Exzellenzcluster
Kognitive Interaktionstechnologie
Kognitronik und Sensorik
Prof. Dr.-Ing. U. Rückert

Analyse verschiedener Architekturvarianten des CoreVA-VLIW-Prozessors

zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEUR (Dr.-Ing.)

der Technischen Fakultät
der Universität Bielefeld

genehmigte Dissertation

von

Dipl.-Ing. Boris Hübener

Referent: Prof. Dr.-Ing. Ulrich Rückert
Korreferent: Prof. Dr.-Ing. Peter Schulz

Tag der mündlichen Prüfung: 21.11.2016

Bielefeld / November 2016
DISS KS / 09

Kurzfassung

Die Leistungsfähigkeit eingebetteter Prozessoren gewinnt in der mobilen Signalverarbeitung aufgrund der stetig steigenden Anforderungen aktueller Multimediaanwendungen immer mehr an Bedeutung. Da mobile Systeme in den meisten Fällen batteriebetrieben sind, steht zur Ausführung dieser Anwendungen jedoch nur ein stark begrenztes Energiebudget zu Verfügung. Aus diesem Grund muss die Energieeffizienz, die sowohl von der Rechenleistung als auch von der Leistungsaufnahme eines Systems abhängt, so hoch wie möglich sein.

Um diesen Anforderungen gerecht zu werden, verfolgt die Arbeitsgruppe Kognitronik und Sensorik einen CPU-gestützten Ansatz auf Basis des 32-Bit VLIW-Prozessors CoreVA. Der CoreVA-Prozessor kann zur Entwurfszeit durch eine variable Anzahl parallel angeordneter Verarbeitungs- und Funktionseinheiten erweitert werden. Da sich durch das Hinzufügen der Verarbeitungseinheiten jedoch nicht nur die Rechenleistung des Prozessors sondern auch seine Ressourcenanforderungen erhöhen, müssen umfangreiche Entwurfsraumexplorationen durchgeführt werden, um diese Wechselwirkungen bewerten zu können. Dies ist angesichts der vielschichtigen Konfigurierbarkeit des CoreVA-Prozessors jedoch äußerst zeitaufwändig.

Vor diesem Hintergrund wurden in dieser Arbeit verschiedene Verfahren einer modellbasierten Entwurfsraumexploration entwickelt. Die Charakterisierung der Zielanwendungen erfolgt hierbei entweder auf Basis einzelner Simulationsdurchgänge des CoreVA-Instruktionssatzsimulators oder durch statische Programmcodeanalysen. Die Ermittlung der Ressourcenanforderungen geschieht mit Hilfe eines Hardwaremodells, das aus wenigen Probesynthesen gewonnen werden kann. Die Energie, die die jeweiligen Prozessorkonfigurationen zur Ausführung der Zielanwendungen benötigen, wird schließlich durch eine Kombination dieser Modelle bestimmt. Hierbei ergibt sich ein durchschnittlicher Approximationsfehler von etwa -5%.

Aufgrund dieses geringen Fehlers können die modellbasierten Verfahren die Entwurfsraumexploration des CoreVA-Prozessors maßgeblich beschleunigen. Für einen Prozessor mit bis zu vier Verarbeitungseinheiten lässt sich die Anzahl der genauer zu untersuchenden Prozessorkonfigurationen von 352 auf weniger als zehn Konfigurationen reduzieren. Bei einem Großteil der vorgestellten Anwendungen kann die modellbasierte Entwurfsraumexploration die energieeffizientesten Konfigurationen sogar direkt detektieren.

Abstract

Mobile computing devices require more and more computational power for the processing of state-of-the-art multimedia applications. On the other hand, the energy budget for the processing of those applications is strictly limited since mobile systems are battery powered devices. Therefore, the energy efficiency, which depends on the throughput and the power consumption, has to be as good as possible.

The Cognitronics and Sensor Systems Group tries to satisfy these requirements with a CPU-based approach that relies on the 32 bit VLIW processor architecture CoreVA. The CoreVA processor allows for a fine-grained configuration of the amount and characteristics of the processor's functional units at design time. Since those additional units increase the processor's throughput as well as it's resource requirements, a comprehensive design-space exploration is needed to evaluate the trade-off between these criteria. Due to the configurability of the CoreVA processor and the huge amount of different processor configurations, the analysis of the whole design space results in a very time consuming process.

This work introduces the development of several approaches for a model-based design-space exploration of the CoreVA processor. Af first, the target-applications are characterized by performing a single simulation run of the CoreVA's instruction-set simulator or by using static program code analyses. Afterwards, a hardware model is deployed by performing a few hardware syntheses. In the end, both models are combined in the evaluation of the energy, that is needed for the processing of those applications on different processor configurations. This approximation shows a mean error of about -5%.

This deviation it that small, that the design-space exploration of the CoreVA Processor can be notably accelerated. The amount of configurations that need to be analyzed for a processor with up to four VLIW issue slots can be decreased from 352 to less than ten configurations. The model-based design-space exploration is even able to find the most energy efficient configuration for the main portion of the examined applications directly.

Inhaltsverzeichnis

1 Einleitung

Eingebettete Prozessoren werden heutzutage in nahezu allen technischen Systemen eingesetzt. Sie dienen beispielsweise der Steuerung von Haushaltsgeräten oder kommen in mobiler Unterhaltung- und Kommunikationselektronik zum Einsatz. Aufgrund der immerwährenden Fortschritte, die insbesondere in dem Bereich der mobilen Signalverarbeitung erzielt werden konnten, wird jedoch immer mehr Rechenleistung benötigt, um den steigenden Datenraten gerecht zu werden und die aktuellen Multimediaanwendungen ausführen zu können. Des Weiteren müssen mobile Systeme eine Vielzahl unterschiedlicher Übertragungsverfahren unterstützen, um einen weltweiten Einsatz zu garantieren. Bisher konnte diese Leistungsfähigkeit in mobilen Systemen nur durch heterogene Hardware-Plattformen mit dedizierten Beschleunigerkomponenten erreicht werden. Da im alltäglichen Betrieb jedoch meist nur ein oder wenige Übertragungsverfahren zeitgleich benötigt werden und somit viele Funktionalitäten ungenutzt sind, setzen moderne Systeme zunehmend auf flexible Architekturen, die auf hochperformanten und gleichzeitig universellen oder rekonfigurierbaren Prozessoren basieren. Als beschränkender Faktor ist hierbei jedoch zu beachten, dass mobile Systeme in den meisten Fällen batteriebetrieben sind und deshalb nur ein stark begrenztes Energiebudget zur Verfügung haben [3, 69, 81].

Um diesen Anforderungen gerecht zu werden entwickelt die Arbeitsgruppe Kognitronik und Sensorik der Universität Bielefeld Verfahren, um die Energieeffizienz eingebetteter Prozessoren zu verbessern. Hierbei kommt der hochgradig konfigurierbare 32-Bit RISC[1]-Prozessor CoreVA zum Einsatz, der eine Vielzahl unterschiedlicher Möglichkeiten zur Erhöhung der Energieeffizienz bietet. So besitzt der CoreVA-Prozessor beispielsweise eine variable Anzahl paralleler Verarbeitungseinheiten, in denen mit Hilfe der VLIW[2]-Technik mehrere datenunabhängige Instruktionen gleichzeitig ausgeführt werden können. Des Weiteren kann der CoreVA in einen SIMD[3]-Modus versetzt werden, wodurch sogar mehrere Berechnungen pro Verarbeitungseinheit stattfinden können. Um den CoreVA-Prozessor möglichst gut an die Charakteristika der jeweiligen Anwendungsbereiche anpassen zu können,

[1] Reduced Instruction Set Computer
[2] Very Long Instruction Word
[3] Single Instruction Multiple Data

lassen sich diese Verarbeitungseinheiten zur Entwurfszeit jeweils durch dedizierte Multiplizier-, Dividier- und Load/Store-Einheiten erweitern [89].

