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

Entwurfsraumexploration eng gekoppelter paralleler Rechnerarchitekturen

zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEUR (Dr.-Ing.)

der Technischen Fakultät
der Universität Bielefeld

genehmigte Dissertation

von

Dipl.-Ing. Gregor Sievers

Referent:	Prof. Dr.-Ing. Ulrich Rückert
Korreferentin:	Prof. Dr. rer. nat. Sybille Hellebrand

Tag der mündlichen Prüfung: 20.07.2016

Bielefeld / Juli 2016
DISS KS / 08

Kurzfassung

Eingebettete mikroelektronische Systeme finden in vielen Bereichen des täglichen Lebens Anwendung. Die Integration von zunehmend mehr Prozessorkernen auf einem einzelnen Mikrochip (On-Chip-Multiprozessor, MPSoC) erlaubt eine Steigerung der Rechenleistung und der Ressourceneffizienz dieser Systeme. In der AG Kognitronik und Sensorik der Universität Bielefeld wird das CoreVA-MPSoC entwickelt, welches ressourceneffiziente VLIW-Prozessorkerne über eine hierarchische Verbindungsstruktur koppelt. Eine enge Kopplung mehrerer Prozessorkerne in einem Cluster ermöglicht hierbei eine breitbandige Kommunikation mit geringer Latenz.

Der Hauptbeitrag der vorliegenden Arbeit ist die Entwicklung und Entwurfsraumexploration eines ressourceneffizienten CPU-Clusters für den Einsatz im CoreVA-MPSoC. Eine abstrakte Modellierung der Hardware- und Softwarekomponenten des CPU-Clusters sowie ein hoch automatisierter Entwurfsablauf ermöglichen die schnelle Analyse eines großen Entwurfsraums. Im Rahmen der Entwurfsraumexploration werden verschiedene Topologien, Busstandards und Speicherarchitekturen untersucht. Insbesondere das Zusammenspiel der Hardware-Architektur mit Programmiermodell und Synchronisierung ist evident für eine hohe Ressourceneffizienz und eine gute Ausnutzung der verfügbaren Rechenleistung durch den Anwendungsentwickler. Dazu wird ein an die Hardwarearchitektur angepasstes blockbasiertes Synchronisierungsverfahren vorgestellt. Dieses Verfahren wird von Compilern für die Sprachen StreamIt, C sowie OpenCL verwendet, um Anwendungen auf verschiedenen Konfigurationen des CPU-Clusters abzubilden. Neun repräsentative Streaming-Anwendungen zeigen bei der Abbildung auf einem Cluster mit 16 CPUs eine durchschnittliche Beschleunigung um den Faktor 13,3 gegenüber der Ausführung auf einer CPU. Zudem wird ein eng gekoppelter gemeinsamer L1-Datenspeicher mit mehreren Speicherbänken in den CPU-Cluster integriert, der allen CPUs einen Zugriff mit geringer Latenz erlaubt. Des Weiteren wird die Verwendung verschiedener Instruktionsspeicher und -caches evaluiert sowie der Energiebedarf für Kommunikation und Synchronisierung im CPU-Cluster betrachtet.

Es wird in dieser Arbeit gezeigt, dass ein CPU-Cluster mit 16 CPU-Kernen einen guten Kompromiss in Bezug auf den Flächenbedarf der Cluster-Verbindungsstruktur sowie die Leistungsfähigkeit des Clusters darstellt. Ein CPU-Cluster mit 16 2-Slot-VLIW-CPUs und insgesamt 512 kB Speicher besitzt bei einer prototypischen Implementierung in einer 28-nm-FD-SOI-Standardzellenbibliothek einen Flächenbedarf von 2,63 mm^2. Bei einer Taktfrequenz von 760 MHz liegt die durchschnittliche Leistungsaufnahme bei 440 mW. Eine FPGA-basierte Emulation auf einem Xilinx Virtex-7-FPGA erlaubt die Evaluierung eines CoreVA-MPSoCs mit bis zu 24 CPUs bei einer maximalen Taktfrequenz von bis zu 124 MHz. Als weiteres Anwendungsszenario wird ein CoreVA-MPSoC mit bis zu vier CPUs auf das FPGA des autonomen Miniroboters AMiRo abgebildet.

Inhaltsverzeichnis

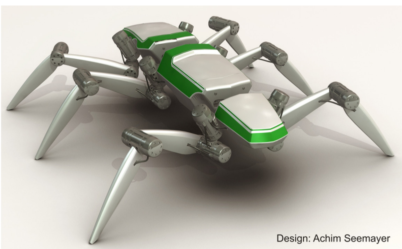
1 Einleitung

Mobile und eingebettete digitale Systeme finden in einer Vielzahl von Lebensbereichen Verwendung. Im Jahr 2020 werden 50 Milliarden vernetze Geräte („Internet der Dinge“) erwartet [36]. Anwendungen wie *Software-Defined Radio* (SDR) [23], 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 [171] verfügt über zwei Kameras sowie drei Mikrofone. Der AMiRo [91; 172] 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 und Rechenleistung. Die stetigen Fortschritte in der Mikroelektronik erlauben die Integration

[1] *Autonomous Mini Robot*



(a) Roboter-Stabheuschrecke Hector

(b) Miniroboter AMiRo

Abbildung 1.1: An der Universität Bielefeld entwickelte Roboter von Milliarden Transistoren auf einem Chip [96]. Die Anzahl der Transistoren pro Chip steigt exponentiell an [138]. Diese hohe Zahl an Transistoren bedingt jedoch gesteigerte Anforderungen an die Entwurfsmethodik und Systemarchitektur von integrierten Bausteinen [111, S. 669]. Zudem sinken – zumindest zum Zeitpunkt der Entstehung dieser Arbeit – durch eine Verkleinerung der Strukturgrößen nicht mehr automatisch die Kosten pro integriertem Transistor [76]. 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 Halbleister-Chips [44].

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 [87]. 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 [108].

Neben den Verarbeitungseinheiten stellen jedoch auch die On-Chip-Verbindungsstrukturen eine zunehmend kritische Größe für die Ressourceneffizienz eines Multiprozessorsystems dar [57]. 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 zwischen verschiedenen Komponenten im System. Ein weiterer Ansatz ist die Verwendung

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

von On-Chip-Netzwerken (NoC[5]). 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 [39]. 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 [39]. 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 relativ wenige (bis ca. 16) leistungsstarke CPU-Kerne. Beispiele sind Prozessoren von Intel (Xeon E5 [73]) oder IBM (Power8 [60; 75]). Die Integration von immer mehr dieser leistungsstarken CPUs auf einem Chip ist aufgrund der hohen Leistungsaufnahme nicht sinnvoll, wie Borkar [26] anführt. Im Gegensatz zu Multi-Core-Systemen bestehen *Many-Core*-MPSoCs aus einer 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 [155, 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 [19], dass ein eingebetteter SPARC V8 Prozessor 70 % seines Energiebedarfs für die Bereitstellung und Behandlung (Dekodierung) von Instruktionen und Daten benötigt. Eine selbst entwickelte, optimierte CPU benötigt hierfür immerhin 51 % der Energie. Dieses Problem vergrößert sich, da der Anteil an Speicher

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

an der Gesamtfläche von MPSoCs kontinuierlich ansteigt [96]. Software-gesteuerte Speicher (*Scratchpad-Memories*) können eine ressourceneffiziente Alternative zu Caches darstellen [20]. 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 [205]. Eine nachrichtenbasierte Kommunikation umgeht diesen Flaschenhals, indem Synchronisation und Daten direkt zwischen einzelnen CPUs ausgetauscht werden [196].

Eine effiziente Partitionierung einer Anwendungen auf dutzende oder hunderte Prozessorkerne stellt eine große Herausforderung für Softwareentwickler dar. Scratchpad-basierte Speicherarchitekturen erfordern eine explizite Verwaltung der verschiedenen Speicher durch die (Anwendungs-) Software. 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] [94] oder StreamIt [195]) 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 [20].

1.2 Das CoreVA-MPSoC

An der Arbeitsgruppe Kognitronik und Sensorik [12] der Universität Bielefeld wird das ressourceneffiziente Multiprozessorsystem CoreVA-MPSoC entwickelt, das Parallelität auf verschiedenen Abstraktionsebenen anwendet, um eine hohe Ressourceneffizienz des Gesamtsystems zu ermöglichen. Parallelität auf Daten- und Instruktionsebene (*Data Level Parallelism*, DLP sowie *Instruction Level Parallelism*, ILP) werden im CoreVA[9]-Prozessor durch die Verwendung von mehreren VLIW[10]-Slots sowie durch die Integration von SIMD[11]-Rechenwerken ausgenutzt. Die CoreVA-CPU besitzt getrennte Speicher für Instruktionen und Daten sowie sechs Pipeline-Stufen. Die Anzahl der Verarbeitungseinheiten kann zur Entwurfszeit konfiguriert werden [217]. Weitere Details zum CoreVA-Prozessor sind in Abschnitt 2.2 zu finden. Parallelität auf Thread-Ebene (*Thread Level Parallelism*, TLP) wird im hierarchischem Multiprozessorsystem CoreVA-MPSoC

[7] *Direct Memory Access*
[8] *Open Computing Language*
[9] *Configurable Ressource Efficient VLIW Architecture*
[10] *Very Long Instruction Word*
[11] *Single Instruction Multiple Data*

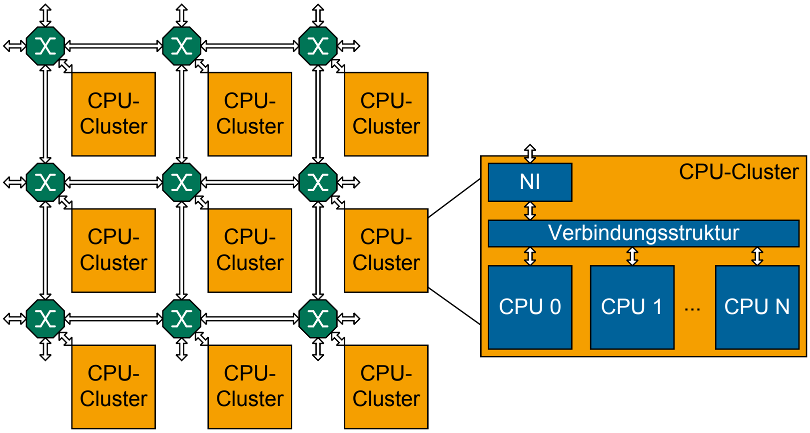


Abbildung 1.2: Blockschaltbild des hierarchischen CoreVA-MPSoCs

angewandt (siehe Abbildung 1.2). 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 [39; 223]. Die Entwurfsraumexploration des CPU-Clusters ist ein Hauptbestandteil der vorliegenden Arbeit.

Mehrere CPU-Cluster werden über ein On-Chip-Netzwerk miteinander verbunden. Dies erlaubt die Integration von vielen hundert CPU-Kernen auf einem einzelnen Chip. Das NoC basiert auf der GigaNoC-Architektur [166] und verwendet eine paketbasierte Kommunikation und das *Wormhole*-Routing-Verfahren. Die einzelnen Routing-Knoten werden *Switch-Box* oder *Router* genannt und empfangen Flits über Eingangsports und leiten diese an Ausgangsports weiter. In Abbildung 1.2 ist das CoreVA-MPSoC mit einer 2D-Mesh-Topologie dargestellt. Jede Switch-Box besitzt fünf Ports, wobei vier Ports mit den jeweiligen direkten Nachbarn verbunden sind. Der fünfte Port ist mit dem CPU-Cluster über ein *Network Interface* (NI) verbunden [100]. Ausgehende Pakete sind in einem Speicher im Cluster gespeichert und werden vom NI hieraus gelesen, segmentiert und über das NoC versandt. Eingehende Pakete werden vom NI entsprechend analysiert und z. B. im Speicher der Empfänger-CPU abgelegt. Eine hohe Konfigurierbarkeit erlaubt eine Anpassung und Optimierung des CoreVA-MPSoCs an den jeweiligen Anwendungsfall.

Die Kommunikation zwischen zwei CPU-Clustern erfolgt nachrichtenbasiert. Innerhalb eines CPU-Clusters existiert ein (verteilter oder zentraler) gemeinsamer Speicher, der für die Kommunikation zwischen den CPUs verwendet werden kann. Abhängig von der Anwendung kann auch innerhalb eines Clusters eine nachrichtenbasierte Kommunikation Vorteile aufweisen (siehe Abschnitt 2.6.4).

Zur Programmierung des CoreVA-MPSoCs steht ein LLVM[12]-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 ein einheitliches Programmiermodell entstanden, das in Abschnitt 2.6.4 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 [229] und StreamIt [195] zur Verfügung. Der CoreVA-MPSoC-Compiler für die Sprache StreamIt ist im Rahmen einer Kooperation mit der Queensland University of Technology (QUT) in Brisbane, Australien entstanden [221]. Eine StreamIt-Anwendung besteht aus mehreren Filtern, die über unidirektionale Kanäle miteinander kommunizieren. Filter können parallel ausgeführt und daher auf verschiedene CPUs partitioniert werden.

Eine Simulation bzw. Emulation des CoreVA-MPSoCs ist auf verschiedenen Abstraktionsebenen möglich. Dies erlaubt Software- und Hardwareentwicklern die funktionale Verifikation einer Anwendung bzw. einzelner Komponenten des CoreVA-MPSoCs. Hierbei ist immer ein Kompromiss bezüglich Ausführungsgeschwindigkeit und zeitlicher Genauigkeit der Simulation zu berücksichtigen (siehe Kapitel 3.3).

1.3 Gliederung der Arbeit

In Kapitel 2 werden mögliche Systemarchitekturen und Teilkomponenten von eng gekoppelten Multiprozessorsystemen vorgestellt. Zudem wird ein Überblick über den Stand der Technik von eingebetteten Multiprozessoren gegeben. Als Prozessorkern wird die im Rahmen dieser Arbeit verwendete CoreVA-CPU eingeführt. Zudem werden Grundlagen, verbreitete Standards und Implementierungen von On-Chip-Verbindungsstrukturen vorgestellt. Anschließend wird die Systemarchitektur des CPU-Clusters des CoreVA-MPSoCs präsentiert sowie ein Überblick über Grundlagen und Implementierungen von Synchronisierungsverfahren gegeben.

In Kapitel 3 wird die im Rahmen dieser Arbeit entstandene Hardware- und Software-Entwurfsumgebung beschrieben. Als Zieltechnologie für den Hardwareentwurf wird eine 28-nm-FD-SOI[13]-Standardzellenbibliothek verwendet. Ein wichtiger Bestandteil der Software-Entwurfsumgebung sind Compiler für die Sprachen C, StreamIt sowie OpenCL. Zudem wird in Kapitel 3 die Simulations- und Emulationsumgebung des CoreVA-MPSoCs vorgestellt. Diese erlaubt das Ausführen von Anwendungen auf verschiedenen Abstraktionsebenen, die unterschiedliche Geschwindigkeiten und Genauig-

[12] Modulare Compilerarchitektur, früher *Low Level Virtual Machine*

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

keiten aufweisen. Insbesondere wird auf den Instruktionssatz-Simulator sowie zwei Plattformen zur FPGA-Emulation des CoreVA-MPSoCs eingegangen.

In Kapitel 4 werden verschiedene Bewertungsmaße, unter anderem Fläche, Energiebedarf sowie Performanz bzw. Rechenleistung, von digitalen mikroelektronischen Systemen eingeführt. Um in einer frühen Entwurfsphase ein MPSoC bewerten zu können, werden zudem analytische Modelle der Hardware- als auch der Software-Komponenten des CoreVA-MPSoCs vorgestellt. Zudem wird eine abstrakte Beschreibung des CoreVA-MPSoCs eingeführt, welche die Realisierung und Bewertung verschiedener MPSoC-Konfigurationen deutlich vereinfacht.

In Kapitel 5 wird eine detaillierte Entwurfsraumexploration des CPU-Clusters des CoreVA-MPSoCs durchgeführt. Die Anwendungsperformanz wird durch die Ausführung von Streaming-Anwendungen bestimmt, die mithilfe des CoreVA-MPSoC-Compilers auf die verschiedenen Konfigurationen des CPU-Clusters abgebildet werden. Hierzu werden in einem ersten Schritt verschiedene Konfigurationen der CoreVA-CPU bezüglich Anwendungsperformanz sowie Flächen- und Energiebedarf verglichen. Es wird eine CPU-Konfiguration mit zwei VLIW-Slots ausgewählt, die für die Untersuchungen auf Ebene des CPU-Clusters im Folgenden verwendet wird. Die Größe des Software-Entwurfsraums wird am Beispiel verschiedener Partitionierungen einer C-basierten Implementierung der Anwendung Matrixmultiplikation veranschaulicht. Anschließend werden verschiedene Busstandards (AXI und Wishbone) sowie Topologien (geteilter Bus sowie volle und partielle Crossbar) der Cluster-Verbindungsstruktur betrachtet. Zudem wird untersucht, wie sich die Integration von zwei Ausprägungen von Registerstufen auf die maximale Taktfrequenz und den Ressourcenbedarf der Verbindungsstruktur auswirkt. Anschließend werden gemeinsame L1- und L2-Datenspeicher in den CPU-Cluster integriert sowie der Energiebedarf von Kommunikation im CPU-Cluster betrachtet. Die Verwendung eines Instruktionscaches ermöglicht die Ausführung von Anwendungen im CPU-Cluster, deren Programmcode zu groß für die L1-Instruktionsspeicher der einzelnen CPUs ist. Neben der automatisierten Partitionierung von Streaming-Anwendungen werden zudem exemplarisch Anwendungen in der Sprache OpenCL auf das CoreVA-MPSoC abgebildet.

In Kapitel 6 werden ausgewählte Konfigurationen des CoreVA-MPSoC-CPU-Clusters prototypisch implementiert. Hierbei wird sowohl eine 28-nm-FD-SOI-Standardzellenbibliothek verwendet als auch die beiden FPGA-basierten Plattformen AMiRo und RAPTOR betrachtet. Abschließend wird in Kapitel 7 die vorliegende Arbeit zusammengefasst und ein Ausblick auf mögliche zukünftige Arbeiten gegeben.

2 Systemarchitektur eines eng gekoppelten Prozessorclusters

In diesem Kapitel werden mögliche Systemarchitekturen und Teilkomponenten von eng gekoppelten Multiprozessorsystemen vorgestellt. Flynn [61] teilt im Jahr 1966 Prozessorarchitekturen in vier Kategorien für die parallele Ausführung von Programmen ein. Basis ist die Aufteilung von Instruktionen und Daten auf verschiedene Verarbeitungseinheiten [90, S. 197]:

- SISD (*Single Instruction Single Data*)
- SIMD (*Single Instruction Multiple Data*)
- MISD (*Multiple Instruction Single Data*)
- MIMD (*Multiple Instruction Multiple Data*)

SISD entspricht einem Einprozessorsystem mit einem Instruktionsstrom. Auch superskalare Prozessoren gehören zu dieser Kategorie, obwohl sie mehrere parallele Verarbeitungseinheiten aufweisen. Der Instruktionsstrom wird erst zur Laufzeit von einer dedizierten Hardwareeinheit auf die verschiedenen Verarbeitungseinheiten aufgeteilt [18, S. 261]. MISD wird bisher nicht eingesetzt.

SIMD-Architekturen wenden einen einzelnen Instruktionsstrom parallel auf mehrere Datenströme an. Instruktionsspeicher und die Einheit zur Instruktionsdekodierung sind nur einmal vorhanden, Datenspeicher und Ausführungseinheiten mehrfach [90, S. 197]. Damit eine Anwendung von SIMD profitieren kann, muss sie Parallelität auf Datenebene (DLP) aufweisen.

MIMD-Architekturen verarbeiten parallel mehrere Datenströme mit unterschiedlichen Instruktionen [180, S. 13]. Die Ausführung der unterschiedlichen Instruktionsströme kann innerhalb eines Prozessors oder in einem Multiprozessorsystem erfolgen. Bei VLIW-Architekturen (siehe Abschnitt 2.2) dekodiert ein Prozessor mehrere Instruktionen in einem Taktzyklus und weist mehrere parallele Ausführungseinheiten auf. Hierbei wird die Parallelität auf Instruktionsebene (ILP) ausgenutzt. Multiprozessor-MIMD-Architekturen nutzen Parallelität auf Programmebene (TLP) aus. Jeder Prozessor besitzt typischerweise einen eigenen Daten- und Instruktionsspeicher bzw. Cache. Die CPUs kommunizieren über Nachrichten oder einen gemeinsamen Speicher. Multiprozessoren besitzen in der Regel mehrere Ebenen an Speichern, die sich in Größe, Übertragungsbandbreite und Latenz unterscheiden. In Abschnitt 2.3 werden verschiedene Speicherarchitekturen beschrieben und auf eine mögliche Verwendung in eingebetteten Multiprozessorsystemen hin untersucht.

CPUs und Speicher sind innerhalb eines MPSoCs über eine Verbindungsstruktur miteinander verbunden. Diese ermöglicht eine Kommunikation zwischen den einzelnen Komponenten. Eine effiziente Kopplung aller Komponenten eines MPSoCs ist erforderlich, um eine hohe Effizienz des Gesamtsystems realisieren zu können [153, S. 3 f.]. In Abschnitt 2.4 werden Verbindungsstrukturen eingeführt sowie verbreitete Standards und Implementierungen von On-Chip-Verbindungsstrukturen vorgestellt und verglichen. Die Architektur des CPU-Clusters des CoreVA-MPSoCs wird in Abschnitt 2.5 beschreiben. In Abschnitt 2.6 werden Grundlagen zur Synchronisierung verschiedener Prozessoren und Anwendungsteilen in Multiprozessorsystemen präsentiert. Zudem werden verschiedene im Rahmen dieser Arbeit entstandene Synchronisierungsverfahren vorgestellt. In Abschnitt 2.7 werden die Ergebnisse dieses Kapitels zusammengefasst.

2.1 Stand der Technik eingebetteter Multiprozessoren

Massiv-parallele Multiprozessoren können anhand ihrer Rechenleistung und Leistungsaufnahme klassifiziert werden. Die höchste Rechenleistung ist in Prozessoren für Großrechner (*Mainframes*) zu finden. MPSoCs für mobile und eingebettete Systeme besitzen eine um eine Größenordnung geringere Leistungsaufnahme, die Rechenleistung ist jedoch ebenfalls entsprechend vermindert. Für die Bestimmung der Rechenleistung der Multiprozessoren wird angenommen, dass jede Funktionseinheit mit ihrer maximalen Taktrate Operationen ausführt. Die Rechenleistung ist in Milliarden (Ganzzahl-) Operationen pro Sekunde (GOPS[1]) sowie Milliarden Fließkomma-Operationen pro Sekunde (GFLOPS[2]) angegeben. Hierbei wird die parallele Ausführung von Operationen durch SIMD- und VLIW-Rechenwerke berücksichtigt.

2.1.1 Mainframe-MPSoCs

Die Leistungsaufnahme von Mainframe-Prozessoren beträgt mehrere hundert Watt und die Kosten pro Chip können mehrere zehntausend Euro betragen. Aus diesem Grund sind Mainframe-CPUs nicht für mobile, eingebettete Systeme geeignet. Zum Vergleich werden im Folgenden einige technische Daten von zwei typischen Mainframe-CPUs angegeben.

Oracle Sparc M7

Der Oracle Sparc M7 [84; 122] besitzt 32 Sparc V9 CPU-Kerne mit 16 Pipelinestufen. Eine CPU erreicht eine Taktfrequenz von bis zu 3,8 GHz und kann bis zu acht Threads parallel ausführen (*Simultaneous Multithreading*, SMT). Jeweils vier CPUs sind in einem CPU-Cluster mit 8 MB[3] L3-Speicher zusammengefasst. Die Cluster sind über ein On-

[1] *Giga Operations Per Second*
[2] *Giga Floating Point Operations Per Second*
[3] In der vorliegenden Arbeit wird das Einheitenkürzel B für Byte und bit für Bit verwendet

Chip-Netzwerk verbunden. Der Prozessor wird in einer 20-nm-Technologie hergestellt und belegt eine Fläche von 700 mm^2.

IBM Power8

Der Power8 Prozessor [60; 75] von IBM besitzt 12 CPU-Kerne sowie 8 MB L3-Cache. Über einen Ringbus mit insgesamt 16 Segmenten kann jede CPU auf den L3-Cache der anderen CPUs zugreifen. Jede CPU kann bis zu acht Threads parallel ausführen (SMT) und besitzt insgesamt 16 parallele Ausführungseinheiten [184]. Hergestellt in einer 22-nm-Technologie erreicht der Power8 eine Taktfrequenz von 4,6 GHz bei einer Leistungsaufnahme von 250 W und einer Fläche von 650 mm^2. Die Rechenleistung von Systemen mit IBM Power8 und Oracle Sparc M7 kann durch die cachekohärente Verschaltung mehrerer Prozessoren weiter gesteigert werden.

2.1.2 MPSoCs für mobile und eingebettete Systeme

Im Gegensatz zu Mainframe-CPUs ist die Leistungsaufnahme von Multiprozessoren für mobile und eingebettete Systeme um eine Größenordnung geringer. Je nach Einsatzgebiet ist eine maximale Leistungsaufnahme von ungefähr 1 W bis 20 W erlaubt. Im Folgenden werden verschiedene kommerzielle und akademische eingebettete Multiprozessoren bzw. Many-Cores vorgestellt und abschließend untereinander und mit dem CoreVA-MPSoC verglichen.

SpiNNaker

Das SpiNNaker[4]-Projekt der Universität Manchester [150; 151; 159] hat das Ziel, große neuronale Netze in Echtzeit zu simulieren. Hierzu wurde ein massiv paralleler MPSoC entworfen, der 18 ARM968-Prozessoren enthält. Die CPU kann eine Operation pro Takt ausführen und besitzt keine Fließkomma-Einheit. Jede CPU verfügt über 32 kB lokalen Instruktions- sowie 64 kB lokalen Datenspeicher. Bis zu 65536 Chips können über ein asynchrones Netzwerk verbunden werden. Hierzu besitzt jedes MPSoC sechs bidirektionale IO-Ports sowie einen zentralen Paket-Router. An diesen Router sind neben den IO-Ports auch alle CPUs des MPSoCs angeschlossen. Ein Kommunikations-Controller entlastet die CPUs beim Versenden von NoC-Paketen. Zusätzlich verfügt SpiNNaker über ein System-NoC von Silistix, an das jede CPU über eine AXI-Schnittstelle angeschlossen ist. Über dieses NoC können beispielsweise 32 kB interner Speicher, 128 MB externer Speicher sowie eine Ethernet-Schnittstelle angesprochen werden. Da der SpiNNaker-MPSoC in einer 130-nm-Technologie gefertigt wird, beträgt die maximale Taktfrequenz 180 MHz bei einer Leistungsaufnahme von 1 W.

Plurality HAL

Der Plurality HAL (*HyperCore Architecture Line*) [161] ist eine massiv-parallele Multiprozessor-Architektur, die bis zu 256 CPU-Kerne an gemeinsame L1-Caches koppelt [198].

[4] *Spiking Neural Network Architecture*

Jede CPU hat eine direkte Anbindung an den Daten- und den Instruktions-Cache mit geringer Latenz. Die Architektur dieser Anbindung ist nicht veröffentlicht, allerdings wurde ein Patent beantragt [21]. Die skalare CPU basiert auf der SPARC V8-Architektur und besitzt fünf Pipelinestufen. Mehrere CPUs teilen sich eine Fließkomma-Einheit. Es stehen 32 32-bit-Register zur Verfügung. Die HAL-Architektur besitzt einen Hardware-Scheduler, um den einzelnen CPU-Kernen Aufgaben zuzuweisen. In [198] ist ein HAL-basierter MPSoC mit 64 CPUs beschrieben. Der Instruktionscache besitzt eine Größe von 128 kB, der Datencache ist 2 MB groß. In einer 40-nm-Technologie enthält der Entwurf auf einer Fläche von 25 mm^2 250 Millionen Transistoren. Die exakte maximale Taktfrequenz ist nicht veröffentlicht, soll jedoch nach [198] 350 MHz bis 500 MHz betragen.

GigaNetIC

Der hierarchische Multiprozessor GigaNetIC [25; 143; 144] ist eine Entwicklung des Fachgebiets Schaltungstechnik der Universität Paderborn. Der skalare N-Core-Prozessor besitzt eine Von-Neumann-Speicherarchitektur mit einem gemeinsamen Daten- und Instruktionsspeicher der Größe 32 kB. Ein CPU-Cluster besteht aus vier N-Core CPUs. Zudem besitzt jeder Cluster eine Schnittstelle zum GigaNoC-On-Chip-Netzwerk [166] und 32 kB Paketspeicher. Ein MPSoC mit acht Clustern und 32 CPUs hat einen Flächenbedarf von 43,7 mm^2 in einer 90-nm-Technologie [143, S. 55]. Die maximale Taktfrequenz beträgt 285 MHz bei einer Leistungsaufnahme von 1,8 W.

Tomahawk2

Der an der Technischen Universität Dresden entwickelte Tomahawk2 [145] ist ein heterogenes MPSoC für SDR-Anwendungen, insbesondere für die Basisbandverarbeitung von 4x4-MIMO[5]-LTE[6]. Das MPSoC enthält zwei CPU-Cluster mit jeweils vier Mehrzweck-CPUs (Duo-PE) sowie zwei weitere Cluster mit IO-Schnittstellen, einem Speichercontroller sowie anwendungsspezifischen CPUs. Eine Duo-PE besteht aus einer Tensilica LX4 RISC[7]-CPU für Fließkommaberechnungen sowie einem Vektorprozessor mit vierfach-SIMD für Ganzzahlberechnungen. Beide Rechenkerne teilen sich einen lokalen Datenspeicher. Als Anwendungsprozessor und für die Steuerung und Konfiguration des MPSoC ist eine Tensilica 570T RISC-CPU mit jeweils 16 kB Instruktions- und Datencache integriert. Die Verbindungsstruktur des Tomahawk2 besteht aus einem NoC mit vier Knoten. Jeder Knoten bindet bis zu vier Systemkomponenten an. Die Knoten sind teilweise über serielle Hochgeschwindigkeitslinks verbunden. Die Partitionierung von Anwendungsteilen auf die verschiedenen CPUs des MPSoCs kann in Software auf dem Anwendungsprozessor oder mittels einer dedizierten Hardwareeinheit (CoreManager) realisiert werden. Zur Reduzierung der dynamischen Verlustleistung erlaubt der Tomahawk2 die dynamische Anpassung von Taktfrequenz und Versorgungsspannung

[5] *Multiple Input Multiple Output*
[6] *Long Term Evolution*
[7] *Reduced Instruction Set Computer*

der einzelnen Systemkomponenten. Hergestellt in einer 65-nm-Technologie beträgt die Chipfläche 36 mm². Die maximale Taktfrequenz der Systemkomponenten beträgt 445 MHz bis 500 MHz. Tomahawk2 erreicht eine Spitzenrechenleistung bezüglich Ganzzahloperationen von 105 GOPS bei 1,39 W Leistungsaufnahme. Die maximale Fließkommarechenleistung beträgt 3,8 GFLOPS. Bei der Dekodierung eines LTE-Datenstroms mit 60 Mbit/s und 4x4-MIMO beträgt die Leistungsaufnahme 480 mW.

Kalray MPPA-256

Der Kalray MPPA-256 [50; 51] ist ein kommerzieller, hierarchischer Multiprozessor mit 288 CPU-Kernen. Das System besteht aus 16 Rechen- sowie vier I/O[8]-Clustern, die über ein zweilagiges (*2-Layer*) NoC mit 2D-Torus-Topologie verbunden sind. Jeder Rechencluster besitzt 16 Rechen-CPUs sowie eine CPU für Verwaltungsaufgaben, einen DMA-Controller und 2 MB geteilten L2-Speicher mit 16 Bänken [109]. Die CPUs verwenden eine VLIW-Architektur mit bis zu acht Pipelinestufen. ALU[9]-Operationen besitzen eine Latenz von fünf Takten, Fließkomma-Operationen eine Latenz von acht Takten. Es können bis zu fünf Instruktionen parallel ausgeführt werden, darunter eine 64-bit- oder zwei 32-bit-Fließkomma-Operationen. Die CPU greift auf jeweils 8 kB lokalen L1-Instruktions- und Daten-Cache zu. Jeder Cluster besitzt einen eigenen Adressraum. Die einzelnen Cluster kommunizieren über NoC-Pakete miteinander. Die CPUs der IO-Cluster können Linux als Betriebssystem ausführen und besitzen ebenfalls jeweils 2 MB L2-Speicher. Linux kann vom Programmierer verwendet werden, um Prozesse auf den einzelnen Rechenclustern zu starten und zu überwachen. In einer 28-nm-Technologie benötigt der MPPA-256 16 W bei einer Taktfrequenz von 600 MHz sowie 24 W bei 800 MHz [109]. Flächenangaben sind nicht verfügbar, allerdings wird in [107] angegeben, dass der gesamte Chip 2,8 Milliarden Transistoren besitzt.

Adapteva Epiphany E64G401

Adapteva Epiphany E64G401 [54; 71; 72] ist ein 64-CPU-Multiprozessor mit insgesamt 2 MB On-Chip-Speicher. Jede CPU besitzt 32 kB lokalen Scratchpad-Speicher, der vollständig in Software verwaltet wird. Bei einer maximalen Taktfrequenz von 800 MHz beträgt die Leistungsaufnahme in einer 28-nm-Technologie 2 W. Die CPU besitzt eine skalare Architektur und ist für die Verarbeitung von Fließkommazahlen mit einfacher Genauigkeit optimiert. Pro Takt können eine Ganzzahl- und eine Fließkomma-Operation ausgeführt werden. Dies resultiert in einer theoretischen Rechenleistung von 102,4 GOPS und 51,2 GFLOPS. Als Verbindungsstruktur wird ein dreilagiges 2D-Mesh-NoC verwendet, CPU-Cluster sind nicht vorhanden. Der MPSoC kann in C, C++ sowie OpenCL programmiert werden. Adapteva gibt an, dass die Architektur des MPSoCs die Integration von bis zu 4096 CPUs erlaubt [3].

Zum Zeitpunkt der Entstehung dieser Arbeit ist ausschließlich der 16-CPU-Multiprozessor E16G301 [53] kommerziell verfügbar, der in einer 65-nm-Technologie gefertigt

[8] *Input/Output*
[9] *Arithmetic Logical Unit*

ist. Bei einer maximalen Taktfrequenz von 1 GHz beträgt die Leistungsaufnahme 1,5 W. Der Flächenbedarf ist 9 mm^2 [3].

STMicroelectronics STHORM

Der STMicroelectronics STHORM [22; 135; 156] besitzt vier CPU-Cluster, die jeweils 17 CPU-Kerne und eng gekoppelten gemeinsamen Speicher (siehe Abschnitt 2.3.2) enthalten. Alternativ zum Produktnamen STHORM werden auch die Begriffe „Platform 2012“ und P2012 für das MPSoC verwendet. Die CPU-Kerne basieren auf dem STxP70-V4. Dieser besitzt sieben Pipelinestufen, 32 32-bit-Register und kann als superskalare Architektur bis zu zwei Instruktionen pro Takt parallel ausführen. Ein Befehl kann hierbei eine Fließkomma-Operation sein. Jede CPU besitzt einen privaten, direkt abgebildeten Instruktionscache mit einer Größe von 16 kB. 16 CPUs des Clusters können auf einen eng gekoppelten gemeinsamen L1-Speicher zugreifen, der eine Größe von 128 kB sowie 32 Speicherbänke aufweist. Jeder Cluster verfügt zusätzlich über eine CPU für Kontrollaufgaben, die einen separaten 32 kB großen Datenspeicher besitzt. Zudem enthält jeder Cluster einen zweikanaligen DMA-Controller sowie eine Schnittstelle für Hardwarebeschleuniger. Die Cluster sind über ein asynchrones NoC verbunden. Der STHORM kann in OpenCL sowie mit einem proprietären *Native Programming Model* programmiert werden. In [191] wird ein Hardware-Beschleuniger für die Synchronisierung innerhalb eines Clusters im STHORM vorgestellt. Der Beschleuniger verwendet atomare Zähler, um dem Programmierer eine flexible und effiziente Synchronisierungsprimitive zur Verfügung zu stellen. Ein STHORM besitzt in einer 28-nm-Technologie einen Flächenbedarf von 26 mm^2 für den gesamten MPSoC sowie 15,2 mm^2 für die vier CPU-Cluster mit insgesamt 68 CPUs, wobei vier CPUs für Kontrollaufgaben vorgesehen sind [22]. Die maximale Taktfrequenz ist 600 MHz bei einer Leistungsaufnahme von 2 W. Die theoretische maximale Rechenleistung beträgt 76 GOPS und 38 GFLOPS.

ARM Cortex-A7 und Cortex-A15

Prozessoren der Cortex-A-Familie von ARM werden in einer Vielzahl von mobilen und eingebetteten Systemen eingesetzt. Cortex-A-CPUs sind zum Zeitpunkt der Entstehung dieser Arbeit der De-facto-Standard beispielsweise in Smartphones und Tablet-PCs. Die Hälfte der bisher vorgestellten MPSoCs sind in einer 28-nm-Technologie gefertigt, die weiteren MPSoCs besitzen eine Strukturgröße von 40 nm bis 130 nm. Aus diesem Grund werden für einen Vergleich die CPUs Cortex-A7 und Cortex-A15 verwendet, die typischerweise in einer 28-nm-Technologie gefertigt werden. Jeweils bis zu vier dieser CPUs können zusammen mit einem gemeinsamen L2-Speicher in einem CPU-Cluster gekoppelt werden.

Die Cortex-A15-CPU ist auf eine hohe Leistungsfähigkeit hin optimiert und kann im Samsung Exynos 5450-MPSoC mit bis zu 1,6 GHz betrieben werden [74]. Pro CPU-Takt werden bis zu drei Instruktionen geladen und in Mikrooperationen (*Microops*) aufgeteilt. Die CPU verfügt über acht parallele Ausführungseinheiten, unter anderem ein Ganzzahl-Multiplizierer und zwei Fließkomma-Einheiten [113]. Zwei SIMD-Einheiten ermöglichen die Ausführung von bis zu acht 32-bit-Operationen pro Takt [197]. Es stehen dedizierte Einheiten für das Laden und Speichern von Daten zur Verfügung, die jeweils eine maximale Datenbreite von 128 bit besitzen. Die Pipeline besteht aus 15 bis 24 Stufen (je nach Funktionseinheit).

Die Cortex-A7-CPU ist im Vergleich zur Cortex-A15-CPU auf eine bessere Flächen- und Energieeffizienz hin optimiert [113]. Dies führt jedoch zu einer verminderten Rechenleistung. Es können bis zu zwei Instruktionen pro Takt geladen werden. Insgesamt stehen fünf parallele Ausführungseinheiten zur Verfügung, darunter eine SIMD/Fließkomma-Einheit. Die Ganzzahl-Pipeline besitzt acht Stufen. Die maximale Taktfrequenz im Samsung Exynos 5450-MPSoC beträgt 1,2 GHz.

Der Exynos 5450-MPSoC besitzt jeweils vier Cortex-A7-CPU und Cortex-A15-CPUs. Der Flächenbedarf der CPU-Cluster beträgt 19 mm^2 (Cortex-A15, inklusive 2 MB L2-Cache) bzw. 3,8 mm^2 (Cortex-A7, inklusive 512 kB L2-Cache). In [74] wird nur die Leistungsaufnahme der einzelnen CPUs (ohne L2-Cache) angegeben. Hieraus ergibt sich eine maximale Leistungsaufnahme von mindestens 4,2 W für den Cortex-A15-Cluster sowie 0,85 W für den Cortex-A7-Cluster.

Eine Bestimmung der Leistungsfähigkeit der ARM-CPUs ist nicht einfach, da nicht alle Ausführungseinheiten gleichzeitig mit Instruktionen versorgt werden können. Unter der Annahme, dass eine Cortex-A15-CPU neun Operationen pro Takt ausführen kann (zwei 128-bit-SIMD-Instruktionen, eine 32-bit-Ganzzahl-Operation), ergibt sich für die vier CPUs des Exynos 5450 eine Leistungsfähigkeit von 57,6 GOPS. Die Fließkomma-Rechenleistung beträgt bis zu 51,2 GFLOPS, da bis zu acht Fließkomma-SIMD-Operationen pro Takt ausgeführt werden können. Analog hierzu ergibt sich für einen Vierfach-Cluster aus Cortex-A7-CPUs eine Rechenleistung von 14,4 GOPS (eine 64-bit-SIMD-Instruktion, eine 32-bit-Ganzzahl-Operation) bzw. 9,6 GFLOPS.

Gegenüberstellung eingebetteter Multiprozessoren

In diesem Abschnitt wird die Architektur (siehe Tabelle 2.1) und der Ressourcenbedarf (siehe Tabelle 2.2) von ausgewählten eingebetteten Multiprozessoren gegenübergestellt. 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 zum Vergleich als hierarchisches MPSoC mit vier CPU-Clustern und insgesamt 64 CPUs aufgeführt. Die Werte hierfür stammen aus Kapitel 5 sowie [214; 223]. Insbesondere die Angaben für Chipfläche und Leistungsaufnahme sind als vorläufig zu betrachten, da zum Zeitpunkt der Entstehung dieser Arbeit die I/O-Schnittstelle des CoreVA-MPSoCs noch nicht vollständig spezifiziert ist.

Ein detaillierter Vergleich der betrachteten MPSoCs ist nicht sinnvoll, da die MPSoCs unterschiedlich viel Speicher (1,28 MB bis 44,5 MB) und eine unterschiedliche Anzahl und Konfigurationen an I/O-Schnittstellen besitzen. Zudem sind die MPSoCs in verschiedenen Strukturgrößen gefertigt bzw. für diese entworfen. Ein direkter Vergleich

Tabelle 2.1: 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
Plurality HAL	Skalar	64	64	gem. L1-Speicher	2,13	40
GigaNetIC	Skalar	32	4	NoC + get. Bus	1,28	90
Tomahawk2	Heterogen	20	2 – 8	NoC	2,13	65
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	16	NoC + Crossbar	2	28
Quad ARM Cortex-A7	5-fach	–	4	Crossbar	0,77	28
Quad ARM Cortex-A15	8-fach	–	4	Crossbar	2,3	28

bezüglich Flächenbedarf, Leistungsaufnahme und maximale Taktfrequenz ist daher nur schwer möglich. Es existieren Faustformeln für die Skalierung auf eine gemeinsame Strukturgröße, die jedoch eine sehr hohe Ungenauigkeit aufweisen. 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.

Tabelle 2.2: Ressourcenbedarf von eingebetteten Multiprozessoren

Name	F_{max} [MHz]	Chipfläche [mm^2]	Leistungsaufnahme [W]	Rechenleistung [GOPS]	Rechenleistung [GFLOPS]	Energieeffizienz [GOPS pro W]	Flächeneffizienz [GOPS pro mm^2]
SpiNNaker	180	102	1	3,24	–	3,2	0,03
Plurality HAL	350	25	k.A.	k.A.	k.A.	k.A.	k.A.
GigaNetIC	285	43,7	1,8	9,1	–	5,1	0,2
Tomahawk2	500	36	1,4[10]	105	3,8	75,5	2,9
Kalray MPPA-256	600	k.A.	16	864[11]	345,6[11]	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	800	12	1,8	102	–	56,7	8,5

Tabelle 2.3: Ressourcenbedarf von CPU-Clustern für Multiprozessoren in verschiedenen 28-nm-Technologien

Name	F_{max} [MHz]	On-Chip-Speicher [MB]	Chipfläche [mm^2]	Leistungsaufnahme [W]	Rechenleistung [GOPS]	Rechenleistung [GFLOPS]	Energieeffizienz [GOPS pro W]	Flächeneffizienz [GOPS pro mm^2]
Kalray MPPA-256	600	2,3	k.A.	< 1	48[11]	19,2[11]	> 48	k.A.
Adapteva Epiphany	800	0,5	2	0,5	25,6	12,8	51,2	12,8
STM STHORM	600	0,5	3,8	0,5	19	9,5	40,8	5,4
CoreVA CPU-Cluster	800	0,5	2,6	0,44	25,6	–	58,2	9,9
Quad ARM Cortex-A7	1200	0,77	3,8	> 0,85	14,4	9,6	< 16,9	3,8
Quad ARM Cortex-A15	1600	2,30	19	> 4,20	57,6	51,2	< 13,7	3,0

Statt eines Vergleichs von MPSoC-ASICs wird daher an dieser Stelle ein Vergleich von CPU-Clustern durchgeführt, die als Teilkomponenten (also z. B. ohne I/O-Pads) eines MPSoCs bzw. Many-Cores verwendet werden können. In Tabelle 2.3 sind Kennzahlen verschiedener CPU-Cluster mit 16 CPUs bei der Verwendung einer 28-nm-Technologie dargestellt. Zum Vergleich ist zudem ein NoC-basierter Adapteva Epiphany [3] mit ebenfalls 16 CPUs sowie CPU-Cluster mit jeweils vier Cortex-A7- und Cortex-A15-CPUs aufgeführt. Die maximale Taktfrequenz der beiden ARM-basierten CPU-Cluster beträgt 1,2 GHz bzw. 1,6 GHz. Alle übrigen Multiprozessoren haben eine maximale Taktfrequenz zwischen 600 MHz und 800 MHz. Der Kalray MPPA-256 besitzt aufgrund der 5-fach-VLIW-CPUs sowie insgesamt 2,3 MB SRAM-Speicher eine deutlich höhere Leistungsfähigkeit im Vergleich zu den anderen Architekturen mit 16 CPU-Kernen. Flächenangaben sind für den Kalray MPPA-256 nicht verfügbar. Adapteva Epiphany, STM STHORM und der CPU-Cluster des CoreVA-MPSoCs verfügen über die gleiche Menge an SRAM-Speicher. Der Adapteva Epiphany sowie der CPU-Cluster des CoreVA-MPSoCs besitzen mit 2,0 mm^2 bzw. 2,6 mm^2 im Vergleich zum STM STHORM einen deutlich geringeren Flächenbedarf. Obwohl die beiden ARM-basierten Cluster nur über jeweils vier CPUs verfügen, weisen sie einen hohen Flächenbedarf von 19 mm^2 (Cortex-A15) bzw. 3,8 mm^2 (Cortex-A7) auf. Eine Kombination aus einer hohen maximalen Taktfrequenz und vielen parallelen Ausführungseinheiten führt indes zu einer Rechenleistung von 57,6 GOPS für den CPU-Cluster mit Cortex-A15-CPUs.

In Bezug auf die Flächeneffizienz (als Verhältnis von der Rechenleistung zur Chipfläche) und Energieeffizienz (Verhältnis Rechenleistung zu Verlustleistung) zeigen der CPU-Cluster des CoreVA-MPSoCs sowie der Adapteva Epiphany Vorteile gegenüber den

[10] Abgeschätzt, basierend auf [145]
[11] Abgeschätzt, basierend auf [109]

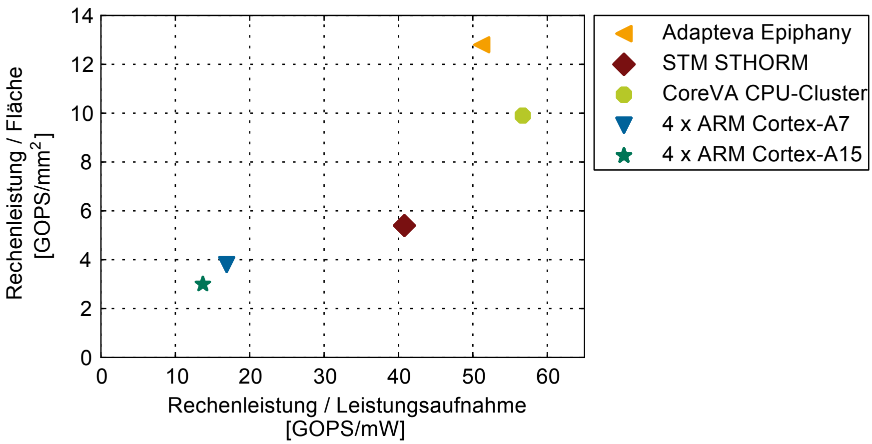


Abbildung 2.1: Verhältis Rechenleistung-Fläche sowie Rechenleistung-Leistungsaufnahme von CPU-Clustern in verschiedenen 28-nm-Technologien

anderen Architekturen (siehe Abbildung 2.1). Dies ist zum Teil jedoch auf die hohe Energieeffizienz der verwendeten FD-SOI-Technologie des CoreVA-MPSoCs zurückzuführen. Zudem verfügt die CoreVA-CPU im Vergleich zu den anderen Prozessoren über keine Fließkomma-Einheit. Hierdurch ergeben sich Vorteile in Bezug auf den Flächenbedarf.

Die beiden ARM-basierten CPU-Cluster besitzen eine deutlich schlechtere Flächen- und Energieeffizienz im Vergleich zu den anderen CPU-Clustern. Die Cortex-A15-CPU kann beispielsweise als superkalare Architektur Instruktionen zur Laufzeit umsortieren (*Out-of-Order-Execution*) sowie Register umbenennen (*Register-Renaming*) [197]. Dies erhöht die Leistungsfähigkeit der CPU, senkt jedoch die Flächen- und Energieeffizienz.

2.2 Der VLIW-Prozessor CoreVA

Als Recheneinheit wird in dieser Arbeit der CoreVA[12]-Prozessor verwendet [217; 99]. Die CoreVA-CPU wurde im Rahmen des MxMobile-Projektes [102] an der Universität Paderborn entwickelt und liegt als synthetisierbare RTL[13]-Beschreibung in der Hardwarebeschreibungssprache VHDL[14] vor [176, S. 379 ff.]. Einsatzgebiete sind energiebeschränkte eingebettete Systeme.

[12] *Configurable Ressource Efficient VLIW Architecture*
[13] *Register Transfer Level*
[14] *Very High Speed Integrated Circuit Hardware Description Language*

Die CoreVA-CPU ist eine konfigurierbare 32-bit-VLIW[15]-Architektur. Die Ausprägung und Anzahl an Ausführungseinheiten kann zur Entwurfszeit festgelegt werden. Durch die parallele Ausführung von Operationen kann (bei gleicher Taktfrequenz) die Leistungsfähigkeit der CPU gesteigert werden. Wie viele Operationen parallel ausgeführt werden können, hängt von der Parallelität auf Instruktionsebene (ILP) des ausgeführten Programms ab. Dem VLIW-Prinzip [59] folgend, wird die Aufteilung des Programmstroms auf die einzelnen Ausführungseinheiten vom Compiler vorgenommen (siehe Abschnitt 3.2.2).

Die Ausführungseinheiten sind in sogenannte *Slots* zusammengefasst. Jeder Slot kann pro Takt eine 32-bit-Instruktion ausführen. Alle in einem Takt ausgeführten Instruktionen werden als Instruktionsgruppe bezeichnet. Durch eine Instruktionskompression mittels *Stop*-Bit wird verhindert, dass Instruktionsgruppen durch Leerinstruktionen aufgefüllt werden müssen, falls eine geringe ILP die Verwendung aller Slots verhindert. Dies kann den Speicherbedarf einer Anwendung signifikant reduzieren. Der CoreVA-Prozessor besitzt getrennte Instruktions- und Datenspeicher (Harvard-Architektur). Die Registerbank (*Register-File*) 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 2.2). 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 (ALU[16]). Die ALU kann beispielsweise

[15] *Very Long Instruction Word*
[16] *Arithmetic Logical Unit*

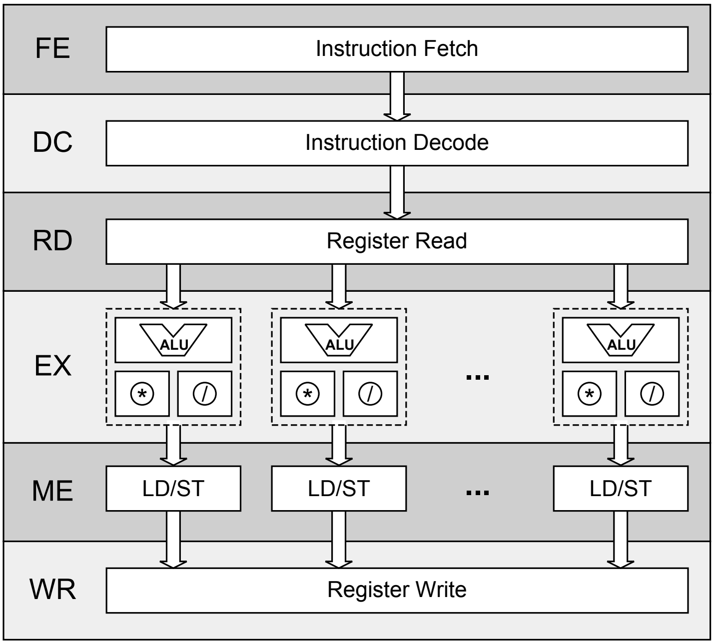


Abbildung 2.2: Blockschaltbild der Pipeline des CoreVA-Prozessors

Additionen, Subtraktionen oder Bit-Schiebeoperationen auf ganzen Zahlen ausführen. Optional kann ein Slot einen Multiplikations-Akkumulierer (MAC[17]), eine Dividiereinheit[18] (DIV) und eine Einheit für Zugriffe auf den Datenspeicher (LD/ST[19]) 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 glei-

[17] *Multiply Accumulate*
[18] Realisiert im Divisions-Schritt-Verfahren
[19] *Load/Store*

che 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.

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 [103] 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.

Die Anzahl der Schreib- und Leseports des Registerfiles sind von der Anzahl und Konfiguration der VLIW-Slots abhängig [217]. Eine ALU benötigt zwei Leseports für die Operanden und einen Schreibport für das Ergebnis von Berechnungen. Eine MAC-Einheit verwendet einen dritten Leseport für den Akkumulator. Ein zweiter Schreibport wird von der LD/ST-Einheit für aktualisierte Adressen bei *Pre-* und *Post-Modify*-Adressoperationen benötigt.

Mittels eines Adressdekoders kann die CPU auf Hardwarebeschleuniger und externe Schnittstellen zugreifen (siehe Abbildung 2.3). 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 (AES[20] und ECC[21]) [218; 222] sowie Prüfsummenberechnung (CRC[22]) [165] zur Verfügung. Die Integration einer Erweiterung beschleunigt den jeweiligen Algorithmus um bis zu mehrere Größenordnungen, bedingt jedoch einen teils deutlich höheren Ressourcenbedarf [99, S. 165 ff.]. Eine serielle Schnittstelle (UART[23]) ermöglicht die Kommunikation der CPU mit der Außenwelt. Die lokalen Daten- und Instruktionsspeicher besitzen jeweils eine maximale Größe von 1 MB. Alternativ können Daten- und Instruktionscaches integriert werden [174]. In Abschnitt 2.3 werden Grundlagen zu Speichern in eingebetteten Multiprozessoren behandelt sowie verschiedene Speicherarchitekturen vorgestellt.

In [234] wurde die CoreVA-CPU um einen Interrupt-Controller erweitert. Ein Interrupt-Controller erlaubt die Unterbrechung der momentan auf der CPU ausgeführten Anwendung [86, S. 103 f.]. Grund hierfür kann beispielsweise das Erreichen einer Zeitschranke oder das Eintreffen von Daten über eine externe Schnittstelle sein. Beim Auslösen eines Interrupts wird der *Program Counter* der CPU gespeichert und ein Sprung zur *Interrupt Service Routine* (ISR) ausgelöst. Hierzu werden alle Register auf dem Sta-

[20] *Advanced Encryption Standard*
[21] *Elliptic Curve Cryptography*
[22] *Cyclic Redundancy Check*
[23] *Universal Asynchronous Receiver Transmitter*

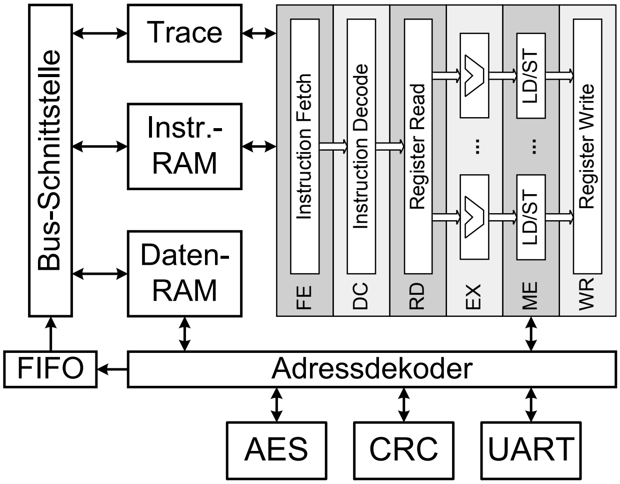


Abbildung 2.3: Blockschaltbild der CoreVA-CPU mit CPU-Pipeline, Instruktions- und Datenspeicher, Hardwareerweiterungen sowie einer Schnittstelle zum CPU-Cluster

pelspeicher (*Stack*) gesichert und abhängig von der Art der Unterbrechung wird eine entsprechende Funktion aufgerufen. Anschließend kann die unterbrochene Anwendung weiter ausgeführt werden, in dem die Register vom Stack zurückgesichert werden und zum entsprechenden Program-Counter gesprungen wird. Ein Interrupt-Controller ist Voraussetzung für den Einsatz der allermeisten Betriebssysteme. So wurde in [235] unter Verwendung des Interrupt-Controllers das Echtzeit-Betriebssystem ChibiOS RT auf die CoreVA-CPU portiert.

Im Rahmen dieser Arbeit ist eine Anbindung der CPU an den CPU-Cluster entstanden. Diese in Abbildung 2.3 als FIFO[24] dargestellte Komponente ermöglicht es der CPU, z. B. auf die lokalen Speicher anderer CPUs innerhalb eines Clusters zuzugreifen [238, S. 9 f.]. 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 SRAM[25] bestehen (siehe Abschnitt 5.2.2). 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 wurde im Rahmen dieser Arbeit

[24] *First-In First-Out*
[25] *Static Random Access Memory*

ein Kommunikationsmodell entwickelt, das (nahezu) vollständig auf Lesezugriffe im Cluster verzichtet (siehe Abschnitt 2.6.4).

Eine *Slave*-Schnittstelle (siehe Abschnitt 2.4) ermöglicht vom CPU-Cluster aus Zugriffe auf Instruktions- und Datenspeicher einer CPU. Zudem erlaubt 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 [99, S. 44 ff.; 101]. 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-ULP[26]) entstanden. Beide Prototypen integrieren jeweils 16 kB Daten- und Instruktionscache und besitzen eine Chip-Fläche von 2,64 mm^2. Der CoreVA-VLIW-ASIC [99, 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 [132; 134]. Zwei CPU-Kerne sind unter Verwendung einer für den Subschwell-Betrieb optimierten Standardzellen-Bibliothek entworfen. Diese Bibliothek erlaubt einen Betrieb der CPUs bei Spannungen von 200 mV bis 1,2 V. Die geringste Energie pro ausgeführtem 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 für den Subschwell-Betrieb optimierten SRAM-Speicher [133]. 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.

2.3 Speicher in eingebetteten Systemen

Halbleiterspeicher wird in eingebetteten Multiprozessoren zur Speicherung von Programmcode und Daten verwendet. Viele Architekturen setzen Speicher zur Kommunikation und Synchronisierung der CPUs untereinander ein (siehe Abschnitt 2.6). Der Anteil von Speicher an der Gesamtfläche von MPSoCs steigt kontinuierlich an [96].

In modernen Halbleitertechnologien sind vielfältige Typen und Ausprägungen von Speichern verfügbar. Die unterschiedlichen Speichertypen weisen verschiedene Eigenschaften bezüglich Geschwindigkeit sowie Flächen- und Energiebedarf auf und werden unterschiedlich eng an einen CPU-Kern gekoppelt [90, S. 288]. Um Anwendungen eine ausreichende Menge an Speicher mit (im Durchschnitt) hoher Geschwindigkeit zur Verfügung zu stellen, wird eine Hierarchie verschieden schneller Speicher verwendet

[26] *Ultra Low Power*

Tabelle 2.4: Speicherhierarchie mit typischen Werten für Speichergröße und Latenz (in CPU-Takten bei einer Frequenz von 1 GHz)

Hierarchie	Speichertyp	Typ. Größe	Typ. Latenz
CPU-Registerfile	Register	128 B	1
L1	SRAM	32 kB	2 – 8
L2	SRAM	128 kB	8 – 32
L3	SRAM/eDRAM	1 MB	32 – 64
Hauptspeicher	DRAM	8 GB	128 – 1024
nichtfl. Speicher	Flash	256 GB	8000
	HDD	6 TB	> 10000

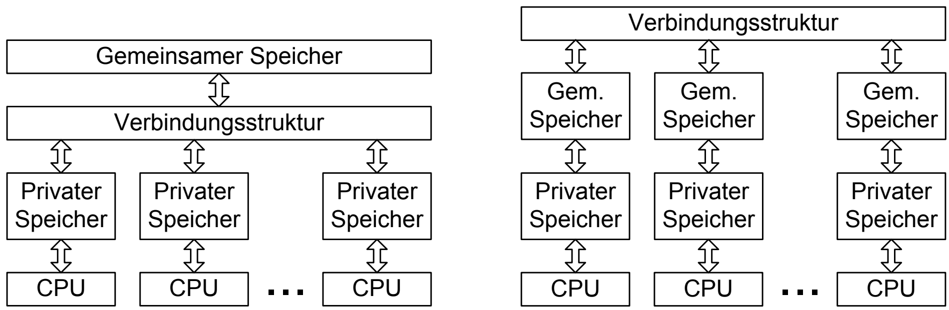
(siehe Tabelle 2.4) [111, S. 475]. Die verschiedenen Hierarchien werden bezogen auf ihre Entfernung zur CPU in Level eingeteilt.

Das Registerfile typischer eingebetteter Prozessorkerne fasst wenige hundert Byte, weist jedoch eine Latenz von einem Taktzyklus auf (siehe Abschnitt 2.2). Über Load/Store-Einheiten ist die CPU an Level-1-SRAM-Datenspeicher gekoppelt. Instruktionen werden aus einem Level-1-Instruktionsspeicher geladen. Die Verzögerungszeit eines SRAM-Speichers steigt mit der Größe an [137, S. 9]. Aus diesem Grund ist die direkt an die CPU gekoppelte Speichermenge begrenzt. Typische Größen von Level-1-Speicher liegen zwischen 4 kB und 64 kB bei einer Latenz von zwei bis vier CPU-Takten.

Benötigt eine Anwendung eine größere Speichermenge, kann zusätzlicher Speicher mit einer höheren Latenz verwendet werden. Dieser kann sich auf dem Chip befinden oder extern angebunden sein. In beiden Fällen kann sowohl SRAM als auch DRAM[27] verwendet werden [88; 98]. DRAM ist langsamer als SRAM, die Chipfläche pro Speicherzelle ist jedoch geringer [111, S. 546 ff.]. Die Latenz von DRAM ist zudem nicht deterministisch. Bei On-Chip-DRAM spricht man auch von *Embedded DRAM* (eDRAM) [98, S. 77]. On-Chip-Speicher höherer Hierarchie werden als Level-2, Level-3, etc. Speicher bezeichnet. Diese können exklusiv von einem Prozessorkern oder auch als geteilter Speicher von mehreren CPUs verwendet werden. Off-Chip Speicher wird Arbeits- oder Hauptspeicher genannt und besteht typischerweise aus DDR-SDRAM[28].

Alle bisher vorgestellten Speicher halten die gespeicherten Informationen nur, solange eine Versorgungsspannung anliegt. Bei Mikrocontrollern wird nichtflüchtiger Speicher (Flash) als On-Chip-Speicher eingesetzt. Größere eingebettete Systeme und Personalcomputer verfügen über nichtflüchtigen Off-Chip-Speicher. Flash-Speicher weist eine wiederum höhere Latenz als DRAM auf [111, S. 489 f.]. Festplatten werden aufgrund ihrer Stoßempfindlichkeit und Baugröße vor allem in Personalcomputern und Servern eingesetzt.

[27] *Dynamic Random Access Memory*
[28] *Double-Data-Rate Synchronous-DRAM*



(a) UMA-Architektur (b) NUMA-Architektur

Abbildung 2.4: Multiprozessorsysteme mit zentralem und verteiltem gemeinsamem Speicher

Architekturen mit gemeinsamem Speicher können bezüglich der Zugriffslatenzen der einzelnen CPUs auf den Speicher unterschieden werden. Wird ein zentraler gemeinsamer Speicher mit einheitlicher Zugriffszeit verwendet, spricht man von einem symmetrischem Multiprozessor (*Symmetric Multiprocessor*, SMP) bzw. einer Architektur mit einheitlichem Speicherzugriff (UMA[29], siehe Abbildung 2.4a) [90, S. 199]. Private Speicher oder Caches werden verwendet, um den Kommunikationsaufwand der einzelnen CPUs zu reduzieren, indem die räumliche und zeitliche Lokalität von Speicherzugriffen ausgenutzt wird.

SMP-Architekturen ermöglichen eine einfache Programmierung, bei Systemen mit einer großen Anzahl an CPUs wird der gemeinsame Speicher jedoch zunehmend zu einem Flaschenhals [205]. Architekturen mit verteiltem gemeinsamen Speicher (*Distributed Shared Memory*, DSM) besitzen mehrere Speicher im System, auf den die CPUs mit unterschiedlichen Latenzen zugreifen können (NUMA[30], siehe Abbildung 2.4b). Eine effiziente Programmierung von NUMA-Systemen erfordert einen höheren Aufwand im Vergleich zu UMA-Systemen. NUMA-Systeme weisen typischerweise eine deutlich höhere Speicherbandbreite bei gleichzeitig reduzierter Latenz[31] auf [18, S. 260]. Der verteilte Speicher kann entweder über einen gemeinsamen Adressraum von allen CPUs direkt angesprochen werden. Alternativ besitzt jede CPU einen eigenen Speicher mit privatem Adressraum. In diesem Fall erfolgt die Kommunikation über Nachrichten, die zwischen den CPUs ausgetauscht werden (siehe Abschnitt 2.6.4).

[29] *Uniform Memory Access*
[30] *Non-Uniform Memory Access*
[31] bei Zugriffen auf Speicher in der „Nähe" einer CPU

2.3.1 Lokaler Level-1-Speicher

Level-1-Speicher ist eng an die Prozessorpipeline gekoppelt und kann als Cache oder als eng gekoppelter Speicher (*Scratchpad-Memory*) realisiert sein [98, S. 80]. Die Größe und Architektur von L1-Instruktions- und Datenspeicher beeinflusst maßgeblich die Ressourceneffizienz und maximale Taktfrequenz einer CPU. Im Rahmen dieser Arbeit werden ausschließlich Harvard-Speicherarchitekturen mit getrenntem Daten- und Instruktionsspeicher betrachtet. Ein gemeinsamer L1-Speicher für Daten und Instruktionen (Von-Neumann-Architektur) wird in aktuellen Prozessorarchitekturen nicht verwendet.

Caches

Caches speichern Zugriffe auf externen DRAM-Speicher in lokalem SRAM-Speicher zwischen [137, S. 9]. Ziel ist es, den Einfluss der hohen DRAM-Latenz auf die Leistungsfähigkeit der CPU abzumildern. Hierbei muss, anders als beim Scratchpad-Speicher, der CPU die Existenz des Caches nicht bekannt sein. Die CPU kann wahlfrei auf den gesamten Adressbereich des externen Speichers zugreifen. Ein angefragtes Datenwort wird aus dem externen Speicher geladen, an die CPU weitergeleitet sowie im Cache zwischengespeichert. Bei darauffolgenden Zugriffen stellt der Cache das Datenwort der CPU ohne Zugriff auf den externen Speicher bereit (zeitliche Lokalität).

Die Wahrscheinlichkeit, dass nach einem Zugriff der CPU ein benachbartes Datenwort angefragt wird, ist hoch (räumliche Lokalität). Aus diesem Grund verwaltet ein Cache zusammenhängende Blöcke von Daten fester Größe, sogenannte Cachezeilen. Durch diese Mechanismen kann der Cache viele Anfragen der CPU direkt beantworten und die CPU wird weniger oft angehalten, um Daten nachzuladen. Ein Treffer im Cache benötigt zudem deutlich weniger Energie im Vergleich zu einem DRAM-Zugriff.

Zusätzlich zum Speicher, der die Cachezeilen zwischenspeichert, benötigt ein Cache einen weiteren Speicher für Metainformationen. Dieser *Tag*-Speicher enthält beispielsweise die Adresse im Hauptspeicher, aus der eine Cachezeile geladen worden ist. Diese Information ist für die Entscheidung notwendig, ob ein Zugriff vom Cache behandelt werden kann (Treffer, *Hit*) oder nicht (Fehlzugriff, *Miss*).

Algorithmen, insbesondere in der Signalverarbeitung, besitzen oft typische Zugriffsmuster auf Daten und Instruktionen, die jedoch deutlich komplexer als die beschriebene zeitliche und räumliche Lokalität sind. Jacob et al. [98, S. 64] bezeichnen diese komplexen Zugriffsmuster als algorithmische Lokalität. Caches können nur von algorithmischer Lokalität profitieren, wenn sie zusätzliche Informationen über den Zugriff erhalten. Dies kann statisch durch den Compiler geschehen (*Software-Prefetching*, siehe [104, S. 269]). Eine weitere Möglichkeit ist die Implementierung einer Vorhersagelogik in der Cache-Hardware (*Hardware-Prefetching*). Instruktionscaches können Informationen aus der Sprungvorhersage des CPU-Kerns verwenden. Datencaches können komplexe Heuristiken anwenden, um Datenzugriffe vorherzusagen. Dies ist vor allem bei superskalaren Prozessoren verbreitet. Wird eine Zeile spekulativ geladen, die anschließend nicht verwendet wird, erhöht dies die Auslastung des Busses der CPU und benötigt unnötig Energie.

Die Organisation und Verwaltung von Caches kann auf unterschiedliche Weise erfolgen [104, S. 10; 137, S. 15]. Bei vollassoziativen Caches kann jedes Datum aus dem Speicher in jeder Cachezeile liegen. Teil- oder satzassoziative Caches teilen den Speicher in Gruppen ein, jeder Gruppe sind mehrere Cachezeilen zugeordnet. Innerhalb einer Gruppe kann jedes Datenwort in jeder Cachezeile liegen. Direkt abgebildete Caches ordnen jeder Speicheradresse genau eine Cachezeile zu. Vollassoziative Caches benötigen einen Komparator für jede Cachezeile. Bei jedem Zugriff der CPU muss zudem jeder Tagspeicher und – je nach Architektur des Caches – jeder Datenspeicher gelesen werden.

Tritt ein Fehlzugriff auf, wird der entsprechende Datenblock aus dem externen Speicher in den Cache geladen. Bei voll- und teilassoziativen Caches kommen mehrere Cachezeilen infrage, in die der Datenblock geschrieben werden kann. Sind alle Zeilen gefüllt, muss entschieden werden, welche Zeile ersetzt wird. Hierzu sind in der Literatur eine Vielzahl an Ersetzungsstrategien für Cachezeilen bekannt [90; 95; 98; 104; 137; 143].

Schreibt die CPU Daten in den Cache, ist die entsprechende Cachezeile verschmutzt. Es gibt zwei Möglichkeiten, wie diese Daten in den Hauptspeicher übernommen werden können [137, S. 16]. Bei der *Write-Through*-Strategie werden die geänderten Daten umgehend in den Hauptspeicher geschrieben. Wird das Datum mehrmals nacheinander von der CPU aktualisiert, wird der Arbeitsspeicher durch unnötige Zugriffe belastet. Im *Write-Back*-Modus werden aktualisierte Daten der CPU erst in den Hauptspeicher geschrieben, wenn die Cachezeile aus dem Cache verdrängt wird. Hierfür muss der Tag-Speicher ein zusätzliches *Dirty*-Bit speichern. In Multiprozessorsystemen wird typischerweise der Write-Back-Modus verwendet, da sonst die On-Chip-Verbindungsstruktur durch eine große Zahl von Schreibzugriffen geringer Größe überlastet wird.

Mohammad [137, S. 17] unterscheidet zusätzlich zwischen parallelen und seriellen Zugriffen auf Daten- und Tagspeicher eines Caches. Bei einem parallelen Zugriff werden alle Daten- und Tagspeicher zeitgleich abgefragt. Zeigt die Ausgabe des Tagspeichers einen Treffer an, so werden die Ausgabedaten des Datenspeichers an die CPU angelegt. Bei einem Fehlzugriff ist die benötigte Energie für den Zugriff auf den Datenspeicher verschwendet. Vollassoziative und teilassoziative Architekturen erfordern eine Aktivierung von mehreren Speichern, was die Energie pro Cachezugriff deutlich steigert. Bei einem seriellen Zugriff wird zuerst der Tagspeicher gelesen. Ein Treffer führt dazu, dass im nächsten Takt die Daten aus dem entsprechenden Datenspeicher gelesen werden. Gegebenenfalls vorhandene weitere Datenspeicher werden nicht aktiviert. Ein Fehlzugriff führt zu einer Leseanfrage an den externen Speicher. Ein Cache mit seriellem Zugriff benötigt signifikant weniger Energie, erhöht jedoch die Latenz von Cachezugriffen. Dies bedeutet, dass die CPU zusätzliche Pipelinestufen für Zugriffe auf den Cache implementieren muss.

Für die Bewertung von Cache-Implementierungen werden, neben der Performanz der auf der CPU ausgeführten Anwendung, weitere Metriken verwendet. Das Verhältnis von Cache-Treffer und Fehlzugriffen wird als Trefferquote (*Hitrate*) bezeichnet. Zudem wird in der Literatur oft die Anzahl bzw. der Anteil von Wartetakten (*Stall*-Zyklen) aufgrund von Cache-Fehlzugriffen angegeben.

Eng gekoppelte Speicher

Es ist möglich, dass Anwendungen mit geringem Speicherbedarf ausschließlich eng gekoppelten Speicher verwenden. Alternativ können Scratchpad-Speicher zusätzlich zu Caches eingesetzt werden. Die Ausführungszeit der CPU kann hierdurch beschleunigt werden, indem beispielsweise der CPU-Stack oder Zwischenergebnisse im Scratchpad-Speicher abgelegt werden. Das Kopieren von Daten aus einem externen Speicher in den Scratchpad-Speicher kann durch die CPU erfolgen. Hierdurch wird die CPU allerdings für diese Zeit blockiert. Eine weitere Möglichkeit ist der Speicherdirektzugriff (*Direct Memory Access*, DMA). Hierbei kopiert ein sogenannter DMA-Controller Daten direkt vom externen in den Scratchpad-Speicher. Die CPU muss den DMA-Controller lediglich konfigurieren und den Transfer starten. Ist die Übertragung der Daten abgeschlossen, wird die CPU informiert und kann auf die Daten im Scratchpad-Speicher zugreifen. Da der Inhalt des Scratchpad-Speichers vollständig durch die CPU in Software verwaltet wird, kann auch von einem explizit verwaltetem Cache gesprochen werden [98, S. 80].

Stand der Technik L1-Instruktionscaches

L1-Instruktionscaches speichern Teile des Programmabbildes einer Anwendung zwischen und profitieren insbesondere von Schleifen in Programmablauf. Caches für Instruktionen sind sehr weit verbreitet und seit Jahrzehnten Gegenstand von Forschung und Entwicklung. Viele aktuelle Arbeiten im Kontext von eingebetteten CPUs zeichnen sich durch neue Architekturansätze aus, um Flächenbedarf, Trefferquote und/oder Energiebedarf von Instruktionscaches zu optimieren. Scratchpad-Speicher weisen eine hohe Ressourceneffizienz auf, sind jedoch in der Größe beschränkt und müssen daher durch die Anwendungssoftware verwaltet werden [202].

Einige Arbeiten ergänzen den L1-Instruktionscache um einen Scratchpad-Speicher, in welchen die Teile der Anwendung, auf die sehr häufig zugegriffen wird, abgelegt werden können. Li et al. [123] erweitern den Instruktionscache eines VLIW-DSPs um einen zusätzlichen Schleifenmodus. In diesem Modus wird der Tagspeicher des Instruktionscaches deaktiviert, bis die CPU die Schleife verlässt. Das Eintreten und Verlassen der Schleife wird durch spezielle Instruktionen im Programmcode angezeigt. Die Größe der Schleife darf die Größe des Caches nicht übersteigen. Der Energiebedarf des Instruktionscaches kann um bis zu 20 % reduziert werden, während der Flächenbedarf um 0,7 % steigt. Die Implementierung von Li et al. kann als ein zusätzlicher, temporärer Scratchpad-Modus für einen Instruktionscache betrachten werden.

Park et al. [152] präsentieren einen L0-Scratchpad-Speicher, der Speicheranfragen der CPU beantworten kann, ohne dass der L1-Instruktionscache aktiviert werden muss.

Ob eine Instruktions im L0-Speicher abgelegt wird, entscheidet der Compiler. Hierzu stellen Park et al. einen feingranularen Platzierungsalgorithmus vor, der festlegt, ob eine Instruktion im L0-Speicher abgelegt wird. Hierdurch kann der Energiebedarf im Vergleich zu einem Cache um 38 % reduziert werden.

Durch das Vorladen von Instruktionen (*Prefetching*) kann die Trefferquote (*Hitrate*) eines Caches deutlich gesteigert werden. Hsu und Smith [95] zeigen, dass das Prefetching früh genug gestartet werden muss, damit die entsprechende Cachezeile der CPU rechtzeitig zur Verfügung steht. Ist dies der Fall können bis zu 90 % aller CPU-Strafzyklen durch Fehlzugriffe des Instruktionscaches vermieden werden. Bei gleicher Cachezeilen-Größe vergrößert der Einsatz von Prefetching die Anzahl der Zugriffe auf den L2-Speicher und somit den Energiebedarf der CPU. Da Prefetching laut Hsu und Smith jedoch mit kleineren Zeilengrößen bessere Ergebnisse erzielt, sinken hierdurch sogar die Anzahl an L2-Zugriffen im Vergleich mit einem Cache ohne Prefetching und größeren Cachezeilen.

Bortolotti et al. [27] vergleichen einen CPU-Cluster mit einem gemeinsamen L1-Instruktionscache und einen Cluster mit lokalen L1-Caches. Führen die CPUs im Cluster die gleichen Funktionen aus, kann der gemeinsame L1-Cache die Performanz um bis zu 60 % steigern. Dies ist allerdings ein sehr spezieller Anwendungsfall, da die CPUs im Cluster hierzu in einem SIMD-ähnlichen Modus betrieben werden müssen. In diesem Fall könnte jedoch auch eine einzelne CPU mit einem entsprechenden SIMD-Rechenwerk verwendet werden. Hierbei muss jede Instruktion – im Gegensatz zur Architektur mit gemeinsamen L1-Instruktionscache – nur einmal dekodiert werden.

Gegenüberstellung lokaler L1-Datenspeicher und Datencaches

Viele kommerzielle CPUs besitzen L1-Datencaches, um die Anwendungsentwicklung und Wiederverwendbarkeit von Software zu vereinfachen. Scratchpad-Speicher kommen vor allem bei eingebetteten Prozessoren zum Einsatz. Hier wird einer hohen Ressourceneffizienz ein höherer Stellenwert gegenüber einer einfachen Programmierbarkeit eingeräumt. Ein weiteres Einsatzgebiet von Scratchpad-Speichern sind massiv parallele Multiprozessoren. Hier ist es bei der Verwendung von Caches erforderlich, Verzeichnisse zur Sicherstellung der Kohärenz zu führen. Diese Verzeichnisse benötigen ungefähr die Chipfläche der Tag-Speicher des L2-Caches [81]. Für die Programmierung von massiv parallelen Multiprozessoren werden zunehmend Sprachen wie OpenCL eingesetzt. Die Laufzeitumgebung von OpenCL kann alle Scratchpad-Speicher im System verwalten und so den Programmierer entlasten (siehe Abschnitt 3.2.4).

