprocessor Report* September 2015 (2015).

[87] D. Kanter. „Multicore Drives Embedded Growth". In: *Microprocessor Report* December (2014). ISSN: 0899-9341.

[88] D. Kanter. „Nvidia Hits HPC First With Pascal". In: *Microprocessor Report* (2016). ISSN: 0899-9341.

[89] D. Kanter und L. Gwennap. „Kalray Clusters Calculate Quickly". In: *Microprocessor Report* (2015). ISSN: 0899-9341.

[90] E. Kasapaki. „An Asynchronous Time-Division-Multiplexed Network-on-Chip for Real-Time Systems". Diss. PhD thesis, Technical University of Denmark (DTU), 2015.

[91] E. Kasapaki und J. Sparso. „The Argo NOC: Combining TDM and GALS". English. In: *2015 European Conference on Circuit Theory and Design (ECCTD)*. IEEE, Aug. 2015, S. 1–4. DOI: 10.1109/ECCTD.2015.7300101.

[92] M. Katevenis, V. Papaefstathiou, S. Kavadias, D. Pnevmatikatos, F. Silla und D. Nikolopoulos. „Explicit Communication and Synchronization in SARC“. In: *IEEE Micro* 30.5 (Sep. 2010), S. 30–41. DOI: 10.1109/MM.2010.77.

[93] J. Kim, W. J. Dally und D. Abts. „Flattened Butterfly: A Cost-efficient Topology for High-radix Networks“. In: *ISCA '7 Proceedings of the 34th annual international symposium on Computer architecture*. 2007, S. 126–137. DOI: 10.1145/1250662.1250679.

[94] H. Klar und T. Noll. *Integrierte Digitale Schaltungen*. 3. Aufl. Springer, 2015. ISBN: 978-3-540-40600-6.

[95] C. Klarhorst. „Partitionierung von Streaming-Anwendungen auf dem CoreVA-MPSoC“. Diss. 2013.

[96] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2011. ISBN: 144198237X.

[97] C. Lattner und V. Adve. „LLVM: A compilation framework for lifelong program analysis & transformation“. In: *International Symposium on Code Generation and Optimization, CGO*. IEEE, 2004, S. 75–86. DOI: 10.1109/CGO.2004.1281665.

[98] J.-J. Lecler und G. Baillieu. „Application driven network-on-chip architecture exploration & refinement for a complex SoC“. In: *Design Automation for Embedded Systems* 15.2 (Apr. 2011), S. 133–158. DOI: 10.1007/s10617-011-9075-5.

[99] H. G. Lee, N. Chang, U. Y. Ogras und R. Marculescu. „On-chip communication architecture exploration“. In: *ACM Transactions on Design Automation of Electronic Systems* 12.3 (Aug. 2007), 23–es. DOI: 10.1145/1255456.1255460.

[100] K. Lee, S. J. Lee und H. J. Yoo. „Low-power network-on-chip for high-performance SoC design“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14.2 (2006), S. 148–160. DOI: 10.1109/TVLSI.2005.863753.

[101] L. F. Leung und C. Y. Tsui. „Energy-aware synthesis of networks-on-chip implemented with voltage islands“. In: *Proceedings - Design Automation Conference* (2007), S. 128–131. DOI: 10.1109/DAC.2007.375138.

[102] S. Liu, A. Jantsch und Z. Lu. „Analysis and Evaluation of Circuit Switched NoC and Packet Switched NoC“. In: *2013 Euromicro Conference on Digital System Design*. IEEE, Sep. 2013, S. 21–28. DOI: 10.1109/DSD.2013.13.

[103] LLVM Foundation. *The LLVM Compiler Infrastructure Project*. URL: http://www.llvm.org/ (besucht am 09. 06. 2017).

[104] I. Loi und L. Benini. „A multi banked - Multi ported - Non blocking shared L2 cache for MPSoC platforms“. In: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 2014, S. 1–6.

[105] B. C. Lopes und R. Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing Ltd, 2014. ISBN: 1782166939.

[106] D. Ludovici, A. Strano, D. Bertozzi, L. Benini und G. N. Gaydadjiev. „Comparing tightly and loosely coupled mesochronous synchronizers in a noc switch architecture“. In: *Proceedings - 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip, NoCS 2009* (2009), S. 244–249. DOI: 10.1109/NOCS.2009.5071473.

[107] S. Lütkemeier. „Ressourceneffiziente Digitalschaltungen für den Subschwellbetrieb“. Diss. Universität Paderborn, 2013.

[108] S. Lütkemeier, T. Jungeblut, H. K. O. Berge, S. Aunet, M. Porrmann und U. Rückert. „A 65 nm 32 b Subthreshold Processor With 9T Multi-Vt SRAM and Adaptive Supply Voltage Control“. In: *IEEE Journal of Solid-State Circuits* 48.1 (2013), S. 8–19. DOI: http://dx.doi.org/10.1109/JSSC.2012.2220671.

[109] S. Lütkemeier, T. Jungeblut, M. Porrmann und U. Rückert. „A 200mV 32b Subthreshold Processor with Adaptive Supply Voltage Control“. In: *2012 IEEE International Solid-State Circuits Conference*. Bd. 57. 9. 2012, S. 484–485. DOI: http://dx.doi.org/10.1109/ISSCC.2012.6177101.

[110] T. Marescaux, E. Brockmeyer und H. Corporaal. „The Impact of Higher Communication Layers on NoC Supported MP-SoCs“. In: *First International Symposium on Networks-on-Chip (NOCS'07)* (2007), S. 107–116. DOI: 10.1109/NOCS.2007.41.

[111] A. Martin und M. Nystrom. „Asynchronous Techniques for System-on-Chip Design“. In: *Proceedings of the IEEE* 94.6 (Juni 2006), S. 1089–1120. DOI: 10.1109/JPROC.2006.875789.

[112] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy und D. Dutoit. „Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications“. In: *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*. 2012, S. 1137–1142. DOI: 10.1145/2228488.2228568.

[113] B. Mesgarzadeh, C. Svensson und A. Alvandpour. „A new mesochronous clocking scheme for synchronization in SoC“. In: *2004 IEEE International Symposium on Circuits and Systems (IEEE Cat. No.04CH37512)*. IEEE, S. II–605–8. DOI: 10.1109/ISCAS.2004.1329344.

[114] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep und A. Agarwal. „Graphite: A distributed parallel simulator for multicores“. In: *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, Jan. 2010, S. 1–12. DOI: 10.1109/HPCA.2010.5416635.

[115] G. Miorandi, A. Celin, M. Favalli und D. Bertozzi. „A built-in self-testing framework for asynchronous bundled-data NoC switches resilient to delay variations“. In: *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*. IEEE, Sep. 2016, S. 1–8. DOI: 10.1109/NOCS.2016.7579332.

[116] I. Miro-Panades, F. Clermidy, P. Vivet und A. Greiner. *Physical Implementation of the DSPIN Network-on-Chip in the FAUST Architecture*. IEEE Computer Society, 2008, S. 220. ISBN: 9780769530987.

[117] I. Miro-Panades, A. Greiner und A. Sheibanyrad. „A Low Cost Network-on-Chip with Guaranteed Service Well Suited to the GALS Approach“. In: *2006 1st International Conference on Nano-Networks and Workshops*. IEEE, Sep. 2006, S. 1–5. DOI: 10.1109/NANONET.2006.346219.