Da durch das Hinzufügen zusätzlicher Verarbeitungs- und Funktionseinheiten neben der Rechenleistung jedoch in den meisten Fällen auch die Leistungsaufnahme und Chipfläche des jeweiligen Prozessors ansteigt, müssen umfangreiche Entwurfsraumexplorationen durchgeführt werden, um die Prozessorkonfiguration zu spezifizieren, die die benötigte Leistungsfähigkeit bei möglichst geringem Ressourcenaufwand erreicht. Aufgrund der vielschichtigen Konfigurierbarkeit des CoreVA-Prozessors müssten für eine vollständige Untersuchung hierzu viele tausend Prozessorkonfigurationen betrachtet werden. Da dies praktisch nicht möglich ist, werden im Rahmen dieser Arbeit verschiedene modellbasierte Verfahren entwickelt, mit deren Hilfe die Anzahl der zu untersuchenden Konfigurationen mit geringem Arbeitsaufwand auf eine handhabbare Größe eingeschränkt werden kann. Dieser Schritt wird zwingend benötigt, da der CoreVA-Prozessor in folgenden interdisziplinären Projekten sowohl als anwendungsspezifischer eingebetteter Prozessor, als auch als Basiskomponente für das derzeit in der Entwicklung befindliche Multiprozessorsystem CoreVA-MPSoC[4] eingesetzt werden soll [89, 61, 73].

Diese Arbeit beginnt mit einem Überblick über den Stand der Technik aktueller eingebetteter Prozessoren (siehe Kapitel 2) und stellt anschließend den CoreVA-Prozessor und seine angegliederte Werkzeugkette vor (siehe Kapitel 3 und 4). In Kapitel 5 und 6 folgt eine Bewertung der derzeit in der Forschung angewandten Verfahren zur Entwurfsraumexploration sowie die Vorstellung der herangezogenen Beispielanwendungen. Anschließend folgt in den Kapiteln 7, 8 und 9 eine detaillierte Beschreibung der Soft- und Hardwaremodellierung sowie der hierauf beruhenden modellbasierten Entwurfsraumexplorationen. Kapitel 7.3 zeigt hierbei die Entwicklung eines Verfahren, mit dessen Hilfe die Laufzeit einer Anwendung auf Basis einer einzelnen konfigurationsunabhängigen Simulation approximiert werden kann. In Kapitel 7.4 folgt ein Modellierungsansatz, bei dem die durchschnittliche Parallelität und die Instruktionsverteilung der jeweiligen Zielanwendungen auf Basis einer statischen Programmcodeanalyse bestimmt werden. Die Modellierung der Leistungsaufnahme und Prozessorfläche erfolgt anhand der in Kapitel 8.3 und 8.4 ermittelten Ressourcenanforderungen. Zur modellbasierten Entwurfsraumexploration werden schließlich drei verschiedene Herangehensweisen vorgestellt (siehe Kapitel 9.1, 9.2 und 9.3). Zum Einen wird die Energie betrachtet, die zur Ausführung der Zielanwendungen auf den jeweiligen Prozessorkonfigurationen benötigt wird. Zum Anderen werden die approximierten Leistungsaufnahmen und Laufzeiten in ein Pareto-Diagramm übertragen, um die effizientesten Konfigurationen in verschiedenen Leistungsgruppen zu finden. In einem letzten Schritt wird durch das direkte Gegenüberstellen der Leistungssteigerungen und Ressourcenmehraufwände ein

[4] Multiprocessor System on Chip

Verfahren vorgestellt, dass eine Einschränkung der Konfigurationsmenge auch ohne Kenntnis der tatsächlichen Laufzeiten ermöglicht. Die Bewertung der jeweiligen Modellierungsverfahren erfolgt anhand von Vergleichen mit den Ergebnissen zahlreicher konfigurationsspezifischer Simulationen und Synthesen. In Kapitel 10 folgt eine abschließende Gegenüberstellung des CoreVA-Prozessors mit einer Vielzahl aktuell am Markt erhältlicher Prozessorsysteme.

2 Stand der Technik eingebetteter Prozessorsysteme

Dieses Kapitel gibt einen Überblick über die aktuell am Markt erhältlichen eingebetteten Prozessoren, sowie über einige Prozessorsysteme, die derzeit zu Forschungszwecken genutzt werden. Der Fokus dieser Gegenüberstellung liegt auf Prozessoren, die in mobilen Systemen oder zur Steuerung technischer Geräte eingesetzt werden und Parallelisierungsverfahren wie VLIW oder SIMD unterstützen. Ausgewiesene Hochleistungsprozessoren werden im Rahmen dieser Arbeit nicht bewertet, da sie aufgrund ihrer völlig anderen Charakteristika nicht mit dem CoreVA-Prozessor vergleichbar sind.

Die Werte, die im Folgenden zum Vergleich der Rechenleistung und der Leistungsaufnahme herangezogen werden, sind auf die Betriebsfrequenzen der jeweiligen Prozessoren normiert. Die Leistungsfähigkeit wird anhand des in Kapitel 6.2 beschriebenen Coremark- und Dhrystone-Benchmarks bestimmt und folglich in Coremarks/MHz und DMIPS/MHz angegeben.

2.1 Die ARM Cortex-Prozessoren

Derzeit basieren die meisten eingebetteten Prozessorsysteme auf IP-Cores[1] der britischen Firma ARM[2] Limited. Abhängig von ihrem späteren Einsatzgebiet werden die ARM-Prozessoren in die Gruppen der Anwendungsprozessoren (Cortex A-Serie), Echtzeitprozessoren[3] (Cortex R-Serie) und Mikrocontroller (Cortex M-Serie) unterteilt [35].

Die Prozessoren der Cortex A-Serie sind wegen ihrer Energieeffizienz und Leistungsfähigkeit heutzutage in einer Vielzahl unterschiedlicher Smartphones und

[1] Intellectual Property Cores, vorgefertigte synthetisierbare Funktionsblöcke
[2] Advanced RISC Machine
[3] Real-Time Processor

Tabletcomputer vertreten. Sie basieren entweder auf der 32-Bit RISC-Architektur ARMv7 oder auf der 2013 hinzugekommenen 64-Bit Architektur ARMv8 und werden größtenteils als superskalare Prozessoren mit bis zu drei parallelen Verarbeitungseinheiten angeboten. Diese Verarbeitungseinheiten können in Abhängigkeit von dem geforderten Funktionsumfang durch eine 128-Bit SIMD-Erweiterung namens NEON und durch Fließkommarecheneinheiten ergänzt werden. Des Weiteren bieten die Prozessoren diverse Schnittstellen, um in einem Multiprozessorsystem eingesetzt zu werden.