Niemann [143, S. 107] zeigt in seiner Dissertation, dass für eingebettete Multiprozessoren Datencaches im Allgemeinen und kohärente Caches im Besonderen einen hohen Mehrbedarf an Chipfläche und Energie bedeuten. Kohärenzprotokolle können spezielle Anwendungen stark beschleunigen, dies wird jedoch für eine Beispielarchitektur mit einem um den Faktor 3,3 erhöhten Flächenbedarf erkauft.

Banakar et al. [20] vergleichen Scratchpad-Speicher und Caches für rechenintensive Anwendungen. Es wird eine Analyseumgebung vorgestellt, mit der verschiedene

Größen und Partitionierungen des Datenspeichers eines eingebetteten Prozessors analysieren werden können. Hierzu wird die Energie pro Speicherzugriff bestimmt sowie die Leistungsfähigkeit mittels verschiedener Benchmarks ermittelt. Es wird ein zweifach satzassoziativer Cache verwendet. Schreib-Fehlzugriffe werden nicht im Cache gespeichert, sondern direkt in den Hauptspeicher geschrieben. Die Untersuchungen von Banakar et al. zeigen, dass die Ressourceneffizienz durch die Verwendung von Scratchpad-Speicher signifikant gesteigert werden kann. Im Durchschnitt kann der Flächenbedarf um 33 % und die Ausführungszeit um 18 % gesenkt werden. Dies resultiert in einer Reduktion des Flächen-Zeit-Produktes von im Mittel 54 %.

Der Cell-Prozessor von IBM [208] besitzt neben einer PowerPC-CPU acht *Synergistic Processing Elements* (SPEs) mit jeweils 256 kB Scratchpad-Speicher sowie einem DMA-Controller. Baer [18, S. 326] gibt an, dass Scratchpad-Speicher im Vergleich zu Caches einen um 50 % geringeren Flächenbedarf besitzen. Zudem profitieren insbesondere Streaming-Anwendungen von den Blocktransfers der DMA-Controller.

Liu und Zhang [124] vergleichen Scratchpad-Speicher sowie Caches in Prozessorsysteme mit jeweils drei Speicher-Hierarchiestufen (L1, L2 und Hauptspeicher). Der L2-Speicher bzw. Cache wird für Daten und Instruktionen verwendet. Das System mit Scratchpad-Speicher besitzt sowohl eine höhere Performanz als auch einen geringeren Energiebedarf. Zudem zeigt die Kombination von L1- und L2-Scratchpad-Speicher Vorteile gegenüber einer Architektur mit einem großen L1-Speicher, da dieser die maximale Taktfrequenz des Systems reduziert. Gleichzeitig erleichtert die Verwendung von Scratchpad-Speichern die Abschätzung der maximalen Ausführungszeit (*Worst Case Execution Time*, WCET) einer Anwendung erheblich.

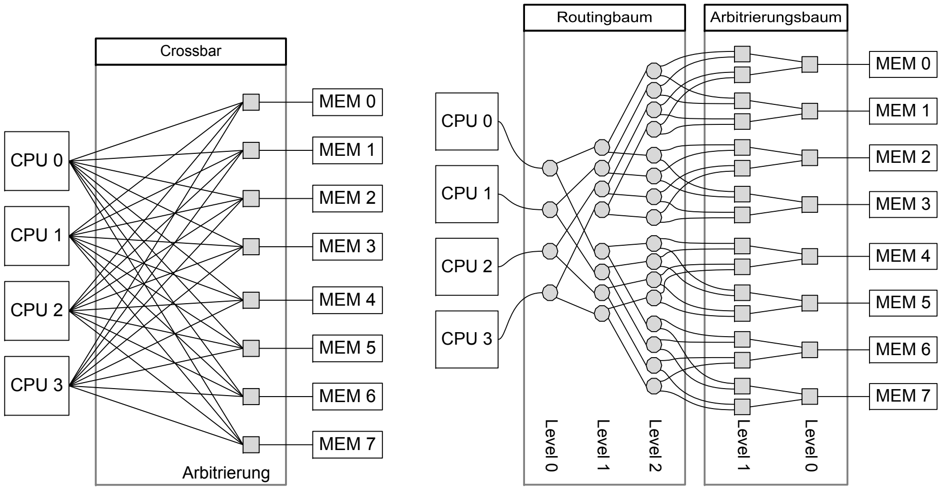
Loghi et al. [126] vergleichen drei Ansätze zur Kohärenzverwaltung in einem Multiprozessorsystem. Die Speicherkohärenz wird vom Betriebssystem, dem Compiler oder von einer Hardwareeinheit sichergestellt. Der betriebssystembasierte Ansatz weist im Vergleich die schlechteste Performanz auf und zeigt die Erfordernis einer auf die jeweilige Hardware und Anwendung angepasste Optimierung. Die hardwarebasierte Kohärenzverwaltung zeigt die höchste Performanz, weist allerdings einen signifikanten Mehrverbrauch an Energie gegenüber den softwarebasierten Ansätzen auf. Mit einer steigenden CPU-Anzahl im MPSoC steigt zudem die Energie pro Speicherzugriff bei der hardwarebasierten Lösung überproportional an. Der compilerbasierte Ansatz verwendet nur für die Daten die Caches, die ausschließlich von einer einzelnen CPU verwendet werden. Geteilte Daten liegen immer im Hauptspeicher und werden hier von den CPUs direkt gelesen und geschrieben. Dieser Ansatz weist die höchste Energieeffizienz und eine gute Skalierbarkeit auf. Anwendungen, die durch Lesezugriffe dominiert werden, zeigen jedoch eine sehr schlechte Performanz.

Der Stand der Technik zeigt, dass die Verwendung insbesondere von kohärenten Caches einen erheblichen Mehraufwand bezüglich Chipfläche und Energiebedarf bedingt. Bei Universalcomputern (*General-Purpose-PCs*) überwiegen jedoch die Vorteile durch eine einfache Programmierbarkeit und die Wiederverwendbarkeit bestehender Software. Im Gegensatz hierzu zeigen in eingebetteten MPSoCs Scratchpad-Speicher mit einer effizienten Compilerunterstützung eine hohe Ressourceneffizienz. Durch die Verwendung von Programmiersprachen wie OpenCL oder StreamIt ist zudem die Portabilität von Anwendungen zwischen MPSoCs verschiedener Hersteller und Generationen gegeben (siehe Abschnitt 3.2). In Abschnitt 2.6.4 wird zudem ein Programmiermodell vorgestellt, das für MPSoC-Architekturen mit verteilten Scratchpad-Speichern optimiert ist.

2.3.2 Eng gekoppelter gemeinsamer Level-1-Datenspeicher

Ein gemeinsamer L1-Speicher erlaubt einen Zugriff aller CPUs eines Clusters mit geringer Latenz. Der Speicher ist hierzu über eine Speicher-Verbindungsstruktur direkt an alle CPUs angebunden. Die Latenz entspricht in der Regel der von lokalen L1-Datenspeichern der CPUs. Bei eingebetteten Prozessoren sind dies typischerweise ein bis drei Takte [22; 52; 63; 198]. Greifen zwei CPUs im gleichen Taktzyklus auf den gemeinsamen Speicher zu, muss eine CPU für einen Takt angehalten werden. Um die Wahrscheinlichkeit solch eines Zugriffskonfliktes zu vermindern, besteht der gemeinsame Speicher in der Regel aus mehrere Speicherbänken. Auf verschiedene Speicherbänke können CPUs gleichzeitig zugreifen. Speicherzugriffe der CoreVA-CPU besitzen eine Latenz von zwei Takten. Um Wartetakte der CPU oder die Integration einer weiteren Pipelinestufe zu vermeiden, muss die Speicher-Verbindungsstruktur ebenfalls eine Latenz von zwei Takten besitzen. Unter Berücksichtigung der Verzögerung der Pipeline-Register der CPU sowie des eigentlichen SRAM-Speicherblockes bedeutet dies, dass die Verbindungsstruktur rein kombinatorisch aufgebaut sein muss. Aus diesem Grund muss die Speicher-Verbindungsstruktur eine sehr geringe kombinatorische Verzögerungszeit aufweisen, um die maximale Taktfrequenz des CPU-Clusters nicht zu beschränken. Im ersten Zyklus eines Speicherzugriffes berechnet die CPU die Speicheradresse in der EX-Stufe (siehe Abschnitt 2.2) und übergibt sie über ihren Adressdekoder an die Speicher-Verbindungsstruktur. Die Anfrage wird durch die Verbindungsstruktur zur entsprechenden Speicherbank weitergeleitet. Zur nächsten Taktflanke wird die Anfrage vom SRAM-Speicher übernommen und verarbeitet. Im Falle einer Leseanfrage werden die gelesenen Daten an die Verbindungsstruktur übergeben, zur anfragenden CPU geleitet und von einem Register der ME-WR-Pipelinestufe gespeichert.

Als Topologie für die Speicher-Verbindungsstruktur wird im Rahmen dieser Arbeit eine Crossbar mit einer *Mesh of Trees* (MoT)-Verbindungsstruktur [168] verglichen. Die Crossbar verbindet über Punkt-zu-Punkt Verbindungen jede CPU mit jeder Speicherbank (siehe Abbildung 2.5a). Jeder Speicherbank ist ein Round-Robin-Arbiter zugeordnet, der Zugriffskonflikte auflöst. Um bei einer Leseanfrage die gelesene Daten an die entsprechende CPU leiten zu können, speichert jede Speicherbank die Nummer der jeweils ausgewählten CPU zwischen. Die MoT-Verbindungsstruktur besteht aus einem Routing- und einem Arbitrierungsbaum (siehe Abbildung 2.5b). Der Routingbaum besteht aus Routingknoten, die Anfragen der CPUs an die entsprechenden Speicherbank weiterleiten. Arbitrierungsknoten verarbeiten Anfragen von jeweils zwei CPUs. Treffen zwei



(a) Crossbar-Verbindungsstruktur

(b) MoT-Verbindungsstruktur [168]

Abbildung 2.5: Topologien von Verbindungsstrukturen für einen gemeinsamen L1-Speicher mit vier CPUs und acht Bänken

Anfragen im gleichen Taktzyklus ein, wird eine *Round-Robin*-Arbitrierung durchgeführt. Bei einer Leseanfrage speichern die Arbitrierungsknoten die Nummer der anfragenden CPU zwischen. Zudem speichern die Routingknoten die Nummer der ausgewählten Speicherbank, um die gelesenen Daten an die entsprechende CPU führen zu können. Gemeinsamer L1-Speicher ermöglicht die Verwendung anderer Programmiermodelle als die im vorangegangenen Abschnitt beschriebene NUMA-Speicherarchitektur (siehe Abschnitt 2.6.4).

Stand der Technik

Rahimi et al. stellen in [168] eine synthetisierbare logarithmische Verbindungsstruktur für Multi-Bank-L1-Datenspeicher vor. Es werden Konfigurationen von CPU-Clustern bezüglich Verlustleistung, Flächenbedarf und maximaler Taktfrequenz in einer 65-nm-Technologie verglichen. Rahimi et al. präsentieren Ergebnisse von Synthesen und Layouts mit bis zu 32 CPUs und 64 Speicherbänken. Leseanfragen von CPUs weisen eine Latenz von einem einzigen Takt auf. Diese sehr geringe Latenz wird durch einen verschobenen Takt für die Speicherblöcke erreicht. Der verschobene Takt bedingt eine komplexe Taktverteilung und begrenzt die maximale Taktfrequenz des Gesamtsystems. Die CoreVA-CPU besitzt eine Leselatenz von zwei Takten, daher ist ein verschobener Takt bei einer Implementierung für das CoreVA-MPSoC nicht notwendig. Kakoee et al. [105] erweitern die Arbeit von Rahimi et al. um die Speicher-Verbindungsstruktur zuverlässiger und robuster gegenüber Prozessschwankungen bei der Fertigung zu machen. Dies wird durch die Integration einer optionalen Pipelinestufe zwischen den CPUs und den Speicherbänken erreicht. In [106] wird die Verbindungsstruktur von Rahimi et al. für die Implementierung von einem gemeinsamen L1-Datencache verwendet. Die Leselatenz im Fall eines Cache-Hits beträgt einen Takt. Im Vergleich zu einem Scratchpad-Speicher zeigt der Cache einen Mehrbedarf an Fläche und Verlustleistung von 5 % bis 30 %, erlaubt jedoch eine einfachere Programmierung.

Gautschi et al. [63] stellen einen CPU-Cluster mit vier OpenRISC-CPUs und einem gemeinsamen L1-Speicher vor. Die CPUs besitzen drei Pipelinestufen. Die Schaltung wird bei einer *Near-Threshold*-Spannung von 0,6 V betrieben. Als Technologie kommt eine 28-nm-FD-SOI-Standardzellenbibliothek zum Einsatz. Durch Verbesserungen an der Architektur der OpenRISC-CPUs können Gautschi et al. die Energieeffizienz um 50 % steigern.

Dogan et al. [52] präsentieren einen Achtkern-Multiprozessor mit einem gemeinsamen L1-Datenspeicher und 16 Bänken sowie einem gemeinsamen L1-Instruktionsspeicher und acht Bänken. Einsatzbereich des MPSoCs sind biomedizinische Anwendungen. Einzelne Bänke des gemeinsamen Datenspeichers können zur Laufzeit exklusiv einer CPU zugeordnet werden („private Bank“). Eine *Memory Management Unit* (MMU) verwaltet hierbei den Zugriff auf private und gemeinsame Bänke. Benötigt eine Anwendung nicht alle Speicherbänke, kann die MMU verwendet werden, um einzelne Bänke abzuschalten (*Power-Gating*).

Die NVIDIA Maxwell-GPU[32]-Familie besteht aus mehreren *Streaming*-Multiprozessoren (SMs), die jeweils 128 *Shader*-ALUs („CUDA[33]-Kerne“) integrieren. Jeder SM beinhaltet zudem einen gemeinsamen L1-Datenspeicher mit 32 Bänken sowie L1-Daten- und Instruktionscaches. Die Datenbreite des gemeinsamen Datenspeichers beträgt 32 bit pro Bank. Wird von einer Bank gelesen, kann das Ergebnis an mehrere Shader-ALUs im gleichen Takt gesendet werden (*Multicast*). Die leistungsfähigste GPU der Familie (GM204) beinhaltet 16 SMs mit insgesamt 2048 Shader-ALUs sowie 2 MB L2-Cache.

Der STM STHORM-MPSoC [22] besitzt pro CPU-Cluster 16 CPUs und einen gemeinsamen L1-Datenspeicher mit 32 Bänken und einer Größe von 256 kB. Die Plurality HAL-Architektur [198] verbindet bis zu 256 CPUs mit einem gemeinsamen Daten- und einem gemeinsamen Instruktionscache. Detailierte Informationen über die Speicher-Verbindungsstruktur sind nicht veröffentlicht. Weitere Informationen zum STM STHORM und Plurality HAL sind im Abschnitt 2.1 zu finden.

Der Stand der Technik zu eng gekoppelten gemeinsamen L1-Speichern zeigt, dass die Verwendung dieser Speicher eine ressourceneffiziente Alternative zu Caches sein kann. Die allermeisten Implementierungen besitzen die doppelte Anzahl an Speicherbänken wie CPUs, also einen *Banking-Faktor* von zwei. Im Rahmen dieser Arbeit werden

[32] *Graphics Processing Unit*
[33] *Compute Unified Device Architecture*

verschiedene Topologien (Crossbar und MoT) in Bezug auf Flächenbedarf und maximale Leistungsaufnahme hin untersucht. Zudem wird die optimale Anzahl an Speicherbänken für verschiedene Anwendungen bestimmt (siehe Abschnitt 5.6).

2.3.3 Gemeinsamer Level-2-Speicher

L2-Speicher besitzt im Vergleich zu L1-Speicher eine höhere Zugriffslatenz, ermöglicht jedoch eine größerer Speicherkapazität. Typische Kapazitäten für L2-Speicher in eingebetteten Systemen sind 64 kB bis 2 MB. General-Purpose-CPUs besitzen teilweise privaten L2-Caches. MPSoCs mit ARM Cortex-A7, -A15, -A53 und/oder -A57 gruppieren bis zu vier CPUs und gemeinsamen L2-Cache in einem CPU-Cluster [74; 78; 197]. Der eingebettete Multiprozessor Kalray MPPA-256 [109] besitzt pro CPU-Cluster 2 MB L2-Scratchpad-Speicher mit 16 Bänken (siehe Abschnitt 2.1.2).

Loi und Benini stellen in [130] einen L2-Cache vor, der mehrere CPU-Cluster mit gemeinsamen L1-Scratchpad-Speicher an einen externen DRAM-Speicher anbindet. Da der L2-Cache der einzige Cache im System ist, sind Mechanismen zur Sicherstellung der Speicherkohärenz nicht notwendig. Die Anzahl der Schnittstellen zu den CPU-Clustern ist konfigurierbar und folgt dem STBus-Standard (siehe Abschnitt 2.4.1). Die Latenz von Zugriffen beträgt 9 bis 16 Takte. In einer 28-nm-FD-SOI-Technologie erreicht der Cache eine maximale Taktfrequenz von 1 GHz. Der Mehrbedarf an Chipfläche gegenüber einem Scratchpad-Speicher der gleichen Größe beträgt 20 % bis 30 %.

L3-Speicher und -Caches besitzen wiederum höhere Latenzen und Speicherkapazitäten im Vergleich zu L2-Speichern. L3-Caches werden häufig in Verbindung mit einem DRAM-Speichercontroller verwendet, um Zugriffe auf den externen DRAM zu beschleunigen. Im CoreVA-MPSoC kann L3-Speicher als eigener Knoten an das On-Chip-Netzwerk (NoC) angebunden werden (siehe Abschnitt 1.2).

2.3.4 Vergleich verschiedener Speicherarchitekturen

Speicherarchitekturen können anhand der verwendeten Ebenen (Level, z. B. L1, L2 etc.) sowie der Architektur und Anbindung der einzelnen Ebenen unterschieden werden. Auf L1-Ebene sind dedizierte Speicher für Daten und Instruktionen vorhanden, auf höheren Ebenen wird diese Unterscheidung typischerweise nicht getroffen. Die Architekturentscheidung ob Caches oder Scratchpad-Speicher verwendet werden, hat große Auswirkungen auf die Programmierung und die Ressourceneffizienz des gesamten MPSoC [20; 130]. Scratchpad-Speicher weisen eine höhere Ressourceneffizienz auf, erfordern jedoch eine Partitionierung von Daten auf die einzelnen Speicher im System.

Die einzelnen Speicherebenen können von einer einzelnen CPU verwendet werden (privat) oder als gemeinsamer Speicher von mehreren CPUs. Es ist auch eine Kombination von beiden Ansätzen möglich. Dies erschwert die Partitionierung der Anwendungsdaten, kann jedoch die Ressourceneffizienz des Systems steigern. Der CoreVA-MPSoC-Compiler (siehe Abschnitt 3.2.3) kann beispielsweise Heap und Stack im lokalen L1-Datenspeicher der einzelnen CPUs ablegen und den gemeinsamen L1-Speicher für CPU-zu-CPU-Kommunikation verwenden (siehe Abschnitt 5.6). Abhängig von der Speicherarchitektur eines MPSoCs ist ein angepasstes Programmier- und Synchronisierungsmodell erforderlich (siehe Abschnitt 2.6).

2.4 Eng gekoppelte On-Chip-Verbindungsstrukturen

Busbasierte Kommunikationssysteme bestehen typischerweise aus einem oder mehreren aktiven Teilnehmern (*Master*) und einem oder mehreren passiven Teilnehmern (*Slaves*, siehe Abbildung 2.6). Ein Master kann über eine Verbindungsstruktur (*Interconnect*) Schreib- oder Leseanfragen an einen Slave stellen. Der Slave speichert bzw. verarbeitet die Daten einer Schreibanfrage und liefert bei einer Leseanfrage Daten an den Master zurück.

Es kann zwischen asynchroner und synchroner Übertragung unterschieden werden [86, S. 227; 153, S. 22]. Bei einer synchronen Übertragung beziehen sich alle Kontroll- und Datensignale aller Busteilnehmer auf einen gemeinsamen Bustakt. Zu jeder steigenden Taktflanke dieses Taktes werden Kontroll- und Datensignale von den Ein- und Ausgangsregistern der Busteilnehmer an den Bus angelegt bzw. ausgewertet. Bei einer asynchronen Übertragung wird ein *Handshake*-Protokoll zwischen Master und Slave verwendet. Hierbei wird jedes Kontroll- und Datensignal durch den jeweiligen Kommunikationspartner bestätigt. Die Übertragungsgeschwindigkeit bei asynchroner Kommunikation ist durch die Verzögerungszeit zwischen Master und Slave bestimmt. Eine asynchrone Übertragung benötigt mindestens vier Signalwechsel zwischen Master und Slave, bei synchroner Übertragung sind nur zwei Signalwechsel notwendig [86, S. 236]. Für On-Chip-Kommunikation wird vorwiegend eine synchrone Übertragung verwendet. Dies gilt auch für alle in diesem Abschnitt vorgestellten Bus-Standards. Der SpiNNaker-Multiprozessor (siehe Abschnitt 2.1.2) verwendet ein asynchrones Kommunikationsnetzwerk zur Kopplung von 65536 Chips mit über einer Millionen CPU-Kernen. Die Netzwerk-Schnittstellen der einzelnen CPUs auf einem Chip sind jedoch synchron ausgeführt.

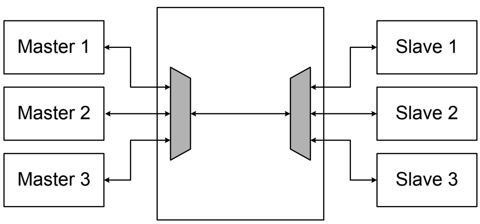


Abbildung 2.6: Verbindungsstruktur mit geteiltem Bus

Die Auswahl zwischen verschiedenen Slaves eines Busses sowie von verschiedenen Ressourcen innerhalb eines Slaves erfolgt auf der Basis von Adressen (*Memory-Mapped*). Jeder Busteilnehmer hat eine definierte Schnittstelle (*Interface*). Manche Busteilnehmer besitzen sowohl Master- als auch Slave-Schnittstellen und werden als hybride Teilnehmer bezeichnet [153, S. 19]. Im Fall vom CoreVA-MPSoC hat eine einzelne CPU beispielsweise einen Master-Port für Zugriffe auf den gemeinsamen Speicher. Ein Slave-Port wird für den (externen) Zugriff auf die lokalen L1-Speicher sowie zur Initialisierung der CPU verwendet.

Ein Bus kann verschiedene Transfermodi verwenden, um Anfragen von einem Master zu einem Slave zu übermitteln. Eine Anfrage kann aus einem einzelnen Datenwort (Einzelzugriff) oder aus mehreren Datenwörtern von aufeinander folgenden Adressen (*Burst*-Zugriff) bestehen [17; 18, S. 265]. Bei einem Burst hat ein Master für mehrere Taktzyklen Zugriff auf den Bus und kann kontinuierlich Daten übertragen, ohne von einem anderen Master unterbrochen zu werden. Dies kann sowohl die Übertragungsbandbreite im gesamten Multiprozessorsystem als auch für einzelne Busteilnehmer (Master und/oder Slaves) steigern. Bei komplexen Bussen kann auf einem dedizierten Adresskanal nur die Startadresse des Burst-Zugriffes übertragen werden [121]. Die Auslastung des Adresskanals sinkt hierdurch signifikant. Bei simplen Bus-Implementierungen legt ein Master Adressen und Daten so lange an den Bus an, bis der Slave die Anfrage bestätigt hat. Weist der Bus und/oder der Slave eine hohe Latenz auf, kann eine Anfrage den Bus für mehrere Takte blockieren. Besitzt ein Bus die Fähigkeit, mehrere Anfragen gleichzeitig zu verarbeiten, wird der Begriff Fließbandverarbeitung (*Pipelined-Bus*) verwendet. Kann ein Master weitere Anfragen an den Bus schicken, obwohl eine vorherige Anfrage noch nicht abgeschlossen ist, spricht man von der Möglichkeit ausstehender Transaktionen (*Outstanding-Bus-Transaction*) [153, S. 29]. Unterstützt ein Bus Pipelining, können auch Registerstufen oder Warteschlangen (FIFOs[34]) innerhalb der Cluster-Verbindungsstruktur verwendet werden. Dies ermöglicht eine Steigerung der maximalen Taktfrequenz der Verbindungsstruktur. Durch die Verwendung von FIFOs kann ein Master weitere Anfragen stellen, obwohl der entsprechende Slave gerade blockiert ist. Zudem kann der Bus von anderen Mastern weiterhin verwendet werden.

Unterstützt ein Bus atomare Transaktionen, kann ein Master ein Datum von einem Slave lesen, es verändern und zurück schreiben (*Read-Modify-Write*), ohne dass ein anderer Master in der Zwischenzeit auf diese Adresse zugreifen darf. Diese Funktion kann zur Synchronisierung zwischen verschiedenen Mastern verwendet werden (z. B. Semaphore, siehe Kapitel 2.6). Eine einfache Realisierung atomarer Operationen hält den gesamten Bus an, bis die atomare Operation abgeschlossen ist. Effizienter ist es, den Bus nur bei einem Zugriff eines anderen Masters auf den entsprechenden Slave bzw. die Speicheradresse anzuhalten.

Besitzt ein Slave eine hohe Latenz bei Leseanfragen, kann dieser den Bus freigeben, bis die Daten vorliegen (Geteilte Transaktionen, *Split-Transactions*). Der Slave

[34] *First-In First-Out*

signalisiert dem Bus, dass nun die angeforderten Daten an den entsprechenden Master versendet werden können. Der Lesezugriff wird in zwei Schreibzugriffe aufgeteilt [170]. In der Zwischenzeit können andere Master den Bus verwenden und der Gesamtdurchsatz im Bus steigt. Eine Erweiterung von geteilten Transaktionen erlaubt dem Slave, die Reihenfolge der Antworten auf Anfragen zu ändern (*Out-of-Order*-Transfer). Hierzu wird jeder Anfrage vom Master eine eindeutige Identifikationsnummer (*ID*) zugeordnet. Dies ermöglicht z. B. einem DRAM-Speichercontroller, Anfragen auf gleiche Bänke und Seiten des DRAM-Speichers zu bündeln. Die Anzahl der gleichzeitig ausstehenden Anfragen ist maßgeblich für den Ressourcenbedarf von entsprechenden Busarchitekturen verantwortlich [153, S. 31].

Manche Busarchitekturen ermöglichen es zudem, Busteilnehmer mit verschiedenen Datenbreiten oder auch Taktfrequenzen an einem Bus zu betreiben. Die Verbindungsstruktur übersetzt zwischen den verschiedenen Teilnehmern und fügt, falls erforderlich, Wartetakte ein [153, S. 18]. Eine unterschiedliche Taktung der Schnittstellen der Busteilnehmer kann den Leistungsbedarf der Verbindungsstruktur verringern.

Anwendungen, die einen kontinuierlichen Datenstrom aufweisen (z. B. Videoverarbeitung), benötigen keine Adressinformationen bei der Übertragung. Die Daten müssen in der Reihenfolge verarbeitet werden, in der sie an einer Verarbeitungseinheit ankommen. Aus diesem Grund besitzen *Streaming*-Busse keine Adressleitungen. Dies vereinfacht sowohl die Verbindungsstruktur als auch den Aufbau der Busteilnehmer [17].

Eng gekoppelte On-Chip-Verbindungsstrukturen können anhand der verwendeten Topologie unterschieden werden. Ein gemeinsam genutzter Bus (Geteilter Bus, *Shared Bus*) ist in Abbildung 2.6 dargestellt. Pro Taktzyklus kann genau ein Master auf einen Slave zugreifen. Greifen im gleichen Takt mehrere Master auf den Bus zu, muss ein Arbiter entscheiden, welchem Master Zugriff gewährt wird. Zur Zugriffsverwaltung sind das Rundlauf-Verfahren (*Round-Robin*), die Verwendung von Prioritäten (*Quality of Service*, QoS) oder eine Kombination von beiden Verfahren weit verbreitet. Verschiedene Algorithmen und Implementierungen von Arbitern werden in Abschnitt 2.4.5 beschrieben.

Ein geteilter Bus kann auf verschiedene Arten realisiert werden. *Tri-State*-Implementierungen verwenden eine gemeinsame, bidirektionale Leitung. Ein Arbiter verhindert, dass mehrere Busteilnehmer gleichzeitig auf die Leitung treiben. Tri-State-Busse besitzen einen geringen Flächenbedarf, werden jedoch aufgrund des hohen Energiebedarfs aktuell nur für die Off-Chip-Kommunikation eingesetzt [170; 183]. In modernen Chip-Technologien werden Busse typischerweise durch Multiplexer oder AND-OR-Schaltungen realisiert [153, S. 21]. Geteilte Busse zeichnen sich durch ihren geringen Ressourcenverbrauch aus. Ein geteilter Bus erlaubt mit wenig Zusatzaufwand, bei einem Buszugriff Daten an mehrere (*Multicast*) oder alle Slaves (*Broadcast*) zu übertragen [18, S. 265]. Ein Nachteil von geteilten Bussen ist jedoch, dass die Leistungsfähigkeit nicht mit steigender Anzahl an Kommunikationsteilnehmern skaliert. Dies ist auf physikalische Beschränkungen (z. B. steigende Länge von Verbindungsleitungen) sowie auf eine steigende Anzahl an Buskonflikten (*Bus-Contention*) zurückzuführen [18,

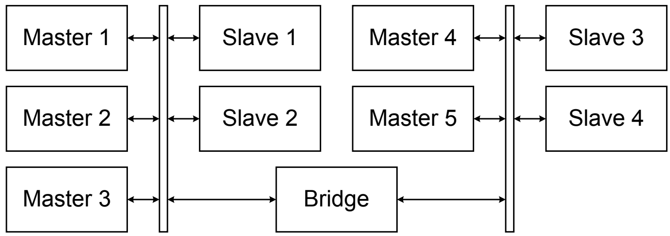
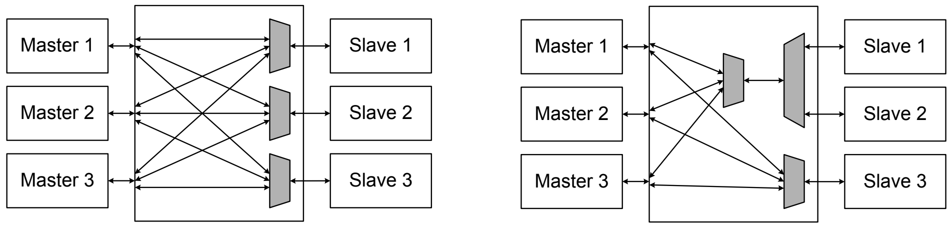


Abbildung 2.7: Hierarchische Verbindungsstruktur

S. 265]. Eine Möglichkeit, mehrere Buszugriffe pro Taktzyklus zu ermöglichen, ist die Verwendung einer hierarchischen Verbindungsstruktur (siehe Abbildung 2.7). Die Busteilnehmer sind an mehrere Busse angeschlossen, zwischen den Bussen übersetzen eine oder mehrere *Bridges*. Wenn die Master mit einem Slave aus dem gleichen Bus kommunizieren, können mehrere Buszugriffe innerhalb der verschiedenen Busse parallel erfolgen. Hierarchische Busse werden auch verwendet, um Slaves mit einer hohen Latenz (z. B. Interrupt-Controller oder eine externe UART-Schnittstelle) von leistungsfähigen Komponenten wie CPUs und On-Chip-Speicher zu entkoppeln [153, S. 33]. Ein so genannter Peripherie-Bus stellt geringe Anforderungen an die Übertragungsbandbreite und ist daher typischerweise ressourcenschonender aufgebaut im Vergleich zu hochperformanten Prozessor-Bussen.

Punkt-zu-Punkt-Verbindungsstrukturen (auch Kreuzverbinder, Koppelfeld, *Switch-Matrix* oder *Crossbar* genannt) verbinden jeden Master direkt mit jedem Slave über einen dedizierten Bus (siehe Abbildung 2.8a) [18, S. 267]. Hierdurch können mehrere Master gleichzeitig mit verschiedenen Slaves kommunizieren. Stellen zwei Master zeitgleich eine Anfrage an den selben Slave, ist wiederum eine Arbitrierung der Zugriffe notwendig. Im Gegensatz zu einem geteilten Bus sind der Flächenbedarf und die Leistungsaufnahme einer Crossbar erheblich größer. Pasricha et al. nennen eine Verdopplung der Größe der Verbindungsstruktur, wenn zwei Master und zwei Slaves über einen geeilten Bus oder eine Crossbar verbunden sind [153, S. 84]. Zudem kann die maximale Taktfrequenz bei der Verwendung einer Crossbar geringer sein [141, S. 15]. Eine Kombination aus geteilten Bussen und einer (vollen) Crossbar stellen partielle Crossbar-Architekturen dar (siehe Abbildung 2.8b). Mehrere geteilte Busse sind über eine Crossbar miteinander verbunden [153, S. 36]. Hierdurch wird Flächenbedarf und Leistungsaufnahme gegenüber einer vollen Crossbar verringert, es können jedoch weniger Anfragen gleichzeitig verarbeitet werden. Die optimale Konfiguration einer partiellen Crossbar ist sehr stark vom Anwendungsszenario abhängig und kann durch eine Simulationsumgebung ermittelt werden (siehe Kapitel 3.3.1).

Weitere Topologien für On-Chip-Verbindungsstrukturen wie Meshes oder Ringe besitzen unterschiedliche Latenzen zwischen verschiedenen Teilnehmern [153, S. 36]. Im CoreVA-MPSoC wird ein Mesh-Netzwerk als weitere Hierarchiestufe zusätzlich zu



(a) Vollständige Crossbar (b) Partielle Crossbar

Abbildung 2.8: Parallele On-Chip-Verbindungsstrukturen

dem in dieser Arbeit vorgestellten eng gekoppelten Prozessorcluster realisiert. Aus diesem Grund werden Ringe und Mesh-Netzwerke im Rahmen dieser Arbeit nicht betrachtet.

2.4.1 Verbreitete Bus-Standards

Um die Interoperabilität der On-Chip-Bus-Implementierungen verschiedener Hersteller sicherzustellen, existieren eine Reihe an Bus-Standards bzw. -Protokolle. Bus-Protokolle legen den Aufbau (z. B. Art und Anzahl von Signalen) sowie das zeitliche Verhalten (z. B. Dauer einer Nachricht) einer Verbindungsstruktur fest [153, S. 17]. Viele Standards basieren auf Entwicklungen von einem einzelnen Hersteller und sind zur freien, kostenlosen Nutzung freigegeben. Typischerweise existieren verschiedene Implementierungen von der Verbindungsstruktur sowie zahlreiche IP[35]-Blöcke mit Schnittstellen für den jeweiligen Standard. Im Folgenden werden die am weitesten verbreiteten Bus-Standards vorgestellt. Neben den hier erwähnten gibt es noch eine Vielzahl weiterer Standards wie z. B. Altera Avalon [16].

ARM AMBA

ARM [13] spezifiziert in der Protokollfamilie AMBA[36] verschiedene Bus-Standards. Einsatzgebiet waren anfangs Prozessorsysteme mit einem oder mehreren ARM-Prozessorkernen. Aktuell gelten die AMBA-Protokolle als De-facto-Standard für viele Anwendungsbereiche. Auch Prozessoren anderer Hersteller bieten AMBA-konforme Schnittstellen. Aktuelle Revision ist AMBA5 [5]. Im Folgenden werden die wichtigsten Protokolle der AMBA-Familie vorgestellt.

AHB (*Advanced High Performance Bus*) wurde 1999 in der AMBA2-Spezifikation [6; 140] von ARM beschrieben. Zum Zeitpunkt seiner Veröffentlichung führte AHB zahlreiche neuartige Funktionen zur Steigerung der Leistungsfähigkeit ein. Hierzu

[35] *Intellectual Property*
[36] *ARM Microcontroller Bus Architecture*

unterstützt AHB Pipelining von Operationen, einen Burst-Modus sowie geteilte Transaktionen [153, S. 45]. Neben einem 32-bit-Adressbus verfügt AHB über getrennte Lese- und Schreibbusse. Es werden Datenbreiten zwischen 8 bit und 1024 bit unterstützt [6]. Als Bus-Topologien sind ein geteilter Bus sowie eine vollständige oder partielle Crossbar möglich [140]. Zugriffe auf den Bus sind in eine Adress- und eine Datenphase unterteilt. In der Adressphase legt ein Master eine Adresse an den Bus an. Ein Arbiter entscheidet, ob der Master Zugriff auf den Bus erhält. Je nach Implementierung des Arbiters kann dies mehrere Takte in Anspruch nehmen. Einfache Arbiter besitzen keine zusätzliche Verzögerung. In der darauffolgenden Datenphase legt der Master entweder die zu schreibenden Daten auf den Schreibkanal oder wartet auf die angeforderten Daten vom Lesekanal. Eine Leseanfrage kann mehrere Takte dauern und wird durch das *HREADY*-Signal abgeschlossen. Die Adress- und Datenphasen von verschiedenen Mastern können parallel ablaufen, um den Durchsatz vom Bus zu steigern (*Pipelined-Bus*) [153, S. 47 ff.]. AHB unterstützt Burst-Zugriffe für mehrere aufeinanderfolgende Datenwörter. Die Arbitrierung muss hierbei nur einmal erfolgen, die Adresse muss jedoch für jedes Datenwort übertragen werden. Ein AHB-konformer Bus kann einen Burst-Zugriff unterbrechen, falls z. B. ein anderer Master einen Zugriff erhöhter Priorität durchführen muss. Die Länge eines Bursts ist nicht beschränkt.

AXI (*Advanced Exensible Interface*) kann als Weiterentwicklung von AHB angesehen werden. AXI wurde in AMBA3 eingeführt und in AMBA4 [5] erweitert. Der AXI-Standard spezifiziert fünf verschiedene Kanäle:

- Lese-Adress-Kanal (*Read-Address*)
- Schreib-Adress-Kanal (*Write-Address*)
- Lese-Daten-Kanal (*Read-Data*)
- Schreib-Daten-Kanal (*Write-Data*)
- Schreib-Bestätigung-Kanal (*Write-Response*)

Durch die Aufteilung in Schreib- und Lese-Kanäle können Schreib- und Leseoperationen gleichzeitig erfolgen. Die beiden Datenkanäle können Datenbreiten von 8 bit bis 1024 bit besitzen. Jeder AXI-Kanal ist unidirektional (Master → Slave oder Slave → Master). Daher ist es möglich, einzelne oder alle Kanäle um Registerstufen zu erweitern, um die Taktfrequenz der AXI-Verbindungsstruktur zu erhöhen. Die Kanäle *Read-* und *Write-Address* sowie *Write-Data* führen von einem AXI-Master zu einem AXI-Slave, *Read-Data* und *Write-Response* vom Slave zum Master. AXI unterstützt Bursts mit bis zu 256 Datenwörtern, die Adresse muss hierbei nur einmal übertragen werden [17, S. 5]. Dies ist eine Einschränkung zu AHB (welches die Länge eines Burst-Zugriffs nicht beschränkt), verringert jedoch den Entwurfsaufwand für AXI-Master und -Slaves [121]. Die AXI-Spezifikation erlaubt zudem Master und Slaves mit unterschiedlichen Datenbreiten und Taktfrequenzen zusammen an einer Verbindungsstruktur zu betreiben. Die AXI-Verbindungsstruktur übersetzt zwischen den verschiedenen Datenbreiten und Frequenzen. Wie die AHB-Spezifikation schreibt AXI keine Topologie für die Verbindungsstruktur vor. Es ist zudem möglich, die fünf verschiedenen AXI-Kanäle mit unterschiedlichen Topologien zu implementieren (siehe Abschnitt 5.5). *Split Transacti-*

ons, atomare Operationen und *Transaction Reordering* sind optionale Erweiterungen von AXI-Verbindungsstrukturen [153, S. 64]. Zudem spezifiziert der AXI4-Standard spezielle Energiesparmodi für AXI-Slaves [5, S. 105]. In dieser Arbeit wird die AXI-Implementierung aus der Bachelorarbeit [241] verwendet.

AMBA AXI-Lite [5, S. 123] ist ein Schnittstellenstandard zur ressourceneffizienten Anbindung von Slaves mit geringen Anforderungen bezüglich der Übertragungsbandbreite. AXI-Lite basiert auf AXI, unterstützt jedoch kein Burst sowie ausschließlich Datenbreiten von 32 bit und 64 bit.

ACE (*AXI Coherency Extensions*) ist eine Erweiterung von AXI und ermöglicht Cache-kohärente Systeme [5, S. 133]. Grundlagen zur Speicherkohärenz sind in Kapitel 2.3 beschrieben. Das ACE-Protokoll erlaubt es dem Entwickler festzulegen, welche Speicherbereiche und welche Bus-Komponenten (CPUs bzw. deren Caches, Speichercontroller etc.) kohärent verwaltet werden sollen. ACE unterstützt verschiedene Kohärenzprotokolle. Mit ACE-Lite existiert zudem eine Cache-kohärente Erweiterung von AXI-Lite. Die Cortex-A9- und Cortex-A15-CPUs von ARM verfügen über eine AXI- bzw ACE-konforme, Cache-kohärente Bus-Schnittstelle [80].

APB (*Advanced Peripheral Bus*) ist ein Peripherie-Bus für Slaves mit geringem Bandbreitenbedarf und hoher Latenz [4]. Als einziger Master an einer APB-Verbindungsstruktur fungiert eine Bridge zu einer AHB- oder AXI-Verbindungsstruktur [129]. Am APB sind typischerweise Zeitgeber, Interrupt-Controller oder langsame externe Schnittstellen (z. B. UART und SPI[37]) angeschlossen. APB unterstützt nur Einzelzugriffe und ist in der Regel deutlich niedriger getaktet als die zugehörige AHB- bzw. AXI-Verbindungsstruktur [153, S. 55].

IBM CoreConnect

Die von IBM spezifizierte CoreConnect-Architektur ähnelt in vielen Punkten AMBA2 von ARM und ist für ähnliche Einsatzbereiche entworfen worden [82]. CoreConnect beinhaltet einen Hochgeschwindigkeitsbus sowie einen separaten Bus für die Anbindung von Peripherie.

PLB (*Processor Local Bus*) ist als Kommunikationsbus für Anwendungen mit hohem Bandbreitenbedarf entwickelt worden [164]. PLB verfügt über einen 32-bit-Adressbus sowie getrennte Schreib- und Lesebusse mit einer Breite von 16 bit bis 128 bit. Ein Buszugriff ist in eine Adress- und eine Datenphase aufgeteilt [37]. In der Adressphase sendet der Master eine Anfrage an den Bus-Arbiter. Sobald der Arbiter dem Master Zugriff auf den Bus gewährt, wird die Adresse der Anfrage an den entsprechenden Slave weitergeleitet. Der Slave bestätigt den Erhalt der Adresse über ein *Acknowledge*-Signal. Die über den Schreibkanal übertragenen Daten werden vom Slave ebenfalls bestätigt. Wie bei AMBA AXI muss bei einem Burst-Zugriff nur die Adresse des ersten Datenwortes übertragen werden. Die Länge eines Burst-Zugriffs ist nicht beschränkt. Damit ein Master bei einem Burst-Zugriff den Bus nicht zu lange blockiert, kann ein Burst nach einer einstellbaren Takt-Anzahl unterbrochen werden [153, S. 70]. Dies stellt

[37] *Serial Peripheral Interface*

eine kurze Latenz für Anfragen mit hoher Priorität sicher. PLB unterstützt Registerstufen, *Split Transactions* und *Transaction Reordering* sowie Cache-Kohärenz [164, S. 17]. Viele PowerPC-CPUs von IBM verfügen über eine PLB-Schnittstelle, der IBM PowerPC-476FP-Prozessor bietet beispielsweise eine Cache-kohärente PLB6-Schnittstelle [80].

Der CoreConnect-Standard sieht für langsame Peripherie den **On-Chip Peripheral Bus (OPB)** vor [147]. Obwohl OPB ähnliche Einsatzbereiche wie der AMBA APB hat, besitzt er eine höhere Leistungsfähigkeit. Es werden mehrere Master, Burst-Zugriffe sowie Datenbreiten von 16 bit bis 64 bit unterstützt. Bei Burst-Zugriffen muss jedoch im Gegensatz zum PLB die Adresse von jedem Datenwort übertragen werden [153, S. 72]. Sowohl OPB als auch PLB erlauben Master und Slaves mit unterschiedlichen Datenbreiten an einer Verbindungsstruktur.

OCP (*Open Core Protocol*)

Ursprünglich von der *Open-Core-Protocol International Partnership* (OCP-IP) entwickelt, wurde der OCP-Standard 2013 von der Accellera Systems Initiative [2] übernommen. Ziel von OCP ist die Vereinheitlichung der Schnittstellen von IP-Komponenten in mikroelektronischen Systemen. OCP definiert ausschließlich Schnittstellen, der Standard trifft keine Aussagen über Implementierungsvarianten von OCP-kompatiblen Verbindungsstrukturen. OCP unterstützt das Pipelining von Bus-Anfragen. Bei Burst-Zugriffen kann entweder nur die Adresse vom ersten Datenwort oder die Adressen von allen Datenwörtern übertragen werden.

Die vielfältige Konfigurierbarkeit macht es schwer, IP-Kerne aus verschiedenen Quellen in einem System zu integrieren. Aus diesem Grund definiert die OCP-Spezifikation eine Reihe von Profilen, um die Entwicklung von OCP-kompatiblen IP-Kernen zu vereinfachen und die Interoperabilität dieser zu erhöhen [148, S. 319]. Zudem ist es möglich, Übersetzer (*Bridges*) zu anderen Standards wie AXI oder PLB zu entwerfen. Gleichzeitig schränken die Profile die hohe Flexibilität von OCP stark ein. Ähnlich wie bei den anderen hier vorgestellten Bus-Standards AMBA APB und CoreConnect OPB existiert ein Profil für einfache Slaves mit hoher Latenz (Peripherie-Bus, *Simple Slave*). Das *High-Speed*-Profil für Anwendungen mit hohem Bedarf an Übertragungsbandbreite erlaubt Burst-Zugriffe und ausstehende Transaktionen [173, S. 9]. Ein Profil für sehr hohe Anforderungen (*Advanced-High-Speed*) besitzt erweiterte Burst- sowie Synchronisationsunterstützung [148, S. 330]. Es existiert eine OCP-Erweiterung für Cache-kohärente Systeme [148, S. 77]. Zudem spezifiziert OCP feste Schnittstellen für die Verifikation [204] und das Testen von IP-Blöcken sowie asynchrone Kontrollsignale wie Interrupts oder Reset. Der MIPS 1074K CPU-Kern besitzt eine OCP-konforme, Cache-kohärente Bus-Schnittstelle [80].

STMicroelectronics STBus

Der von STMicroelectronics [182] entwickelte STBus besteht aus drei einzelnen Standards [181]. Anwendungsgebiet sind eingebettete Systeme aus dem Bereich der Unterhaltungs- und Haushaltselektronik [153, S. 73].

STBus Typ1 ist für die Anbindung von Peripherie ausgelegt und wird daher auch *Peripheral*-STBus genannt. Adressen besitzen eine Breite von 32 bit, die Datenbreiten kann zwischen 8 bit und 64 bit betragen. Es werden Bursts unterstützt, der Master muss jedoch die Adressen von jedem Datenwort an den Bus anlegen. Es ist optional möglich, dass der Slave Fehlerinformationen an den Master zurückgibt [181, S. 17].

STBus Typ2 (*Basic*) erweitert STBus Typ1 um die Möglichkeit, Registerstufen einzufügen. Die Datenbreite kann bis zu 256 bit betragen, Burst-Zugriffe können unterbrochen werden (*Split Transaction*). Es ist möglich, verschiedenen Mastern unterschiedliche Prioritäten zuzuweisen. Dies ist statisch zur Entwurfszeit oder dynamisch zur Laufzeit möglich [153, S. 76]. Atomare Operationen wie *Read-Modify-Write* vereinfachen die Synchronisierung in Multiprozessorsystemen.

STBus Typ3 (auch *Advanced* genannt) erweitert STBus Typ2 um *Out-of-Order*-Transaktionen. Ähnlich wie AXI ermöglicht Typ3 einen erweiterten Burst-Modus, bei dem nur einmal die Adresse für eine komplette Burst-Anfrage an den Slave gesendet werden muss. Zudem wird bei einem Schreib-Burst nur das letzte Datenwort vom Slave bestätigt [181, S. 48].

Der STBus-Standard erlaubt die Verwendung von Typ1-, Typ2- und Typ3-Komponenten an einem Bus, Typ-Konverter übersetzen zwischen den verschiedenen Bus-Standards. Ebenso kann zwischen Komponenten verschiedener Datenbreite oder Taktfrequenz vermittelt werden [153, S. 78]. In [130] wird ein L2-Cache mit STBus Typ3-Schnittstellen vorgestellt (siehe Abschnitt 2.3.3).

OpenCores Wishbone

Der *Open-Source*-Verbindungsstandard Wishbone [210] wurde vom OpenCores-Projekt [149] spezifiziert. Die Breite der Adressen ist nicht explizit festgelegt und beträgt zumeist 32 bit. Lese- und Schreibdaten können eine Breite von 8 bit bis 64 bit aufweisen. Wishbone unterstützt Burst-Zugriffe unbeschränkter Länge, hierbei muss die Adresse jedes Datenworts vom Master an den Bus angelegt werden. Die Topologie der Verbindungsstruktur ist nicht vorgegeben, der Standard nennt Punkt-zu-Punkt, geteilter Bus und Crossbar als Beispiele.

In der Revision B4 vom Wishbone-Standard [210] werden zwei verschiedene Schnittstellen beschrieben. *Wishbone-Classic* ist fast vollständig synchron aufgebaut. Das einzige asynchrone Signal dient der Bestätigung von Zugriffen auf einen Slave. Hierbei führt ein kombinatorischer Pfad vom Master zum Slave und zurück zum Master. Der asynchrone Pfad begrenzt die maximale Taktfrequenz vom Bus [210, S. 66], Wishbone-Classic weist bei geringen Taktfrequenzen allerdings eine hohe Ressourceneffizienz auf. Der *Wishbone-Registered-Feedback*-Bus trennt den kombinatorischen Pfad über ein Register auf, dies führt jedoch zu einem Takt zusätzlicher Latenz. Um diesen Nachteil abzuschwächen, wird der Burst-Modus im Vergleich zu Wishbone-Classic erweitert. Hierbei kann bei einem Burst in jedem Takt ein Datenwort übertragen werden. Das Ende von einem Burst-Zugriff wird dem Slave über ein spezielles Signal mitgeteilt.

Eine zusätzliche Erweiterung vom Wishbone-Standard erlaubt die Einführung von Registern (Pipelining) zwischen Master und Slave. Hierdurch kann die maximale Taktfrequenz der Verbindungsstruktur gesteigert werden. Der Master muss Adressen und Daten nur einen Takt lang an den Bus anlegen, im nächsten Takt kann bereits der nächste Zugriff erfolgen. Um die Synchronisierung zwischen Master und Slave sicherzustellen, wird ein zusätzliches *Stall*-Signal eingeführt. Mit diesem Signal kann der Slave dem Master mitteilen, dass er gerade keine Anfragen entgegennehmen kann. Ein Nachteil von zusätzlichen Registerstufen ist die höhere Latenz von Buszugriffen. Wishbone-Classic und Wishbone-Registered-Feedback existieren jeweils in einer Ausprägung mit und ohne Pipelining. Der Wishbone-Standard hat somit vier verschiedene Ausprägungen. Eine Umsetzung von einer Implementierungsvariante auf eine andere ist jedoch möglich. So ist im Standard beispielsweise beschrieben, wie ein Registered-Feedback-Master mit einem Classic-Slave kommunizieren kann. Der Wishbone-Standard schreibt vor, dass jeder Wishbone-kompatible IP-Kern ein Datenblatt mit Informationen über die Bus-Schnittstelle enthalten muss [210, S. 25].

Der OpenRISC 1000-Prozessor verfügt über eine Wishbone-Schnittstelle. Auf der Website des OpenCores-Projekts [149] sind etliche weitere freie Prozessorkerne sowie Peripherie mit Wishbone-Schnittstelle unter freien Lizenzen veröffentlicht. Im Rahmen dieser Arbeit ist eine Wishbone-kompatible Verbindungsstruktur entstanden. Eine Entwurfsraumexploration dieser Verbindungsstruktur ist in Abschnitt 5.5.3 zu finden.

2.4.2 Implementierungen von Verbindungsstrukturen

Im Folgenden werden einzelne Verbindungsstrukturen vorgestellt, die Schnittstellen für einen oder mehrere der genannten Bus-Standards bieten. Die Hersteller vieler kommerzieller Multiprozessoren verwenden selbst entwickelte Verbindungsstrukturen und veröffentlichen keine Details über verwendete Bus-Standards oder Funktionsumfang.

ARM vertreibt unter dem Namen CoreLink CCN (*Cache Coherent Network*) mehrere On-Chip-Verbindungsstrukturen für Cache-kohärente Multiprozessoren mit bis zu 48 ARM-CPUs [31; 41; 42]. Vier Prozessoren sind jeweils zu einem Cluster zusammengefasst, zusätzlich enthält dieser private L1-Caches sowie einen gemeinsamen L2-Cache. Innerhalb eines MPSoCs können Cluster mit verschiedenen ARM-CPUs eingesetzt werden. Dies ermöglicht beispielsweise das *Big-Little*-Konzept, bei dem typischerweise zwei CPU-Cluster mit unterschiedlichen ARM-CPUs auf einem MPSoC integriert werden [74; 197]. Hierbei besteht ein Cluster beispielsweise aus hochperformanten ARM Cortex-A72-CPUs, ein zweiter Cluster verwendet energieeffiziente Cortex-A53-CPUs [43]. An das Verbindungsnetzwerk können bis zu vier Speichercontroller sowie bis zu 24 weitere Komponenten (z. B. GPU, DSP[38] oder Hardwarebeschleuniger) per AXI- bzw. AXI-Lite-Schnittstelle angeschlossen werden. CCN-508 und CCN-512 integrieren bis zu 32 MB L3-Cache [42]. Um Busteilnehmer mit unterschiedlichen Anforderungen an Durchsatz

[38] *Digitaler Signalprozessor*

und Latenz effizient im gleichen Bus verwenden zu können, besitzt CCN QoS-Funktionen. Byrne vermutet in [31], dass CCN-504 eine partielle Crossbar als Topologie verwendet, ARM veröffentlicht zur Topologie von CCN keine Informationen. Der CCN implementiert viele, jedoch nicht alle optionalen Erweiterungen des AXI-Standards. So erlaubt der Standard beispielsweise Datenbreiten von bis zu 1024 bit, die CCN-Implementierungen verwenden nur Datenbreiten bis maximal 256 bit [42].

Xilinx vertreibt für seine konfigurierbaren Hardwarebausteine (*Field Programmable Gate Array*, FPGA) eine AXI-kompatible Verbindungsstruktur [128]. Die Verbindungsstruktur ist nicht kompatibel zu EDA[39]-Software anderer Hersteller und somit auf Xilinx-FPGAs als Zielplattform beschränkt. Es ist möglich, AMBA AXI4-, AXI4-Lite- sowie AXI3-Komponenten anzubinden. Es werden bis zu 16 Master und 16 Slaves unterstützt. Master und Slaves können nur Schreib- oder Leseports besitzen, um die Ressourceneffizienz zu erhöhen. AXI3- und AXI4-Komponenten können Datenbreiten von 32 bit bis 1024 bit besitzen, AXI4-lite unterstützt 32 bit und 64 bit breite Datenbusse. Der Adressbus kann eine Breite von bis zu 64 bit aufweisen. Alle fünf AXI-Kanäle können als geteilter Bus sowie als volle oder partielle Crossbar konfiguriert werden. Die Verbindungsstruktur kann optional Out-of-Order-Transaktionen unterstützen. Eine Arbitrierung ist durch das Round-Robin-Verfahren oder statische Prioritäten möglich. Es stehen verschiedene weitere Funktionen wie FIFOs, Registerstufen sowie Konverter für Datenbreite und Taktfrequenz zur Verfügung [128, S. 5]. Xilinx bietet viele weitere IP-Blöcke mit AXI-Schnittstelle für seine FPGAs an. Diese sind teilweise kostenlos, teils kostenpflichtig. Im Rahmen dieser Arbeit wird beispielsweise ein CAN[40]-Controller von Xilinx mit AXI-Schnittstelle [127] eingesetzt (siehe Abschnitt 3.3.2 sowie Abschnitt 6.1.3).

Altera stellt ähnlich wie Xilinx eine On-Chip-Verbindungsstruktur für seine FPGA-basierten Produkte bereit. Qsys [167] stellt Schnittstellen für AMBA AXI3, AXI4, APB sowie das von Altera spezifizierte Avalon [16] bereit. Die Anzahl an Master und Slaves ist nicht begrenzt. Die Breite des Adressbus kann bis zu 64 bit betragen. Es sind dedizierte, entkoppelte Übertragungskanäle von den Mastern zu den Slaves sowie von den Slaves zu den Mastern vorhanden. Eine Qsys-Verbindungsstruktur kann vollständig kombinatorisch aufgebaut sein. Dies führt zu einer Schreiblatenz von null Takten und einer Leselatenz von einem Takt [11]. Registerstufen können verwendet werden, um die maximale Taktfrequenz der Verbindungsstruktur zu steigern. Die Topologie von der Verbindungsstruktur hängt von der Konfiguration der angeschlossen Master und Slaves ab. Die Datenbreite der Verbindungsstruktur ergibt sich beispielsweise aus der maximalen Datenbreite aller angeschlossenen Busteilnehmer.

Der PLB Bus Controller [160] von **IBM** verbindet bis zu 16 Master- und acht Slave-Segmente nach dem CoreConnect PLB6-Standard. Jedes Slave-Segment kann bis zu vier Slaves anbinden. Bis zu acht Master und ein Slave-Segment können Cache-kohärent verbunden sein. Die getrennten Schreib- und Lesebusse sind jeweils 128 bit

[39] *Electronic Design Automation*
[40] *Controller Area Network*

breit. Es sind Out-of-Order-Lesezugriffe sowie das Pipelining von Buszugriffen möglich. Treten Verklemmungen von Buszugriffen auf, werden diese erkannt und von der Verbindungsstruktur aufgelöst.

FlexNoC von **Arteris** [14] ist eine lizenzierbare, paketbasierte Verbindungsstruktur. Die Schnittstellen der Master und Slaves können kompatibel zu den Standards OCP sowie AMBA AHB, AXI und APB sein. Zudem wird die PIF[41]-Schnittstelle von Tensilica-Prozessoren unterstützt. FlexNoC übersetzt die verschiedenen Schnittstellenstandards in ein internes, paketbasiertes Protokoll. Die Datenbreite kann 8 bit bis 256 bit betragen. Die Anzahl an Busteilnehmern ist nicht begrenzt. Optional unterstützt FlexNoC Takt- und Spannungsabschaltung von Busteilnehmern sowie DVFS. Zudem ist die Verwendung von statischen und dynamischen Prioritäten für die Arbitrierung von Busteilnehmern möglich. FlexNoC wird unter anderem in MPSoCs von Qualcomm, Altera und Samsung eingesetzt. Das geistige Eigentum an FlexNoC sowie zahlreiche Arteris-Mitarbeiter wurden 2013 von Qualcomm übernommen [46]. Arteris vertreibt jedoch weiterhin FlexNoC-Produkte und gewinnt neue Kunden [81]. Ein Beispiel hierfür ist der Bildverarbeitungs-SoC[42] EyeQ4 von Mobileye [45]. Die einzelnen IP-Blöcke des EyeQ4-SoCs verfügen über OCP-konforme Schnittstellen und sind über eine NoC von Arteris verbunden.

Sonics [83; 177] vertreibt verschiedene lizenzierbare On-Chip-Verbindungsstrukturen. Das aktuellste Produkt SonicsGN [178] bietet native Unterstützung für AMBA AXI und ACE sowie OCP. AMBA AHB und APB können mittels Protokollübersetzer verwendet werden. Ähnlich dem FlexNoC von Arteris verwendet SonicsGN intern ein paketbasiertes Protokoll. Es besteht Unterstützung für DVFS, Takt- und Spannungsabschaltung von Busteilnehmern sowie Out-of-Order-Transaktionen. Produkte von Sonics werden unter anderem von Broadcom, Fujitsu, Samsung und Toshiba eingesetzt [83].

Die DesignWare-Bibliothek von **Synopsys** [187] enthält verschiedene AMBA-Verbindungsstrukturen und IP-Blöcke wie beispielsweise DMA-Controller, Protokollübersetzer oder Zeitgeber. Die fortschrittlichste Verbindungsstruktur DW_AXI [47] unterstützt die Protokolle AMBA AXI und ACE-Lite. DW_AXI erlaubt die Verwendung von jeweils bis zu 16 Mastern und Slaves, 64-bit-Adressen sowie eine Datenbreite von bis zu 512 bit. Alle Master und Slaves müssen über die gleiche Datenbreite verfügen und mit der gleichen Taktfrequenz betrieben werden. DW_AXI unterstützt Out-of-Order-Transaktionen. Zur Steigerung der maximalen Taktfrequenz können Registerstufen in die Verbindungsstruktur eingefügt werden.

2.4.3 Stand der Technik

In der Literatur sind sowohl Vergleiche von verschiedenen Bus-Standards als auch von verschiedenen Architekturen bzw. Topologien eines einzelnen Standards zu finden.

[41] *Processor Interface*
[42] *System on a Chip*

Kumar et al. führen in [115] eine Entwurfsraumexploration eines MPSoCs mit 4, 8 und 16 Power4-ähnlichen CPUs durch. Als Topologien werden ein geteilter Bus, eine Crossbar sowie eine hierarchische Verbindungsstruktur bestehend aus zwei geteilten Bussen betrachtet. Da ein Autor der Arbeit bei IBM beschäftigt ist, ist zu vermuten, dass IBM CoreConnect als Bus-Standard verwendet wird. Es wird gezeigt, dass die Topologie der Verbindungsstruktur einen hohen Einfluss auf die Architektur des MPSoCs hat. Beispielsweise erlaubt der verringerte Flächenbedarf des geteilten Busses die Verwendung von größeren Caches innerhalb der CPUs. Dies erhöht die Performanz des Systems im Vergleich zu den anderen Topologien mit kleineren Caches.

Angiolini et al. [8; 9] vergleichen einen geteilten Bus und eine partielle Crossbar nach dem AMBA AHB-Standard mit einem NoC. Das MPSoC besteht aus 30 Komponenten mit insgesamt 15 Master- und 15 Slave-Ports. Die partielle Crossbar integriert fünf geteilte Busse. Das NoC besteht aus 15 Knoten, jeder Knoten integriert jeweils einen Master und einen Slave. Für die betrachteten Anwendungen bietet der geteilte Bus eine zu geringe Übertragungsbandbreite. In Bezug auf Flächen- und Energiebedarf zeigt die partielle Crossbar Vorteile gegenüber dem NoC.

Salminen et al. [170] vergleichen verschiedene Bus-Standards auf Protokollebene. Alle betrachteten Standards weisen eine große Ähnlichkeit auf, einzelne Standards zeigen lediglich in speziellen Anwendungsgebieten Vorteile. Die Autoren empfehlen eine Entwurfsraumexploration, um die Bus-Konfiguration mit der höchsten Ressourceneffizienz für eine spezifische Anwendung zu finden. Zum Zeitpunkt der Entstehung der vorliegenden Arbeit hat sich AMBA AXI als De-facto-Standard durchgesetzt.

Loghi et al. stellen in [125] einen MPSoC mit vier ARM-Kernen sowie Verbindungsstrukturen nach den Standards AMBA AHB und STBus Typ3 gegenüber. Hierzu wird eine SystemC-basierte Simulationsumgebung verwendet. Die AMBA AHB-Implementierung verwendet einen geteilten Bus. Die Implementierungen vom STBus bietet einen geteilten Bus, zwei partielle sowie eine volle Crossbar als Topologie. Der Vergleich der beiden Varianten mit geteiltem Bus zeigt einen kleinen Vorteil für den STBus in Bezug auf die Ausführung von Beispielanwendungen. Die Crossbar-Implementierungen zeigen einen deutlichen Performanzvorteil bei Anwendungen mit hoher Auslastung der Verbindungsstruktur. Dieses Ergebnis ist aufgrund der deutlich erhöhten maximalen Übertragungsbandbreite der Crossbar zu erwarten. Loghi et al. betonen den hohen Einfluss von schnellen gemeinsamen Speichern auf die Performanz des Gesamtsystems.

Das et al. präsentieren in [39] ein MPSoC mit 32 CPUs und einer hierarchischen Verbindungsstruktur. In einem CPU-Cluster sind jeweils vier CPUs und vier L2-Cache-Bänke über einen geteilten Bus verbunden. Die insgesamt acht CPU-Cluster kommunizieren über ein NoC mit *Mesh*-Topologie. Im Vergleich mit drei rein NoC-basierten MPSoCs zeigt das hierarchische MPSoC eine Verbesserung des Energie-Zeit-Produktes für verschiedene Anwendungen um durchschnittlich 70 % (Mesh-NoC), 30 % (*Concentrated-Mesh*-NoC) und 22 % (*Flattened-Butterfly*-NoC). Die NoC-Konfigurationen verfügen über insgesamt 64 Knoten, wobei jeweils 32 Knoten eine CPU bzw. L2-Cache enthalten. Die Performanz des Gesamtsystems kann durch die Verwendung der hierarchischen Verbindungsstruktur um bis zu 14 % gesteigert werden. Ein Vorteil der hierarchischen Verbindungsstruktur ist eine Verringerung der durchschnittlichen Latenz der Kommunikation. Innerhalb eines CPU-Clusters beschränkt der geteilte Bus jedoch den Durchsatz. Im CPU-Cluster des CoreVA-MPSoCs kann statt eines geteilten Bus eine partielle oder vollständige Crossbar verwendet werden. Dies erlaubt eine Anpassung der Übertragungsbandbreite im CPU-Cluster an die Anforderungen des jeweiligen Anwendungsszenarios (siehe Abschnitt 5.5).

2.4.4 Bus-Standards im Vergleich

Die vorgestellten Bus-Protokolle unterscheiden sich deutlich in Bezug auf Komplexität, Leistungsfähigkeit und Ressourcenbedarf. In Tabelle 2.5 werden wichtige Eigenschaften der in diesem Kapitel betrachteten Bus-Standards zusammengefasst. Die optimalen Erweiterungen der einzelnen Standards erschweren jedoch einen Vergleich. Innerhalb der verschiedenen Protokollfamilien wie ARM AMBA oder IBM CoreConnect existieren Protokolle verschiedener Komplexität. Einfache Busse wie AMBA APB, CoreConnect OPB und STBus Typ1 werden für die Anbindungen von langsamer Peripherie wie UART-Schnittstellen oder Zeitgeber verwendet. Hochperformante Busse (z. B. AMBA APB, AXI; CoreConnect PLB) sind für einen hohen Bandbreitenbedarf von Prozessoren, Caches oder DMA-Controllern ausgelegt. Mechanismen wie Burst-Zugriffe oder die Verwendung von (partiellen) Crossbars und Registerstufen ermöglichen es, die maximale Übertragungsbandbreite gegenüber einfachen Bussen drastisch zu erhöhen.

Eingebettete Multiprozessoren integrieren zunehmend IP-Blöcke aus vorangegangenen Projekten oder von Drittfirmen. Die Bus-Schnittstellen dieser IP-Blöcke sind häufig mit unterschiedlichen Bus-Standards kompatibel. Um verschiedene Bus-Standards in einen MPSoC integrieren zu können, ist der Einsatz von Konverter-Blöcken erforderlich. Alternativ können Verbindungsstrukturen wie SonicsGN oder Arteris FlexNoC verwendet werden, die nativ verschiedene Standards unterstützen. Die Interoperabilität von

Tabelle 2.5: Überblick über Standards von On-Chip-Verbindungsstrukturen

Name	Hersteller	Master	Burst	Datenbreite [bit]
AMBA-AHB	ARM	∞	ja	8 – 1024
AMBA-AXI	ARM	∞	ja	8 – 1024
AMBA-APB	ARM	1	nein	32
CoreConnect PLB	IBM	16	ja	16 – 128
CoreConnect OPB	IBM	1	nein	32 – 64
OCP	Accellera	∞	ja	beliebig
Wishbone	OpenCores	∞	ja	8 – 64
STBus Typ1	STMicroelectronics	∞	ja	8 – 64
STBus Typ2,Typ3	STMicroelectronics	∞	ja	8 – 256

IP-Blöcken und Verbindungsstrukturen aus verschiedenen Quellen ist oft nicht gegeben und bedingt aufwendige Tests [204]. Dies ist auf die hohe Konfigurierbarkeit der einzelnen Standards sowie auf die Verwendung von verschiedenen Bus-Standards innerhalb eines MPSoCs zurückzuführen.

AMBA, CoreConnect und STBus wurden von einzelnen Firmen spezifiziert und sind daher teilweise auf die Anforderungen der jeweiligen Firmen ausgerichtet. Im Gegensatz dazu werden OCP und Wishbone von nicht-kommerziellen Organisationen spezifiziert und verwaltet. Ungeachtet dieser Tatsache hat sich AMBA AXI in den letzten Jahren zu einem Quasi-Industrie-Standard für MPSoC-Komponenten entwickelt. AXI bietet – bei Verwendung aller optionalen Erweiterungen – die höchste Leistungsfähigkeit aller hier verglichenen Bus-Standards. Gleichzeitig erlaubt die hohe Konfigurierbarkeit, die Architektur der Verbindungsstruktur an den Anwendungsfall anzupassen und so eine hohe Ressourceneffizienz zu erreichen. Der FPGA-Hersteller Xilinx verwendet seit 2011 AXI als Verbindungsstandard für FPGA-basierte SoC-Produkte mit Microblaze oder ARM Cortex-A9-Prozessoren [85]. Für FPGAs bis zur Virtex-6- und Spartan-6-Serie unterstützte Xilinx alternativ IBM CoreConnect mit den Bussen PLB und OPB, die FPGAs ab der 7er-Serie unterstützen ausschließlich AXI [17]. ARM-Prozessoren besitzen eine weite Verbreitung in eingebetteten Systemen und ersetzen zunehmend PowerPC- und MIPS-CPUs. Dementsprechend steigt auch die Verbreitung des AMBA AXI-Standards. In Netzwerkprozessoren werden PowerPC-CPUs mit CoreConnect-Schnittstelle durch ARM-CPUs verdrängt [31]. Intel verwendet in Mikrocontrollern der Quark-Familie AMBA zur Anbindung von externen Schnittstellen [79; 209].

Aufgrund der großen Verbreitung von ARM-basierten CPU-Kernen hat sich AXI zum Quasi-Industrie-Standard für MPSoC-Komponenten entwickelt. Die kommerziellen Verbindungsstrukturen von Arteris, Sonics, Synopsis und Xilinx unterstützen AXI. Daher ist es sinnvoll, AXI als Bus-Standard im Rahmen dieser Arbeit zu evaluieren. AXI verspricht zudem eine hohe Performanz bei der Anbindung eines CPU-Clusters an das On-Chip-Netzwerk des CoreVA-MPSoCs [214], da durch die insgesamt fünf dedizierten Kanäle parallel geschrieben und gelesen werden kann. Als Beispiel für einen Bus-Standard mit geringerem Ressourcenbedarf wird zudem eine Wishbone-Implementierung betrachtet. Wishbone zeichnet sich durch eine große Anzahl an frei verfügbaren IP-Blöcken aus und wird in einer Vielzahl von wissenschaftlichen Veröffentlichungen sowie in vielen quelloffenen SoC-Projekten verwendet. Im Vergleich zu AXI ist jedoch ein geringerer maximaler Durchsatz bei gleicher Datenbreite zu erwarten.

2.4.5 Arbiter

Greifen mehrere Systemkomponenten zeitgleich auf die selbe Ressource zu, ist eine Zugriffsverwaltung (Arbitrierung) erforderlich. Im Kontext von On-Chip-Verbindungsstrukturen können beispielsweise mehrere CPUs zeitgleich eine Anfrage für den Zugriff auf einen geteilten Bus oder denselben gemeinsamen Speicher stellen. Die Zugriffsverwaltung wird in diesem Fall von einer Hardwareeinheit, dem sogenannten Arbiter, durchgeführt [86, S. 237 f.]. Es muss entschieden werden, welche CPU Zugriff auf die Ressource erhält. Die Anfragen der übrigen CPUs werden zurückgestellt (*Stall*), bis der Zugriff der ausgewählten CPU abgeschlossen ist.

In der Literatur werden eine Vielzahl an Verfahren beschrieben, um Zugriffe auf einen Bus zu arbitrieren [153, S. 26 ff.; 170]. Ein wichtiges Bewertungsmaß dieser Verfahren ist die Fairness, also die Frage, ob alle Master gleichberechtigt Zugriff auf die Verbindungsstruktur bzw. einen Slave bekommen (siehe auch Abschnitt 2.6). Weitere Bewertungsmaße sind die Verzögerungszeit sowie der Flächenbedarf des Arbiters. Bei der Verwendung von **statischen Prioritäten** wird jedem Master eine Priorität zugeordnet. Der Master mit der höchsten Priorität erhält immer Zugriff auf den Bus. Ein Nachteil dieses Verfahren ist, dass Master mit geringer Priorität bei einer hohen Bus-Auslastung keinen Zugriff auf den Bus bekommen und somit „verhungern" können.

Beim **Zeitschlitze-Verfahren** (*Time Division Multiple Access*, TDMA) werden jedem Master ein oder mehrere Zeitabschnitte zugeteilt, in dem sie auf den Bus zugreifen dürfen. Greift der Master in dem ihm zugeteilten Zeitschlitz nicht auf dem Bus zu, verfällt der Zeitschlitz und der Bus wird nicht verwendet. Es kann also vorkommen, dass der Bus nicht verwendet wird, obwohl ein anderer Master in diesem Moment auf einen Slave zugreifen möchte.

Weit verbreitet ist das Rundlauf-Verfahren (**Round-Robin**), bei dem alle Master nacheinander die Möglichkeit erhalten, auf den Bus zuzugreifen [153, S. 26 f.]. Hat ein Master Zugriff erhalten, wird er erst wieder berücksichtigt, wenn alle anderen Master die Möglichkeit eines Zugriffs hatten. Dies vermeidet das „Verhungern" von Busteilnehmern. Es ist jedoch ebenfalls möglich, dass ein Master ununterbrochen Zugriff auf den Bus erhält, wenn die anderen Master keine Anfragen stellen. Das Round-Robin-Verfahren ist vor allem für Systeme geeignet, bei denen alle Master über die gleiche Priorität verfügen.

Hierarchische Arbitrierungsverfahren bestehen aus mehreren Arbitern, also beispielsweise einer Kombination aus Zeitschlitzen und Round-Robin-Arbitrierung [153, S. 27]. Verwendet ein Master den ihm zugeteilten Zeitschlitz nicht, wird im Round-Robin-Verfahren ein Master bestimmt, der auf den Bus zugreifen darf.

Im CPU-Cluster des CoreVA-MPSoCs werden Arbiter in den Implementierungen der AXI- und Wishbone-Verbindungsstrukturen (siehe Abschnitt 2.5) sowie im gemeinsamen L1-Datenspeicher (siehe Abschnitt 2.3.2) eingesetzt. In dieser Arbeit wird eine Round-Robin-Arbitrierung verwendet, da dieses Verfahren eine geringe Komplexität und (lokale) Fairness besitzt [58]. Aus diesem Grund kommt das Round-Robin-Verfahren in sehr vielen MPSoCs und Verbindungsstrukturen zum Einsatz [52; 143; 157; 165; 168]. Die Evaluierung von hierarchischen Arbitrierungsverfahren mit einer Priorisierung einzelner CPUs des CoreVA-MPSoCs erscheinen im Rahmen von weiterführenden Arbeiten sinnvoll. Allerdings ist hierzu insbesondere eine Erweiterung des CoreVA-MPSoC-Compilers (siehe Abschnitt 3.2.3) notwendig, um bestimmen zu können, welche CPUs eines Clusters priorisiert werden müssen.

2.5 Der CPU-Cluster des CoreVA-MPSoCs

Nachdem in den vorangegangenen Abschnitten mögliche Bestandteile eines eng gekoppelten CPU-Clusters beschrieben wurden, wird im Folgenden die Architektur des CPU-Clusters des CoreVA-MPSoCs vorgestellt (siehe Abbildung 2.9). Als zentrale Verarbeitungseinheiten können bis zu 128 CoreVA-CPUs integriert werden. Die Anzahl an CPUs ist durch die Adressaufteilung beschränkt (siehe Tabelle 2.6) und kann bei Bedarf erhöht werden. Jede CPU besitzt jeweils bis zu 1 MB lokalen Scratchpad-Speicher bzw. Cache für Instruktionen und Daten. Zusätzlich können bis zu 32 eng an die CPU gekoppelte Hardwareerweiterungen verwendet werden. Jede Erweiterung kann einen maximalen Adressraum von 1 MB ansprechen. 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 Scratchpad-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. Das generische Protokoll erlaubt eine Übersetzung auf verbreitete Bus-Standards und wird in [241, S. 5 f.] detailliert beschrieben.

Im Rahmen dieser Arbeit 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 Takte. Durch die Verwendung von Registerstufen steigt die Latenz entsprechend an. Alle Komponenten eines Clusters teilen sich einen 32-bit-Adressraum. Die Arbitrierung von

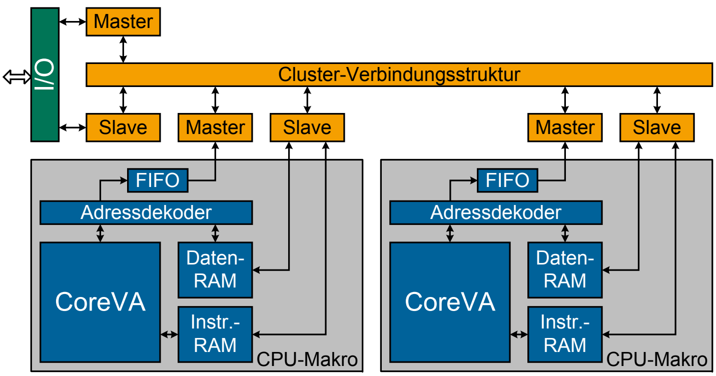


Abbildung 2.9: Blockschaltbild eines CPU-Clusters mit zwei CPUs und I/O-Schnittstelle

Tabelle 2.6: Adresstabelle des CoreVA-MPSoC-CPU-Clusters

Name	Basisadresse	Größe [MB]
L1-Speicher oder L1-Caches	0x00000000	128
32 CPU-interne HW-Erweiterungen	0x08000000	32x1
Gemeinsamer L2-Speicher	0x10000000	128
NoC-Cluster-Schnittstelle (NI)	0x18000000	128
Externer Zugriff auf 128 CPUs (Lokale Speicher + 6 HW-Erweiterungen)	0x20000000	128x8
Eng gekoppelter gemeinsamer Datenspeicher	0x60000000	128
DMA-Controller	0x70000000	1
Off-Chip-Schnittstelle	0x80000000	2048

Buszugriffen wird durch das Round-Robin-Verfahren 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 [238], eine Off-Chip-Schnittstelle sowie eine Schnittstelle zum On-Chip-Netzwerk [214] des CoreVA-MPSoCs erweitert werden.