[118] G. E. Moore. „Cramming more components onto integrated circuits“. In: *Electronics* 38 (1965), S. 114–117.

[119] MPI Forum. *Meassage Passing Interface*. 2012. URL: http://www.mcs.anl.gov/research/projects/mpi/ (besucht am 17. 03. 2018).

[120] F. Mu und C. Svensson. „Self-tested self-synchronization circuit for mesochronous clocking“. In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 48.2 (2001), S. 129–140. DOI: 10.1109/82.917781.

[121] NaNoC Project. *Methodologies for the design of physical-level-aware NoC topologies*. 2012. URL: http://cordis.europa.eu/docs/projects/cnect/2/248972/080/deliverables/001-D35revised.pdf.

[122] D. Nassimi und S. Sahni. „Bitonic Sort on a Mesh-Connected Parallel Computer“. In: *IEEE Transactions on Computers* C-28.1 (1979), S. 2–7. DOI: 10.1109/TC.1979.1675216.

[123] J. Navaridas, M. Luján, J. Miguel-Alonso, L. A. Plana und S. Furber. „Understanding the interconnection network of SpiNNaker“. In: *Proceedings of the 23rd international conference on Conference on Supercomputing - ICS '09*. ACM Press, 2009, S. 286. DOI: 10.1145/1542275.1542317.

[124] J. Navaridas, M. Luján, L. A. Plana, S. Temple und S. B. Furber. „On Generating Multicast Routes for SpiNNaker“. In: *CF '14 Proceedings of the 11th ACM Conference on Computing Frontiers Article No. 2*. 2014. DOI: 10.1145/2597917.2597938.

[125] Netspeed. *Netspeed - Orion, Gemini, Pegasus*. 2017. URL: www.netspeedsystems.com.

[126] L. Ni und P. McKinley. „A survey of wormhole routing techniques in direct networks“. In: *Computer* 26.2 (Feb. 1993), S. 62–76. DOI: 10.1109/2.191995.

[127] J. Niemann. „Ressourceneffiziente Schaltungstechnik eingebetteter Parallelrechner: GigaNetIC.“ Diss. 2008. ISBN: 978-3-939350-66-8.

[128] J. Niemann, C. Puttmann, M. Porrmann und U. Rückert. „Resource efficiency of the GigaNetIC chip multiprocessor architecture“. In: *Journal of Systems Architecture* 53.5-6 (Mai 2007), S. 285–299. DOI: 10.1016/j.sysarc.2006.10.007.

[129] B. Noethen, O. Arnold, E. P. Adeva, T. Seifert, E. Fischer, S. Kunze, E. Matus, G. Fettweis, H. Eisenreich, G. Ellguth, S. Hartmann, S. Hoppner u. a. „A 105GOPS 36mm2 heterogeneous SDR MPSoC with energy-aware dynamic scheduling and iterative detection-decoding for 4G in 65nm CMOS“. In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, S. 188–189. DOI: 10.1109/ISSCC.2014.6757394.

[130] U. Y. Ogras, R. Marculescu, P. Choudhary und D. Marculescu. „Voltage-frequency island partitioning for GALS-based networks-on-chip“. In: *Proceedings - Design Automation Conference* (2007), S. 110–115. DOI: 10.1109/DAC.2007.375135.

[131] T. Ohkawa, K. Ootsu, T. Yokota, K. Kikuchi und M. Aoyagi. „Designing Efficient Parallel Processing in 3D Standard-Chip Stacking System with Standard Bus“. In: *2017 IEEE 11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, Sep. 2017, S. 128–135. DOI: 10.1109/MCSoC.2017.27.

[132] C. H. M. Oliveira, M. T. Moreira, R. A. Guazzelli und N. L. V. Calazans. „ASCEnD-FreePDK45: An open source standard cell library for asynchronous design“. In: *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, Dez. 2016, S. 652–655. DOI: 10.1109/ICECS.2016.7841286.

[133] C. H. M. Oliveira, M. T. Moreira, R. A. Guazzelli und N. L. V. Calazans. „ASCEnD-FreePDK45: An open source standard cell library for asynchronous design“. In: *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, Dez. 2016, S. 652–655. DOI: 10.1109/ICECS.2016.7841286.

[134] A. Olofsson. *Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip*. Techn. Ber. Adapteva Inc., 2016.

[135] N. Onizawa, A. Matsumoto, T. Funazaki und T. Hanyu. „High-Throughput Compact Delay-Insensitive Asynchronous NoC Router“. In: *IEEE Transactions on Computers* 63.3 (März 2014), S. 637–649. DOI: 10.1109/TC.2013.81.

[136] E. Painkras, L. a. Plana, J. Garside, S. Temple, S. Davidson, J. Pepper, D. Clark, C. Patterson und S. Furber. „SpiNNaker: A multi-core System-on-Chip for massively-parallel neural net simulation“. In: *Proceedings of the IEEE 2012 Custom Integrated Circuits Conference*. Bd. 4. Ieee, Sep. 2012, S. 1–4. DOI: 10.1109/CICC.2012.6330636.

[137] E. Painkras, L. A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown und S. B. Furber. „SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation“. In: *IEEE Journal of Solid-State Circuits* 48.8 (Aug. 2013), S. 1943–1953. DOI: 10.1109/JSSC.2013.2259038.

[138] P. Pande, C. Grecu, M. Jones, A. Ivanov und R. Saleh. „Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures“. In: *IEEE Transactions on Computers* 54.8 (Aug. 2005), S. 1025–1040. DOI: 10.1109/TC.2005.134.

[139] S. Pasricha und N. Dutt. *On-chip communication architectures: system on chip interconnect*. Morgan Kaufmann, 2008. ISBN: 978-0123738929.

[140] D. A. Patterson und J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 4. Aufl. Morgan Kaufmann; 2011. ISBN: 0123747503.

[141] W. Plummer. „Asynchronous Arbiters“. In: *IEEE Transactions on Computers* C-21.1 (Jan. 1972), S. 37–42. DOI: 10.1109/T-C.1972.223429.

[142] M. Porrmann, J. Hagemeyer, J. Romoth, M. Strugholtz und C. Pohl. „RAPTOR–A Scalable Platform for Rapid Prototyping and FPGA-based Cluster Computing“. In: *Parallel Computing: From Multicores and GPU's to Petascale*. 2009, S. 592–599. ISBN: 978-1-60750-529-7.

[143] A. Pullini, F. Angiolini, S. Murali, D. Atienza, G. De Micheli und L. Benini. „Bringing NoCs to 65 nm“. In: *IEEE Micro* 27.5 (Sep. 2007), S. 75–85. DOI: 10.1109/MM.2007.79.

[144] C. Puttmann. „Ressourceneffiziente Hardware-Software-Kombinationen für Kryptographie mit elliptischen Kurven“. Diss. Universität Bielefeld, 2014.

[145] C. Puttmann, J. Niemann, M. Porrmann und U. Rückert. „GigaNoC - A Hierarchical Network-on-Chip for Scalable Chip-Multiprocessors“. In: *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*. IEEE, Aug. 2007, S. 495–502. DOI: 10.1109/DSD.2007.4341514.