Der ARM-Prozessor mit der geringsten Leistungsfähigkeit in dieser Serie ist der seit 2009 am Markt erhältliche Cortex A5. Dieser Prozessor besitzt eine einzelne 32-Bit Verarbeitungseinheit und wurde lange Zeit in preiswerten Mobiltelefonen eingesetzt. Da die Rechenleistung des A5 jedoch schon damals für viele Anwendungsszenarien nicht ausreichend war, wurde zeitgleich der Cortex A9 konzipiert, der sich in den folgenden Jahren zum führenden Prozessor in Standard-Smartphones entwickelt hat (beispielsweise als Exynos 4 Prozessor im Galaxy S3 oder als Apple A5 Prozessor im iPhone 4S). Der A9 besitzt durch seine zwei Verarbeitungseinheiten bei der Ausführung des Coremark- und Dhrystone-Benchmarks eine um 26,09% und 59,24% höhere Rechenleistung als der Cortex A5. Diese Rechenleistung kann jedoch nur durch eine fast vierfach höhere Leistungsaufnahme erreicht werden (siehe Tabelle 10.2 und 10.1) [21, 45, 46].

Ab 2011 wurden die Prozessoren A5 und A9 sukzessive durch die Prozessoren A7 und A15 ersetzt. Die Leistungsfähigkeit des A7 konnte hierbei im Vergleich zum A5 mit +13,04% für den Coremark- und +24,20% für den Dhrystone-Benchmark deutlich gesteigert werden. Die Leistungsaufnahme erhöht sich lediglich um 33,33%. Die Rechenleistung des A15 liegt dank seiner dritten Verarbeitungseinheit um 26,92% und 79,49% über der Leistungsfähigkeit des Cortex A7. Da diese Werte jedoch nur durch eine Verdoppelung der Pipelinestufen erreicht werden können, weist der A15 im Vergleich zum A7 nun eine fünffach höhere Leistungsaufnahme auf [21, 33, 45].

Da die hohe Leistungsaufnahme des Cortex A15 die Betriebsdauer mobiler Systeme jedoch stark einschränkt, wurde das „Big.Little“-Konzept eingeführt, dass das Verschalten mehrerer A7 und A15 Prozessoren in einem Multiprozessorsystem ermöglicht. Anwendungen, die kurzzeitig hohe Rechenlasten erzeugen, können somit durch die A15 Prozessoren verarbeitet werden. Sobald die Rechenlast sinkt, können die A15 Prozessoren deaktiviert werden und die sparsameren A7 übernehmen die Ausführung der verbleibenden Hintergrundanwendungen. Da das Big.Little-Konzept durch das Verschalten mehrerer Prozessoren jedoch eine vergleichsweise hohe Prozessorfläche benötigt (ein einzelner A15 benötigt bereits 86,67% mehr Fläche als der A9) kann dieses Konzept aus Kostengründen nur in Hochleistungssystemen eingesetzt werden (beispielsweise im Exynos 5 Octa Prozessor des Samsung Galaxy S4). Für mobile Systeme des mittleren Preissegments wurde die Cortex A-

Serie daher ab 2014 durch den Cortex A12 ergänzt, dessen Leistungsfähigkeit mit 3,50 Coremarks/MHz und 3,00 DMIPS/MHz zwischen der des A7 und des A15 liegt [21, 33, 44].

Im Unterschied zu den Prozessorsystemen der Cortex A-Serie werden die Prozessoren der Cortex M-Serie als Mikrocontroller zur Steuerung technischer Geräte eingesetzt. Da in diesem Umfeld nur eine sehr geringe Rechenleistung benötigt wird, können diese Prozessoren deutlich energieeffizienter konzipiert werden. Die Zielbetriebsfrequenz liegt beispielsweise weit unter 500 MHz. Zudem wird in den meisten Fällen auf einen Cache und auf SIMD-Erweiterungen oder Fließkommarecheneinheiten verzichtet. Durch eine auf zwei oder drei Stufen begrenzte Prozessorpipeline kann der Einsatz von Sprungvorhersagemechanismen ebenfalls entfallen. Des Weiteren unterstützen Cortex M-Prozessoren nicht den vollständigen ARMv6 oder ARMv7 Instruktionssatz, sondern einen deutlich eingeschränkteren ARMv6-M beziehungsweise ARMv7-M Befehlssatz. Der 2009 erschienene Cortex M0 und sein Nachfolger M0+ (2012) bilden derzeit die Prozessoren mit dem geringsten Funktionsumfang. In ihrer Grundkonfiguration ermöglichen sie beispielsweise 32x32-Bit Multiplikationen nur mit einer Latenz von 32 Rechentakten. Durch diesen Minimalismus erreicht der M0+ bei der Verarbeitung des Coremark im Vergleich zum Cortex A12 zwar nur eine Rechenleistung von 70,29%, diese Rechenleistung erreicht er jedoch bei nur 2,45% der Leistungsaufnahme (siehe Tabelle 10.2 und 10.1) [14, 22, 30, 36].

2.2 Die MIPS Aptiv- und Warrior-Prozessoren

Die Firma Imagination Technologies, vormals MIPS[4] Technologies Inc., vertreibt ebenfalls IP-Cores für eingebettete Prozessorsysteme verschiedenster Größenordnungen. Die angebotenen Prozessoren basieren hierbei entweder auf den 32-Bit RISC-Instruktionssätzen MIPS32 und microMIPS, oder auf dem deutlich leistungsfähigeren 64-Bit Instruktionssatz MIPS64. Bis Dezember 2012 wurden diese Instruktionssätze in der Version R3 angeboten. Seitdem werden sie auch in einer Version R5 bereitgestellt, die unter anderem eine deutlich erweiterte SIMD-Funktionalität bietet.

Die aktuellsten IP-Cores der Version R3 bilden die 2012 veröffentlichten Prozessoren der Aptiv-Serie. Diese Serie fasst die Vielzahl der bis dahin verfügbaren Prozessorvarianten erstmals in drei getrennten Klassen mit unterschiedlicher Leistungsfähigkeit zusammen. In Anlehnung an die Namensgebung der ARM-Prozessoren wurden diese Klassen MicroAptiv (Mikrocontroller), InterAptiv (mittlere Leistungsklasse) und

[4] Microprocessor without Interlocked Pipeline Stages

ProAptiv (Hochleistungsprozessoren) genannt. Auf Basis der R5-Instruktionssätze wurde im August 2013 schließlich die Warrior-Prozessorserie entwickelt, die wie die Aptiv-Serie Prozessoren in den drei Leistungsklassen Warrior-M, Warrior-I und Warrior-P enthält [31, 32].