Als weitere Komponente kann ein eng gekoppelter gemeinsamer L1-Speicher in den CPU-Cluster integriert werden. Als Topologie für die dedizierte Verbindungsstruktur kann eine Crossbar oder ein 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 L1-Speichers beträgt 128 MB.

2.6 Synchronisierung und Kommunikationsmodelle

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 [200, S. 96]. 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 [90, S. 237 f.; 155, S. 137 f.].

Eine Synchronisierung kann auf verschiedenen Blockgrößen oder – je nach Sichtweise – Abstraktionsebenen erfolgen. 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 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 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.

2.6.1 Mutex

Das Verfahren des wechselseitigen Ausschlusses (auch **Mutex**[43] oder *Lock* genannt) bezeichnet einen Synchronisierungsmechanismus, bei dem gleichzeitig nur ein Prozess den Mutex besitzen bzw. einen **kritischen Abschnitt** betreten kann [200, S. 97]. Hierdurch wird der Zugriff auf eine geteilte Ressource serialisiert [114, S. 293]. Allen Implementierungen ist gemein, dass Methoden zum Erwerb (*Acquire*) und zur Freigabe (*Release*) des Mutex' existieren. 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 [200, S. 97]. Mutex-Algorithmen können die folgenden Eigenschaften besitzen [189, S. 252; 231, S. 18]:

- Durch **wechselseitigen Ausschluss** muss sichergestellt werden, dass immer nur ein Prozess Zugriff auf den kritischen Abschnitt bekommen kann.
- Verklemmungen können auftreten, wenn zwei Prozesse je einen Mutex besitzen und den jeweils anderen Mutex erwerben möchten. Um **Verklemmungsfreiheit** sicherzustellen, muss es möglich sein, Anfragen an den Mutex in einer anderen Reihenfolge abzuarbeiten, als diese gestellt worden sind.
- Bei der **Starvationsfreiheit** muss sichergestellt werden, dass jeder Prozess immer nur endlich lange auf einen Mutex warten muss.

[43] *Mutual Exclusion*

- Eine Mutex-Implementierung ist **fair**, wenn alle Prozesse gleichberechtigt Zugriff auf einen Mutex erhalten. Fairness spielt auch bei der Bus-Arbitrierung eine entscheidende Rolle, siehe Abschnitt 2.4.5.

Tanenbaum und van Steen [189, S. 252 ff.] unterscheiden mehrere Klassen von Mutex-Verfahren. Bei *Token*-basierten Verfahren wird eine Nachricht (Token) von einem Prozess zum nächsten weitergereicht [114, S. 293]. Möchte ein Prozess auf den kritischen Abschnitt zugreifen, wartet er auf den Token und behält ihn so lange, bis er den kritischen Abschnitt verlässt. Dieser einfache Ansatz hat vor allem bei Anwendungen mit vielen Prozessen, die selten den kritischen Abschnitt betreten, Nachteile. Zudem kann der Token verloren gehen, wenn ein Prozess abgebrochen wird, während er im Besitz des Tokens ist.

Die im Folgenden vorgestellten Mutex-Verfahren sind zustimmungsbasiert. Ein Prozess, der den kritischen Abschnitt betreten möchte, bittet bei einem oder allen anderen Prozessen um Erlaubnis.

Zentralisierte Mutex-Verfahren verwenden einen Prozess, der Zugriffe auf die zu teilende Ressource verwaltet. Hierdurch ist Starvationsfreiheit und Fairness einfach umzusetzen, allerdings kann der verwaltende Prozess den Flaschenhals für die Performanz des Gesamtsystems darstellen [189, S. 253]. Zudem fällt das gesamte System aus, wenn der verwaltende Prozess ausfällt.

Verteilte Mutex-Verfahren haben im Gegensatz zu zentralisierten Verfahren den Vorteil, dass einer oder mehrere Prozesse ausfallen dürfen [114, S. 293 ff.]. Bei diesen Verfahren ist ein gemeinsamer Speicher erforderlich, auf den alle Prozesse zugreifen können [118]. Besitzt die Multiprozessor-Architektur eine Speicherhierarchie mit Caches, muss durch zusätzliche Hardware die Kohärenz zwischen den einzelnen Caches sichergestellt sein. Dies kann zu einem erheblichen Mehraufwand bezüglich Ausführungszeit und Verlustleistung führen (siehe Abschnitt 2.3).

Viele Prozessorarchitekturen bieten Hardwareunterstützung für ***Read-Modify-Write***[44]-Operationen bei Speicherzugriffen [18, S. 282 f.]. Es existieren Implementierungen mit verschiedenen Bezeichnungen und Semantiken (z. B. *Test-and-Set*, *Compare-and-Swap* oder *Fetch-and-Add*) [200, S. 18]. Da RISC[45]-Architekturen nur einfache Speicherzugriffe erlauben (nur Lesen oder nur Schreiben), werden hier die speziellen Befehle *Load-Linked* und *Store-Conditional* für die Realisierung von atomaren Speicherzugriffen verwendet [90, S. 239]. Nach einem Lesezugriff mittels *Load-Linked* ist der folgende *Store-Conditional*-Schreibzugriff auf die gleiche Adresse nur erfolgreich, wenn in der Zwischenzeit nicht auf die Speicherstelle zugegriffen worden ist [155, S. 137 f.]. Im Erfolgsfall schreibt *Store-Conditional* beispielsweise eine Eins in ein Register der CPU, im Fehlerfall eine Null.

Steht keine *Read-Modify-Write*-Operation zur Verfügung (es gibt nur atomare Schreib- und Lese-Operationen), sind spezielle Software-Implementierungen für den gegenseiti-

[44] Lese-Modifizieren-Schreiben
[45] *Reduced Instruction Set Computer*

gen Ausschluss notwendig. Lamport hat in [117] einen Algorithmus zum gegenseitigen Ausschluss vorgestellt, der im besten Fall lediglich sieben Speicherzugriffe benötigt. Diese Zahl ist unabhängig von der Anzahl der CPU-Kerne bzw. Prozesse, die den Mutex verwenden wollen. Der Algorithmus wurde im Rahmen der Bachelorarbeit [231] für das CoreVA-MPSoC implementiert.

Ein Mutex ist Grundlage von vielen weiteren Synchronisierungsmechanismen. Im Folgenden werden ausgewählte Verfahren vorgestellt. Für weitere Synchronisierungsmechanismen sei auf die Literatur verwiesen [18, S. 284 f.; 143, S. 107; 200, S. 100 ff.].

Ein **Semaphor** erlaubt es mehreren Prozessen, einen kritischen Abschnitt zu betreten. Hierzu wird ein Zähler mit der Anzahl der gleichzeitig erlaubten Prozesse N initialisiert. Wenn ein Prozess den kritischen Abschnitt betritt, wird der Zähler dekrementiert. Verlässt ein Prozess den kritischen Abschnitt, wird der Zähler inkrementiert. Ein Prozess darf den Abschnitt nur betreten, wenn der Zähler größer als Null ist. Ein Semaphor mit $N = 1$ wird binärer Semaphor genannt und entspricht einem Mutex [200, S. 99 f.].

Eine Synchronisierungsoperation, bei der alle Prozesse an einer bestimmten Stelle im Programmfluss blockieren bis alle anderen beteiligten Prozesse diesen Punkt erreicht haben, wird **Barriere** genannt [18, S. 281]. Ein typischer Anwendungsfall ist die parallele Ausführung einzelner Iterationen einer Schleife parallel auf mehreren CPUs. Erst wenn alle Iterationen bearbeitet worden sind, kann die Anwendung weiter ausgeführt werden.

In [18, S. 287] wird das Voll-Leer (***Full-Empty***)-Verfahren vorgestellt. Jede Speicherstelle besitzt ein spezielles Synchronisierungs-Bit, welches angibt, ob die Speicherstelle gültige Daten enthält (Bit = 1). In diesem Fall kann ein Konsument die Daten auslesen und das Bit auf Null setzen. Ist das Bit Null, kann ein Erzeuger Daten in die Speicherstelle schreiben und das Bit wird auf Eins gesetzt. Das Setzen des Synchronisierungs-Bits kann sowohl in Hardware als auch in Software erfolgen. Bei einer Hardware-Realisierung sind spezielle Schreib- und Lese-Befehle sowie ein dedizierter Speicher für die Synchronisierungs-Bits notwendig. Software-Implementierungen benötigen jeweils einen zusätzlichen Speicherzugriff für das Auslesen und Schreiben des Bits. Zudem muss das Bit im Speicher alloziert werden. Dies führt dazu, dass Software-Implementierungen auf Wortebene nicht effizient umzusetzen sind. Besitzt die verwendete CPU einen *Read-Modify-Write*-Befehl, können auch mehrere Erzeuger und/oder Konsumenten eine Speicherstelle für den Datenaustausch verwenden. Die *Full-Empty*-Synchronisierung ist Basis eines effizienten Synchronisierungsverfahrens für das CoreVA-MPSoC, welches in Abschnitt 2.6.4 vorgestellt wird.

Ein **Ringpuffer** (auch Ringbuffer, zyklischer Speicher oder *Circular-Buffer*) ist eine Software-Implementierung einer Warteschlange (FIFO), bei der Daten in einem kontinuierlichen Speicherbereich abgelegt werden. Eine Anwendung, die einen Ringpuffer verwendet, kann kontinuierlich Daten lesen und schreiben. Aus Anwendungssicht erscheint die Datenstruktur daher als Ring. Ein Ringpuffer besitzt in der Regel zwei Zeiger (*Pointer*). Der *Write-Pointer* zeigt immer auf den nächsten freien Speicherplatz, in den geschrieben werden kann. Ein *Read-Pointer* zeigt auf das erste (älteste) Element im Puffer. Wird ein Element in den Ringpuffer geschrieben bzw. ausgelesen, wird der entsprechende Pointer um Eins erhöht. Ist das letzte Element des Puffers erreicht, wird der Pointer auf den ersten Speicherplatz gesetzt. Zeigen beide Zeiger auf den gleichen Speicherplatz, kann dies entweder bedeuten, dass der Puffer voll oder leer ist. Um eine Unterscheidung beider Zustände zu ermöglichen, kann immer ein Speicherplatz leer gelassen werden. Alternativ kann eine zusätzliche Variable verwendet werden, die anzeigt, ob der Puffer leer oder voll ist. Wenn jeweils nur ein Konsument und ein Erzeuger vorhanden sind, ist die Verwendung eines Mutex' zur Synchronisierung nicht erforderlich. Manche Prozessorarchitekturen bieten spezielle Befehle, um einen Ringpuffer effizient umsetzen zu können [18, S. 283 f.]. Ein Beispiel hierfür ist die *Fetch-and-Increment*-Instruktion, welche eine Speicherstelle in ein Register kopiert und zeitgleich den Wert der Speicherstelle um Eins erhöht. In Abschnitt 2.6.4 wird eine Ringpuffer-Implementierung für das CoreVA-MPSoC vorgestellt.

Bei den bisher vorgestellten Verfahren prüft jeder Prozess vor jedem Zugriff auf einen kritischen Bereich, ob ein exklusiver Zugriff möglich ist. Ist die Wahrscheinlichkeit eines gleichzeitigen konkurrierenden Zugriffs gering, bedeutet dies einen erheblichen Mehraufwand. **Transaktionaler Speicher** (*Transactional Memory*, TM) umgeht dieses Problem, indem spekulativ der kritische Abschnitt ausgeführt wird. Nach der Ausführung wird überprüft, ob ein konkurrierender Zugriff aufgetreten ist. Ist dies der Fall, wird die gesamte Ausführung des kritischen Abschnitts rückgängig gemacht [18, S. 288; 92]. Der kritische Abschnitt wird also vollständig oder gar nicht ausgeführt und daher Transaktion genannt. Transaktionaler Speicher kann sowohl in Hardware als auch in Software realisiert werden [200, S. 101 f.]. Cain et al. stellen in [34] eine TM-Hardwareunterstützung für die IBM-Power-Architektur vor. Ferri et al. [56] betrachten die Verwendung von TM in eingebetteten Systemen. Der Einsatz von TM erfordert eine detaillierte Entwurfsraumexploration, damit die durch TM erzielte Beschleunigung der Anwendung die deutlich gesteigerte Leistungsaufnahme kompensiert.

2.6.2 Nachrichtenbasierte Kommunikation

Nachrichtenbasierte Kommunikation ist eine Alternative zur Synchronisierung mittels Mutex. Ein Sender verschickt die Nutzdaten zusammen mit Zusatzinformationen wie die Länge der Nachricht an den Empfänger [155, 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. In einem Prozessor-Cluster mit gemeinsamem Adressraum kann mithilfe von mutexbasierten Verfahren (siehe vorheriger Abschnitt) ebenfalls eine nachrichtenbasierte Kommunikation realisiert werden. Diese kann abhängig von der Anwendung effizienter sein als die Synchronisierung und Kommunikation mittels gemeinsamen Speichers. Die

MPSoCs SpiNNaker und Kalray MPPA-256 (siehe Abschnitt 2.1) kommunizieren primär über Nachrichten.

2.6.3 Stand der Technik

Thabet et al. stellen in [191] einen Hardware-Beschleuniger für die Synchronisierung innerhalb eines CPU-Clusters des STM STHORM (siehe Abschnitt 2.1.2) vor. Der Beschleuniger verwendet für die Synchronisierung atomare Zähler, die von den CPUs über die Cluster-Verbindungsstruktur angesprochen werden können. Über dedizierte Interrupt-Leitungen kann jede CPU über das Eintreten von bestimmten Ereignissen informiert werden. Für die Programmierer stehen flexible Synchronisierungsprimitive zur Verfügung. Die Integration des Beschleunigers in einen CPU-Cluster erhöht dessen Flächenbedarf um weniger als 1 % und verringert die Latenzen für die Synchronisierung um bis zu 280 %.

Braojos et al. [28] erweitern den in [52] vorgestellten MPSoC für biomedizinische Anwendungen um eine Hardware/Software-Komponente zur Synchronisierung. Mittels einer speziellen Instruktion können die CPUs des MPSoCs einen gemeinsamen Hardware-Beschleuniger konfigurieren. Dieser kann basierend auf mehreren Zählern eine oder mehrere CPUs anhalten, indem der Takt abgeschaltet wird (*Clock-Gating*). Über einen Befehl einer anderen CPU oder durch einen Interrupt eines Analog-Digital-Wandlers (*Analog to Digital Converter*, ADC) können pausierende CPUs wieder aufgeweckt werden und beispielsweise Sensordaten verarbeiten.

In [62] vergleichen Francesco et al. CPU-Cluster-Architektur mit lokalem L1- und geteiltem L2-Speicher. Ähnlich wie im CPU-Cluster des CoreVA-MPSoCs können die CPUs über die Verbindungsstruktur auf den lokalen L1-Speicher der anderen CPUs zugreifen. Francesco et al. verwenden für die Synchronisierung der CPUs Hardware-Semaphore, die entweder als Slave-Komponente im CPU-Cluster oder eng gekoppelt innerhalb der CPU-Kerne integriert sind. Insbesondere bei einer Kommunikation mit Blockgrößen von mehr als 32 B zeigt der verteilte Ansatz mit lokalem L1-Speicher und lokalen Semaphore einen höheren Durchsatz und eine höhere Energieeffizienz.

2.6.4 Synchronisierung im CPU-Cluster des CoreVA-MPSoCs

Aufbauend auf der *Full-Empty*-Synchronisierung (siehe Abschnitt 2.6.1) ist im Rahmen dieser Arbeit ein neuartiges Synchronisierungsverfahren auf Blockebene entstanden. 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 2.3). 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

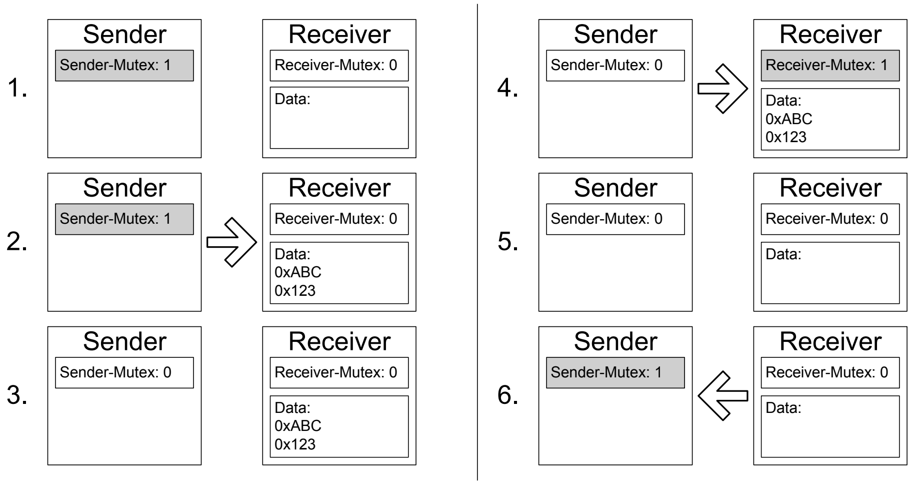


Abbildung 2.10: Blockbasierte Kommunikation im CPU-Cluster des CoreVA-MPSoCs

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). Dementsprechend gibt eine Eins im Receiver-Mutex an, dass die Daten gültig sind und vom Konsumenten gelesen werden können. In Abbildung 2.10 ist anhand eines Beispiels der Ablauf eines Synchronisierungszyklus dargestellt. Das Synchronisierungsverfahren wird auch für die Synchronisierung zwischen einer CPU und der NoC-Schnittstelle des CPU-Clusters verwendet, um eine effiziente Synchronisierung mehrerer CPUs eines Clusters und des NoCs sicherzustellen.

Ein weiteres im Rahmen dieser Arbeit implementiertes Synchronisierungsverfahren ist ein Ringpuffer auf Wortebene. Im lokalen Scratchpad-Speicher des Senders wird der Schreib-Zähler abgelegt, im lokalem Speicher des Empfängers der Lese-Zähler sowie die Nutzdaten. Bedingt durch diese Aufteilung muss der Empfänger nach jedem Lesezugriff den aktuellen Wert des Schreib-Zählers über den Bus lesen und den aktualisierten Zähler in den Speicher des Senders speichern. Eine Aktualisierung des Lese-Zählers durch den Sender verläuft ebenso. Jeder Zugriff auf den Ringpuffer führt also zu einem Mehraufwand für die Verarbeitung der Zähler des Ringpuffers. Ein Vergleich beider Synchronisierungsverfahren anhand verschiedener Beispielprogramme ist in Abschnitt 5.3 zu finden.

Die Anzahl an Kommunikationskanälen innerhalb eines CPU-Clusters ist nur durch die Größe der On-Chip-Speicher begrenzt. Hierzu wird im Datenspeicher jeder CPU ein statischer Speicherbereich alloziert, dessen Adresse den anderen CPUs bekannt ist. In diesem statischen Speicherbereich befindet sich jeweils ein Mutex-Paar für die Kommunikation mit jeder anderen CPU. Der eigentliche Speicher für Mutex bzw. Ringpuffer eines (dynamischen) Kommunikationskanals wird auf der jeweiligen CPU zur Laufzeit alloziert. Die entsprechende Adresse wird über den statischen Speicherbereich dem jeweils anderen Kommunikationspartner mitgeteilt.

Die vorgestellten Synchronisierungsverfahren sind in der Programmiersprache C implementiert. Es stehen Funktionsaufrufe für die Initialisierung der Kommunikationskanäle zur Verfügung. Zudem kann ein Datenwort bzw. ein Datenblock angefordert und freigegeben werden. In Abschnitt 5.8.2 wird der Energiebedarf des blockbasierten Synchronisierungsverfahrens untersucht sowie eine Mutex-Hardwareerweiterung für die CoreVA-CPU vorgestellt. Das Synchronisierungsverfahren wurde zudem in den Veröffentlichungen [221; 223] beschrieben.

2.7 Zusammenfassung

In diesem Kapitel wurde die Systemarchitektur eines eng gekoppelten CPU-Clusters für das CoreVA-MPSoCs vorgestellt. Hierzu wurde zunächst der Stand der Technik von MPSoCs betrachtet und der CoreVA-VLIW-Prozessor als wichtige Systemkomponente beschrieben. Die CoreVA-CPU eignet sich aufgrund einer hohen Ressourceneffizienz gut für die Verwendung in einem eng gekoppelten Multiprozessorsystem. Anschließend wurden Grundlagen zu Speichern in eingebetteten Systemen, On-Chip-Verbindungsstrukturen sowie Synchronisierungsverfahren erläutert.

Eng gekoppelte On-Chip-Verbindungsstrukturen ermöglichen eine Kommunikation der einzelnen Systemkomponenten des CPU-Clusters mit hoher Bandbreite und geringer Latenz. Aus diesem Grund wurden verbreitete Bus-Standards und Implementierungen vorgestellt und verglichen. Hierbei wurde gezeigt, dass die Speicherarchitektur und die Art der Synchronisierung die Architektur der Verbindungsstrukturen maßgeblich beeinflussen. Insbesondere das Zusammenspiel von Kommunikations- und Speicherarchitektur sowie Synchronisierung ist ein offenes Feld in der Forschung. Der Stand der Technik zeigt eine Vielzahl an verschiedenen Architekturen. Die in diesem Kapitel vorgestellte Systemarchitektur eines CPU-Clusters für das CoreVA-MPSoC stellt einen Beitrag zu diesem Feld dar. Der CPU-Cluster zeichnet sich insbesondere durch eine Kombination von lokalem und verteiltem L1-Datenspeicher aus. Zudem wurde in diesem Kapitel ein neuartiges blockbasiertes Synchronisierungsverfahren und ein darauf aufbauendes Kommunikationsmodell vorgestellt. Dies erlaubt eine Synchronisierung im CPU-Cluster des CoreVA-MPSoCs mit einem geringen Mehraufwand sowohl in Bezug auf die Hardware- als auch auf die Softwarekomponenten des Systems. Der große Entwurfsraum von eng gekoppelten Prozessorclustern erfordert eine abstrakte

Modellierung der Hardware- und Softwarekomponenten des Systems in einem frühen Entwurfsstadium (siehe Kapitel 4). Hierauf aufbauend wird in Kapitel 5 eine detaillierte Entwurfsraumexploration des CoreVA-MPSoC-CPU-Clusters durchgeführt. In Kapitel 3 wird jedoch zunächst die Hardware- und Software-Entwurfsumgebung des CoreVA-MPSoCs vorgestellt, die als Grundlage für Modellierung und Entwurfsraumexploration dient.

3 Hardware- und Software-Entwurfsumgebung

Neben den im vorangegangenen Kapitel beschriebenen Hardwarekomponenten und Kommunikationsmodellen ist für die Entwicklung von elektronischen eingebetteten Systemen eine leistungsfähige Entwurfsumgebung notwendig. Für die Hardware- bzw. Chipentwicklung wird in Abschnitt 3.1 ein Hardware-Entwurfsablauf vorgestellt. In Abschnitt 3.2 wird die Software-Entwurfsumgebung beschrieben, die vor allem aus Compilern für die Sprachen C, StreamIt und OpenCL besteht. Die verschiedenen Abstraktionsebenen, auf denen das CoreVA-MPSoC simuliert bzw. emuliert werden kann, werden in Abschnitt 3.3 vorstellt. Eine Simulation bzw. Emulation wird sowohl in der Software- als auch in der Hardwareentwicklung in allen Schritten des Entwurfsablaufs eingesetzt.

Um die schnelle Markteinführung eines Produktes (*Time-to-Market*) sicherzustellen, ist eine parallele Entwicklung von Hardware und Software eines eingebetteten Systems unabdingbar [40]. Die gemeinsame Entwicklung und ganzheitliche Betrachtung von Software- und Hardwarekomponenten eines eingebetteten Systems wird als *Hardware/Software-Codesign* bezeichnet. An dieser Stelle sei angemerkt, dass die Hardware-Software-Partitionierung nicht Bestandteil dieser Arbeit ist. Nichtsdestotrotz können die in diesem Kapitel vorgestellte Werkzeugkette und die in Kapitel 4 beschriebenen Modelle als Bestandteil einer Entwurfsumgebung für Hardware/Software-Codesign eingesetzt werden.

Die Entwicklung eines mikroelektronischen eingebetteten Systems beginnt mit der Spezifikation der Systemanforderungen und einer abstrakten **ausführbaren Spezifikation** z. B. in den Sprachen C, C++ oder SystemC [190]. Neben den Eigenschaften der Anwendung (siehe Abschnitt 4.2) werden hierbei auch die Eigenschaften von vorhandenen Hardwarekomponenten wie Prozessoren, On-Chip-Verbindungsstrukturen und Beschleunigern berücksichtigt. Die Spezifikation wird typischerweise auf einem x86-System[1] ausgeführt, da hier keine relevanten Beschränkungen bezüglich Speicherbedarf oder Zahlendarstellung (z. B. Fließkomma) bestehen. Zudem können Bibliotheken verwendet werden, die auf dem eingebetteten System z. B. zu viel Speicherplatz belegen oder sich als nicht effizient genug erweisen. Durch dieses Vorgehen kann die Entwicklung des Gesamtsystems stark beschleunigt werden [190].

[1] mit Windows oder Linux als Betriebssystem

Anschließend wird die Entscheidung getroffen, welche Anwendungsteile in Software auf eingebetteten CPUs oder als anwendungsspezifische Schaltung realisiert werden. Zudem wird die Architektur der CPU-Kerne und der Verbindungsstruktur des MPSoCs festgelegt. Die einzelnen Teile der Anwendung werden funktional vollständig implementiert, beispielsweise indem die ausführbare Spezifikation (C, C++ oder SystemC) erweitert wird. Parallel hierzu werden Testdaten für die einzelnen Anwendungsteile erstellt, um die Einhaltung der Spezifikation zu überprüfen. Diese Testdaten können bei allen weiteren Arbeitsschritten sowohl für die Hardware- als auch für die Softwareentwicklung wiederverwendet und erweitert werden. Erfüllen alle Anwendungsteile die Spezifikation, ist eine Integration in einem Gesamtsystem möglich, welches wiederum mit Tests gegen die Spezifikation geprüft werden kann.

3.1 Hardware-Entwurfsablauf

In modernen Fertigungstechnologien hergestellte Chips enthalten eine Milliarde und mehr Transistoren [96]. Diese große Zahl an Transistoren erfordert eine hohe Entwurfsproduktivität [111, S. 669]. Im Rahmen dieser Arbeit wird eine *Semi-Custom*-Entwurfsmethode verwendet (siehe Abbildung 3.1) [219, S. 33 f.; 111, S. 673 f.]. Ausgehend von einer (abstrakten) Spezifikation und einer HDL[2]-Beschreibung wird durch mehrere automatisierte oder teilautomatisierte Entwurfsschritte das CoreVA-MPSoC auf eine Zieltechnologie abgebildet. Hierbei kommt eine 28-nm-FD-SOI[3]-Technologie von STMicroelectronics [182] mit zehn Metallisierungsebenen (*Metal-Layer*) zum Einsatz. 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-Density*-Speicher eingesetzt. Es existieren zudem Speicherblöcke mit geringeren Verzögerungszeiten (*High-Speed*-SRAM), die jedoch eine geringere Ressourceneffizienz besitzen.

In einem ersten Schritt wird die HDL-Beschreibung in einer RTL-Simulation auf eine korrekte Funktion hin überprüft (siehe Abschnitt 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 Software für die Synthese wird Encounter RTL Compiler 14.11 [33] 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.

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

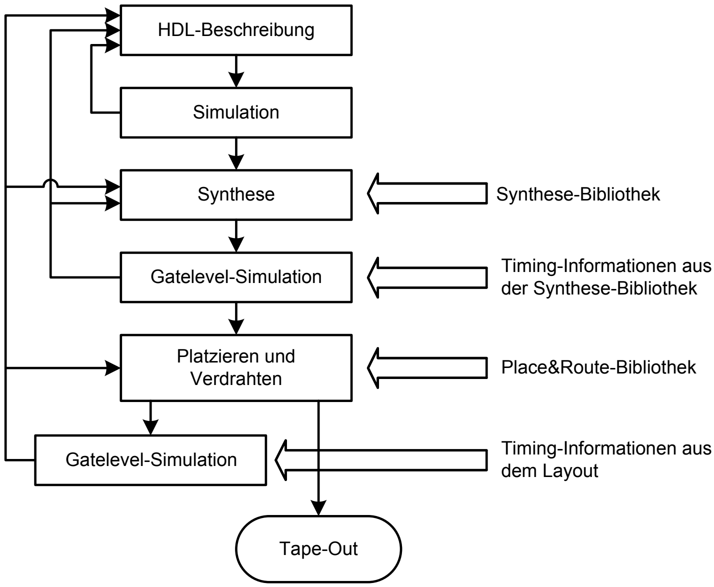


Abbildung 3.1: Hardware-Entwurfsablauf für Standardzellen-Technologien nach [111, S. 673]

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 Gatelevel-Simulation (siehe Abschnitt 3.3) verwendet werden.

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 die Anwendung Cadence Encounter Digital Implementation System 14.13 [32] verwendet. Die P&R-Bibliothek wird auch als Standardzellen-Bibliothek bezeichnet, da alle Logikzellen eine identische Höhe besitzen [111, S. 674 f.; 176, S. 142 ff.]. Die Logikfunktionen der einzelnen Zellen sind durch zwei oder mehr Transistoren in CMOS[4]-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 [111, S. 677, Abbildung 8.5]).

[4]*Complementary Metal Oxide Semiconductor*

Der Entwickler muss die Abmessungen des Chips sowie die Position von Makros (z. B. Speicher oder CPU-Kerne, siehe Kapitel 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 die P&R-Software. 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 der P&R-Software 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 5.2.3 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 [111, 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 überprüft und wiederum auf Gatterebene simuliert. Sind alle Anforderungen der Spezifikation an den Chipentwurf erfüllt, kann dieser in das GDSII[5]-Format überführt, an den Chiphersteller gesendet und von diesem gefertigt werden (*Tape-Out*) [111, S. 675 f.].

Der in diesem Abschnitt beschriebene Entwurfsablauf findet in Kapitel 5 bei der Entwurfsraumexploration des CoreVA-MPSoCs sowie für die Realisierung eines ASIC-Prototypen Anwendung (siehe Abschnitt 6.2).

[5] *Graphic Database System 2*

3.2 Software-Entwurfsumgebung

Softwareentwickler von eingebetteten Systemen benötigen eine Entwurfsumgebung, um Anwendungen in einer Hochsprache entwickeln und anschließend in die Maschinensprache der Zielplattform abbilden zu können. Hierzu gehören Übersetzer (Compiler) für eine oder mehrere Hochsprachen sowie ein Assembler und ein Linker. Um Anwendungsentwicklern von eingebetteten Systemen eine komfortable Entwurfsumgebung zur Verfügung stellen zu können, werden Anwendungen typischerweise nicht direkt auf der Zielplattform übersetzt. Stattdessen wird ein *Cross-Compiler* beispielsweise auf einem x86-Linux ausgeführt, um Maschinencode für die Ausführung auf dem CoreVA-MPSoC zu erzeugen. Bestandteil der Entwurfsumgebung eines eingebetteten Systems ist typischerweise ein Instruktionssatzsimulator (*Instruction Set Simulator*, ISS). Der ISS des CoreVA-MPSoCs wird in Abschnitt 3.3.1 beschrieben.

3.2.1 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 [86, 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 3.3).

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- und L2-Speichern eines CPU-Clusters zu allozieren (siehe Abschnitt 2.3 sowie [231, S. 18 ff.]). Zudem stehen Funktionen zur Synchronisierung von CPU-zu-CPU-Kommunikation zur Verfügung (siehe Abschnitt 2.6.4).

Eine effiziente Off-Chip-Kommunikation basiert auf dem in Abschnitt 2.6.4 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 [232]. 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 [245] sowie den RAPTOR-Modulen DB-ETG[6] und DB-TFT[7] [232] zur Verfügung (siehe Abschnitt 3.3.2). Das DB-TFT kann zudem für die Darstellung der Standardausgabe verwendet werden.

3.2.2 LLVM-basierter Compiler für die CoreVA-CPU

Das *Open-Source*-Projekt LLVM[8] [192] 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 [120]. Typischerweise verwenden moderne Compiler einen dreistufigen Übersetzungsprozess [131, 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[9]. 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 [246]. Es unterstützt neben VLIW-Vektorisierung viele Eigenschaften der CoreVA-CPU wie den *Delay*-Slot bei Sprüngen oder das bedingte Ausführen von Instruktionen. 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. LLVM und das CoreVA-Backend sind zudem Basis der OpenCL-Laufzeitumgebung für das CoreVA-MPSoC (siehe Abschnitt 3.2.4).

3.2.3 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. Hierbei existieren eine Vielzahl an gültigen Partitionierungen (siehe Abschnitt 5.4). Es ist für den Programmierer nur schwer möglich, eine optimale Partitionierung z. B. in Bezug auf den Durchsatz der Anwendung zu

[6] *Daughterboard Ethernet Gigabit*
[7] *Daughterboard Thin-Film Transistor*, ein Bildschirmmodul
[8] Modulare Compilerarchitektur, früher *Low Level Virtual Machine*
[9] *LLVM Intermediate Representation*

finden. Aus diesem Grund wird an 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 Multimedia, Basisbandverarbeitung und Verschlüsselung. In Kooperation mit der QUT in Brisbane, Australien ist ein Compiler für die am MIT[10] spezifizierte Sprache StreamIt entstanden [66; 193; 195]. Die StreamIt-Sprache ermöglicht die Beschreibung einer Streaming-Anwendung als Datenflussgraph, der aus sogenannten Filtern besteht. Ein Beispiel ist im Anhang A 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. 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 (siehe Abschnitt 4.2).

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.

Hierzu stehen drei Algorithmen zur Verfügung [221]. Im Rahmen dieser Arbeit wird als Optimierungsziel der Durchsatz der Anwendung maximiert. Aktuelle Forschungsprojekte erweitern den Compiler um weitere Optimierungsziele wie z. B. die Latenz der Anwendung. Der *Greedy*[11]-Algorithmus weist in jedem Schritt den Filter mit der längsten Laufzeit der CPU mit der geringsten Auslastung zu [112, 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 [221]

[10] *Massachusetts Institute of Technology*
[11] gierig

veröffentlichter *Simulated-Annealing*[12]-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 zu bewerten, kann eine Simulation auf einem Instruktionssatzsimulator 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 wurde daher ein Schätzer für die Performanz von Streaming-Anwendungen („*Performance-Estimator*“) entwickelt, der eine Partitionierung wesentlich schneller bewertet (siehe Abschnitt 4.2.4). 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. Zudem wird der Speicherbedarf der Partitionierung abgeschätzt und diese verworfen, wenn mehr Speicher verwendet wird als das MPSoC zur Verfügung stellt. Details zum Performance-Estimator sind in einem technischen Bericht [213] und dem Konferenzbeitrag [216] veröffentlicht. Als dritter Partitionierungs-Algorithmus steht eine Portierung des in [67] 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 der selben 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 2.6.4 vorgestellten Synchronisierungsverfahren eingesetzt. Sind beide Filter auf unterschiedlichen Clustern des MPSoCs platziert kommt ein *NoC-Channel* zum Einsatz (siehe auch [221]).

Abschließend erzeugt der CoreVA-MPSoC-Compiler C-Code für jede CPU des MPSoCs. Dieser kann mithilfe des LLVM-basierten C-Compilers in Maschinencode übersetzt werden. Weitere Details zur Implementierung des CoreVA-MPSoC-Compilers für Streaming-Anwendungen sind in der Bachelorarbeit [112] und den Veröffentlichungen [216; 221] zu finden. Der CoreVA-MPSoC-Compiler wird in Kombination mit dem *Simulated-Annealing*-Partitionierungsalgorithmus in Kapitel 5 für eine Entwurfsraumexploration verschiedener Hardware-Konfigurationen des CoreVA-MPSoCs verwendet.

[12] simulierte Abkühlung

3.2.4 Das CoreVA-MPSoC als OpenCL-Plattform

OpenCL (*Open Computing Language*) ist ein offener Industriestandard für die parallele Programmierung von heterogenen Systemen und wird von der Khronos Group spezifiziert und weiterentwickelt [94, S. 12]. Bestandteile von OpenCL sind eine Programmierschnittstelle, Bibliotheken, eine Laufzeitumgebung sowie die Programmiersprache OpenCL C [94, S. 25]. Eine OpenCL-Plattform besteht aus einem *Host* sowie einem oder mehreren OpenCL-Geräten (*Devices*). Jedes Gerät verfügt über einen globalen Speicher. Ein Gerät ist in *Compute-Units* und *Processing-Elements* aufgeteilt, die jeweils über einen lokalen bzw. privaten Speicher verfügen (siehe Abbildung 3.2). Ziel von OpenCL ist es, eine Anwendung ohne Änderungen am Quelltext auf verschiedenen Geräten ausführen zu können. Beispiele für Geräte sind GPUs, DSPs, FPGAs sowie CPUs bzw. MPSoCs.

Eine OpenCL-Anwendung besteht aus zwei Teilen. Der Host führt das Hauptprogramm aus und stellt den Geräten z. B. Daten für eine Berechnung in einem gemeinsamen Speicher bereit [97]. Auf den Geräten werden *Kernel* ausgeführt, welche die einzelnen Teilberechnungen der Anwendung darstellen. Die Kernel werden auf dem Host zur Laufzeit für ein Gerät übersetzt. Aus diesem Grund muss zur Entwurfszeit nicht feststehen, auf welchem Gerät der Kernel ausgeführt werden soll. Die OpenCL-Plattform muss hierfür über einen Compiler für jedes Gerät verfügen. Ein OpenCL-Beispielprogramm ist im Anhang A zu finden. Die Multiprozessoren Adapteva Epiphany und STMicroelectronics STHORM bieten Unterstützung für OpenCL (siehe Abschnitt 2.1).

In der Masterarbeit [229] wurde eine OpenCL-Plattform für das CoreVA-MPSoC entwickelt. Basis ist die freie OpenCL-Implementierung pocl[13] [97; 162]. pocl verwendet LLVM als Kernel-Übersetzer sowie Clang als OpenCL C-Frontend. Daher kann das

[13] *Portable Computing Language*

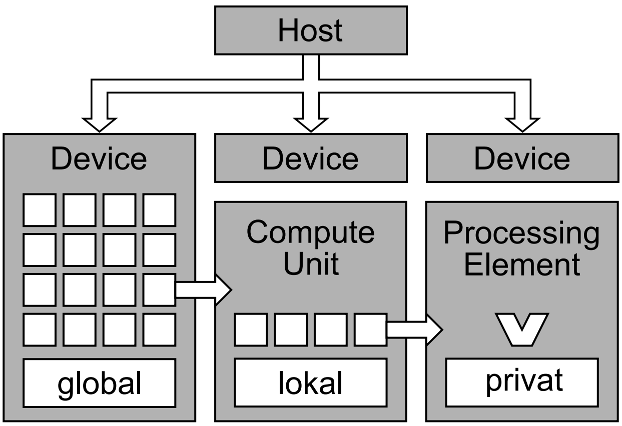


Abbildung 3.2: Hierarchie einer OpenCL-Plattform [229]

CoreVA-Backend für LLVM auch für die CoreVA-MPSoC OpenCL-Plattform verwendet werden. Als OpenCL-Gerät wird der Instruktionssatzsimulator des CoreVA-MPSoC (siehe Abschnitt 3.3.1) verwendet, der um eine Schnittstelle zu pocl erweitert worden ist. Zudem existiert eine FPGA-basierte Implementierung auf dem RAPTOR-System [229, S. 25]. Die CoreVA-MPSoC-OpenCL-Plattform wird in Kapitel 5 im Rahmen einer Entwurfsraumexploration verwendet.

3.3 Funktionale Verifikation durch Simulation und Emulation

Integraler Bestandteil der Software- und Hardwareentwicklung von eingebetteten Systemen ist die Simulation und Emulation des Ziel-MPSoCs auf verschiedenen Abstraktionsebenen [190]. In Abbildung 3.3 sind verschiedene Abstraktionsebenen dargestellt, die für den Software- und Hardwareentwurf im Rahmen des CoreVA-MPSoC-Projektes verwendet werden können.

Ausgehend von der Spezifikation kann der Softwareanteil der Anwendung weiter konkretisiert werden. Wurde für die ausführbare Spezifikation eine andere Sprache gewählt, als für die eigentliche Implementierung, so ist an dieser Stelle eine Portierung bzw. Neuimplementierung notwendig. In einem Multiprozessor-System tauschen die einzelnen (Software-) Anwendungsteile untereinander und mit Hardwareblöcken Daten aus. Die erforderliche Synchronisierung der Kommunikation wird durch ein Programmiermodell gekapselt. Für die in Abschnitt 2.6.4 vorgestellten Programmiermodelle des CoreVA-MPSoCs steht eine Schnittstelle (*Wrapper*) für die Betriebssysteme Windows und Linux zur Verfügung, der auf POSIX[14]-Threads (**x86-Threads** oder Pthreads) basiert. Durch die Verwendung dieses x86-Thread-Wrappers ist es möglich, die ausführ-

[14] *Portable Operating System Interface*

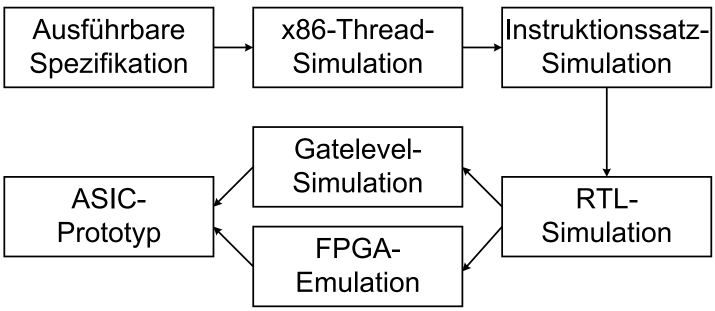


Abbildung 3.3: Simulation bzw. Emulation des CoreVA-MPSoCs auf verschiedenen Abstraktionsebenen

bare Spezifikation in den Sprachen C und C++ weiter zu verwenden und zu erweitern. Dieses Vorgehen bietet zudem den Vorteil, dass Entwicklungsumgebungen wie beispielsweise Eclipse oder Microsoft Visual Studio für die Entwicklung, Codeanalyse und Debugging verwendet werden können. Alternativ kann die ausführbare Spezifikation in die StreamIt-Sprache portiert werden (siehe Abschnitt 3.2.3). Der CoreVA-MPSoC-Compiler bietet die Möglichkeit, eine Anwendung auf einem x86-System sequenziell in einem Thread auszuführen. Zudem ist die Verwendung mehrere Betriebssystem-Threads möglich, die mithilfe des in Abschnitt 2.6.4 vorgestellten blockbasierten Synchronisierungsverfahren kommunizieren.

Im nächsten Schritt der Anwendungsentwicklung wird die Anwendung auf die Prozessorarchitektur des CoreVA-MPSoCs, also die CoreVA-CPU, abgebildet und zyklengenau[15] simuliert. Im Kontext der Simulation einzelner Prozessorkerne wird der Begriff **Instruktionssatzsimulator** (ISS[16]) verwendet. Ein ISS abstrahiert die Eigenschaften einer Mikroarchitektur [99, S. 42 f.]. Beispielsweise wird die Pipeline-Architektur der CPU nicht simuliert. Im Kontext eines Multiprozessorsystems kann der Begriff **zyklengenauer Multiprozessor-Simulator** verwendet werden. Es wird im Folgenden jedoch ISS als Bezeichnung für den Simulator verwendet, da dieser Begriff in der Fachwelt weit verbreitet ist. Ein ISS kann basierend auf der Hardware-Spezifikation entwickelt werden und steht zu einem frühen Zeitpunkt im Entwicklungsprozess zur Verfügung. Die Architektur des CoreVA-MPSoC-ISS ist in Abschnitt 3.3.1 beschrieben. Durch die Verwendung des ISS kann das erste Mal im Entwurfsprozess eine verlässliche Aussage über die Laufzeit bzw. Ausführungsgeschwindigkeit der Anwendung getroffen werden. Daher ist es möglich, die Anwendung zu optimieren. Es können die Testdaten der x86-basierten Simulation verwendet werden, um die Einhaltung der Spezifikation zu überprüfen. Der ISS wird typischerweise zusammen mit der Hardware entwickelt und Änderungen an der MPSoC-Hardware werden umgehend auf den ISS portiert [190]. Dies verbessert die Aussagekraft vom ISS über die spätere Zielplattform (ASIC). Ein Beispiel für das Zusammenspiel von Hardware- und Software-Entwicklung ist die Integration von Registerstufen in die Cluster-Verbindungsstruktur, um eine geforderte Taktfrequenz erreichen zu können. Diese Registerstufen müssen auch vom ISS abgebildet werden. Die Simulation mittels ISS kann jedoch auch Änderungen an der MPSoC-Hardware bedingen. Beispielsweise kann sich die ursprünglich geplante Verwendung eines geteilten Busses als nicht performant genug für eine Anwendung erweisen. Es ist der Einsatz einer Crossbar notwendig, die sowohl in der Hardwarebeschreibung als auch im ISS integriert werden muss (siehe Abschnitt 2.4).

Die einzelnen Schritte dieses Erstellungsprozesses (*Build-Prozess*) sind für den Software-Entwickler in einem *Makefile* gekapselt. Es wird zunächst der gesamte Quellcode der Anwendung mit dem in Abschnitt 3.2.2 vorgestellten LLVM-Compiler und dessen CoreVA-spezifischem Backend übersetzt. Die CoreVA-CPU unterstützt zurzeit nicht die

[15] Im Vergleich zur Ausführung auf einem Chip-Prototypen
[16] *Instruction Set Simulator*

Verwendung von Fließkomma-Arithmetik. Falls die Beschreibung der Anwendung bisher Fließkomma-Arithmetik enthält, muss diese durch eine Festkomma-Arithmetik ersetzt werden. Zudem steht für die CoreVA-CPU nur eine eingeschränkte Systembibliothek (LibC) zur Verfügung. Nicht unterstützte Systemaufrufe müssen daher aus dem C/C++-Code entfernt oder ersetzt werden. Auch ein StreamIt-Programm kann Fließkomma-Arithmetik enthalten, die ersetzt werden muss. Ansonsten ist bei der Verwendung von StreamIt keine Anpassung des Quellcodes erforderlich. Der CoreVA-MPSoC-Compiler erzeugt CoreVA- und LLVM-konformen C-Code. Die vom LLVM-Compiler und dem Linker erzeugte Binärdatei kann vom CoreVA-MPSoC-ISS eingelesen und simuliert werden.

Die **RTL-Simulation** einer Multiprozessor-Architektur führt Anwendungen unter Verwendung der HDL[17]-Beschreibung des MPSoCs aus[18]. Die RTL-Simulation ist für die Hardware-Entwicklung unverzichtbar, um die Übereinstimmung der HDL-Beschreibung mit der Spezifikation sicherzustellen [111, 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 3.1 sowie Kapitel 5), kann eine **Gatelevel-Simulation** durchgeführt werden [111, 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. Parallel zur Gatelevel-Simulation kann eine **Emulation** des MPSoCs auf einem **FPGA** durchgeführt

[17] *Hardware Description Language*
[18] Das CoreVA-MPSoC ist in Sprache VHDL beschrieben

werden [111, S. 686]. Zwei FPGA-basierte Prototypen des CoreVA-MPSoC werden in Abschnitt 3.3.2 beschrieben. An dieser Stelle soll auf die Verwendung des Prototypen in der Software-Entwicklung eingegangen werden. Der FPGA-Prototyp des CoreVA-MPSoCs erlaubt die Emulation von bis zu 24 CoreVA-CPUs bei einer Geschwindigkeit von bis zu 124 MHz (siehe Abschnitt 6.1). Es steht eine Host-Software zur Verfügung, mit deren Hilfe es möglich ist, Programme auf das MPSoC zu übertragen und auszuführen. Zudem können Testdaten auf das MPSoC geladen und Ergebnisse zurück gelesen werden. Durch die hohe Taktfrequenz der Emulation kann z. B. das erste Mal im Entwicklungsprozess ein Betriebssystem auf dem MPSoC ausgeführt werden. Des Weiteren ist eine prototypische Integration von externen Schnittstellen wie z. B. Ethernet möglich.

In einem späten Stadium der Entwicklung eines mikroelektronischen, eingebetteten Systems steht ein **ASIC-Prototyp** für die Softwareentwicklung zur Verfügung. Im Gegensatz zum finalen Chipentwurf kann der ASIC-Prototyp beispielsweise nicht die geforderte Taktfrequenz erreichen, funktionale Fehler enthalten oder erhöhte Leistungsaufnahme aufweisen. Nichtsdestotrotz stellt ein ASIC-Prototyp eine wichtige Plattform für den Softwareentwurf dar. Es ist möglich, das Gesamtsystem aus Hardware und Software bei hoher Ausführungsgeschwindigkeit und Genauigkeit mit der Spezifikation zu vergleichen. Hierbei wird der ASIC-Prototyp allerdings typischerweise weiterhin auf einem Prototyping-System integriert.

Alle vorgestellten Simulatoren bzw. Emulatoren des CoreVA-MPSoCs unterstützen eine Standardausgabe (`printf`) sowie eine Schnittstelle zur Datei-Ein- und -Ausgabe („FileIO“, siehe Abschnitt 3.2.1). Dies ermöglicht die Durchführung eines Regressionstests. Für die in Abschnitt 5.1 vorgestellten StreamIt-Anwendungen existieren Referenzausgaben. Anhand dieser kann die korrekte Funktion der Simulatoren und der HDL-Beschreibung des CoreVA-MPSoCs durch Ausführung eines Regressionstests überprüft werden.

Tabelle 3.1 zeigt einen quantitativen Vergleich der verschiedenen Simulations- und Emulationsebenen des CoreVA-MPSoCs. Mit steigender Genauigkeit bezüglich der benötigten Taktzyklen und Energie einer Anwendung sinkt bei der Verwendung von Simulatoren die Ausführungsgeschwindigkeit. Im Gegensatz hierzu bieten die FPGA- und ASIC-Prototypen eine hohe Ausführungsgeschwindigkeit bei hoher Genauigkeit. Diese stehen jedoch erst zu einem späten Zeitpunkt im Entwicklungsprozess zur Verfügung. Im Vergleich zu einer Simulation bieten die FPGA- und ASIC-Prototypen nur eine begrenzte Beobachtbarkeit von internen Zuständen und Signalwechseln des MPSoCs.

In Abbildung 3.4 ist die Ausführungszeit der Anwendung Filterbank dargestellt, die benötigt wird, um 100 Ausgabeelemente zu erzeugen. Hierbei werden die verschiedenen Simulations- und Emulationsebenen des CoreVA-MPSoCs sowie 1 bis 128 CPUs betrachtet. Filterbank wird durch den CoreVA-MPSoC-Compiler auf die verschiedenen CPU-Cluster-Konfigurationen partitioniert (siehe Abschnitt 3.2.3 und Kapitel 5). Anschließend werden die Simulationen auf einem Linux-PC mit Ubuntu Linux 12.04, einem Intel Xeon E5-1650 Prozessor mit sechs physikalischen CPUs und 3,5 GHz Taktfre-

Tabelle 3.1: Vergleich der verschiedenen Simulations- und Emulationsebenen des CoreVA-MPSoCs

	Simulation				Emulation	
	x86-Threads	ISS	RTL	Gatelevel	FPGA	ASIC
Genauigkeit Zyklen	– –	+	++	++	++	++
Genauigkeit Energie	– –	–	+	++	–	+++
Verfügbarkeit	++	+	–	–	–	– – –
Skalierung mit CPUs	+	–	–	–	+	++
Taktfrequenz	2 GHz	10 MHz	1 kHz	10 Hz	100 MHz	1 GHz

quenz sowie 128 GB Arbeitsspeicher ausgeführt. Um Messfehler zu reduzieren werden ISS- und x86-Thread-Simulation 100-mal sowie die RTL-Simulation 30-mal ausgeführt und die gemessene Ausführungszeit anschließend durch die Anzahl der Durchläufe geteilt. Für die FPGA-Emulation und den ASIC-Prototypen basiert die Ausführungszeit auf der benötigten Anzahl an Taktzyklen der RTL-Simulation und einer zu erwartenden Ziel-Taktfrequenz von 80 MHz (FPGA) bzw. 800 MHz (ASIC). Die Analyse der Ausführungszeit der verschiedenen Simulationsebenen des CoreVA-MPSoCs wurde als Bestandteil des Konferenzbeitrags [216] veröffentlicht. Die RTL-Simulation zeigt

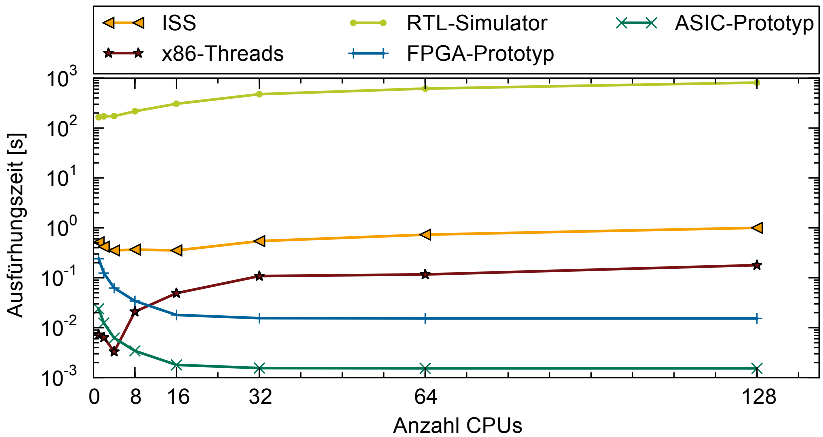


Abbildung 3.4: Ausführungszeit, um 100 Ausgabeelemente der Anwendung Filterbank auf verschiedenen Simulations- und Emulationsebenen des CoreVA-MPSoCs zu erzeugen. Es werden CPU-Cluster mit einer bis 128 CPUs betrachtet.

mit 164 s für eine CPU und 812 s für einen CPU-Cluster mit 128 CPUs die längste Ausführungszeit. Im Vergleich hierzu zeigt die Instruktionssatz-Simulation eine um zwei bis drei Größenordnungen schnellere Ausführungszeit von 0,35 s bis 1,00 s. Mit steigender Anzahl an CPU-Kernen sinkt die Anzahl der zu simulierenden bzw. emulierenden MPSoC-Takte, da die Anwendung beschleunigt ausgeführt wird. Aus diesem Grund sinkt die Ausführungszeit der FPGA-Emulation von 0,24 s (1 CPU) auf 15,41 ms (128 CPUs) mit steigender Anzahl an CPUs ab. Der ASIC-Prototyp zeigt ein ähnliches Verhalten mit Ausführungszeiten von 24,03 ms (1 CPU) bis 1,54 ms (128 CPUs). Bei der Simulation steigt der Aufwand linear mit der Anzahl zu simulierenden CPUs an. ISS- und RTL-Simulator verwenden nur einen Thread auf dem Simulations-PC und zeigen daher einen linearen Anstieg der Ausführungszeit, wenn MPSoCs mit mehr als vier CPU-Kernen simuliert werden. Bei der x86-Thread-Simulation ist ein besonders starker Anstieg zu beobachten, wenn deutlich mehr Threads ausgeführt werden, als der Simulations-PC physikalische CPU-Kerne aufweist.

3.3.1 Der Instruktionssatzsimulator des CoreVA-MPSoCs

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 [228, S. 6 f.]. Zuerst werden die Maschinenbefehle dekodiert und in eine interne Datenstruktur überführt. In einem zweiten Schritt werden die Ergebnisse der Instruktionen berechnet. Diese werden nicht direkt in die Registerbank übernommen, stattdessen wird eine Differenzstruktur der Änderungen erzeugt. Die Differenzstruktur ermöglicht die Erstellung eines Prozessor-Traces (siehe Abschnitt 2.2) und wird anschließend 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 über den Bus auf die Slave-Schnittstelle der CPU ein Event erzeugt.

Der **Cluster-Simulator** instantiiert mehrere CPU-Simulatoren und bindet die einzelnen CPUs des Clusters an ein Simulationsmodell der AXI- bzw. Wishbone-Verbindungsstruktur an. Weitere Komponenten sind Simulationsmodelle der NoC-Cluster-Schnittstelle (siehe Abschnitt 1.2) sowie des eng gekoppelten gemeinsamen L1-Datenspeichers (siehe Abschnitt 2.3.2). Für die Synchronisierung der Kommunikation der einzelnen CoreVA-CPUs sowie der Verbindungsstruktur wurden im Master-Projekt [228] verschiedene Implementierungen verglichen, die im Folgenden vorgestellt werden.

Sequenzielle Implementierung

Die sequenzielle Referenzimplementierung des CoreVA-MPSoC-Simulators simuliert jeweils einen Prozessortakt von jeder CPU sowie der Verbindungsstruktur nacheinander in einem einzigen Thread des Simulations-PCs. Es ist keine Synchronisierung erforderlich.

Pthread

Die parallele Ausführung der simulierten Prozessorkerne sowie der Verbindungsstruktur verspricht auf Mehrkernprozessoren eine Beschleunigung der Simulation. Um eine zyklengenaue Simulation des Prozessorclusters zu ermöglichen, müssen nach jedem simulierten Prozessortakt die nebenläufig ausgeführten Threads synchronisiert werden. Linux-Betriebssysteme stellen eine POSIX-konforme Threading-Bibliothek (*Pthread*) für die Synchronisierung verschiedener Threads zur Verfügung. Die einzelnen CPUs sowie die Verbindungsstruktur sind untereinander über zwei Pthread-Barrieren synchronisiert.

libfiber

Werden mehr CPUs simuliert, als physikalische CPUs auf dem Host-System vorhanden sind, kann dies zu einem Mehraufwand durch die Synchronisierung führen. Die Bibliothek libfiber implementiert ein $M : N$-Threading [206]. Hierbei werden M Software-Threads auf N physikalische Threads aufgeteilt (es gilt $M \geq N$). Um mehrere Software-Threads auf einem physikalischen Thread auszuführen ($M > N$), wird Zeitmultiplexing verwendet. Libfiber verwendet Pthread zur Synchronisierung der physikalischen Threads. Die Anzahl der Software-Threads wird durch die Anwendung vorgegeben, hier entspricht sie der Anzahl der simulierten CoreVA-CPUs sowie der Verbindungsstruktur. Die Anzahl der physikalischen Threads wird von eins bis acht variiert.

SystemC

SystemC ist eine C++-Klassenbibliothek zur Modellierung und Simulation von elektronischen Systemen und wird von der Accellera Systems Initiative [2] spezifiziert und weiterentwickelt. Die SystemC-Implementierung des Cluster-Simulators integriert die CPUs und die Verbindungsstruktur als SystemC-Module, wobei die Verbindungsstruktur nativ in SystemC implementiert ist. Jedes SystemC-Modul wird als eigener Thread realisiert, wobei bei der vorliegenden Implementierung QuickThreads eingesetzt werden [201].

Vergleich der Simulator-Implementierungen

Im Folgenden werden zwei Testprogramme verwendet, um die Performanz der vier unterschiedlichen Implementierungen des CPU-Cluster-Simulators zu vergleichen. Die Simulationen wurden auf einem Linux PC mit vier physikalischen CPU-Kernen[19] ausgeführt. Das erste Programm besteht aus einer 100-mal ausgeführten Schleife mit 100 leeren Instruktionen (*No Operation*, NOP). Hierdurch soll die Leistungsfähigkeit der Simulatoren für Programme ohne bzw. mit wenig Buszugriffen untersucht werden.

[19] Intel Xeon W3565 3,2 GHz, 24 GB RAM, Ubuntu Linux 10.04

Abbildung 3.5a zeigt die Ausführungsdauer eines simulierten Taktzyklus einer CoreVA-CPU für Prozessor-Cluster mit zwei bis 31 CPUs. Von jeder Konfiguration werden jeweils zehn Simulationen durchgeführt und das Ergebnis gemittelt. Obwohl die CPUs untereinander nicht kommunizieren, muss nach jedem simulierten Prozessortakt eine Synchronisierung stattfinden.

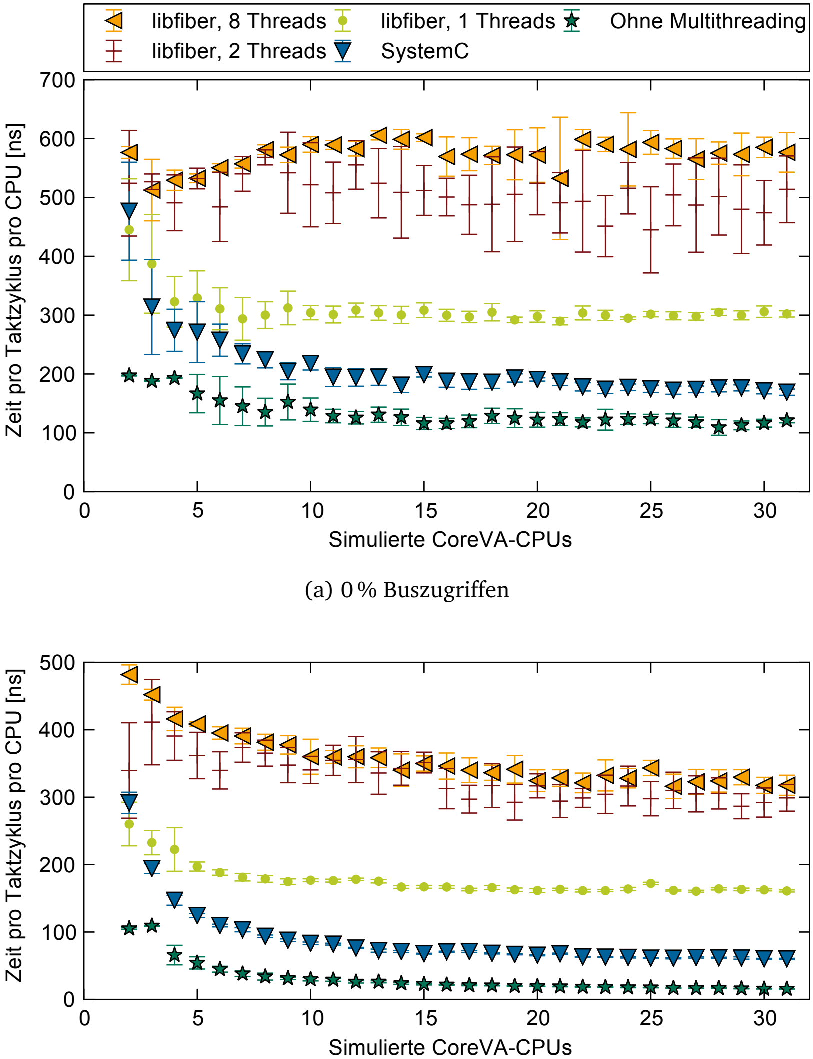
Um eine hohe Auslastung der Verbindungsstruktur zu erreichen, führt das zweite Programm 100-mal eine Schleife mit jeweils 100 Schreiboperationen aus (siehe Abbildung 3.5b). Jede CPU beschreibt hierbei den Datenspeicher der *nächsten* CPU (CPU1 → CPU2, CPU2 → CPU3, etc.). Aufgrund der hohen Busauslastung werden die einzelnen CPUs oft angehalten. Die Simulation einer angehaltenen CPU ist deutlich weniger aufwendig, daher ist die Simulationszeit pro simuliertem CPU-Takt bei hoher Busauslastung geringer als beim Testprogramm ohne Buszugriffe.

Der Pthread-basierte Simulator ist um den Faktor 20 langsamer als der Simulator ohne Nebenläufigkeit und daher aus Gründen der Übersichtlichkeit in Abbildung 3.5 nicht dargestellt. Die schlechte Performanz des Pthread-basierten Simulators ist auf die langsame Implementierung der Barriere in der Pthread-Bibliothek zurückzuführen.

Die von SystemC verwendete QuickThread-Bibliothek weist eine höhere Performanz als die in libfiber verwendete Barriere auf. Die schnellste Implementierung des Cluster-Simulators stellt die Variante ohne Nebenläufigkeit dar. Dieses Ergebnis ist insbesondere bemerkenswert, weil nur ca. 5 % der Simulationszeit einer CPU für die Simulation der Verbindungsstruktur aufgewendet werden (Pthread-Implementierung). Der Aufwand für Kontextwechsel und Synchronisierung kann durch die parallele Ausführung auf mehreren CPU-Kernen nicht aufgewogen werden. Insgesamt zeigt sich, dass die Simulationszeit stark von der Anzahl der verwendeten Hardware- und Software-Threads abhängt. Weitere Simulationsergebnisse für unterschiedliche Auslastungen der Verbindungsstruktur sind in [228] zu finden.

In [230] wurde der CoreVA-MPSoC-Simulator um das NoC sowie die Anbindung des NoCs an den CPU-Cluster erweitert. Mehrere Cluster-Simulatoren werden als eigenständige Prozesse gestartet und kommunizieren über gemeinsamen Speicher (*POSIX-Shared-Memory*) mit dem NoC-Simulationsmodell. Dieses bildet das Verhalten der Router und der einzelnen Verbindungen (*Links*) des NoCs nach.

Der Simulator des CoreVA-MPSoCs bietet eine Unterstützung für den GNU Debugger (GDB) [64]. GDB erlaubt eine Ablaufverfolgung der Ausführung einer Anwendung sowie die Betrachtung und Manipulation von Variablen zur Laufzeit.



(a) 0 % Buszugriffen

(b) 100 % Buszugriffe

Abbildung 3.5: Vergleich der Ausführungsgeschwindigkeit verschiedener Implementierungen des CoreVA-MPSoC-Simulators

3.3.2 FPGA-Emulation

Zur FPGA-Emulation wird das modulare *Rapid-Prototyping*-System RAPTOR [163] sowie der Miniroboter AMiRo verwendet. Beide Plattformen erlauben den Einsatz des CoreVA-MPSoCs in realen Umgebungen.

RAPTOR (siehe Abbildung 3.6) wird an der Arbeitsgruppe Kognitronik und Sensorik der Universität Bielefeld entwickelt. Als Basisboard stehen die drei Varianten RAPTOR2000, RAPTOR-X64 sowie RAPTOR-XPress zur Verfügung. Die Basisboards besitzen eine Schnittstelle zu einem Host-PC (PCI[20], PCI-X oder PCI Express). Zudem bieten RAPTOR-X64 und RAPTOR-XPress eine USB[21]-Schnittstelle für den Betrieb unabhängig vom Host-PC. Jedes Basisboard besitzt sechs (RAPTOR2000, RAPTOR-X64) bzw. vier (RAPTOR-XPress) Steckplätze für Erweiterungsmodule. Als Erweiterungsmodule stehen verschiedene FPGA- (z. B. Xilinx Spartan6, Virtex-5, Virtex-7) sowie I/O-Module (z. B. Ethernet) und ein Display zur Verfügung. Die FPGA-Module ermöglichen eine partielle Rekonfiguration der eingesetzten FPGAs zur Laufzeit. Zudem erlauben verschiedene parallele und serielle I/O-Schnittstellen eine schnelle Kommunikation der Module untereinander und mit dem Host-PC.

Die hohe Konfigurierbarkeit von RAPTOR erlaubt den schnellen und flexiblen Aufbau von Prototyping-Systemen. Ein Einsatzgebiet ist das im Rahmen dieser Arbeit beschriebene ASIC-Prototyping. Hierbei können ASICs mit mehreren hundert Millionen Transistoren emuliert werden. Ein weiteres Einsatzgebiet ist die beschleunigte Emulation neuronaler Netze (z. B. selbstorganisierender Karten) mithilfe von FPGAs [116]. Zudem ist das RAPTOR-System Basis einer Plattform zur dynamischen Rekonfiguration der Datenverarbeitung auf Satelliten [77].

[20] *Peripheral Component Interconnect*
[21] *Universal Serial Bus*

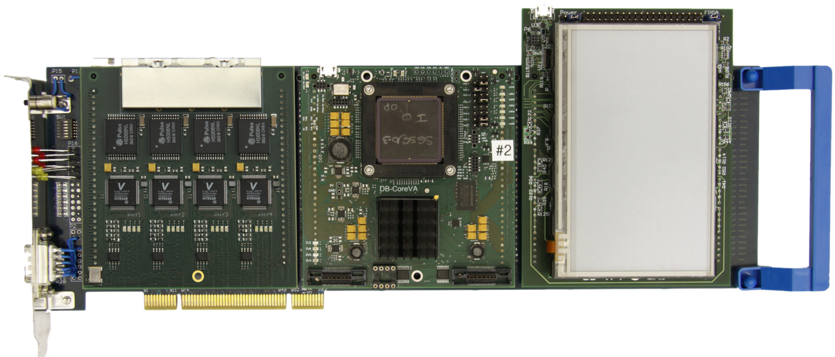


Abbildung 3.6: RAPTOR2000-Basisboard mit den drei Modulen DB-ETG, DB-CoreVA und DB-TFT (von links nach rechts)

Im Rahmen dieser Arbeit wird das CoreVA-MPSoC auf drei FPGA-Module abgebildet. Das Modul DB-CoreVA [132, S. 118 f.] ist für den Test der beiden in einer 65-nm-Technologie gefertigten ASIC-Prototypen der CoreVA-CPU entworfen (siehe Abschnitt 2.2). Dieses Modul verfügt über ein Xilinx Spartan-6 XC6SLX100 oder XC6SLX150-FPGA [179], einen Sockel für die ASIC-Prototypen der CoreVA-CPU sowie bis zu 512 MB DDR-SDRAM.

Die FPGA-Module DB-V5 [199] und DB-V7 [136; 207] integrieren jeweils zwei FPGAs und sind für eine Verwendung auf dem RAPTOR-XPress-System vorgesehen. Die sehr leistungsfähigen FPGAs Virtex-5 LX100T (DB-V5) und Virtex-7 VX690T (DB-V7) stehen dem Anwender für die Abbildung der eigenen Schaltung zur Verfügung. Jeweils ein weiterer FPGA (Virtex-5 LX30T bzw. Kintex-7 K70T) stellt unter anderem Zugriff auf die PCIe[22]-Schnittstelle bereit. Die FPGAs eines Moduls kommunizieren über eine Inter-FPGA-Schnittstelle [169]. Beide FPGA-Module verfügen über einen Steckplatz für DRAM-Speichermodule (DDR2 bzw. DDR3)

Als weitere Emulationsplattform wird der ebenfalls an der Arbeitsgruppe Kognitronik und Sensorik der Universität Bielefeld entwickelte Miniroboter **AMiRo**[23] (siehe Abbildung 1.1b) verwendet [91; 172]. Der AMiRo ist für den Einsatz in der wissenschaftlichen Forschung und in der universitären Lehre konzipiert und besitzt eine modulare Architektur. Mehrere runde Module können übereinander gesteckt werden, die Basisarchitektur besteht aus drei Modulen. Für die Kommunikation zwischen den Modulen wird unter anderem ein CAN[24]-Bus verwendet. Die beiden Motoren des AMiRos werden über das Modul *DiWheelDrive* ansteuert. Dieses Modul integriert zusätzlich vier Infrarotbasierte Annäherungssensoren, die an der Unterseite des Roboters angebracht sind. Das *PowerManagement*-Modul überwacht die beiden Akkumulatoren des Roboters und bindet eine Erweiterungsplatine zur Annäherungserkennung an. Abschließend bestitzt das *LightRing*-Modul acht farbige LEDs[25] und ermöglicht den Anschluss einer Kamera und eines Laserscanners. Zwischen den *PowerManagement*- und *LightRing*-Modulen können weitere Module gesteckt werden. Das *Cognition*-Modul integriert unter anderem einen SoC mit ARM Cortex-A8-CPU und Linux-Betriebssystem sowie WLAN[26]. Auf dem *ImageProcessing*-Modul ist ein Xilinx Spartan-6 XC6SLX100-FPGA integriert [68], welches für die Bildverarbeitung der optionalen Kamera vorgesehen ist. Im Rahmen der vorliegen Arbeit wird der FPGA jedoch für die prototypische Integration eines CoreVA-MPSoCs mit bis zu vier CPUs verwendet.

Die beiden FPGA-Prototypen (RAPTOR und AMiRo) des CoreVA-MPSoCs verwenden eine einheitliche Code-Basis, um die Entwicklung zu vereinheitlichen und somit zu vereinfachen. Hierzu wird eine Aufteilung des FPGA-Systems in zwei Komponenten vorgenommen. Hauptkomponente ist das CoreVA-MPSoC, welches über eine generi-

[22] *Peripheral Component Interconnect Express*
[23] *Autonomous Mini Robot*
[24] *Controller Area Network*
[25] *Light-Emitting Diode*
[26] *Wireless Local Area Network*

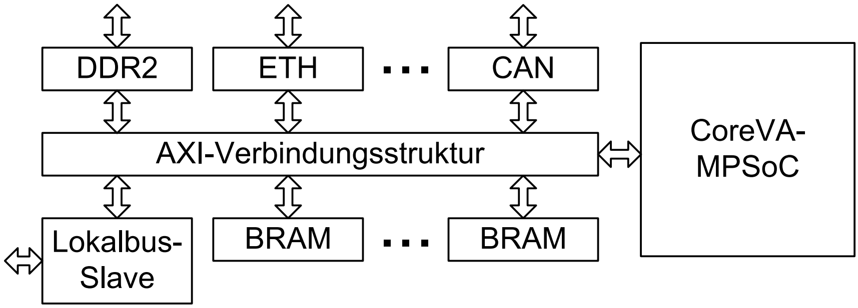


Abbildung 3.7: Peripherie-Subsystem des CoreVA-MPSoC-FPGA-Prototypen

sche Schnittstelle (siehe Abschnitt 2.5) verfügt. Das CoreVA-MPSoC kann aus einer einzelnen CPU, einem CPU-Cluster oder dem hierarchischen MPSoC mit NoC bestehen. Im Unterschied zur ASIC-Implementierung des CoreVA-MPSoCs (siehe Kapitel 5 sowie Abschnitt 6.2) werden für die SRAM-Speicher des Systems *Block RAM* (BRAM) von Xilinx verwendet.

FPGA-spezifische IP-Komponenten sind in einem Peripherie-Subsystem zusammengefasst, welches auf dem Xilinx Platform Studio (XPS) [212] basiert. XPS erlaubt eine einfache Verwaltung und Konfiguration von IP-Kernen, insbesondere FPGA-spezifische IP-Kerne von Xilinx. Die Verbindungsstruktur ist AXI-basiert und bezüglich der Topologie (Crossbar oder geteilter Bus) konfigurierbar (siehe Abschnitt 2.4.2). Die FPGA-Prototypen des CoreVA-MPSoCs integrieren beispielsweise einen Speichercontroller für DDR-SDRAM, eine Ethernet-MAC, BRAM-Speicher sowie eine Lokalbus-Schnittstelle für das RAPTOR-System bzw. eine CAN[27]-Schnittstelle für das AMiRo-System [245] (siehe Abbildung 3.7).

Für den Entwurf eines FPGA-Prototypen wird in einem ersten Schritt mit Synopsys Synplify Premier [188] eine Netzliste der CoreVA-CPU erstellt. Dies ist notwendig, da innerhalb der CoreVA-CPU verschiedene IP-Kerne aus der DesignWare-Bibliothek von Synopsys verwendet werden. Zudem wird mit XPS eine Netzliste des Peripherie-Subsystems erstellt. In einem letzten Schritt wird mit Xilinx ISE 14.7 [211] eine Konfigurationsdatei für das FPGA (Bitstrom) erzeugt.

In Abschnitt 6.1 wird der Ressourcenbedarf von den RAPTOR- und AMiRo-basierten FPGA-Prototypen des CoreVA-MPSoCs untersucht.

3.4 Zusammenfassung

In diesem Kapitel wurde eine Hardware- und Software-Entwurfsumgebung vorgestellt, die im Rahmen der vorliegenden Arbeit Anwendung findet. Der Hardware-Entwurfsab-

[27] *Controller Area Network*

lauf ermöglicht eine Chip-Realisierung des CoreVA-MPSoCs unter Verwendung einer 28-nm-Standardzellentechnologie. Als Basis für die Software-Entwurfsumgebung wurden Compiler für die Sprachen C, StreamIt und OpenCL eingeführt. Hierbei stellt insbesondere die automatisierte Partitionierung von Anwendungen auf MPSoCs eine offene Problemstellung dar. Der CoreVA-MPSoC-Compiler für die Sprache StreamIt zeigt neue Lösungsansätze für hierarchische eingebettete Multiprozessoren wie das CoreVA-MPSoC.

Anschließend wurden die verschiedenen Abstraktionsebenen eingeführt, auf welchen das CoreVA-MPSoC simuliert bzw. emuliert werden kann. Eine Simulation bzw. Emulation ist sowohl in der Software- als auch in der Hardwareentwicklung in allen Schritten des Entwurfs eines MPSoCs notwendig. Die Systemarchitektur und Implementierung des CoreVA-MPSoC-Instruktionssatzsimulators und verschiedener FPGA-Prototypen wurden detailliert beschrieben. Die vorgestellte Entwurfsumgebung wird für die Entwurfsraumexploration des CPU-Clusters des CoreVA-MPSoCs in Kapitel 5 sowie für die Realisierung von FPGA- und ASIC-Prototypen in Kapitel 6 verwendet. Zudem ist die Entwurfsumgebung Basis für die Modellierung der Hardware- und Softwarekomponenten des CoreVA-MPSoCs im folgenden Kapitel.

4 Bewertungsmaße und Modellierung

Eine Bewertung von verschiedenen Konfigurationen eines MPSoCs ist in allen Schritten der Entwicklung erforderlich, um Entwurfsentscheidungen treffen zu können. Aus diesem Grund werden in diesem Kapitel die verschiedene Ressourcen eines MPSoCs sowie Bewertungsmaße eingeführt. Zudem ist es notwendig, bereits in einer frühen Entwurfsphase (z. B. nach der abstrakten Spezifikation, siehe Abschnitt 3.3) die Ressourceneffizienz eines MPSoCs bzw. des gesamten Hardware-/Softwaresystems zu bewerten. Hierzu werden in diesem Kapitel verschiedene analytische Modelle sowohl der Software- als auch der Hardwarekomponenten des CoreVA-MPSoCs vorgestellt.

4.1 Ressourcen und Bewertungsmaße eingebetteter Multiprozessorsysteme

Ressourcen sind Betriebsmitteln, die für den Betrieb eines Systems erforderlich sind. Im Kontext von eingebetteten mikroelektronischen Systemen existieren die Ressourcen Chipfläche, Leistungsaufnahme sowie Performanz [143, S. 59 f.; 165, S. 54 f.]

Die **Chipfläche** A bezeichnet die Fläche, die eine Schaltung benötigt, wenn sie auf eine bestimmte Herstellungstechnologie abgebildet wird. 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. 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 3.1). Von der Chipfläche lassen sich die **Herstellungskosten** ableiten (diese hängen maßgeblich von der Fertigungstechnologie ab) sowie das benötigte Volumen des eingepassten Chips (*Chip-Package*).

Eine digitale, synchrone Schaltung wird mit einer **Taktfrequenz** f betrieben. Zudem besitzt jede Schaltung eine maximale Taktfrequenz f_{max}, bei der ein fehlerfreier Betrieb möglich ist. Prozessorarchitekturen mit mehreren parallelen Ausführungseinheiten können mehrere Instruktionen pro Takt (*Instructions per Cycle*, IPC) ausführen. Zusammen mit der Taktfrequenz ergibt sich hieraus die Anzahl an Milliarden ausgeführter Operationen pro Sekunde (*Giga Operations Per Second*, GOPS).