[146] A. Radulescu, J. Dielissen, S. Pestana, O. Gangwal, E. Rijpkema, P. Wielage und K. Goossens. „An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.1 (Jan. 2005), S. 4–17. DOI: 10.1109/TCAD.2004.839493.

[147] A. Rahimi, I. Loi, M. R. Kakoee und L. Benini. „A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters“. In: *2011 Design, Automation & Test in Europe*. IEEE, 2011, S. 1–6. DOI: 10.1109/DATE.2011.5763085.

[148] D. Rahmati, S. Murali, L. Benini, F. Angiolini, G. De Micheli und H. Sarbazi-Azad. „Computing Accurate Performance Bounds for Best Effort Networks-on-Chip“. In: *IEEE Transactions on Computers* 62.3 (März 2013), S. 452–467. DOI: 10.1109/TC.2011.240.

[149] E. Rijpkema, K. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage und E. Waterlander. „Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip“. In: *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE Comput. Soc, 2003, S. 350–355. DOI: 10.1109/DATE.2003.1253633.

[150] J. Romoth. „FPGA-Cluster – Anwendungsgebiete und Kommunikationsstrukturen“. Diss. 2018.

[151] D. Rossi u. a. „A 60 GOPS/W, -1.8V to 0.9V Body Bias ULP Cluster in 28nm UTBB FD-SOI Technology“. In: *Solid-State Electronics* 117 (2016), S. 170–184. DOI: 10.1016/j.sse.2015.11.015.

[152] S. K. Sadasivam, B. W. Thompto, R. Kalla und W. J. Starke. „IBM Power9 Processor Architecture“. In: *IEEE Micro* 37.2 (März 2017), S. 40–51. DOI: 10.1109/MM.2017.40.

[153] K. Sakuma, P. S. Andry, C. K. Tsang, S. L. Wright, B. Dang, C. S. Patel, B. C. Webb, J. Maria, E. J. Sprogis, S. K. Kang, R. J. Polastre, R. R. Horton u. a. „3D chip-stacking technology with through-silicon vias and low-volume lead-free interconnections“. In: *IBM Journal of Research and Development* 52.6 (Nov. 2008), S. 611–622. DOI: 10.1147/JRD.2008.5388567.

[154] E. Salminen, A. Kulmala und T. D. Hamalainen. „Survey of network-on-chip proposals“. In: *white paper, OCP-IP* (2008), S. 1–13.

[155] R. Samanta, G. Venkataraman und Jiang Hu. „Clock Buffer Polarity Assignment for Power Noise Reduction“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17.6 (Juni 2009), S. 770–780. DOI: 10.1109/TVLSI.2008.2009187.

[156] S. Saponara, T. Bacchillone, E. Petri, L. Fanucci, R. Locatelli und M. Coppola. „Design of an NoC Interface Macrocell with Hardware Support of Advanced Networking Functionalities“. English. In: *IEEE Transactions on Computers* 63.3 (März 2014), S. 609–621. DOI: 10.1109/TC.2012.70.

[157] S. Saponara, L. Fanucci und M. Coppola. „Design and coverage-driven verification of a novel network-interface IP macrocell for network-on-chip interconnects“. In: *Microprocessors and Microsystems* 35.6 (Aug. 2011), S. 579–592. DOI: 10.1016/j.micpro.2011.06.005.

[158] S. Saponara, F. Vitullo, R. Locatelli, P. Teninge, M. Coppola und L. Fanucci. „LIME: A Low-latency and Low-complexity On-chip Mesochronous Link with Integrated Flow Control“. In: *2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*. IEEE, 2008, S. 32–35. DOI: 10.1109/DSD.2008.15.

[159] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber u. a. „T-CREST: Time-predictable multi-core architecture for embedded systems“. In: *Journal of Systems Architecture* 61.9 (Okt. 2015), S. 449–471. DOI: 10.1016/j.sysarc.2015.04.002.

[160] A. Sheibanyrad, I. M. Panades und A. Greiner. „Systematic comparison between the asynchronous and the multi-synchronous implementations of a network on chip architecture“. In: *Design, Automation and Test in Europe*. [IEEE Computer Society], 2007, S. 1090–1095. ISBN: 9783981080124.

[161] Z. Shi und A. Burns. „Real-Time Communication Analysis for On-Chip Networks with Wormhole Switching“. In: *NOCS '08 Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*. IEEE Computer Society, Apr. 2008, S. 161–170. ISBN: 978-0-7695-3098-7.

[162] G. Sievers. „Entwurfsraumexploration eng gekoppelter paralleler Rechnerarchitekturen“. Diss. 2016, S. 228.

[163] G. Sievers. „Implementierung eines Cache-Controllers für einen eingebetteten VLIW-Prozessor“. Diplomarbeit. Universität Paderborn, 2009.

[164] M. Singh und S. Nowick. „MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 15.6 (Juni 2007), S. 684–698. DOI: 10.1109/TVLSI.2007.898732.

[165] M. J. S. Smith. *Application-Specific Integrated Circuits*. Addison-wesley Vlsi Systems. Addison Wesley Professional, 1997. ISBN: 9780321602756.

[166] Sonics. *Sonics - SonicsGN*. 2017. URL: https://sonicsinc.com/wp-content/uploads/SonicsGN%7B%5C_%7DProductBrief%7B%5C_%7D151103.pdf.

[167] V. Soteriou, R. Ramanujam und B. Lin. „A High-Throughput Distributed Shared-Buffer NoC Router“. In: *IEEE Computer Architecture Letters* 8.1 (Jan. 2009), S. 21–24. DOI: 10.1109/L-CA.2009.5.

[168] V. Soteriou, N. Eisley, H. Wang, B. Li und L. S. Peh. „Polaris: A system-level roadmap for on-chip interconnection networks“. In: *IEEE International Conference on Computer Design, ICCD 2006* (2006), S. 134–141. DOI: 10.1109/ICCD.2006.4380806.

[169] J. Sparso, E. Kasapaki und M. Schoeberl. „An Area-efficient Network Interface for a TDM-based Network-on-Chip“. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*. IEEE Conference Publications, 2013, S. 1044–1047. DOI: 10.7873/DATE.2013.217.

[170] Standard Corporation Standard Performance Evaluation. *SPEC MPI 2007*. 2007. URL: https://www.spec.org/mpi2007/ (besucht am 17. 03. 2018).

[171] M. Stanisavljevic, M. Krstic und D. Bertozzi. *Advantages of GALS-based design*. Techn. Ber. 2010, S. 1–61. URL: http://www.galaxy-project.org/files/D24%7B%5C_%7DEPFL%7B%5C_%7DR002%7B%5C_%7DAdvantages%20of%20GALS-based%20design.pdf.

[172] *STMicroelectronics*. URL: www.st.com.

[173] StreamIt. *StreamIt Bechmark-Suite*. URL: http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml (besucht am 11. 03. 2018).

[174] *Synopsys, Inc.* URL: http://www.synopsys.com/.

[175] P. Teehan, M. Greenstreet und G. Lemieux. „A Survey and Taxonomy of GALS Design Styles“. In: *IEEE Design & Test of Computers* 24.5 (Sep. 2007), S. 418–428. DOI: 10.1109/MDT.2007.151.