Die Prozessoren MicroAptiv und Warrior-M sind zwei zum ARM Cortex M vergleichbare 32-Bit RISC-Prozessoren, die als Mikrocontroller in tief eingebetteten Systemen eingesetzt werden sollen. Bis auf kleine Designunterschiede, die aus den Erweiterungen des Instruktionssatzes resultieren, haben diese Prozessoren einen identischen Aufbau. Sie basieren beide auf einer fünfstufigen Prozessorpipeline und besitzen eine gewisse Konfigurierbarkeit. Neben den fixen Komponenten wie dem Registerfile[5], der Verarbeitungseinheit oder der Multiplizier- und Dividiereinheit können sie durch zusätzliche Funktionseinheiten erweitert werden. Hierzu zählt beispielsweise eine Vektorrecheneinheit, die das Ausführen zweifacher 16-Bit oder vierfacher 8-Bit SIMD-Instruktionen ermöglicht. Des Weiteren lässt sich die Funktionalität des Multiplizierers so erweitern, dass zwei 16x16-Bit oder eine 32x32-Bit MAC[6]-Berechnungen innerhalb eines Prozessortaktes durchgeführt werden können. Weitere optionale Komponenten bilden eine 32-Bit Fließkommarecheneinheit und diverse Bus-Schnittstellen. Die Prozessoren werden jeweils als Mikrocontroller in einer Cache-losen Version angeboten (MicroAptiv MCU, Warrior-M5100), oder in einer Version mit einem bis zu 64 KByte großen Instruktions-und Datencache (MicroAptiv MPU, Warrior-M5150) [23, 31].

2.3 Die Cadence Xtensa Dataplane-Prozessoren

Seit der Übernahme der Firma Tensilica im April 2013 bietet Cadence seinen Kunden neben den in Kapitel 4 beschrieben Entwicklungswerkzeugen nun auch den individuell konfigurierbaren Signalverarbeitungsprozessor Xtensa an. Ein Vorteil dieses Prozessors liegt in der Kombination eines universell einsetzbaren 32-Bit RISC-CPU[7] mit speziellen Prozessorelementen zur Leistungssteigerung bei der Verarbeitung von Signalverarbeitungsalgorithmen. Diese sogenannten DSP[8]-Elemente besitzen zwar nur einen sehr eingeschränkten Instruktionsumfang, ihr Durchsatz liegt durch den Einsatz von massiv parallelen SIMD-Feldern jedoch weit über dem Durchsatz des RISC-Prozessors. Als Beispiel für die verfügbaren DSP-Elemente seien hier die auf die Basisbandsignalverarbeitung optimierten Co-Prozessoren der ConnX-BBE Reihe genannt. Diese Co-Prozessoren sind in der Lage bis zu 64 SIMD-

[5] zentraler Registerspeicher
[6] Multiply-Accumulate, Multiplizier- und Akkumulierberechnungen der Form $A = B \cdot C + D$
[7] Central Processing Unit, Hauptprozessor
[8] Digital Signal Processor

Berechnungen gleichzeitig durchzuführen. In Kombination mit speziellen Vektor-Multipliziereinheiten können diese Co-Prozessoren sogar 64 MAC-Instruktionen pro Rechentakt verarbeiten [11, 24, 29].

Mit Hilfe der Eclipse-basierten Entwicklungsumgebung Xtensa Xplorer kann ein Hardwareentwickler neben der Verhaltensbeschreibung der jeweiligen Prozessorkonfigurationen automatisch eine konfigurationsspezifische Werkzeugkette inklusive eines angepassten Compilers (Übersetzers) und Instruktionssatzsimulators erzeugen. Hierbei kann er in einem ersten Schritt die diversen Konfigurationselemente des RISC-Prozessors einstellen, wie beispielsweise die Anzahl der Pipelinestufen und die Leistungsfähigkeit der Multipliziereinheiten. Anschließend kann das System gegebenenfalls durch die oben genannten DSP-Elemente oder durch eigene Instruktionssatzerweiterungen ergänzt werden. Diese werden ebenfalls automatisch in die Werkzeugkette integriert und in die Verhaltensbeschreibung des Prozessors übertragen [11, 24, 29].

2.4 Die Synopsys ARC-Prozessoren

Die amerikanische Firma Synopsys ist neben Cadence ein weiterer Anbieter von Programmen zur automatisierten Entwicklung elektronischer Halbleitersysteme. Seit der Übernahme der Firma Virage Logic im Jahre 2010 ist Synopsys nun ebenfalls in der Lage, seinen Kunden mit der ARC[9]-EM- und ARC-HS-Reihe komplette Prozessorsysteme zur Verfügung stellen zu können.

Die Prozessoren der EM-Reihe basieren auf einer dreistufigen Pipelinestruktur und weisen mit 8,4 μW/MHz eine sehr geringe Leistungsaufnahme auf (ARC-EM4). Da die Rechenleistung dieser Prozessoren mit 2,29 Coremarks/MHz und 1,52 DMIPS/MHz jedoch vergleichsweise begrenzt ist, werden sie hauptsächlich als tief eingebettete Systeme eingesetzt und dienen beispielsweise zur Steuerung von Speichern oder Sensorknoten. Dem gegenüber können die Prozessoren der HS-Reihe als vollwertige DSP-Systeme angesehen werden. Sie besitzen eine zehnstufige Prozessorpipeline und können zur echtzeitfähigen Steuerung von SSD[10]-Festplatten und NAS[11]-Systemen oder für die Signalverarbeitung in Digitalkameras und Tabletcomputern genutzt werden. Die Leistungsfähigkeit des ARC-HS34 liegt bei der Verarbeitung des Coremark- beziehungsweise Dhrystone-Benchmarks um 48,47% und 29,97% über der Leistungsfähigkeit des ARC-EM4. Die Leistungsaufnahme des ARC-HS34 ist jedoch mit 33 μW/MHz fast viermal größer [30, 38].

[9] Argonaut RISC Core
[10] Solid State Drive, Halbleiterlaufwerk
[11] Network Attached Storage, netzgebundener Speicher

Wie der Großteil der hier vorgestellten Prozessoren basieren auch die ARC-Prozessoren auf einer konfigurierbaren 32-Bit RISC-Architektur. Das Programm ARChitect ermöglicht hierbei dem Hardwareentwickler, die Prozessoren nach seinen Wünschen anzupassen. Er kann beispielsweise die Größen der Speicher und des Registerfiles festlegen oder die Prozessoren mit optionalen Multiplizierern, Dividierern, Schleifenzählern, Interrupt-Controllern und Fließkommarecheneinheiten ausstatten. Neben diesen grundlegenden Entscheidungen kann der Entwickler bei der Erstellung anwendungsspezifischer Prozessoren in den meisten Ebenen noch weitaus tiefer ins Detail gehen. So kann er bei der Konfiguration des Registerfiles beispielsweise festlegen, ob die Register durch schnelle Flip-Flop-Schaltungen oder platzsparende Speicherzellen abgebildet werden und ob ein oder zwei Schreibports implementiert werden sollen. Des Weiteren ermöglicht das ARChitect Programm, den Prozessorkern durch Instruktionssatzerweiterungen zu ergänzen oder Hardwarebeschleuniger als externe Co-Prozessoren anzubinden. Die Auswirkungen der Prozessorkonfigurationen sowie der Instruktionssatzerweiterungen und Hardwarebeschleuniger können frühzeitig mit Hilfe von Instruktionssatzsimulatoren bestimmt werden. Hierbei besteht die Wahl zwischen dem schnellen Simulator nSIM, der die Laufzeit nur grob abschätzen kann, oder dem zyklenakkuraten Simulator xCAM [30, 38].

2.5 Der Aeroflex Gaisler LEON-Prozessor

Der LEON3-Mikroprozessor entstammt der seit 1997 im Auftrag der europäischen Weltraumorganisation ESA[12] vorangetriebenen Entwicklung von Prozessoren für den Einsatz im Weltraum. Ursprünglich vom europäischen Weltraumforschungs- und Technologiezentrum ESTEC[13] implementiert, erfolgt die Weiterentwicklung und der Vertrieb dieser Prozessorfamilie nun durch das schwedischen Unternehmen Aeroflex Gaisler [6].