Die **Leistungsaufnahme** P von CMOS-Schaltungen setzt sich aus der dynamischen und der statischen Verlustleitung zusammen [143, S. 65 f.; 111, S. 269 f.]:

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

Die statische Verlustleistung lässt sich vor allem auf Subschwellströme von sperrenden Transistoren zurückführen [132, S. 24 f.] und wird auch als *Leakage* bezeichnet. Die dynamische Verlustleistung entsteht, wenn sich ein oder mehrere Eingangssignale eines (kombinatorischen) Gatters ändern. Dies führt zum einen zu einem Umladen von Lastkapazitäten innerhalb des Gatters sowie an dessen Ausgang. Zum anderen tritt ein Querstrom auf, wenn sowohl die p-Kanal- als auch die n-Kanal-Transistoren des Gatters kurzzeitig zeitgleich leitend sind. Der Querstrom wird auch als Kurzschlussstrom bezeichnet. Die dynamische Verlustleistung besteht zu ca. 90 % aus dem Umladen von Lastkapazitäten [165, S. 56]. Aus diesem Grund kann die dynamische Verlustleistung durch folgende Gleichung approximiert werden [111, 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. Eine hohe Verlustleistung 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.

Von der Verlustleistung kann die pro Takt umgesetzte **Energie** abgeleitet werden, die unabhängig von der Frequenz ist [132, 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) Ausführung einer Anwendung verwendet werden.

Der Begriff **Performanz** fasst mehrere Kenngrößen zusammen, die das zeitliche Verhalten eines mikroelektronischen Systems bei der Bearbeitung einer Aufgabe (z. B. die Ausführung einer Anwendung) charakterisieren. 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 (Byte pro Sekunde) angegeben. Der Kehrwert des Durchsatzes ist die Ausführungszeit. 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:

$$B = \frac{D_1}{D_0}. \tag{4.4}$$

Im Rahmen dieser Arbeit ist die Basiskonfiguration zumeist die Ausführung der Anwendung auf einer CPU.

Aus der Beschleunigung und der Chipfläche kann als weiteres Bewertungsmaß die **Beschleunigung pro Chipfläche** BA abgeleitet werden:

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

Für die Basiskonfiguration, die zur Bestimmung der Beschleunigung verwendet wird, gilt $D_1 = D_0$ und somit $BA = D_0/(A \cdot D_0) = 1/A$.

Die **Latenz** L kennzeichnet in digitalen Systemen die Zeit von der Eingabe bis zu dem Zeitpunkt, an dem die entsprechende Ausgabe das System verlässt. Besonders in Systemen mit Echtzeitanforderungen stellt die Latenz eine wichtige Kennzahl für die Performanz dar. Kann nur ein Ergebniswert gleichzeitig von einem System berechnet werden, so entspricht die Latenz der Ausführungszeit. Werden in einem System mehrere Ergebniswerte parallel verarbeitet (z. B. durch Pipelining), kann die Ausführungszeit geringer als die Latenz ausfallen. **Jitter** ist die Varianz der Latenz. Ein Jitter kann z. B. auftreten, wenn die Dauer einer Berechnung von den Eingangsdaten abhängt.

Die **Speicherkapazität** gibt an, wie viele Daten das mikroelektronische System speichern kann. Bei eingebetteten Systemen ist der Speicher zumeist als SRAM-, DRAM- und/oder Flash-Speicher integriert (siehe Abschnitt 2.3). Anwendungen haben typischerweise einen minimalen Speicherbedarf, der für die Ausführung benötigt wird. Zusätzlicher Speicher kann verwendet werden, um z. B. durch größere Verarbeitungsblöcke den Durchsatz zu erhöhen (siehe Abschnitt 3.2.3).

Bei der Bewertung der Verbindungsstruktur eines Multiprozessor-Systems beschreibt die **maximale Übertragungsbandbreite** die Kapazität jedes Kommunikationslinks multipliziert mit der Anzahl dieser Links [35, S. 521]. Zur Bestimmung der **Bisektionsbandbreite** wird die Verbindungsstruktur in zwei gleich große Teile geteilt, sodass die Anzahl der Verbindungen zwischen den Teilen minimal ist. Die Bisektionsbandbreite ist die Summe der Übertragungsbandbreite dieser Verbindungen. Sie ist ein realistischeres Maß für die Bewertung der verfügbaren Bandbreite eines MPSoCs als die maximale Bandbreite [236, S. 12]. Der **Durchmesser** (*Diameter*) bezeichnet die maximale Distanz zwischen den Kommunikationspartnern. Bei einer hierarchischen Verbindungsstruktur wie dem CoreVA-MPSoC ist dies die Summe aus der maximalen Distanz zwischen zwei CPU-Clustern sowie die Verzögerungen (z. B. durch Registerstufen) innerhalb der Cluster.

Die **Latenz der Kommunikation** bezeichnet die Verzögerungszeit bzw. die Anzahl an Takten, die eine Übertragung von einem Master zu einem Slave benötigt. Die Größe der

Latenz ist insbesondere bei (blockierenden) Lesezugriffen über die Verbindungsstruktur von Bedeutung. Wichtig ist in diesem Zusammenhang, dass die Latenz einer Anwendung nicht direkt von der Latenz der Kommunikation im MPSoC abgeleitet werden kann.

Fairness bewertet den gleichberechtigten Zugriff aller Master auf die Verbindungsstruktur bzw. die Slaves. Entscheidende Bedeutung besitzt hierbei die Bus-Arbitrierung (siehe Abschnitt 2.4.5).

Über die bisher genannten Bewertungsmaße hinaus werden in [143, S. 68 f.] Wiederverwendbarkeit, Fehlertoleranz, Skalierbarkeit sowie Programmierbarkeit genannt.

4.2 Modellierung der Ausführungszeit von parallelen Anwendungen

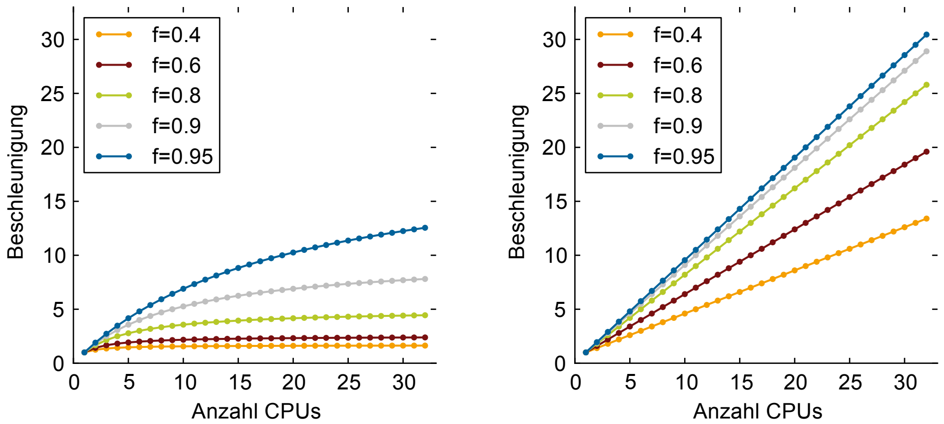
Gene Amdahl hat bereits 1967 ein Modell zur Beschleunigung von Anwendungen durch die Ausführung auf parallelen Architekturen veröffentlicht [7]. Amdahl nimmt an, dass eine Anwendung aus einem parallelisierbaren Anteil $f \in [0,1]$ sowie einem sequenziellen Anteil $(1-f)$ besteht. Bei der Abbildung der Anwendung auf eine Architektur mit der Parallelität m ergibt sich die Beschleunigung B [200, S. 77 f.]:

$$B_{Amdahl} = \frac{1}{(1-f)+\frac{f}{m}}. \tag{4.6}$$

Diese Gleichung wird auch als Amdahls Gesetz bezeichnet. Besitzt eine Anwendung beispielsweise einen parallelisierbaren Anteil von 80 %, so ergibt sich bei einer Ausführung auf 8 CPUs eine Beschleunigung um den Faktor 3,33 im Vergleich zu der Ausführung auf einer CPU. Bei einer Verdopplung der CPUs auf 16 liegt die Beschleunigung gegenüber einer CPU bei einem Faktor von 4,0 bzw. lediglich 20 % mehr verglichen mit 8 CPUs. Der sequenzielle Programmbestandteil von 20 % beschränkt die Skalierbarkeit deutlich. In Abbildung 4.1a ist die Beschleunigung nach Amdahl bei einem parallelisierbaren Anteil einer Anwendung von 40 % bis 95 % dargestellt. Wenn die Parallelität einer Architektur ins unendliche skaliert wird, ergibt sich eine obere Grenze für die Beschleunigung einer Anwendung [186]:

$$\lim_{m\to\infty} B_{Amdahl} = \frac{1}{1-f}. \tag{4.7}$$

Amdahl berücksichtigt hierbei jedoch nicht, dass sich durch die Parallelisierung der sequenzielle Anteil der Anwendung vergrößern kann. Ein Beispiel ist der Mehraufwand durch das Aufteilen der Eingangsdaten auf die parallelen Rechenknoten (siehe Abschnitt 5.4). Eine Möglichkeit, die Beschränkungen durch Amdahl zu überwinden, ist die Verwendung heterogener Multiprozessoren [200, S. 79 f.]. Diese besitzen eine (oder wenige) leistungsfähige CPUs für die Verarbeitung des sequenziellen Anteils der Anwendung sowie viele kleine, ressourceneffiziente Kerne für die Verarbeitung des



(a) Beschleunigung nach Amdahl

(b) Beschleunigung nach Gustafson

Abbildung 4.1: Theoretische Beschleunigung von Anwendungen durch die Verwendung von Multiprozessoren gegenüber der Ausführung auf einer CPU

parallelen Teils. Eine weitere Möglichkeit zur Überwindung von Amdahls Gesetz ist die Ausführung von einer Vielzahl von Anwendungen auf einem Multiprozessorsystem.

Nach Amdahl wird immer die gleiche Problemgröße einer Anwendung auf beiden (Multi-) Prozessorsystemen vermessen. Stattdessen schlägt Gustafson vor, die Problemgröße zu skalieren und die Anwendung auf beiden Systemen gleich lange auszuführen [69; 70]. Hierdurch ist es möglich, auf massiv-parallelen Systemen Problemgrößen zu verwenden, die auf einem Einprozessorsystem, z. B. durch begrenzten Speicher, nicht ausgeführt werden können. Bei vielen Algorithmen erhöht sich der parallele Anteil stärker als der sequenzielle, wenn die Problemgröße erhöht wird (z. B. Matrixmultiplikation, siehe Abschnitt 5.1). Durch größere Problemgrößen skalieren diese Algorithmen demnach besser auf massiv-parallelen Systemen. Gustafson führt zudem an, dass manche Anwendungen eine festgelegte (maximale) Ausführungszeit besitzen und der Anwender in dieser Zeit bestmögliche Resultate wünscht. Beispiele sind die Berechnung eines Bildes eines Computerspiels, für das die Grafikkarte z. B. 25 ms Zeit hat. Auf einer schnelleren (parallelen) Grafikkarte kann der Anwender z. B. die Auflösung erhöhen und erhält somit ein *besseres* Resultat. Ein weiteres Beispiel ist die Genauigkeit bei numerischen Berechnungen. Wenn ausschließlich der parallele Programmteil von einer größeren Problemgröße beeinflusst wird ergibt sich nach [186]:

$$B_{Gustafson} = (1 - f) + mf. \tag{4.8}$$

Das Beispiel einer Anwendung mit 80 % parallelem Anteil ergibt gegenüber der Ausführung auf einer CPU eine Beschleunigung um dem Faktor 6,60 für 8 CPUs sowie 13,00 für 16 CPUs (siehe Abbildung 4.1b). Für viele Anwendungen kann die Problemgröße jedoch nicht sinnvoll erhöht werden.

Sun und Ni [185] führen bereits 1993 an, dass der verfügbare Speicher pro CPU oft die Skalierbarkeit von Anwendungen begrenzt. In dem von ihnen vorgeschlagene Modell besitzt jeder CPU-Kern einen Speicher fester Größe. Durch das Hinzufügen von CPUs zum System steigt somit auch die insgesamt verfügbare Speichergröße. Sun und Ni gehen hierbei von einem eng gekoppeltem Speicher bzw. L1-Cache aus (siehe Abschnitt 2.3). Speicher höherer Hierarchien (z. B. Level 2 oder DRAM Speicher) weisen eine deutlich höhere Latenz auf und begrenzen die Performanz einer Anwendung stark, falls sie verwendet werden müssen. Sei $g(x)$ die Funktion der Zunahme des parallelen Teils einer Anwendung, wenn der Speicher um den Faktor x vergrößert wird. Für Anwendungen mit polynomieller Komplexität kann der größte Faktor von $g(x)$ verwendet werden, der im Folgenden mit $\overline{g}(x)$ bezeichnet wird [185; 186]. Hieraus ergibt sich das Gesetz nach Sun und Ni:

$$B_{Sun\_und\_Ni} = \frac{(1-f) + f \cdot \overline{g}(m)}{(1-f) + \frac{f \cdot \overline{g}(m)}{m}}. \tag{4.9}$$

Als Beispiel ist in [186] eine Matrixmultiplikation angegeben, bei der der Rechenaufwand mit dem Faktor $2N^3$ sowie der Speicherbedarf mit $3N^2$ skaliert. N gibt hierbei die Dimension der beiden Eingabe-Matrizen an. Hieraus ergibt sich $\overline{g}(m) = m^{\frac{3}{2}}$ sowie

$$B_{Sun\_und\_Ni} = \frac{(1-f) + f \cdot m^{\frac{3}{2}}}{(1-f) + f \cdot m^{\frac{1}{2}}}. \tag{4.10}$$

Unter der Annahme, dass jedes Speicherelement zumindest einmal verwendet wird, gilt $B_{Sun\_und\_Ni} \geq B_{Gustafson}$. Das speicherbegrenzte Modell zeigt also eine gleiche oder höhere Beschleunigung als das Modell mit fester Ausführungszeit [186].

Die bisher vorgestellten Modelle gehen alle davon aus, dass der parallele Anteil einer Anwendung beliebig aufgeteilt werden kann und hierdurch keine zusätzlichen Kosten (z. B. Taktzyklen oder Speicherbedarf) entstehen. Vajda [200, S. 90] unterteilt im Gegensatz hierzu Parallelität in Daten- und funktionale Parallelität.

Ein Anwendungsteil besitzt **Datenparallelität**, wenn aufeinanderfolgende Ausführungen des Anwendungsteils unabhängig voneinander sind. Der Anwendungsteil kann als zustandslos (*Stateless*) bezeichnet werden. Die Ein- und Ausgaben müssen aufgeteilt bzw. zusammengefügt werden, wobei ein Mehraufwand entstehen kann. Die Skalierbarkeit von Datenparallelität hängt von diesem Mehraufwand ab. Datenparallelität kann auch auf Prozessorebene durch die Verwendung von SIMD-, VLIW- oder superskalaren Prozessorarchitekturen ausgenutzt werden (siehe Abschnitt 2.2).

Funktionale Parallelität bezeichnet zwei (unterschiedliche) Teile einer Anwendung, die nebenläufig (auf verschiedenen CPUs) ausgeführt werden können. Diese nicht weiter parallelisierbaren Teile einer Anwendung werden im Folgenden als atomare (Anwendungs-) Blöcke bezeichnet.

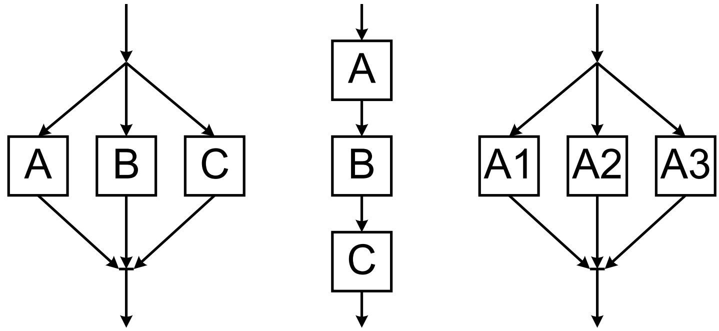


Abbildung 4.2: Arten von Parallelität: Task- (links), Pipeline- (mitte) und Datenparallelität (rechts)

Gordon et al. [67] unterteilen funktionale Parallelität in Task- und Pipeline-Parallelität (siehe Abbildung 4.2). **Task-Parallelität** bezeichnet Anwendungsteile, die auf den gleichen Eingabedaten verschiedene Berechnungen durchführen, die unabhängig voneinander sind. Voraussetzung hierzu ist, dass jeder Anwendungsteil niemals die Eingangsdaten des jeweils anderen Teiles verarbeitet. Als Beispiel kann die Funktion $h(x) = f(x) + g(x)$ genannt werden, wobei f und g auf zwei CPUs parallelisiert werden können [200, S. 90]. Um Task-Parallelität ausnutzen zu können, müssen die Eingabedaten aufgeteilt (*Split*) bzw. dupliziert sowie zusammengefügt (*Join*) werden. Analog zur Datenparallelität kann dies zu einem Mehraufwand führen, durch den eine Aufteilung auf mehrere CPUs nicht sinnvoll sein kann. Zudem besitzen viele Anwendungen nur eine beschränkte Task-Parallelität oder es ist sehr aufwendig für den Programmierer, diese zu erkennen und zu implementieren.

Pipeline-Parallelität erlaubt eine Verkettung von Anwendungsteilen, wobei die Ausgabe eines Teils (Erzeuger) als Eingabe des nächsten dient (Konsument). Hierbei muss sichergestellt sein, dass der Erzeuger immer ausreichend Daten für den Konsumenten produziert. Ein Beispiel ist die Funktion $h(x) = f(g(x))$. g und f können auf zwei CPUs ausgeführt werden, wobei das Ergebnis von g als Eingabe von f dient. Während f auf einem Datensatz ausgeführt wird, kann für einen nächsten Datensatz bereits g berechnet werden. Um eine Anwendung effizient auf ein Multiprozessorsystem zu partitionieren, ist zumeist die Ausnutzung von allen Arten von Parallelität notwendig [67]. Hierbei muss die funktionale Parallelität vom Programmierer erkannt und modelliert werden, während Datenparallelität von einem Compiler automatisch ausgenutzt werden kann. Die Partitionierung der einzelnen Anwendungsteile auf die zur Verfügung stehenden CPUs ist von entscheidender Bedeutung für die Performanz der Anwendung und Gegenstand aktueller Forschungstätigkeit (siehe Abschnitt 3.2.3).

4.2.1 Stand der Technik

Die Abschätzung der Ausführungszeit von Anwendungen auf einzelnen CPUs oder Multiprozessorsystemen ist Gegenstand einer Vielzahl von wissenschaftlichen Arbeiten. Patel und Rajawat geben in [154] einen Überblick über Arbeiten zur Bewertung der Performanz von eingebetteten Systemen. Hierbei wird zwischen einer analytischen Modellierung und simulationsbasierten Ansätzen unterschieden. In der vorliegenden Arbeit wird eine analytische Modellierung vorgestellt. Verschiedene Simulatoren des CoreVA-MPSoCs werden in Abschnitt 3.3 beschrieben.

Ein Schwerpunkt vieler Arbeiten liegt bei einer frühzeitigen Bewertung verschiedener Hardware-Software-Partitionierungen sowie der Evaluierung verschiedener (heterogener) Hardwarearchitekturen für die Ausführung einer spezifischen Anwendung [190]. Hierbei wird eine Vielzahl an unterschiedlichen Architekturen in kurzer Zeit evaluiert.

Sesame [158] ist ein Beispiel für eine Plattform zur Entwurfsraumexploration eingebetteter Systeme zu einem frühen Zeitpunkt im Entwurfsprozess. In einem ersten Schritt wird eine Vielzahl an möglichen Architekturen anhand eines analytischen Modells untersucht. Hierauf aufbauend werden anschließend ausgewählte Architekturen auf Systemebene simuliert und optimiert.

Die Abschätzung der Skalierbarkeit einer Anwendung bei der Ausführung auf einem Many-Core-System mit hunderten an CPU-Kernen ist ein weiterer Schwerpunkt der Forschung. Grundlage hierfür bilden die in diesem Kapitel bereits vorgestellten Arbeiten von Amdahl [7], Gustafson [70] sowie Sun und Ni [185].

Hill und Marty [93] betrachten die Skalierung von parallelen Anwendungen bei der Ausführung auf homogenen und heterogenen MPSoCs. Die Anwendungen verfügen wie bei Amdahl über einen parallelisierbaren Anteil $f \in [0, 1]$ sowie einen sequenziellen Anteil $1 - f$. Ein heterogener MPSoC kombiniert eine leistungsfähige CPU mit vielen ressourceneffizienten CPU-Kernen. Die leistungsfähige CPU weist einen hohen Flächenbedarf auf und führt den sequenziellen Teil der Anwendung aus. Der parallele Teil wird auf den ressourceneffizienten CPU-Kernen berechnet. Hill und Marty berücksichtigen den Flächenbedarf der verschiedenen CPUs und zeigen, dass nur Anwendungen mit einer hohen Parallelität von vielen ressourceneffizienten CPU-Kernen profitieren. Eine heterogene Architektur zeigt einen deutlich geringeren Einfluss des sequenziellen Anteils auf die Performanz der Anwendung.

Esmaeilzadeh et al. [55] betrachten die Performanz von parallelen Anwendungen bei sechs verschiedenen Technologiestufen (45 nm bis 8 nm). Hierbei wird unter anderem das Modell von Hill und Marty um die Leistungsaufnahme erweitert. Für Flächenbedarf und Leistungsaufnahme eines MPSoCs werden Obergrenzen festgelegt. Bei acht von 12 betrachteten Anwendungen wird die Skalierbarkeit durch eine zu geringe Parallelität beschränkt. Vier Anwendungen werden durch die Leistungsaufnahme des MPSoCs begrenzt und besitzen eine ausreichende Parallelität.

Das im Rahmen dieser Arbeit entwickelte analytische Modell sowie das in [216] veröffentlichte simulationsbasierte Modell haben das Ziel, für eine gegebene Konfiguration die Performanz des CoreVA-MPSoCs abzuschätzen. Diese Zielsetzung unterscheidet sich von einem Großteil der vorgestellten Forschungsarbeiten, die vor allem für Hardware/-Software-Codesign entwickelt worden sind (siehe hierzu die Einleitung im Kapitel 3). In Abschnitt 4.3 wird ein analytisches Modell für den Flächenbedarf des CoreVA-MPSoC-CPU-Clusters vorgestellt.

4.2.2 Ein analytisches Modell für die Performanz von parallelen Anwendungen

Die Modelle nach Amdahl, Gustafson sowie Sun und Ni lassen offen, welche Eigenschaften der parallelisierbare Anteil einer Anwendung besitzen muss, um eine Aufteilung auf mehrere CPUs zu ermöglichen. Für Amdahls Gesetz kann nur Datenparallelität angewendet werden, da funktionale Parallelität bei gleicher Problemgröße nur eine konstante Nebenläufigkeit besitzen. Eine Skalierung der Problemgröße nach Gustafson ist nach allen drei Arten der Parallelität möglich. Die funktionale Parallelität skaliert allerdings nicht bei allen Anwendungen mit der Problemgröße. Das Modell nach Sun und Ni kann ebenfalls von allen Arten der Parallelität profitieren, jedoch führt das Aufteilen und Zusammenfügen von Datenströmen (bei Daten- und Task-Parallelität) oft zu einem erheblichen Speicher-Mehraufwand. Insbesondere bei funktionaler Parallelität kann zudem nicht mehr von einer Aufteilung auf sequenziellen und parallelisierbaren Programmteil gesprochen werden. Die CPU mit der höchsten Gesamtlaufzeit bestimmt die Gesamtlaufzeit des Gesamtsystems bzw. stellt den Flaschenhals für den Durchsatz dar:

$$Gesamtlaufzeit = \max(Laufzeiten\ aller\ CPUs). \tag{4.11}$$

Im Folgenden wird ein Modell entwickelt, welches die Skalierbarkeit von Anwendungen mit funktionaler und Datenparallelität für Multiprozessorsysteme modelliert. Dieses Modell wurde als Teil des Konferenzbeitrags [216] veröffentlicht. Es wird eine Anwendung betrachtet, die bei der Ausführung auf einer einzelnen CPU die Ausführungszeit w besitzt. Es wird angenommen, dass die Problemgröße der Anwendung konstant ist. Zunächst wird eine Anwendung betrachtet, bei der alle Anwendungsteile Datenparallelität besitzen. Für den parallelisierbaren Anteil f der Anwendung gilt $f = 1$. Hierbei ist es unerheblich, ob die Anwendung darüber hinaus funktionale Parallelität besitzt. Abhängig von der CPU-Anzahl m ergibt sich nach Gleichung 4.8 eine lineare Beschleunigung

$$B_{Daten} = (1 - f) + mf = m. \tag{4.12}$$

Dieser Zusammenhang ist in Abbildung 4.3a grafisch dargestellt. Hierbei und im Folgenden wird vorausgesetzt, dass alle Anwendungsteile gleichmäßig bzw. optimal auf alle verfügbaren CPUs partitioniert werden.

Eine weitere Anwendung besitzt eine funktionale Parallelität von $p \in \mathbb{N}$, es ist also eine Aufteilung der Anwendung in p atomare Anwendungsblöcke möglich. Jeder atomare Block kann nicht weiter parallelisiert werden und besitzt somit keine weitere Task-, Daten- oder Pipeline-Parallelität. Es wird angenommen, dass alle atomaren Blöcke die gleiche Ausführungszeit besitzen:

$$w_{Block} = \frac{w}{p}. \tag{4.13}$$

Diese Einschränkung ermöglicht das Aufstellen eines analytischen Modells. Eine verallgemeinerte Betrachtung von Anwendungen mit atomaren Blöcken beliebiger Laufzeit ist in Abschnitt 4.2.3 zu finden.

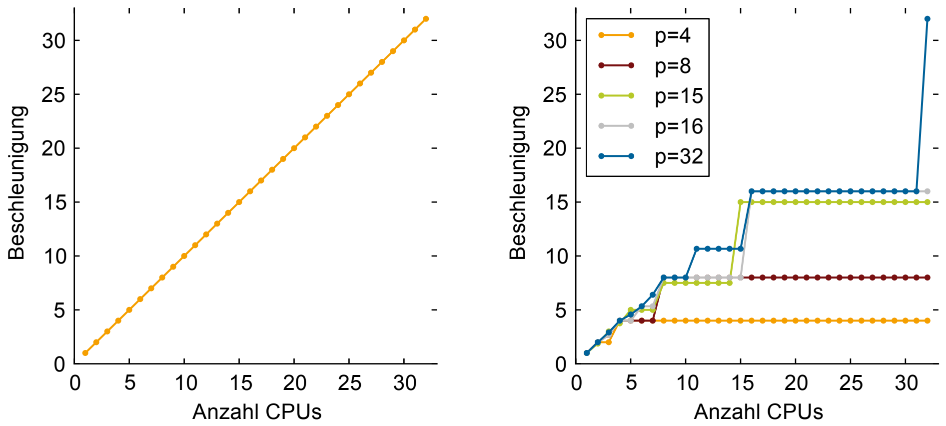
Ist der Quotient der Anzahl von Blöcken p und CPUs m eine natürliche Zahl, also $\frac{p}{m} \in \mathbb{N}$, so ergibt sich für die parallelisierte Ausführungszeit

$$w_{funktional\_ideal} = \frac{p}{m} \cdot w_{Block} = \frac{p}{m} \cdot \frac{w}{p} = \frac{w}{m}. \tag{4.14}$$

Hieraus folgt eine lineare Beschleunigung für Anwendungen mit funktionaler Parallelität p und $\frac{p}{m} \in \mathbb{N}$ von

$$B_{funktional\_ideal} = \frac{w}{w_{funktional\_ideal}} = \frac{w}{\frac{w}{m}} = m. \tag{4.15}$$

Für beliebige $p, m \in \mathbb{N}$, also $\frac{p}{m} \notin \mathbb{N}$, müssen einige CPUs mehr atomare Blöcke ausführen als andere. Eine oder mehrere CPUs führen $\left\lceil \frac{p}{m} \right\rceil$ atomare Blöcke aus. Diese



(a) Datenparallelität

(b) Funktionale Parallelität, 4 bis 32 atomare Blöcke

Abbildung 4.3: Beschleunigung für Anwendungen mit reiner Daten- bzw. funktionaler Parallelität gegenüber der Ausführung auf einer CPU

CPUs werden als kritische oder Flaschenhals-CPUs bezeichnet und sind definiert als die CPU(s), welche die Laufzeit der Anwendung bestimmen. Für die Ausführungszeit bedeutet dies

$$w_{funktional} = \left\lceil \frac{p}{m} \right\rceil \cdot \frac{w}{p}. \tag{4.16}$$

Hieraus folgt für die Beschleunigung

$$B_{funktional} = \frac{w}{w_{funktional}} = \frac{w}{\left\lceil \frac{p}{m} \right\rceil \cdot \frac{w}{p}} = \frac{p}{\left\lceil \frac{p}{m} \right\rceil}. \tag{4.17}$$

In Abbildung 4.3b ist dieser Zusammenhang für Anwendungen mit vier bis 32 atomaren Blöcken grafisch dargestellt.

Anwendungen besitzen typischerweise nur begrenzte funktionale Parallelität. Zudem existieren Anwendungen mit reiner Datenparallelität in der Praxis nicht (wie schon von Amdahl in [7] beschrieben). Aus diesem Grund wird im Folgenden ein analytisches Modell entwickelt, welches funktionale und Datenparallelität von Anwendungen berücksichtigt. Die Anwendung besitzt neben p atomaren Blöcken eine gewisse Datenparallelität $f' \in [0, 1]$. Diese kann aus einem Programmteil (siehe Abbildung 4.2) oder mehreren Teilen bestehen, die an verschiedenen Stellen im Anwendungsgraph als Blöcke integriert sind (siehe Abbildung 4.4). Der datenparallele Anteil der Anwendung f' kann ohne zusätzliche Kosten auf beliebig viele CPU-Kerne parallelisiert werden. Zudem können Anteile von f' zusammen mit einem oder mehreren atomaren Blöcken auf eine CPU partitioniert werden. Der Anteil an f', den einzelnen CPUs ausführen, kann daher unterschiedlich groß sein. Die Ausführungszeit eines einzelnen atomaren Blockes ist

$$w_{Block'} = \begin{cases} \frac{w}{p} \cdot (1 - f') & \text{falls} \quad p \in \mathbb{N} \\ 0 & \text{falls} \quad p = 0, \end{cases} \tag{4.18}$$

da eine Anwendung mit reiner Datenparallelität ($f' = 1$) keine funktionale Parallelität und somit keine atomaren Blöcke ($p = 0$) besitzt.

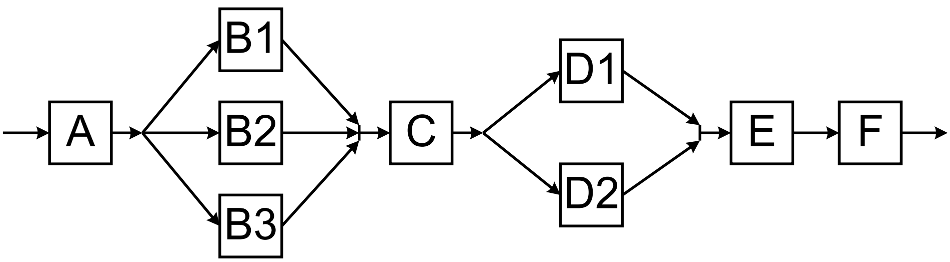


Abbildung 4.4: Beispielanwendung mit Daten- und funktionaler Parallelität und $p = 4$ (A, C, E und F). Die datenparallelen Blöcke B und D sind beispielhaft drei- bzw. zweimal aufgeteilt.

In einem ersten Schritt werden alle atomaren Blöcke p auf die zur Verfügung stehenden CPUs m aufgeteilt. Analog zur vorangegangenen Betrachtung mit rein funktionaler Parallelität müssen manche CPUs mehr atomare Blöcke ausführen als andere. Dies ist genau der Fall[1], falls $\frac{p}{m} \notin \mathbb{N}$. Die Anzahl der Flaschenhals- oder kritischen CPUs ergibt sich aus

$$M'_{kritisch} = p \bmod m \tag{4.19}$$

mit $M'_{kritisch} \in \{1, .., m-1\}$.

Jede kritische CPU führt dabei $I = \left\lceil \frac{p}{m} \right\rceil$ atomare Blöcke aus. Gilt allerdings $\frac{p}{m} \in \mathbb{N}$, sind alle CPUs kritische CPUs ($M_{kritisch} = m$, alle CPUs begrenzen die Performanz der Anwendung) und $M_{kritisch} \in \{1, .., m\}$. In diesem Fall muss Gleichung 4.19 erweitert werden, um bei der Verwendung der mod-Operation $M_{kritisch} = 0$ zu vermeiden:

$$M_{kritisch} = ((p + m - 1) \bmod m) + 1. \tag{4.20}$$

Die Anzahl der unkritischen CPUs ist

$$M_{unkritisch} = m - M_{kritisch} = m - 1 - (p + m - 1) \bmod m. \tag{4.21}$$

Jede nicht voll ausgelastete CPU $M_{unkritisch}$ kann noch genau eine Pipeline-Stufe der Laufzeit p ausführen, ohne dass sich hierdurch der Durchsatz der Anwendung verschlechtert[1]. In der Summe ergibt dies eine *verfügbare* CPU-Zeit von

$$\begin{aligned} w_{verfügbar} &= w_{Block'} \cdot M_{unkritisch} \\ &= \begin{cases} \frac{w}{p} \cdot (1 - f') \cdot (m - 1 - (p + m - 1) \bmod m) & \text{falls } p \in \mathbb{N} \\ 0 & \text{falls } p = 0. \end{cases} \end{aligned} \tag{4.22}$$

Falls $w \cdot f' < w_{verfügbar}$, so kann der datenparallele Anwendungsteil f' vollständig auf die unkritischen CPUs partitioniert werden, ohne den Durchsatz der Anwendung zu beeinflussen. Mit Gleichung 4.16 gilt

$$w_{Daten+funktional'} = w_{funktional} = \left\lceil \frac{p}{m} \right\rceil \cdot \frac{w}{p} \cdot (1 - f'). \tag{4.23}$$

Falls $w \cdot f' \geq w_{verfügbar}$, so verlängert sich die Gesamtlaufzeit der Anwendung um den Anteil von f', der größer ist als $w_{verfügbar}$, aufgeteilt auf alle CPUs:

$$w_{zusätzlich} = \frac{1}{m} \cdot (w \cdot f' - w_{verfügbar}). \tag{4.24}$$

Für die Laufzeit der Anwendung folgt hieraus

$$\begin{aligned} w_{Daten+funktional} &= w_{Daten+funktional'} + w_{zusätzlich} \\ &= \left\lceil \frac{p}{m} \right\rceil \cdot \frac{w}{p} \cdot (1 - f') + \frac{1}{m} \cdot (w \cdot f' - w_{verfügbar}). \end{aligned} \tag{4.25}$$

[1]Unter der Annahme einer optimalen Partitionierung

Da $w_{zusätzlich}$ gleichmäßig auf alle CPUs partitioniert werden kann, ist in diesem Fall, wie bei einer Anwendung mit reiner Datenparallelität, eine lineare Abnahme der Laufzeit der Anwendung abhängig von m zu erwarten (siehe Gleichung 4.12):

$$w_{Daten+funktional} = \frac{w}{m} \text{ falls } w \cdot f' \geq w_{verfügbar}. \tag{4.26}$$

Diese Aussage soll für $p \in \mathbb{N}_0, m \in \mathbb{N}$ im Folgenden in drei Fällen bewiesen werden.
1. Fall: $p = 0$. In diesem Fall gilt $f' = 1$ (siehe Gleichung 4.18) sowie (da alle CPUs gleich ausgelastet werden können) $w_{Daten+funktional'} = 0$ und somit

$$\begin{aligned} w_{Daten+funktional} &= w_{Daten+funktional'} + w_{zusätzlich} \\ &= 0 + \frac{1}{m} \cdot (w \cdot f' - w_{verfügbar}) \\ &= \frac{1}{m} \cdot (w \cdot 1 - 0) \\ &= \frac{w}{m}. \end{aligned} \tag{4.27}$$

2. Fall: $\frac{p}{m} \in \mathbb{N}$. Es gilt $w_{verfügbar} = 0$, da in diesem Fall alle CPUs durch den funktional-parallelen Anteil des Programms gleichmäßig ausgelastet werden. Zudem gilt in diesem Fall $\left\lceil \frac{p}{m} \right\rceil = \frac{p}{m}$. Dies erlaubt eine Vereinfachung von Gleichung 4.25 zu

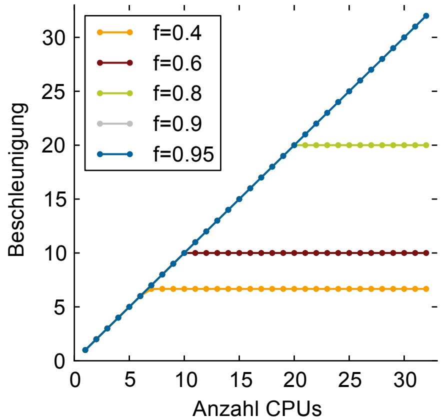
$$\begin{aligned} w_{Daten+funktional} &= w_{Daten+funktional'} + w_{zusätzlich} \\ &= \left\lceil \frac{p}{m} \right\rceil \cdot \frac{w}{p} \cdot (1 - f') + \frac{1}{m} \cdot (w \cdot f' - w_{verfügbar}) \\ &= \frac{p}{m} \cdot \frac{w}{p} \cdot (1 - f') + \frac{1}{m} \cdot (w \cdot f' - 0) \\ &= \frac{w}{m} \cdot ((1 - f') + f') \\ &= \frac{w}{m}. \end{aligned} \tag{4.28}$$

3. Fall $\frac{p}{m} \notin \mathbb{N}$. Hier gilt $M_{kritisch} = M'_{kritisch}$ sowie $M'_{unkritisch} = m - p \bmod m$ und

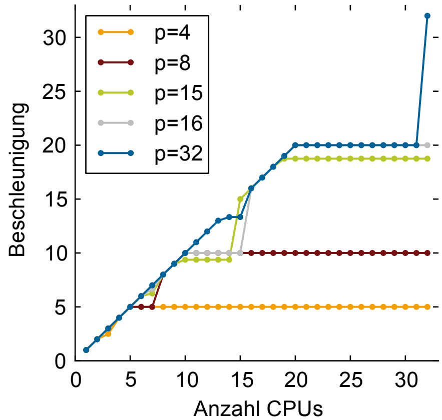
$$w_{verfügbar'} = w_{Block'} \cdot M'_{unkritisch} = \frac{w}{p} \cdot (1 - f') \cdot (m - p \bmod m). \tag{4.29}$$

Die Laufzeit der Anwendung ergibt sich zu

$$\begin{aligned} &w_{Daten+funktional} = w_{Daten+funktional'} + w_{zusätzlich} \\ &= \left\lceil \frac{p}{m} \right\rceil \cdot \frac{w}{p} \cdot (1 - f') + \frac{1}{m} \cdot (w \cdot f' - w_{verfügbar'}) \\ &= \left\lceil \frac{p}{m} \right\rceil \cdot \frac{w}{p} \cdot (1 - f') + \frac{1}{m} \cdot \left(w \cdot f' - \frac{w}{p} \cdot (1 - f') \cdot (m - p \bmod m) \right). \end{aligned} \tag{4.30}$$



(a) 40 % bis 95 % Datenparallelität sowie 4 atomare Blöcke



(b) 4 bis 32 atomare Blöcke sowie 20 % Datenparallelität

Abbildung 4.5: Beschleunigung für Anwendungen mit kombinierter Daten- und funktionaler Parallelität gegenüber der Ausführung auf einer CPU

Mit $p \bmod m = p - \left\lfloor \frac{p}{m} \right\rfloor \cdot m$ und (da $p \in \mathbb{N}_0, m \in \mathbb{N}$) $\left\lfloor \frac{p}{m} \right\rfloor = \left\lceil \frac{p}{m} \right\rceil - 1$ folgt

$$w_{Daten+funktional} = \frac{w}{m} = w_{Daten}. \tag{4.31}$$

Die Herleitung hierzu ist im Anhang B zu finden. Werden die Gleichungen 4.23 und 4.26 kombiniert, ergibt sich für die Beschleunigung einer Anwendung mit Daten- als auch funktionaler Parallelität

$$B_{Daten+funktional} = \begin{cases} \frac{w}{w_{Pipeline}} & = & \frac{p}{\left\lceil \frac{p}{m} \right\rceil \cdot (1-f')} & w \cdot f' < w_{verfügbar} \\ \frac{w}{w_{Daten}} & = & m & w \cdot f' \geq w_{verfügbar}. \end{cases} \tag{4.32}$$

Hierbei entspricht f' dem datenparallelen Anteil der Anwendung, $w_{verfügbar}$ ist in Gleichung 4.22 beschrieben. Die Abbildungen 4.5a und 4.5b stellen Gleichung 4.32 grafisch dar, wobei hier der datenparallele bzw. der funktional-parallele Programmanteil variiert wird.

4.2.3 Integration des analytischen Modells in den CoreVA-MPSoC-Compiler

Bisher wurde angenommen, dass alle atomaren Blöcke, die die funktionale Parallelität der Anwendung darstellen, die gleiche Laufzeit besitzen. Im Folgenden wird diese

Annahme aufgegeben und das vorgestellte analytische Modell entsprechend angepasst. Zudem wird die Integration des Modells in den CoreVA-MPSoC-Compiler für Streaming-Anwendungen (siehe Abschnitt 3.2.3) vorgestellt.

Die parallele Programmiersprache StreamIt [195] teilt alle atomaren Blöcke (Filter) einer Anwendung in *Stateless*-Filter mit Datenparallelität sowie *Stateful*-Filter ohne ebendiese ein. Eine StreamIt-Anwendung besteht aus einem gerichteten Graphen, dessen Knoten den Filtern entsprechen. Hierdurch wird implizit eine mögliche parallele Ausführung von Filtern und somit funktionale Parallelität ausgedrückt (z. B. durch eine Pipeline von Filtern).

Um das in diesem Abschnitt beschriebene Modell auf StreamIt-Programme anwenden zu können, muss die Laufzeit aller Filter einer Anwendung bekannt sein. Hierfür liest der CoreVA-MPSoC-Compiler die entsprechende Anwendung ein und erzeugt C-Code für die Ausführung auf einer CoreVA-CPU. Anschließend wird der C-Code mit dem LLVM-Compiler übersetzt und mit dem ISS des CoreVA-MPSoCs simuliert. Durch den Aufruf von speziellen Trace-Befehlen vor und nach der Ausführung von jedem Filter wird die Ausführungszeit aller Filter bestimmt. Um das analytische Modell für Anwendungen zu verwenden, die in einer anderen Programmiersprache als StreamIt implementiert sind, muss die Möglichkeit bestehen, die Laufzeit aller atomaren Blöcke der Anwendung zu bestimmen.

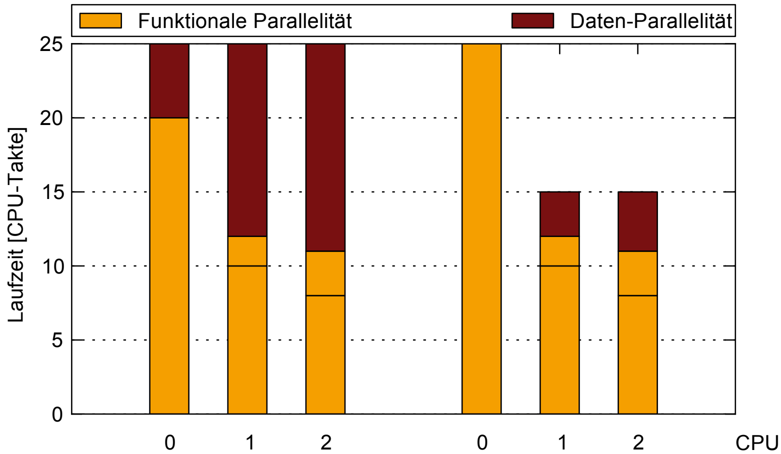
Die Ausführungszeit w bei der Ausführung auf einer einzelnen CPU ist die Summe der Laufzeit aller Filter für einer Iteration der Anwendung (*Steady-State*-Iteration). Das Verhältnis der Ausführungszeiten von allen datenparallelen Filtern und der gesamten sequenziellen Ausführungszeit entspricht dem datenparallelen Anteil der Anwendung f'. Es werden alle funktional-parallelen Filter auf die vorhandenen CPUs m partitioniert. Hierzu wird ein Greedy-Algorithmus verwendet (siehe Abschnitt 3.2.3). Die CPU dieser Partitionierung, die die größte Rechenlast aufweist (Flaschenhals-CPU), besitzt die Laufzeit w_{max} und bestimmt die Laufzeit der gesamten Anwendung.

Von dieser Partitionierung wird $w_{verfügbar}$ bestimmt und überprüft, ob $f' \geq w_{verfügbar}$ ist. Hieraus kann nun die Beschleunigung der Anwendung abgeschätzt werden:

$$B = \begin{cases} \frac{w}{w_{max}} & \text{falls } w \cdot f' < w_{verfügbar} \\ m & \text{falls } w \cdot f' \geq w_{verfügbar}. \end{cases} \tag{4.33}$$

Eine Beispielanwendung mit $w = 75$ CPU-Takten und $f' = 0{,}43$ ist in Abbildung 4.6a dargestellt. Die Anwendung wird auf drei CPUs aufgeteilt. Hieraus folgt $w_{max} = 20$ und $w_{verfügbar} = 17$. Der datenparallele Teil der Anwendung ist mit $w \cdot f' = 75 \cdot 0{,}43 = 32{,}25$ größer als $w_{verfügbar}$. Dies führt mit Gleichung 4.33 zu einer Beschleunigung gegenüber der Ausführung auf einer CPU von $B = 3$, was genau der CPU-Anzahl entspricht.

In Abbildung 4.6b ist eine Anwendung mit $w = 55$, $f' = 0{,}13$, $w_{max} = 25$ und $w_{verfügbar} = 27$ dargestellt. Da der datenparallele Teil der Anwendung mit $w \cdot f' = 7{,}15$ kleiner als $w_{verfügbar}$ ist, ergibt sich als Beschleunigung $B = 55/25 = 2{,}2$.



(a) $w \cdot f' \geq w_{verfügbar}$ (b) $w \cdot f' < w_{verfügbar}$

Abbildung 4.6: Beispiel von zwei Anwendungen mit funktionaler und Datenparallelität

Das vorgestellte Modell kann als obere Grenze für die Beschleunigung einer Anwendung bei einer Ausführung auf m CPUs gelten. Hierbei wird der Mehraufwand bezüglich Taktzyklen und Speicherbedarf für die Kommunikation zwischen Filtern, die Aufteilung von datenparallelen Filtern sowie eine mögliche Überlast der Kommunikationsinfrastruktur nicht berücksichtigt. Diese Aussage ist für die im Rahmen dieser Arbeit betrachteten Anwendungen und MPSoC-Konfigurationen gültig, kann jedoch durch Effekte wie z. B. eine superskalare Beschleunigung verletzt werden. Ein möglicher Grund für eine superskalare Beschleunigung sind Optimierungen, die der LLVM-Compiler bei der Übersetzung vornimmt. Ein Beispiel ist das Abrollen von Schleifen. Zudem findet der Greedy-Algorithmus, der zur Partitionierung der funktional-parallelen Filter verwendet wird, nicht zwangsläufig eine optimale Partitionierung.

In Abbildung 4.7 sind die Abschätzungen des Modells für neun StreamIt-Anwendungen und einem CPU-Cluster mit vier bis 32 CPUs dargestellt. Eine Beschreibung der verwendeten Streaming-Anwendungen ist in Abschnitt 5.1 zu finden. Für CPU-Cluster mit vier CPUs sagt das Modell eine lineare Beschleunigung von 4,0 gegenüber der Ausführung auf einer einzelnen CPU voraus.

BubbleSort ist die einzige Anwendung ohne datenparallele Filter. Da die Anwendung jedoch 66 Filter bei einer sequenziellen Laufzeit von $w = 2125$ Takte besitzt, beträgt für einen CPU-Cluster mit acht CPUs $w_{verfügbar}$ lediglich 91 Takte. Dies führt zu einer annähernd idealen Beschleunigung von 7,8. Die Anwendung MatrixMult weist für acht CPUs eine Beschleunigung von 6,8 auf. MatrixMult besitzt eine sequenziel-

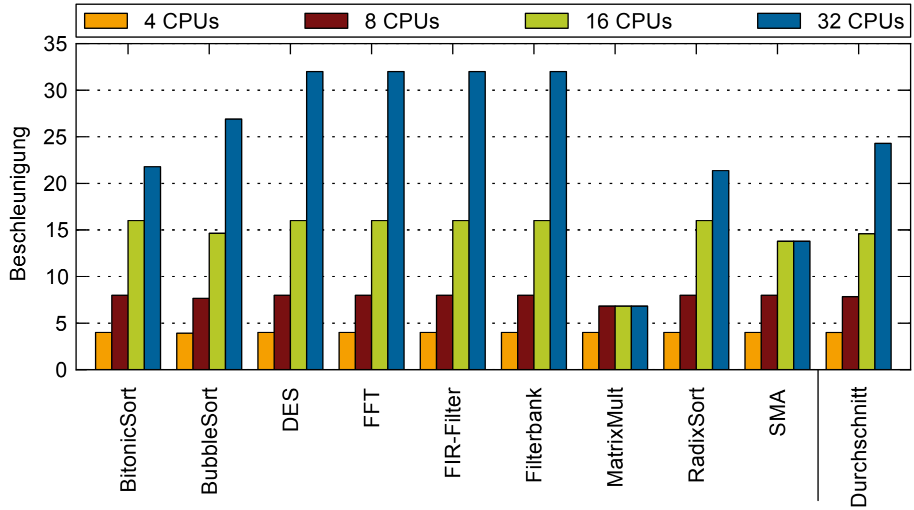


Abbildung 4.7: Abgeschätzte Beschleunigung von StreamIt-Anwendungen gegenüber der Ausführung auf einer CPU

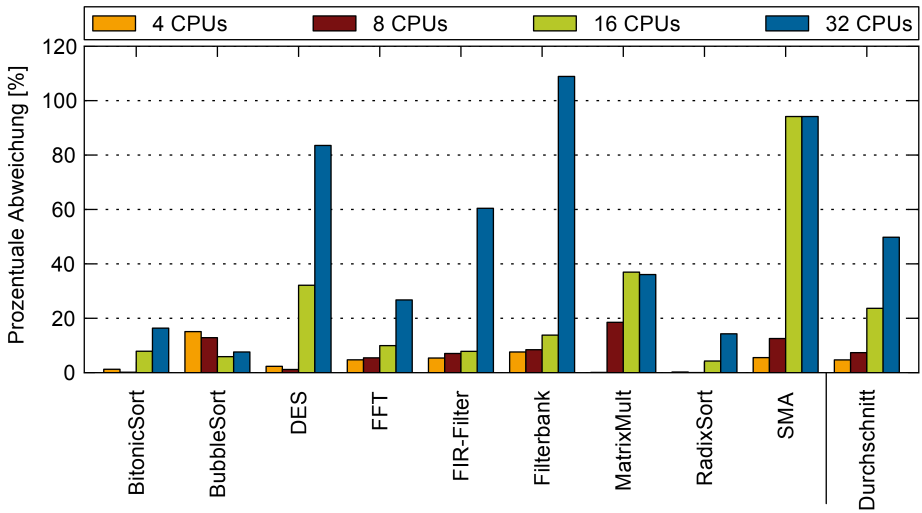


Abbildung 4.8: Prozentuale Abweichung des analytischen Modells verglichen mit einer RTL-Simulation der Anwendungen

le Laufzeit von $w = 7909$ Takten und einen datenparallelen Anteil von $f' = 0{,}17$. Durch die Partitionierung der funktional-parallelen Filter ergibt sich $w_{max} = 1157$ Takte und $w_{verfügbar} = 2667$ Takte. Somit ist $w_{verfügbar}$ größer als der datenparallele Teil der Anwendung ($w \cdot f' = 1320$ Takte) und für die Beschleunigung folgt $B = w/w_{max} = 7909/1157 = 6{,}8$. Alle weiteren Anwendungen zeigen mit acht CPUs eine Beschleunigung von 8,0.

Die Konfigurationen mit 16 CPUs zeigen eine durchschnittliche Beschleunigung von 14,6. Die Anwendungen BubbleSort, MatrixMult und SMA weisen keine ideale Beschleunigung auf. MatrixMult zeigt gegenüber der Konfiguration mit acht CPUs keine weitere Verbesserung, da bereits für 8 CPUs der funktional-parallele Anteil der Anwendung die Ausführungszeit bestimmt. Bei der Cluster-Konfiguration mit 32 CPUs besitzen die betrachteten Anwendungen eine durchschnittliche Beschleunigung von 24,3. Die Anwendung FIR-Filter besteht, bis auf Ein- und Ausgabefilter, nur aus datenparallelen Filtern. Dies führt zu einem datenparallelen Anteil von $f' = 0{,}98$ und bei der Verwendung von 32 CPUs zu einer Beschleunigung von 32 gegenüber der Ausführung auf einer CPU. Die Anwendungen DES, FFT und Filterbank zeigen ebenfalls eine Beschleunigung von 32. Alle weiteren betrachteten Anwendungen zeigen keine ideale Beschleunigung, da die funktional-parallelen Filter die Laufzeit bestimmen.

Im Folgenden wird die prozentuale Abweichung („Fehler“) des analytischen Modells gegenüber einer RTL-Simulation der Anwendungen auf dem CoreVA-MPSoC betrachtet. Hierzu ist es notwendig, Ergebnissen aus Abschnitt 5.5 vorzugreifen. Die Anwendungen werden unter Verwendung des CoreVA-MPSoC-Compilers für die verschiedenen Cluster-Konfiguration partitioniert. Anschließend wird für jede CPU eine ausführbare Datei erstellt und die Anwendung mithilfe einer RTL-Simulation des CoreVA-MPSoCs ausgeführt. Der CPU-Cluster besitzt eine AXI-Verbindungsstruktur mit einer Crossbar-Topologie.

Der arithmetische Mittelwerte der Fehler aller Anwendungen und aller CPU-Cluster-Konfigurationen beträgt 21,4 % (siehe Abbildung 4.8). Der Fehler der betrachteten Anwendungen unterscheidet sich stark. Den geringsten durchschnittliche prozentuale Abweichung besitzt RadixSort mit 4,7 %. SMA weist mit 51,6 % den höchsten Fehler auf, da die Laufzeit dieser Anwendung durch das Aufteilen von Daten begrenzt wird (siehe Abschnitt 5.5.1). Die Konfigurationen mit vier und acht CPUs zeigen mit 4,7 % und 7,4 % die geringste, die Konfiguration mit 32 CPUs mit 49,8 % die höchste Abweichung. Die geringere Abweichung bei der Verwendung von wenig CPUs ist darauf zurückzuführen, dass hier weniger zwischen den CPUs kommuniziert wird[2].

Die vorgestellte Abschätzung der Beschleunigung kann von Entwicklern zur Bewertung einer StreamIt-Anwendung bzw. des CoreVA-MPSoC-Compilers verwendet werden. Besitzt eine Anwendung beispielsweise zu wenig datenparallele Filter, kann der Entwickler einen anderen Algorithmus mit höherer Parallelität auswählen. Für die Entwicklung des CoreVA-MPSoC-Compilers, der Synchronisierungs- und Kommu-

[2] Der CoreVA-MPSoC-Compiler versucht, die Kommunikationskosten zwischen CPUs zu minimieren

nikationsbibliotheken oder der Hardware-Architektur des CoreVA-MPSoCs zeigt ein hoher Unterschied zwischen abgeschätzter und tatsächlicher Laufzeit einer Anwendung zudem Ansätze für mögliche Verbesserungen auf.

4.2.4 Ein Modell zur simulationsbasierten Abschätzung der Ausführungszeit

Im Rahmen der Entwicklung des CoreVA-MPSoC-Compilers (siehe Abschnitt 3.2.3) ist ein Werkzeug zur Schätzung der Performanz von Streaming-Anwendungen („*Performance-Estimator*“) entstanden, um die Performanz verschiedener Partitionierungen vergleichen zu können. Da der Schätzer analytische und simulationsbasierte Ansätze vereint, wird von simulationsbasierter Abschätzung (*Simulation-Based Estimation*, SBE) gesprochen. SBE wurde in der Veröffentlichung [216] beschrieben.

Das Verfahren bestimmt durch Simulationen die Ausführungszeit von einzelnen StreamIt-Filtern, Kommunikationsfunktionen (siehe Abschnitt 2.6 und Abschnitt 5.3) sowie den Aufwand für das Aufteilen 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 (siehe Abschnitt 3.2.3).

In Abbildung 4.9 ist der Fehler von SBE gegenüber einer Ausführung auf dem Instruktionssatzsimulator des CoreVA-MPSoCs für die Anwendung DES dargestellt. Es werden CPU-Cluster mit zwei CPUs (1x1x2) bis 16 CPUs (1x1x16) sowie verschiedene hierarchische CoreVA-MPSoC-Konfigurationen betrachtet. Die Konfiguration 2x2x4 besteht beispielsweise aus CPU-Clustern mit je vier CPUs, die über ein 2x2-NoC verbunden sind. Der mittlere Fehler der Abschätzung von SBE für die Anwendung DES beträgt 2,5 %, wobei die Konfiguration 2x1x8 mit 4,8 % den höchsten Fehler aufweist. Gegenüber einer Ausführung auf dem Instruktionssatzsimulator kann durch SBE eine Beschleunigung der Abschätzung um den Faktor 2848 erreicht werden. Weitere Informationen zu SBE sind in der Veröffentlichung [216] sowie im technischen Bericht [213] zu finden.

SBE und das im Rahmen dieser Arbeit entwickelte analytische Modell (siehe vorheriger Abschnitt) werden für verschiedene Einsatzzwecke im CoreVA-MPSoC-Projekt verwendet. Das analytische Modell wird als Referenz für die Abschätzung der theoretischen Leistungsfähigkeit der Hardware- und Softwarekomponenten eingesetzt. Durch die Verwendung des analytischen Modells wird die Identifizierung von Komponenten mit Verbesserungspotential deutlich vereinfacht. Im Gegensatz hierzu wird SBE im CoreVA-MPSoC-Compiler eingesetzt, um die Performanz einer zu evaluierenden Partitionierung einer Anwendung sehr schnell und möglichst genau zu bestimmen.

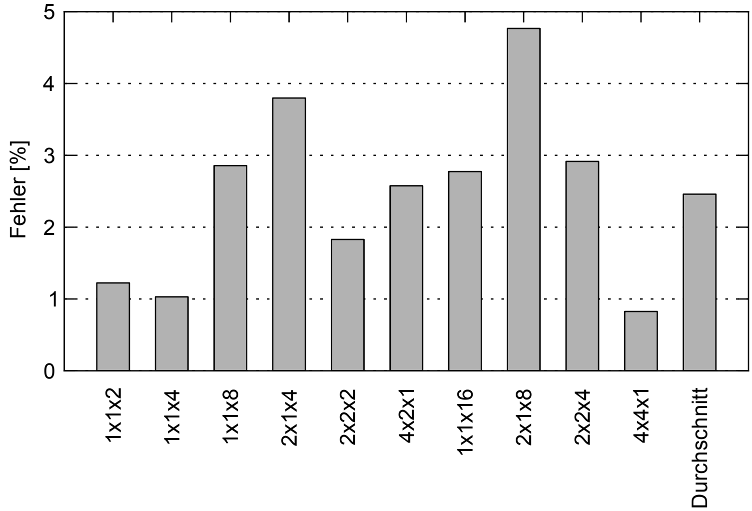


Abbildung 4.9: Fehler des SBE-Modells verglichen mit der Ausführung auf dem Instruktionssatzsimulator des CoreVA-MPSoCs für die Anwendung DES [216]

4.3 Ein Modell für den Hardware-Ressourcenbedarf des CoreVA-MPSoCs

In diesem Abschnitt wird ein Modell für den Flächenbedarf eines CPU-Clusters des CoreVA-MPSoCs vorgestellt. Die Flächenabschätzungen der verschiedenen Komponenten des CPU-Clusters basieren auf der in Kapitel 5 vorgestellten Entwurfsraumexploration und Synthesen in einer 28-nm-FD-SOI-Technologie. Als Taktfrequenz für das Modell wird die maximale Frequenz der CPU-Makros (800 MHz) verwendet. Für andere Taktfrequenzen müsste das Modell bzw. die Abschätzungen der Einzelkomponenten entsprechend angepasst werden. Die Modellierung des Flächenbedarfs verschiedener Konfigurationen der CPU-Pipeline (siehe Abschnitt 2.2) und der NoC-Verbindungsstruktur sind Bestandteil aktueller Forschung im Rahmen des CoreVA-MPSoC-Projektes.

Die Fläche eines CPU-Makros A_{CPU} ergibt sich aus der Summe der Flächen der CPU-Pipeline ($A_{CPU\_Pipeline}$), des Instruktions- und Datenspeichers (A_{IMEM}, A_{DMEM}) sowie der CPU-Cluster-Schnittstelle (A_{FIFO}, siehe Gleichung 4.34). Die Fläche von weiteren CPU-Komponenten wie der Trace-Schnittstelle oder der CPU-Kontrolllogik ist in $A_{CPU\_Pipeline}$ enthalten.

$$A_{CPU} = A_{CPU\_Pipeline} + A_{IMEM} + A_{DMEM} + A_{FIFO} \tag{4.34}$$

Für die Fläche der CPU-Pipeline wird für eine 1-Slot-CPU 0,11 mm^2 angenommen. Eine 2-Slot-CPU mit jeweils einer LD/ST- und einer MAC-Einheit belegt eine Fläche von

Tabelle 4.1: Statischer und dynamischer Flächenbedarf der Cluster-Verbindungsstruktur

Verbindungsstruktur	A_{Bus_stat} [µm²]	A_{Bus_dyn} [µm²]
Wishbone-Bus	2000	4000
Wishbone-Crossbar	4000	685
AXI-Bus	4000	9500
AXI-Crossbar	20000	1100

0,13 mm². Eine 4-Slot-CPU mit einer LD/ST- und zwei MAC-Einheiten belegt 0,30 mm². Der Instruktionsspeicher der 1-Slot- und der 2-Slot-CPUs ist aus zwei Speichermakros mit einer Größe von 8 kB aufgebaut. Die 4-Slot-CPU besitzt einen Instruktionsspeicher mit vier jeweils 4 kB großen Speichermakros. Der Datenspeicher verwendet immer zwei 8-kB-Speichermakros. Ein 4-kB-Speichermakro ist 9343 µm² groß, ein 8-kB-Makro 15 093 µm². Die CPU-Cluster-Schnittstelle besitzt eine Fläche von 1322 µm² für ein FIFO mit zwei Einträgen (A_{FIFO_stat}) und zusätzlich 605 µm² für jeden weiteren Eintrag (A_{FIFO_dyn}). A_{FIFO} ergibt sich aus Gleichung 4.35, wobei n der FIFO-Tiefe entspricht.

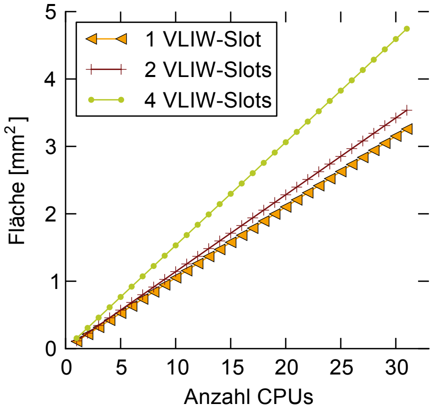
$$A_{FIFO} = A_{FIFO\_stat} + A_{FIFO\_dyn} \cdot (n-2) \tag{4.35}$$

Die Fläche eines CPU-Clusters $A_{CPU\_Cluster}$ setzt sich aus den Flächen der CPU-Makros A_{CPU}, der Verbindungsstruktur bei der Verwendung von einer CPU (A_{Bus_stat}) sowie einem Mehraufwand für die Integration von weiteren CPUs (A_{Bus_dyn}) zusammen (siehe Gleichung 4.36). A_{Bus_stat} und A_{Bus_dyn} sind von dem verwendeten Bus-Standard (AXI oder Wishbone) sowie der Topologie abhängig (siehe Tabelle 4.1). Mit m wird die Anzahl der CPU-Kerne des Clusters bezeichnet. Bei der Verwendung einer Crossbar-Verbindungsstruktur muss m quadratisch berücksichtigt werden (siehe Gleichung 4.37). Dies ist darauf zurückzuführen, dass bei einer Crossbar jeder Master direkt mit jedem Slave verbunden ist.

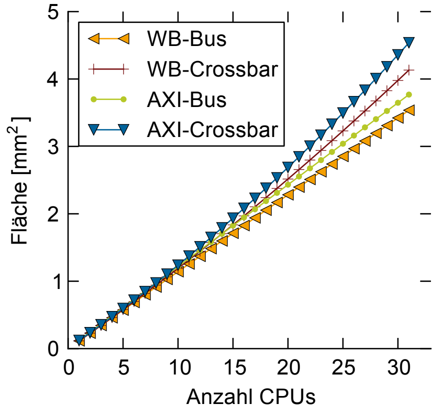
$$A_{CPU\_Cluster} = A_{Bus\_stat} + A_{Bus\_dyn} \cdot m + A_{CPU} \cdot m \tag{4.36}$$

$$A_{CPU\_Cluster} = A_{Bus\_stat} + A_{Bus\_dyn} \cdot m^2 + A_{CPU} \cdot m \tag{4.37}$$

In Abbildung 4.10a ist der abgeschätzte Flächenbedarf für CPU-Cluster mit einer bis 32 CPUs dargestellt, wobei die Anzahl der VLIW-Slots der CPUs von eins bis vier variiert wird. Als Fläche der CPUs werden die in Abschnitt 5.2.3 angegebenen Werte der dort vorgestellten CPU-Makros verwendet. Der CPU-Cluster integriert einen geteilten AXI-Bus als Verbindungsstruktur. Die Fläche der drei verschiedenen Konfigurationen steigt linear mit der Anzahl der CPUs an. Für einen CPU-Cluster mit zwei CPUs, die jeweils einen VLIW-Slot besitzen, beträgt der abgeschätzte Flächenbedarf 0,23 mm². Zwei CPUs mit je zwei bzw. vier VLIW-Slots belegen 0,25 mm² bzw. 0,33 mm². CPU-



(a) Verschiedene CoreVA-CPUs



(b) Verschiedene Bus-Standards und Topologien

Abbildung 4.10: Abschätzung der Fläche verschiedener Konfigurationen des CPU-Clusters

Cluster mit 32 CPUs belegen 3,60 mm^2 mit einem VLIW-Slot, 3,86 mm^2 mit zwei VLIW-Slots sowie 5,14 mm^2 für vier VLIW-Slots.

Der abgeschätzte Flächenbedarf eines CPU-Clusters unter Verwendung verschiedener Bus-Standards und Topologien ist in Abbildung 4.10b dargestellt. Jede CPU besitzt jeweils 16 kB Instruktions- und Datenspeicher sowie zwei VLIW-Slots. Der CPU-Cluster belegt mit 32 CPUs und geteiltem Wishbone-Bus eine Fläche von 3,71 mm^2. Die Konfiguration mit Wishbone-Crossbar belegt 4,29 mm^2. Die AXI-Konfigurationen haben eine Fläche von 3,89 mm^2 (geteilter Bus) und 4,73 mm^2 (Crossbar).

Der Fehler (bzw. die prozentuale Abweichung) des Modells gegenüber den in Abschnitt 5.2.4 und Abschnitt 5.5.3 vorgestellten Syntheseergebnissen beträgt maximal 1,5 %. Das Modell für den Flächenbedarf des CPU-Clusters ist in die CoreVA-MPSoC-GUI integriert (siehe nächster Abschnitt).

4.4 Eine abstrakte Systembeschreibung des CoreVA-MPSoCs

Der Entwurfsraum des CoreVA-MPSoCs ist durch die drei Hierarchiestufen CPU, CPU-Cluster und MPSoC bzw. NoC sowie verschiedene Speicher und Arten der Synchronisierung sehr groß. Eine einheitliche, abstrakte Beschreibung aller relevanten Hardware- und Softwarekomponenten des MPSoCs erleichtert die Arbeit in einem Entwicklerteam.

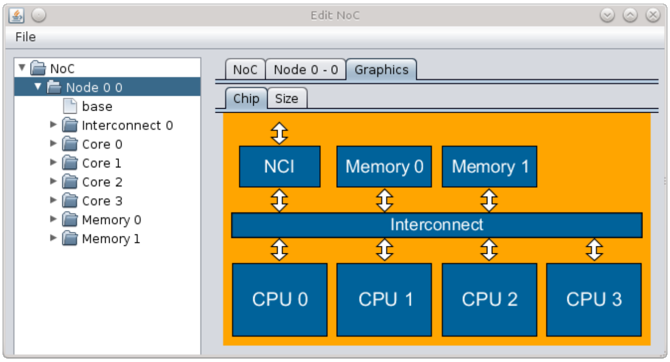


Abbildung 4.11: Bildschirmfoto der CoreVA-MPSoC-GUI

Wenn beispielsweise Parameter eines Compileraufrufes aus der einheitlichen Beschreibung abgeleitet werden, kann das Ergebnis im Nachhinein reproduziert werden. Die Beschreibung erfolgt in der Auszeichnungssprache XML[3]. Das Einlesen einer XML-Datei ist in einer Vielzahl von Programmiersprachen mit geringem Aufwand möglich. Gleichzeitig ist eine XML-Datei einfach von Menschen lesbar und editierbar. In Abbildung A.3 im Anhang ist ein Beispiel für eine XML-Beschreibung des CoreVA-MPSoCs dargestellt.

Die XML-Beschreibung ermöglicht es, für einen CPU-Kern die Anzahl und Ausprägung von VLIW-Slots anzugeben. Zudem kann die Größe der lokalen Scratchpad-Speicher für Instruktionen und Daten sowie die Anzahl der Einträge des FIFOs der CPU-Cluster-Schnittstelle festgelegt werden. Für einen CPU-Cluster ist es möglich, die Anzahl der CPU-Kerne sowie die Art und Topologie der Verbindungsstruktur zu definieren. Des Weiteren kann für gemeinsame L1- und L2-Speicher die Architektur (z. B. Anzahl an Bänken) und Größe angegeben werden. Außerdem ist eine Konfiguration der NoC-Cluster-Schnittstelle (NI) möglich. Für das On-Chip-Netzwerk kann beispielsweise die Topologie und die Anzahl der CPU-Cluster konfiguriert werden. Die XML-Beschreibung bietet zudem Unterstützung für heterogene MPSoCs. Als Beispiel hierfür kann ein CPU-Cluster genannt werden, dessen CPUs über unterschiedlich viele VLIW-Slots verfügen.

Für jede Komponente, die in der XML-Datei beschrieben ist, kann zudem der Flächenbedarf und eine Leistungsaufnahme angegeben werden. Dies erlaubt zusammen mit dem im Abschnitt 4.3 vorgestellten Hardware-Systemmodell eine automatisierte Bestimmung der Gesamtfläche von verschiedenen Konfigurationen des CoreVA-MPSoCs.

Um die Genauigkeit der Abschätzungen von Programmlaufzeiten im CoreVA-MPSoC-Compiler zu verbessern, kann die XML-Beschreibung Taktzyklen für die Ausführung von Kommunikationsprimitiven für die jeweilige Konfiguration enthalten (siehe Abschnitt 4.2.4). Ein Beispiel ist das Setzen eines Mutex im Speicher einer CPU im gleichen

[3] *Extensible Markup Language*

CPU-Cluster (siehe Abschnitt 5.3). Diese Taktzyklen können automatisiert mithilfe des CoreVA-MPSoC-Compilers ermittelt und in die XML-Beschreibung integriert werden.

Die XML-Beschreibung wird im Software-Entwurfsablauf insbesondere vom CoreVA-MPSoC-Compiler verwendet (siehe Abschnitt 3.2.3). Eine Integration in den Hardware-Entwurfsablauf (siehe Abschnitt 3.1) ist Bestandteil aktueller Arbeiten im Rahmen des CoreVA-MPSoC-Projektes.

Um die Erstellung und Verwaltung verschiedener CoreVA-MPSoC-Konfigurationen zu vereinfachen, ist im Rahmen dieser Arbeit eine **grafische Oberfläche** (*Graphical User Interface*, GUI) entstanden (siehe Abbildung 4.11). Die GUI emöglicht es, die verschiedenen Hierarchieebenen des MPSoCs grafisch als Blockdiagramm darzustellen. Eine Integration des in Abschnitt 4.3 vorgestellten Hardware-Systemmodells ermöglicht eine Abschätzung der Gesamtfläche einer CoreVA-MPSoC-Konfiguration. Die abgeschätzte Fläche der verschiedenen Komponenten des MPSoCs kann zudem grafisch dargestellt werden.

4.5 Zusammenfassung

In diesem Kapitel wurden Ressourcen und Bewertungsmaße für eingebettete MPSoCs eingeführt. Diese ermöglichen eine Bewertung von Hardware- und Software-Komponenten des CoreVA-MPSoCs, was insbesondere für die Entwurfsraumexploration in Kapitel 5 erforderlich ist. Des Weiteren wurde in diesem Kapitel ein abstraktes Modell für die Ausführungszeit von Anwendungen auf MPSoCs vorgestellt. Dieses Modell berücksichtigt Daten- und funktionale Parallelität und kann als obere Schranke für die Performanz von Anwendungen verwendet werden. Hierdurch ist es möglich, das Optimierungspotential sowohl von Anwendungen als auch der Hardware- und Softwarearchitektur des CoreVA-MPSoCs zu identifizieren. Zudem wurde ein abstraktes Modell für den Ressourcenbedarf des CPU-Clusters beschrieben und eine abstrakte Systembeschreibung des CoreVA-MPSoCs vorgestellt. Diese abstrakte Beschreibung vereinfacht die Untersuchung von dutzenden verschiedenen MPSoC-Konfigurationen im Rahmen der Entwurfsraumexploration in Kapitel 5.

5 Entwurfsraumexploration des CPU-Clusters im CoreVA-MPSoC

In diesem Kapitel wird eine Entwurfsraumexploration für den in Kapitel 2 vorgestellten CPU-Cluster des CoreVA-MPSoCs durchgeführt. Hierzu wird die in Kapitel 3 vorgestellte Entwurfsumgebung verwendet. Für die Analyse verschiedener Hardware- und Softwarekonfigurationen werden die Bewertungsmaße aus Abschnitt 4.1 angewendet. Als Zieltechnologie für die physikalische Implementierung des CPU-Clusters kommt eine 28-nm-FD-SOI-Technologie von STMicroelectronics zum Einsatz. Es stehen zehn Metallisierungsebenen für die Verdrahtung zur Verfügung. Es wird die schlechteste PVT[1]-Betriebsbedingung (*Worst Case Corner*, WCC) mit einer Versorgungsspannung von 1 V sowie einer Temperatur von 125 °C verwendet (siehe Abschnitt 3.1).

Die im Rahmen dieser Arbeit verwendeten Beispielanwendungen werden in Abschnitt 5.1 beschrieben. In Abschnitt 5.2 werden verschiedene Konfigurationen der CoreVA-CPU für eine Verwendung innerhalb des CoreVA-MPSoCs untersucht. Zudem werden sechs CPU-Makros vorgestellt, die für die physikalische Implementierung des CPU-Clusters verwendet werden.

Darauf basierend werden in Abschnitt 5.3 verschiedene Synchronisierungsverfahren und Speicherarchitekturen in Bezug auf den Aufwand (in CPU-Takten) verglichen. Zur Programmierung des CoreVA-MPSoCs können die drei Programmiersprachen C, OpenCL und StreamIt verwendet werden (siehe Abschnitt 3.2). Der CoreVA-MPSoC-Compiler erlaubt eine automatisierte Partitionierung und Optimierung einer StreamIt-Anwendung auf verschiedene Hardwarekonfigurationen. Eine C-Implementierung einer Anwendung kann mindestens die gleiche Performanz im Vergleich mit der vom CoreVA-MPSoC-Compiler erzeugten Partitionierung erreichen. Hierfür ist jedoch ein deutlich höherer Aufwand des Anwendungsentwicklers erforderlich und es ist schwer, alle Varianten einer Anwendung vergleichbar gut zu optimieren. Da der Programmierer die Kommunikation innerhalb einer StreamIt-Anwendung als gerichteten Datenflussgraphen beschreibt, kann bei einer Verwendung des CoreVA-MPSoC-Compilers sichergestellt werden, dass die Anwendung ohne Verklemmungen arbeitet. Aus diesen Gründen wird in einem ersten Schritt in Abschnitt 5.4 anhand einer C-basierten Implementierung der Anwendung Matrixmultiplikation beispielhaft der große Entwurfsraum bei der Partitionierung von Anwendungen auf MPSoCs dargestellt. Anschließend wird eine Entwurfsraumexploration von verschiedenen Hardwarekomponenten des CoreVA-MPSoCs

[1] *Process Voltage Temperature*

vorgestellt. Hierzu werden Synthesen in einer 28-nm-FD-SOI-Technologie durch- sowie verschiedene Streaming-Anwendungen auf dem CoreVA-MPSoC ausgeführt. In diesem Kapitel wird als Software-Bewertungsmaß der Durchsatz von Anwendungen verwendet (siehe Abschnitt 4.1). Die weiteren möglichen Bewertungsmaße Latenz und Energie sind Gegenstand von aktuellen Arbeiten im Rahmen des CoreVA-MPSoC-Projektes.

Unter Verwendung dieser Bewertungsmaße werden in Abschnitt 5.5 verschiedene Konfigurationen der Cluster-Verbindungsstruktur betrachtet. Hierbei werden verschiedene Bus-Standards und Topologien untersucht sowie die Anzahl der integrierten Registerstufen und CPUs variiert. Basierend auf diesen Untersuchungen wird in Abschnitt 5.6 eine Integration von gemeinsamem L1-Datenspeicher evaluiert. Ein Instruktionscache wird in Abschnitt 5.7 in den CPU-Cluster integriert und kann über die Cluster-Verbindungsstruktur auf gemeinsamen L2-Speicher zugreifen. Im Anschluss wird in Abschnitt 5.8 der Energiebedarf von Kommunikation und Synchronisierung im CPU-Cluster des CoreVA-MPSoCs analysiert. In Abschnitt 5.9 werden abschließend verschiedene OpenCL-Anwendungen auf dem CoreVA-MPSoC ausgeführt. Basierend auf den Ergebnissen dieses Kapitels werden in Kapitel 6 verschiedene FPGA- und ASIC-basierte Prototypen des CoreVA-MPSoCs präsentiert.