[176] J. Teich. „Hardware/Software Codesign: The Past, the Present, and Predicting the Future“. In: *Proceedings of the IEEE* 100.Special Centennial Issue (2012), S. 1411–1430. DOI: 10.1109/JPROC.2011.2182009.

[177] W. Thies. „Language and compiler support for stream programs“. In: (2009). URL: http://pdf.aminer.org/000/329/363/using%7B%5C_%7Dthe%7B%5C_%7Dnovel%7B%5C_%7Dflc%7B%5C_%7Ddynamic%7B%5C_%7Dbuffer%7B%5C_%7Dsize%7B%5C_%7Dtuning%7B%5C_%7Dtechnique%7B%5C_%7Dto.pdf.

[178] W. Thies. *The StreamIt Compiler Infrastructure*. 2016. URL: https://github.com/bthies/streamit/ (besucht am 09. 07. 2017).

[179] W. Thies, M. Karczmarek und S. Amarasinghe. „StreamIt: A Language for Streaming Applications“. In: *International Conference on Compiler Construction*. Hrsg. von R. Horspool. Bd. 2304. Lecture Notes in Computer Science. Springer, 2002, S. 179–196. DOI: 10.1007/3-540-45937-5_14.

[180] Y. Thonnart, P. Vivet und F. Clermidy. „A fully-asynchronous low-power framework for GALS NoC integration“. In: *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, März 2010, S. 33–38. DOI: 10.1109/DATE.2010.5457239.

[181] J. Tlatlik. „CoreVA inside – Integrating the CoreVA-MPSoC with the AMiRo Platform“. Diss. 2015.

[182] J. Tlatlik. „Enhancement of an LLVM Compiler Backend for the CoreVA VLIW architecture“. Bachelorarbeit. Universität Bielefeld, 2013.

[183] S. V. Tota, M. R. Casu, M. R. Roch, L. Rostagno und M. Zamboni. „MEDEA: a hybrid shared-memory/message-passing multiprocessor NoC-based architecture“. In: *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 2010, S. 45–50. DOI: 10.1109/DATE.2010.5457237.

[184] A. T. Tran und B. M. Baas. „Achieving High-Performance On-Chip Networks With Shared-Buffer Routers“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.6 (Juni 2014), S. 1391–1403. DOI: 10.1109/TVLSI.2013.2268548.

[185] A. T. Tran, D. N. Truong und B. M. Baas. „A GALS many-core heterogeneous DSP platform with source-synchronous on-chip interconnection network“. In: *2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*. IEEE, 2009, S. 214–223. DOI: 10.1109/NOCS.2009.5071470.

[186] J. Turley. „Plurality Gets Ambitious with 256 CPUs“. In: *Microprocessor Report* (2010). ISSN: 0899-9341.

[187] A. Vajda. *Programming Many-Core Chips*. Springer US, 2011. DOI: 10.1007/978-1-4419-9739-5.

[188] X. Wang, G. Gan, J. Manzano, D. Fan und S. Guo. „A Quantitative Study of the On-Chip Network and Memory Hierarchy Design for Many-Core Processor“. In: *2008 14th IEEE International Conference on Parallel and Distributed Systems*. IEEE, 2008, S. 689–696. DOI: 10.1109/ICPADS.2008.18.

[189] M. Winter und G. P. Fettweis. „Guaranteed service virtual channel allocation in NoCs for run-time task scheduling“. In: *2011 Design, Automation & Test in Europe*. IEEE, März 2011, S. 1–6. DOI: 10.1109/DATE.2011.5763073.

[190] H. Wörn und U. Brinkschulte. *Echtzeitsysteme - Grundlagen, Funktionsweisen, Anwendungen*. Springer, Berlin, Heidelberg, 2005. DOI: https://doi.org/10.1007/b139050.

[191] J. Wu, S. Furber und J. Garside. „A Programmable Adaptive Router for a GALS Parallel System“. In: *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*. IEEE, Mai 2009, S. 23–31. DOI: 10.1109/ASYNC.2009.17.

[192] I. Xilinx. *Virtex-5 Familiy Overview*. 2015. URL: https://www.xilinx.com/support/documentation/data%7B%5C_%7Dsheets/ds100.pdf (besucht am 01. 11. 2018).

[193] P. M. Yaghini, A. Eghbal, S. A. Asghari und H. Pedram. „Power comparison of an asynchronous and synchronous network on chip router“. In: *2009 14th International CSI Computer Conference*. IEEE, Okt. 2009, S. 242–246. DOI: 10.1109/CSICC.2009.5349422.

[194] Yow-Tyng Nieh, Shih-Hsu Huang und Sheng-Yu Hsu. „Minimizing peak current via opposite-phase clock tree“. In: *Proceedings. 42nd Design Automation Conference, 2005.* IEEE, 2005, S. 182–185. DOI: 10.1109/DAC.2005.193797.

[195] S. Zeidler und M. Krstic. „A survey about testing asynchronous circuits“. In: *2015 European Conference on Circuit Theory and Design (ECCTD)*. IEEE, Aug. 2015, S. 1–4. DOI: 10.1109/ECCTD.2015.7300128.

[196] Y. Zhang, L. S. Heck, M. T. Moreira, D. Zar, M. Breuer, N. L. Calazans und P. A. Beerel. „Design and Analysis of Testable Mutual Exclusion Elements“. In: *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*. IEEE, Mai 2015, S. 124–131. DOI: 10.1109/ASYNC.2015.28.

Eigene Veröffentlichungen

[197] J. Ax, A. Buda, D. Schneider, J. Hartfiel, L. Dürkop, T. Jungeblut, J. Jasperneite, A. Vedral und U. Rückert. „Universelle Echtzeit-Ethernet Architektur zur Integration in rekonfigurierbare Automatisierungssysteme“. In: *45. Jahrestagung der Gesellschaft für Informatik (INFORMATIK)*. 2015.

[198] J. Ax, M. Flasskamp, G. Sievers, C. Klarhorst, T. Jungeblut und W. Kelly. *An Abstract Model for Performance Estimation of the Embedded Multiprocessor CoreVA-MPSoC Technical Report (v1.0)*. Techn. Ber. Universit{\"{a}}t Bielefeld, 2015, S. 3. DOI: http://dx.doi.org/10.13140/RG.2.1.1090.2483.

[199] J. Ax, N. Kucza, M. Vohrmann, T. Jungeblut, M. Porrmann und U. Rückert. „Comparing synchronous, mesochronous and asynchronous NoCs for GALS based MPSoC“. In: *IEEE 11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-17)*. 2017.

[200] J. Ax, G. Sievers, J. Daberkow, M. Flasskamp, M. Vohrmann, T. Jungeblut, W. Kelly, M. Porrmann und U. Ruckert. „CoreVA-MPSoC: A Many-Core Architecture with Tightly Coupled Shared and Local Data Memories“. In: *IEEE Transactions on Parallel and Distributed Systems* (2017), S. 1–1. DOI: 10.1109/TPDS.2017.2785799.

[201] J. Ax, G. Sievers, M. Flasskamp, W. Kelly, T. Jungeblut und M. Porrmann. „System-Level Analysis of Network Interfaces for Hierarchical MPSoCs“. In: *International Workshop on Network on Chip Architectures (NoCArc)*. ACM, 2015. DOI: http://dx.doi.org/10.1145/2835512.2835513.