Der LEON3 basiert ebenfalls auf einer modular erweiterbaren 32-Bit RISC-Prozessorarchitektur. Er besitzt eine siebenstufige Prozessorpipeline und unterstützt den SPARC[14]-Instruktionssatz in der Version 8. Die Verhaltensbeschreibung des LEON3 steht unter der GNU General Public License (GPL) und ist somit für nichtkommerzielle Zwecke frei verfügbar. Aeroflex Gaisler bietet den LEON3 in Verbindung mit der Gaisler Research IP Library (GRLIB) an. Mit Hilfe dieser Bibliothek ist der Entwickler in der Lage, den Prozessorkern des LEON3 anwendungsspezifisch anzupassen. Der Entwickler kann beispielsweise den Funktionsumfang der implementierten Multiplizier- und Dividiereinheiten festlegen und die Größe der Caches

[12] European Space Agency
[13] European Space Research and Technology Centre
[14] Scalable Processor ARCitecture

einstellen. Des Weiteren kann er den Prozessorkern mit optionalen Komponenten wie Interrupt-Controllern, Hardwarebeschleunigern oder Fließkommarecheneinheiten ausstatten. Neben diesen Erweiterungen existiert eine Vielzahl externer Co-Prozessoren, die über AMBA AHB[15] 2.0 Busse an den Prozessorkern angeschlossen werden können [27, 28].

Die optionalen Komponenten und Co-Prozessoren sind ebenfalls größtenteils frei verfügbar. Lediglich die Fließkommarecheneinheiten und die speziell für den Einsatz in strahlungsintensivem Umfeld konzipierte fehlertolerante Version des LEON3 sowie der seit 2010 verfügbare LEON4 werden unter kommerzieller Lizenz vertrieben. Zur Programmentwicklung und zur Konfiguration des Prozessors bietet Aeroflex Gaisler eine frei verfügbare Entwicklungsumgebung inklusive Compiler (BCC), Simulator (TSIM) und Debugger (GRMON) [27, 28].

2.6 Der ρ-VEX-Prozessor

Das VEX[16]-Prozessorsystem wurde ursprünglich von Fisher [25] zusammengestellt, um als Anschauungsobjekt in einem Lehrbuch über eingebettete Computersysteme zu dienen. Das System basiert auf drei Hauptkomponenten, dem VEX-Instruktionssatz, einem angepassten C-Compiler und einem Instruktionssatzsimulator.

Der VEX-Instruktionssatz beschreibt einen hochgradig skalier- und konfigurierbaren 32-Bit VLIW-Prozessor. Er basiert auf einer zu Lehrzwecken vereinfachten Version des Instruktionssatzes des ST200 Prozessors, der in einem Zusammenschluss der Firmen Hewlett-Packard und STMicroelectronics entwickelt und vertrieben wurde. Die ebenfalls konfigurierbare Entwicklungsumgebung auf Basis des Lx/ST200 C-Compilers und des zugehörigen Instruktionssatzsimulators wurde ebenfalls von Hewlett-Packard zur Verfügung gestellt [25].

Die technische Realisierung des VEX-Prozessors erfolgt im Rahmen des quelloffenen Projekts ρ-VEX[17], welches auf eine prototypische Implementierung des VEX-Prozessors auf FPGAs[18] abzielt. Der ρ-VEX-Prozessors besitzt in seiner derzeitigen Standardkonfiguration eine fünfstufige Prozessorpipeline mit vier parallel angeordneten Verarbeitungseinheiten. Neben den stets implementierten 32-Bit ALUs[19] sind zwei dieser Verarbeitungseinheiten mit zusätzlichen 16x32-Bit Multiplizierern

[15] Advanced Microcontroller Bus Architecture, Advanced High-performance Bus
[16] VLIW-Example
[17] Reconfigurable VEX, rekonfigurierbarer VEX
[18] Field Programmable Gate Array, feldprogrammierbare Gatteranordnung
[19] Arithmetic Logic Unit, arithmetisch-logische Verarbeitungseinheit

und eine dritte mit einer dedizierten Load/Store-Einheit ausgestattet. Die vierte Verarbeitungseinheit steuert die Sprungverarbeitung, die sowohl in der Lage ist bedingte Sprünge auszuführen, als auch Sprungvorhersagen zu betreiben. Das Registerfile des ρ-VEX besteht aus 64 32-Bit Datenregistern zur Zwischenspeicherung der benötigten Operanden und acht 1-Bit Registern zum Speichern der Sprungbedingungen. Um das Verarbeiten aufeinander folgender Instruktionen zu beschleunigen, besitzt der ρ-VEX-Prozessor diverse Bypasspfade zur schnellen Rückführung dieser Sprungbedingungen und der Rechenergebnisse der jeweiligen Verarbeitungseinheiten [25, 75].

Neben dieser Standardkonfiguration bietet das ρ-VEX-System die Möglichkeit, neun Parameter des Prozessorkerns und neun Parameter der Prozessorumgebung zu variieren. Hierzu zählt beispielsweise Anzahl der parallelen Verarbeitungs-, Multiplizier- und Load/Store-Einheiten und die Größe des Registerfiles. Des Weiteren ist die Anzahl der vorhandenen Bypasspfade sowie die Größe des Daten- und Instruktionscaches einstellbar. Auf diese Weise können VLIW-Prozessoren mit bis zu 32 parallelen Verarbeitungseinheiten erzeugt werden. Diese Größe wird jedoch nur in rein simulationsbasierten Analysen genutzt. Prototypische Implementierungen beschränken sich derzeit auf Prozessoren mit bis zu acht parallelen Verarbeitungseinheiten [25, 75].

Aufgrund der hohen Konfigurierbarkeit des Prozessors und der Flexibilität seiner Entwicklungsumgebung wird der ρ-VEX in zahlreichen Untersuchungen zu Entwurfsraumexplorationen eingesetzt. Hier seien beispielsweise die in Kapitel 5 vorgestellten Arbeiten von Saptono [73], Seedorf [75] und Wong [83] genannt.

2.7 Der Qualcomm Hexagon-Prozessor

Der Hexagon ist ein VLIW-Prozessor der Firma Qualcomm. Er zählt zu den CPU-DSP-Hybriden und kann sowohl als Hauptprozessor in eingebetteten Systemen, als auch als spezialisierter Signalverarbeitungsprozessor in Multiprozessorsystemen eingesetzt werden. In seiner Funktion als Signalverarbeitungsprozessor findet er in zweifacher Ausführung in Qualcomms mobilem Multiprozessorsystem Snapdragon 800 Verwendung, um dort die ARM-basierten Krait-Prozessoren bei der Sprach-, Bild- und Videoverarbeitung zu unterstützen (Hexagon aDSP v5) und die Modem-Funktionalität zu übernehmen (Hexagon mDSP v5) [17, 34].