5.1 Beispielanwendungen

In diesem Abschnitt werden die im Rahmen dieser Arbeit verwendeten Beispielanwendungen vorgestellt. Die Anwendungen stammen aus typischen Einsatzgebieten von eingebetteten Systemen wie Signal- und Bildverarbeitung sowie Verschlüsselung. Da die CoreVA-CPU zurzeit über keine Fließkomma-Einheit verfügt (siehe Abschnitt 2.2), verwenden die Anwendungen ausschließlich Ganzzahl- bzw. Festkommaarithmetik. Von allen Anwendungen steht eine Implementierung in der Sprache StreamIt zur Verfügung. Zudem werden von der Matrixmultiplikation Implementierungen in den Sprachen C (siehe Abschnitt 5.4) sowie OpenCL (siehe Abschnitt 5.9) verwendet. Weitere OpenCL-Anwendungen werden in Abschnitt 5.9 beschrieben. Ein Teil der StreamIt-Anwendungen basiert auf der StreamIt-Benchmark-Bibliothek [66; 193–195]. Diese Anwendungen sind nicht speziell für eine Abbildung auf die Architektur des CoreVA-MPSoCs optimiert worden. In Anhang C sind verschiedene Eigenschaften der Anwendungen wie beispielsweise die Anzahl der StreamIt-Filter, die Größe des Programmabbilds sowie der Anteil von CPU-zu-CPU-Kommunikation an allen ausgeführten Instruktionen aufgeführt. Mithilfe dieser Analysen kann gezeigt werden, dass die betrachteten Anwendungen unterschiedliche Kommunikationsmuster sowie Anforderungen an die CoreVA-CPU und die Cluster-Verbindungsstruktur aufweisen.

DES

Der *Data Encryption Standard* (DES) ist ein symmetrisches Verschlüsselungsverfahren mit einer Schlüssellänge von 56 bit bei einer Blockgröße von 64 bit [175]. Durch eine wiederholte Anwendung des Verfahrens auf ein Datenwort kann die effektive Schlüssellänge und dadurch die Sicherheit erhöht werden (z. B. Triple-DES). Die verwendete StreamIt-Implementierung wird in [66, S. 72 f.] vorgestellt und besteht aus fünf Berechnungsschritten, die insgesamt viermal nacheinander ausgeführt werden. Da DES unabhängig von der Menge der zu verschlüsselnden Daten mit einer festen Blockgröße arbeitet, beträgt die Komplexität $\mathcal{O}(n)$.

Digitale Filter

Digitale Filter verändern ein digitales Signal abhängig von der Frequenz. Ein Beispiel ist die Veränderung der Amplitude des Signals in einem bestimmten Frequenzbereich.

FIR[2]-Filter sind in der digitalen Signalverarbeitung weit verbreitet. Aus einem diskreten Eingangssignal x und der Übertragungsfunktion (Filterkoeffizienten) h lässt sich die Ausgabe y des Filters bestimmen (siehe Gleichung 5.1). M entspricht der „Tiefe“ des Filters bzw. der Anzahl der Filterkoeffizienten.

$$y(n) = \sum_{k=0}^{M-1} h(k) \cdot x(n-k) \tag{5.1}$$

Die Anwendung **FIR-Filter** implementiert einen Tiefpassfilter und stammt aus der StreamIt-Benchmark-Bibliothek [193, S. 30 f.]. Es werden 64 Filterkoeffizienten verwendet. Zudem wird das Signal um den Faktor zwei heruntergetaktet (*Downsampling*).

Ein FIR-Filter mit einer Eins als Übertragungsfunktion berechnet den gleitenden Mittelwert (***Simple Moving Average***, **SMA**) eines Signals bzw. einer Zahlenfolge. Die Multiplikation von Eingangssignal und Übertragungsfunktion (siehe Gleichung 5.1) entfällt. Ein weiteres Einsatzgebiet von SMA ist die Finanzmathematik [139]. Die verwendete Implementierung bildet den Mittelwert über jeweils zehn Elemente.

Die Anwendung **Filterbank** zerlegt ein digitales Signal in acht verschiedene Frequenzbänder, filtert diese und kombiniert die einzelnen Bänder anschließend wieder [66, 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. Alle drei betrachteten digitalen Filter besitzen eine Komplexität von $\mathcal{O}(n)$, da die Berechnung von einem Ausgabeelement nur von der Summe über die Filterkoeffizienten (siehe Gleichung 5.1) und nicht von der Gesamtanzahl der Eingabeelementen abhängt.

FFT

Die schnelle Fourier-Transformation (*Fast Fourier Transformation*, FFT) wird in einer Vielzahl von Anwendungsfeldern wie Signal-, Audio- und Videoverarbeitung verwendet und kann durch die parallele Ausführung auf mehreren CPUs beschleunigt werden [29].

[2] *Finite Impulse Response*

Im Rahmen dieser Arbeit wird eine Radix-2-FFT mit 64 Abtastpunkten aus der StreamIt-Benchmark-Bibliothek verwendet [66, S. 74 f.]. Die Komplexität einer Radix-2-FFT ist $\mathcal{O}(n \log_2(n))$ [30, S. 9].

Matrixmultiplikation

Matrixmultiplikation, auch als Matrizenmultiplikation bezeichnet, wird in vielen Anwendungsgebieten eingesetzt [35, S. 551 ff.]. Ein Beispiel ist die Bildverarbeitung, 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 5.2 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 $m \times p$.

$$C_{ij} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j} \quad (1 \leq i \leq m, 1 \leq j \leq p) \tag{5.2}$$

C_{ij} ist demnach das Skalarprodukt der Zeile i der ersten Matrix und der Spalte j der zweiten Matrix.

In Abbildung 5.1 ist der Programmcode einer Implementierung der Matrixmultiplikation dargestellt. Die äußere Schleife durchläuft alle Zeilen von A, während die zweite Schleife alle Spalten von B durchläuft. Die innere Schleife berechnet ein Element der Ergebnismatrix C. Die Komplexität ist $\mathcal{O}(m \cdot n \cdot p)$. Für quadratische Matrizen der Größe $n \times n$ ist die Komplexität $\mathcal{O}(n^3)$. Die beiden äußeren Schleifen der Matrixmultiplikation können ohne Änderungen am Algorithmus parallel auf verschiedenen CPUs ausgeführt werden. Zur Berechnung eines Ergebniswertes C_{ij} benötigt die CPU Zeile i von Matrix A und Zeile j von Matrix B. Die verwendete StreamIt-Implementierung [194] multipliziert zwei Matrizen der Größe 8×8.

```
int i, j, k;
for(i = 0; i < m; i++)
    for(j = 0; j < p; j++)
    {
        C[i][j] = 0
        for(k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
```

Abbildung 5.1: Implementierung einer Matrixmultiplikation [35, S. 551]

Sortieralgorithmen

Der Sortieralgorithmus **BitonicSort** [142] ist besonders gut für eine parallele Implementierung geeignet. Bei n zu sortierenden Elementen werden $\mathcal{O}(n \log_2(n)^2)$ Vergleichs-operationen benötigt [119]. Die Reihenfolge der Vergleiche ist durch den Algorithmus festgelegt und nicht von den zu sortierenden Daten abhängig. Implementierungen für Multiprozessorsysteme teilen die zu sortierenden Daten auf n CPUs auf. Hiernach sortieren diese CPUs den ihnen zugewiesenen Anteil der Daten. Abschließend werden die Daten zusammengefasst (*Merge*). Eine detaillierte Beschreibung von BitonicSort ist in [119] zu finden. Es wird eine Liste mit 32 Elementen sortiert.

BubbleSort sortiert eine Liste, indem n-mal jedes Element mit dem nächsten Element („rechter Nachbar") verglichen wird [15]. Ist der Nachbar größer als das betrachtete Element, werden beide getauscht. BubbleSort besitzt eine Laufzeit von $\mathcal{O}(n^2)$. Eine Parallelisierung von BubbleSort ist einfach möglich, indem jeder Durchlauf der Liste auf einer anderen CPU ausgeführt wird (Pipeline-Parallelität, siehe Abschnitt 4.2). Die verwendete Implementierung sortiert eine Liste der Länge 32.

RadixSort sortiert eine Liste nacheinander für jede Stelle eines Zahlensystems [38, S. 197 ff.]. In digitalen Rechnersystemen wird typischerweise das Binärsystem verwendet. Die Liste wird aufsteigend nach jedem Bit sortiert (beginnend mit Bit 0), ohne hierbei die Reihenfolge der Zahlen zu verändern, wenn der Wert des jeweiligen Bits gleich ist. Somit ist RadixSort ein stabiles Sortierverfahren. Der Algorithmus besitzt bei n zu sortierenden Elementen und einer maximalen Größe der Zahlen w eine Komplexität von $\mathcal{O}(n \log_2(w))$. Die im Rahmen dieser Arbeit verwendete Implementierung sortiert Zahlen bis zu einer Größe von 2048, was in 11 Sortierstufen resultiert. Die zu sortierende Liste hat eine Länge von 16 Elementen.

5.2 Analyse der CoreVA-CPU für die Verwendung in einem Multiprozessorsystem

In diesem Abschnitt werden verschiedene Konfigurationen der CoreVA-CPU auf eine mögliche Verwendung im CoreVA-MPSoC hin untersucht. Eine Entwurfsraumexploration der CPU-Pipeline wurde im Rahmen der Dissertation [99] durchgeführt und ist unter anderem Gegenstand der Veröffentlichungen [215; 217; 220; 224]. In Bezug auf diese Arbeit ist insbesondere die Architektur der lokalen Daten- und Instruktionsspeicher sowie die Schnittstelle zum CPU-Cluster von Interesse, da diese Komponenten einen Einfluss auf die Performanz von CPU-zu-CPU-Kommunikation im Cluster besitzen. Um die Reproduzierbarkeit des Hardwareentwurfes vom CoreVA-MPSoC zu steigern und Synthesen des CPU-Clusters zu beschleunigen, werden in diesem Abschnitt Makros der CoreVA-CPU („CPU-Makros", siehe Abschnitt 3.1) vorgestellt und im weiteren Verlauf der Arbeit verwendet.

5.2.1 Untersuchung der betrachteten VLIW-Konfigurationen

Die CoreVA-CPU zeichnet sich durch eine hohe Konfigurierbarkeit bezüglich der VLIW-Slots und Ausführungseinheiten aus (siehe Abschnitt 2.2). Daher werden in diesem Abschnitt verschiedene Konfigurationen der CoreVA-CPU im Hinblick auf die Ausführungszeit von Streaming-Anwendungen und den Ressourcenbedarf bei der Implementierung in einer 28-nm-FD-SOI-Standardzellenbibliothek untersucht.

Basierend auf den Ergebnissen der Veröffentlichungen [99] und [217] werden drei Konfigurationen der CPU-Pipeline ausgewählt. Die 1-Slot-Konfiguration besitzt einen VLIW-Slot und dementsprechend eine ALU (siehe Tabelle 5.1). Die 2-Slot- und 4-Slot-Konfigurationen verfügen über zwei bzw. vier ALUs. Konfigurationen mit einem und zwei VLIW-Slots besitzen jeweils eine MAC-Einheit, während die Konfiguration mit vier VLIW-Slots über zwei MAC-Einheiten verfügt. Alle Konfigurationen besitzen jeweils eine LD/ST- und eine DIV-Einheit. SIMD-Einheiten werden nicht verwendet, da der LLVM-basierte Compiler für die CoreVA-CPU bisher nur rudimentäre Unterstützung hierfür bietet (siehe Abschnitt 3.2.2). Die CPUs verfügen jeweils über 16 kB Instruktions- und Datenspeicher. Die Schnittstelle des Instruktionsspeichers ist bei den 1-Slot- und 2-Slot-Konfigurationen 64 bit breit. Die 4-Slot-CPU verfügt über eine 128 bit breite Schnittstelle. Das FIFO der CPU-Cluster-Schnittstelle besitzt vier Einträge (siehe Abschnitt 5.2.2).

Im Unterschied zu den in [99] und [132] vorgestellten Chipprototypen der CoreVA-CPU wird keine dedizierte Schnittstelle zur Standardausgabe (UART) in die einzelnen CPU-Kerne des CoreVA-MPSoCs integriert. Stattdessen wird die in Abschnitt 3.2.1 beschriebene CoreVA-MPSoC-Standardausgabe verwendet. Im Vergleich zu einer dedizierten UART-Schnittstelle wird hierdurch die Chipfläche der einzelnen CPUs reduziert. Gleichzeitig werden keine dedizierten I/O-Anschlüsse für die UART-Schnittstellen benötigt.

Für die Synthesen der CoreVA-CPU wird die in Abschnitt 3.1 vorgestellte Hardware-Entwurfsumgebung sowie eine 28-nm-FD-SOI-Standardzellenbibliothek verwendet. Um die Nutzung der CPU-Makros auf CPU-Cluster-Ebene zu ermöglichen bzw. zu

Tabelle 5.1: Eigenschaften von verschiedenen Konfigurationen der CoreVA-CPU

VLIW-Slots	MAC	DIV	LD/ST	Größe Datenspeicher [kB]	Instr. Speicher [kB]	Datenbreite Instr. Speicher [bit]	max. Taktfrequenz f_{max} [MHz]	Fläche@f_{max} [mm^2]	Fläche@800MHz [mm^2]
1	1	1	1	16	16	64	900	0,116	0,113
2	1	1	1	16	16	64	900	0,139	0,136
4	2	1	1	16	16	128	909	0,209	0,205

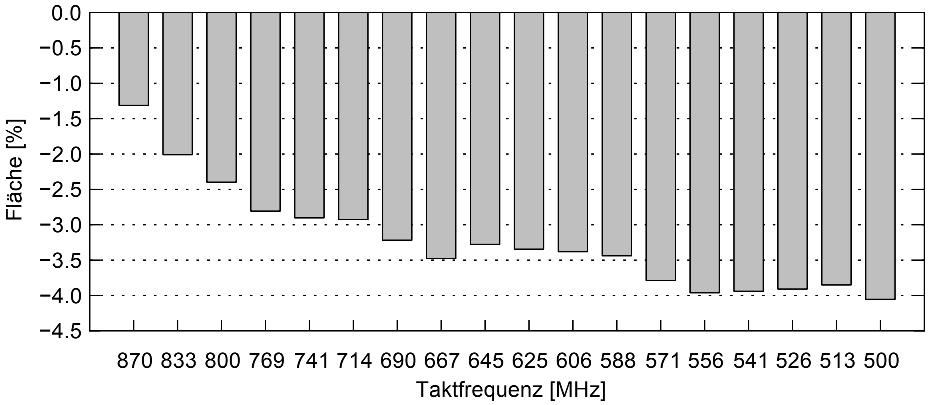


Abbildung 5.2: Prozentuale Verringerung des Flächenbedarfs einer 2-Slot-CoreVA-CPU in Abhängigkeit der Taktfrequenz gegenüber einer Synthese mit 900 MHz

erleichtern, ist es notwendig, Vorgaben für die Verzögerung der Ein- und Ausgabepfade des Makros (*I/O-Constraints*) festzulegen. Hierzu wird ein CPU-Cluster mit zwei CPUs „flach“, also ohne die Verwendung von CPU-Makros, synthetisiert. Anschließend werden die Verzögerungszeiten der einzelnen Ein- und Ausgabepfade zwischen den CPUs und der Cluster-Verbindungsstruktur bestimmt. Zudem wird die Verzögerungszeit des Taktbaumes innerhalb der CoreVA-CPUs ermittelt. Mithilfe dieser Verzögerungszeiten können die Vorgaben für Ein- und Ausgabesignale des CPU-Makros berechnet werden.

In Tabelle 5.1 sind die maximalen Taktfrequenzen der betrachteten CPU-Konfigurationen sowie deren entsprechender Flächenbedarf dargestellt. Zur Bestimmung der maximalen Taktfrequenz wird das in [219] beschriebene *Get-Best-Timing*-Skript benutzt. Die eingesetzte Synthese-Software verwendet nichtdeterministische Optimierungsverfahren. Aus diesem Grund variieren die maximalen Taktfrequenzen der betrachteten VLIW-Konfigurationen im Rahmen von Synthese-Ungenauigkeiten leicht und liegen bei 909 MHz bzw. 900 MHz. Ausgehend vom Instruktionsspeicher durchläuft der kritische Pfad bei den meisten Konfigurationen die CPU-Pipeline und endet am Datenspeicher. Weitere kritische Pfade gehen vom Datenspeicher bzw. Instruktionsspeicher zur Slave-Schnittstelle des Makros. Die 1-Slot-Konfiguration besitzt einen Flächenbedarf von 0,116 mm^2, wobei die Speicher-Makros für die Daten- und Instruktionsspeicher jeweils 0,030 mm^2 belegen. Im Vergleich zur 1-Slot-CPU zeigt die 2-Slot-Konfiguration einen um 20 % und die 4-Slot-Konfiguration einen um 80 % höheren Flächenbedarf.

In einem nächsten Schritt wird der Zusammenhang von Taktfrequenz und Flächenbedarf durch verschiedene Synthesen der CoreVA-CPU analysiert. In Abbildung 5.2 ist die prozentuale Verringerung des Flächenbedarfs einer 2-Slot-CoreVA-CPU in Abhängigkeit

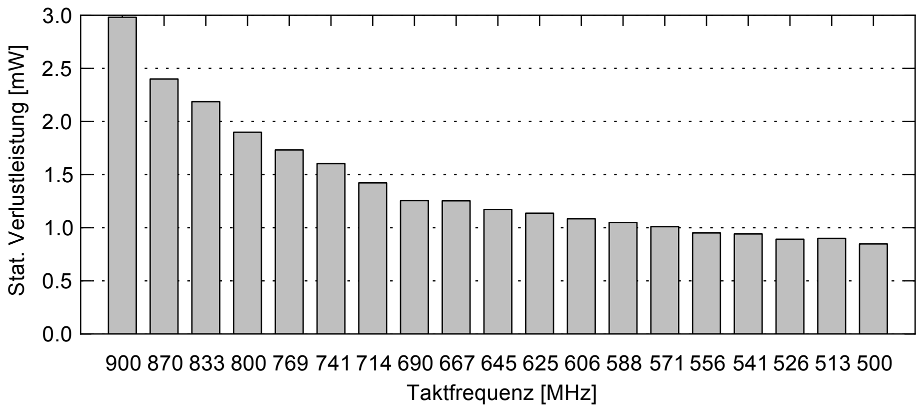


Abbildung 5.3: Statische Verlustleistung (*Leakage*) einer 2-Slot-CoreVA-CPU in Abhängigkeit von der Taktfrequenz

der Taktfrequenz gegenüber einer Synthese mit maximaler Taktfrequenz (900 MHz) dargestellt. Eine Zielfrequenz von 800 MHz resultiert in einem um 2,4 % verringerten Flächenbedarf. Bei 667 MHz und 500 MHz beträgt die Reduktion der Fläche 3,5 % bzw. 4,1 %. Die Schwankungen bei den Ergebnissen für Konfigurationen zwischen 667 MHz und 500 MHz treten ebenfalls durch verschiedene Optimierungen der Synthese-Software auf. Die Reduzierung des Flächenbedarfs bei verminderter Taktfrequenz ist darauf zurückzuführen, dass Transistoren mit geringerer Treiberstärke und geringerem Flächenbedarf von der Synthese-Software verwendet werden können.

Die Leistungsaufnahme von mikroelektronischen Schaltungen setzt sich aus der dynamischen und der statischen Verlustleitung zusammen (siehe Abschnitt 4.1). In [132, S. 139 ff.] wurde gezeigt, dass die dynamische Verlustleistung für die betrachteten Anwendungsszenarien der CoreVA-CPU gut durch eine mittlere Schaltwahrscheinlichkeit von 10 % bis 20 % approximiert werden kann. Für die in diesem Abschnitt durchgeführten Analysen wird daher eine Schaltwahrscheinlichkeit von 10 % verwendet.

Die Leistungsaufnahme einer 2-Slot-CoreVA-CPU bei einer Synthese-Zielfrequenz von 900 MHz beträgt 20,20 mW und verringert sich für eine Synthesefrequenz von 800 MHz und 500 MHz auf 17,24 mW (-14,71 %) bzw. 11,04 mW (-45,36 %). Der Energiebedarf pro ausgeführtem CPU-Takt stellt jedoch ein aussagekräftigeres Bewertungsmaß dar und hängt von der ausgeführten Instruktion sowie den verarbeiteten Daten ab. In diesem Abschnitt wird der durchschnittliche Energiebedarf pro Takt betrachtet, weiterführende Untersuchungen sind unter anderem in [99; 103] zu finden. Die Konfiguration mit 900 MHz, 800 MHz und 500 MHz besitzen einen Energiebedarf pro Takt von 22,23 pJ, 21,55 pJ sowie 22,08 pJ. Der geringste Energiebedarf pro Takt ist bei einer Synthesefrequenz von 645 MHz zu beobachten (20,34 pJ, -8,5 % im Vergleich zu 900 MHz). Eine höhere Taktfrequenz steigert im gleichen Maße die Rechenleistung der CPU. Der Energiebedarf pro CPU-Takt steigt hierbei nicht signifikant. Die Leistungsaufnahme einer CPU lässt sich senken, indem die CPU mit verminderter Taktfrequenz betrieben wird, wenn eine Anwendung die maximale Rechenleistung nicht benötigt. Es ist zudem möglich, in diesem Fall die Versorgungsspannung abzusenken, um die Leistungsaufnahme weiter zu verringern [133].

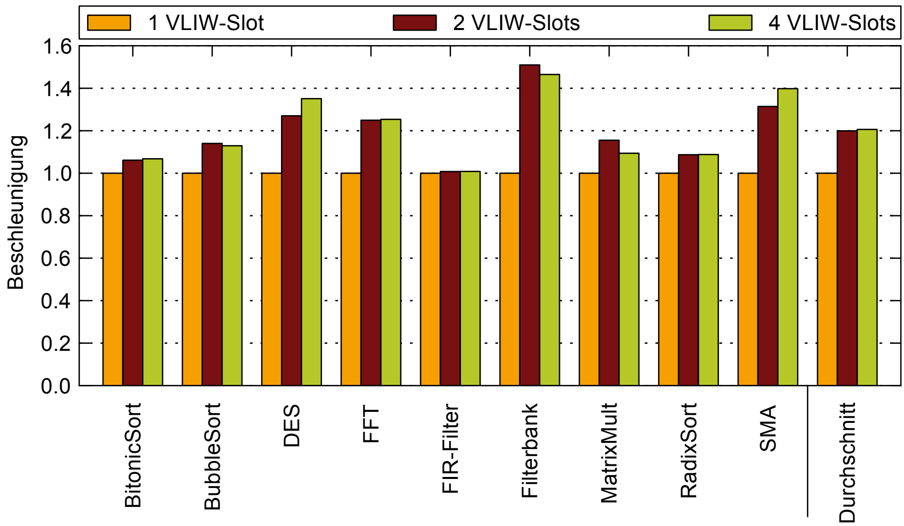
Die statische Verlustleistung lässt sich vor allem auf Subschwellströme von sperrenden Transistoren zurückführen (*Leakage*) und ist unabhängig von der Taktfrequenz bzw. Schaltaktivität der CPU. In Abbildung 5.3 ist die statische Verlustleistung in Abhängigkeit von der Synthesefrequenz dargestellt. Im Vergleich zu der maximalen Taktfrequenz von 900 MHz sinkt die statische Verlustleistung bei 800 MHz um 36,3 % auf 1,90 mW. Bei einer Frequenz von 500 MHz sinkt die statische Verlustleistung auf 0,85 mW (-71,6 %). Diese starke Abnahme ist vor allem auf die Verwendung von Transistoren mit unterschiedlicher Schwellspannung zurückzuführen. LVT[3]-Transistoren besitzen eine Schwellspannung von ca. 400 mV und weisen eine geringere Verzögerungszeit auf. Die Leckströme erhöhen sich im Vergleich zu RVT[4]-Transistoren jedoch deutlich, die eine Schwellspannung von ca. 480 mV besitzen. Bei einer Synthese mit geringer Frequenz kann die Synthese-Software vermehrt RVT-Transistoren verwenden.

Wie in diesem Abschnitt gezeigt werden konnte, ist es bei der Vorgabe der Synthesefrequenz einer mikroelektronischen Schaltung notwendig, zwischen Rechenleistung und Energieeffizienz abzuwägen. Für die CoreVA-CPU stellt eine Taktfrequenz von 800 MHz einen guten Kompromiss zwischen Flächenbedarf, Rechenleistung und statischer sowie dynamischer Verlustleistung dar. Aus diesem Grund werden im weiteren Verlauf dieser Arbeit die Synthesen für die Analyse des Flächenbedarfs von verschiedenen Konfigurationen des CPU-Clusters mit einer Zielfrequenz von 800 MHz durchgeführt.

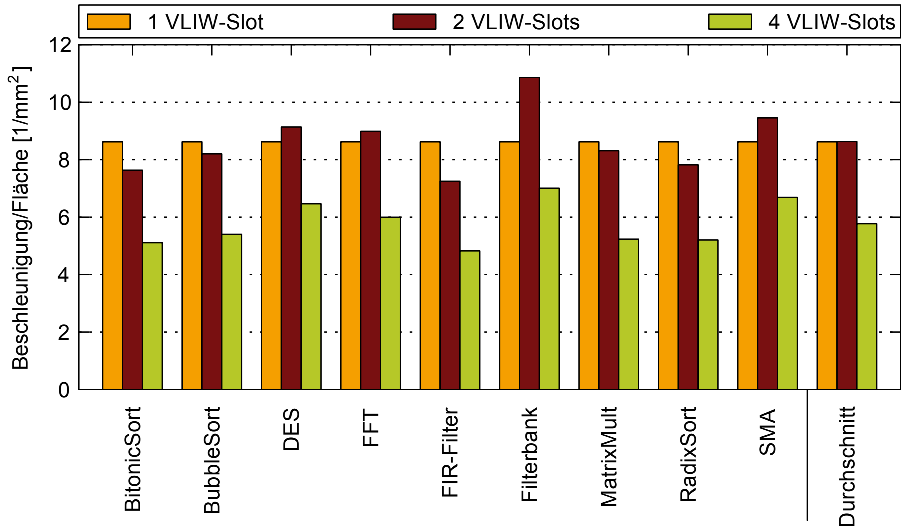
Im Folgenden wird die Performanz von verschiedenen Konfigurationen der CoreVA-CPU betrachtet. Die Beschleunigung von Streaming-Anwendungen bei der Verwendung von VLIW-Konfigurationen mit einem, zwei und vier Slots im Vergleich zu der Ausführung auf einer CPU mit einem VLIW-Slot ist in Abbildung 5.4a dargestellt. Die Anwendungen liegen in der Sprache StreamIt vor (siehe Abschnitt 5.1) und werden vom CoreVA-MPSoC-Compiler auf eine CoreVA-CPU abgebildet (siehe Abschnitt 3.2.3). Im Gegensatz zur Abbildung auf ein Multiprozessorsystem ist hierbei keine Partitionierung notwendig. Der CoreVA-MPSoC-Compiler erzeugt C-Code, der anschließend mit dem LLVM-Compiler für die verschiedenen VLIW-Konfigurationen übersetzt wird.

Die Beschleunigung gegenüber einer 1-Slot-CPU ist stark von der Anwendung abhängig und reicht für die Konfiguration mit zwei VLIW-Slots von lediglich Faktor 1,01 (1 %) bei der Anwendung FIR-Filter bis zu 1,51 (51 %) bei der Anwendung Filterbank. Die durchschnittliche Beschleunigung der 2-Slot-Konfiguration beträgt 1,20 bzw. 20 %. Die betrachteten Anwendungen können nicht signifikant von vier VLIW-Slots profitieren. Im Fall von BubbleSort, Filterbank und MatrixMult ist die Ausführungsgeschwindigkeit

[3] *Low Vt, Low Threshold Voltage*
[4] *Regular Vt, Regular Threshold Voltage*



(a) Beschleunigung



(b) Beschleunigungs-Flächen-Verhältnis

Abbildung 5.4: Performanz von Streaming-Anwendungen und CPU-Konfigurationen mit einem, zwei und vier VLIW-Slots gegenüber der Ausführung auf einer CPU mit einem VLIW-Slot im Vergleich zur 2-Slot-CPU sogar etwas geringer. Dies ist auf unpassende Abschätzungen im LLVM-Compiler zurückzuführen. Gegenüber einer 1-Slot-CPU beträgt die durchschnittliche Beschleunigung der 4-Slot-Konfiguration 21 %.

Der geringe Performanzgewinn sowohl der 2-Slot- als auch insbesondere der 4-Slot-Konfiguration gegenüber der 1-Slot-CPU ist auf verschiedene Faktoren zurückzuführen. Sowohl die C-Codeerzeugung des CoreVA-MPSoC-Compilers als auch der CoreVA-spezifische Teil des LLVM-Compilers bieten Möglichkeiten zur Optimierung, die sich gegenseitig beeinflussen können. Der CoreVA-MPSoC-Compiler verwendet beispielsweise in einigen Fällen globale Variablen, was den LLVM-Compiler bei VLIW-Parallelisierung behindert. Zudem ist es auch möglich, die Beschreibung der StreamIt-Anwendungen zu verbessern. Ein Beispiel hierfür ist das manuelle Abrollen von Schleifen. Solche händischen Eingriffe in den Quellcode der Anwendungen wurden im Rahmen dieser Arbeit jedoch nicht durchgeführt.

In Abbildung 5.4b ist das Verhältnis aus der Beschleunigung von Streaming-Anwendungen und dem Flächenbedarf verschiedener Konfigurationen der CoreVA-CPU dargestellt. Diese Metrik kann als Bewertungsmaß für die Flächeneffizienz der verschiedenen CPU-Konfigurationen verwendet werden.

Da die 1-Slot-CPU als Referenz für die Beschleunigung verwendet wird, ist das Beschleunigungs-Flächen-Verhältnis aller 1-Slot-Konfigurationen der Kehrwert des Flächenbedarfs (0,116 mm^2), also 8,62 mm^{-2}. Eine CPU mit zwei VLIW-Slots besitzt einen um 19,8 % höheren Flächenbedarf im Vergleich zu einer 1-Slot-CPU. Damit eine Anwendung bei Ausführung auf einer 2-Slot-CPU eine bessere Flächeneffizienz besitzt, muss die Beschleunigung durch den zweiten VLIW-Slot folglich größer als 19,8 % sein. Dies ist bei den Anwendungen DES, FFT, Filterbank und SMA der Fall. Die Anwendung Filterbank besitzt mit 10,86 mm^{-2} das beste Beschleunigungs-Flächen-Verhältnis der betrachteten Anwendungen. FIR-Filter weist mit 7,25 mm^{-2} das schlechteste Beschleunigungs-Flächen-Verhältnis auf. Im Durchschnitt aller neun Anwendungen beträgt das Verhältnis 8,63 mm^{-2} und ist somit 0,01 mm^{-2} besser als die CoreVA-CPU mit einem VLIW-Slot.

Eine 4-Slot-CPU ist 80 % bzw. 0,093 mm^2 größer als die 1-Slot-CPU, die Beschleunigung durch vier VLIW-Slots beträgt jedoch maximal 46 %. Hieraus resultiert ein sehr schlechtes durchschnittliches Beschleunigungs-Flächen-Verhältnis der 4-Slot-CPU von 5,77 mm^{-2}. Die 4-Slot-CPU ist also 2,85 mm^{-2} (33,1 %) schlechter als die CPU-Konfiguration mit einem VLIW-Slot.

Zusammenfassend kann festgestellt werden, dass die CPU-Konfigurationen mit einem und zwei VLIW-Slots für die betrachteten Anwendungen im Durchschnitt ein annähernd gleiches Beschleunigungs-Flächen-Verhältnis aufweisen, die 4-Slot-CPU jedoch 33,1 % schlechter ist. Eine CPU mit vier VLIW-Slots ist nur durch den Einsatz speziell hierauf abgestimmter und optimierter Anwendungen sinnvoll einzusetzen. In Abschnitt 5.2.4 werden CPU-Cluster mit verschiedenen VLIW-Konfigurationen und gleichem Flächenbedarf betrachtet. Zudem wird der Einfluss verschiedener VLIW-Konfigurationen auf die Performanz von CPU-zu-CPU-Kommunikation im Cluster untersucht. Daher wird erst in

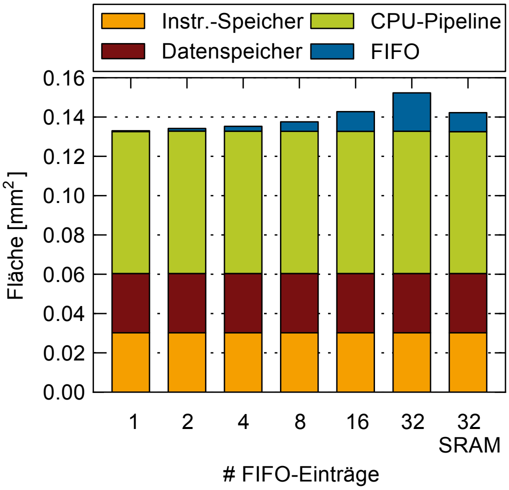
Abschnitt 5.2.4 festgelegt, welche CPU-Konfiguration am besten für eine Verwendung im CPU-Cluster geeignet ist.

5.2.2 CPU-Cluster-Schnittstelle

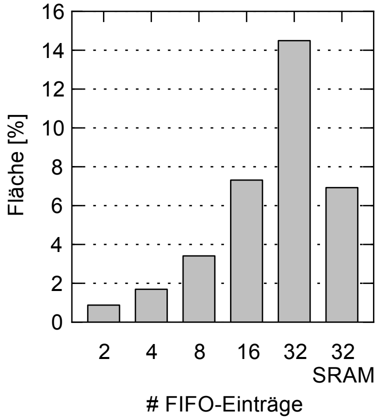
Die CPU-Cluster-Schnittstelle erlaubt der CoreVA-CPU schreibenden und lesenden Zugriff auf die Verbindungsstruktur des CPU-Clusters bzw. auf die L1- und L2-Speicher des Clusters (siehe Abschnitt 2.2, Abbildung 2.9) sowie [238, S. 9 ff.]. Um Schreibzugriffe der CPU auf den Bus zu entkoppeln, wird ein FIFO verwendet. In diesem Abschnitt werden die Auswirkungen verschiedener FIFO-Tiefen auf die Ausführungszeit von Streaming-Anwendungen sowie den Flächenbedarf der CoreVA-CPU betrachtet. Zudem werden verschiedene Implementierungen von FIFO-Warteschlangen (Register und SRAM) betrachtet.

Es ist möglich, Nutzdaten im FIFO zu speichern und zeitgleich bereits gespeicherte Daten auszulesen. Zwei Signale geben an, ob das FIFO vollständig gefüllt (*full*) bzw. vollständig geleert (*empty*) ist. Die minimale Anzahl an Einträgen beträgt zwei. Eine weitere Implementierung ohne FIFO-Funktionalität verfügt über eine Registerstufe und wird im Folgenden mit einer FIFO-Tiefe von eins bezeichnet. Die Datenbreite des FIFOs beträgt 67 bit. Hiervon werden 32 bit für die Schreibdaten, 30 bit für die Adresse sowie 4 bit für die Schreibmaske verwendet. Ein Bit gibt an, ob ein Schreib- oder ein Lesezugriff vorliegt. Die beiden in diesem Abschnitt untersuchten FIFO-Implementierungen sind Bestandteil der DesignWare-Bibliothek von Synopsys [188]. Die Implementierung DW_fifo_s1_sf verwendet Register für die Zwischenspeicherung der Nutzdaten [49]. DW_fifoctl_s1_sf stellt lediglich die Steuerlogik für das FIFO bereit, für die Speicherung der Nutzdaten wird ein externer SRAM oder eine Registerbank benötigt [48]. Diese müssen über zwei getrennte Ein-/Ausgabeschnittstellen verfügen (*Dual-Port-SRAM*). Der Dual-Port-SRAM der verwendeten 28-nm-FD-SOI-Bibliothek besitzt eine Mindestanzahl an Einträgen von 64 und ist daher ungeeignet für eine Verwendung in der CPU-Cluster-Schnittstelle. Zudem steht eine SRAM-basierte Registerbank als Makro zur Verfügung. Die minimale Anzahl an Einträgen beträgt hier 16, im Rahmen dieser Arbeit wird eine Registerbank mit 32 Einträgen evaluiert. Beide FIFO-Implementierungen sind synchron aufgebaut. Die Registerimplementierung verwendet einen gemeinsamen Takt für Ein- und Ausgabe. Die SRAM-Implementierung ermöglicht verschiedene Takte für Lesen und Schreiben, in der CPU-Cluster-Schnittstelle wird jedoch ein gemeinsamer Takt verwendet.

In Abbildung 5.5a ist der Flächenbedarf einer CoreVA-CPU mit zwei VLIW-Slots und verschiedenen FIFO-Tiefen innerhalb der CPU-Cluster-Schnittstelle dargestellt. Die Taktfrequenz beträgt 800 MHz. Der Flächenbedarf der CPU ist aufgeteilt in CPU-Cluster-Schnittstelle („FIFO“), Instruktions- und Datenspeicher (je 0,030 mm^2) sowie die weiteren Komponenten wie CPU-Pipeline und Kontrolllogik (0,072 mm^2). Der Flächenbedarf der CPU-Cluster-Schnittstelle beträgt bei der Verwendung von Registern 408 $\mu\mathrm{m}^2$ (1 FIFO-Eintrag) bis 19 531 $\mu\mathrm{m}^2$ (32 Einträge).



(a) Absolute Fläche



(b) Prozentuale Veränderung gegenüber einem FIFO mit einem Eintrag

Abbildung 5.5: Flächenbedarf einer 2-Slot-CoreVA-CPU und verschiedener Konfigurationen der CPU-Cluster-Schnittstelle

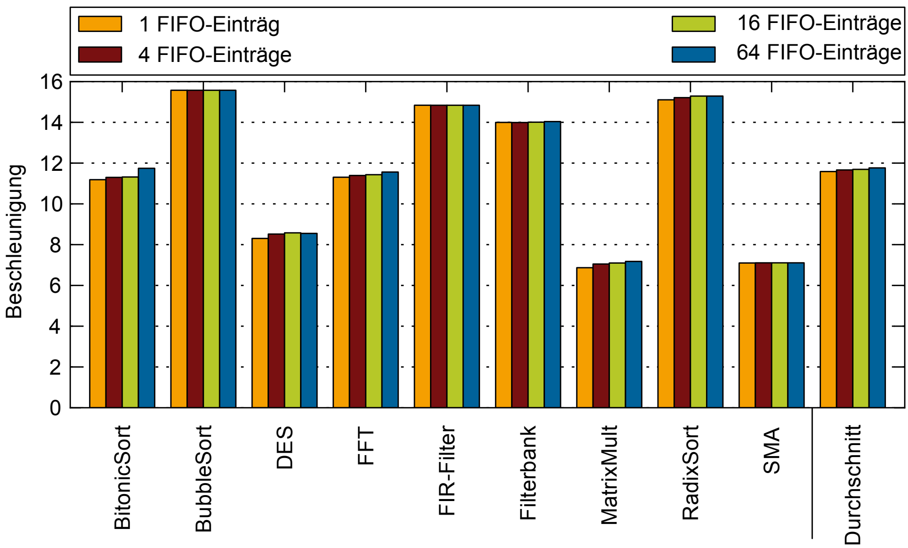


Abbildung 5.6: Beschleunigung von Streaming-Anwendungen mit 16 CPUs und einer Tiefe des CPU-Cluster-FIFOs von einem bis 64 Einträgen

Bezogen auf die Gesamtfläche der CPU bedeutet dies eine Steigerung um 1,7 % für die Konfiguration mit vier Einträgen oder um 14,5 % mit 32 Einträgen (siehe Abbildung 5.5b). Eine Verwendung der SRAM-basierten Registerbank mit 32 Einträgen resultiert in einem Flächenbedarf von 9680 μm^2, was sogar geringfügig kleiner als das registerbasierte FIFO mit 16 Einträgen (10 044 μm^2) ist. Keine der betrachteten Konfigurationen der CPU-Cluster-Schnittstelle beeinflusst die maximale Taktfrequenz der CPU. Zur Bestimmung des Flächenbedarfs der CPU-Cluster-Schnittstelle ist es notwendig, bei der Synthese die CPU-Cluster-Schnittstelle als separate Komponente zu erhalten. Dies kann den Flächenbedarf geringfügig erhöhen, da die Synthese-Software nicht über Komponentengrenzen hinweg optimieren kann.

Die Beschleunigung der Ausführungszeit von verschiedenen Streaming-Anwendungen auf einem CPU-Cluster mit 16 CPUs gegenüber der Ausführung auf einer CPU ist in Abbildung 5.6 dargestellt. Die Anzahl an Einträgen im FIFO der CPU-Cluster-Schnittstelle wird von eins bis 64 variiert und es wird ein geteilter AXI-Bus ohne Registerstufen als Verbindungsstruktur verwendet.

Im Durchschnitt aller neun betrachteten Anwendungen ist gegenüber der Verwendung eines FIFOs mit einem Eintrag eine Beschleunigung von 0,7 % (4 Einträge), 0,9 % (16 Einträge) bzw. 1,5 % (64 Einträge) zu beobachten. Die Konfiguration des FIFOs hat keinen Einfluss auf die Performanz der Anwendungen BubbleSort, FIR-Filter und SMA. Dies ist darauf zurückzuführen, dass bei diesen Anwendungen mehrere CPUs große Blöcke an Daten in einem kurzen Zeitraum übertragen. Die FIFOs der einzelnen CPUs sind schnell vollständig gefüllt, daher müssen CPUs sehr oft angehalten werden.

BitonicSort weist bei Verwendung eines FIFOs mit 64 Einträgen mit 5,0 % die höchste Beschleunigung auf. Ein FIFO mit 16 Einträge zeigt hier nur eine Beschleunigung um 1,2 %. Diese deutliche Verbesserung bei 64 Einträgen ist darauf zurückzuführen, dass ein Datenblock der Anwendung nun nahezu vollständig im FIFO gespeichert werden kann und die CPUs daher deutlich seltener angehalten werden müssen. Die Anwendung MatrixMult zeigt hingegen eine gleichmäßige Beschleunigung von 2,6 % (4 Einträge), 3,3 % (16 Einträge) bzw. 4,5 % (64 Einträge).

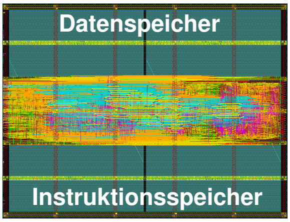
Bei den in Abbildung 5.6 dargestellten Untersuchungen sind keine Registerstufen in die Verbindungsstruktur des CPU-Clusters integriert. Registerstufen speichern ebenfalls Anfragen der CPUs zwischen (siehe Abschnitt 5.5.2). Daher ist bei der Verwendung von Registerstufen der Einfluss der Anzahl an FIFO-Einträgen auf die Performanz von Streaming-Anwendungen deutlich geringer. Die Beschleunigung gegenüber der Verwendung von nur einem FIFO-Eintrag beträgt mit je einer Master- und Slave-Registerstufe nur 0,1 % (4 FIFO-Einträge), 0,4 % (16 Einträge) bzw. 0,7 % (64 Einträge). Wie in Abschnitt 5.5.2 aufgezeigt wird, ist hierfür insbesondere die Slave-Registerstufe verantwortlich. Diese speichert Anfragen zwischen, wenn die Slave-Schnittstelle einer CPU gerade blockiert ist. Wird eine (partielle) Crossbar als Cluster-Verbindungsstruktur verwendet, sinkt der Einfluss des FIFOs auf die Performanz des CPU-Clusters weiter.

Die Anzahl der FIFO-Einträge der CPU-Cluster-Schnittstelle hat einen signifikanten Einfluss auf die Chipfläche der CoreVA-CPU. Die Performanz des CPU-Clusters bei der

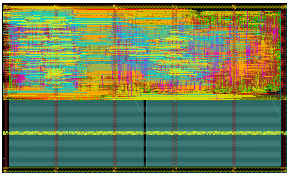
Ausführung von Streaming-Anwendungen steigt jedoch nicht im gleichen Umfang. Aus diesem Grund wird im Folgenden eine FIFO mit vier Einträgen verwendet. Diese Konfiguration stellt einen guten Kompromiss zwischen dem Mehrbedarf an Chipfläche (1,7 % Steigerung gegenüber einem FIFO mit einem Eintrag) und der Performanz (2,6 % Steigerung) dar.

5.2.3 CPU-Makros für die Verwendung im CoreVA-MPSoC

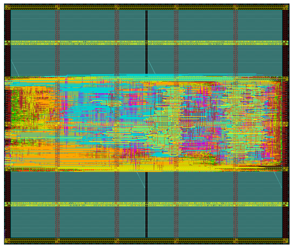
CPU-Makros der CoreVA-CPU werden im Rahmen dieser Arbeit verwendet, um die Reproduzierbarkeit und die Geschwindigkeit von Synthesen auf Ebene des CPU-Clusters zu steigern. In Abbildung 5.7 sind die Layouts von sechs verschiedenen CPU-Makros dargestellt (Makro (a) bis (f)). In Tabelle 5.2 sind verschiedene Eigenschaften dieser Makros aufgeführt. Die maximale Taktfrequenz beträgt bei jedem Makro 800 MHz. Die Speicherblöcke des lokalen L1-Datenspeichers befinden sich im oberen Bereich der



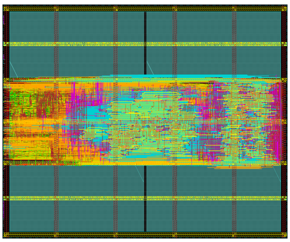
(a) 1 VLIW-Slot, 16 kB DMem



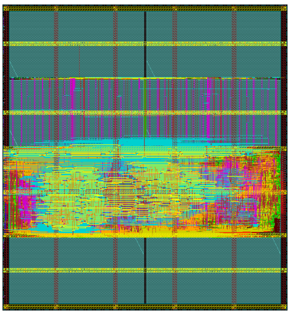
(b) 2 VLIW-Slots, 0 kB DMem, L1-Speicherschnittstelle



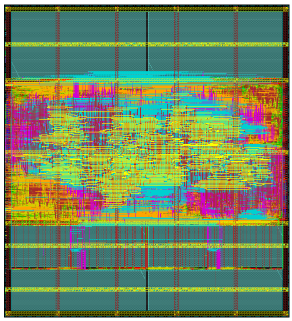
(c) 2 VLIW-Slots, 16 kB DMem, L1-Speicherschnittstelle



(d) 2 VLIW-Slots, 16 kB DMem



(e) 2 VLIW-Slots, 32 kB DMem



(f) 4 VLIW-Slots, 16 kB DMem

Abbildung 5.7: Layouts von verschiedenen CoreVA-CPU-Makros mit unterschiedlichen Datenspeicher-Konfigurationen (DMem), jeweils 16 kB Instruktionsspeicher und einer Taktfrequenz von 800 MHz

Tabelle 5.2: Eigenschaften von CPU-Makros der CoreVA-CPU

Makro	VLIW-Slots	MAC	Größe Datenspeicher [kB]	Schnittst. für gem. L1-Datenspeicher	Instr. Speicher [kB]	Datenbreite Instr. Speicher [bit]	Fläche [mm^2]
(a)	1	1	16	nein	16	64	0,103
(b)	2	1	0	ja	16	64	0,082
(c)	2	1	16	ja	16	64	0,116
(d)	2	1	16	nein	16	64	0,112
(e)	2	1	32	nein	16	64	0,148
(f)	4	2	16	nein	16	128	0,151

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 (b) verfügt über keinen Datenspeicher. Die Datenspeicher der Makros (a), (c), (d) und (f) bestehen aus jeweils zwei 8-kB-Speicherblöcken (siehe Abbildung 5.7b). Das Makro (e) 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. Das Makro mit vier VLIW-Slots integriert vier 4-kB-Speicher, die restlichen Makros jeweils zwei 8-kB-Speicher. Zwischen Instruktions- und Datenspeicher sind die Standardzellen platziert und verdrahtet.

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. Die I/O-Anschlüsse der CPU sind an der linken Seite des Makros platziert. Die Makros besitzen jeweils 129 I/Os für die

Master- sowie 103 I/Os für die Slave-Bus-Schnittstelle. 14 weitere I/Os können z. B. zur späteren Festlegung der CPU-ID der Makros verwendet werden. Die Makros (b) und (c) verfügen zudem über 91 I/O-Anschlüsse für den gemeinsamen L1-Datenspeicher

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 (b)) bis 407 µm (Makro (f)). Der hieraus resultierende Flächenbedarf liegt zwischen 0,082 mm^2 und 0,151 mm^2. Das Makro (c) verfügt im Gegensatz zu Makro (d) 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 Makros um 3,6 %. Das Makro (b) verfügt ebenfalls über diese Schnittstelle, integriert jedoch keinen lokalen L1-Datenspeicher. Die Flächenausnutzung (*Utilization*, siehe Abschnitt 3.1) der CPU-Makros weist im Standardzellenbereich sehr hohe Werte zwischen 90 % (Makro (a)) und 96 % (Makro (d)) auf.

5.2.4 Vergleich verschiedener VLIW-Konfigurationen im CPU-Cluster

In diesem Abschnitt wird die Integration von verschiedenen Konfigurationen der CoreVA-CPU in den CPU-Cluster des CoreVA-MPSoCs analysiert. Der Flächenbedarf von CPU-Clustern mit einer bis 32 CPUs bestehend aus CPUs mit einem, zwei oder vier VLIW-Slots ist in Abbildung 5.8 dargestellt. Für die Synthesen werden die CPU-Makros (a), (d) und (f) verwendet (siehe Abschnitt 5.2.3). Als Cluster-Verbindungsstruktur kommt ein geteilter AXI-Bus zum Einsatz. Es werden eine Master- und eine Slave-Registerstufe in die Verbindungsstruktur integriert, um eine Taktfrequenz von 800 MHz zu ermöglichen (siehe Abschnitt 5.5.2).

Der Flächenbedarf des CPU-Clusters steigt linear mit der Anzahl der integrierten CoreVA-CPUs an. Bei der Verwendung von zwei CPUs beträgt der Flächenbedarf der Verbindungsstruktur 0,022 mm^2. Bezogen auf die Gesamtfläche des Clusters ist der Anteil der Verbindungsstruktur 9,4 % (1 VLIW-Slot), 9,0 % (2 VLIW-Slots) bzw 6,4 % (4 VLIW-Slots). Dies resultiert in einem Flächenbedarf des gesamten CPU-Clusters von 0,23 mm^2, 0,25 mm^2 bzw. 0,32 mm^2. Die Konfiguration mit 32 CPUs besitzt einen Flächenbedarf von 3,64 mm^2, 3,94 mm^2 bzw. 5,17 mm^2. Der prozentuale Anteil der Verbindungsstruktur an der Gesamtfläche ist vergleichbar mit der des CPU-Clusters mit zwei CPUs.

In Abschnitt 5.2.1 wurde das Beschleunigungs-Flächen-Verhältnis von einzelnen 1-Slot-, 2-Slot- sowie 4-Slot-CoreVA-CPUs betrachtet. Die 1-Slot- und 2-Slot-CPUs zeigen vergleichbare Ergebnisse, während die 4-Slot-Konfiguration 33,1 % schlechter ist. Im Folgenden wird die Ausführungszeit von Anwendung bei einer Abbildung auf CPU-Cluster mit gleichem Flächenbedarf und unterschiedlichen CPU-Konfigurationen untersucht. Hierzu werden CPU-Cluster mit 14 1-Slot-CPUs (1,56 mm^2), 13 2-Slot-CPUs (1,57 mm^2) sowie 10 4-Slot-CPUs (1,59 mm^2) und AXI-Verbindungsstruktur betrachtet. In Abbildung 5.9 ist die Beschleunigung verschiedener Streaming-Anwendungen

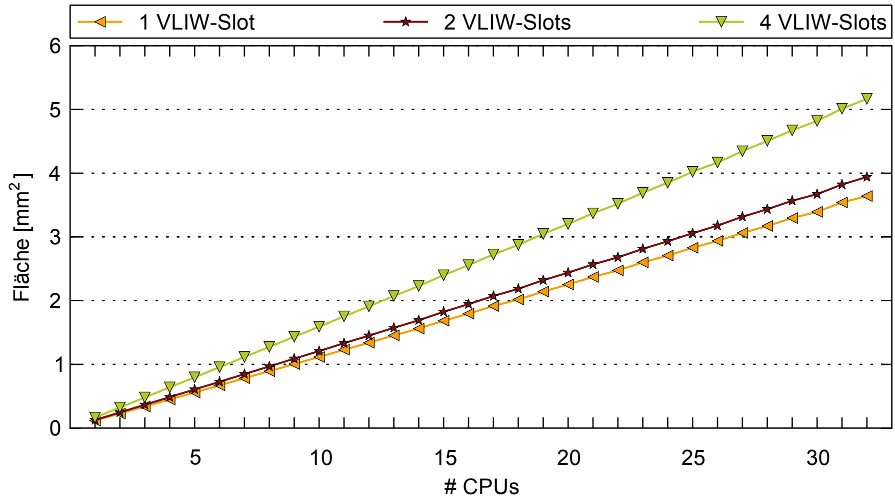


Abbildung 5.8: Flächenbedarf von CPU-Clustern mit einer bis 32 CPUs bestehend aus verschiedenen VLIW-Konfigurationen der CoreVA-CPU bei einer Frequenz von 800 MHz

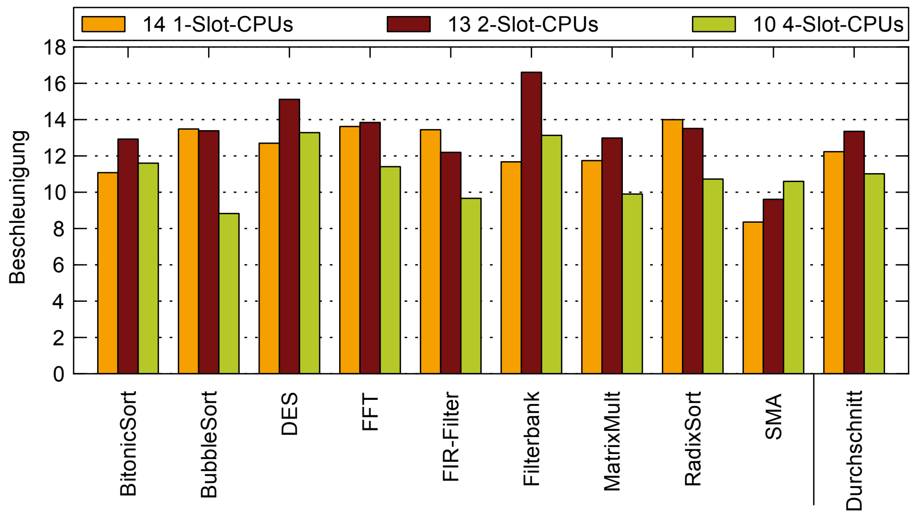


Abbildung 5.9: Beschleunigung von Streaming-Anwendungen bei der Ausführung auf CPU-Clustern mit verschiedenen VLIW-CPUs und einem Flächenbedarf von jeweils ca. 1,57 mm² gegenüber der Ausführung auf einer 1-Slot-CoreVA-CPU

gegenüber der Ausführung auf einer CoreVA-CPU mit einem VLIW-Slot dargestellt. Sowohl der LLVM- als auch der CoreVA-MPSoC-Compiler haben hierbei Einfluss auf die Performanz der Anwendungen.

Bei der Anwendung SMA zeigt die 4-Slot-Konfiguration die beste Performanz. SMA profitiert in einem hohen Maße von vier VLIW-Slots (siehe Abschnitt 5.2.1), verwendet jedoch nur sieben CPUs (siehe Abschnitt 5.5.1). FIR-Filter zeigt das entgegengesetzte Verhalten, da diese Anwendung nicht durch zusätzlichen VLIW-Slots beschleunigt wird, jedoch sehr gut auf viele CPUs parallelisierbar ist. Daher weist bei FIR-Filter der CPU-Cluster mit 14 1-Slot-CPUs die höchste Performanz auf. Dies ist ebenfalls bei BubbleSort und RadixSort der Fall. Bei den übrigen betrachteten Anwendungen zeigt die 2-Slot-Konfiguration die höchste Performanz. Insbesondere Filterbank profitiert von einem zweiten VLIW-Slot, ist gut auf 13 CPUs parallelisierbar und zeigt eine Beschleunigung von 16,6 gegenüber einer einzelnen 1-Slot-CPU.

Gemittelt über alle neun betrachteten Anwendungen weist der CPU-Cluster mit 13 2-Slot-CPUs eine Beschleunigung von 13,4 und somit die höchste Performanz auf. Die durchschnittliche Beschleunigung der 1-Slot-Konfiguration beträgt 12,2, die der 4-Slot-Konfiguration lediglich 11,0. Dieses Ergebnis ist darauf zurückzuführen, dass viele Anwendungen und insbesondere auch die Funktionen für die CPU-zu-CPU-Kommunikation von einem zweiten VLIW-Slot profitieren. Bei der 4-Slot-CPU überwiegt hingegen der Mehrbedarf an Chipfläche für zwei zusätzliche VLIW-Slots und die dadurch bedingte Reduzierung von 13 auf zehn CPUs pro Cluster. Hieraus folgt, dass, zumindest mit den vorliegenden Versionen des LLVM- und des CoreVA-MPSoC-Compilers, der Einsatz einer CoreVA-CPU mit zwei VLIW-Slots die höchste Performanz und Flächeneffizienz verspricht. Daher werden die folgenden Untersuchungen mit 2-Slot-CPUs durchgeführt.

5.3 Synchronisierungsverfahren und Speicherarchitekturen im Vergleich

Die Synchronisierung von parallel ausgeführten Anwendungsteilen bedingt einen Mehraufwand gegenüber der Ausführung einer Anwendung auf einer CPU. In diesem Abschnitt wird der Zeitaufwand[5] der CPUs für eine synchronisierte Kommunikation innerhalb eines CPU-Clusters betrachtet. Als Speicherarchitekturen werden ein lokaler L1-Speichers (L1-L) sowie gemeinsame L1- und L2-Speicher (L1-S bzw. L2-S) evaluiert (siehe Abschnitt 2.3). Zudem werden Kombinationen aus lokalem und gemeinsamem Speicher betrachtet. Für die Untersuchungen wird ein synthetischer C-Benchmark verwendet. Die Synchronisierung der Kommunikation erfolgt auf Wort- sowie auf Blockebene (siehe Abschnitt 2.6). Es werden jeweils 64 B bzw. 1 kB von einer CPU produziert bzw. gesendet und von einer zweiten CPU empfangen bzw. gelesen. In Tabelle 5.3 und Tabelle 5.4 sind die CPU-Takte für den Aufruf der jeweiligen Synchro-

[5] in CPU-Takten

Tabelle 5.3: Vergleich verschiedener Speicherarchitekturen bei der Verwendung eines wortbasierten Ringpuffer-Synchronisierungsverfahrens bezüglich Ausführungszeit (in CPU-Takten). Als Speicherarchitekturen werden hierbei ein lokaler L1-Speicher (L1-L), gemeinsame L1- und L2-Speicher (L1-S bzw. L2-S) sowie Kombinationen hiervon betrachtet.

	L1-L	L1-S	L1-S + L1-L	L2-S	L2-S + L1-L
pushElement	21	20	20	23	21
popElement	18	18	18	22	20
Schreiben 64 B	329	329	327	375	329
Lesen 64 B	308	308	308	404	356
Schreiben 1 kB	5127	5130	5127	6376	5128
Lesen 1 kB	5120	5107	5120	6406	5637

nisierungsfunktionen aufgeführt. Zudem sind die CPU-Takte angegeben, die Sender und Empfänger insgesamt für das Schreiben bzw. Lesen des Datenblocks sowie die Synchronisierung benötigen. Alle in diesem Abschnitt angegebenen Messergebnisse stellen den bestmöglichen Fall (*Best-Case*) dar. Hierzu wird jede Konfiguration 30-mal ausgeführt und es wird die Messung mit der geringsten Anzahl an Takten ausgewählt. Die durchschnittliche Dauer von Speicherzugriffen im CPU-Cluster ist stark von der Auslastung der Cluster-Verbindungsstruktur sowie den Zugriffsmustern der Anwendung abhängig. Aus diesem Grund wird an dieser Stelle kein Durchschnittswert oder ein Maximum der Messungen angegeben. Die Konfiguration des CPU-Clusters ist für die Untersuchungen so dimensioniert, dass eine Überlastung der Kommunikationsinfrastruktur weitgehend vermieden wird. Die CoreVA-CPUs besitzen zwei VLIW-Slots und das FIFO der CPU-Cluster-Schnittstelle verfügt über 32 Einträge. Als Cluster-Verbindungsstruktur wird eine vollständige AXI-Crossbar ohne Registerstufen verwendet. Der gemeinsame L1-Datenspeicher besitzt 16 Bänke. In einem ersten Schritt wird eine optimierte Implementierung eines Ringpuffers betrachtet, der als Block- bzw. Elementgröße ein Datenwort (4 B) verwendet. Es wird also vor jedem Lese- bzw. Schreibzugriff auf den Ringpuffer überprüft, ob das jeweilige Element gelesen bzw. beschrieben werden darf. Die Implementierung ist auf eine Elementgröße von 4 B hin optimiert. Das Rücksetzen der Ringpuffer-Zähler ist nicht über eine Modulo-Operation realisiert. Stattdessen wird eine bitweise Und-Verknüpfung mit der Anzahl der Elemente des Ringpuffers durchgeführt. Hierdurch ist die Größe des Ringpuffers auf Potenzen von Zwei begrenzt. Bei der Verwendung von lokalem L1-Speicher für die Zähler des Puffers sind zudem keine Lesezugriffe über die Cluster-Verbindungsstruktur erforderlich. Ohne diese Optimierungen ist die Laufzeit der Ringpuffer-Implementierung im Vergleich zu den hier vorgestellten Werten deutlich höher. In Abbildung A.4 im Anhang ist ein Beispielprogramm mit einem wortbasierten Ringpuffer dargestellt.

Tabelle 5.4: Vergleich verschiedener Speicherarchitekturen bei der Verwendung eines blockbasierten Synchronisierungsverfahrens bezüglich Ausführungszeit (in CPU-Takten)

	L1-L	L1-S	L1-S + L1-L	L2-S	L2-S + L1-L
GetBuffer	16	16	16	18	16
SetBuffer	16	16	16	19	19
Schreiben 64 B	54	54	54	70	54
Lesen 64 B	53	53	53	116	104
Schreiben 1 kB	1317	1573	1317	1559	1317
Lesen 1 kB	1573	1573	1573	2384	2323

In Tabelle 5.3 ist die Ausführungszeit in CPU-Takten für eine Ringpuffer-Synchronisierung auf verschiedenen Speicherarchitekturen dargestellt. Durch Optimierungen des LLVM-Compilers können die Ergebnisse der einzelnen Konfigurationen um wenige Takte variieren. Die erste Zeile der Tabelle zeigt die geringstmöglichen CPU-Takte, um ein Element in den Ringpuffer zu schreiben (*pushElement*). Alle betrachteten Konfigurationen benötigen 20 bzw. 21 Takte, lediglich bei der ausschließlichen Verwendung vom gemeinsamen L2-Speicher (L2-S) sind aufgrund eines Lesezugriffes auf ebendiesen Speicher 23 Takte notwendig. Das Lesen eines Elementes (*popElement*) zeigt ein ähnliches Verhalten. L1-L, L1-S sowie L1-S + L1-L benötigen 18 Takte, während die beiden L2-Konfigurationen zwei bzw. vier Takte mehr benötigen. Wird die Zeit für die Übertragung eines Datenblocks betrachtet, benötigen die L1-basierten Speicherarchitekturen 329 Takte für das Schreiben bzw. 308 Takte für das Lesen von 64 B. Dies entspricht im Durchschnitt 5,14 CPU-Takte für das Schreiben und 4,81 CPU-Takte für das Lesen eines Bytes. Die Übertragung eines Datenblocks mit 1 kB Größe dauert pro Byte 5,00 bzw. 4,99 Takte.

Bei der Verwendung des L2-Speichers für Synchronisierung und Daten (L2-S) werden aufgrund einer höhere Zugriffslatenz und durch Zugriffskonflikte 25 % mehr CPU-Takte im Vergleich zu den Konfigurationen mit L1-Speicher benötigt. Wird der L2-S-Speicher für Daten und der lokale L1-Datenspeicher für die Synchronisierung verwendet, dauert das Lesen der Daten 10 % länger im Vergleich zu L1-S. Das Schreiben des Datenblocks erfordert keine zusätzlichen Takte im Vergleich zu L1-L, da alle Zugriffe auf den L2-Speicher durch das FIFO der CPU-Cluster-Schnittstelle zwischengespeichert werden können.

In Tabelle 5.4 ist die Ausführungszeit für die Kommunikation mit dem im Rahmen dieser Arbeit entworfenen blockbasierten Synchronisierungsverfahren (siehe Abschnitt 2.6.4) dargestellt. Vor dem ersten Zugriff auf einen Datenblock (Puffer) muss überprüft werden, ob der Puffer schreib- bzw. lesbar ist (*GetBuffer*). Zudem muss nach dem Beschreiben bzw. Lesen eines Datenblocks dem Kommunikationspartner Zugriff auf den Datenblock gewährt werden (*SetBuffer*). In Abbildung A.5 im Anhang ist ein

Beispielprogramm mit blockbasierter Synchronisierung dargestellt. Die Konfigurationen mit L1-Speicher benötigen hierfür jeweils 16 CPU-Takte. Die beiden Konfigurationen mit L2-Speicher besitzen wiederum eine um zwei bzw. drei Takte verlängerte Laufzeit aufgrund einer erhöhten Zugriffslatenz. Im Vergleich zum Ringpuffer wird die Ausführungszeit für die Übertragung eines 64 B großen Datenblocks drastisch auf 54 CPU-Takte (Schreiben) bzw. 53 Takte (Lesen) reduziert, also um bis zu 83,59 %. Für die Übertragung von 1 kB beträgt die Reduktion bis zu 74,31 %. Dies ist darauf zurückzuführen, dass nur zweimal pro Datenblock (GetBuffer und SetBuffer) statt bei jedem Datenwort synchronisiert wird.

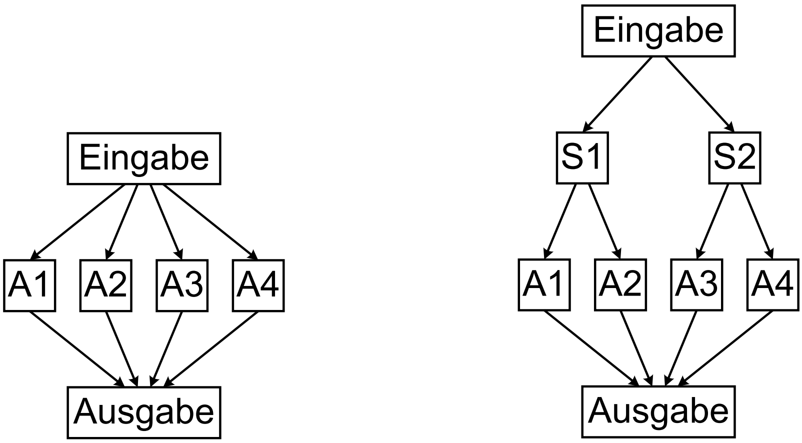
Die Speicherkonfigurationen L1-S und L2-S sind auf Seiten der Sende-CPU beim 1-kB-Datenblock aufgrund von Zugriffskonflikten 19,44 % bzw. 18,38 % im Vergleich zur L1-L-Konfiguration langsamer. Bei der Empfänger-CPU werden 51,56 % (L2-S) bzw. 47,68 % (L2-S + L1-L) mehr Takte benötigt. Hierfür ist wiederum die erhöhte Latenz von Lesezugriffen auf den L2-Speicher verantwortlich.

In diesem Abschnitt konnten die Vorteile von blockbasierter Synchronisierung gegenüber einer Synchronisierung auf Wortebene quantifiziert werden. Hierfür ist eine konstante Blockgröße während der Ausführung der Anwendung notwendig, was jedoch bei einer Vielzahl von Streaming-Anwendungen der Fall ist. Die Verwendung von lokalem und/oder gemeinsamem L1-Speicher für die CPU-zu-CPU-Kommunikation im CPU-Cluster zeigt Vorteile gegenüber der Verwendung von gemeinsamem L2-Speicher. Hauptgrund hierfür sind die hohen Latenzen von Lesezugriffen auf den gemeinsamen L2-Speicher. Der L2-Speicher kann jedoch für Instruktionen in Verbindung mit einem L1-Instruktionscache verwendet werden (siehe Abschnitt 2.3.1 und Abschnitt 5.7). Zudem kann der L2-Speicher für die Ablage von Daten verwendet werden, deren Zugriffslatenzen nicht relevant für die Performanz der Anwendung sind.

5.4 C-basierte Implementierung einer Matrixmultiplikation

In diesem Abschnitt wird eine direkt in der Programmiersprache C beschriebene Implementierung der Anwendung Matrixmultiplikation betrachtet. Im Gegensatz zu in StreamIt implementierten Anwendungen ist keine automatisierte Partitionierung durch den CoreVA-MPSoC-Compiler möglich. Aus diesem Grund muss der Programmierer einen iterativen Entwicklungsprozess durchführen, bei dem die Partitionierung der Anwendung auf die verfügbaren CPUs des CoreVA-MPSoCs optimiert wird. Insbesondere bei der Verwendung von vielen CPU-Kernen existieren sehr viele mögliche Partitionierungen. Ein Ziel dieses Abschnitts ist es, die Größe dieses Entwurfsraums aufzuzeigen.

Die Matrixmultiplikation $C = A \cdot B$ findet in vielen Einsatzgebieten eingebetteter Prozessorsysteme Anwendung und ist in Abschnitt 5.1 detailliert beschrieben. Die im Rahmen dieser Arbeit verwendete Implementierung der Matrixmultiplikation nutzt eine oder mehrere CPUs, um die Eingabematrizen zu verteilen (*Splitter-CPUs*). Dies vermeidet Lesezugriffe über die Cluster-Verbindungsstruktur. Die Arbeiter- oder *Worker-*



(a) Eine CPU teilt die Daten auf

(b) Drei CPUs (Eingabe, S1, S2) teilen die Daten in zwei Stufen auf

Abbildung 5.10: Mögliche Partitionierungen einer Matrixmultiplikation mit Aufteilung der Berechnung auf vier Arbeiter-CPUs (A)

CPUs erhalten die gesamte Matrix B sowie eine oder mehrere Zeilen der Matrix A. Jede Arbeiter-CPU berechnet eine oder mehrere Zeilen von C und sendet diese an eine weitere CPU, welche die einzelnen Zeilen von C zusammenfügt. Die Berechnung der Matrixmultiplikation wird somit zeilenweise parallelisiert. Eine Konfiguration der Matrixmultiplikation mit insgesamt sechs CPUs ist in Abbildung 5.10a dargestellt. Eine Eingabe-CPU verteilt die jeweils benötigten Teile der Eingabematrizen A und B auf die vier Arbeiter-CPUs. Bei der Verwendung von 8×8-Matrizen berechnet jede Arbeiter-CPU zwei Zeilen der Ergebnismatrix C. Hierzu benötigt die Arbeiter-CPU die gesamte Matrix B sowie zwei Zeilen der Matrix A. Die Ausgabe-CPU kombiniert die berechneten Ergebnisse der Arbeiter-CPUs zur Ergebnismatrix.

In Abbildung 5.11 ist die Beschleunigung für verschiedenen Matrixgrößen (8×8, 16×16, 32×32 sowie 64×64) und zwei bis 32 Arbeiter-CPUs dargestellt. Als Referenz dient die Ausführung auf einer einzelnen CPU. Die 8×8-Matrix zeigt bei der Verwendung von zwei Arbeiter-CPUs und jeweils einer Eingabe- und Ausgabe-CPU eine Beschleunigung um den Faktor 2,06 gegenüber der Ausführung auf einer CPU. Alle Arbeiter-CPUs sind voll ausgelastet und bestimmen den Durchsatz der Anwendung. Die Ein- und Ausgabe-CPUs weisen nur eine geringe Auslastung auf. Bei der Verwendung von vier Arbeiter-CPUs beträgt die Beschleunigung 2,75. Die Performanz wird von der Eingabe-CPU begrenzt. Bei der Verwendung von acht Arbeiter-CPUs wird eine Beschleunigung von lediglich 1,53 erreicht. Mit einer zunehmenden Anzahl an CPUs sinkt also die

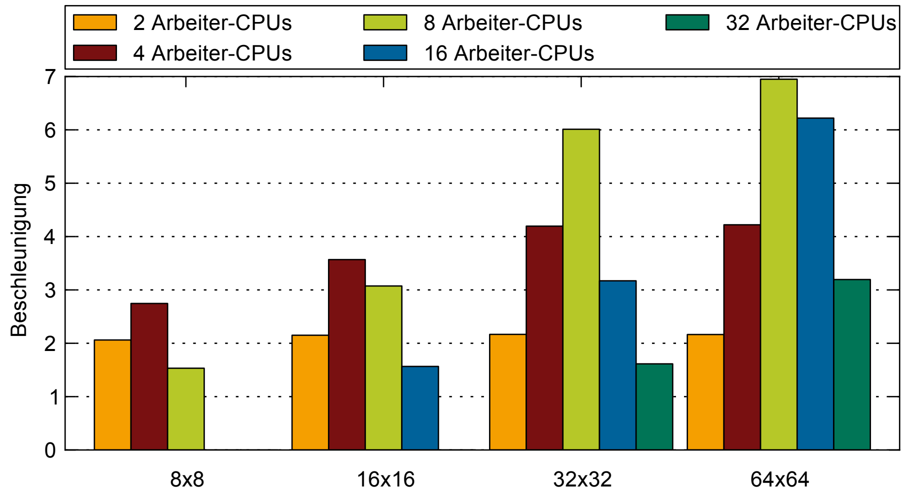


Abbildung 5.11: Beschleunigung der Anwendung Matrixmultiplikation mit verschiedenen Matrixgrößen bei der Ausführung auf zwei bis 32 Arbeiter-CPUs und einer Splitter-CPU gegenüber der Ausführung auf einer CPU

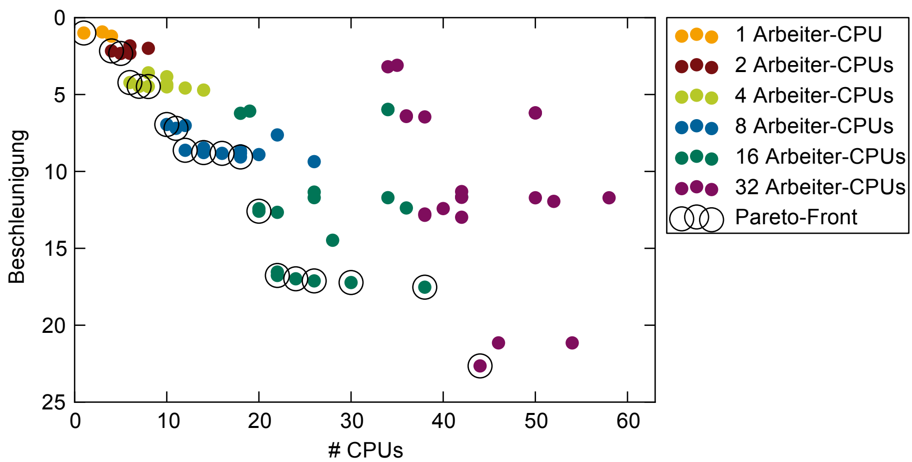


Abbildung 5.12: Beschleunigung verschiedener Konfigurationen der Matrixmultiplikation mit 64×64 Elementen. Die Datenpunkte sind nach der Anzahl an Arbeiter-CPUs eingefärbt. Die Punkte der Pareto-Front sind mit einem Kreis markiert.

Performanz. Dies ist auf den hohen Aufwand für das Aufteilen der Eingabematrizen auf die Arbeiter-CPUs zurückzuführen. Eine Verwendung von 16 und 32 Arbeiter-CPUs ist aufgrund der Matrixgröße von 8×8 nicht möglich. Bei der Verwendung einer 16×16-Matrix beträgt die maximale Beschleunigung 3,57 (vier Arbeiter-CPUs). Acht und 16 Arbeiter-CPUs verschlechtern die Performanz der Anwendung.

Für eine Matrixgröße von 32×32 zeigen die Konfigurationen mit zwei und vier CPUs eine ähnliche Beschleunigung wie die 16×16-Matrix (2,17 bzw. 4,20). Eine Verwendung von acht Arbeiter-CPUs führt zu einer Beschleunigung von 6,01. Hier begrenzt wiederum die Eingabe-CPU den Durchsatz. Dies ist ebenfalls bei der Verwendung von 16 und 32 Arbeiter-CPUs der Fall. Gegenüber der 16×16-Matrix ist nahezu eine Verdopplung der Beschleunigung zu beobachten. Dies liegt daran, dass der Aufwand der Berechnungen stärker steigt als der Aufwand der Eingabe-CPU für das Aufteilen der Daten. Die 64×64 Matrix zeigt eine maximale Beschleunigung von 6,95 bei der Verwendung von acht Arbeiter-CPUs.

Je nach Matrixgröße ist bei vier oder acht Arbeiter-CPUs die größte Beschleunigung zu beobachten. Werden mehr CPUs verwendet, begrenzt die Eingabe-CPU den Durchsatz und die Geschwindigkeit gegenüber der Ausführung auf einer CPU sinkt. Um den Durchsatz der Matrixmultiplikation weiter zu erhöhen, können zwei oder drei Stufen für das Verteilen der Daten an die Arbeiter-CPUs verwendet werden. In Abbildung 5.10b ist eine Konfiguration mit insgesamt drei Splitter-CPUs dargestellt.

Jeder Punkt in Abbildung 5.12 stellt eine Konfiguration der Matrixmultiplikation für eine Matrixgröße von 64×64 dar. Die x-Achse zeigt die Anzahl der verwendeten CPUs, die y-Achse die Beschleunigung verglichen mit der Ausführung auf einer einzelnen CPU. Die Konfigurationen sind anhand der Anzahl der verwendeten Arbeiter-CPUs (1 bis 32) eingefärbt. Zudem wird die Anzahl der Splitter-CPUs (1 bis 25) und der Splitter-Stufen (1 bis 3) variiert. Es ist nicht möglich, aus den insgesamt 70 betrachteten Konfigurationen „die“ beste Konfiguration auszuwählen. Der Grund hierfür ist, dass die Zielgrößen Beschleunigung und Anzahl verwendeter CPUs in Konkurrenz zueinander stehen. Hierbei soll die Beschleunigung maximiert und die Anzahl der CPUs minimiert werden. In Abbildung 5.12 sind alle Konfigurationen mit einem schwarzen Kreis hervorgehoben, für die keine andere Konfiguration existiert, die in allen Zielgrößen mindestens genauso gut und in mindestens einer Zielgröße echt besser ist. Diese Konfigurationen sind Teil der sogenannten Pareto-Front[6] [24, S. 10; 65]. Jedes n-Tupel $x = (x_1, x_2, ..., x_n)$ aus der Menge A ist Teil der Pareto-Front, wenn kein Tupel $y = (y_1, y_2, ..., y_n)$ existiert, für das gilt $y_i \geq x_i \ \forall i \in \{1, .., n\}$ und $y_i > x_i \ \exists i \in \{1, .., n\}$ [233, S. 58]. Für die in Abbildung 5.12 dargestellte Analyse gilt $n = 2$ und die Anzahl an CPUs soll minimiert statt maximiert werden.