[202] A. Buda, M. Walter, J. Hartfiel, J. Ax, K. Nussbaum, T. Jungeblut und M. Porrmann. „Automatische Protokollanpassung von Echtzeit-Ethernet-Standards durch FPGA-Technologien“. In: *VDI-Kongress AUTOMATION 2015*. 2015.

[203] M. Flasskamp, G. Sievers, J. Ax, C. Klarhorst, T. Jungeblut, W. Kelly, M. Thies und M. Porrmann. „Performance Estimation of Streaming Applications for Hierarchical MPSoCs“. In: *Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*. ACM Press, 2016. DOI: 12.345/678.

[204] T. Jungeblut, J. Ax, M. Porrmann und U. Rückert. „A TCMS-based architecture for GALS NoCs“. In: *2012 IEEE International Symposium on Circuits and Systems*. IEEE, 2012, S. 2721–2724. DOI: 10.1109/ISCAS.2012.6271870.

[205] T. Jungeblut, J. Ax, G. Sievers, B. Hübener, M. Porrmann und U. Rückert. „Resource Efficiency of Scalable Processor Architectures for SDR-based Applications“. In: *Proceedings of the Radar, Communication and Measurement Conference (RADCOM)*. 2011. URL: http://pub.uni-bielefeld.de/publication/2476993.

[206] W. Kelly, M. Flasskamp, G. Sievers, J. Ax, J. Chen, C. Klarhorst, C. Ragg, T. Jungeblut und A. Sorensen. „A communication model and partitioning algorithm for streaming applications for an embedded MPSoC“. In: *2014 International Symposium on System-on-Chip (SoC)*. IEEE, 2014. DOI: 10.1109/ISSOC.2014.6972436.

[207] C. Klarhorst, M. Flasskamp, J. Ax, T. Jungeblut, W. Kelly, M. Porrmann und U. Rückert. „Development of Energy Models for Design Space Exploration of Embedded Many-Core Systems“. In: *HIP3ES*. 2018. arXiv: 1801.04242. URL: http://arxiv.org/abs/1801.04242.

[208] S. Korf, G. Sievers, J. Ax, D. Cozzi, T. Jungeblut, J. Hagemeyer, M. Porrmann und U. Rückert. „Dynamisch rekonfigurierbare Hardware als Basistechnologie für intelligente technische Systeme“. In: *Wissenschaftsforum 2013 Intelligente Technische Systeme*. Proceedings Wissenschaftsforum 2013 Intelligente Technische Systeme. 2013, S. 79–90. ISBN: 978-3-942647-29-8.

[209] G. Sievers, J. Ax, N. Kucza, M. Flaßkamp, T. Jungeblut, W. Kelly, M. Porrmann und U. Rückert. „Evaluation of Interconnect Fabrics for an Embedded MPSoC in 28nm FD-SOI“. In: *Iscas*. IEEE, Mai 2015, S. 1925–1928. DOI: 10.1109/ISCAS.2015.7169049.

[210] G. Sievers, J. Daberkow, J. Ax, M. Flasskamp, W. Kelly, T. Jungeblut, M. Porrmann und U. Rückert. „Comparison of Shared and Private L1 Data Memories for an Embedded MPSoC in 28nm FD-SOI“. eng. In: *International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)* (2015). URL: https://pub.uni-bielefeld.de/publication/2760622.

[211] G. Sievers, B. Hübener, J. Ax, M. Flasskamp, W. Kelly, T. Jungeblut und M. Porrmann. „The CoreVA-MPSoC: A Multiprocessor Platform for Software-Defined Radio“. In: *Computing Platforms for Software-Defined Radio*. Hrsg. von W. Hussain, J. Nurmi, J. Isoaho und F. Garzia. Springer International Publishing, 2017, S. 29–59. DOI: 10.1007/978-3-319-49679-5_3.

[212] M. Walter, J. Ax, A. Buda, K. Nussbaum, J. Hartfiel, T. Jungeblut und M. Porrmann. „Dynamische Rekonfiguration von Echtzeit-Ethernet-Standards mit harten Echtzeitanforderungen“. In: *Kommunikation in der Automation (KommA 2014)*. 2014.

Betreute Arbeiten

[213] C. Ascheberg. „Bewertung von Trace-Systemen für den Einsatz während der Entwicklung und im produktiven Betrieb eingebetteter Systeme“. In: Dezember (2015).

[214] L.-D. Braun. „Ein Simulator für eingebettete Multiprozessoren“. Master-Projekt. Universität Bielefeld, 2013.

[215] L.-D. Braun. „Implementierung eines SystemC-basierten virtuellen Prototypen für ein MPSoC“. Diss. 2012.

[216] L.-D. Braun. „OpenCL Support for the CoreVA-MPSoC“. Masterarbeit. Universität Bielefeld, 2015.

[217] J. Daberkow. „Eng gekoppelte, gemeinsame Datenspeicher im CoreVA-Cluster“. Bachelorarbeit. Universität Bielefeld, 2014.

[218] J. Daberkow, A. Gatting und T. Michalski. „Schnittstellenoptimierung des CoreVA-MPSoC-FPGA-Prototypen zur Durchführung und Visualisierung von Regressionstests“. Diss. 2016.

[219] A. Gatting. „NoC-Topologien und Routingstrategien im CoreVA-MPSoC“. Bachelorarbeit. Universität Bielefeld, 2014.

[220] K. Gravermann, P. Wallbaum, N. Kucza und P. Geisler. „Hardware-based Data Transmission in an Embedded Multiprocessor“. Master-Projekt. Universität Bielefeld, 2015.

[221] J. Homburg. „Optimierung eines eingebetteten VLIW-Prozessors für die effiziente Berechnung von künstlichen neuronalen Netzen“. In: März (2017).

[222] P. Jünemann. „Ein L1-Instruktionscache für das CoreVA-MPSoC“. Diss. 2015.

[223] C. Klarhorst. „An Energy-Model for the Automatic Partitioning of Applications on Embedded Many-Cores“. Diss. 2017, S. 61.

[224] N. Kucza. „Eine asynchrone Kommunikationsinfrastruktur für das eingebettete Multiprozessorsystem CoreVA-MPSoC“. Masterarbeit. Universität Bielefeld, 2016.

[225] N. Kucza. „Implementierung eines AXI-Interconnects für ein eingebettetes Multiprozessorsystem“. Bachelorarbeit. Universität Bielefeld, 2013.

[226] J. Schwuchow. „Ein Multi-FPGA Prototyp des CoreVA-MPSoC“. Bachelorarbeit. Universität Bielefeld, 2014.