Der Hexagon-Prozessor besitzt vier Verarbeitungseinheiten und ist somit in der Lage, pro Rechentakt bis zu vier voneinander unabhängige Instruktionen auszuführen. Zwei der vier Verarbeitungseinheiten besitzen jeweils eine Load/Store-Einheit, um

Zugriffe auf den Datencache zu ermöglichen, sowie ein Rechenwerk zur Adressberechnung und zur Ausführung einfacher 32-Bit Operationen. Die zwei verbleibenden Verarbeitungseinheiten sind als 64-Bit SIMD-Einheiten ausgelegt, die alle gängigen arithmetisch-logischen Operationen, sowie eine Vielzahl verschiedener Multiplizierinstruktionen unterstützen. Die Datenbreite kann hierbei zwischen 8, 16, 32 oder 64 Bit variiert werden. Der Hexagon-Prozessor kann dementsprechend acht 8-Bit Berechnungen, vier 16-Bit Berechnungen, zwei 32-Bit Berechnungen oder eine 64-Bit Berechnung pro SIMD-Einheit ausführen. Die eingebundenen Multiplizierereinheiten sind in der Lage, entweder vier 16x16-Bit Multiplikationen, zwei 32x16-Bit Multiplikationen oder eine 32x32-Bit Multiplikation pro SIMD-Verarbeitungseinheit ausführen [17, 34].

Neben den Standardberechnungen unterstützt der Hexagon diverse Instruktionen, die speziell auf die Kernalgorithmen der digitalen Signalverarbeitung ausgelegt sind. Hierzu zählt beispielsweise eine Instruktion zur Multiplikation zweier komplexwertiger 16-Bit Operanden. Diese Spezialinstruktion ersetzt vier Multiplikationen, vier Schiebeoperationen, vier Additionen und zwei Operationen zum mathematischen Runden des Endergebnisses. Um die hierfür erforderlichen Eingangsdaten bereitstellen zu können, besitzt der Hexagon ein Registerfile mit 32 32-Bit Datenregistern, die einzeln oder paarweise als ein 64-Bit Register angesprochen werden können [17, 34].

2.8 Der CEA-Leti Mephisto-Prozessor

Ein weiterer erwähnenswerter Entwickler anwendungsspezifischer VLIW-Prozessoren ist das Forschungsinstitut für Elektronik und Informationstechnologie CEA[20]-Leti[21] in Grenoble. Dieses Institut hat im Jahre 2011 den Mephisto-Prozessor vorgestellt, dessen Haupteinsatzgebiet in der Basisbandsignalverarbeitung aktueller Software-defined Radio Anwendungen liegt (siehe Kapitel 6) [10, 49].

Da die Rechenlast in diesem Anwendungsbereich hauptsächlich durch Matrizenmultiplikationen und diskrete Fouriertransformationen erzeugt wird, wurde bei der Entwicklung des Mephisto-Prozessors ein besonderer Augenmerk auf eine energieeffiziente Verarbeitung komplexwertiger Multiply-Accumulate-Operationen gelegt. Durch dieses Vorgehen wurde der Instruktionssatz des Mephisto jedoch so stark spezialisiert, dass der Prozessor praktisch keine universell einsetzbaren Verarbeitungseinheiten mehr besitzt und somit eher als DSP-Erweiterung, denn

[20] Commissariat à l'énergie atomique et aux énergies alternatives
[21] Laboratoire d'electronique des technologies de l'information

als eigenständiger VLIW-Prozessor zu betrachten ist. Der Entwickler nennt den Mephisto in diesem Zusammenhang daher auch einen ASIP[22]-Prozessor, der Spezialaufgaben in einem Multiprozessorsystem übernimmt [10, 49].

Die zehnstufige Pipeline des Mephisto-Prozessors ist maßgeblich durch vier 16x16-Bit Multiplizierereinheiten geprägt, die in Kombination mit zwei Akkumulationseinheiten und zwei Einheiten zur Bildung von Zweierkomplementen, eine komplexwertige 16x16-Bit MAC-Berechnung pro Takt ermöglichen. Neben dieser MAC-Einheit besitzt der Mephisto eine Funktionseinheit zur Berechnung trigonometrischer Funktionen (Cordic-Einheit) sowie einen Komparator, der beispielsweise zur Abbildung von Viterbi-Dekodierern verwendet werden kann. Auf den Einsatz von SIMD-Erweiterungen wird verzichtet, da diese von den Entwicklern als zu unflexibel erachtet werden [10, 49].

Die zu verarbeitenden Daten können entweder dem 64x32-Bit großen Registerfile entnommen werden, oder über zwei dedizierte Load/Store-Einheiten aus einem 8 KByte großen Datenspeicher gelesen werden. Der Instruktionsspeicher des Mephisto-Prozessors besitzt ebenfalls eine Größe von 8 KByte. Er wird zusätzlich durch spezielle Instruktionswortregister unterstützt, in die besonders häufig vorkommende Instruktionswortteile geladen werden. Das Instruktionswort kann hierdurch sehr komprimiert dargestellt werden, da anstelle der hinterlegten Instruktionswortteile nur die Registeradressen gespeichert werden müssen [10, 49].

2.9 Zusammenfassung und Eignungsbewertung

In diesem Kapitel wurde ein Überblick über den Stand der Technik aktueller eingebetteter Prozessoren gegeben. Die Untersuchung hat gezeigt, dass eingebettete Prozessoren häufig mehrere Verarbeitungseinheiten besitzen, die durch die VLIW-Technik oder mit Hilfe eines superskalaren Ansatzes mehrere datenunabhängige Instruktionen gleichzeitig ausführen können (siehe abschließende Gegenüberstellung in Kapitel 10). Ergänzt werden die Verarbeitungseinheiten durch spezielle Multiplizierer, durch SIMD-Rechenfelder oder durch Instruktionssatzerweiterungen und Hardwarebeschleuniger, die zur Leistungssteigerung bei der Berechnung von Signalverarbeitungsalgorithmen eingesetzt werden.

Manche Anbieter, wie beispielsweise ARM und MIPS, setzen auf ein breites Produktportfolio, um unterschiedliche Leistungsanforderungen bedienen zu können. Der Großteil der vorgestellten Anbieter benutzt jedoch konfigurierbare Systeme, um die

[22] Application Specific Instructionset Processor, Prozessor mit anwendungsspezifischem Befehlssatz

Prozessoren an die Kundenwünsche anzupassen. So lässt sich in den meisten Fällen die Anzahl der Verarbeitungseinheiten und der zugehörigen Multiplizier- und Load/Store-Einheiten vorgeben, sowie die Größe der Speicher und des Registerfiles variieren. Ein besonderer Augenmerk sei an dieser Stelle auf den ρ-VEX-Prozessor gelegt, der über achtzehn Konfigurationsparameter verfügt und bis zu 32 parallele Verarbeitungseinheiten besitzen kann.

Wie bereits in Kapitel 1 beschrieben wurde, werden in dieser Arbeit verschiedene Architekturvarianten eines VLIW-Prozessors untersucht, der anschließend als eingebettetes System in einem Standardzellen-ASIC[23] Verwendung finden soll. Da hierzu die Verhaltensbeschreibung des Prozessorkerns analysiert und angepasst werden muss, was von den meisten Entwicklerfirmen unterbunden wird, scheidet ein Einsatz kommerzieller Systeme aus. Die nicht-kommerziellen Anbieter gewähren zwar Einblick in die Verhaltensbeschreibung ihrer Prozessoren, da jedoch die wenigsten Forschungseinrichtungen Standardzellen-ASICs fertigen, sind die quelloffenen Systeme auf eine prototypische Implementierung auf FPGAs ausgelegt. Verhaltensbeschreibungen von FPGA-Implementierungen lassen sich wiederum nicht effizient auf ASICs abbilden, da sie eine Vielzahl FPGA-eigener Baugruppen wie spezielle Speicherblöcke und eingebettete Multiplizierer verwenden.