[6] Benannt nach Vilfredo Pareto

Insgesamt sind 19 Konfigurationen Teil der Pareto-Front. Die Konfiguration der Pareto-Front mit der geringsten Anzahl an CPUs ist die sequenzielle Implementierung, die auf genau einer CPU ausgeführt wird. Die höchste Beschleunigung von 22,6 zeigt eine Konfiguration mit insgesamt 44 CPUs. Hiervon teilen 11 CPUs in drei Stufen die Daten auf, 32 CPUs führen die eigentliche Matrixmultiplikation aus und eine CPU fügt die Ergebnismatrix zusammen. Die erste Aufteilungsstufe besteht hierbei aus einer CPU, die zweite aus zwei und die dritte Stufe aus acht CPUs. Sechs Konfigurationen aus der Pareto-Front nutzen 16 Arbeiter-CPUs. Hiervon erreichen fünf Konfigurationen eine Beschleunigung im Bereich von 16,8 bis 17,5. Die Arbeiter-CPUs sind bei diesen Konfigurationen voll ausgelastet und begrenzen den Durchsatz. Eine Konfiguration der Pareto-Front nutzt 16 Arbeiter-CPUs, erreicht jedoch nur eine Beschleunigung von 12,6. Hier begrenzen die Splitter-CPUs den Durchsatz. Die Konfigurationen der Pareto-Front mit acht, vier bzw. zwei Arbeiter-CPUs erreichen eine Beschleunigung von bis zu 9,1, 4,5 bzw. 2,3, wobei hier die Arbeiter-CPUs den Durchsatz begrenzen.

Zusammenfassend kann festgestellt werden, dass, obwohl die Matrixmultiplikation eine einfache Struktur besitzt, eine gleichmäßige Auslastung aller verwendeten CPUs nur schwer zu erreichen ist. Bei der Verwendung von vielen Arbeiter-CPUs begrenzt das Aufteilen der Eingabematrizen die Performanz der Anwendung. Dies ist von der Größe der Matrizen abhängig. Wird das Aufteilen der Daten in mehreren Stufen durchgeführt, kann die Performanz der Anwendung deutlich gesteigert werden. Dies vergrößert den Entwurfsraum deutlich und führt zu vielen Konfigurationen, die Teil der Pareto-Front sind.

Die in diesem Abschnitt vorgestellten Untersuchungen einer C-basierten Anwendung veranschaulichen beispielhaft die Größe des Software-Entwurfsraums bei der Abbildung von Anwendungen auf ein Multiprozessorsystem. Eine vergleichbar gute Optimierung bzw. Partitionierung mehrerer Beispielanwendungen auf verschiedene Konfigurationen des CoreVA-MPSoCs wäre nur mit einem sehr hohen Aufwand möglich. Daher wird im Folgenden der CoreVA-MPSoC-Compiler verwendet, um automatisiert Streaming-Anwendungen auf das CoreVA-MPSoC abzubilden und hierauf basierend eine Entwurfsraumexploration des CPU-Clusters durchzuführen.

5.5 Entwurfsraumexploration der Cluster-Verbindungsstruktur

In diesem Abschnitt werden verschiedene Konfigurationen von eng gekoppelten Cluster-Verbindungsstrukturen bezüglich des Ressourcenbedarfs und der Performanz verglichen. Es werden ein geteilter Bus, partielle und vollständige Crossbar-Topologien sowie die Bus-Standards OpenCores Wishbone und ARM AMBA AXI betrachtet (siehe Abschnitt 2.4). Für die Untersuchungen wird das CPU-Makro vom Typ (d) der CoreVA-CPU mit zwei VLIW-Slots und jeweils 16 kB Daten- und Instruktionsspeicher verwendet (siehe Abschnitt 5.2.3). Teile der Ergebnisse dieses Abschnittes wurden in [223] und [226] veröffentlicht.

5.5.1 Abbildung von Streaming-Anwendungen auf den CPU-Cluster

Im vorangegangen Abschnitt wurde die Größe des Entwurfsraums bei der Abbildung von Anwendungen auf MPSoCs anhand einer C-basierten Anwendung aufzeigt. In diesem Abschnitt wird die Performanz von Streaming-Anwendungen analysiert, die vom CoreVA-MPSoC-Compiler automatisiert auf CPU-Cluster mit vier bis 32 CPUs abgebildet werden. Details zum CoreVA-MPSoC-Compiler werden in Abschnitt 3.2.3 beschrieben.

Die Beschleunigung von neun Streaming-Anwendungen bei der Abbildung auf vier verschiedene CPU-Cluster ist in Abbildung 5.13 dargestellt. Als Referenz wird die Ausführungszeit der jeweiligen Anwendung bei der Ausführung auf einer CPU verwendet. Zudem ist der Durchschnitt der Beschleunigung aller neun Anwendungen aufgeführt. Die Cluster-Verbindungsstruktur verwendet eine AXI-Crossbar mit je einer Master- und Slave-Registerstufe vom Typ 2 (siehe Abschnitt 5.5.2). Die verwendeten Streaming-Anwendungen werden in Abschnitt 5.1 beschrieben. Der CoreVA-MPSoC-Compiler verwendet einen *Simulated-Annealing*-Optimierungsalgorithmus, um eine gute Partitionierung der Anwendung für die gegebene Konfiguration des CPU-Clusters zu finden. Da

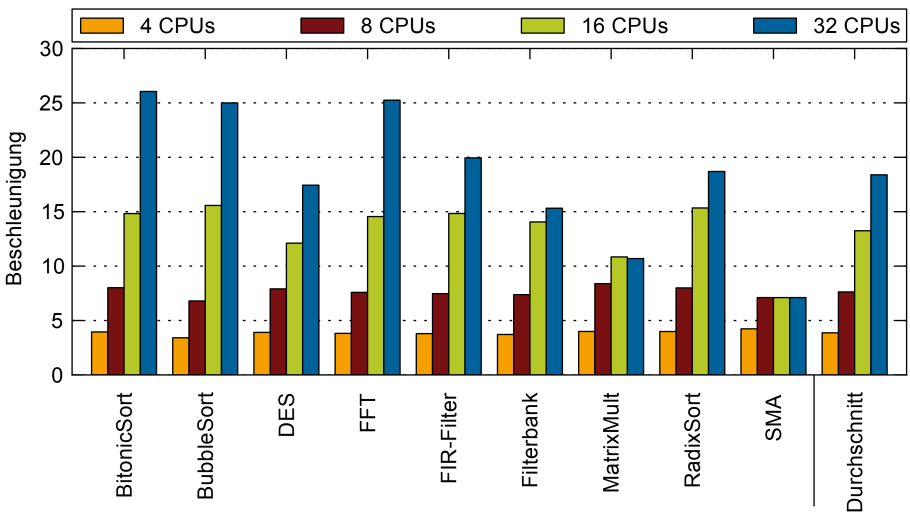


Abbildung 5.13: Beschleunigung von Streaming-Anwendungen bei der Abbildung auf vier bis 32 CPUs gegenüber der Ausführung auf einer CPU

der Suchraum des Optimierungsalgorithmus sehr groß ist, ist es nur schwer möglich, die „optimale“ Partitionierung mit der bestmöglichen Beschleunigung bzw. Performanz zu finden.

Bei der Verwendung von vier CPUs beträgt die durchschnittliche Beschleunigung aller neun Anwendungen 3,9 gegenüber der Ausführung auf einer CPU. Die Anwendung SMA weist mit 4,2 eine superskalare Beschleunigung auf. Die Beschleunigung ist also höher als der Zuwachs an CPU-Kernen. Dies ist im vorliegenden Fall auf den Zuwachs an Datenspeicher durch die Verwendung von mehr CPUs zurückzuführen. Der CoreVA-MPSoC-Compiler kann hierdurch die Blockgröße der Berechnung (Multiplikator) der Anwendungen vergrößern (siehe Abschnitt 3.2.3). Infolge der höheren Blockgröße verbessert sich das Verhältnis von Berechnung und Synchronisierung.

Die Anwendung BubbleSort weist mit 3,4 die geringste Beschleunigung aller Anwendungen auf. Dies ist darauf zurückzuführen, dass BubbleSort mit 66 eine sehr hohe Anzahl an Filtern besitzt. Die zu sortierenden Daten durchlaufen nacheinander alle Filter in einer Pipeline, was in 65 Kommunikationskanälen zwischen den Filtern resultiert. Der CoreVA-MPSoC-Compiler partitioniert nur elf aufeinanderfolge Filter auf der gleichen CPU. 54 Filter kommunizieren hingegen mithilfe der *Cluster-Channel* über die Verbindungsstruktur (siehe Abschnitt 2.6.4). Die Synchronisierung von CPU-zu-CPU-Kommunikation benötigt ca. 4,8-mal mehr Takte als die Verwaltung eines CPU-internen Kommunikationskanals (ca. 145 gegenüber 30 Takten). Aus diesem Grund weist BubbleSort nur eine Beschleunigung von 3,4 bei der Verwendung von vier CPUs auf. Zudem erhöht der CoreVA-MPSoC-Compiler die Blockgröße der Berechnungen nicht. BubbleSort könnte beschleunigt werden, wenn der CoreVA-MPSoC-Compiler Filter zusammenfassen könnte („Filter-Fusion“). Dies würde zu weniger Kommunikationskanälen zwischen CPUs führen und ist als Bestandteil von zukünftigen Versionen des CoreVA-MPSoC-Compilers geplant.

Der CPU-Cluster mit acht CPUs erreicht eine durchschnittliche Beschleunigung von 7,6. Eine superskalare Beschleunigung von 8,4 ist bei der Anwendung MatrixMult zu beobachten. Die Anwendung BubbleSort erreicht mit 6,8 wiederum die geringste Beschleunigung aller Anwendungen.

Die Verwendung von 16 CPUs führt zu einer durchschnittlichen Beschleunigung von 13,3. Die Anwendung SMA zeigt gegenüber der Ausführung auf einem CPU-Cluster mit acht CPUs keine weitere Beschleunigung, da nur sieben CPUs Programmcode der Anwendung ausführen. SMA besteht aus lediglich drei Filtern und besitzt im Vergleich zu den anderen betrachteten Anwendungen ein schlechtes Verhältnis von Rechen- und Kommunikationsaufwand. Eine CPU verteilt die Eingabedaten an die fünf Rechen-CPUs[7] und begrenzt hierdurch den Durchsatz. Die Performanz von SMA könnte durch die Verwendung von mehreren CPUs für das Aufteilen der Daten beschleunigt werden (siehe Abschnitt 5.4). Eine Integration dieser Optimierung in den CoreVA-MPSoC-Compiler wird zum Zeitpunkt der Entstehung dieser Arbeit evaluiert. Die Anwendung BubbleSort

[7] Der mittlere Filter ist vom CoreVA-MPSoC-Compiler fünfmal dupliziert worden

zeigt mit einer Beschleunigung von 15,6 die höchste Performanz. Im Vergleich zu den Cluster-Konfigurationen mit vier und acht CPUs können bei der Verwendung von 16 CPUs die 66 Filter der Anwendung effizient auf alle verfügbaren CPUs aufgeteilt werden, sodass der Mehraufwand für die CPU-zu-CPU-Kommunikation in den Hintergrund tritt. Zudem ist die Blockgröße der Anwendung vom CoreVA-MPSoC-Compiler im Vergleich zu den Konfigurationen mit weniger CPUs deutlich erhöht worden.

Bei der Verwendung von 32 CPUs zeigt SMA die gleiche Performanz wie bei den CPU-Clustern mit acht und 16 CPUs. Die Anwendung MatrixMult weist eine geringfügig schlechtere Performanz im Vergleich zum CPU-Cluster mit 16 CPUs auf. Dies ist auf Ungenauigkeiten in der Abschätzung des CoreVA-MPSoC-Compilers zurückzuführen (siehe Abschnitt 4.2.4). Die durchschnittliche Beschleunigung der betrachteten Anwendungen bei der Ausführung auf einem CPU-Cluster mit 32 CPUs beträgt 18,4. Eine zu geringe Komplexität und Parallelität der Anwendungen beschränkt die Performanz bei der Verwendung von 32 CPUs. Anwendungen profitieren in der Regel immer von zusätzlichem Datenspeicher, wenn zusätzliche CPUs verwendet werden. Hierdurch kann die Blockgröße der Berechnung erhöht werden. Allerdings beschränkt das Aufteilen von Daten auf verschiedene CPUs bei allen Anwendungen die Performanz. Eine Lösung hierfür ist die Verwendung zusätzlicher CPUs für das Aufteilen von Daten. Als weitere Möglichkeit kann ein gemeinsamer eng gekoppelter L1-Datenspeicher (siehe Abschnitt 5.6) für CPU-zu-CPU-Kommunikation verwendet werden. Hierfür ist allerdings der Einsatz eines angepassten Synchronisierungs- und Kommunikationsmodells sinnvoll.

5.5.2 Registerstufen

Um die maximale Taktfrequenz der Verbindungsstruktur zu steigern, können Registerstufen integriert werden. Es ist möglich, Registerstufen zwischen den Master-Ports der CPUs und der Verbindungsstruktur sowie zwischen der Verbindungsstruktur und den CPU-Slave-Ports einzufügen. Hierfür besitzt jede Registerstufe zwei Schnittstellen nach dem AXI- bzw. Wishbone-Standard. Zudem stehen zwei verschiedene Ausprägungen an Registerstufen zur Verfügung.

Typ 1 entkoppelt alle Daten- und Adresssignale von und zur Verbindungsstruktur mit einem Register. Bei der Implementierung nach dem Wishbone-Standard sind dies 114 Register (Master – Verbindungsstruktur) bzw. 109 Register (Verbindungsstruktur – Slave). Eine Registerstufe nach dem AXI-Standard benötigt insgesamt 175 Register, wenn alle fünf AXI-Kanäle registriert werden. Eine neue Anfrage darf allerdings erst in diesen Registern gespeichert werden, wenn die Verbindungsstruktur bzw. der Slave die vorherige Anfrage entgegennimmt. Dies wird durch ein *Ready*- (AXI) bzw. *Stall*-Signal (Wishbone) signalisiert. Dieses Signal ist für einen kombinatorischen Pfad durch die Registerstufe verantwortlich, der die maximale Taktfrequenz eines CPU-Clusters beschränken kann.

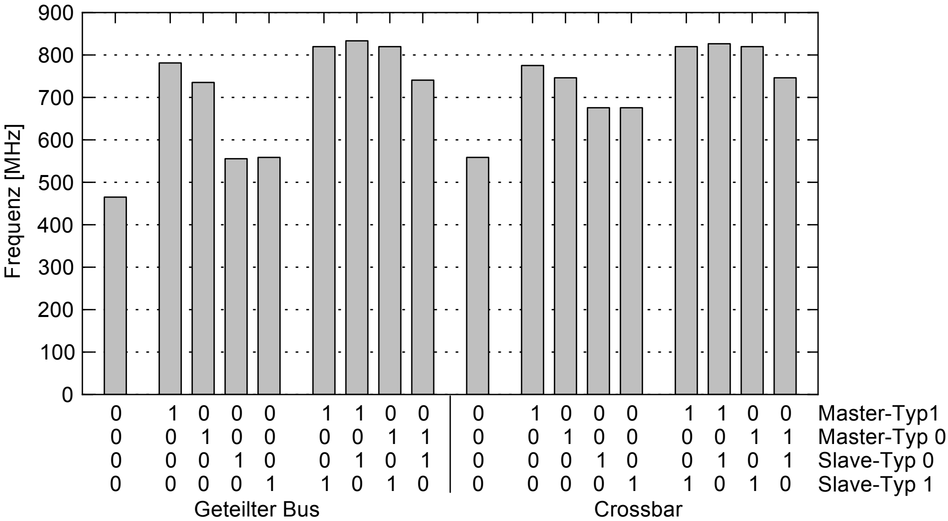


Abbildung 5.14: Maximale Taktfrequenz von CPU-Clustern mit acht CPUs, AXI-Verbindungsstruktur und verschiedenen Registerstufen. Registerstufen sind entsprechend der Anzahl von Master-Typ 2-, Master-Typ 1-, Slave-Typ 1- bzw. Slave-Typ 2-Registern angegeben.

Die zweite Variante („Typ 2") vermeidet diesen kombinatorischen Pfad, indem jeweils zwei Register für alle Signale der Registerstufe zur Verfügung stehen. Anfragen eines Masters bzw. der Verbindungsstruktur werden abwechselnd in beiden Registern gespeichert. Hierdurch muss der Master bzw. die Verbindungsstruktur erst mit einem Takt Verzögerung angehalten werden, falls eine Anfrage nicht weitergeleitet werden kann. Diese Implementierung einer Registerstufe wird auch als Doppelregister bezeichnet.

Im Folgenden wird der Einfluss auf die maximale Taktfrequenz und den Flächenbedarf eines CPU-Clusters mit acht CPUs analysiert, wenn verschiedene Konfigurationen von Registerstufen integriert werden. Als Cluster-Verbindungsstruktur wird ein geteilter AXI-Bus sowie eine AXI-Crossbar verwendet.

In Abbildung 5.14 ist die maximale Taktfrequenz bei der Integration von bis zu zwei Registerstufen dargestellt. Die x-Achse ist mit der Anzahl und dem Typ der jeweils verwendeten Registerstufen (Master-Typ 2, Master-Typ 1, Slave-Typ 1 und Slave-Typ 2, siehe Abschnitt 2.5) beschriftet. Ohne die Verwendung von Registerstufen (0 0 0 0) beträgt die maximale Taktfrequenz des CPU-Clusters 465 MHz (geteilter Bus) bzw. 559 MHz (Crossbar). Der kritische Pfad führt vom Master-Port einer CPU durch die Verbindungsstruktur und einen Arbiter zu einem Slave-Port einer anderen CPU. Die Crossbar-Topologie ermöglicht eine höhere Frequenz als der geteilte Bus, da es keinen zentralen Arbiter gibt, sondern jeder Slave-Port einen Arbiter besitzt.

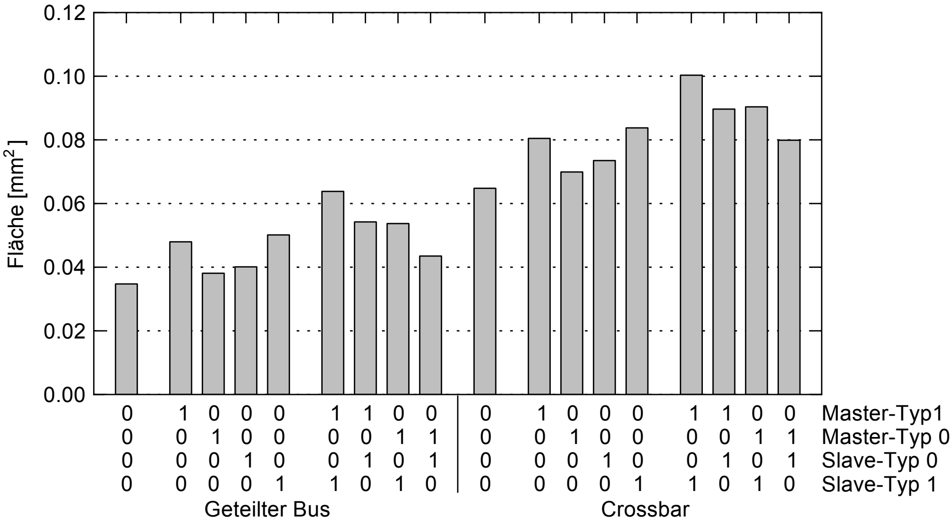


Abbildung 5.15: Flächenbedarf der AXI-Verbindungsstruktur von CPU-Clustern mit acht CPUs und verschiedenen Registerstufen sowie einer Frequenz von 400 MHz. Registerstufen sind entsprechend der Anzahl von Master-Typ 2-, Master-Typ 1-, Slave-Typ 1- bzw. Slave-Typ 2-Registern angegeben.

Bei den Konfigurationen mit einer Registerstufe weist eine Integration einer Registerstufe von Typ 2 zwischen den Mastern und der Verbindungsstruktur (1 0 0 0) die höchste Taktfrequenz auf (Bus: 781 MHz, Crossbar: 775 MHz). Eine Master-Registerstufe vom Typ 1 (0 1 0 0) ist 26 MHz bzw. 29 MHz langsamer, da das *Ready*-Signal hier im kritischen Pfad liegt. Die Verwendung von Slave-Registerstufen führt zu deutlich geringeren Taktfrequenzen, da ein kombinatorischer Pfad vom Master-Adressausgang eines CPU-Makros durch die Verbindungsstruktur zum *Ready*-Eingang des Masters eines anderen CPU-Makros verläuft.

Drei Konfigurationen mit zwei Registerstufen erreichen sowohl bei Verwendung des geteilten Bus als auch bei der Crossbar Taktfrequenzen von mehr als 820 MHz. Diese Frequenz ist 20 MHz höher als die maximale Taktfrequenz des CPU-Makros. Die Konfigurationen mit zwei Registern vom Typ 1 (0 1 1 0) erreichen lediglich 741 MHz (geteilter Bus) bzw. 746 MHz (Crossbar). Dies ist auf einen kombinatorischen Pfad von einem Slave zu einem Master (*Ready*-Signal) zurückzuführen.

Die Synthesen zur Bestimmung des Flächenbedarfs werden mit einer Taktfrequenz von 400 MHz durchgeführt, da der CPU-Cluster ohne Registerstufen und mit geteiltem Bus eine maximale Taktfrequenz von lediglich 465 MHz aufweist. In Abbildung 5.15 ist der Flächenbedarf der AXI-Verbindungsstruktur inklusive Registerstufen dargestellt.

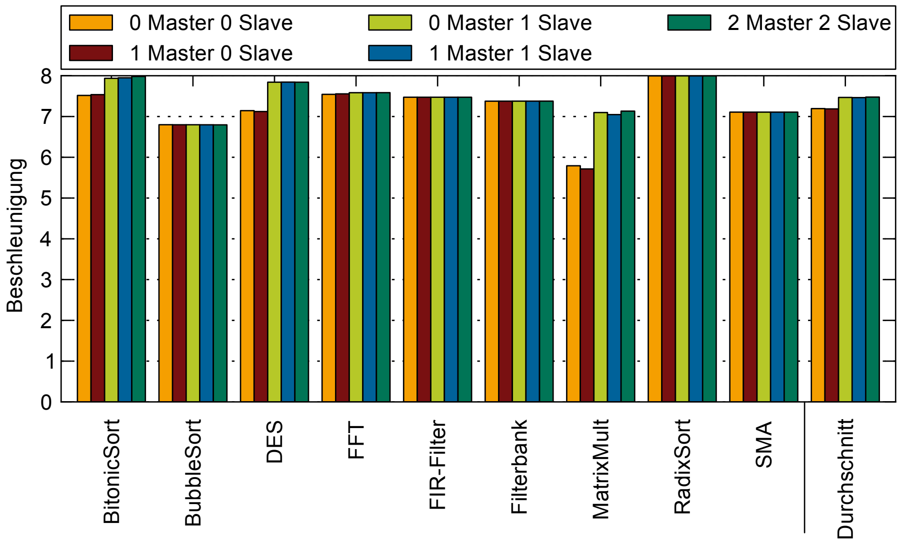


Abbildung 5.16: Beschleunigung von Streaming-Anwendungen mit acht CPUs und verschiedenen Master- und Slave-Registerstufen vom Typ 2 gegenüber der Ausführung auf einer CPU

Eine Aufschlüsselung der Fläche der Einzelkomponenten ist nicht möglich, da die HDL-Hierarchien von der Synthese-Software aufgebrochen werden müssen, um die gesamte Verbindungsstruktur effizient auf die 28-nm-FD-SOI-Technologie abbilden zu können. Die Fläche der Verbindungsstruktur ohne Registerstufen beträgt 0,037 mm^2 (Geteilter Bus) bzw. 0,066 mm^2 (Crossbar). Die Gesamtfläche des CPU-Clusters inklusive CPU-Makros beträgt 0,936 mm^2 bzw. 0,965 mm^2.

Sowohl bei Verwendung der Crossbar als auch beim geteilten Bus benötigen Register vom Typ 2 deutlich mehr Fläche als Typ 1-Register. Ein geteilter Bus mit Typ 2-Master-Register (1 0 0 0) ist beispielsweise 25,9 % größer als die gleiche Konfiguration mit Typ 1-Register (0 1 0 0). Dies ist auf die Verwendung von Doppelregistern zurückzuführen.

Zudem weisen die Slave-Registerstufen einen höheren Flächenbedarf im Vergleich zu den Master-Registern auf. Für einen geteilten Bus beträgt dieser Mehrbedarf 5,27 % für Typ 1-Register sowie 4,5 % für Typ 2-Register. Der Mehrbedarf ist durch interne Verzögerungszeiten der Ein- und Ausgänge eines CPU-Makros (*Setup* und *Clock-to-Output*) bedingt, die für den Slave-Port höher sind als für den Master-Port. Daher müssen bei den Slave-Registerstufen Transistoren mit höherer Treiberstärke verwendet werden. Der Flächenbedarf der Konfigurationen mit zwei Registerstufen ergibt sich aus der Summe der Fläche der verwendeten einzelnen Registerstufen.

Eine Integration von Registerstufen in die Verbindungsstruktur hat einen Einfluss auf die Performanz des CPU-Clusters bei der Ausführung von Streaming-Anwendungen. In Abbildung 5.16 ist die Beschleunigung von neun Streaming-Anwendungen bei der Ausführung auf CPU-Clustern mit verschiedenen Registerstufen-Konfigurationen dargestellt. Es werden acht CPUs, ein geteilter AXI-Bus sowie Registerstufen vom Typ 2 verwendet. Durch die Integration einer Master-Registerstufe sinkt die Performanz von MatrixMult um 1,3 %, da die Latenz von Buszugriffen ansteigt. Bei der Verwendung einer Slave-Registerstufe steigt die Performanz hingegen um 22,5 %, da diese Registerstufe Anfragen zwischenspeichern kann, wenn die Slave-Schnittstelle einer CPU blockiert ist. Der Grund für diese Blockierung sind Zugriffe der CPU auf ihren lokalen L1-Datenspeicher, die gegenüber externen Zugriffen über die Verbindungsstruktur bevorzugt behandelt werden. BitonicSort, DES sowie FFT profitieren ebenfalls von der Integration einer Slave-Registerstufe. Die Performanz von BubbleSort, FIR-Filter, Filterbank, RadixSort und SMA zeigt keine Abhängigkeit von der Anzahl und Art der Registerstufen, da diese Anwendungen im Vergleich den Bus nur gering auslasten. Im Durchschnitt der betrachteten Anwendungen ist der CPU-Cluster ohne Registerstufen 3,8 % langsamer als die Konfiguration mit je zwei Master- und Slave-Registerstufen. Bei der Verwendung von Typ 1-Registern unterscheidet sich die Performanz gegenüber Typ 2-Registern nur minimal aufgrund veränderter Zugriffsmuster der CPUs auf die Verbindungsstruktur.

In diesem Abschnitt konnte gezeigt werden, dass durch die Verwendung von Registerstufen die Taktfrequenz der Verbindungsstruktur des CPU-Clusters signifikant gesteigert werden kann. Registerstufen vom Typ 2 ermöglichen höhere Taktfrequenzen im Vergleich zum Typ 1, weisen jedoch einen deutlich höheren Flächenbedarf auf. Durch die Integration von Slave-Registerstufen kann die Performanz von Streaming-Anwendungen um bis zu 23,1 % gesteigert werden. Basierend auf diesen Ergebnissen wird im nächsten Abschnitt eine Entwurfsraumexploration durchgeführt, um für verschiedene Konfigurationen des CPU-Clusters den geringsten Flächenbedarf bei einer Taktfrequenz von 800 MHz zu ermitteln.

5.5.3 Bus-Standards und Topologien

In diesem Abschnitt werden Verbindungsstrukturen mit den Bus-Standards OpenCores Wishbone und ARM AMBA AXI [241] sowie verschiedenen Topologien betrachtet. Die Anzahl der CPUs pro Cluster wird von vier bis 32 variiert. Die Taktfrequenz des CPU-Clusters wird im Folgenden auf die maximale Taktfrequenz des CPU-Makros von 800 MHz festgelegt. Dies ist notwendig, da der Flächenbedarf einer Schaltung stark von der verwendeten Taktfrequenz abhängt (siehe Abschnitt 5.2.1). Um die geforderte Taktfrequenz zu ermöglichen, ist die Integration von Registerstufen in die Verbindungsstruktur notwendig. Es werden Synthesen eines CPU-Clusters ohne Verwendung von Registerstufen durchgeführt und deren Anzahl anschließend so lange erhöht, bis bei der Synthese die Ziel-Taktfrequenz von 800 MHz eingehalten wird. Registerstufen

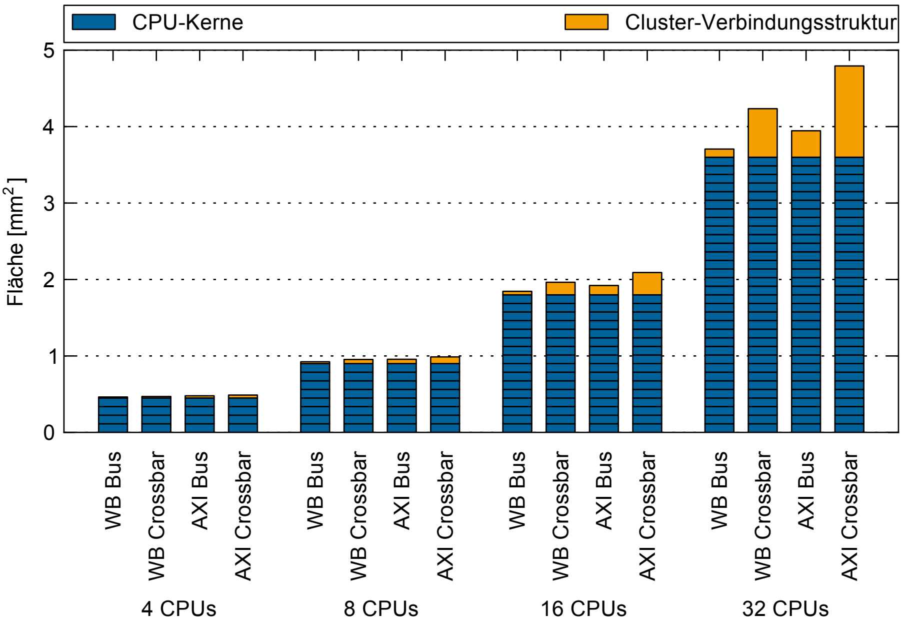


Abbildung 5.17: Flächenbedarf von CPU-Clustern mit vier bis 32 CPUs sowie verschiedenen Bus-Standards und Topologien

können hierbei sowohl zwischen den Bus-Mastern und der Verbindungsstruktur als auch zwischen der Verbindungsstruktur und den Slaves eingefügt werden. Es stehen Registerstufen vom Typ 1 und Typ 2 zur Verfügung (siehe Abschnitt 5.5.2). Erreichen mehrere Konfigurationen mit der gleich Registeranzahl die geforderte Taktfrequenz, wird die Konfiguration mit dem geringsten Flächenbedarf ausgewählt.

In Abbildung 5.17 ist der Flächenbedarf von CPU-Clustern mit vier bis 32 CPUs und verschiedenen Konfigurationen der Verbindungsstruktur dargestellt. Die Konfigurationen mit vier CPUs und Wishbone-Crossbar sowie die Konfigurationen mit vier bis 16 CPUs und geteiltem Wishbone-Bus verfügen über eine Master-Registerstufe vom Typ 2. Die übrigen Wishbone- sowie alle AXI-Konfigurationen benötigen jeweils eine Master- und eine Slave-Registerstufe, um die geforderte Taktfrequenz zu erreichen. Hierbei wird immer jeweils eine Stufe vom Typ 1 sowie eine Stufe vom Typ 2 verwendet.

Der CPU-Cluster mit vier CPUs und einem geteilten Wishbone-Bus hat einen Flächenbedarf von 0,46 mm², wobei die Verbindungsstruktur hiervon lediglich 2,8 % (0,013 mm²) in Anspruch nimmt. Die Verbindungsstrukturen mit Wishbone-Crossbar, geteiltem AXI-Bus sowie AXI-Crossbar benötigen jeweils 0,020 mm², 0,029 mm² und 0,038 mm². Die Konfigurationen mit acht CPUs und geteiltem Bus belegen mit 0,92 mm² (Wishbone) und 0,96 mm² (AXI) doppelt so viel Fläche wie die Cluster mit vier CPUs.

Der Verwendung von Wishbone- und AXI-Crossbar führt jedoch zu einem stärkeren Anstieg des Flächenbedarfs und einem deutlich größeren Anteil der Verbindungsstruktur an der Gesamtfläche.

Konfigurationen mit 16 CPUs besitzen einen Flächenbedarf von 1,85 mm^2 (geteilter Wishbone-Bus) bis 2,09 mm^2 (AXI-Crossbar). Die Wishbone-Crossbar ist um den Faktor 3,6 größer als der geteilte Wishbone-Bus. Die AXI-Crossbar benötigt 2,4-mal mehr Fläche als der geteilte AXI-Bus, bezogen auf die Gesamtfläche des CPU-Clusters bedeutet dies jedoch nur eine Steigerung um 8,8 %. Die Verbindungsstrukturen der beiden Konfigurationen mit 32 CPU-Kernen und geteiltem Bus benötigen 2,9 % (Wishbone) bzw. 8,8 % (AXI) der Gesamtfläche. Die Crossbar-Konfigurationen belegen mit 15,0 % (Wishbone) und 24,9 % (AXI) einen wesentlich größeren Anteil an der Gesamtfläche des CPU-Clusters.

Zusammenfassend kann festgestellt werden, dass der Flächenbedarf eines geteilten Busses nahezu linear mit der Anzahl an integrierten CPU-Kernen ansteigt. Die Crossbar-Konfigurationen zeigen einen quadratischen Anstieg. Dies ist durch die Verwendung von dedizierten Kommunikationskanälen zwischen jedem Master und jedem Slave bedingt. Dieser Zusammenhang ist auch im Modell für den Flächenbedarf des CoreVA-MPSoCs berücksichtigt (siehe Abschnitt 4.3).

Die AXI-Implementierung weist einen wesentlich höheren Flächenbedarf als der Wishbone-basierte CPU-Cluster auf. Dies ist darauf zurückzuführen, dass AXI fünf unidirektionale Kanäle besitzt (siehe Abschnitt 2.4). Bei einem geteilten Bus als Topologie integriert die AXI-Verbindungsstruktur fünfmal mehr Arbiter im Vergleich zur Wishbone-Implementierung. Wird eine volle Crossbar als Topologie verwendet, benötigt AXI zweimal mehr Arbiter als Wishbone. Ein Vorteil von AXI ist, dass paralleles Lesen und Schreiben innerhalb der Verbindungsstruktur möglich ist. Dies vereinfacht beispielsweise die Synchronisierung zwischen NoC-Cluster-Schnittstelle und den CPUs innerhalb des Cluster, da Verklemmungen mit deutlich weniger Aufwand ausgeschlossen werden können.

In Abbildung 5.18 ist der Flächenbedarf verschiedener Topologien eines AXI-basierten CPU-Clusters dargestellt. Neben den bereits in Abbildung 5.17 dargestellten Topologien geteilter Bus und volle Crossbar werden nun verschiedene partielle Crossbar-Topologien betrachtet. Eine partielle Crossbar ist eine Kombination aus geteilten Bussen und einer vollen Crossbar. Mehrere geteilte Busse sind über eine Crossbar miteinander verbunden. Ein Beispiel ist in Abbildung 2.8b in Abschnitt 2.4 dargestellt. Die Anzahl der CPUs wird von acht bis 32 variiert. Für CPU-Cluster mit 32 CPUs wird eine partielle Crossbar mit acht geteilten Bussen und vier CPUs pro Bus (8/4) sowie eine weitere Konfiguration mit 16 geteilten Bussen und zwei CPUs pro Bus (16/2) betrachtet. Jeder CPU-Cluster integriert eine Master- und eine Slave-Registerstufe vom Typ 2. Bei der Verwendung von 32 CPUs und einer geteilten 8/4-Crossbar steigt der Flächenbedarf des CPU-Clusters lediglich um 3,7 % im Vergleich zum geteiltem Bus an. Der CPU-Cluster mit einer 16/2-Crossbar besitzt einen um 10,7 % erhöhten Flächenbedarf, während der CPU-Cluster mit voller Crossbar im Vergleich zum geteiltem Bus 21,5 % größer ist. Bezogen auf

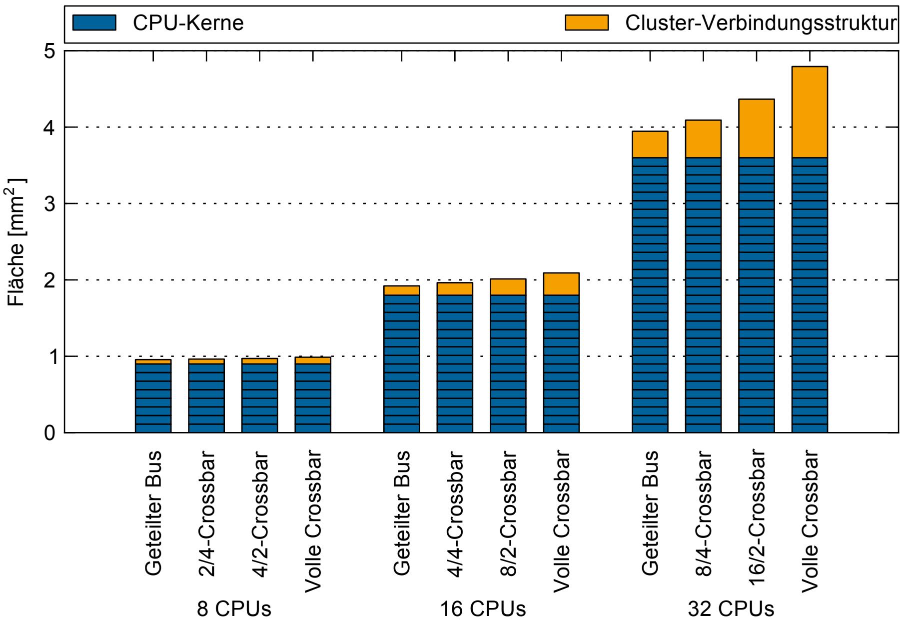


Abbildung 5.18: Flächenbedarf von CPU-Clustern mit acht bis 32 CPUs, AXI-Busstandard und geteiltem Bus sowie (partieller) Crossbar als Topologie

die Cluster-Verbindungsstruktur beträgt die Steigerung der Fläche jedoch 42,1 % (8/4-Crossbar), 121,4 % (16/2-Crossbar) bzw. 244,9 % (volle Crossbar).

CPU-Cluster mit 16 und acht CPUs weisen einen deutlich geringen Flächenzuwachs bei der Verwendung einer (partiellen) Crossbar auf. Die CPU-Cluster mit 16 CPUs besitzen einen Flächenbedarf von 1,92 mm^2 (geteilter Bus), 1,96 mm^2 (4/4-Crossbar), 2,01 mm^2 (8/2-Crossbar) sowie 2,09 mm^2 (volle Crossbar). Ein CPU-Cluster mit acht CPUs und 2/4-Crossbar ist lediglich 0,7 % größer als der Cluster mit geteiltem Bus. Bei Verwendung einer 4/2-Crossbar steigt der Flächenbedarf um 1,5 %, während der CPU-Cluster mit einer vollen Crossbar 3,3 % größer ist.

In Abbildung 5.19 ist die Performanz von Streaming-Anwendungen bei der Ausführung auf einem CPU-Cluster mit 16 CPUs sowie verschiedenen Bus-Topologien dargestellt. Die Performanz ist als Beschleunigung gegenüber der Ausführung auf einer CPU angegeben. Verbindungsstrukturen nach den Standards AXI und Wishbone zeigen im Rahmen von Messungenauigkeiten die gleiche Performanz. Die im Folgenden diskutierten Ergebnisse basieren auf einem CPU-Cluster mit AXI-Verbindungsstruktur sowie je einer Master- und Slave-Registerstufe vom Typ 2. Der CoreVA-MPSoC-Compiler kann bisher die Cluster-Verbindungsstruktur ausschließlich als vollständige Crossbar

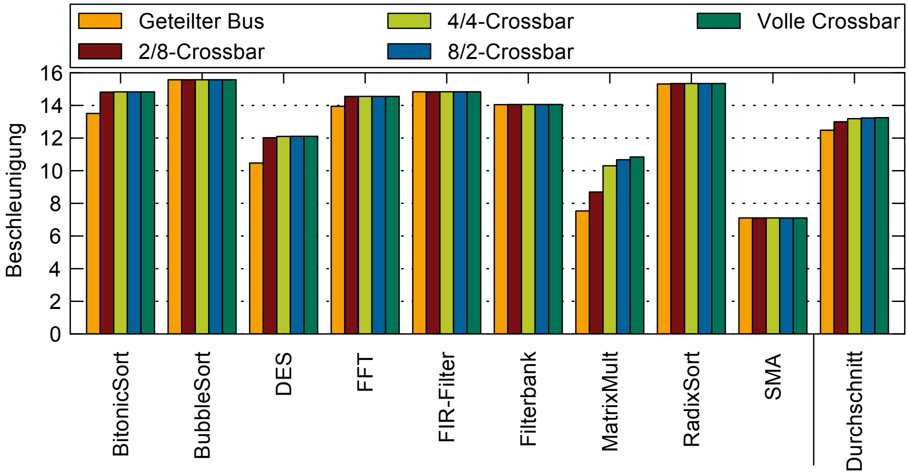


Abbildung 5.19: Beschleunigung von Streaming-Anwendungen mit 16 CPUs und verschiedenen Topologien der Cluster-Verbindungsstruktur gegenüber der Ausführung auf einer CPU

modellieren und optimiert hierauf die Partitionierung der Anwendung. In Zukunft ist es denkbar, dass der Compiler verschiedene Topologien berücksichtigt. Die hierzu notwendigen Informationen sind bereits in der abstrakten XML-Systembeschreibung des CoreVA-MPSoCs enthalten (siehe Abschnitt 4.4).

Die Anwendungen BubbleSort, FIR-Filter, Filterbank, RadixSort und SMA zeigen die gleiche Performanz für alle betrachteten Bus-Topologien. Dies ist darauf zurückzuführen, dass diese Anwendungen, im Vergleich zur Dauer der Berechnungen auf den einzelnen CPUs, wenig kommunizieren. Zudem speichert das FIFO der CPU-Cluster-Schnittstelle sowie die Registerstufen der Verbindungsstruktur Anfragen der jeweiligen CPU zwischen, falls die Verbindungsstruktur belegt ist (siehe Abschnitt 5.2.2). Die Anwendungen BitonicSort, DES, FFT und MatrixMult weisen bei der Verwendung eines geteilten Busses eine schlechtere Performanz im Vergleich zur vollen Crossbar auf. Mehrere CPUs greifen zeitgleich auf den geteilten Bus zu, der hierdurch eine hohe Auslastung aufweist. Aus diesem Grund ist das FIFO der entsprechenden CPU(s) oft vollständig gefüllt und die CPU muss angehalten werden. Der Bus ist hierbei nur zeitweise voll ausgelastet, z. B. wenn alle CPUs bei der Ausführung von BitonicSort einen Sortierschritt abgeschlossen haben und die Daten an eine andere CPU senden.

Die Anwendung MatrixMult zeigt mit 30,5 % den größten Verlust an Performanz bei der Verwendung eines gemeinsamen Busses. Eine partielle Crossbar mit zwei geteilten Bussen und acht CPUs pro Bus (2/8) ist 19,8 % langsamer als eine volle Crossbar. Die Konfigurationen mit vier (4/4) und zwei (8/2) CPUs pro Bus zeigen 5,0 % und 1,6 %

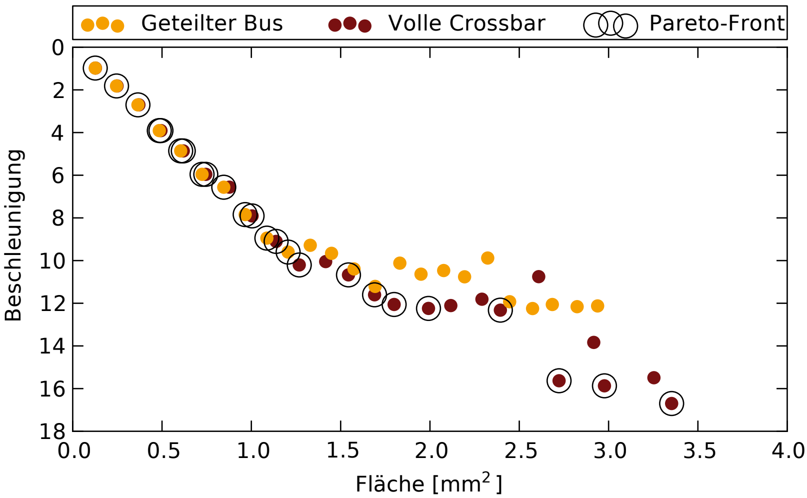


Abbildung 5.20: Beschleunigung und Flächenbedarf von CPU-Cluster mit einer bis 24 CPUs bei der Ausführung der Anwendung DES. Die Datenpunkte sind nach der Topologie der Verbindungsstruktur eingefärbt. Die Punkte der Pareto-Front sind mit einem Kreis markiert.

weniger Performanz. DES ist bei der Verwendung eines gemeinsamen Busses 13,5 % und bei der 2/8-Crossbar 0,7 % langsamer gegenüber der Verwendung einer vollen Crossbar. Alle übrigen Anwendungen zeigen keine geringere Performanz bei der Verwendung einer partiellen Crossbar. Im Durchschnitt aller betrachteten Anwendungen ist der geteilte Bus 5,8 %, die 2/8-Crossbar 1,9 % sowie die 4/4-Crossbar 0,5 % langsamer als die vollständige Crossbar.

Im CoreVA-MPSoC werden mehrere CPU-Cluster über ein On-Chip-Netzwerk verbunden [214; 223]. Jeder Cluster integriert hierzu eine Netzwerkschnittstelle. Es ist davon auszugehen, dass durch die Verwendung dieser Schnittstelle die Auslastung der Cluster-Verbindungsstruktur deutlich ansteigt.

Im Folgenden wird die Performanz und die Chipfläche verschiedener CPU-Cluster gemeinsam betrachtet. In Abbildung 5.20 ist die Beschleunigung der Anwendung DES sowie der Flächenbedarf von CPU-Clustern mit einer bis 24 CPUs dargestellt. Als Topologie wird ein geteilter Bus sowie eine volle Crossbar verwendet. Der Flächenbedarf steigt von 0,13 mm^2 (eine CPU, geteilter Bus) bis 3,35 mm^2 (24 CPUs, Crossbar) an. Die Performanz (Beschleunigung) von DES erhöht sich bei der Verwendung von 24 CPUs um den Faktor 16,7. Insgesamt 24 der 48 betrachteten Konfigurationen sind Teil der Pareto-Front (siehe Abschnitt 5.4), diese Konfigurationen sind in Abbildung 5.20 mit einem

Kreis markiert. Bei den CPU-Clustern mit einer bis drei sowie sieben CPUs sind nur die Konfigurationen mit geteiltem Bus Teil der Pareto-Front. Beide Topologien (geteilter Bus, Crossbar) sind jeweils bei den Konfigurationen mit vier bis sechs sowie acht bis zehn CPUs Teil der Pareto-Front. Die Konfigurationen der Pareto-Front mit mehr als zehn CPUs verfügen ausschließlich über eine Crossbar-Topologie. Hierbei ist jedoch nicht jede Crossbar-Konfiguration Teil der Pareto-Front, da Ungenauigkeiten in der Abschätzung des CoreVA-MPSoC-Compilers (siehe Abschnitt 3.2.3 sowie Abschnitt 4.2.4) zu einer suboptimalen Partitionierung und somit zu einer verminderten Performanz dieser Konfigurationen führen. Dies ist beispielsweise bei den Konfigurationen mit 19, 21 sowie 23 CPUs der Fall. Die Konfigurationen mit zehn und weniger CPUs zeigen eine stetige Zunahme der Performanz. Die Crossbar-Konfigurationen mit neun und zehn CPUs weisen sogar eine superskalare Beschleunigung von 9,1 bzw. 10,2 gegenüber der Ausführung auf einer CPU auf (siehe Abschnitt 5.5.1). Bei der Verwendung von mehr als zehn CPUs beschränkt das Aufteilen von Daten auf verschiedene CPUs zunehmend die Performanz. Zudem treten vermehrt Konflikte bei Zugriffen auf den lokalen L1-Speicher einzelner CPUs auf, wenn mehrere CPUs zeitgleich über die Cluster-Verbindungsstruktur sowie die lokale CPU auf diesen Speicher zugreifen.

In diesem Abschnitt wurde die Performanz sowie der Flächenbedarf verschiedener Konfigurationen der Verbindungsstruktur eines CPU-Clusters untersucht. Eine AXI-kompatible Implementierung der Verbindungsstruktur weist einen höheren Flächenbedarf im Vergleich zu einer Wishbone-Verbindungsstruktur auf, verspricht jedoch eine deutlich höhere Performanz bei der Anbindung eines CPU-Clusters an das On-Chip-Netzwerk des CoreVA-MPSoCs [214]. Ein geteilter Bus weist im Vergleich verschiedener Topologien den geringsten Flächenbedarf, jedoch auch die geringste Performanz auf. Durch die Verwendung einer partiellen oder vollständigen Crossbar ist die Performanz einzelner Anwendungen um bis zu 43,9 % steigerbar. Der Flächenbedarf einer vollständigen Crossbar-Verbindungsstruktur steigt jedoch bei der Verwendung von mehr als 16 CPUs stark an. Zusammenfassend kann festgestellt werden, dass für jede Anwendung eine andere Cluster-Konfiguration die höchste Ressourceneffizienz aufweist und daher eine anwendungsspezifische Entwurfsraumexploration der Verbindungsstruktur eines CPU-Clusters sinnvoll ist.

5.5.4 Gemeinsamer Level-2-Speicher

In diesem Abschnitt wird die Integration eines gemeinsamen Level-2-Speichers in den CPU-Cluster des CoreVA-MPSoCs untersucht. Grundlagen zu L2-Speicher werden in Abschnitt 2.3.3 beschrieben. Der L2-Speicher wird als Bus-Slave in den CPU-Cluster integriert und kann über mehrere Speicherbänke verfügen, um parallelen Zugriff mehrerer CPUs auf unterschiedliche Bänke zu ermöglichen. In diesem Fall ist allerdings die Verwendung einer partiellen oder vollen Crossbar als Cluster-Verbindungsstruktur notwendig. Ein geteilter Bus würde pro Taktzyklus nur einem Master Zugriff auf die

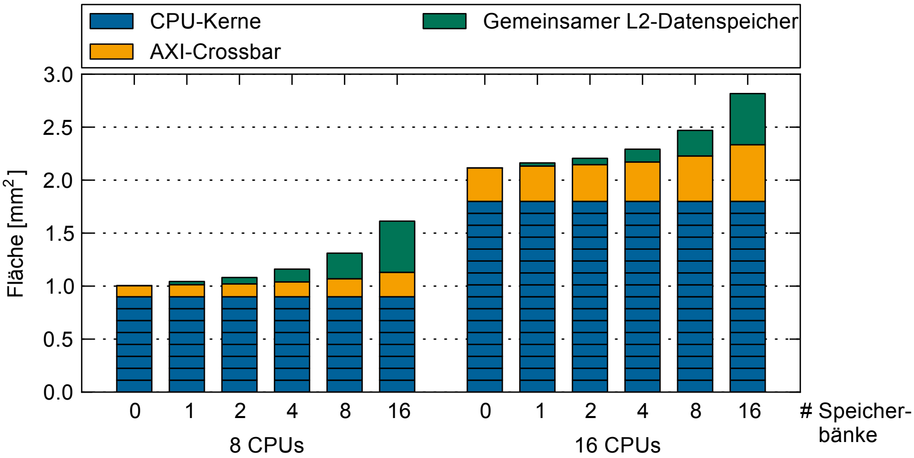


Abbildung 5.21: Flächenbedarf von CPU-Clustern mit gemeinsamem L2-Speicher, bis zu 16 Speicherbänken, 16 kB pro Bank, acht und 16 CPUs sowie AXI-Crossbar bei einer Frequenz von 800 MHz

Speicherbänke erlauben[8]. Im CPU-Cluster des CoreVA-MPSoCs ist, beginnend ab der Adresse 0x10000000, ein Adressbereich von 128 MB für den gemeinsamen L2-Speicher reserviert.

Für die Bestimmung des Flächenbedarfs verschiedener L2-Konfigurationen wird ein CPU-Cluster mit acht und 16 CPUs, einer vollen AXI-Crossbar sowie jeweils einer Master- und Slave-Registerstufe vom Typ 2 verwendet. Die Latenz eines Lesezugriffs auf den L2-Speicher beträgt bei dieser Cluster-Konfiguration sieben Takte (siehe Abschnitt 5.3).

In Abbildung 5.21 ist der Flächenbedarf bei der Verwendung von bis zu 16 L2-Speicherbänken dargestellt. Jede Speicherbank besitzt eine Größe von 16 kB. Zudem enthält Abbildung 5.21 eine Konfiguration ohne gemeinsamen L2-Speicher. Die Taktfrequenz beträgt 800 MHz. Bei den betrachteten Konfigurationen hat die Anzahl der L2-Speicherbänke keinen Einfluss auf die maximale Taktfrequenz des CPU-Clusters, wenn zwei Registerstufen verwendet werden. Jede Speicherbank besitzt einen Flächenbedarf von 0,031 mm^2. Bei der Integration von 16 Speicherbänken ist der Flächenbedarf entsprechend 0,496 mm^2. Da mit zunehmender Anzahl an Speicherbänken jedoch auch die Anzahl der Bus-Slaves steigt, vergrößert sich der Flächenbedarf der Verbindungsstruktur bei der Verwendung von acht CPUs von 0,104 mm^2 (null Bänke) auf bis zu 0,230 mm^2 (16 Bänke). Dies entspricht einer Steigerung um über 120 %. Der gemeinsame L2-Speicher belegt bei der Integration von 16 Bänken 29,9 % der Gesamtfläche des CPU-Clusters.

[8]Bei Verwendung einer geteilten AXI-Verbindungsstruktur kann eine CPU schreibend und eine CPU lesend auf den Speicher zugreifen

Konfigurationen mit 16 CPUs besitzen einen Flächenbedarf von 2,12 mm^2 (0 Speicherbänke) bis 2,82 mm^2 (16 Speicherbänke) für den gesamten CPU-Cluster. Der Anteil der AXI-Verbindungsstruktur an der Gesamtfläche beträgt 14,9 % (0,316 mm^2, 0 Bänke) bis 19,0 % (0,534 mm^2, 16 Bänke). Durch die Integration von 16 L2-Bänken steigt der Flächenbedarf der Verbindungsstruktur also um 0,218 mm^2 an. Hauptgrund für diese Zunahme sind die zusätzlichen Arbiter pro Slave, welche die Zugriffe der Master arbitrieren sowie die Registerstufe zwischen Verbindungsstruktur und den L2-Speicherbänken.

Der Flächenbedarf des eigentlichen L2-Speichers wird maßgeblich durch die verwendeten SRAM-Speicherblöcke bestimmt. Insbesondere bei der Integration von vielen Speicherbänken steigt der Flächenbedarf der Cluster-Verbindungsstruktur stark an. Zudem weisen Lesezugriffe der CPUs auf den L2-Speicher eine hohe Latenz auf (siehe Abschnitt 5.3). Aus diesen Gründen wird im nächsten Abschnitt die Integration eines gemeinsamen L1-Datenspeichers in das CoreVA-MPSoC untersucht.

5.6 Eng gekoppelter gemeinsamer L1-Datenspeicher

In diesem Abschnitt wird der CPU-Cluster des CoreVA-MPSoCs um einen eng gekoppelten gemeinsamen L1-Datenspeicher erweitert (siehe Abschnitt 2.3.2). Es wird untersucht, wie sich die Integration dieses Speichers auf die Ausführungszeit von Streaming-Anwendungen und den Ressourcenbedarf des MPSoCs auswirkt. Der eng gekoppelte gemeinsame L1-Datenspeicher erlaubt den CPUs des Clusters gleichberechtigten Zugriff auf den Speicher [231]. Gleichzeitig ist die Latenz von Schreib- und Lesezugriffen sowie die Speicherbandbreite im besten Fall identisch zu lokalen L1-Datenspeichern der CPUs. Um die Wahrscheinlichkeit von Zugriffskonflikten zu senken, besitzt der gemeinsame L1-Datenspeicher mehrere Speicherbänke. Die Ergebnisse dieses Abschnitts wurden in [225] veröffentlicht.

Die maximale Taktfrequenz für verschiedene Konfigurationen des gemeinsamen L1-Datenspeichers ist in Abbildung 5.22 dargestellt. Der CPU-Cluster besteht aus acht CPU-Makros mit zwei VLIW-Slots und 16 kB lokalem Instruktionsspeicher. Ein lokaler L1-Datenspeicher wird nicht verwendet. Jede Speicherbank des gemeinsamen L1-Datenspeichers besitzt eine Größe von 16 kB. Die Anzahl an Speicherbänken wird von zwei bis 128 variiert. Zudem wird ein geteilter AXI-Bus für die Initialisierung und Steuerung der CPUs verwendet. Durch die Integration von zwei Registerstufen innerhalb der AXI-Verbindungsstruktur beträgt die maximale Taktfrequenz ohne gemeinsamen L1-Datenspeicher 800 MHz.

Bei Konfigurationen mit gemeinsamem L1-Datenspeicher, MoT-Verbindungsstruktur und zwei bis acht Speicherbänken wird die maximale Taktfrequenz durch den AXI-Bus beschränkt. Der kritische Pfad aller anderen betrachteter Konfigurationen führt durch die L1-Speicher-Verbindungsstruktur. Ausgehend von der Schnittstelle des CPU-Makros verläuft der kritische Pfad durch die Speicher-Verbindungsstruktur bis zum

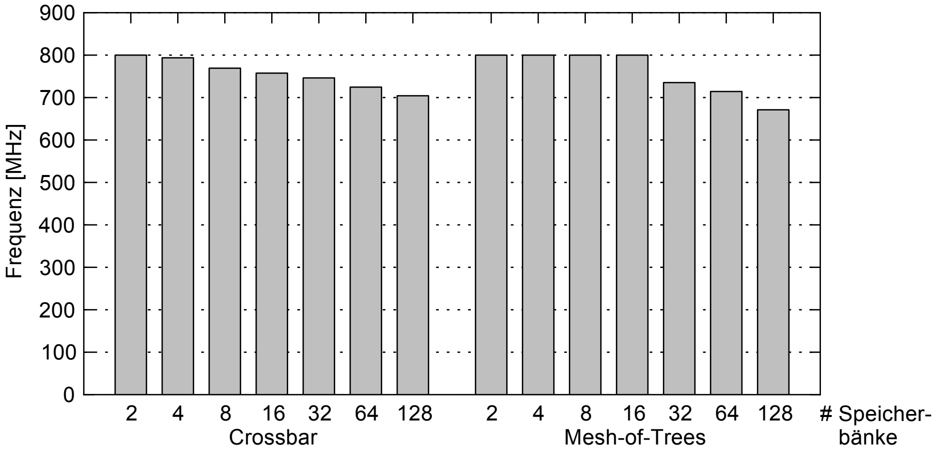


Abbildung 5.22: Maximale Taktfrequenz von CPU-Clustern mit verschiedenen Speicher-Verbindungsstrukturen, Anzahl an Speicherbänken und 16 kB pro Bank

letzten Arbitrierungsknoten und zurück zu einem CPU-Makro. Die Speichermakros des L1-Datenspeichers sind nicht Bestandteil des kritischen Pfades. Aus diesem Grund können flächeneffiziente *High-Density*-Speicher verwendet werden (siehe Abschnitt 3.1). CPU-Cluster mit MoT-Verbindungsstruktur und 32 bis 128 Bänken erreichen lediglich 735 MHz bis 671 MHz. Die Konfiguration mit 32 Bänken ist also 65 MHz langsamer als die Konfiguration mit 16 Bänken. Dieser große Unterschied kann durch verschiedene Optimierungsstrategien des Synthesewerkzeugs erklärt werden. Das Synthesewerkzeug bricht die HDL-Hierarchie der Verbindungsstruktur auf, um Optimierungen vornehmen zu können. Wird dies dem Werkzeug verboten, tritt der große Taktfrequenz-Unterschied zwischen 16 und 32 Bänken nicht auf, die maximalen Taktfrequenzen sind jedoch wesentlich geringer. Crossbar-Konfigurationen zeigen einen gleichmäßigen Abfall der maximalen Taktfrequenz von 800 MHz (zwei Bänke) bis 704 MHz (128 Bänke). Für zwei bis 16 Bänke zeigen die MoT-Konfigurationen eine höhere maximale Taktfrequenz, während für mehr als 16 Bänke die Crossbar höhere Frequenzen als MoT erlaubt. CPU-Cluster mit zwei, vier und 16 CPU-Kernen zeigen ebenfalls einen Vorteil der Crossbar-Konfigurationen für eine hohe Anzahl an Bänken.

In Abbildung 5.23 ist der Flächenbedarf eines CPU-Clusters mit acht CPUs, Crossbar-Speicher-Verbindungsstruktur und verschiedenen Konfigurationen des geteilten L1-Datenspeichers dargestellt. Die Summe der Größe aller Datenspeicher im Cluster beträgt bei allen Konfigurationen 256 kB. Die Taktfrequenz ist auf 650 MHz festgelegt, um einen fairen Vergleich aller Konfigurationen zu ermöglichen. Der Flächenbedarf der MoT-Verbindungsstruktur unterscheidet sich bei identischen Parametern nicht signifikant vom Flächenbedarf der Crossbar. Die *Shared-Memory*-Konfiguration verwendet

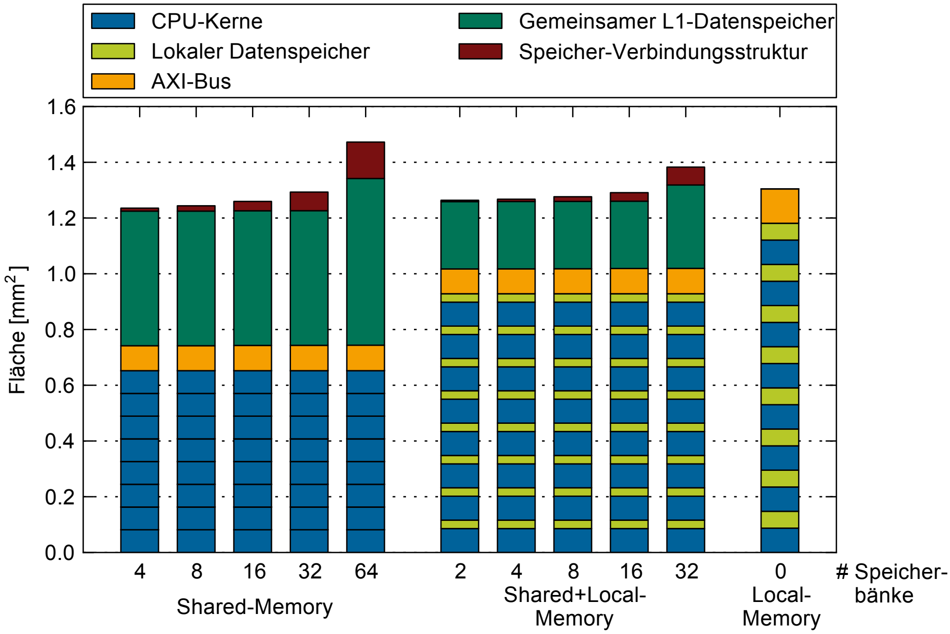


Abbildung 5.23: Flächenbedarf von CPU-Clustern mit 256 KB privatem und/oder gemeinsamem L1-Datenspeicher, Crossbar-Verbindungsstruktur und zwei bis 64 Speicherbänken bei einer Frequenz von 650 MHz

CPU-Makros ohne lokalen L1-Datenspeicher (siehe Abschnitt 5.2.3) sowie einen gemeinsamen L1-Datenspeicher der Größe 256 kB auf Cluster-Ebene. Dieser besteht aus vier bis 64 Speicherbänken. Die acht CPU-Makros haben zusammen einen Flächenbedarf von 0,77 mm^2. Die Konfigurationen mit vier bis 32 Speicherbänken sind aus 8-kB-Speichermakros aufgebaut und besitzen eine Größe von 0,48 mm^2. Die Konfiguration mit 64 Speicherbänken verwendet 4-kB-Speichermakros und ist 0,56 mm^2 groß. Die Fläche für die Speicher-Verbindungsstruktur liegt zwischen 0,011 mm^2 (vier Bänke) und 0,128 mm^2 (64 Bänke). Auf die gesamte Fläche bezogen ist die Konfiguration mit 64 Speicherbänken 17 % größer als die Konfiguration mit vier Bänken.

Die *Shared+Local-Memory*-Konfiguration besitzt pro CPU-Makro 16 kB lokalen L1-Datenspeicher sowie 128 kB gemeinsamen L1-Datenspeicher mit zwei bis 32 Bänken. Mit insgesamt 128 kB lokalen L1-Datenspeicher benötigen die CPU-Makros eine Fläche von 1,06 mm^2. Die Speichermakros des geteilten L1-Speichers besitzen eine Fläche von 0,241 mm^2 (vier bis 16 Bänke) bzw. 0,299 mm^2 (32 Bänke). Die Größe der Speicher-Verbindungsstruktur ist etwas geringer verglichen mit der Shared-Memory-Konfiguration, da die Speicherbänke nur die halbe Größe besitzen. Kleinere Speicher weisen eine geringere Latenz auf. Aus diesem Grund können bei der Synthese der Verbindungsstruktur Transistoren mit einer geringeren Treiberstärke verwenden werden.

Die *Local-Memory*-Konfiguration besitzt 32 kB lokalen L1-Datenspeicher pro CPU und integriert keinen geteilten L1-Datenspeicher auf Cluster-Ebene. Die Konfiguration verwendet eine AXI-Verbindungsstruktur mit Crossbar-Topologie, um eine parallele Kommunikation der CPUs zu ermöglichen. Der Flächenbedarf der Crossbar-Verbindungsstruktur beträgt 0,102 mm^2. Die Konfigurationen mit geteiltem L1-Speicher verwenden einen geteilten AXI-Bus (0,90 mm^2), da dieser hier nur für Initialisierung und Steuerung der CPUs verwendet wird. Dieser geringe Unterschied des Flächenbedarfs ist auf die zwei Registerstufen zurückzuführen, die in beide Verbindungsstrukturen integriert sind.

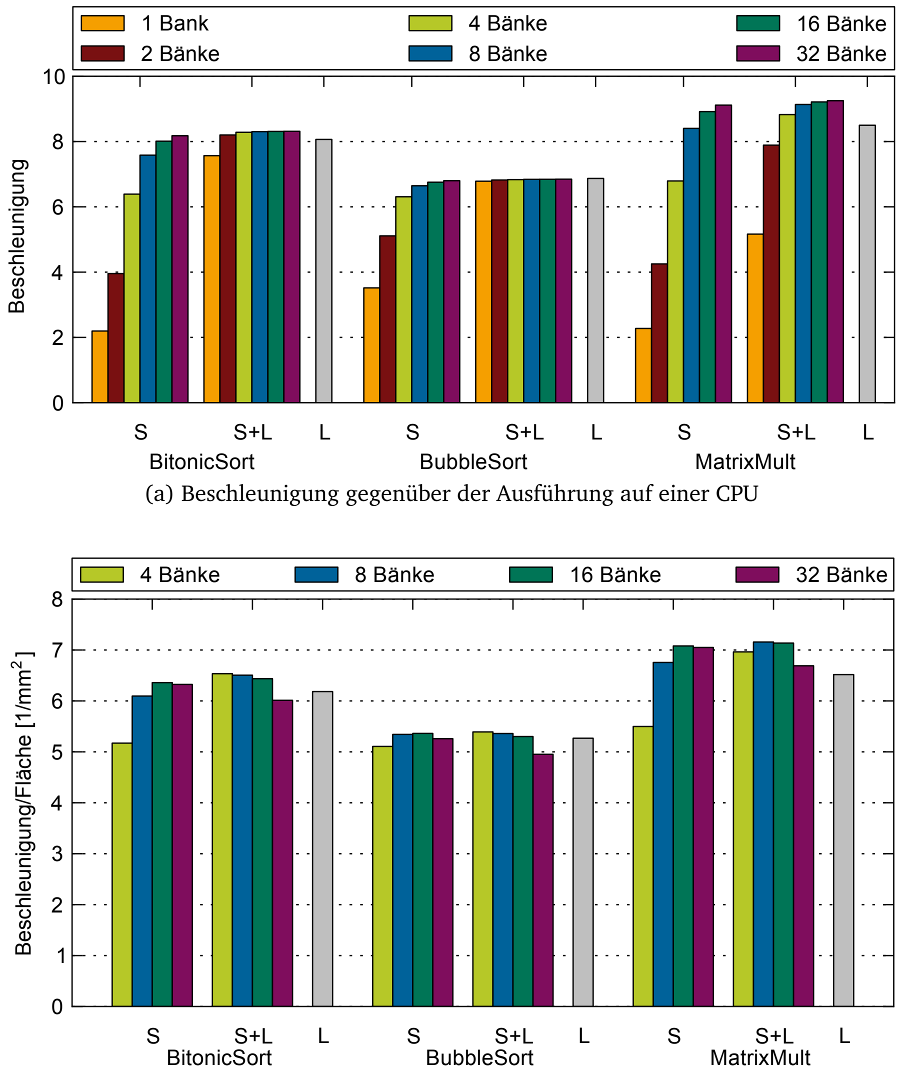
Die drei Konfigurationen *Shared-Memory*, *Shared+Local-Memory* und *Local-Memory* haben für eine geringe Anzahl an Speicherbänken einen vergleichbaren Flächenbedarf. Für eine hohe Anzahl an Speicherbänken (32 bis 64) zeigt die *Shared-Memory*-Konfiguration eine um bis zu 17 % größere Fläche. Die *Shared+Local-Memory*-Konfiguration zeigt ein bis zu 12 % erhöhte Fläche. Dies ist auf den steigenden Flächenbedarf der Speicher-Verbindungsstruktur bei einer zunehmenden Anzahl an Bänken zurückzuführen.

Im Folgenden wird die Ausführungszeit von Streaming-Anwendungen für die verschiedenen Speicher-Konfigurationen (*Shared-Memory*, *Shared+Local-Memory* und *Local-Memory*) verglichen (siehe Abbildung 5.24a). Hierzu wird ein CPU-Cluster mit acht CPUs betrachtet. Die Cluster-Konfigurationen werden mit identischer Software-Partitionierung verglichen, die mit dem CoreVA-MPSoC-Compiler erzeugt worden sind. Dies führt dazu, dass identische Speicher-Zugriffsmuster verwendet werden. Es werden die StreamIt-Anwendungen BitonicSort, BubbleSort und MatrixMult betrachtet.

Die *Local-Memory*-Konfiguration besitzt keinen geteilten L1-Datenspeicher und verwendet ausschließlich den lokalen L1-Datenspeicher der CPUs. Heap und Stack sind bei der *Shared+Local-Memory*-Konfiguration im lokalem Datenspeicher abgelegt, während die CPU-zu-CPU-Kommunikationskanäle (siehe Abschnitt 2.6.4) den gemeinsamen L1-Speicher verwenden. Die *Shared-Memory*-Konfiguration alloziert Heap, Stack sowie die Kommunikationskanäle im gemeinsamen L1-Datenspeicher.

22 % von allen ausgeführten Instruktionen der Anwendung BubbleSort sind Speicheroperationen (LD/ST), wobei 8 % Schreib- und 14 % Leseoperationen sind (siehe Anhang C). 1,3 % von allen Instruktionen werden für die CPU-zu-CPU-Kommunikation verwendet. Die Anwendung MatrixMult besitzt einen deutlich größeren Anteil an Speicheroperationen von 42 % (19 % schreiben, 23 % lesen). Der Anteil der CPU-zu-CPU-Kommunikation an allen Operationen beträgt 8 %. Von allen Instruktionen der BitonicSort-Anwendung sind 9 % Schreiboperationen sowie 22 % Leseoperationen. 1,8 % von allen Instruktionen werden für die Kommunikation im Cluster verwendet.

In Abbildung 5.24a ist die Beschleunigung von Streaming-Anwendungen für Konfigurationen mit einer bis 32 Speicherbänken und acht CPUs gegenüber der Ausführung auf einer einzelnen CoreVA-CPU dargestellt. Die Konfiguration mit einer CPU besitzt einen lokalen L1-Datenspeicher der Größe 32 kB. BitonicSort und MatrixMult zeigen die höchste Beschleunigung (8,31 bzw. 9,25) für die *Shared+Local-Memory*-Konfiguration



(a) Beschleunigung gegenüber der Ausführung auf einer CPU

(b) Beschleunigungs-Flächen-Verhältnis

Abbildung 5.24: Performanz von Streaming-Anwendungen und *Shared* (S), *Shared+Local* (S+L) und *Local-Memory* (L, graue Balken) Konfigurationen, eine bis 32 Speicherbänken und acht CPUs mit 32 Bänken. Die *Local-Memory*-Konfiguration ohne gemeinsamen L1-Datenspeicher ist 3,1 % bzw. 8,8 % langsamer. Dies ist auf die hohe Auslastung der lokalen L1-Speicher und der AXI-Verbindungsstruktur zurückzuführen. BubbleSort zeigt die höchste Beschleunigung von 6,87 bei der *Local-Memory*-Konfiguration. Die *Shared+Local-Memory*-Konfiguration mit 32 Speicherbänken erreicht eine Beschleunigung von 6,85, während bei der Verwendung von zwei Bänken eine Beschleunigung von 6,82 erreicht wird. Hieraus folgt, dass mehr als zwei bis vier Speicherbänke für BubbleSort und die *Shared+Local-Memory*-Konfiguration nicht benötigt werden.

Die Anzahl an Bankkonflikten wird durch die Verwendung von mehr Speicherbänken reduziert. Dies führt dazu, dass die Beschleunigung umso größer ist, je mehr Speicherbänke in den CPU-Cluster integriert werden. Die maximale Taktfrequenz des CPU-Clusters sinkt jedoch mit zunehmender Anzahl an Speicherbänken (siehe Abbildung 5.22), daher ist ein Kompromiss zwischen der Häufigkeit von Bankkonflikten und der maximalen Taktfrequenz notwendig. Die *Shared-Memory*-Konfiguration weist für die Anwendung BitonicSort bei der Verwendung einer Speicherbank eine durchschnittliche Speicherlatenz von 6,11 Takten auf. Dies ist auf eine hohe Anzahl an Bankkonflikten zurückzuführen. Die resultierende Beschleunigung liegt bei lediglich 2,20. Werden zwei bzw. vier Bänke verwendet, sinkt die durchschnittliche Latenz auf 3,83 bzw. 2,65 Takte. Die Konfiguration mit 32 Bänken zeigt eine mittlere Latenz von 2,05 Takten und eine Beschleunigung von 8,18.

Wird die gleiche Anzahl an Speicherbänken verwendet, zeigt die *Shared+Local-Memory*-Konfiguration immer eine höhere Beschleunigung als die *Shared-Memory*-Konfiguration. Dies ist darauf zurückzuführen, dass der gemeinsame Speicher bei der *Shared+Local-Memory*-Konfiguration ausschließlich für die CPU-zu-CPU-Kommunikation verwendet wird. Der lokale L1-Datenspeicher enthält den Heap und Stack und entlastet den gemeinsamen L1-Datenspeicher von bis zu 77 % aller Speicheroperationen

Im Folgenden wird die Anwendungsperformanz und der Flächenbedarf des gemeinsamen L1-Datenspeichers über das Verhältnis von Beschleunigung und Fläche bewertet (siehe Abschnitt 4.1). In Abbildung 5.24b ist das Beschleunigungs-Flächen-Verhältnis für Cluster-Konfigurationen mit vier bis 32 Speicherbänken dargestellt. Die *Shared+Local-Memory*-Konfiguration mit vier Bänken zeigt für die Anwendungen BitonicSort und BubbleSort das höchste Verhältnis von 6,53 (BitonicSort) bzw. 5,39 (BubbleSort). Die Anwendung MatrixMult zeigt das beste Verhältnis von 7,16 für die *Shared+Local-Memory*-Konfiguration mit acht Speicherbänken. Die *Shared-Memory*-Konfigurationen mit vier Bänken zeigen für alle betrachteten Anwendungen ein schlechteres Verhältnis, da hier sehr viele Bankkonflikte auftreten. Dies führt zu einer reduzierten Beschleunigung gegenüber Konfigurationen mit mehr Bänken. Bei der Verwendung von 32 Speicherbänken steigt der Flächenbedarf der Speicher-Verbindungsstruktur stark an. Konfigurationen mit 32 Bänken und *Shared+Local-Memory* weisen zudem einen hohen Flächenbedarf aufgrund der verwendeten 4-kB-Speichermakros auf. Dies führt zu einem schlechteren Beschleunigungs-Flächen-Verhältnis im Vergleich mit den Konfigurationen mit acht und 16 Speicherbänken. Die *Local-Memory*-Konfigurationen zeigt im Vergleich für alle drei betrachteten Anwendungen eine schlechteres Verhältnis aus Flächenbedarf und Performanz. Dies ist auf den hohen Flächenbedarf der AXI-Crossbar sowie auf eine verringerte Performanz aufgrund einer hohen Auslastung der lokalen Datenspeicher der CPUs zurückzuführen. Durch die Verwendung einer partielle Crossbar als Cluster-Verbindungsstruktur kann der Flächenbedarf verringert werden. Die Performanz kann durch eine Integration von zwei oder mehr Speicherbänke in den lokalen Datenspeicher der CPUs gesteigert werden.

In diesem Abschnitt konnte gezeigt werden, dass die Leistungsfähigkeit des CPU-Clusters durch die Integration eines gemeinsamen L1-Datenspeichers gesteigert werden kann. Eine *Mesh-of-Trees*-Verbindungsstruktur weist bei einer geringen Anzahl an Speicherbänken eine höhere maximale Taktfrequenz als eine Crossbar auf, während eine Crossbar bei 32 oder mehr Bänken höhere Taktfrequenzen ermöglicht. Drei der in Abschnitt 6.2 vorgestellten Chip-Layouts des CPU-Cluster integrieren 32 kB bis 128 kB gemeinsamen L1-Datenspeicher.

5.7 Ein L1-Instruktionscache für die Verwendung im CoreVA-MPSoC

Bei den bisher vorgestellten Untersuchungen des CoreVA-MPSoCs hat jede CPU einen lokalen L1-Scratchpad-Speicher für Instruktionen verwendet, in dem das Programmabbild der jeweiligen CPU abgelegt ist. Dieser Instruktionsspeicher kann jede Anfrage der CPU innerhalb eines Taktes beantworten, ist jedoch – vergleichbar zum lokalen L1-Datenspeicher – in seiner Größe beschränkt (siehe Abschnitt 5.2). Zudem führen verschiedene CPUs im Cluster oft die gleichen Programmteile einer Anwendung aus. Hierdurch ist der gleiche Programmcode mehrfach in verschiedenen Speichern eines Clusters vorhanden. Ein L1-Instruktionscache ersetzt den lokalen Instruktionsspeicher der CPUs und speichert Instruktionen zwischen (siehe Abschnitt 2.3.1). Das Programmabbild der Anwendung wird im gemeinsamen L2-Speicher abgelegt (siehe Abschnitt 5.5.4). Fragt die CPU Instruktionen an, die nicht im Cache gespeichert sind, lädt dieser die entsprechenden Daten aus dem L2-Speicher nach. Hierzu ist der Instruktionscache an die Bus-Master-Schnittstelle der CPU angebunden. Bis der Cache die angefragten Instruktionen der CPU übergeben kann, muss diese angehalten werden. Die Daten werden im Datenspeicher des Caches in einer Cachezeile zwischengespeichert, um zukünftige Anfragen der CPU auf diese Adresse ohne Verzögerungen beantworten zu können. Zudem müssen Adress- und Verwaltungsinformationen zur Cachezeile in einem Tagspeicher abgelegt werden. Weitere Details zum Aufbau und zur Funktionsweise des Instruktionscaches sind in den Abschlussarbeiten [174] und [239] zu finden.