Anhang

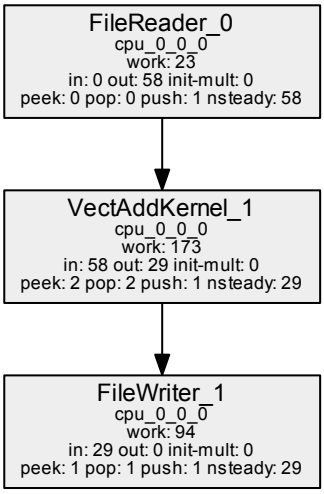
A NoC-Grundlagen

Legende	
Kantengrad	Maximale Anzahl I/O-Ports pro Knoten
Symmetrie	Überall identische Sichtweise auf das Netzwerk
Homogenität	Alle Knoten besitzen den gleichen Kantengrad
Bisektionsbandbreite	Belastung pro Verbindung - unidirektional
Hop-Anzahl	Anzahl der Hops zw. zwei Knoten (Worst Case)
Durchschnittl. Hop-Anzahl	Durchschnittl. Anzahl der Hops zw. zwei Knoten
Konnektivität	Ausfallsicherheit - bidirektional
Netzwerkverbindungen	Gesamte Netzwerkverbindungen - unidirektional
n	Anzahl Dimensionen
k	Anzahl Knoten pro Dimension
m	Anzahl Endknoten pro Knoten
N	Gesamtanzahl Knoten
e	Kantengrad des Knoten zum Netzwerk
l	Ausdehnungslevel
h	Stufen des Baumes
B	Breite der Topologie
L	Länge der Topologie
t	Anzahl Ringe aus Hexagone

Tabelle A.1: Legende zur Topologie Übersicht und deren Eigenschaften, verwendete Variablen und Besonderheiten

B Programmbeispiele für das CoreVA-MPSoC

```
void->void pipeline VectAdd {
add FileReader<int>("input.stream");
add VectAddFilter();
add FileWriter<int>("output.stream");
}
int->int filter VectAddFilter {
work pop 2 push 1 {
int t1, t2;
t1 = pop();
t2 = pop();
push(t1 + t2);
}
}
```



C Eigenschaften der betrachteten Beispielanwendungen

Tabelle C.1: Allgemeine Eigenschaften der betrachteten Beispielanwendungen

Name	Komplexität	Filter-anzahl	theor. Beschleunig. 32 CPUs	theor. Beschleunig. 128 CPUs	Verhältnis Ein-/Ausgabe	Verhältnis int. Komm./ Ausgabe
LowPassFiler	$\mathcal{O}(n)$	3	32,0	128,0	3	4
Filterbank	$\mathcal{O}(n)$	51	32,0	128,0	8	51
AutoCor	$\mathcal{O}(n * \tau)$	10	32,0	128,0	32	33
FFT	$\mathcal{O}(n \log_2(n))$	24	32,0	128,0	1	12
SVM	$\mathcal{O}(n^3)$	11	32,0	128,0	27	616
MatrixMult	$\mathcal{O}(n^3)$	38	22,2	22,2	2	36
DES	$\mathcal{O}(n)$	71	32,0	79,5	4	32
BubbleSort	$\mathcal{O}(n^2)$	66	32,0	62,8	1	65
RadixSort	$\mathcal{O}(n \log_2(w))$	13	32,0	128,0	1	12
BitonicSort	$\mathcal{O}(n \log_2(n)^2)$	274	32,0	128,0	1	17

Dieser Abschnitt zeigt verschiedene Eigenschaften der in Abschnitt 4.2.3 vorgestellten Benchmarks. Die allgemeinen Eigenschaften der Anwendungen sind in Tabelle C.1

aufgeführt. Zunächst ist die allgemeine Komplexität der Anwendungen als asymptotische obere Schranke angegeben [37, S. 47 f.]. n bezeichnet hierbei die Blockgröße bzw. die Anzahl der Eingangsdaten. Für AutoCor ist τ die diskrete Verschiebung und damit die Anzahl von Ausgangsdaten. Bei RadixSort ist w die maximale Größe der zu sortierenden Zahlen. Des Weiteren ist die Anzahl der StreamIt-Filter der Anwendungen angegeben. Beispielcode einer StreamIt-Anwendung mit drei Filtern ist in Abbildung B zu finden. Die theoretisch maximal mögliche Beschleunigung der Anwendungen für 32 und 128 CPUs ist mithilfe des in der Dissertation [162, S. 91 f.] vorgestellten analytischen Modells bestimmt. Der durch dieses Modell ermittelte Wert zeigt das Potenzial einer Anwendung für eine parallele Verarbeitung, ohne die Berücksichtigung von Kommunikations- und Speicherkosten.

Zudem ist das Verhältnis von Eingangs- und Ausgangsdatenrate sowie das Verhältnis der Summe der internen Kommunikation (Task-zu-Task bzw. Filter-zu-Filter) und der Ausgangsdatenrate angegeben. Von der Anzahl der Filter einer Anwendung kann nicht die Menge der internen Kommunikation abgeleitet werden. Dies wird deutlich am Beispiel BitonicSort, diese Anwendung verfügt zwar über 274 Filter, besitzt jedoch nur ein Verhältnis von interner Kommunikation und Ausgangsdatenrate von 17. Dies ist darauf zurückzuführen, dass die Filter von BitonicSort als Baum aufgebaut sind und jeder Ast dieses Baums nur einen Teil der Daten sortiert.

Tabelle C.2: Eigenschaften der betrachteten Beispielanwendungen bei einer Abbildung auf das CoreVA-MPSoC

Name	Filter-	Instr.-Speicher in kB		Kommunikationsdatenrate[1]			
	anzahl	krit. CPU	32 CPUs	Eingang	Ausgang	Intern	NoC
LowPassFiler	13	28	120	1,28	0,42	3,87	2,26
Filterbank	51	13	307	1,28	0,16	7,98	6,84
AutoCor	26	13	226	4,22	0,13	4,35	3,99
FFT	38	10	252	2,76	2,76	33,09	31,02
SVM	20	52	886	0,03	0,0002	0,12	0,11
MatrixMult	38	11	295	2,41	1,205	43,37	41,26
DES	104	14	360	2,27	0,57	18,17	16,14
BubbleSort	66	13	343	0,41	0,41	26,7	24,23
RadixSort	35	10	238	1,84	1,84	28,54	22,13
BitonicSort	274	39	608	0,90	0,90	15,31	9,96

In Tabelle C.2 sind verschiedene Eigenschaften der betrachteten Anwendungen dargestellt, die sich bei einer Partitionierung auf eine Konfiguration des CoreVA-MPSoCs mit 32 CPUs ergeben. Beispielhaft sind hier die Ergebnisse für ein CoreVA-MPSoC mit einem 4x2-Mesh-NoC und jeweils 4 CPUs pro Cluster aufgeführt.

[1] in Bits/CPU-Takt.

Bei der Partitionierung einer Anwendung auf eine konkrete Hardwarekonfiguration kann der CoreVA-MPSoC-Compiler 3.4.4 verschiedene Strategien durchführen um die Anwendung zu Beschleunigen. So können beispielsweise Zustandslose Filter vervielfältigt werden, um die Datenparallelität einer Anwendung auszunutzen. Dies spiegelt sich in TabelleC.2 bei der Anzahl Filter wieder, die sich bei manchen Anwendungen von der aus TabelleC.1 unterscheidet (z.B. LowPassFiler).

Des Weiteren sind die Größe des benötigten Instruktionsspeichers (Größe des Programmabbilds) angegeben. Dies ist zum einen die maximale Größe des benötigten Instruktionsspeichers einer einzelnen CPU und zum anderen die gesamte Größe des Programmabbilds aller CPUs zusammen. Hierbei muss angemerkt werden, dass im Rahmen des CoreVA-MSoC-Projekts zurzeit an einer Verringerung der Größe des Programmabbilds gearbeitet wird. Durch die automatische Code-Generierung besteht hier noch das Potenzial den Programmcode zu verringern.