Aus diesem Grund fiel die Wahl bei der Suche nach einem geeigneten System auf den in Kapitel 3 beschriebenen VLIW-Prozessor CoreVA. Dieser Prozessor vereint die positiven Eigenschaften der vorgestellten Prozessoren und umgeht die eben genannten Ausschlusskriterien. So erlaubt er zum Einen den vollständigen Zugriff auf die Verhaltensbeschreibung des Prozessorkerns, zum Anderen ist er von Anfang an als ASIC-Implementierung konzipiert worden und verfügt beispielsweise über passende Schnittstellen für ASIC-Speicherblöcke. Des Weiteren besitzt der CoreVA die für diese Arbeit benötigte Konfigurierbarkeit, sowie eine Werkzeugkette, die aus einem hardwarespezifischen Compiler und einem anpassbaren Instruktionssatzsimulator besteht (siehe Kapitel 4 und 5.3). Zur effizienten Abbildung von Signalverarbeitungsalgorithmen kann der CoreVA-Prozessor ebenfalls durch spezielle Multiply-Accumulate- und SIMD-Funktionseinheiten erweitert werden. Ein in Anlehnung an ARM-Prozessoren entwickelter Instruktionssatz sichert letztlich die Kompatibilität zu gängigen Entwicklungswerkzeugen.

[23] Application Specific Integrated Circuit, Anwendungsspezifische integrierte Schaltung

3 Die Architektur des CoreVA-Prozessors

Der CoreVA-Prozessor und die zugehörige Entwicklungsumgebung wurden ursprünglich im Rahmen einer interdisziplinären Zusammenarbeit der Fachgruppen „Schaltungstechnik“ und „Programmiersprachen und Übersetzer“ der Universität Paderborn entwickelt, wobei die Prozessorarchitektur im Wesentlichen von Jungeblut [41] konzipiert wurde. Das geplante Einsatzgebiet dieses Prozessors bildete die digitale Signalverarbeitung in mobilen Multiband-Multistandard-Systemen. Das Hauptziel des Entwicklungsprozesses war dementsprechend die Konzeptionierung eines Prozessors, der eine hohe Leistungsfähigkeit aufweist, flexibel und kostengünstig ist und dabei trotzdem eine möglichst geringe Leistungsaufnahme hat. Mit dem Umzug dieser Fachgruppe an die Universität Bielefeld (Arbeitsgruppe Kognitronik und Sensorik) ergeben sich als zusätzliche Anwendungsfelder Einsätze in mobilen Systemen wie beispielsweise Robotern. Der CoreVA-Prozessor soll in folgenden Projekten als anwendungsspezifischer eingebetteter Prozessor eingesetzt werden und als Standard-Prozessor innerhalb eines Multiprozessorsystems Verwendung finden. Hierbei wird weiterhin das Ziel einer möglichst hohen Energieeffizienz verfolgt.

Wie im Folgenden detailliert erläutert wird, basiert der CoreVA-Prozessor auf einer Very Long Instruction Word Architektur. Die Anzahl der parallelen Verarbeitungseinheiten (ALUs) sowie die Anzahl der implementierten Funktionseinheiten (Multiplizierer, Dividierer, Load/Store-Einheiten) ist zur Entwurfszeit konfigurierbar. Dies ermöglicht dem Entwickler eine komfortable Entwurfsraumexploration über eine Vielzahl unterschiedlicher Parameter. Aufgrund dieser Eigenschaften trägt der Prozessor den Namen CoreVA (Configurable resource-efficient VLIW Architecture).

Der Instruktionssatz des CoreVA-Prozessors, der in Anlehnung an die Instruktionssätze der in Kapitel 2.1 vorgestellten ARM-Prozessoren entwickelt wurde, weist 46 skalare Instruktionen und 20 SIMD-Instruktionen auf (siehe Anhang). Der CoreVA-Prozessor gehört somit ebenfalls zur Klasse der RISC-Prozessoren. Durch den Einsatz eines RISC-Instruktionssatzes besitzt der CoreVA im Vergleich zu CISC[1]-Systemen eine deutlich geringere Anzahl an Befehls- und Adressierungsarten. Bei

[1] Complex Instruction Set Computer

RISC-Prozessoren werden komplexe Berechnungen nämlich nicht durch einzelne umfangreiche Instruktionen abgebildet, sondern bereits vom Compiler durch das Verbinden mehrerer Grundinstruktionen mit geringer Komplexität dargestellt. Dieser Minimalismus ermöglicht eine deutlich effizientere Dekodierung und Verarbeitung der Instruktionen, da die Instruktionen eines CISC-Systems im Prozessor erst mit Hilfe eines Interpreters in mehrere Mikroinstruktionen aufgeteilt werden müssen. Neben den bereits genannten Vorteilen von RISC-Architekturen erleichtert ein reduzierter Instruktionssatz den effizienten Einsatz verschiedener Parallelisierungstechniken auf Instruktions- oder Datenebene, die in den Kapiteln 3.1, 3.2 und 3.3 näher erläutert werden [77].

3.1 Das Befehlspipelining

Das Befehlspipelining, beziehungsweise die Fließbandverarbeitung, bildet eine Form der Parallelisierung auf Instruktionsebene. Beim Pipelining wird der Datenpfad, den die Instruktionen während ihrer Verarbeitung durchlaufen, in einzelne Teilschritte aufgespalten. Die so entstehenden Pipelinestufen sind durch Register voneinander getrennt, was bewirkt, dass die Instruktionen jeweils nur einen Teilschritt pro Taktzyklus durchlaufen. Da die einzelnen Pipelinestufen jedoch parallel betrieben werden, lässt sich für mehrere aufeinander folgende Instruktionen eine deutliche Durchsatzsteigerung (Speedup) erreichen. Durch die Fließbandverarbeitung können innerhalb eines Prozessortaktes beispielsweise der dritte Teilschritt der ersten Instruktion, der zweite Schritt der zweiten Instruktion und der erste Schritt der dritten Instruktion parallel verarbeitet werden (siehe Abbildung 3.1) [77].