Der Flächenbedarf einer CoreVA-CPU mit zwei VLIW-Slots und verschiedenen Konfigurationen des Instruktionscaches ist in Abbildung 5.25 dargestellt. Es wird ein direkt abgebildeter Cache verwendet. Pro Cachezeile werden 16 64-bit-Wörter (128 B) gespei-

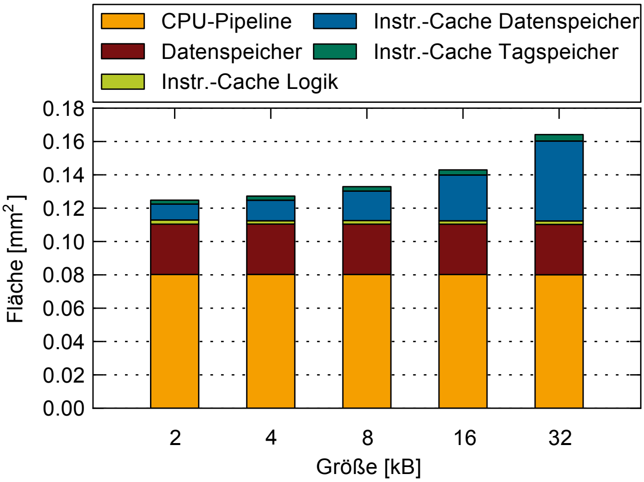


Abbildung 5.25: Flächenbedarf einer CoreVA-CPU mit Instruktionscache und verschiedenen Größen des Instruktionscache-Datenspeichers bei einer Taktfrequenz von 700 MHz

chert, wobei die Anzahl der Cachezeilen zwischen 16 und 256 variiert wird. Hieraus folgt, dass der Datenspeicher des Caches 2 kB bis 32 kB groß ist. Ein Cache mit 16 Cachezeilen besitzt einen Flächenbedarf von 0,014 mm^2, was 11,5 % der Gesamtfläche der CPU (0,125 mm^2) entspricht. Der Datenspeicher des Caches nimmt hiervon mit 0,010 mm^2 einen Großteil der Fläche ein. Bei der Integration von mehr Cachezeilen steigt der Flächenbedarf insbesondere der Daten- und Tagspeicher an. Da die Fläche der Speicher nicht proportional mit der Anzahl der Einträge wächst, benötigt der Datenspeicher des Caches mit 256 Zeilen lediglich um den Faktor 5 mehr Fläche, obwohl die Kapazität 16-mal so groß ist. Dies führt zu einer Fläche des Instruktionscaches von 0,054 mm^2 bei einer Gesamtfläche der CPU von 0,164 mm^2. Die Chipfläche für die Steuerlogik des Caches verändert sich bei den verschiedenen Konfigurationen nur im Rahmen der Mess- bzw. Synthese-Ungenauigkeiten und beträgt ca. 0,002 mm^2. Der Mehrbedarf an Chipfläche eines Instruktionscaches mit 16 kB Datenspeicher gegenüber einem Scratchpad-Speicher gleicher Größe beträgt 0,009 mm^2 bzw. 6,4 % der Gesamtfläche der CPU.

Die CPU ist für eine Taktfrequenz von 700 MHz synthetisiert, da Konfigurationen mit 128 und 256 Cachezeilen (16 kB und 32 kB) die maximale Taktfrequenz der CPU auf 735 MHz bzw. 746 MHz beschränken. Dies ist auf den vergrößerten Tagspeicher des Caches zurückzuführen, der bei diesen Konfigurationen im kritischen Pfad der CPU liegt. Um die CPU mit einer maximalen Taktfrequenz von 800 MHz betreiben zu

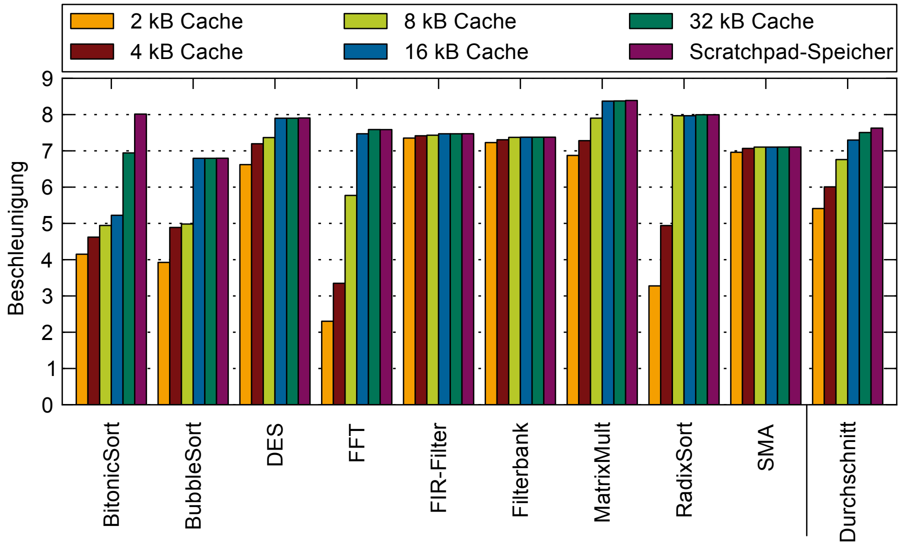


Abbildung 5.26: Beschleunigung von Streaming-Anwendungen mit acht CPUs und verschiedenen Konfigurationen von Scratchpad-Speicher und Caches für Instruktionen

können, muss ein Instruktionscache mit maximal 64 Cachezeilen verwendet werden. Alternativ ist der Einsatz von Speicherblöcken mit geringeren Verzögerungszeiten (*High-Speed*-Speicher) möglich (siehe Abschnitt 3.1). Dies würde jedoch zu einem erhöhten Flächen- und Energiebedarf im Vergleich zum verwendeten flächeneffizienten *High-Density*-Speicher führen.

In Abbildung 5.26 ist die Beschleunigung eines CPU-Clusters mit acht CPUs und verschiedenen Instruktionscache-Konfigurationen gegenüber der Ausführung auf einer CPU mit L1-Scratchpad-Speicher dargestellt. Zudem ist als Referenz eine Konfiguration mit acht CPUs und L1-Scratchpad-Speicher aufgeführt, da hier die CPUs die Instruktionen ohne Wartetakte laden können. Als Cluster-Verbindungsstruktur wird eine AXI-Crossbar verwendet, die auch für CPU-zu-CPU-Kommunikation genutzt wird. Der L2-Speicher besteht aus mehreren Speicherbänken, um Zugriffskonflikte durch parallele Zugriffe der CPUs zu vermeiden. Jede Cachezeile des Instruktionscaches ist 128 B (16 64-bit-Wörter) groß. Wie bei den Untersuchungen zum Flächenbedarf wird die Anzahl der Cachezeilen von 16 bis 256 variiert. Hieraus folgt, dass die Datenspeicher der betrachteten Caches eine Größe zwischen 2 kB und 32 kB besitzen (pro CPU).

Im Vergleich zum Scratchpad-Speicher zeigen die Cache-Konfigurationen im Durchschnitt eine um 1,6 % (32 kB) bis 29,1 % (2 kB) schlechtere Performanz. Die Konfiguration mit 16 kB ist im Durchschnitt 4,3 % schlechter. Dies ist insbesondere auf die Anwendungen BitonicSort, BubbleSort, FFT und RadixSort zurückzuführen. Um eine mit dem Scratchpad-Speicher vergleichbare Performanz zu erzielen, benötigen BubbleSort und FFT mindestens 16 kB Cache-Speicher sowie RadixSort mindestens 8 kB Speicher. Die Anwendung BitonicSort ist selbst mit einem Instruktionscache von 32 kB 13,4 % langsamer als die CPU mit Scratchpad-Speicher. Mit 2 kB Cache-Speicher ist der Instruktionscache bis zu 59,0 % (RadixSort) bzw. 69,6 % (FFT) langsamer als die Scratchpad-Speicher. Dies ist auf ungünstige Zugriffsmuster der Anwendungen auf den Instruktionscache zurückzuführen. Es ist davon auszugehen, dass durch die Verwendung eines satzassoziativen Caches die Performanz dieser Anwendungen signifikant gesteigert werden könnte. Hierbei kann jede Adresse des L2-Speichers in zwei oder mehr Cachezeilen zwischengespeichert werden (siehe Abschnitt 2.3.1).

Bei der Verwendung von 16 kB Instruktionscache zeigen die Anwendungen BubbleSort, DES, Filterbank, FIR-Filter, MatrixMult sowie SMA eine um weniger als 1 % schlechtere Performanz als der Scratchpad-Speicher. Hier ist nach einem Durchlauf (*Steady-State*-Iteration) der Streaming-Anwendungen der gesamte benötigte Programmcode im Cache gespeichert und es muss (fast) nicht mehr auf den L2-Speicher zugegriffen werden. Während die Anwendung FIR-Filter mit 8 kB Cache-Speicher eine Trefferquote[9] (Hitrate) von 99,98 % aufweist, liegt diese für FFT bei lediglich 99,81 % sowie für BitonicSort bei 98,92 %.

Bei den hier präsentierten Untersuchungen verfügt jede CPU über ein eigenes Programmabbild, die hintereinander im L2-Speicher abgelegt sind. Im Rahmen von weiterführenden Arbeiten wird daran gearbeitet, dass der CoreVA-MPSoC-Compiler ein Programmabbild für alle CPUs eines Clusters erstellt. Hierdurch kann der Bedarf an L2-Speicher signifikant reduziert werden.

In diesem Abschnitt wurde die Integration eines Instruktionscaches in den CPU-Cluster des CoreVA-MPSoCs vorgestellt. Der Mehrbedarf an Chipfläche gegenüber einem Scratchpad-Speicher gleicher Größe (16 kB) beträgt 6,4 %. Ein Instruktionscache mit 32 kB Datenspeicher ist bei acht von neun betrachteten Anwendungen weniger als 0,3 % langsamer als der Scratchpad-Speicher. Dies zeigt, dass die Speicher-Zugriffsmuster, die durch die wiederholte Ausführung von Streaming-Anwendungen entstehen, prinzipiell gut für Instruktionscaches geeignet sind. Ein Cache mit 32 kB Datenspeicher führt jedoch zu einem sehr hohen Flächenbedarf. Aus diesem Grund sind im Rahmen von weiterführenden Arbeiten verschiedene Optimierungen am Instruktionscache denkbar. Ein Beispiel hierfür ist das Vorladen von Instruktionen (*Prefetching*) sowie die Verwendung von satzassoziativen Caches (siehe Abschnitt 2.3.1). Die verwendeten Compiler für die Sprachen C und StreamIt können zudem für die Verwendung eines Instruktionscaches optimiert werden, indem beispielsweise Schleifen weniger häufig abgerollt werden. Durch diese Optimierungen sollte es möglich sein, einen Instruktionscache mit 8 kB bis

[9]Bestimmt über die gesamte Laufzeit der Anwendung incl. Initialisierungsphase

16 kB Datenspeicher mit einem geringen Einfluss auf die Performanz des CPU-Clusters einzusetzen.

5.8 Analyse des Energiebedarfs von Kommunikation im CPU-Cluster

In diesem Abschnitt wird der Energiebedarf für die Kommunikation und Synchronisierung innerhalb des CPU-Clusters des CoreVA-MPSoCs analysiert. Hierzu wird die Energie für den Zugriff auf die verschiedenen Speicher im CPU-Cluster sowie für den Aufruf von Synchronisierungsfunktionen bestimmt.

Ein mögliches Optimierungsziel des CoreVA-MPSoC-Compilers (siehe Abschnitt 3.2.3) ist die Minimierung der benötigten Energie für die Ausführung einer Anwendung. Hierbei können Ober- bzw. Untergrenzen für Metriken wie Durchsatz, Latenz und Speicherbedarf berücksichtigt werden. Der CoreVA-MPSoC-Compiler muss hierfür unter anderem den Energiebedarf für die Ausführung einer Anwendung auf einer einzelnen CPU modellieren. Details hierzu sind unter anderem in den Veröffentlichungen [224; 226] zu finden. Um diese Modelle um die Ebene des CPU-Clusters zu erweitern, muss lediglich der Mehraufwand für die Kommunikation und Synchronisierung im Cluster bestimmt werden.

Der Energiebedarf wird mithilfe von Gatelevel-Simulationen mit den extrahierten Netzlisten der in Abschnitt 6.2 beschriebenen platzierten und verdrahteten CPU-Cluster bestimmt (*Post-Place-and-Route*-Simulation). Die CPU-Cluster werden bei der maximalen Taktfrequenz des platzierten und verdrahteten Layouts mit acht CoreVA-CPUs und 128 kB gemeinsamen L1-Speicher von 714 MHz simuliert.

5.8.1 Energiebedarf von Speicherzugriffen

Kommunikation im CPU-Cluster des CoreVA-MPSoCs wird durch den Zugriff auf einen lokalen L1-Speicher einer anderen CPU im Cluster bzw. den gemeinsamen L1-Datenspeicher realisiert. Daher ergibt sich der Energiebedarf für Kommunikation im Cluster aus dem Mehraufwand für Zugriffe auf diese Speicher. Der Mehraufwand ist die Differenz des Energiebedarfs von lokalen Speicherzugriffen und Zugriffe auf andere Speicher im CPU-Cluster.

Auf den CPUs des simulierten Clusters werden synthetische Assembler-Benchmarks ausgeführt, die mithilfe eines Skriptes automatisiert erzeugt werden. Eine CPU im Cluster greift alle zehn Takte auf den eigenen lokalen L1-Datenspeicher (L1-L CPU0), den L1-Datenspeicher einer anderen CPU (L1-L CPU1) bzw. den gemeinsamen L1-Datenspeicher (gem. L1) zu. In den übrigen Takten werden Leerinstruktionen (NOPs) ausgeführt. Die anderen CPUs führen ebenfalls NOPs aus. Als Referenz wird ein Benchmark verwendet, bei dem die Instruktionen aller CPUs aus NOPs bestehen. Insgesamt werden pro Benchmark 10000 Instruktionen ausgeführt.

Aus den bei der Simulation aufgenommenen Schaltaktivitäten wird die Leistungsaufnahme des CPU-Clusters bei der Ausführung des jeweiligen Benchmarks bestimmt und hieraus die Energie pro Speicherzugriff berechnet. In Abbildung 5.27 ist links der Energiebedarf beim Schreiben bzw. Lesen von Nullen (0x00000000) und in der Mitte der Energiebedarf bei der Verwendung von Einsen (0xFFFFFFFF) abgebildet. Als Vergleich ist rechts der Energiebedarf von Leerinstruktionen (NOPs) aufgeführt. Untersuchungen mit verschiedenen Datenwörtern haben ergeben, dass Nullen das Minimum und Einsen das Maximum bezüglich des Energiebedarfs darstellen (siehe Anhang D). Die Simulationen sind mit einem CPU-Cluster mit vier CPUs durchgeführt. Simulationen von CPU-Clustern mit zwei bis 16 CPUs zeigen vergleichbare Ergebnisse (siehe Anhang D). Der Energiebedarf ist in einen statischen Anteil (verursacht vor allem durch Leckströme der Transistoren) und einen dynamischen Anteil (bedingt durch z. B. das Schalten der Transistoren) aufgeteilt.

Die statische Verlustleistung des gesamten CPU-Clusters beträgt 13,18 mW bzw. 18,46 pJ pro Takt. Dies entspricht 4,61 pJ pro Takt und CPU. Die Ausführung einer NOP-Operation auf einer CPU benötigt 27,46 pJ mit einem dynamischen Anteil von 22,84 pJ. Ungefähr 20 % der Energie wird vom Taktbaum für die Taktverteilung aufgewendet. Die Instruktionsspeicher benötigt ca. 12 % der Energie jeder CPU, die Beispielanwendung liest hierbei jeden zweiten Takt eine 64-bit-Instruktionsgruppe. Bei einem Zugriff alle

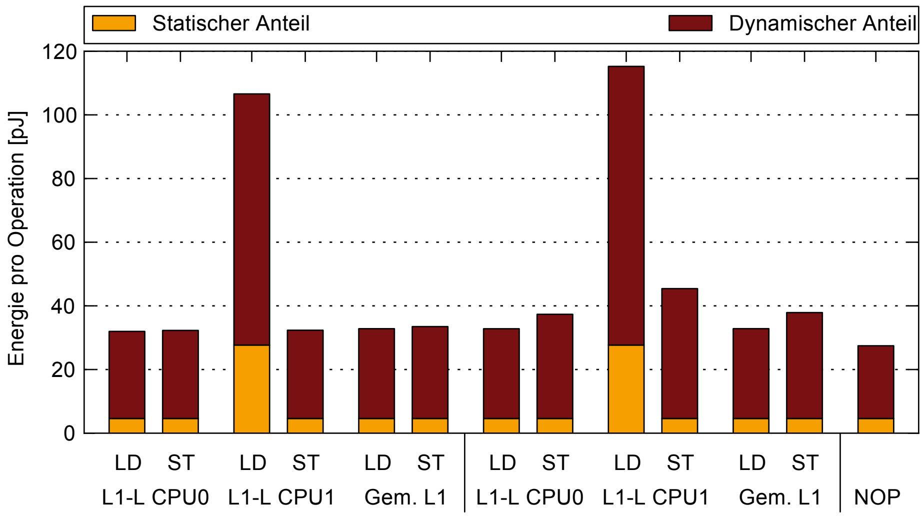


Abbildung 5.27: Energiebedarf für den Zugriff auf verschiedene Speicher des CPU-Clusters. Links: Lesen (*Load*, LD) und Schreiben (*Store*, ST) von Nullen, Mitte: Lesen/Schreiben von Einsen, Rechts: NOPs.

zehn Takte benötigt der Datenspeicher weniger als 2 % der Energie der jeweiligen CPU. Eine Aufschlüsselung des Energiebedarfs der CPU-Einzelkomponenten ist nicht möglich, da die HDL-Hierarchien von der Synthese-Software aufgebrochen werden müssen, um die CoreVA-CPU effizient auf die 28-nm-FD-SOI-Technologie abbilden zu können.

Schreibt eine CPU in ihren lokalen L1-Datenspeicher (L1-L CPU0) Nullen, benötigt dies 32,08 pJ. Ein entsprechender Lesezugriff benötigt 31,77 pJ. Enthält das Datenwort nur Einsen steigt der Energiebedarf auf 37,14 pJ (Schreiben) bzw. 32,61 pJ (Lesen). Bei der Verwendung des Datenwortes `0xAAAAAAAA` (mit je 50 % Einsen und Nullen) beträgt der Energiebedarf 34,59 pJ (Schreiben) bzw. 32,18 pJ (Lesen). Das Testmuster `0x12345678` führt zu einem Energiebedarf von 34,35 pJ (Schreiben) bzw. 32,12 pJ (Lesen). Das Schreiben eines Datenwortes in den lokalen Speicher einer CPU benötigt somit immer mehr Energie als das Lesen von Daten. Die Energie pro Zugriff ist abhängig von den jeweiligen geschriebenen bzw. gelesenen Daten. Dies ist zum Teil auf die Messmethodik zurückzuführen, da zwischen jeder Speicheroperation neun NOPs ausgeführt werden. Für diesen Zeitraum liegen jeweils Nullen am Datenspeicher der CPU an. Falls nicht anders angegeben, sind im Folgenden Messungen mit `0xFFFFFFFF` als Datenwort aufgeführt.

Ein Schreibzugriff auf den L1-Speicher einer anderen CPU im Cluster (L1-L CPU1) benötigt 36,82 pJ (`0x00000000`) bzw. 45,21 pJ (`0xFFFFFFFF`). Im Vergleich zum Schreibzugriff auf den eigenen lokalen Speicher bedeutet dies eine Steigerung um 14,8 % bzw. 21,7 %. Diese Steigerung ist darauf zurückzuführen, dass die zu schreibenden Daten im FIFO der CPU-System-Schnittstelle (siehe Abschnitt 5.2.2) sowie den Registerstufen der Cluster-Verbindungsstruktur (siehe Abschnitt 5.5.2) zwischengespeichert werden.

Ein Lesezugriff auf den lokalen L1-Speicher einer anderen CPU dauert insgesamt sechs Takte und benötigt 113,97 pJ. Die lesende CPU wird hierbei für fünf Takte angehalten, bis die angefragten Daten aus dem entsprechenden Speicher ausgelesen und zur CPU zurückgesendet worden sind. Der statische Anteil des Energiebedarfs erhöht sich um den Faktor sechs auf 27,66 pJ, da die CPU nun insgesamt sechs Takte für die Operation benötigt. Diese Ergebnisse zeigen, dass Lesezugriffe auf Speicher im CPU-Cluster nach Möglichkeit vermieden werden sollten. Hierzu kann beispielsweise das in Abschnitt 2.6.4 vorgestellte Synchronisierungs- und Kommunikationsverfahren auf Blockebene verwendet werden. Zugriffe auf den gemeinsamen L1-Datenspeicher benötigen nur 0,01 % (lesend, 32,63 pJ) bzw. 1,5 % (schreibend, 37,68 pJ) mehr Energie als der Zugriff auf den lokalen L1-Speicher. Je nach Hardware-Konfiguration und Anwendung besteht jedoch die Möglichkeit von Zugriffskonflikten (siehe Abschnitt 5.5.2). Im Fall eines Konfliktes muss eine CPU für mindestens einen Takt angehalten werden, was den Energiebedarf für die Speicheroperation erhöht.

5.8.2 Energiebedarf von Synchronisierung

Die Synchronisierung der Kommunikation zwischen verschiedenen CPUs des CoreVA-MPSoCs erfolgt über das in Abschnitt 2.6.4 vorgestellte blockbasierte Synchronisierungsverfahren. In diesem Abschnitt wird der Energiebedarf für den Aufruf der Synchronisierungsfunktionen ohne Blockieren bestimmt. Anschließend wird der zusätzliche Energiebedarf bestimmt, falls eine Synchronisierungsfunktion blockiert und auf eine andere CPU gewartet werden muss.

Für die erste Untersuchung wird angenommen, dass der gewünschte Datenpuffer zur Verfügung steht und der entsprechende Mutex freigegeben ist. Daher fällt lediglich der Zeitaufwand[10] für den Aufruf der Funktionen GetBuffer und SetBuffer an (siehe Abschnitt 5.3).

Es wird ein synthetischer C-Benchmark verwendet, bei dem eine CPU den Zustand von 100 Puffern überprüft (GetBuffer) und den entsprechenden Puffer anschließend sofort wieder frei gibt (SetBuffer). Für die Simulation wird ein platzierter und verdrahteter CPU-Cluster mit 16 CPUs verwendet (siehe Abschnitt 6.2). Bei einer simulierten Taktfrequenz von 714 MHz beträgt die Leistungsaufnahme der CPU 22,04 mW bzw. 30,86 pJ pro Takt. Die Aufrufe von GetBuffer und SetBuffer benötigen jeweils 16 Takte (siehe Abschnitt 5.3). Dies führt zu einem Energiebedarf von 987,39 pJ für eine Synchronisierung einer CPU, wenn der entsprechende Puffer bzw. Mutex verfügbar ist.

Ist der Mutex gesperrt, muss die CPU blockieren, bis eine andere CPU den Mutex freigibt. Der Mutex liegt bei einer reinen Softwareimplementierung im lokalen L1-Datenspeicher der CPU. In diesem Fall muss die CPU regelmäßig den Zustand des Mutex überprüfen und somit Instruktionen ausführen (*Busy-Waiting*). Damit die CPU hierbei nicht jeden Takt auf den L1-Datenspeicher zugreift, werden nach jeder Abfrage des Speichers zehn Leerinstruktionen (NOPs) ausgeführt.

Um Busy-Waiting zu vermeiden, wird im Folgenden eine Mutex-Hardwareimplementierung zur Synchronisierung vorgestellt und untersucht. Diese ist als Hardwareerweiterung für die CoreVA-CPU (siehe Abschnitt 2.2) realisiert. Die Mutex-Hardwareerweiterung verfügt über 1-bit-Register, die von der lokalen CPU sowie über die Bus-Slave-Schnittstelle der CPU geschrieben und gelesen werden können. Versucht die lokale CPU einen Mutex (bzw. das entsprechende Register) auszulesen, der Null ist („Mutex gesperrt“), wird die CPU angehalten, indem der Takt abgeschaltet wird (Clock-Gating). Erst wenn eine andere CPU über den Bus-Slave auf die Mutex-Hardwareerweiterung zugreift, läuft die CPU weiter. Die Anzahl der Register ist zur Entwurfszeit konfigurierbar.

In Abbildung 5.28 ist der Flächenbedarf einer CoreVA-CPU mit zwei VLIW-Slots und 16 bis 2096 Mutex-Register dargestellt. Hierbei ist der Flächenbedarf der lokalen Instruktions- und Datenspeicher, der CPU-Pipeline sowie der Mutex-Hardwareerweiterung getrennt aufgeführt. Die Taktfrequenz beträgt 800 MHz. Keine der betrachteten Konfigurationen beeinflusst die maximale Taktfrequenz der CPU von ca. 900 MHz.

[10] in CPU-Takten

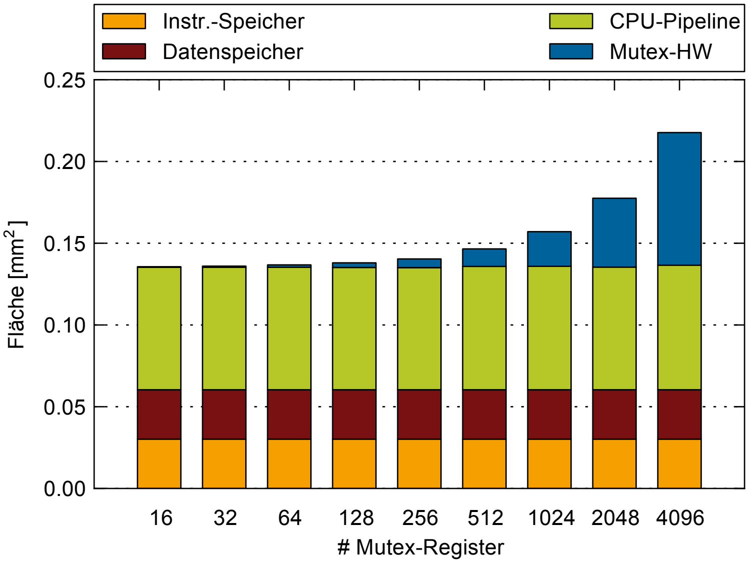


Abbildung 5.28: Flächenbedarf einer 2-Slot-CoreVA-CPU und verschiedener Konfigurationen der Mutex-Hardwareerweiterung (Mutex-HW)

Die Konfigurationen mit 16 und 32 Mutex-Registern belegen weniger als 1% der Gesamtfläche der CPU. Bei einer Verdopplung der Register verdoppelt sich auch nahezu linear der Flächenbedarf der Hardwareerweiterung. Eine Konfiguration mit 256 Registern belegt 0,005 mm^2 (3,8 % der Gesamtfläche). Die Integration von 4096 Registern führt zu einem Flächenbedarf der Hardwareerweiterung von 0,081 mm^2. Dies entspricht 37,3 % der Gesamtfläche der CPU von 0,218 mm^2.

Die Anzahl der benötigten Mutex-Register hängt sehr stark von der Anwendung und der MPSoC-Konfiguration ab. Der CoreVA-MPSoC-Compiler kann die durchschnittliche Anzahl an Wartetakten für jeden Mutex abschätzen. Hierauf basierend kann entschieden werden, ob für einen Mutex die Hardwareerweiterung oder der lokalen Datenspeicher der CPU verwendet wird.

Um den Energiebedarf für ein Blockieren einer CPU zu ermitteln, wird ein weiterer synthetischer Benchmark verwendet. Eine CPU greift auf einen Puffer zu (GetBuffer) und blockiert für 10000 Takte. Der Energiebedarf bei der Verwendung eines Software-Mutex beträgt 17,04 mW bzw. 23,86 pJ pro Takt. Bei der Verwendung eines Hardware-Mutex ist eine Verringerung auf 5,51 mW bzw. 7,72 pJ pro Takt zu beobachten. Dieser Rückgang um 67 % ist darauf zurückzuführen, dass der Takt der CPU-Pipeline angehalten wird. Die Bus-Schnittstelle und der lokale Speicher der CPU sind hierbei weiterhin aktiv. Der Energiebedarf der CPU kann – abhängig von den Anforderungen der Anwendung – weiter reduziert werden, falls der Takt dieser Komponenten ebenfalls abgeschaltet wird. Unter Annahme einer mittleren Schaltwahrscheinlichkeit von 10 % ergibt sich für einen CPU-Cluster mit 16 CPUs eine durchschnittliche Leistungsaufnahme von ca. 440 mW. Die statische Verlustleistung beträgt hierbei 63 mW. Jede CoreVA-CPU mit zwei VLIW-Slots nimmt ca. 25 mW auf [226]. Die Leistungsaufnahme der Cluster-Verbindungsstruktur wird mit 40 mW abgeschätzt. Ein CPU-Cluster mit acht CPUs sowie 128 kB eng gekoppelten gemeinsamen L1-Speicher besitzt eine mittlere Leistungsaufnahme von ca. 236 mW bei einer statischen Verlustleistung von 28 mW. Der Energiebedarf eines CPU-Clusters hängt sehr stark von der ausgeführten Anwendung ab und kann deutlich geringer ausfallen, falls CPUs oft auf Daten anderer CPUs warten müssen.

5.9 OpenCL-Anwendungen mit gemeinsamem L1- oder L2-Datenspeicher

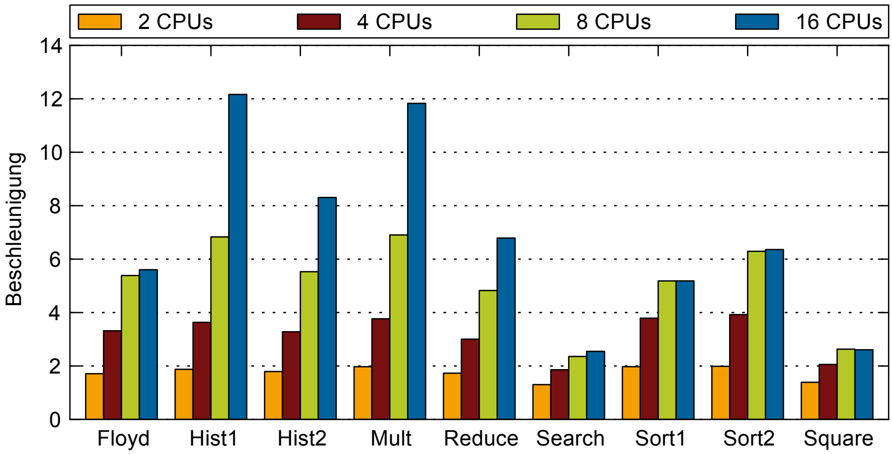
Die in Abschnitt 3.2.4 vorgestellte OpenCL-Plattform ermöglicht die Ausführung von OpenCL-Anwendungen auf dem CoreVA-MPSoC. Das Programmiermodell von OpenCL basiert auf der Verwendung eines gemeinsamen Speichers, aus dem die einzelnen Kernel-Instanzen Eingabedaten lesen und Ausgabedaten schreiben. Aus diesem Grund werden im Folgenden CoreVA-MPSoC-CPU-Cluster mit gemeinsamem L1- sowie L2-Datenspeicher als OpenCL-Geräte (*OpenCL-Devices*) verglichen (siehe Abschnitt 2.3.2 und Abschnitt 2.3.3).

Im Gegensatz zum CoreVA-MPSoC-Compiler für Streaming-Anwendungen steuert in einer OpenCL-Umgebung ein Host ein oder mehrere Devices. Als OpenCL-Host wird im Folgenden ein x86-Linux-System sowie ein CPU-Cluster des CoreVA-MPSoCs als OpenCL-Device verwendet. Die OpenCL-Plattform des CoreVA-MPSoCs basiert auf pocl [97; 162] und unterstützt daher nicht das asynchrone Kopieren von Daten auf ein Device, wenn zeitgleich ein OpenCL-Kernel auf dem Device ausgeführt wird. Dies führt dazu, dass nacheinander die Eingabedaten der Kernel auf das Device kopiert, anschließend die eigentliche Berechnung ausgeführt und die Ergebnisse vom Host zurück kopiert werden müssen. Erst hiernach können die Eingabedaten für die nächste Berechnung auf das Device kopiert werden. Anders als bei der Verwendung des CoreVA-MPSoC-Compilers und Streaming-Anwendungen ist also keine kontinuierliche Verarbeitung eines Datenstroms möglich. Aus diesem Grund ist ein direkter Vergleich zwischen StreamIt- und OpenCL-basierten Anwendungen nicht möglich. Die Leistungsfähigkeit in Bezug auf den Durchsatz liegt bei beiden Programmierumgebungen jedoch in der gleichen Größenordnung [229, S. 30 ff.].

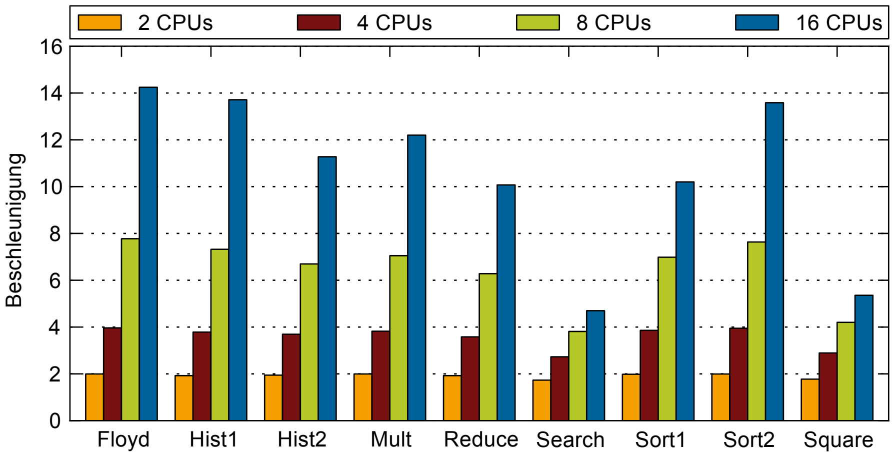
Für die Analyse verschiedener Konfigurationen des CoreVA-MPSoCs werden OpenCL-Beispielanwendungen von AMD [10], Apple [89], Intel [110] sowie NVIDIA [146] verwendet. Der Floyd-Warshall-Algorithmus sucht den kürzesten Pfad in einem Graphen. Hist1 und Hist2 berechnen ein Histogramm mit jeweils 256 Klassen. Mult führt eine Matrixmultiplikation von zwei Matrizen der Größe 16×16 aus (siehe Abschnitt 5.1). Reduce summiert alle Elemente eines Arrays auf. Search sucht nach einem gegebenen Wert in einem Array. Sort1 und Sort2 sortieren ein Array von Zahlen nach dem BitonicSort-Algorithmus (siehe Abschnitt 5.1). Square quadriert jedes Element eines Eingabearrays. Die Kernel der OpenCL-Anwendungen sind nicht für die Ausführung auf dem CoreVA-MPSoC optimiert worden und sind auf jedem OpenCL-Device ausführbar.

In Abbildung 5.29a ist die Beschleunigung von verschiedenen OpenCL-Anwendungen bei der Ausführung auf zwei bis 16 CPUs gegenüber der Ausführung auf einer CPU dargestellt. Die Kernel und der OpenCL-Host kommunizieren über gemeinsamen L2-Speicher, der über eine Speicherbank verfügt. Der Programmcode sowie die lokalen Daten der Kernel befinden sich in den lokalen L1-Speichern der CPUs. Die als Referenz verwendete Konfiguration mit einer CPU verwendet ausschließlich lokalen Speicher. Die betrachteten Anwendungen zeigen bei der Verwendung von zwei, vier und acht CPUs eine durchschnittliche Beschleunigung von 1,75, 3,18 bzw. 5,10 gegenüber der Ausführung auf einer CPU. Die Konfiguration mit 16 CPUs weist eine Beschleunigung von 6,82 auf. Hierbei besitzt Hist1 die höchste (12,16) und Search die geringste Beschleunigung (2,55). Die geringe Beschleunigung von Search und Square ist auf die geringe Parallelität der Anwendung zurückzuführen. Insbesondere bei der Verwendung von 16 CPUs gibt es bei Floyd, Hist2, Sort1 sowie Sort2 viele Zugriffskonflikte auf den L2-Datenspeicher, da nur eine Speicherbank zur Verfügung steht.

Wird gemeinsamer L1-Datenspeicher mit 32 Speicherbänken verwendet, ist im Vergleich mit L2-Speicher eine höhere Performanz zu beobachten (siehe Abbildung 5.29b). Bei der MPSoC-Konfiguration mit zwei CPUs wird eine Beschleunigung von durchschnittlich 1,92 gegenüber der Ausführung auf einer CPU erzielt. Bei der Verwendung von vier, acht und 16 CPUs beträgt die Beschleunigung 3,59, 6,42 sowie 10,59. In Abbildung 5.30 ist die Beschleunigung der betrachteten Anwendungen in Abhängigkeit von der Anzahl der L1-Speicherbänke dargestellt. Es werden 16 CPUs verwendet. Die Anwendung Floyd zeigt bei der Verwendung von zwei Bänken eine Beschleunigung von 8,52, während bei 16 Bänken eine Beschleunigung von 14,24 zu beobachten ist. Auch Sort1 und Sort2 profitieren von der Verwendung von mehreren Speicherbänken. Der Grund hierfür ist eine hohe Zahl an Zugriffen auf den gemeinsamen L1-Datenspeicher und hieraus folgend eine hohe Wahrscheinlichkeit von Zugriffskonflikten. Durch die Verwendung von mehr Speicherbänken sinkt die Wahrscheinlichkeit von solchen Konflikten. Die übrigen betrachteten Anwendungen zeigen keine bzw. nur eine geringe Abhängigkeit der Beschleunigung von der Anzahl der Speicherbänke. Dies ist darauf zurückzuführen, dass diese Anwendungen vor allem auf den lokalen L1-Datenspeicher zugreifen. Der Vergleich der Konfigurationen mit L1- und L2-Datenspeicher zeigt eine deutlich höhere Performanz bei der Verwendung des L1-Datenspeichers. Dies kann durch geringere Zugriffslatenzen des gemeinsamen L1-Datenspeicher erklärt werden.



(a) Gemeinsamer L2-Speicher mit einer Speicherbank



(b) Gemeinsamer L1-Datenspeicher mit 32 Speicherbänken

Abbildung 5.29: Beschleunigung von OpenCL-Anwendungen mit zwei bis 16 CPUs im Vergleich zur Ausführung auf einer CPU

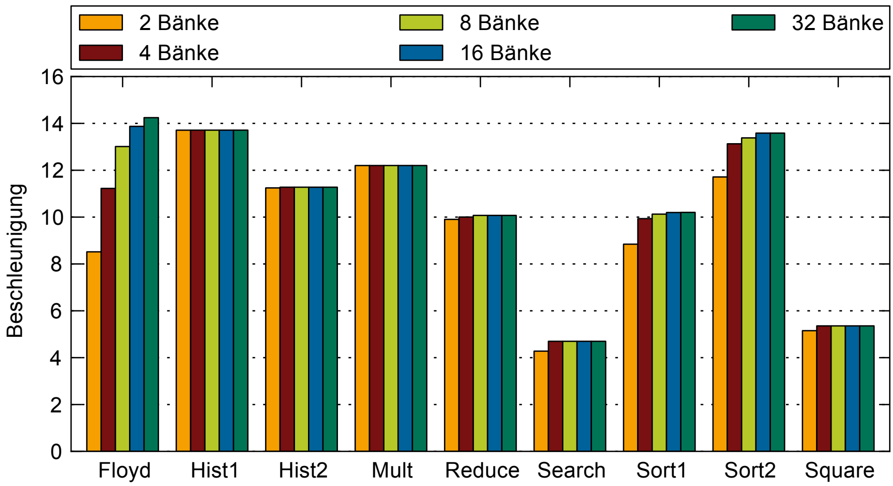


Abbildung 5.30: Beschleunigung von OpenCL-Anwendungen mit gemeinsamem L1-Datenspeicher, zwei bis 32 Speicherbänken und 16 CPUs im Vergleich zu der Ausführung auf einer CPU

In diesem Abschnitt konnte gezeigt werden, dass der CPU-Cluster des CoreVA-MPSoCs für die Verwendung als OpenCL-Device geeignet ist. Da die CoreVA-CPU bislang keine Fließkomma-Arithmetik unterstützt, können viele vorhandene OpenCL-Anwendungen nicht auf dem CoreVA-MPSoC ausgeführt werden. Die Erweiterung der CoreVA-CPU um Fließkomma-Arithmetik ist Bestandteil von aktuellen Arbeiten im Rahmen des CoreVA-MPSoC-Projektes. Jeder Cluster des CoreVA-MPSoCs muss bisher als eigenes OpenCL-Device angesprochen werden. Dies erfordert eine Anpassung von bestehenden OpenCL-Anwendungen. In weiterführenden Arbeiten ist daher die Verwendung des gesamten CoreVA-MPSoCs mit mehreren CPU-Clustern als ein OpenCL-Device sinnvoll.

5.10 Zusammenfassung

In diesem Kapitel wurde eine Entwurfsraumexploration des in Kapitel 2 eingeführten CPU-Clusters des CoreVA-MPSoCs vorgestellt. Hierbei kam eine 28-nm-FD-SOI-Standardzellentechnologie sowie Anwendungen in den Sprachen C, OpenCL und StreamIt zum Einsatz. Es wurden verschiedene Konfigurationen der CoreVA-CPU hinsichtlich ihrer Eignung für einen Einsatz im CPU-Cluster untersucht und optimiert. Hierbei sind insgesamt sechs CPU-Makros entstanden, wobei im weiteren Verlauf der Untersuchungen CPUs mit zwei VLIW-Slots zum Einsatz kamen. Anschließend wurden verschiedene

Synchronisierungsverfahren und Speicherarchitekturen mithilfe von synthetischen Benchmarks verglichen. Anhand einer C-basierten Implementierung der Anwendung Matrixmultiplikation wurde exemplarisch die Größe des Software-Entwurfsraums bei der Abbildung von Anwendungen auf das CoreVA-MPSoC aufgezeigt. Hierauf aufbauend wurde eine Entwurfsraumexploration der Verbindungsstruktur des CPU-Clusters durchgeführt. Es wurden neun Streaming-Anwendungen mit dem CoreVA-MPSoC-Compiler auf CPU-Cluster mit vier bis 32 CPUs abgebildet. Zudem wurden die Auswirkungen von Registerstufen, Bus-Standard sowie Topologie auf den Flächenbedarf und die Leistungsfähigkeit der Cluster-Verbindungsstruktur analysiert. Hieraus konnte eine empfohlene Konfiguration des CPU-Clusters mit 16 CPUs, AXI-Verbindungsstruktur, partieller oder vollständiger Crossbar sowie zwei Registerstufen abgeleitet werden. Die maximale Taktfrequenz wird durch die CoreVA-CPU beschränkt und beträgt 800 MHz. Die betrachteten Streaming-Anwendungen erreichen in dieser Konfiguration eine durchschnittliche Beschleunigung von 13,3 gegenüber der Ausführung auf einer CPU.

In einem nächsten Schritt wurde ein gemeinsamer L1-Datenspeicher in den CPU-Cluster integriert. Hierbei wurden zwei Verbindungsstrukturen (Mesh-of-Trees, Crossbar) untersucht. Zudem wurden Konfigurationen nur mit lokalem bzw. gemeinsamem Speicher sowie eine hybride Konfiguration mit beiden Speichern analysiert. Die Integration des gemeinsamen L1-Datenspeicherns hat – je noch Konfiguration – keinen oder nur einen sehr geringen Einfluss auf die maximale Taktfrequenz des CPU-Clusters. Bei der Ausführung von Streaming-Anwendungen zeigt die hybride Konfiguration deutlich weniger Zugriffskonflikte als die Konfiguration mit gemeinsamem L1-Speicher, da der gemeinsame L1-Datenspeicher hier ausschließlich für CPU-zu-CPU-Kommunikation verwendet wird.

Die Integration eines L1-Instruktionscaches erlaubt die Ausführung von Anwendungen, deren Programmabbild zu groß für einen L1-Scratchpad-Speicher ist. Es konnte gezeigt werden, dass der Flächenbedarf einer CPU bei der Verwendung eines Caches nur um 6,4 % höher ausfällt als bei einem Scratchpad-Speicher gleicher Größe.

Durch *Post-Place-and-Route*-Simulationen mit generierten synthetischen Benchmarks wurde zudem der Energiebedarf für Kommunikation und Synchronisierung im CPU-Cluster bestimmt. Für das Schreiben eines Datenwortes in den Speicher einer anderen CPU im Cluster wird 21,7 % mehr Energie als für einen lokalen Zugriff benötigt. Zugriffe auf den gemeinsamen L1-Datenspeicher benötigen mit 37,68 pJ 1,5 % mehr Energie. Durch die Verwendung einer CPU-Hardwareerweiterung kann die Leistungsaufnahme der CPU um 67 % gesenkt werden, wenn die CPU auf Daten einer anderen CPU wartet. Abschließend wurden insgesamt neun OpenCL-Anwendungen auf das CoreVA-MPSoC abgebildet und somit die Leistungsfähigkeit des CPU-Clusters als OpenCL-Gerät aufgezeigt. Basierend auf den Ergebnissen dieses Kapitels werden in Kapitel 6 verschiedene FPGA- und ASIC-basierte Prototypen des CoreVA-MPSoCs vorgestellt.

6 Prototypische Implementierung

Ziel einer prototypischen Implementierung von mikroelektronischen Systemen ist es, die Funktionsfähigkeit in realen Systemumgebungen aufzeigen zu können. Eine prototypische FPGA- oder ASIC-Implementierung ermöglicht eine funktionale Verifikation mit hoher Ausführungsgeschwindigkeit und Genauigkeit (siehe Abschnitt 3.3). Zudem können Prototypen für die Softwareentwicklung verwendet werden, bevor der finale ASIC des MPSoCs zur Verfügung steht.

Die in diesem Kapitel vorgestellten FPGA- und ASIC-Prototypen des CoreVA-MPSoCs basieren auf den Ergebnissen der im vorangegangen Kapitel vorgestellten Entwurfsraumexploration. Für die Erstellung der Prototypen wird der in Kaptitel 3 vorgestellte Entwurfsablauf verwendet.

6.1 FPGA-Prototypen

In diesem Abschnitt wird der Ressourcenbedarf verschiedener Konfigurationen des CoreVA-MPSoCs bei einer Abbildung auf die FPGA-basierten Plattformen RAPTOR und AMiRo beschrieben (siehe Abschnitt 3.3.2). Im Vergleich zur HDL-Beschreibung für die ASIC-Implementierung des CoreVA-MPSoCs werden lediglich andere SRAM-Speicherblöcke verwendet. Die Anbindung an die I/O-Schnittstellen der jeweiligen Plattform erfolgt über ein Xilinx XPS-basiertes Peripherie-Subsystem.

Alle betrachteten Konfigurationen des CPU-Clusters integrieren pro CPU jeweils 16 kB lokalen Instruktions- und Datenspeicher. Als Cluster-Verbindungsstruktur wird ein Wishbone-Bus verwendet. Die in diesem Abschnitt angegebenen Werte für den Ressourcenbedarf des CoreVA-MPSoCs sind nach der Synthese sowie dem Platzieren und Verdrahten des vollständigen Entwurfes mit den Werkzeugen Xilinx ISE 14.7 und Synopsys Synplify Premier aufgenommen. Es wird jeweils das größte auf der jeweiligen Plattform verfügbare FPGA verwendet.

6.1.1 RAPTOR2000: DB-CoreVA

RAPTOR ist ein modulares *Rapid-Prototyping*-System und wird im Rahmen dieser Arbeit für Regressionstests mit Anwendungen verwendet, die in den Sprachen StreamIt sowie C beschrieben sind. Zudem ist es einfach möglich, das System um zusätzliche I/O-Module zu erweitern und somit das CoreVA-MPSoC in realen Systemumgebungen zu testen. Hierzu steht eine grafische Oberfläche zur Initialisierung, Steuerung und

Tabelle 6.1: Ressourcenbedarf des CoreVA-MPSoCs bei der Abbildung auf das FPGA Spartan-6 XC6SLX150 des DB-CoreVAs

	Slices	Slice-LUT	Slice-Register	BRAM	Max. Taktfrequenz
1 CPU	4729	14 271	9046	16	50 MHz
	20,5 %	15,5 %	4,9 %	6,0 %	
2 CPUs	8031	24 334	15 521	32	40 MHz
	34,9 %	26,4 %	8,4 %	11,9 %	
6 CPUs	19 130	63 805	41 040	96	25 MHz
	83,0 %	69,2 %	22,3 %	35,8 %	

Überwachung des CoreVA-MPSoCs zur Verfügung. Weitere Details zum RAPTOR-System sind in Abschnitt 3.3.2 sowie in [163] zu finden.

In diesem Abschnitt werden zunächst Untersuchungen mit einem RAPTOR2000-Basisboard und dem Modul DB-CoreVA vorgestellt. Als FPGA kommt ein Spartan-6 XC6SLX150 mit der Geschwindigkeit (*Speedgrade*) -2 zum Einsatz, das in einer 45-nm-Technologie gefertigt wird [179]. Die Anbindung an den Host-PC erfolgt über eine Local-Bus-Schnittstelle. Weitere optionale I/O-Komponenten sind eine Ethernet-MAC sowie eine UART-Schnittstelle zur Anbindung des RAPTOR-Moduls DB-TFT. Zudem kann über das I/O-Subsystem L2-SRAM- oder DDR-SDRAM-Speicher für die Ein- und Ausgaben der Daten des Regressionstests in das System integriert werden (siehe Abschnitt 3.2.1).

In Tabelle 6.1 ist der Ressourcenbedarf verschiedener CPU-Cluster des CoreVA-MPSoCs bei einer Abbildung auf das DB-CoreVA dargestellt. Hierbei werden Cluster-Konfigurationen mit einer bis sechs CPUs auf dem FPGA integriert. Die rekonfigurierbaren Logikressourcen des Spartan-6-FPGAs sind in sogenannte *Slices* aufgeteilt. Jedes Slice enthält vier LUTs (*Lookup Table*) sowie acht Register [179]. Jeder BRAM-Block des FPGAs fasst 2 kB[1] an Daten, sodass pro CoreVA-CPU 16 BRAM-Blöcke für L1-Instruktions- und Datenspeicher verwendet werden. Die maximale Taktfrequenz des FPGA-Prototypen ist abhängig von der Anzahl der integrierten CPU-Kerne und beträgt bis zu 50 MHz. Der Ressourcenbedarf ist für alle Konfigurationen bei einer Taktfrequenz von 25 MHz angegeben.

Der CPU-Cluster mit einer CPU belegt 20,5 % der verfügbaren Slices sowie 15,5 % aller LUTs. Lediglich 4,9 % aller Slice-Register sowie 6,0 % aller BRAM-Blöcke werden verwendet. Bei der Integration von sechs CPUs werden 83,0 % aller Slices sowie 69,2 % aller LUTs belegt. Der CPU-Cluster verwendet also vor allem LUTs und vergleichsweise wenig Register. Dies ist darauf zurückzuführen, dass das CoreVA-MPSoC für die Implementierung in einer ASIC-Standardzellentechnologie optimiert ist.

[1] Ein BRAM-Block fasst 18 kbit, allerdings können bei einer Datenbreite von 32 bit nur 16 kbit bzw. 2 kB verwendet werden [179]

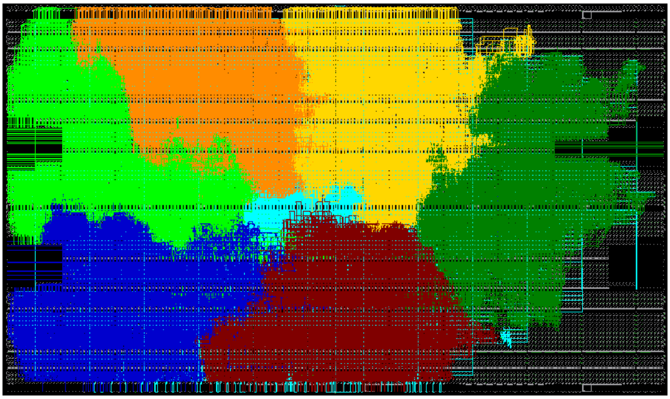


Abbildung 6.1: Layout eines CPU-Clusters mit sechs CPUs bei der Abbildung auf einem Spartan-6 XC6SLX150-FPGA. Die einzelnen CPUs sind farbig markiert, ein Teil der Cluster-Verbindungsstruktur ist in türkis in der Mitte der Abbildung zu erkennen.

In Abbildung 6.1 ist das Layout eines CPU-Clusters mit sechs CPUs dargestellt. Die Verbindungsleitungen der einzelnen CPUs sind farblich hervorgehoben, wobei sich Leitungen verschiedener Schaltungsteile überdecken können. Der türkise Bereich in der Mitte des FPGAs zeigt die Cluster-Verbindungsstruktur, wobei auch hier einzelne Teile von Verbindungsleitungen der CPUs überdeckt werden.

6.1.2 RAPTOR-XPress: DB-V5 und DB-V7

Das RAPTOR-XPress-Basisboard verfügt in Kombination mit den Modulen DB-V5 und DB-V7 über deutlich mehr Logikressourcen im Vergleich zum RAPTOR2000-basierten DB-CoreVA. Zudem besitzen die beiden FPGA-Module jeweils ein zweites FPGA („Schnittstellen-FPGA“) für die Anbindung von Peripherie-Schnittstellen wie PCIe. Die Verwendung der PCIe-Schnittstelle des RAPTOR-XPress ermöglicht eine deutlich höhere Datenrate bei der Kommunikation mit dem Host-PC im Vergleich zur PCI-Schnittstelle des RAPTOR2000-Systems. Auf dem Host-PC kann die gleiche Software zur Initialisierung und Steuerung der einzelnen CPUs des CoreVA-MPSoCs verwendet werden. Die beiden FPGAs sind über eine Inter-FPGA-Schnittstelle gekoppelt [169]. Das I/O-Subsystem des CoreVA-MPSoC kann auf dem Schnittstellen-FPGA integriert werden, um auf dem Anwender-FPGA (Virtex-7) mehr CPU-Kerne abbilden zu können. Ein weiterer Vorteil des RAPTOR-XPress-Systems ist die Möglichkeit, über serielle Hochgeschwindigkeitsverbindungen mehrere RAPTOR-XPress-Basisboards zu koppeln. Dies ermöglicht die Emulation von MPSoCs mit mehreren 100 CPUs, was Gegenstand von zukünftigen Arbeiten im Rahmen des CoreVA-MPSoC-Projektes ist. Jeder BRAM-

Tabelle 6.2: Ressourcenbedarf des CoreVA-MPSoCs bei der Abbildung auf das FPGA Virtex-7 VX690T des DB-V7

	Slices	Slice-LUT	Slice-Register	BRAM	Max. Taktfrequenz
1 CPU	5932	16 663	7496	8	124 MHz
	5,5 %	3,8 %	0,9 %	0,5 %	
2 CPUs	13 411	33 040	14 479	16	115 MHz
	12,4 %	7,6 %	1,7 %	1,1 %	
24 CPUs	107 642	388 047	168 159	192	80 MHz
	99,4 %	89,6 %	19,4 %	13,1 %	

Speicher der in diesem Abschnitt verwendeten Virtex-5- und Virtex-7-FPGAs besitzt im Vergleich zum Spartan-6-FPGA die doppelte Kapazität von 36 kbit [1; 203].

Der Ressourcenbedarf verschiedener Konfigurationen des CoreVA-MPSoCs bei der Abbildung auf das Virtex-7 VX690T-FPGA[2] [1] des DB-V7 ist in Tabelle 6.2 dargestellt. Das Virtex-7-FPGA wird in einer 28-nm-Technologie gefertigt. Eine Konfiguration mit einer CPU belegt lediglich 5,5 % aller Slices und 3,8 % aller LUTs. Die maximale Taktfrequenz beträgt 124 MHz. Die Integration von zwei CPUs belegt 12,4 % aller Slices und ermöglicht eine maximale Taktfrequenz von 115 MHz. Ein CPU-Cluster mit 24 CPUs belegt bei einer maximalen Taktfrequenz von 80 MHz 107 642 Slices (90,4 %) sowie 388 047 LUTs (89,6 %). Es werden 168 159 Register (19,4 %) sowie 192 BRAM-Speicher (13,1 %) verwendet. Wie bei dem Spartan-6-basierten Prototypen ist wiederum eine große Abhängigkeit der maximalen Taktfrequenz von der Anzahl der integrierten CPU-Kerne zu beobachten. Der kritische Pfad liegt typischerweise innerhalb der CPU-Pipeline einer CoreVA-CPU.

Das RAPTOR-Modul DB-V5 integriert ein Virtex-5 LX100T- sowie ein LX30T-FPGA [199], welche in einer 65-nm-Technologie gefertigt werden. Das Virtex-5 LX100T-FPGA[2] [203] ermöglicht die Emulation eines CPU-Clusters mit bis zu 6 CPUs. Hierbei werden 15 702 Slices (98,1 %), 54 878 LUTs (85,7 %) sowie 48 BRAM-Speicher (21,1 %) belegt. Die maximale Taktfrequenz beträgt 90 MHz. Bei der Integration von einer CoreVA-CPU wird eine maximale Taktfrequenz von 95 MHz erreicht.

Im Vergleich zum DB-CoreVA ermöglicht das DB-V5 eine höhere Taktfrequenz, jedoch nicht die Integration von zusätzlichen CPU-Kernen. Wird das DB-V7 mit Virtex-7 VX690T-FPGA verwendet, ist eine Integration eines CPU-Clusters mit bis zu 24 CPUs und Taktfrequenzen von bis zu 124 MHz möglich.

[2] Speedgrade: -2

6.1.3 Miniroboter AMiRo

Der Miniroboter AMiRo wurde wie das RAPTOR-System an der Arbeitsgruppe Kognitronik und Sensorik der Universität Bielefeld entwickelt. Das *ImageProcessing*-Modul des AMiRos verfügt wie das RAPTOR-Modul DB-CoreVA über ein Spartan-6-FPGA. Allerdings besitzt das Spartan-6 XC6SLX100-FPGA[2] 33 % weniger Logikressourcen im Vergleich zum Spartan-6 XC6SLX150-FPGA des DB-CoreVAs. Weitere Informationen zum AMiRo sind in Abschnitt 3.3.2 sowie in [91; 172] zu finden. Im I/O-Subsystem des FPGA-Entwurfs ist ein CAN-Controller [127] integriert, über den das Auslesen von Sensoren sowie das Ansprechen von Aktoren des Roboters möglich ist. Optional kann das I/O-Subsystem zudem um weitere Komponenten zur Ansteuerung von DDR-Speicher, einer Kamera sowie des Prozessors des *Cognition*-Moduls erweitert werden. Für die Initialisierung und Steuerung der CPUs des CoreVA-MPSoCs wird eine UART-Master-Schnittstelle verwendet [245].

Auf dem FPGA des AMiRos kann ein CPU-Cluster mit bis zu vier CPUs integriert werden. In Tabelle 6.3 ist der Ressourcenbedarf von Konfigurationen mit einer, zwei sowie vier CPUs dargestellt. Die maximale Taktfrequenz beträgt bis zu 60 MHz und ist wie bei dem RAPTOR-basierten Prototypen von der Anzahl der integrierten CPUs abhängig. Es werden insgesamt 16 BRAM-Speicherblöcke für die lokalen L1 Daten- und Instruktionsspeicher jeder CPU verwendet. Zwei weitere BRAM-Blöcke sind in die FIFOs der UART-Master-Schnittstelle integriert. Der CPU-Cluster mit einer CPU belegt 30,2 % der verfügbaren Slices sowie 19,2 % aller LUTs. Bei der Verwendung von vier CPUs werden 93,5 % aller Slices sowie 67,1 % der LUTs verwendet.

Aufgrund der geringeren Komplexität der verwendeten I/O-Schnittstellen belegt der AMiRo-basierte Prototyp bei gleicher CPU-Anzahl weniger FPGA-Ressourcen und besitzt eine höhere maximale Taktfrequenz im Vergleich zum RAPTOR-basierten Prototypen. Die CAN-Schnittstelle des AMiRos bietet mit 1 Mbit/s jedoch auch eine wesentlich geringere Übertragungsbandbreite als die Lokalbus-Schnittstelle des RAPTOR-Systems (800 Mbit/s bei einer Frequenz von 25 MHz).

Tabelle 6.3: Ressourcenbedarf des CoreVA-MPSoCs bei der Abbildung auf das FPGA Spartan-6 XC6SLX100 des AMiRo-Roboters

	Slices	Slice-LUT	Slice-Register	BRAM	Max. Taktfrequenz
1 CPU	4779	12 163	7903	18	60 MHz
	30,2 %	19,2 %	6,2 %	6,7 %	
2 CPUs	7747	22 452	13 795	34	50 MHz
	49,0 %	35,5 %	10,9 %	12,7 %	
4 CPUs	14 800	43 503	25 602	66	40 MHz
	93,5 %	67,1 %	20,2 %	24,6 %	

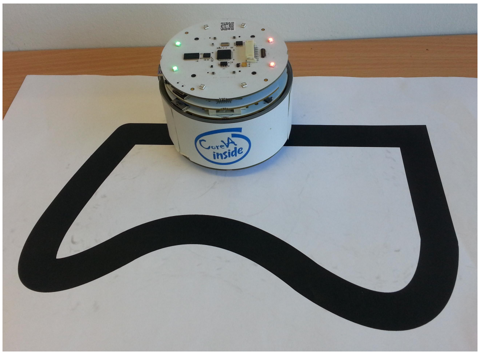


Abbildung 6.2: Demonstrator-Anwendung des Miniroboters AMiRo und des CoreVA-MPSoCs. Zwei CoreVA-CPUs führen einen Linienverfolgungsalgorithmus aus und kommunizieren mit den Komponenten des Roboters über einen CAN-Bus.

In Abbildung 6.2 ist ein AMiRo-Miniroboter bei der Ausführung einer Beispielanwendung zur Linienverfolgung dargestellt. Die Steuerung des Roboters erfolgt durch einen CPU-Cluster mit zwei CoreVA-CPUs. Der CPU-Cluster wird mit einer Taktfrequenz von 50 MHz betrieben. Eine CPU kommuniziert über den CAN-Controller des I/O-Subsystems mit Sensoren und Aktoren des AMiRo. Die zweite CPU führt die eigentliche Linienverfolgung durch. Der AMiRo verfügt über vier Infrarot-basierte Näherungssensoren [172], die an der Unterseite des Roboters angebracht sind. Mithilfe dieser Sensoren kann durch Änderungen des Reflexionskoeffizienten feststellt werden, ob sich der Roboter auf einer weißen oder schwarzen Oberfläche befindet. Diese Informationen werden verwendet, um die beiden Motoren des Roboters so anzusteuern, dass möglichst beide Sensoren eine schwarze Fläche detektieren. Hierdurch folgt der AMiRo mit konstanter Geschwindigkeit einer schwarzen Linie. Beide CPUs kommunizieren mithilfe des in Abschnitt 2.6.4 vorgestellten blockbasierten Synchronisierungsverfahrens. Weitere Informationen zum AMiRo-basierten CoreVA-MPSoC-Demonstrator sind in [245] zu finden.

6.2 ASIC-Prototypen in einer 28-nm-FD-SOI-Standardzellentechnologie

Im Entwurfsprozess von Standardzellen-ASICs werden Chip-Prototypen verwendet, um die Funktions- und Leistungsfähigkeit eines Schaltungsentwurfs aufzuzeigen. ASIC-Prototypen bieten im Vergleich zur Simulation oder FPGA-Emulation eine deutlich höhere Ausführungsgeschwindigkeit von Anwendungen und ermöglichen genaue Aussagen über Verlustleistung und maximale Taktfrequenz der Schaltung (siehe Abschnitt 3.3). Aus Kostengründen werden ASIC-Prototypen oft zusammen mit anderen Entwürfen auf einem Wafer gefertigt (*Multi Wafer Project*, MWP). Hierdurch verringern sich die Kosten für die Masken sowie die eigentliche Fertigung der Schaltung. Bei MWP-Fertigung stehen typischerweise nur einige dutzend bis hundert Chipexemplare zur Verfügung.

In diesem Abschnitt wird das Layout von verschiedenen Konfigurationen des CoreVA-MPSoC-CPU-Clusters vorgestellt. Diese Layouts können als Bestandteil eines vollständigen Chipentwurfs eines ASICs als Makro eingebettet werden. Zudem ist es möglich, einen CPU-Cluster um eine oder mehrere I/O-Schnittstellen zu erweitern und als eigenständigen ASIC zu fertigen. Als Zieltechnologie wird eine 28-nm-FD-SOI-Standardzellentechnologie verwendet. Diesbezügliche Details hierzu und zum Hardware-Entwurfsablauf werden in Abschnitt 3.1 beschrieben.

Ein Großteil der Fläche des Chip-Layouts eines CPU-Clusters wird durch CPU- und Speicher-Makros belegt (siehe Abschnitt 5.2.3). Die Speichermakros dürfen lediglich um 180° gedreht werden und schränken hierdurch die Freiheiten bei der Platzierung der Speicher und der CPU-Makros[3] ein. Nicht von Makros belegte Fläche kann für die Platzierung von Standardzellen und die Verdrahtung aller Komponenten verwendet werden. Steht hierfür zu wenig Fläche zur Verfügung, kann entweder das Platzieren und Verdrahten der Schaltung nicht erfolgreich durchgeführt oder die gewünschte Taktfrequenz nicht erreicht werden.

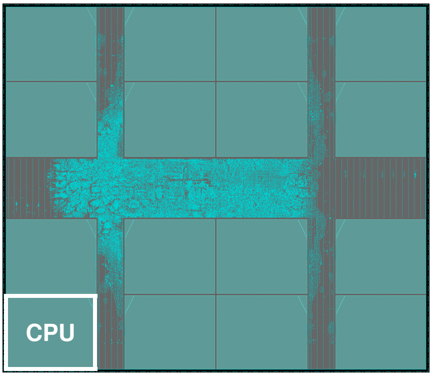
Alle Makros werden von Ringen aus Versorgungsleitungen (*Power-Ring*, Verdrahtungsebenen M9 und M10) für VDD und GND umschlossen. Über den Bereichen für Standardzellen sind horizontale und vertikale Versorgungsleitungen platziert. Zudem besitzt das gesamte Layout einen Power-Ring. Hierdurch ist sichergestellt, dass alle Komponenten des CPU-Clusters niederohmig mit VDD und GND versorgt werden.

Im Folgenden werden verschiedene Beispielkonfigurationen des CPU-Clusters prototypisch platziert und verdrahtet. Die Konfigurationen basieren auf den Synthesen und der Entwurfsraumexploration, die in Kapitel 5 vorgestellt werden.

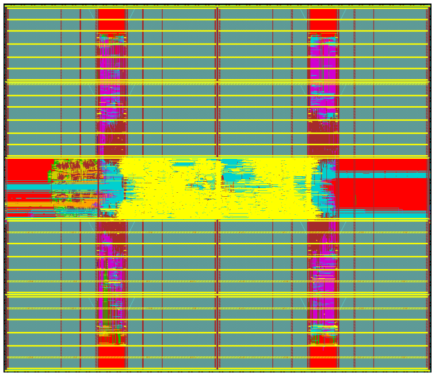
Ein CPU-Cluster mit 16 CPUs und AXI-Crossbar als Verbindungsstruktur ist in Abbildung 6.3a dargestellt. Die CPUs sind in einer 4×4-Matrix angeordnet, wobei zwischen den unteren und den oberen beiden Zeilen ein Bereich für die Standardzellen der AXI-Verbindungsstruktur frei gelassen ist. Zwischen der ersten und zweiten sowie der dritten und vierten Spalte der CPUs sind Bereiche für den Anschluss der CPUs vorgesehen. Es werden CPU-Makros vom Typ (d) verwendet (siehe Abschnitt 5.2.3).

Der verdrahtete Entwurf in Abbildung 6.3b zeigt eine sehr hohe Auslastung der obersten Verdrahtungsebene (gelb) in der Mitte des Entwurfs (*Routing-Hotspot*). Dies ist auf den hohen Verdrahtungsaufwand der AXI-Crossbar zurückzuführen. Der Flächenbedarf des Entwurfs beträgt 2,637 mm^2, was einer Steigerung von 24,6 % gegenüber der Synthese entspricht. Hierfür sind vor allem die nicht durch Standardzellen belegten Flächen verantwortlich (rot in Abbildung 6.3b). Die maximale Taktfrequenz beträgt

[3] Die CPU-Makros enthalten ebenfalls Speicher, siehe Abschnitt 5.2.3

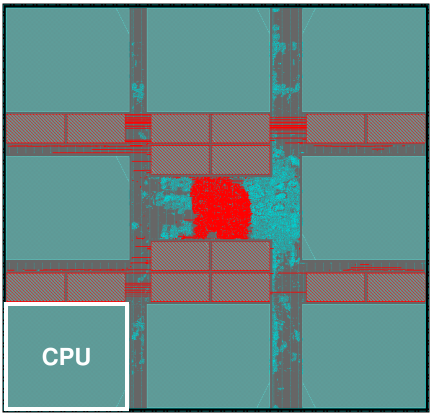


(a) Platzierter Entwurf

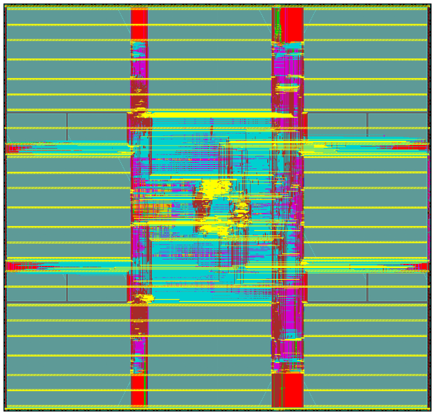


(b) Verdrahteter Entwurf

Abbildung 6.3: Chip-Layout eines CPU-Clusters mit 16 CPUs und AXI-Crossbar-Verbindungsstruktur



(a) Platzierter Entwurf



(b) Verdrahteter Entwurf

Abbildung 6.4: Chip-Layout eines CPU-Clusters mit acht CPUs sowie 128 kB eng gekoppeltem gemeinsamen L1-Speicher mit MoT-Verbindungsstruktur (rot markiert)

760 MHz und wird durch die AXI-Verbindungsstruktur begrenzt. Der Flächenbedarf des Entwurfs kann durch die Verwendung einer partiellen Crossbar gesenkt werden, wobei dies die verfügbare Übertragungsbandbreite im System verringert (siehe Abschnitt 5.5). Ähnlich wie bei der Synthese besteht ein Zusammenhang zwischen Chipfläche und maximaler Taktfrequenz. Der Flächenbedarf des Entwurfs kann gesenkt werden, indem eine geringere maximale Taktfrequenz angestrebt wird.

Aus einem fertig platziert und verdrahteten Entwurf können Informationen über Leitungsverzögerungen und -kapazitäten sowie eine Netzliste extrahiert werden. Diese Daten ermöglichen eine *Post-Place-and-Route*-Simulation und die Aufnahme von Schaltaktivitäten verschiedener Anwendungen und Benchmarks. Mithilfe dieser Schaltaktivitäten kann die Leistungsaufnahme einer Schaltung ermittelt werden (siehe Abschnitt 3.1). Für den hier vorgestellten Entwurf mit 16 CoreVA-CPUs ergibt sich eine durchschnittliche Leistungsaufnahme von ca. 440 mW (siehe Abschnitt 5.8) . Die statische Verlustleistung beträgt 63 mW.

In Abbildung 6.4 ist das Layout eines CPU-Clusters mit acht CPUs sowie 128 kB eng gekoppeltem gemeinsamen L1-Speicher dargestellt. Die CPUs sind in einer 3 × 3-Matrix angeordnet (siehe Abbildung 6.4a). Es wird das in Abschnitt 5.2.3 vorgestellte CPU-Makro vom Typ (c) verwendet. Der gemeinsame L1-Speicher sowie dessen MoT-Verbindungsstruktur sind rot markiert. Acht der insgesamt 16 Speicherbänke sind zwischen den CPUs in der linken und rechten CPU-Spalte angeordnet. Die restlichen acht Speicherbänke befinden sich in der mittleren CPU-Spalte. In der Mitte des Layouts sind die Standardzellen der MoT-Verbindungsstruktur platziert. Der AXI-Bus befindet sich zwischen der Speicher-Verbindungsstruktur und der rechten, mittleren CPU.

Der verdrahtete Entwurf ist in Abbildung 6.4b dargestellt. Die verschiedenen Verdrahtungsebenen sind in unterschiedlichen Farben gekennzeichnet. Die obersten drei Metalllagen (M10 bis M8) sind in gelb, dunkelrot und türkis markiert. Werden diese Lagen verwendet, ist dies ein Anzeichen für eine hohe Dichte der Verdrahtung (*Routing-Hotspot*). Im vorliegenden Layout ist dies in der Mitte des Entwurfs der Fall. Zudem ist die Verdrahtung der Speicherbänke des gemeinsamen L1-Datenspeichers sehr aufwendig. Die Speicher-Verbindungsstruktur begrenzt die maximale Taktfrequenz auf 718 MHz. Der Flächenbedarf beträgt 1,572 mm^2. Dies entspricht einer Steigerung gegenüber der Synthese um 24,0 %. Für diesen Mehrbedarf sind maßgeblich nicht durch Standardzellen belegte Flächen verantwortlich, die in Abbildung 6.4b rot dargestellt sind. Damit der Entwurf mit einer höheren Taktfrequenz betrieben werden kann, ist die Integration einer zweiten Pipeline-Stufe für Speicherzugriffe in die CoreVA-CPU denkbar. Die mittlere Leistungsaufnahme beträgt ca. 236 mW bei einer statischen Verlustleistung von 28 mW (siehe Abschnitt 5.8).

In Tabelle 6.4 sind verschiedene Eigenschaften der in diesem Abschnitt vorgestellten Layouts von CPU-Clustern des CoreVA-MPSoCs zusammengefasst. Neben den beiden in diesem Abschnitt bereits vorgestellten Entwürfen mit 16 und acht CPUs sind zusätzlich zwei Entwürfe mit vier bzw. zwei CPUs aufgeführt. Die Funktionsfähigkeit der Layouts ist durch *Post-Place-and-Route*-Simulationen von extrahierten Netzlisten

Tabelle 6.4: Eigenschaften von Chip-Layouts von verschiedenen Konfigurationen des CPU-Clusters

	Layout 1	Layout 2	Layout 3	Layout 4
Anzahl CPUs	16	8	4	2
Verwendetes CPU-Makro	(d)	(c)	(c)	(c)
Speicher [kB]	512	384	192	96
davon gem. Speicher [kB]	0	128	64	32
Fläche [mm^2]	2,63	1,57	0,70	0,37
Max. Taktfrequenz [MHz]	760	718	718	718
Leistungsaufnahme [mW]	440	236	115	58
Transistoren	34 947 454	23 226 466	11 082 933	5 605 898

überprüft worden. Das Layout 1 mit 16 CPUs verfügt ausschließlich über lokalen L1-Datenspeicher, während die anderen drei Layouts zusätzlich gemeinsamen L1-Datenspeicher integrieren (siehe Abbildung 6.4). Die Taktfrequenz für die vier Layouts ist für die schlechtest mögliche PVT[4]-Betriebsbedingung (*Worst Case Corner*, WCC mit 1,0 V und 125 °C) angegeben und beträgt 760 MHz für Layout 1 und 718 MHz für die übrigen Layouts. Es ist davon auszugehen, dass die maximale Taktfrequenz eines ASIC-Prototypen des CoreVA-MPSoCs bei typischen Betriebsbedingungen (z. B. 1,15 V und 25 °C)) deutlich über diesen abgeschätzten Werten liegen wird. Die Leistungsaufnahme ist für eine durchschnittliche Schaltaktivität von 10 % angegeben. Der statische Anteil an der Leistungsaufnahme beträgt 14,1 % (63 mW) für das Layout 1 mit 16 CPUs. Zudem ist die Anzahl der verwendeten Transistoren angegeben. Jede CPU verwendet ca. 322 200 Transistoren für die CPU-Logik im Standardzellenbereich sowie ca. weitere 1 572 864 Transistoren für den lokalen L1-Daten- und Instruktionsspeicher[5]. Die Cluster-Verbindungsstruktur von Layout 1 verwendet 4 626 014 Transistoren, zusammen mit den 16 integrierten CPU-Kernen verfügt das Layout über insgesamt 34 947 454 Transistoren. Layout 2 integriert insgesamt 23 226 466 Transistoren, wobei 6 291 456 Transistoren für den gemeinsamen L1-Speicher und 1 774 290 Transistoren für die Verbindungsstruktur verwendet werden.

[4] *Process Voltage Temperature*

[5] Abgeschätzt mit sechs Transistoren pro SRAM-Zelle, die Anzahl der Transistoren für die Ansteuerung der SRAM-Zellen ist nicht bekannt

6.3 Zusammenfassung

In diesem Kapitel wurden verschiedene FPGA- und ASIC-Prototypen des CoreVA-MPSoC-CPU-Clusters vorgestellt. Die hierbei verwendeten Konfigurationen basieren auf den Ergebnissen der Entwurfsraumexploration des CPU-Clusters in Kapitel 5. Die eingesetzte Entwurfsumgebung wurde in Kapitel 3 beschrieben.

Die beiden FPGA-basierten Plattformen RAPTOR und AMiRo ermöglichen den Einsatz des CoreVA-MPSoCs in realen Systemumgebungen, bevor ein ASIC-Prototyp zur Verfügung steht. In diesem Kapitel wurde gezeigt, dass CPU-Cluster mit bis zu 24 CPUs bei Taktfrequenzen von bis zu 124 MHz auf einem Virtex-7-FPGA emuliert werden können. Zudem wurden verschiedene Chip-Layouts des CPU-Clusters für eine Fertigung in einer 28-nm-FD-SOI-Standardzellentechnologie vorgestellt. Ein CPU-Cluster mit 16 CPUs benötigt hierbei eine Fläche von 2,637 mm^2 bei einer maximalen Taktfrequenz von 760 MHz und einer durchschnittlichen Leistungsaufnahme von 440 mW. Hierdurch konnte die Leistungsfähigkeit der im Rahmen dieser Arbeit entwickelten Systemarchitektur eines CPU-Clusters aufgezeigt werden.

7 Zusammenfassung und Ausblick

Der Einsatz von Multiprozessoren in eingebetteten Systemen verspricht eine Kombination aus hoher Rechenleistung, geringem Energiebedarf sowie hoher Flexibilität durch Programmierbarkeit. Many-Core-Systeme mit vielen Dutzend bis Hunderten ressourceneffizienten CPU-Kernen ermöglichen eine höhere Energieeffizienz im Vergleich zu Systemen mit wenigen, jedoch leistungsfähigeren CPUs. Anwendungen können beispielsweise aus den Bereichen Signalverarbeitung sowie Sprach- oder Bilderkennung stammen. Ein mögliches Einsatzszenario für einen Many-Core ist ein mobiler Roboter, der durch eine ressourceneffiziente Datenverarbeitung über einen langen Zeitraum autonom betrieben werden kann. Das innerhalb der Arbeitsgruppe Kognitronik und Sensorik der Universität Bielefeld entwickelte CoreVA-MPSoC ist ein hierarchisches Many-Core-System. In einem ersten Schritt ermöglicht eine enge Kopplung mehrerer Prozessorkerne in einem CPU-Cluster eine breitbandige Kommunikation mit geringer Latenz. Mehrere solcher Cluster können in einem zweiten Schritt über ein On-Chip-Netzwerk (NoC) zu einem MPSoC mit mehreren hundert CPUs verbunden werden.