Um das Kommunikationsaufkommen einer Anwendung zu zeigen, sind neben der Ein- und Ausgangsdatenrate auch die gesamte interne Kommunikation die sich zwischen den Filtern ergibt aufgeführt. Alle Kommunikationsraten sind in Bits pro Taktzyklus angegeben. Des Weiteren ist auch die Kommunikationsrate die sich über das NoC ergibt aufgeführt. Dabei fällt auf, dass zum Beispiel die Anwendung SVM nur wenig kommuniziert und die Berechnungen innerhalb der Filter im Verhältnis dazu recht aufwendig sind. Bei anderen Anwendungen, z.B. FFT oder MatrixMult fällt die Kommunikation jedoch deutlich mehr ins Gewicht.

D Spice Simulationen asynchroner Schaltungen

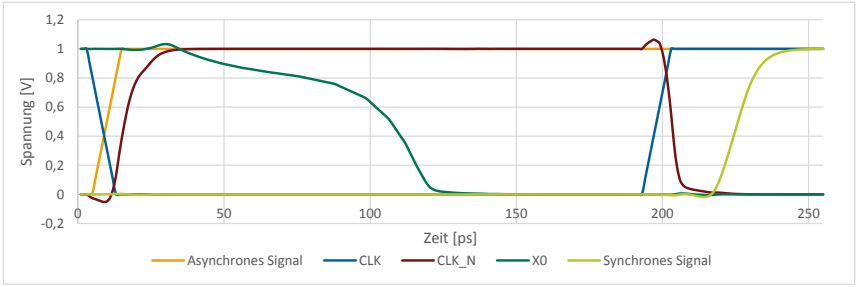


Abbildung A.1: SPICE-Simulation des Synchronisierers für ein Bit. Dargestellt ist der Wechsel von logisch 0 zu logisch 1 im ungünstigsten Fall.

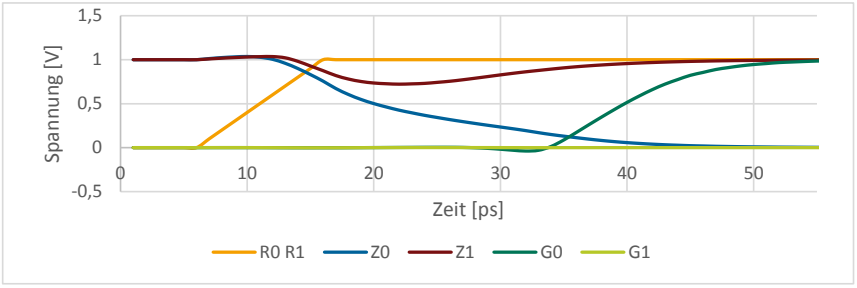


Abbildung A.2: SPICE-Simulation des Arbiterverhaltens (Arbiter aus Abbildung 5.17a) bei gleichzeitiger Anfrage an beiden Eingängen. Die Metastabilität wird dabei von den NORs erfolgreich herausgefiltert.

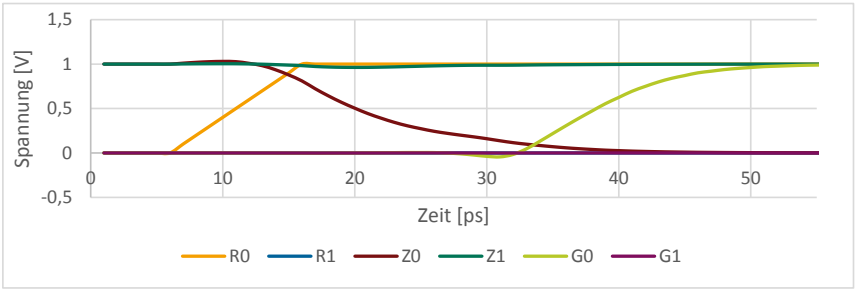


Abbildung A.3: SPICE-Simulation des Arbiterverhaltens (Arbiter aus Abbildung 5.17a) bei nur einer eingehenden Anfrage.

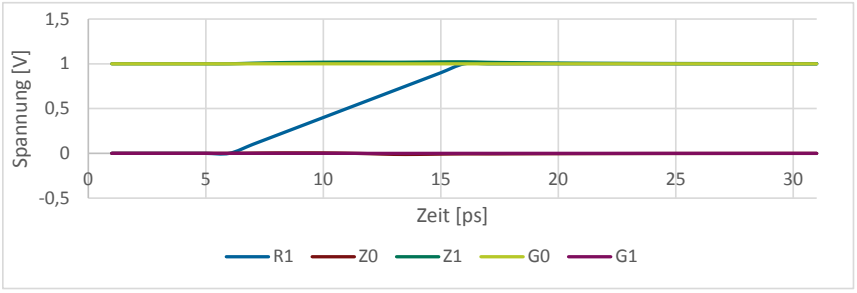


Abbildung A.4: SPICE-Simulation des Arbiterverhaltens (Arbiter aus Abbildung 5.17a) einer zusätzlichen Anfrage bei einer bereits bestehenden Gewährung.

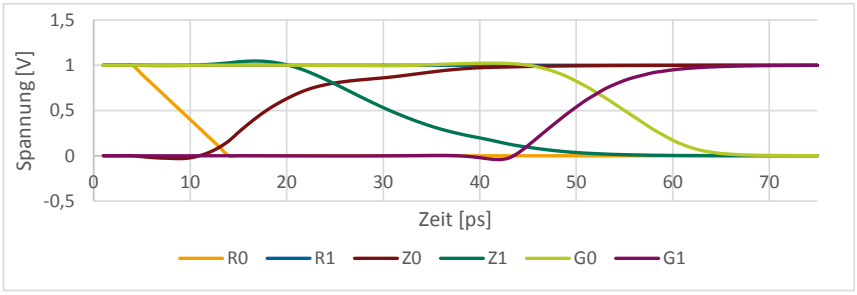


Abbildung A.5: SPICE-Simulation des Arbiterverhaltens (Arbiter aus Abbildung 5.17a) beim Entfernen einer gewährten Anfrage mit bestehender zusätzlichen Anfrage am anderen Eingang.

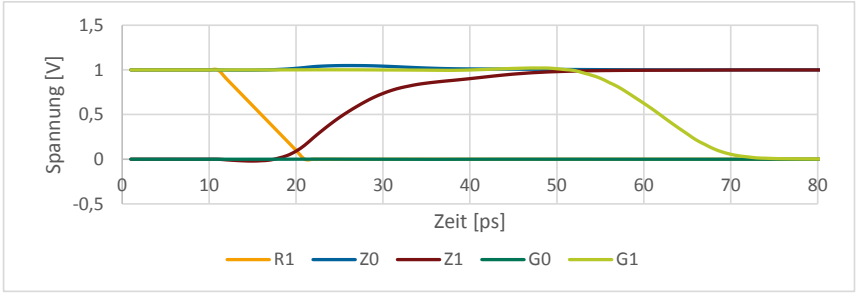


Abbildung A.6: SPICE-Simulation des Arbiterverhaltens (Arbiter aus Abbildung 5.17a) beim Entfernen einer gewährten Anfrage ohne bestehender Anfrage am anderen Eingang.

E Ergebnisse für den CPU-Cluster

Abbildung A.7 stellt die Ergebnisse zur maximalen Taktfrequenz für verschiedene Konfigurationen des gemeinsamen L1 Speichers dar. Das verwendete CPU-Cluster für diese Untersuchungen besteht aus 8 CPUs ohne lokale Datenspeicher. Die Anzahl Speicherbänke variiert zwischen 2 und 128 mit 16 kB pro Bank

Abbildung A.9 stellt die Ergebnisse für verschiedene CoreVA-MPSoC-Konfigurationen dar. Eine CoreVA-CPU besteht dabei aus dem CPU-Makro mit je 16 kB Instruktionsspeicher und 16 kB Datenspeicher. Die Ergebnisse wurden bereits in [209] veröffentlicht. Bei einer Zielfrequenz von 830 MHz variierte die maximale Taktfrequenz der verschiedenen MPSoC-Konfigurationen zwischen 812 MHz und 830 MHz.

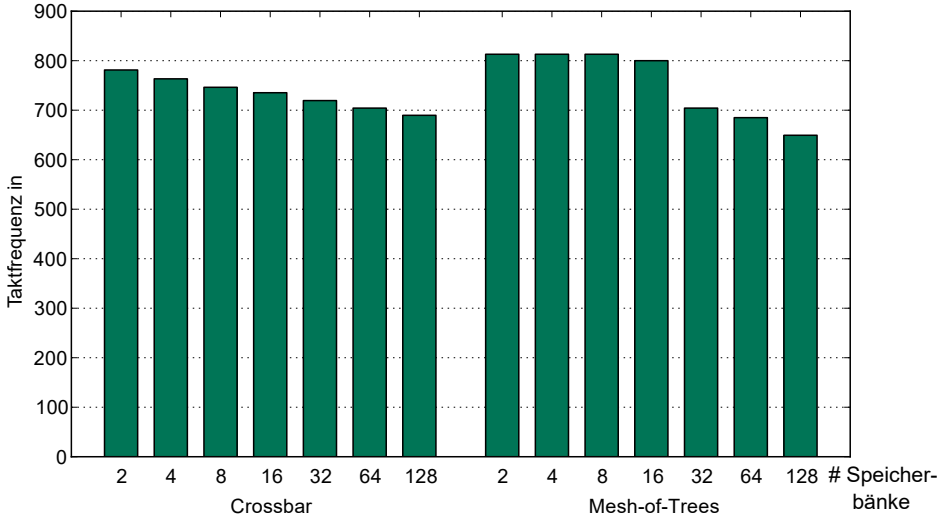


Abbildung A.7: Maximale Taktfrequenz beim gemeinsamen L1 Speicher für eine verschiedene Anzahl von Speicherbänken (16 kB pro Bank).[162]

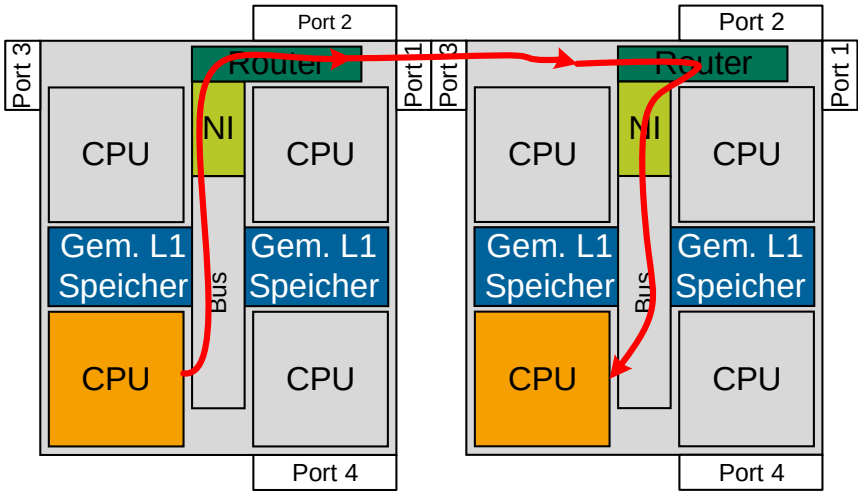


Abbildung A.8: Block-Diagramm zwei verbundener Cluster-Knoten. Die bei der Kommunikation über das NoC involvierten Komponenten sind hervorgehoben

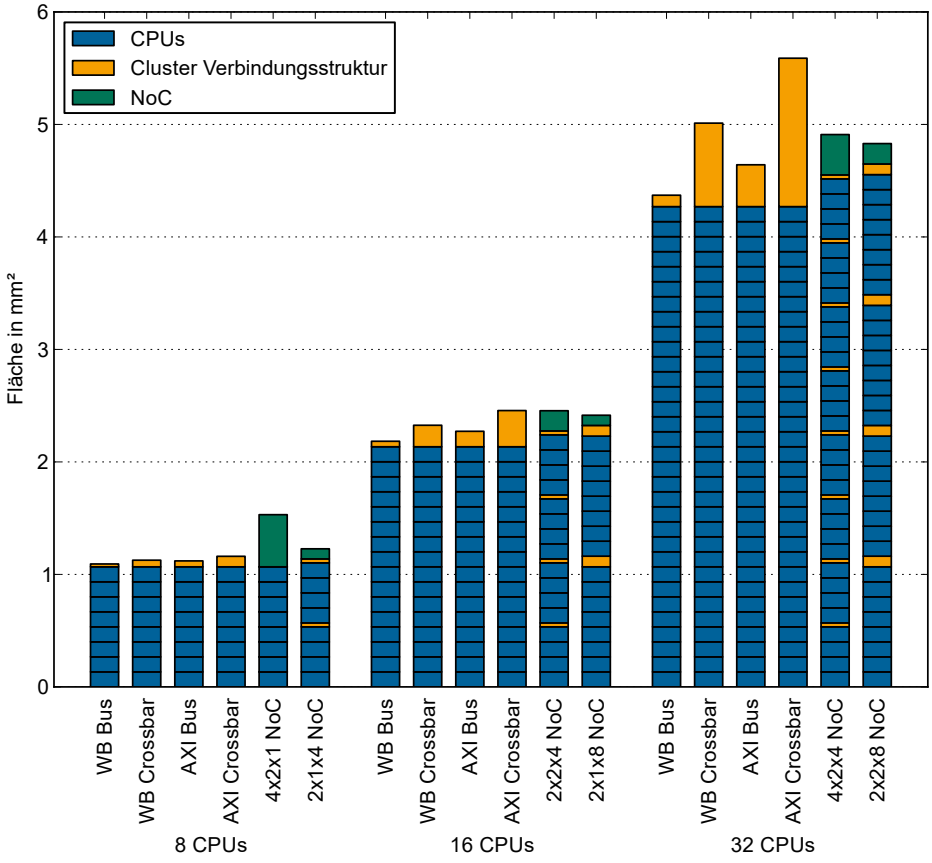


Abbildung A.9: Vergleich verschiedener CoreVA-MPSoC Konfigurationen). [209]