In der vorliegenden Arbeit wurde der CPU-Cluster des CoreVA-MPSoCs entworfen und analysiert. Hierbei kamen CoreVA-VLIW-CPUs als Rechenwerke sowie eine 28-nm-Standardzellentechnologie zum Einsatz. Zudem wurden Anwendungen in den Sprachen StreamIt, C und OpenCL auf das CoreVA-MPSoC abgebildet sowie verschiedene Speicherarchitekturen betrachtet. Im Folgenden wird die vorliegende Arbeit auf Basis der einzelnen Kapitel zusammengefasst.

Systemarchitektur eines eng gekoppelten Prozessorclusters

In Kapitel 2 wurden verschiedene Systemarchitekturen von eingebetteten CPU-Clustern vorgestellt. Dies umfasste die Betrachtung des aktuellen Stands der Technik zu eingebetteten Multiprozessorsystemen sowie eine detaillierte Beschreibung der CoreVA-CPU als wichtige Systemkomponente des CoreVA-MPSoCs. Die CoreVA-CPU basiert auf einer ressourceneffizienten VLIW-Architektur und ist daher gut für die Verwendung in einem eingebetteten ressourceneffizienten MPSoC geeignet. Weitere wichtige Systemkomponenten sind private und gemeinsame Speicher für Daten und Instruktionen, die in Multiprozessoren typischerweise für die Kommunikation zwischen den einzelnen CPUs im System verwendet werden. Es konnte gezeigt werden, dass Scratchpad-Speicher Vorteile gegenüber Datencaches in Hinsicht auf Energiebedarf und Chipfläche aufweisen. Zudem wurde die Architektur eines eng gekoppelten gemeinsamen L1-Datenspeichers eingeführt. Eine On-Chip-Verbindungsstruktur ermöglicht eine Kommunikation der CPUs im System untereinander und mit dem gemeinsamen Speicher.

Im Rahmen dieser Arbeit ist ein auf dem NUMA-Prinzip basierendes Multiprozessorsystem entstanden, bei dem die einzelnen CPUs auf die Speicher der anderen CPUs im Cluster schreibend und lesend zugreifen können. Es wurden verbreitete Standards für Verbindungsstrukturen vorgestellt und miteinander verglichen. Basierend hierauf erfolgte eine Implementierung der beiden Standards OpenCores Wishbone und ARM AMBA AXI. Hierbei stehen jeweils die Topologien geteilter Bus, partielle Crossbar sowie vollständige Crossbar zur Verfügung. Die Synchronisierung der gleichzeitig auf den verschiedenen CPUs eines Clusters ausgeführten Anwendungsteile ist maßgeblich für die Performanz der gesamten Anwendung. Aus diesem Grund ist in dieser Arbeit ein neuartiges blockbasiertes Synchronisierungsverfahren und ein darauf aufbauendes Kommunikationsmodell entstanden. Dies erlaubt eine effiziente Synchronisierung im CPU-Cluster des CoreVA-MPSoCs mit einem geringen Mehraufwand in Bezug auf die Hardware- und Softwarekomponenten des Systems.

Hardware- und Software-Entwurfsumgebung

Die in dieser Arbeit verwendete und erweiterte Hardware- und Software-Entwurfsumgebung wurde in Kapitel 3 vorgestellt. Der hochautomatisierte Hardware-Entwurfsablauf ermöglicht eine detaillierte Entwurfsraumexploration sowie die Realisierung eines Chip-Prototypen des CoreVA-MPSoCs in einer 28-nm-Standardzellentechnologie. Die Software-Entwurfsumgebung basiert auf Compilern für die Sprachen C, StreamIt und OpenCL. Der CoreVA-MPSoC-Compiler ermöglicht eine automatisierte Partitionierung von Streaming-Anwendungen und kam in der vorliegenden Arbeit für eine Entwurfsraumexploration der Hardwarekomponenten des CPU-Clusters im CoreVA-MPSoC zum Einsatz. Zudem ist eine Simulation bzw. Emulation des CoreVA-MPSoCs auf verschiedenen Abstraktionsebenen möglich. Abhängig von den Anforderungen an Genauigkeit, Simulationsgeschwindigkeit sowie Verfügbarkeit im Entwurfsprozess kann eine Simulations- bzw. Emulationsmethode gewählt werden. Detailliert wurden der in dieser Arbeit entwickelte Instruktionssatzsimulator des CPU-Clusters sowie verschiedene FPGA-basierte Prototypen beschrieben.

Bewertungsmaße und Modellierung

Bewertungsmaße für eingebettete Multiprozessorsysteme wurden in Kapitel 4 eingeführt. Des Weiteren ist ein abstraktes Modell für die Ausführungszeit von Anwendungen auf MPSoCs entstanden. Dieses Modell verwendet Informationen über die Daten- und die funktionale Parallelität einer Anwendung, um eine obere Schranke für die Performanz von Anwendungen zu ermitteln. Anschließend folgte die Einführung eines Modells für den Ressourcenbedarf des CPU-Clusters sowie einer abstrakten XML-basierten Systembeschreibung des CoreVA-MPSoCs.

Entwurfsraumexploration des CPU-Clusters im CoreVA-MPSoC

In Kapitel 5 wurde eine detaillierte Entwurfsraumexploration des CPU-Clusters beschrieben. Es kamen Anwendungen in den Sprachen C, OpenCL und StreamIt sowie eine 28-nm-FD-SOI-Standardzellentechnologie zum Einsatz. In einem ersten Schritt wurden verschiedene Konfigurationen der CoreVA-CPU hinsichtlich ihrer Eignung für einen Einsatz im CPU-Cluster untersucht und optimiert. CPU-Konfigurationen mit zwei VLIW-Slots bieten einen guten Kompromiss zwischen Flächenbedarf und Performanz und sind daher am besten für eine Verwendung im CoreVA-MPSoC geeignet. Insgesamt sind sechs platzierte und verdrahtete CPU-Makros für die weiteren Untersuchungen entstanden.

Anschließend folgte ein Vergleich verschiedener Synchronisierungsverfahren und Speicherarchitekturen (lokaler L1-, gemeinsamer L1- sowie L2-Speicher) mithilfe von synthetischen Benchmarks. Hierbei konnte der Vorteil einer blockbasierten gegenüber einer wortbasierten Synchronisierung aufgezeigt werden. L2-Speicher weisen hohe Zugriffslatenzen auf und sind daher für CPU-zu-CPU-Kommunikation wenig geeignet.

Mithilfe der manuellen Abbildung einer C-basierten Implementierung der Anwendung Matrixmultiplikation auf einen CPU-Cluster konnte die Größe des Software-Entwurfsraums aufgezeigt werden. Hierauf aufbauend folgte eine Entwurfsraumexploration der Verbindungsstruktur des CPU-Clusters. Dies umfasste die Abbildung von neun repräsentativen Streaming-Anwendungen auf einen CPU-Cluster mit vier bis 32 CPUs mithilfe des CoreVA-MPSoC-Compilers. So zeigt beispielsweise eine Konfiguration mit 16 CPUs im Durchschnitt eine Beschleunigung von 13,3 gegenüber der Ausführung auf einer CPU. Durch den Einsatz von Registerstufen kann die maximale Taktfrequenz der Cluster-Verbindungsstruktur signifikant gesteigert werden. Für einen CPU-Cluster mit 16 CPUs konnte gezeigt werden, dass eine vollständige Crossbar eine im Durchschnitt um 6,1 % höhere Performanz im Vergleich zu einem geteilten Bus ermöglicht. Die Implementierung nach dem AXI-Standard ist im Vergleich zur Wishbone-Implementierung 8,8 % größer, erlaubt jedoch das parallele Schreiben und Lesen von Daten. Bei der Verwendung von 32 CPUs und einem AXI-Bus erhöht sich der Flächenbedarf der Crossbar-Konfiguration jedoch um 21,5 % gegenüber einem geteilten Bus. Daher ist die Verwendung einer vollständigen Crossbar nur für CPU-Cluster bis maximal 16 CPUs sinnvoll. Eine partielle Crossbar besitzt im Vergleich zur vollständigen Crossbar einen geringeren Ressourcenbedarf, ermöglicht daher die Integration von mehr als 16 CPUs und bietet für viele Anwendungen eine vergleichbare Performanz.

Der CPU-Cluster wurde anschließend um einen gemeinsamen L1-Datenspeicher erweitert. In Bezug auf die maximale Taktfrequenz zeigt eine *Mesh-of-Trees*-Verbindungsstruktur Vorteile bei einer geringen Anzahl an Speicherbänken, während eine Crossbar bei 32 oder mehr Bänken höhere Taktfrequenzen erlaubt. Sowohl Konfigurationen mit ausschließlich lokalem bzw. gemeinsamem Speicher als auch eine hybride Konfiguration mit beiden Speichern zeigen – in Abhängigkeit von der Anzahl der Speicherbänke – einen vergleichbaren Flächenbedarf. Bei der Ausführung von Streaming-Anwendungen weist die hybride Konfiguration schon mit wenigen Speicherbänken eine gute Performanz auf, da der gemeinsame L1-Datenspeicher hier ausschließlich für CPU-zu-CPU-Kommunikation verwendet wird. Die Konfiguration mit gemeinsamem L1-Speicher besitzt aufgrund einer hohen Anzahl an Zugriffskonflikten eine geringere Performanz.

Die Integration eines L1-Instruktionscaches erlaubt die Ausführung von Anwendungen, deren Programmabbild zu groß für einen L1-Scratchpad-Speicher ist. Es konnte gezeigt werden, dass der Flächenbedarf einer CPU bei der Verwendung eines Caches nur um 6,4 % höher ausfällt als bei einem Scratchpad-Speicher gleicher Größe. Die Performanz des CPU-Clusters verringert sich bei der Verwendung von 16 kB Cache-Speicher um durchschnittlich 4,3 %.

Der Energiebedarf für Kommunikation und Synchronisierung im CPU-Cluster wurde anhand von *Post-Place-and-Route*-Simulationen mit synthetischen Benchmarks bestimmt. Für das Schreiben eines Datenwortes in den Speicher einer anderen CPU im Cluster wird eine Energie von 45,21 pJ benötigt. Dies entspricht einer Steigerung von 21,7 % gegenüber einem lokalen Zugriff (37,14 pJ). Zugriffe auf den gemeinsamen L1-Datenspeicher benötigen 37,68 pJ. Auf Basis dieser Ergebnisse kann der CoreVA-MPSoC-Compiler in Zukunft um eine Abschätzung des Energiebedarfs von Anwendungen erweitert werden.

Zudem wurden neun OpenCL-Anwendungen auf dem CoreVA-MPSoC ausgeführt und somit die Leistungsfähigkeit des CPU-Clusters als OpenCL-Gerät aufgezeigt. Die Verwendung eines CPU-Clusters mit 16 CPUs ermöglicht eine durchschnittliche Beschleunigung von 10,59 im Vergleich zur Ausführung auf einer einzelnen CPU.

Prototypische Implementierung

Verschiedene FPGA- und ASIC-Prototypen des CPU-Clusters bzw. des CoreVA-MPSoCs wurden in Kapitel 6 vorgestellt. Die Plattformen RAPTOR und AMiRo ermöglichen den Einsatz des CoreVA-MPSoCs in realen Systemumgebungen unter Verwendung von FPGAs der Firma Xilinx. Es konnte gezeigt werden, dass eine Emulation eines CPU-Clusters mit bis zu 24 CPUs bei Taktfrequenzen von bis zu 124 MHz auf einem Virtex-7 VX690T-FPGA möglich ist. Auf dem Miniroboter AMiRo konnte unter Verwendung von Sensorik und Aktorik des Roboters die Funktion des CoreVA-MPSoCs prototypisch verifiziert werden. Die RAPTOR-Plattform kann unter anderem für automatisierte Software-Regressionstests verwendet werden.

Für eine mögliche Fertigung in einer 28-nm-FD-SOI-Standardzellentechnologie wurden zudem verschiedene platzierte und verdrahtete Hardmakros des CPU-Clusters erstellt. Diese Makros können mithilfe eines On-Chip-Netzwerks zu einem MPSoC mit vielen hundert CPUs verbunden werden. Ein CPU-Cluster mit 16 CPUs besitzt einen Flächenbedarf von 2,637 mm^2 bei einer maximalen Taktfrequenz von 760 MHz. Die durchschnittliche Leistungsaufnahme beträgt 440 mW, wobei 63 mW auf statische Verluste zurückzuführen sind. Durch die verschiedenen FPGA- und ASIC-Prototypen konnte die Leistungsfähigkeit der in dieser Arbeit entwickelten Systemarchitektur eines CPU-Clusters aufgezeigt werden.

Weiterführende Arbeiten und Verwertbarkeit der Ergebnisse

Im Rahmen von weiterführenden Arbeiten kann die Leistungsfähigkeit der CoreVA-CPU bei einer Verwendung im CoreVA-MPSoC gesteigert werden. Beispielsweise kann die Speicherschnittstelle der CPU auf eine Datenbreite von 64 bit erweitert werden, um die maximale Datenrate der Kommunikation im CPU-Cluster zu verdoppeln. Zudem würde die Integration einer zusätzlichen Pipelinestufe in die LD/ST-Einheit der CPU höhere Taktfrequenzen für CPU-Cluster mit gemeinsamem L1-Datenspeicher ermöglichen. Das CoreVA-MPSoC wird zudem in Zukunft als Beispielanwendung für einen FPGA-Cluster verwendet, der zurzeit von der Arbeitsgruppe Kognitronik und Sensorik aufgebaut wird. Der FPGA-Cluster koppelt über serielle Hochgeschwindigkeitsverbindungen 16 RAPTOR-XPress-Basisboards mit insgesamt 64 Virtex-5 FX100T-FPGAs. Dieses System wird voraussichtlich die Emulation eines CoreVA-MPSoCs mit 384 CPUs erlauben. Durch die geplante Verwendung von Virtex-7-FPGAs ist die prototypische Implementierung von noch größeren MPSoCs möglich.

Der im Rahmen dieser Arbeit entwickelte CPU-Cluster wird in verschiedenen aktuellen und zukünftigen Arbeiten der Arbeitsgruppe Kognitronik und Sensorik der Universität Bielefeld eingesetzt werden. Der CPU-Cluster wird integraler Bestandteil eines ASIC-Prototyps des CoreVA-MPSoCs sein, der voraussichtlich 2016 in einer 28-nm-FD-SOI-Standardzellentechnologie gefertigt wird. Die Ergebnisse dieser Arbeit ermöglichen für diesen Chip-Prototypen einen bestmöglichen Kompromiss zwischen Leistungsfähigkeit und Energiebedarf unter Berücksichtigung der Chipfläche. Der Chip-Prototyp wird aller Voraussicht nach vier CPU-Cluster mit jeweils vier CoreVA-CPUs enthalten. In jeden CPU-Cluster wird ein gemeinsamer L1-Datenspeicher integriert, der auch für die NoC-basierte Kommunikation zwischen CPU-Clustern verwendet werden kann. Neben einer Integration in das RAPTOR-System (in Form eines Moduls ähnlich des DB-CoreVAs) ist ein Einsatz des ASICs in den mobilen Robotern AMiRo und Hector denkbar. Insbesondere für eine Verwendung mit diesen Robotern werden zum Zeitpunkt der Entstehung dieser Arbeit verschiedene Bildverarbeitungsalgorithmen auf das CoreVA-MPSoC portiert. Außerdem werden zurzeit neuronale Netze (z. B. selbstorganisierende Karten) auf dem CPU-Cluster des CoreVA-MPSoCs abgebildet. Hierbei ist geplant, die Architektur von CoreVA-CPU und Cluster-Verbindungsstruktur auf die Anforderungen dieser Anwendung hin zu optimieren. Ein zusätzliches Anwendungsgebiet des CPU-Clusters sind ressourceneffiziente Implementierungen von MAC[1]-Algorithmen, die im Rahmen des vom BMBF[2] geförderten Projektes „TreuFunk – Zuverlässige Funktechnologien für industrielle Steuerungen“ (Laufzeit 01/2015 – 12/2017) entwickelt werden.

Des Weiteren kann der CPU-Cluster für die Evaluierung einer Subschwell-Standardzellenbibliothek verwendet werden, die zurzeit im Rahmen des CoreVA-MPSoC-Projektes entwickelt wird. Diese Bibliothek ist für den Betrieb bei einer Versorgungsspannung von 300 mV optimiert und ermöglicht einen sehr geringen Energiebedarf pro (CPU-) Takt und somit eine hohe Energieeffizienz. Da diese geringe Spannung die maximale Taktfrequenz stark einschränkt, ist es sinnvoll, einen CPU-Cluster mit mehreren CPUs zu verwenden, um die Rechenleistung des Systems zu steigern. Zudem werden

[1] *Medium Access Control*
[2] Bundesministerium für Bildung und Forschung

zurzeit verschiedene Verfahren zur Reduktion der statischen und dynamischen Verlustleistung des CoreVA-MPSoCs untersucht. Beispiele hierfür sind die Abschaltung der Versorgungsspannung eines CPU-Clusters oder einzelner CPUs (*Power-Gating*) sowie die Verwendung einer (positiven oder negativen) Substratvorspannung (*Backgate-Biasing*).

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ACE	AXI Coherency Extensions
ADC	Analog to Digital Converter
AES	Advanced Encryption Standard
AHB	Advanced High Performance Bus
ALU	Arithmetic Logical Unit
AMBA	ARM Microcontroller Bus Architecture
AMiRo	Autonomous Mini Robot
APB	Advanced Peripheral Bus
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AST	Abstract Syntax Tree
AXI	Advanced Exensible Interface
BRAM	Block RAM
CAN	Controller Area Network
CCN	Cache Coherent Network
CMOS	Complementary Metal Oxide Semiconductor
CoreVA	Configurable Ressource Efficient VLIW Architecture
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CUDA	Compute Unified Device Architecture
DDR-SDRAM	Double-Data-Rate Synchronous-DRAM
DES	Data Encryption Standard
DLP	Data Level Parallelism
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSM	Distributed Shared Memory
DSP	Digitaler Signalprozessor
DVFS	Dynamic Voltage and Frequency Scaling
ECC	Elliptic Curve Cryptography
EDA	Electronic Design Automation

eDRAM	Embedded DRAM
FD-SOI	Fully-Depleted Silicon-on-Insulator
FFT	Fast Fourier Transformation
FIFO	First-In First-Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GDB	GNU Debugger
GDSII	Graphic Database System 2
GFLOPS	Giga Floating Point Operations Per Second
GOPS	Giga Operations Per Second
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HAL	HyperCore Architecture Line
HDL	Hardware Description Language
I/O	Input/Output
ILP	Instruction Level Parallelism
IP	Intellectual Property
IPC	Instructions per Cycle
ISR	Interrupt Service Routine
ISS	Instruction Set Simulator
LD/ST	Load/Store
LED	Light-Emitting Diode
LLVM	Modulare Compilerarchitektur, früher Low Level Virtual Machine
LLVM IR	LLVM Intermediate Representation
LTE	Long Term Evolution
LUT	Lookup Table
LVT	Low Vt, Low Threshold Voltage
MAC	Multiply Accumulate
MIMD	Multiple Instruction Multiple Data
MIMO	Multiple Input Multiple Output
MISD	Multiple Instruction Single Data
MIT	Massachusetts Institute of Technology
MMIO	Memory Mapped I/O
MMU	Memory Management Unit
MoT	Mesh of Trees

MPSoC	Multiprocessor System on a Chip
Mutex	Mutual Exclusion
MWP	Multi Wafer Project
NI	Network Interface
NoC	Network on a Chip
NOP	No Operation
NUMA	Non-Uniform Memory Access
OCP	Open Core Protocol
OCP-IP	Open-Core-Protocol International Partnership
OPB	On-Chip Peripheral Bus
OpenCL	Open Computing Language
P&R	Place and Route
PC	Program Counter
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PIF	Processor Interface
PLB	Processor Local Bus
pocl	Portable Computing Language
POSIX	Portable Operating System Interface
PVT	Process Voltage Temperature
QoS	Quality of Service
QUT	Queensland University of Technology
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
RVT	Regular Vt, Regular Threshold Voltage
SBE	Simulation-Based Estimation
SDR	Software-Defined Radio
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMA	Simple Moving Average
SMP	Symmetric Multiprocessor
SMT	Simultaneous Multithreading
SoC	System on a Chip
SPI	Serial Peripheral Interface
SpiNNaker	Spiking Neural Network Architecture
SRAM	Static Random Access Memory

TDMA	Time Division Multiple Access
TLP	Thread Level Parallelism
TM	Transactional Memory
UART	Universal Asynchronous Receiver Transmitter
ULP	Ultra Low Power
UMA	Uniform Memory Access
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
WCC	Worst Case Corner
WCET	Worst Case Execution Time
WLAN	Wireless Local Area Network
XML	Extensible Markup Language
XPS	Xilinx Platform Studio

Referenzen

[1] *7 Series FPGAs Overview v1.17*. Techn. Ber. Xilinx, Inc., 2015.

[2] *Accellera Systems Initiative*. URL: http://www.accellera.org (besucht am 01. 12. 2015).

[3] *Adapteva, Inc.: Epiphany Multicore Intellectual Property*. URL: http://www.adapteva.com/epiphany-multicore-intellectual-property (besucht am 14. 10. 2015).

[4] *AMBA APB Protocol*. Spezifikation. ARM Ltd., 2010. URL: http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.

[5] *AMBA AXI and ACE Protocol Specification*. Spezifikation. ARM Ltd., 2013. URL: http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.

[6] *AMBA Specification (Rec 2.0)*. Spezifikation. ARM Ltd., 1999. URL: http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.

[7] G. M. Amdahl. „Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities“. In: *Spring Joint Computer Conference*. ACM Press, 1967, S. 483. DOI: 10.1145/1465482.1465560.

[8] F. Angiolini, P. Meloni, S. M. Carta, L. Raffo und L. Benini. „A Layout-Aware Analysis of Networks-on-Chip and Traditional Interconnects for MPSoCs“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.3 (März 2007), S. 421–434. DOI: 10.1109/TCAD.2006.888287.

[9] F. Angiolini, P. Meloni, S. Carta, L. Benini und L. Raffo. „Contrasting a NoC and a Traditional Interconnect Fabric with Layout Awareness“. In: *Design, Automation & Test in Europe (DATE)*. IEEE, 2006, S. 124–129. DOI: 10.1109/DATE.2006.244033.

[10] *APP SDK – A Complete Development Platform*. URL: http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/ (besucht am 05. 10. 2015).

[11] *Applying the Benefits of Network on a Chip Architecture to FPGA System Design*. White Paper. Altera Corp., 2011. URL: http://www.altera.com.

[12] *Arbeitsgruppe Kognitronik und Sensorik, CITEC, Universität Bielefeld*. URL: http://www.ks.cit-ec.uni-bielefeld.de (besucht am 06. 12. 2015).

[13] *ARM Inc.* URL: `http://www.arm.com` (besucht am 30. 11. 2015).

[14] *Arteris, Inc.* URL: `http://www.arteris.com/` (besucht am 15. 11. 2015).

[15] O. Astrachan. „Bubble Sort: An Archaeological Algorithmic Analysis". In: *Technical Symposium on Computer Science Education (SIGCSE)*. ACM Press, 2003. DOI: `10.1145/611892.611918`.

[16] *Avalon Interface Specifications*. Techn. Ber. Altera Corp., 2014. URL: `http://www.altera.com`.

[17] *AXI Reference Guide*. Techn. Ber. Xilinx Inc., 2012.

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

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

[20] 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: *International Symposium on Hardware/Software Codesign (CODES)*. ACM Press, 2002, S. 73–78. DOI: `10.1145/774789.774805`.

[21] N. Bayer und A. Peleg. *Shared Memory System for a Tightly-Coupled Multiprocessor*. Patent. 2009.

[22] L. Benini, E. Flamand, D. Fuin und D. Melpignano. „P2012: Building an Ecosystem for a Scalable, Modular and High-Efficiency Embedded Computing Accelerator". In: *Design, Automation & Test in Europe (DATE)*. IEEE, 2012, S. 983–987. DOI: `10.1109/DATE.2012.6176639`.

[23] C. H. van Berkel. „Multi-Core for Mobile Phones". In: *Design, Automation & Test in Europe (DATE)*. IEEE, 2009, S. 1260–1265. ISBN: 978-3-9810801-5-5.

[24] M. Blesken. „Ein Mehrzieloptimierungsansatz zur Dimensionierung Ressourceneffizienter Integrierter Schaltungen". Diss. Paderborn, 2012.

[25] O. Bonorden, N. Bruls, U. Kastens, D. Le, F. Meyer auf der Heide, J. Niemann, M. Porrmann, U. Rückert, A. Slowik und M. Thies. „A Holistic Methodology for Network Processor Design". In: *Local Computer Networks*. 2003, S. 583–592. DOI: `10.1109/LCN.2003.1243185`.

[26] S. Borkar. „Thousand Core Chips—A Technology Perspective". In: *Design Automation Conference (DAC)*. ACM Press, 2007, S. 746–749. DOI: `10.1145/1278480.1278667`.

[27] D. Bortolotti, F. Paterna, C. Pinto, A. Marongiu, M. Ruggiero und L. Benini. „Exploring Instruction Caching Strategies for Tightly-Coupled Shared-Memory Clusters". In: *International Symposium on System-on-Chip (SoC)*. IEEE, 2011, S. 34–41. DOI: `10.1109/ISSOC.2011.6089691`.

[28] R. Braojos, A. Dogan, I. Beretta, G. Ansaloni und D. Atienza. „Hardware/Software Approach for Code Synchronization in Low-Power Multi-Core Sensor Nodes“. In: *Design, Automation & Test in Europe (DATE)*. IEEE, 2014. DOI: `10.7873/DATE.2014.181`.

[29] C. Brunelli, R. Airoldi und J. Nurmi. „Implementation and Benchmarking of FFT Algorithms on Multicore Platforms“. In: *International Symposium on System-on-Chip (SoC)*. IEEE, 2010, S. 59–62. DOI: `10.1109/ISSOC.2010.5625561`.

[30] P. Bürgisser, M. Clausen und A. Shokrollahi. *Algebraic Complexity Theory*. Springer, 1997. DOI: `10.1007/978-3-662-03338-8`.

[31] J. Byrne. „ARM CoreLink Fabric Links 16 CPUs“. In: *Microprocessor Report* (Okt. 2012). ISSN: 0899-9341.

[32] *Cadence Encounter Digital Implementation System*. URL: `http://www.cadence.com/products/di/edi_system/` (besucht am 14. 05. 2015).

[33] *Cadence Encounter RTL Compiler*. URL: `http://www.cadence.com/products/ld/rtl_compiler/` (besucht am 14. 05. 2015).

[34] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams und H. Le. „Robust Architectural Support for Transactional Memory in the Power Architecture“. In: *International Symposium on Computer Architecture (ISCA)*. ACM Press, 2013, S. 225. DOI: `10.1145/2508148.2485942`.

[35] J. D. Carpinelli. *Computer Systems Organization and Architecture*. Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201612534.

[36] *Cisco Systems, Inc.: The Internet of Things*. URL: `http://share.cisco.com/internet-of-things.html` (besucht am 03. 04. 2015).

[37] *CoreConnect Bus Architecture*. White Paper. IBM Corp., 1999.

[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest und C. Stein. *Introduction to Algorithms*. 3. Aufl. The MIT Press, 2009. ISBN: 9780262033848.

[39] 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: *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2009, S. 175–186. DOI: `10.1109/HPCA.2009.4798252`.

[40] G. De Micheli und R. Gupta. „Hardware/Software Co-Design“. In: *Proceedings of the IEEE* 85.3 (1997), S. 349–365. DOI: `10.1109/5.558708`.

[41] M. Demler. „ARM Doubles Up on CoreLink“. In: *Microprocessor Report* (Nov. 2013). ISSN: 0899-9341.

[42] M. Demler. „ARM Stretches CoreLink at Both Ends“. In: *Microprocessor Report* (Nov. 2014). ISSN: 0899-9341.

[43] M. Demler. „Cortex-A72 Takes Big Step Forward“. In: *Microprocessor Report* (Feb. 2015). ISSN: 0899-9341.

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

[45] M. Demler. „Mobileye Increases Car EyeQ“. In: *Microprocessor Report* (Juli 2015). ISSN: 0899-9341.

[46] M. Demler. „Qualcomm Scores a NoC-Out“. In: *Microprocessor Report* (Nov. 2013). ISSN: 0899-9341.

[47] *DesignWare DW_axi Databook*. Techn. Ber. Synopsys, Inc., 2013.

[48] *DesignWare DW_fifoctrl_s1_sf Datasheet*. Techn. Ber. Synopsys, Inc., 2015.

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

[50] 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: *High Performance Extreme Computing Conference (HPEC)*. IEEE, 2013. DOI: 10.1109/HPEC.2013.6670342.

[51] 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 (Jan. 2013), S. 1654–1663. DOI: 10.1016/j.procs.2013.05.333.

[52] A. Y. Dogan, J. Constantin, M. Ruggiero, A. Burg und D. Atienza. „Multi-Core Architecture Design for Ultra-Low-Lower Wearable Health Monitoring Systems“. In: *Design, Automation & Test in Europe (DATE)*. IEEE, 2012, S. 988–993. DOI: 10.1109/DATE.2012.6176640.

[53] *E16G301 Epiphany 16-Core Microprocessor*. Techn. Ber. Adapteva, Inc., 2014. URL: http://www.adapteva.com/epiphanyiii.

[54] *E64G401 Epiphany 64-Core Microprocessor*. Techn. Ber. Adapteva, Inc., 2014. URL: http://www.adapteva.com/epiphanyiv.

[55] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam und D. Burger. „Dark Silicon and the End of Multicore Scaling“. In: *International Symposium on Computer Architecture (ISCA)*. IEEE, 2011, S. 365–376. ISBN: 1450304729.

[56] C. Ferri, S. Wood, T. Moreshet, R. Iris Bahar und M. Herlihy. „Embedded-TM: Energy and Complexity-Effective Hardware Transactional Memory for Embedded Multicore Systems“. In: *Journal of Parallel and Distributed Computing* 70.10 (Okt. 2010), S. 1042–1052. DOI: 10.1016/j.jpdc.2010.02.003.

[57] 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 (Jan. 2013), S. 104–117. DOI: 10.1109/JSSC.2012.2222814.

[58] E. Fischer und G. Fettweis. „An Accurate and Scalable Analytic Model for Round-Robin Arbitration in Network-on-Chip“. In: *International Symposium on Networks-on-Chip (NoCS)*. IEEE, 2013. DOI: 10.1109/NoCS.2013.6558403.

[59] J. A. Fisher. „Very Long Instruction Word Architectures and the ELI-512“. In: *International Symposium on Computer Architecture (ISCA)*. ACM Press, 1983, S. 140–150. DOI: 10.1145/800046.801649.

[60] E. J. Fluhr, J. Friedrich, D. Dreps, V. Zyuban, G. Still, C. Gonzalez, A. Hall, D. Hogenmiller, F. Malgioglio, R. Nett, J. Paredes, J. Pille u. a. „POWER8: A 12-Core Server-Class Processor in 22nm SOI with 7.6Tb/s Off-Chip Bandwidth“. In: *International Solid-State Circuits Conference (ISSCC)*. IEEE, 2014, S. 96–97. DOI: 10.1109/ISSCC.2014.6757353.

[61] M. J. Flynn. „Some Computer Organizations and Their Effectiveness“. In: *IEEE Transactions on Computers* C-21.9 (Sep. 1972), S. 948–960. DOI: 10.1109/TC.1972.5009071.

[62] P. Francesco, P. Antonio und P. Marchal. „Flexible Hardware/Software Support for Message Passing on a Distributed Shared Memory Architecture“. In: *Design, Automation & Test in Europe (DATE)*. IEEE, 2005, S. 736–741. DOI: 10.1109/DATE.2005.156.

[63] 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: *Great Lakes Symposium on VLSI (GLSVLSI)*. ACM Press, 2014, S. 87–88. DOI: 10.1145/2591513.2591569.

[64] *GDB: The GNU Project Debugger*. URL: http://www.gnu.org/software/gdb/ (besucht am 07. 05. 2015).

[65] M. Geilen, T. Basten, B. Theelen und R. Otten. „An Algebra of Pareto Points“. In: *International Conference on Application of Concurrency to System Design (ACSD)*. IEEE, 2005, S. 88–97. DOI: 10.1109/ACSD.2005.2.

[66] M. Gordon. „Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures“. Diss. Massachusetts Institute of Technology, Mai 2010. URL: http://dspace.mit.edu/handle/1721.1/60146.

[67] M. I. Gordon, W. Thies und S. Amarasinghe. „Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs“. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, 2006, S. 151–162. DOI: 10.1145/1168919.1168877.

[68] R. Griessl. „Eine partiell rekonfigurierbare Bildverarbeitungsarchitektur für ressourceneffiziente Systeme". Diplomarbeit. Universität Paderborn, 2012.

[69] J. L. Gustafson. „Reevaluating Amdahl's Law". In: *Communications of the ACM* 31.5 (Mai 1988), S. 532–533. DOI: 10.1145/42411.42415.

[70] J. L. Gustafson. „The Scaled Sized Model: A Revision of Amdahl's Law". In: *Supercomputing*. Bd. 88. 1988, S. 130–133.

[71] L. Gwennap. „Adapteva Demos 100Gflops". In: *Microprocessor Report* (Okt. 2012). ISSN: 0899-9341.

[72] L. Gwennap. „Adapteva: More Flops, Less Watts". In: *Microprocessor Report* (Juni 2011). ISSN: 0899-9341.

[73] L. Gwennap. „Haswell Crams 18 Cores Into Xeon E5". In: *Microprocessor Report* (Sep. 2014). ISSN: 0899-9341.

[74] L. Gwennap. „How Cortex-A15 Measures Up". In: *Microprocessor Report* (Mai 2013). ISSN: 0899-9341.

[75] L. Gwennap. „Power8 Muscles Up for Servers". In: *Microprocessor Report* (Sep. 2013). ISSN: 0899-9341.

[76] L. Gwennap. „Samueli Sees New Focus on Design Skill". In: *Microprocessor Report* (Mai 2015). ISSN: 0899-9341.

[77] J. Hagemeyer, A. Hilgenstein, D. Jungewelter, D. Cozzi, C. Felicetti, U. Rueckert, S. Korf, M. Koester, F. Margaglia, M. Porrmann, F. Dittmann, M. Ditze u. a. „A Scalable Platform for Run-time Reconfigurable Satellite Payload Processing". In: *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, 2012, S. 9–16. DOI: 10.1109/AHS.2012.6268642.

[78] T. R. Halfhill. „EZchip's Tile-MX Grows 100 ARMs". In: *Microprocessor Report* (März 2015). ISSN: 0899-9341.

[79] T. R. Halfhill. „First Quark MCUs Target IoT". In: *Microprocessor Report* (Jan. 2016). ISSN: 0899-9341.

[80] T. R. Halfhill. „MIPS Boosts Multiprocessing". In: *Microprocessor Report* (Okt. 2010). ISSN: 0899-9341.

[81] T. R. Halfhill. „Opportunity NoCs, NetSpeed Answers". In: *Microprocessor Report* (Dez. 2014). ISSN: 0899-9341.

[82] T. R. Halfhill. „PowerPC 405GP Has CoreConnect Bus". In: *Microprocessor Report* (Juli 1999). ISSN: 0899-9341.

[83] T. R. Halfhill. „Sonics Gains Acceptance". In: *Microprocessor Report* (Dez. 2003). ISSN: 0899-9341.

[84] T. R. Halfhill. „Sparc M7 Tops 10 Billion Transistors". In: *Microprocessor Report* (Sep. 2014). ISSN: 0899-9341.

[85] T. R. Halfhill. „Xilinx ReARMs FPGAs“. In: *Microprocessor Report* (März 2011). ISSN: 0899-9341.

[86] C. Hamacher, Z. Vranesic, S. Zaky und N. Manjikian. *Computer Organization and Embedded Systems*. 6. Aufl. McGraw-Hill, 2011. ISBN: 0073380652.

[87] 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: *International Symposium on Computer Architecture (ISCA)*. ACM Press, 2010, S. 37. DOI: 10.1145/1816038.1815968.

[88] J. Handy. *The Cache Memory Book*. Morgan Kaufmann, 1998. ISBN: 0123229804.

[89] *Hello World: Square, Apple Inc.* URL: https://developer.apple.com/library/mac/samplecode/OpenCL_Hello_World_Example/Listings/hello_c.html (besucht am 30. 09. 2014).

[90] J. L. Hennessy und D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 4. Aufl. Morgan Kaufmann, 2006. ISBN: 0123704901.

[91] S. Herbrechtsmeier, U. Rückert und J. Sitte. „AMiRo – Autonomous Mini Robot for Research and Education“. In: *Advances in Autonomous Mini Robots* (2012), S. 101–112. DOI: 10.1007/978-3-642-27482-4_12.

[92] M. Herlihy und J. E. B. Moss. „Transactional Memory: Architectural Support for Lock-Free Data Structures“. In: *International Symposium on Computer Architecture (ISCA)*. ACM Press, 1993, S. 289–300. DOI: 10.1145/173682.165164.

[93] M. D. Hill und M. R. Marty. „Amdahl's Law in the Multicore Era“. In: *IEEE Computer* 41.7 (Juli 2008), S. 33–38. DOI: 10.1109/MC.2008.209.

[94] L. Howes und A. Munshi. *The OpenCL Specification, Version 2.1*. Spezifikation. Khronos OpenCL Working Group, 2015. URL: http://www.khronos.org/registry/cl/.

[95] W.-C. Hsu und J. Smith. „A Performance Study of Instruction Cache Prefetching Methods“. In: *IEEE Transactions on Computers* 47.5 (Mai 1998), S. 497–508. DOI: 10.1109/12.677221.

[96] *International Technology Roadmap for Semiconductors*. URL: http://www.itrs.net/ (besucht am 03. 12. 2015).

[97] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala und H. Berg. „pocl: A Performance-Portable OpenCL Implementation“. In: *International Journal of Parallel Programming* (Aug. 2014). DOI: 10.1007/s10766-014-0320-y.

[98] B. Jacob, S. Ng und D. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2010. ISBN: 0080553842.

[99] T. Jungeblut. „Entwurfsraumexploration ressourceneffizienter VLIW-Prozessoren“. Diss. Bielefeld University, 2011.

[100] T. Jungeblut, J. Ax, M. Porrmann und U. Rückert. „A TCMS-based Architecture for GALS NoCs“. In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2012, S. 2721–2724. DOI: 10.1109/ISCAS.2012.6271870.

[101] T. Jungeblut, R. Dreesen, M. Porrmann, U. Rückert und U. Hachmann. „Design Space Exploration for Resource Efficient VLIW-Processors“. In: *Design, Automation & Test in Europe (DATE)*. IEEE, 2008.

[102] T. Jungeblut, R. Dreesen, M. Porrmann, M. Thies, U. Rückert und U. Kastens. *“Netz der Zukunft” - MxMobile - Multi-Standard Mobile Plattform - Schlussbericht*. 2009.

[103] 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.

[104] D. Kaeli und P.-C. Yew. *Speculative Execution in High Performance Computer Architectures*. Chapman und Hall/CRC, 2005. ISBN: 1584884479.

[105] M. R. Kakoee, I. Loi und L. Benini. „A Resilient Architecture for Low Latency Communication in Shared-L1 Processor Clusters“. In: *Design, Automation & Test in Europe (DATE)*. IEEE, 2012, S. 887–892. DOI: 10.1109/DATE.2012.6176623.

[106] M. R. Kakoee, V. Petrovic und L. Benini. „A Multi-Banked Shared-L1 Cache Architecture for Tightly Coupled Processor Clusters“. In: *International Symposium on System-on-Chip (SoC)*. IEEE, 2012. DOI: 10.1109/ISSoC.2012.6376362.

[107] *Kalray MPPA: A New Era of Processing*. URL: http://www.irit.fr/torrents/seminars/20121214/kalray.pdf (besucht am 14. 10. 2015).

[108] D. Kanter. „Multicore Drives Embedded Growth“. In: *Microprocessor Report* (Dez. 2014). ISSN: 0899-9341.

[109] D. Kanter und L. Gwennap. „Kalray Clusters Calculate Quickly“. In: *Microprocessor Report* (Feb. 2015). ISSN: 0899-9341.

[110] V. Kartoshkin. *Bitonic Sorting*. URL: https://software.intel.com/en-us/vcsource/samples/bitonic-sorting (besucht am 29. 10. 2014).

[111] H. Klar und T. Noll. *Integrierte Digitale Schaltungen*. 3. Aufl. Springer, 2015. ISBN: 978-3-540-40600-6.

[112] C. Klarhorst. „Partitionierung von Streaming-Anwendungen auf dem CoreVA-MPSoC“. Bachelorarbeit. Universität Bielefeld, 2013.

[113] K. Krewell. „ARM Pairs Cortex-A7 With A15“. In: *Microprocessor Report* (Nov. 2011). ISSN: 0899-9341.

[114] A. D. Kshemkalyani und M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008. ISBN: 1139470310.

[115] R. Kumar, V. Zyuban und D. Tullsen. „Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling“. In: *International Symposium on Computer Architecture (ISCA)*. IEEE, 2005, S. 408–419. DOI: 10.1109/ISCA.2005.34.

[116] J. Lachmair, E. Merényi, M. Porrmann und U. Rückert. „A Reconfigurable Neuroprocessor for Self-Organizing Feature Maps“. In: *Neurocomputing* 112 (Juli 2013), S. 189–199. DOI: 10.1016/j.neucom.2012.11.045.

[117] L. Lamport. „A Fast Mutual Exclusion Algorithm“. In: *ACM Transactions on Computer Systems* 5.1 (Jan. 1987), S. 1–11. DOI: 10.1145/7351.7352.

[118] L. Lamport. „A New Solution of Dijkstra’s Concurrent Programming Problem“. In: *Communications of the ACM* 17.8 (Aug. 1974), S. 453–455. DOI: 10.1145/361082.361093.

[119] H. W. Lang. *Sequenzielle und parallele Sortierverfahren*. URL: http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/ (besucht am 14. 05. 2015).

[120] 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.

[121] M. Levy. „ARM Makes Bus Announcement“. In: *Microprocessor Report* (Juni 2003). ISSN: 0899-9341.

[122] P. Li, J. L. Shin, G. Konstadinidis, F. Schumacher, V. Krishnaswamy, H. Cho, S. Dash, R. Masleid, C. Zheng, Y. David Lin, P. Loewenstein, H. Park u. a. „A 20nm 32-Core 64MB L3 Cache SPARC M7 Processor“. In: *International Solid-State Circuits Conference (ISSCC)*. IEEE, 2015, S. 72–73. DOI: 10.1109/ISSCC.2015.7062931.

[123] Q. Li, L. Bao, T. Zhang und C. Hou. „Low Power Optimization of Instruction Cache Based on Tag Check Reduction“. In: *International Conference on Solid-State and Integrated Circuit Technology*. IEEE, 2012, S. 1–3. DOI: 10.1109/ICSICT.2012.6467763.

[124] Y. Liu und W. Zhang. „Two-Level Scratchpad Memory Architectures to Achieve Time Predictability and High Performance“. In: *Journal of Computing Science and Engineering* 8.4 (2014), S. 215–227. DOI: 10.5626/JCSE.2014.8.4.215.

[125] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini und R. Zafalon. „Analyzing On-Chip Communication in a MPSoC Environment“. In: *Design, Automation & Test in Europe (DATE)*. IEEE, 2004, S. 752–757. DOI: 10.1109/DATE.2004.1268966.

[126] M. Loghi, M. Poncino und L. Benini. „Cache Coherence Tradeoffs in Shared-Memory MPSoCs“. In: *ACM Transactions on Embedded Computing Systems* 5.2 (Mai 2006), S. 383–407. DOI: 10.1145/1151074.1151081.

[127] *LogiCORE IP AXI Controller Area Network v1.03.a*. Techn. Ber. Xilinx, Inc., 2011.

[128] *LogiCORE IP AXI Interconnect Product Guide v2.1*. Techn. Ber. Xilinx Inc., 2014.

[129] *LogiCORE IP AXI to APB Bridge v3.0*. Techn. Ber. Xilinx Inc., 2014.

[130] I. Loi und L. Benini. „A Multi Banked - Multi Ported - Non Blocking Shared L2 Cache for MPSoC Platforms“. In: *Design, Automation & Test in Europe (DATE)*. IEEE, 2014. DOI: 10.7873/DATE2014.093.

[131] B. C. Lopes und R. Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing Ltd, 2014. ISBN: 1782166939.

[132] S. Lütkemeier. „Ressourceneffiziente Digitalschaltungen für den Subschwellbetrieb“. Diss. Universität Paderborn, 2013.

[133] S. Lütkemeier, T. Jungeblut, H. Kristian, 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: 10.1109/JSSC.2012.2220671.

[134] S. Lütkemeier, T. Jungeblut, M. Porrmann und U. Rückert. „A 200mV 32b Subthreshold Processor with Adaptive Supply Voltage Control“. In: *International Solid-State Circuits Conference (ISSCC)*. IEEE, 2012, S. 484–485. DOI: 10.1109/ISSCC.2012.6177101.

[135] 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: *Design Automation Conference (DAC)*. ACM Press, 2012, S. 1137–1142. DOI: 10.1145/2228488.2228568.

[136] K. Mika. „Entwurf und Layout eines FPGA-Moduls für eine skalierbare Prototyping Plattform“. Studienarbeit. Hochschule Ostwestfalen-Lippe, 2015.

[137] B. Mohammad. *Embedded Memory Design for Multi-Core and Systems on Chip*. Springer, 2014. ISBN: 978-1461488804.

[138] G. E. Moore. „Cramming More Components onto Integrated Circuits“. In: *Electronics* 38.8 (1965), S. 114–117.

[139] *Moving Average Simple (SMA)*. URL: http://www.tradesignalonline.com/lexicon/view.aspx?id=Moving+Average+Simple+(SMA) (besucht am 02. 12. 2015).

[140] *Multi-layer AHB Overview*. Techn. Ber. ARM Ltd., 2004.

[141] S. Murali. *Designing Reliable and Efficient Networks on Chips*. Springer, 2009. ISBN: 1402097573.

[142] D. Nassimi und S. Sahni. „Bitonic Sort on a Mesh-Connected Parallel Computer". In: *IEEE Transactions on Computers* C-28.1 (Jan. 1979), S. 2–7. DOI: 10.1109/TC.1979.1675216.

[143] J. Niemann. „Ressourceneffiziente Schaltungstechnik eingebetteter Parallelrechner: GigaNetIC." Diss. 2008. ISBN: 978-3-939350-66-8.

[144] J. Niemann, C. Puttmann, M. Porrmann und U. Rückert. „GigaNetIC – A Scalable Embedded On-Chip Multiprocessor Architecture for Network Applications". In: *International Conference on Architecture of Computing Systems (ARCS)*. Springer, 2006, S. 268–282. DOI: 10.1007/11682127_19.

[145] 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 36mm^2 Heterogeneous SDR MPSoC with Energy-Aware Dynamic Scheduling and Iterative Detection-Decoding for 4G in 65nm CMOS". In: *International Solid-State Circuits Conference (ISSCC)*. IEEE, 2014, S. 188–189. DOI: 10.1109/ISSCC.2014.6757394.

[146] *NVIDIA OpenCL SDK Code Samples*. URL: https://developer.nvidia.com/opencl (besucht am 29. 10. 2014).

[147] *On-Chip Peripheral Bus Architecture Specifications Version 2.1*. Spezifikation. IBM Corp., 2001.

[148] *Open Core Protocol Specification Release 3.0*. Spezifikation. OCP-IP Association, 2009. URL: http://www.ocpip.org.

[149] *OpenCores Project*. URL: http://opencores.org/ (besucht am 02. 12. 2015).

[150] 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: *Custom Integrated Circuits Conference (CICC)*. IEEE, 2012. DOI: 10.1109/CICC.2012.6330636.

[151] 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.

[152] J. Park, J. Balfour und W. J. Dally. „Fine-grain Dynamic Instruction Placement for L0 Scratch-Pad Memory". In: *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. ACM Press, 2010, S. 137. DOI: 10.1145/1878921.1878943.

[153] S. Pasricha und N. Dutt. *On-Chip Communication Architectures: System on Chip Interconnect*. Morgan Kaufmann, 2008. ISBN: 978-0123738929.

[154] R. Patel und A. Rajawat. „Recent Trends in Embedded System Software Performance Estimation“. In: *Design Automation for Embedded Systems* 17.1 (Jan. 2014), S. 193–213. DOI: 10.1007/s10617-013-9125-2.

[155] D. A. Patterson und J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 4. Aufl. Morgan Kaufmann; 2011. ISBN: 0123747503.

[156] P. Paulin. „Programming Challenges & Solutions for Multi-Processor SoCs: An Industrial Perspective“. In: *Design Automation Conference (DAC)*. ACM Press, 2011, S. 262–267. DOI: 10.1145/2024724.2024785.

[157] S. Penolazzi, I. Sander und A. Hemani. „Predicting Bus Contention Effects on Energy and Performance in Multi-Processor SoCs“. In: *Design, Automation & Test in Europe (DATE)*. IEEE, 2011. DOI: 10.1109/DATE.2011.5763312.

[158] A. Pimentel, C. Erbas und S. Polstra. „A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels“. In: *IEEE Transactions on Computers* 55.2 (Feb. 2006), S. 99–112. DOI: 10.1109/TC.2006.16.

[159] L. a. Plana, D. Clark, S. Davidson, S. Furber, J. Garside, E. Painkras, J. Pepper, S. Temple und J. Bainbridge. „SpiNNaker: Design and Implementation of a GALS Multicore System-on-Chip“. In: *ACM Journal on Emerging Technologies in Computing Systems* 7.4 (Dez. 2011). DOI: 10.1145/2043643.2043647.

[160] *PLB6 Bus Controller Core Databook*. Techn. Ber. IBM Corp., 2013.

[161] *Plurality Ltd.* URL: http://www.plurality.com (besucht am 22. 01. 2015).

[162] *pocl Portable Computing Language*. URL: http://portablecl.org (besucht am 06. 05. 2015).

[163] 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*. IOS Press, 2009, S. 592–599. DOI: 10.3233/978-1-60750-530-3-592.

[164] *Processor Local Bus Architecture Specification Version 6 (PLB 6)*. Spezifikation. IBM Corp., 2012.

[165] C. Puttmann. „Ressourceneffiziente Hardware-Software-Kombinationen für Kryptographie mit elliptischen Kurven“. Diss. Universität Bielefeld, 2014.

[166] C. Puttmann, J. Niemann, M. Porrmann und U. Rückert. „GigaNoC – A Hierarchical Network-on-Chip for Scalable Chip-Multiprocessors“. In: *Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*. IEEE, 2007, S. 495–502. DOI: 10.1109/DSD.2007.4341514.

[167] „Qsys Interconnect“. In: *Quartus II Handbook*. 13. Aufl. Altera Corp., 2013. Kap. 8. URL: http://www.altera.com.

[168] A. Rahimi, I. Loi, M. R. Kakoee und L. Benini. „A Fully-Synthesizable Single-Cycle Interconnection Network for Shared-L1 Processor Clusters“. In: *Design, Automation & Test in Europe (DATE)*. IEEE, 2011. DOI: `10.1109/DATE.2011.5763085`.

[169] J. Romoth, D. Jungewelter, J. Hagemeyer, M. Porrmann und U. Rückert. „Optimizing Inter-FPGA Communication by Automatic Channel Adaptation“. In: *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE, 2012. DOI: `10.1109/ReConFig.2012.6416767`.

[170] E. Salminen, V. Lahtinen, K. Kuusilinna und T. Hamalainen. „Overview of Bus-Based System-on-Chip Interconnections“. In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2002, S. 372–375. DOI: `10.1109/ISCAS.2002.1011002`.

[171] A. Schneider, J. Paskarbeit, M. Schilling und J. Schmitz. „HECTOR, A Bio-Inspired and Compliant Hexapod Robot“. In: *International Conference on Biomimetics and Biohybrid Systems. Living Machines*. Springer, 2014. DOI: `10.1007/978-3-319-09435-9_51`.

[172] T. Schöpping, T. Korthals, S. Herbrechtsmeier und U. Rückert. „AMiRo: A Mini Robot for Scientific Applications“. In: *Advances in Computational Intelligence* 9094 (2015), S. 199–205. DOI: `10.1007/978-3-319-19258-1_17`.

[173] W. D. Schwaderer. *Introduction to Open Core Protocol: Fastpath to System-on-Chip Design*. Springer, 2012. ISBN: 978-1-4614-0103-2.

[174] G. Sievers. „Implementierung eines Cache-Controllers für einen eingebetteten VLIW-Prozessor“. Diplomarbeit. Universität Paderborn, 2009.

[175] M. Smid und D. Branstad. „Data Encryption Standard: Past and Future“. In: *Proceedings of the IEEE* 76.5 (Mai 1988), S. 550–559. DOI: `10.1109/5.4441`.

[176] M. J. S. Smith. *Application-Specific Integrated Circuits*. Addison-Wesley, 1997. ISBN: 0-201-50022-1.

[177] *Sonics, Inc.* URL: `http://sonicsinc.com/` (besucht am 17. 12. 2015).

[178] *SonicsGN On-Chip Network*. Techn. Ber. Sonics, Inc., 2014. URL: `http://sonicsinc.com/resources/product-briefs/`.

[179] *Spartan-6 Family Overview v2.0*. Techn. Ber. Xilinx, Inc., 2011.

[180] S. Sriram und S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. 2. Aufl. CRC Press, 2009. ISBN: 1420048015.

[181] *STBus communication system concepts and definitions*. Spezifikation. STMicroelectronics, 2007. URL: `http://www.st.com`.

[182] *STMicroelectronics*. URL: `http://www.st.com` (besucht am 02. 12. 2015).

[183] M. Strugholtz. „Analyse der Ressourceneffizienz leitungsgebundener Kommunikation in Multiprozessorsystemen“. Diss. Universität Paderborn, 2013.

[184] J. Stuecheli. *An Introduction to POWER8 Processor*. Techn. Ber. IBM Corporation, 2013. URL: `http://www.idh.ch/IBM_TU_2013/Power8.pdf`.

[185] X. Sun und L. Ni. „Scalable Problems and Memory-Bounded Speedup“. In: *Journal of Parallel and Distributed Computing* 19.1 (Sep. 1993), S. 27–37. DOI: `10.1006/jpdc.1993.1087`.

[186] X.-H. Sun und Y. Chen. „Reevaluating Amdahl’s Law in the Multicore Era“. In: *Journal of Parallel and Distributed Computing* 70.2 (2010), S. 183–188. DOI: `10.1016/j.jpdc.2009.05.002`.

[187] *Synopsys, Inc.* URL: `http://www.synopsys.com/` (besucht am 02. 12. 2015).

[188] *Synopsys Synplify Premier*. URL: `http://www.synopsys.com/Tools/Implementation/FPGAImplementation/FPGASynthesis/Pages/SynplifyPremier.aspx` (besucht am 11. 05. 2015).

[189] A. S. Tanenbaum und M. van Steen. *Distributed Systems*. 2. Aufl. Prentice Hall, 2006. ISBN: 0-13-239227-5.

[190] J. Teich. „Hardware/Software Codesign: The Past, the Present, and Predicting the Future“. In: *Proceedings of the IEEE* 100 (Mai 2012), S. 1411–1430. DOI: `10.1109/JPROC.2011.2182009`.

[191] F. Thabet, Y. Lhuillier, C. Andriamisaina, J.-M. Philippe und R. David. „An Efficient and Flexible Hardware Support for Accelerating Synchronization Operations on the STHORM Many-Core Architecture“. In: *Design, Automation & Test in Europe (DATE)*. IEEE, 2013, S. 531–534. DOI: `10.7873/DATE.2013.119`.

[192] *The LLVM Compiler Infrastructure Project*. URL: `http://www.llvm.org` (besucht am 05. 05. 2015).

[193] W. Thies. „Language and Compiler Support for Stream Programs“. Diss. Massachusetts Institute of Technology, 2009.

[194] W. Thies. *StreamIt – The StreamIt Compiler Infrastructure*. URL: `https://github.com/bthies/streamit/` (besucht am 16. 11. 2015).

[195] W. Thies, M. Karczmarek und S. Amarasinghe. „StreamIt: A Language for Streaming Applications“. In: *International Conference on Compiler Construction (CC)*. Springer, 2002, S. 179–196. DOI: `10.1007/3-540-45937-5_14`.

[196] 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: *Design, Automation & Test in Europe (DATE)*. IEEE, 2010, S. 45–50. DOI: `10.1109/DATE.2010.5457237`.

[197] J. Turley. „Cortex-A15 „Eagle“ Flies The Coop“. In: *Microprocessor Report* (Nov. 2010). ISSN: 0899-9341.

[198] J. Turley. „Plurality Gets Ambitious with 256 CPUs“. In: *Microprocessor Report* (Juni 2010). ISSN: 0899-9341.

[199] H. Urban. „Entwurf eines FPGA-basierten Prototyping Systems unter besonderer Berücksichtigung der Signalintegrität“. Diplomarbeit. Universität Paderborn, 2010.

[200] A. Vajda. *Programming Many-Core Chips*. Springer, 2011. ISBN: 978-1-4419-9738-8.

[201] J. Vennin, S. Meftali und J.-L. Dekeyser. „Understanding and Extending SystemC User Thread Package to IA-64 Platform.“ In: *International Workshop on IP Based SoC Design (IP-SoC)*. 2004.

[202] M. Verma, L. Wehmeyer und P. Marwedel. „Dynamic Overlay of Scratchpad Memory for Energy Minimization“. In: *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. ACM Press, 2004, S. 104–109. DOI: `10.1145/1016720.1016748`.

[203] *Virtex-5 Family Overview v5.1*. Techn. Ber. Xilinx, Inc., 2015.

[204] C.-W. Wang, C.-S. Lai, C.-F. Wu, S.-A. Hwang und Y.-H. Lin. „On-Chip Interconnection Design and SoC Integration with OCP“. In: *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, 2008, S. 25–28. DOI: `10.1109/VDAT.2008.4542404`.

[205] 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: *International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2008, S. 689–696. DOI: `10.1109/ICPADS.2008.18`.

[206] B. Watling. *libfiber: A User Space Threading Library Supporting Multi-Core Systems*. URL: `https://github.com/brianwatling/libfiber` (besucht am 08. 05. 2015).

[207] P. Wennemers. „Realisierung eines Hochleistungs-FPGA-Moduls für eine skalierbare Rapid-Prototyping-Umgebung“. Masterarbeit. Universität Paderborn, 2013.

[208] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands und K. Yelick. „The Potential of the Cell Processor for Scientific Computing“. In: *International Conference on Computing Frontiers (CF)*. ACM Press, 2006, S. 9–20. DOI: `10.1145/1128022.1128027`.

[209] L. Wirbel. „Intel’s Quark Harks Back to 486“. In: *Microprocessor Report* (Nov. 2013). ISSN: 0899-9341.

[210] *Wishbone Specification, Revision B.4*. Spezifikation. opencores.org, 2010. URL: http://opencores.org.

[211] *Xilinx ISE Design Suite*. URL: http://www.xilinx.com/products/design-tools/ise-design-suite.html (besucht am 11. 05. 2015).

[212] *Xilinx Platform Studio (XPS)*. URL: http://www.xilinx.com/tools/xps.htm (besucht am 11. 05. 2015).

Eigene Veröffentlichungen

[213] 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 (v1.0)*. Techn. Ber. Universität Bielefeld, 2015. DOI: 10.13140/RG.2.1.1090.2483.

[214] 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 Press, 2015, S. 3–8. DOI: 10.1145/2835512.2835513.

[215] P. Christ, G. Sievers, J. Einhaus, T. Jungeblut, M. Porrmann und U. Rückert. „Pareto-optimal Signal Processing on Low-Power Microprocessors". In: *International Conference on Sensors*. IEEE, 2013, S. 1843–1846. DOI: 10.1109/ICSENS.2013.6688593.

[216] 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: 10.1145/2852339.2852342.

[217] B. Hübener, G. Sievers, T. Jungeblut, M. Porrmann und U. Rückert. „CoreVA: A Configurable Resource-Efficient VLIW Processor Architecture". In: *International Conference on Embedded and Ubiquitous Computing (EUC)*. IEEE, 2014, S. 9–16. DOI: 10.1109/EUC.2014.11.

[218] T. Jungeblut, J. Ax, G. Sievers, B. Hübener, M. Porrmann und U. Rückert. „Resource Efficiency of Scalable Processor Architectures for SDR-based Applications". In: *Radar, Communication and Measurement Conference (RADCOM)*. 2011. DOI: 10.13140/RG.2.1.2757.6488.

[219] T. Jungeblut, S. Lütkemeier, G. Sievers, M. Porrmann und U. Rückert. „A Modular Design Flow for Very Large Design Space Explorations". In: *CDNLive! EMEA*. 2010. DOI: 10.13140/RG.2.1.2888.7202.

[220] T. Jungeblut, G. Sievers, M. Porrmann und U. Rückert. „Design Space Exploration for Memory Subsystems of VLIW Architectures". In: *International Conference on Networking, Architecture, and Storage (NAS)*. IEEE, 2010, S. 377–385. DOI: 10.1109/NAS.2010.14.

[221] 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: *International Symposium on System-on-Chip (SoC)*. IEEE, 2014. DOI: 10.1109/ISSOC.2014.6972436.

[222] 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 Intelligente Technische Systeme*. 2013, S. 79–90. ISBN: 978-3-942647-29-8.

[223] 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 28 nm FD-SOI“. In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2015, S. 1925–1928. DOI: 10.1109/ISCAS.2015.7169049.

[224] G. Sievers, P. Christ, J. Einhaus, T. Jungeblut, M. Porrmann und U. Rückert. „Design-space Exploration of the Configurable 32 bit VLIW Processor CoreVA for Signal Processing Applications“. In: *Norchip Conference*. IEEE, 2013. DOI: 10.1109/NORCHIP.2013.6702002.

[225] 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“. In: *International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2015, S. 175–181. DOI: 10.1109/MCSoC.2015.25.

[226] 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 J. Nurmi, J. Isoaho und F. Garzia. Signals and Communication Technology. Zur Veröffentlichung angenommen. Springer, 2016. ISBN: 978-94-007-1833-3.

Betreute Arbeiten

[227] J. Ax. „Optimierung der Anbindung von Hardwareerweiterungen an einen VLIW-Prozessor“. Studienarbeit. Universität Paderborn, 2010.

[228] L.-D. Braun. „Ein Simulator für eingebettete Multiprozessoren“. Master-Projekt. Universität Bielefeld, 2013.

[229] L.-D. Braun. „OpenCL Support for the CoreVA-MPSoC“. Masterarbeit. Universität Bielefeld, 2015.

[230] L.-D. Braun und S. Gaulik. „Simulator und Anwendungen für ein eingebettetes Multiprozessor-System“. Master-Projekt. Universität Bielefeld, 2014.

[231] J. Daberkow. „Eng gekoppelte, gemeinsame Datenspeicher im CoreVA-Cluster“. Bachelorarbeit. Universität Bielefeld, 2014.

[232] J. Daberkow, A. Gatting und T. Michalski. „Schnittstellenoptimierung des CoreVA-MPSoC-FPGA-Prototypen zur Durchführung und Visualisierung von Regressionstests“. Master-Projekt. Universität Bielefeld, 2016.

[233] J. Einhaus. „Entwurfsumgebung zur ressourceneffizienten Mustererkennung auf dem CoreVA-Prozessor“. Masterarbeit. Universität Bielefeld, 2013.

[234] J. Einhaus. „Implementierung eines Interrupt-Controllers für den CoreVA-Prozessor“. Master-Projekt. Universität Bielefeld, 2012.

[235] J. Exner. „Ein Echtzeit-Betriebssystem für den CoreVA-Prozessor“. Bachelorarbeit. Universität Bielefeld, 2014.

[236] A. Gatting. „NoC-Topologien und Routingstrategien im CoreVA-MPSoC“. Bachelorarbeit. Universität Bielefeld, 2014.

[237] P. Geisler. „Evaluierung der Leon 3 CPU für die Verwendung in einem Multiprozessorsystem“. Bachelorarbeit. Universität Bielefeld, 2013.

[238] K. Gravermann, P. Wallbaum, N. Kucza und P. Geisler. „Hardware-based Data Transmission in an Embedded Multiprocessor“. Master-Projekt. Universität Bielefeld, 2015.

[239] P. Jünemann. „Ein L1-Instruktionscache für das CoreVA-MPSoC“. Bachelorarbeit. Universität Bielefeld, 2015.

[240] S. Kowalzik. „Entwicklung einer Memory Protection Unit in VHDL für den CoreVA“. Bachelorarbeit. Universität Bielefeld, 2015.

[241] N. Kucza. „Implementierung eines AXI-Interconnects für ein eingebettetes Multiprozessorsystem“. Bachelorarbeit. Universität Bielefeld, 2013.

[242] N. Kucza und P. Geisler. „Exploring Frameworks for Streaming Languages and Shared Buffer Communication for StreamIt“. Master-Projekt. Queensland University of Technology, Universität Bielefeld, 2015.

[243] P. Pekala. „Ein Sensorknoten-Demonstrator für den Ultra Low Power Prozessor CoreVA“. Bachelorarbeit. Universität Bielefeld, 2013.

[244] J. Schwuchow. „Ein Multi-FPGA Prototyp das CoreVA-MPSoC“. Bachelorarbeit. Universität Bielefeld, 2014.

[245] J. Tlatlik. „CoreVA inside – Integrating the CoreVA-MPSoC with the AMiRo Platform“. Master-Projekt. Universität Bielefeld, 2015.

[246] J. Tlatlik. „Enhancement of an LLVM Compiler Backend for the CoreVA VLIW architecture“. Bachelorarbeit. Universität Bielefeld, 2013.

Anhang

A Beispiele für die Programmierung des CoreVA-MPSoCs

```
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);
   }
}
```

Abbildung A.1: StreamIt-Beispielprogramm mit Dateiein- und Ausgabe und einem Filter

```
__kernel void vectAddKernel (__global int  input_a,
      __global int  input_b, __global int  output) {
   int i = get_global_id(0);
   output[i] = input_a[i] + input_b[i];
}
```

Abbildung A.2: OpenCL-Beispielprogramm mit einem Kernel

```xml
<?xml version="1.0" encoding="UTF-8"?>
<cluster>
  <interconnects>
    <interconnect id="0" width="1" latency="0" standard="axi"
      topology="switchmatrix"/>
  </interconnects>
  <cores>
    <core id="0" if_read_latency="2" if_fifo_depths="2"
      if_write_latency="1" data_mem_size="16384">
    </core>
    <core id="1" if_read_latency="2" if_fifo_depths="2"
      if_write_latency="1" data_mem_size="16384">
    </core>
  </cores>
  <memories>
    <memory id="0" size="32768" latency="1"/>
    <memory id="1" size="32768" latency="1"/>
  </memories>
</cluster>
```

Abbildung A.3: Beispiel einer XML-Beschreibung des CoreVA-MPSoCs mit einem CPU-Cluster, zwei CPUs und zwei L2-Speichern

```c
buffer = InitRingbuffer();
data[N];
//Sender-CPU:
for (j = 0; j < N; j++){
  pushRingbuffer(buffer,data[j]);
}
//Receiver-CPU:

for (j = 0; j < N; j++){
  popRingbuffer(buffer,data[j]);
}
```

Abbildung A.4: Beispielprogramm für die Synchronisierung mit einem wortbasierten Ringbuffer

```
channel = InitMultichannel();
data[N];
tmp[N]
//Sender-CPU:
tmp=waitChannel(channel);
for (j = 0; j < N; j++){
   tmp[j] = data[j];
}
setChannel(channel);

//Receiver-CPU:
tmp=waitChannel(channel);
for (j = 0; j < N; j++){
   data[j] = tmp[j];
}
setChannel(channel);
```

Abbildung A.5: Beispielprogramm mit blockbasierter Synchronisierung

B Herleitung des analytischen Modells für die Performanz von Anwendungen

Im Folgenden wird die Vereinfachung der Gleichung 4.30 aus Abschnitt 4.2 dargestellt. Hierbei entspricht f' dem datenparallelen Anteil einer Anwendung. p ist die Anzahl der funktionalparallelen atomaren Blöcke. m entspricht der Anzahl an CPU-Kernen. Des Weiteren gilt $p, m \in \mathbb{N}$ sowie $f' \in [0,1]$.

$$\begin{aligned}
&w_{Daten+funktional''''} = w_{Daten+funktional'} + w_{zusätzlich} \\
&= \left\lceil \frac{p}{m} \right\rceil \cdot \frac{w}{p} \cdot (1-f') + \frac{1}{m} \cdot (w \cdot f' - w_{verfügbar'}) \\
&= \left\lceil \frac{p}{m} \right\rceil \cdot \frac{w}{p} \cdot (1-f') + \frac{1}{m} \cdot \left(w \cdot f' - \frac{w}{p} \cdot (1-f') \cdot (m - p \bmod m) \right) \\
&= w \cdot \left[\left\lceil \frac{p}{m} \right\rceil \cdot \frac{1}{p} \cdot (1-f') + \frac{1}{m} \cdot \left(f' - \frac{1}{p} \cdot (1-f') \cdot (m - p \bmod m) \right) \right] \\
&= w \cdot \left[\left\lceil \frac{p}{m} \right\rceil \cdot \frac{1}{p} \cdot (1-f') + \frac{f'}{m} - \frac{1}{m \cdot p} \cdot (1-f') \cdot (m - p \bmod m) \right] \\
&= w \cdot \left[(1-f') \cdot \left(\left\lceil \frac{p}{m} \right\rceil \cdot \frac{1}{p} - \frac{1}{m \cdot p} \cdot (m - p \bmod m) \right) + \frac{f'}{m} \right]
\end{aligned} \tag{B.1}$$

mit $p \bmod m = p - \left\lfloor \frac{p}{m} \right\rfloor \cdot m$ folgt

$$
\begin{aligned}
&= w \cdot \left[(1-f') \cdot \left(\left\lceil \frac{p}{m} \right\rceil \cdot \frac{1}{p} - \frac{1}{m \cdot p} \cdot \left(m - p + \left\lfloor \frac{p}{m} \right\rfloor \cdot m\right) \right) + \frac{f'}{m} \right] \\
&= w \cdot \left[(1-f') \cdot \left(\left\lceil \frac{p}{m} \right\rceil \cdot \frac{1}{p} - \frac{1}{p} + \frac{1}{m} - \frac{1}{p} \cdot \left\lfloor \frac{p}{m} \right\rfloor \right) + \frac{f'}{m} \right] \\
&= w \cdot \left[(1-f') \cdot \left(\frac{1}{m} + \frac{1}{p} \cdot \left(\left\lceil \frac{p}{m} \right\rceil - \left\lfloor \frac{p}{m} \right\rfloor - 1 \right) \right) + \frac{f'}{m} \right] \\
&= w \cdot \left[\frac{1-f'}{p} \cdot \left(\left\lceil \frac{p}{m} \right\rceil - \left\lfloor \frac{p}{m} \right\rfloor - 1 \right) + \frac{1-f'}{m} + \frac{f'}{m} \right] \\
&= w \cdot \left[\frac{1-f'}{p} \cdot \left(\left\lceil \frac{p}{m} \right\rceil - \left\lfloor \frac{p}{m} \right\rfloor - 1 \right) + \frac{1-f'+f'}{m} \right]
\end{aligned}
\tag{B.2}
$$

da $p, m \in \mathbb{N}$ gilt $\left\lfloor \frac{p}{m} \right\rfloor = \left\lceil \frac{p}{m} \right\rceil - 1$ und es folgt

$$
\begin{aligned}
&= w \cdot \left[\frac{1-f'}{p} \cdot \left(\left\lceil \frac{p}{m} \right\rceil - \left\lceil \frac{p}{m} \right\rceil + 1 - 1 \right) + \frac{1}{m} \right] \\
&= w \cdot \left[\frac{1-f'}{p} \cdot 0 + \frac{1}{m} \right] \\
&= \frac{w}{m}
\end{aligned}
\tag{B.3}
$$

C Eigenschaften der betrachteten Beispielanwendungen

In diesem Abschnitt werden verschiedene Eigenschaften der in Abschnitt 5.1 vorgestellten Beispielanwendungen betrachtet. In Tabelle C.1 sind allgemeine Eigenschaften der Anwendungen aufgeführt. Es ist die Komplexität der Anwendungen als asymptotische obere Schranke angegeben [38, S. 47 f.]. n bezeichnet hierbei die Blockgröße bzw. die Anzahl der Eingabedaten. Bei RadixSort ist w die maximale Größe der zu sortierenden Zahlen. Des Weiteren ist die Anzahl der StreamIt-Filter der Anwendungen angegeben. Ein Beispiel einer StreamIt-Anwendung mit drei Filtern ist in Abbildung A.1 zu finden. Die theoretische Beschleunigung der Anwendungen für acht und 128 CPUs ist mithilfe des in Abschnitt 4.2 vorgestellten analytischen Modells bestimmt. Zudem ist das Verhältnis von Eingabe- und Ausgabedatenrate sowie das Verhältnis der Summe der internen Kommunikation (Task-zu-Task bzw. Filter-zu-Filter) und der Ausgabedatenrate angegeben. Von der Anzahl der Filter einer Anwendung kann nicht die Menge der internen Kommunikation abgeleitet werden. Die Anwendung BitonicSort verfügt über 275 Filter, besitzt jedoch nur ein Verhältnis von interner Kommunikation und Ausgabedatenrate von 18. 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.1: Allgemeine Eigenschaften der betrachteten Beispielanwendungen

Name	Komplexität	Filter-anzahl	theor. Beschleunig. 8 CPUs	128 CPUs	Verhältnis Ein-/Ausgabe	Verhältnis int. Komm./ Ausgabe
BitonicSort	$\mathcal{O}(n\log_2(n)^2)$	275	8,0	22	1	18
BubbleSort	$\mathcal{O}(n^2)$	66	7,7	64	1	65
DES	$\mathcal{O}(n)$	72	8,0	37	0,12	110
FFT	$\mathcal{O}(n\log_2(n))$	25	8,0	42	1	13
FIR-Filter	$\mathcal{O}(n)$	4	8,0	47	3	7
Filterbank	$\mathcal{O}(n)$	52	8,0	128	1	51
MatrixMult	$\mathcal{O}(n^3)$	35	6,8	7	2	36
RadixSort	$\mathcal{O}(n\log_2(w))$	14	8,0	21	1	13
SMA	$\mathcal{O}(n)$	4	8,0	14	1	3

In Tabelle C.2 sind verschiedene Eigenschaften der betrachteten Anwendungen dargestellt, die sich bei einer Partitionierung auf das CoreVA-MPSoC ergeben. Die Größe des benötigten Instruktionsspeichers (Größe des Programmabbilds) ist für eine einzelne CPU sowie für einen CPU-Cluster mit acht CPUs angegeben. Die Anwendungen FIR-Filter und SMA verfügen mit 12 kB bzw. 11 kB über die kleinsten Programmabbilder. BitonicSort und DES besitzen aufgrund einer Vielzahl an Filtern Programmabbilder der Größe 211 kB bzw. 137 kB. Hierbei muss angemerkt werden, dass im Rahmen des CoreVA-MSoC-Projekts zurzeit an einer Verringerung der Größe des Programmabbilds gearbeitet wird. So sind beispielsweise aktuell die Systembibliothek (siehe Abschnitt 3.2.1) sowie die Bibliothek mit dem Synchronisierungsfunktionen (siehe Abschnitt 2.6.4) für jede CPU (also achtmal) im CPU-Cluster vorhanden (siehe Abschnitt 5.7). Es sind die CPU-

Tabelle C.2: Eigenschaften der betrachteten Beispielanwendungen bei einer Abbildung auf das CoreVA-MPSoC

Name	Instr.-Speicher [kB] 1 CPU	8 CPUs	CPU-Takte pro Ausgabe	Anteil LD/ST	Anteil Kommunikation
BitonicSort	211	876	264	31,4 %	1,8 %
BubbleSort	68	612	2277	21,8 %	1,3 %
DES	137	681	1120	29,3 %	5,0 %
FFT	30	593	225	30,2 %	3,8 %
FIR-Filter	12	608	803	18,5 %	0,9 %
Filterbank	51	567	3066	24,3 %	1,0 %
MatrixMult	69	617	347	42,2 %	8,0 %
RadixSort	23	517	380	20,0 %	2,6 %
SMA	11	496	89	18,2 %	2,7 %

Takte angegeben, die benötigt werden, um ein Ausgabeelement bei der Ausführung auf einer 2-Slot-CPU zu produzieren. Zudem ist der Anteil an Speicherzugriffen bzw. LD/ST-Instruktionen an allen ausgeführten Instruktionen bei einer Abbildung auf einen CPU-Cluster mit acht CPUs angegeben. Des Weiteren ist der Anteil der Instruktionen für CPU-zu-CPU-Kommunikation an allen Instruktionen aufgeführt. Die Anwendung MatrixMult kommuniziert mit 42,2 % Speicherzugriffen und 8,0 % CPU-zu-CPU-Kommunikation deutlich mehr als die anderen betrachteten Anwendungen.

D Energiebedarf von Speicherzugriffen

In diesem Abschnitt ist der in Abschnitt 5.8.1 diskutierte Energiebedarf von Speicherzugriffen im CPU-Cluster des CoreVA-MPSoCs aufgeführt. Eine CPU im Cluster greift alle zehn Takte auf den eigenen lokalen L1-Datenspeicher (L1-L CPU0), den L1-Datenspeicher einer anderen CPU (L1-L CPU1) bzw. den gemeinsamen L1-Datenspeicher (gem. L1) zu. In den übrigen Takten werden Leerinstruktionen (NOPs) ausgeführt. Zudem ist der Energiebedarf von Leerinstruktionen (NOPs) aufgeführt.

Tabelle D.1: Energiebedarf von Speicherzugriffen eines CPU-Clusters mit zwei CPUs in pJ

Testmuster	L1-L CPU0		L1-L CPU1		Gem. L1		NOP
	LD	ST	LD	ST	LD	ST	
0x00000000	30,08	30,24	105,66	32,23	30,44	30,75	27,93
0x12345678	30,26	31,38	108,19	33,79	30,45	31,92	27,93
0x55555555	30,29	31,47	108,71	33,97	30,45	31,96	27,93
0xAAAAAAAA	30,29	31,50	108,71	33,97	30,45	31,80	27,93
0xFFFFFFFF	30,50	32,77	111,76	35,54	30,45	32,88	27,93

Tabelle D.2: Energiebedarf von Speicherzugriffen eines CPU-Clusters mit vier CPUs in pJ

Testmuster	L1-L CPU0		L1-L CPU1		Gem. L1		NOP
	LD	ST	LD	ST	LD	ST	
0x00000000	31,77	32,08	105,33	36,82	32,62	33,28	27,46
0x12345678	32,12	34,35	108,89	40,67	32,63	35,68	27,46
0x55555555	32,19	34,54	109,70	41,23	32,63	35,78	27,46
0xAAAAAAAA	32,18	34,59	109,60	41,19	32,63	35,44	27,46
0xFFFFFFFF	32,61	37,14	113,97	45,21	32,63	37,68	27,46