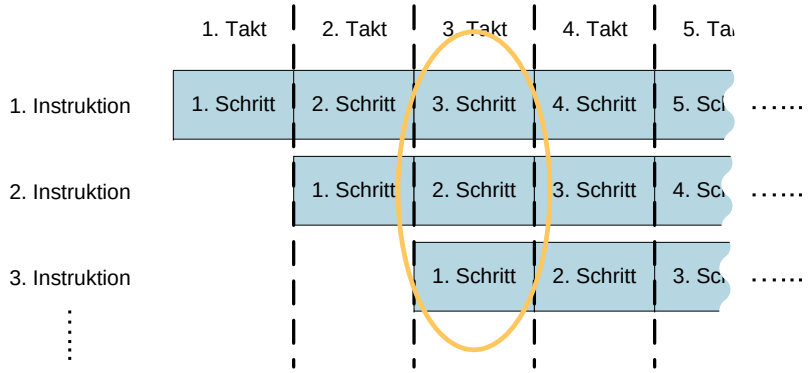

Abbildung 3.1: Gleichzeitiges Verarbeiten mehrerer Instruktionen in einer Pipeline

Bei einer n-stufigen Pipeline kann sich somit, falls die Latenz des kritischen Pfades optimal auf die einzelnen Pipelinestufen aufgeteilt wird und falls die Stufen ideal ausgelastet sind, eine n-fache Steigerung des Durchsatzes ergeben. Eine solche

Durchsatzsteigerung kann im realen Betrieb jedoch nur selten erreicht werden, da die Pipeline aufgrund von Datenkonflikten häufig durch Wartezyklen (stall) und No-Operation Instruktionen (NOP) angehalten werden muss (siehe Kapitel 7.1). Datenkonflikte resultieren aus Datenabhängigkeiten zwischen Instruktionen, die in einem zu geringem Abstand aufeinander folgen. Eine Instruktion am Anfang des Datenpfades kann beispielsweise nicht auf Registerinhalte zugreifen, wenn diese zeitgleich durch vorangegangene Instruktionen verändert werden [74, 77]. Um diese Fälle zu minimieren, werden optimierende und transformierende Compiler eingesetzt, die diese Konflikte durch geschickte Instruktionsanordnung (Scheduling) reduzieren. Ein weiterer Ansatzpunkt ist die Verwendung von Bypasspfaden, die detailliert in Kapitel 3.14 beschrieben werden.

Erfahrungen aus der Hardwareentwicklung zeigen weitere Vorteile des Befehlspipelinings. Der kritische Pfad, der die maximale Betriebsfrequenz eines Prozessors maßgeblich beeinflusst und häufig durch den Prozessorkern verläuft, lässt sich durch geschicktes Pipelining verkürzen. Des Weiteren ermöglichen RISC-Instruktionssätze im Vergleich zu CISC-Systemen eine effizientere Entwicklung von Pipeliningstrukturen. Da jede Pipelinestufe eine bestimmte Aufgabe in dem Datenpfad übernimmt, muss für alle Instruktionen eine einheitliche Folge von Teilschritten gefunden werden. Diese Aufteilung ist aufgrund der geringeren Komplexität für RISC-Instruktionssätze deutlich besser realisierbar [58].

Der CoreVA-Prozessor besitzt eine Pipelinestruktur, die in die folgenden sechs Pipelinestufen unterteilt ist (siehe Abbildung 3.2). Die Aufgaben und Funktionalität der einzelnen Stufen werden detailliert in den Kapiteln 3.5 bis 3.10 erläutert [89].

- Instruction-Fetch-Stufe (FE)
- Instruction-Decode-Stufe (DC)
- Register-Read-Stufe (RD)
- Execute-Stufe (EX)
- Memory-Access-Stufe (ME)
- Register-Write-Stufe (WR)

3.2 Der VLIW-Ansatz

Der VLIW-Ansatz nutzt ebenfalls die Parallelität auf Instruktionsebene, um den Durchsatz eines Prozessors zu erhöhen. Als Erweiterung zur Fließbandverarbeitung besitzen VLIW-Prozessoren mehrere Verarbeitungseinheiten, um eine parallele Ausführung voneinander unabhängiger Instruktionen zu ermöglichen. Diese Verarbei-

tungseinheiten bilden jeweils einen vertikalen VLIW-Slot, der ein Instruktionswort durch die einzelnen Pipelinestufen führt (siehe Abbildung 3.2).

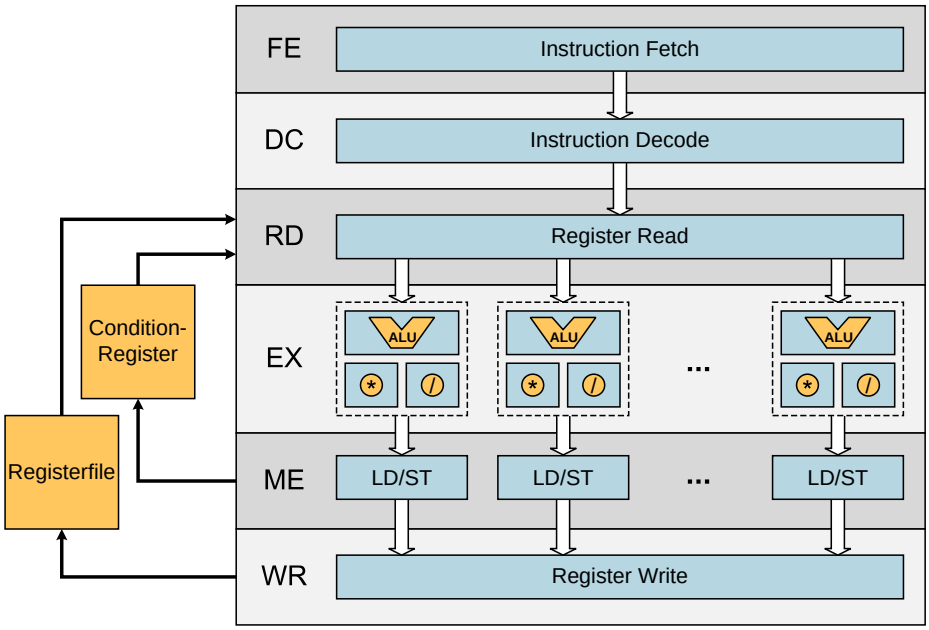

Abbildung 3.2: Generische Pipelinestruktur des CoreVA-Prozessors

Im Unterschied zur Parallelverarbeitung mit Hilfe der SIMD-Technik (siehe Kapitel 3.3), erlaubt VLIW das parallele Ausführen verschiedener Berechnungen auf unterschiedlichen Eingangsdaten. Das parallele Anordnen der Instruktionen erfolgt bei VLIW-Prozessoren bereits während des Kompilierens. Der Compiler analysiert hierzu die aufeinander folgenden Instruktionen bezüglich ihrer Parallelisierbarkeit und fasst mehrere Instruktionen in einer Instruktionsgruppe zusammen. Hierbei dürfen die parallelen Instruktionen keine Datenabhängigkeiten untereinander aufweisen, was bedeutet, dass Instruktionen, deren Rechenergebnisse aufeinander aufbauen, nicht parallel verarbeitet werden dürfen. Des Weiteren ist der Compiler durch die Anzahl der parallelen Verarbeitungseinheiten und durch etwaige Heterogenitäten in ihrer Funktionalität beschränkt [77].

Bei der Ausführung eines VLIW-Programms werden die Instruktionen einer Instruktionsgruppe vollständig in die erste Pipelinestufe eingelesen und von dort aus automatisch auf die vertikalen VLIW-Slots verteilt und zeitgleich verarbeitet. Aus diesem Grund muss die Breite aller Pipelinestufen, und somit beispielsweise die Anzahl der durchzureichenden Operanden und Rechenergebnisse, mit der Anzahl der Verarbeitungseinheiten skalieren. Im Vergleich zu superskalaren Prozessoren, bei denen die einzelnen Instruktionen erst innerhalb des Prozessors parallelisiert werden, lässt sich in VLIW-Systemen ein Großteil der Steuerlogik durch die Verlagerung des Parallelisierens einsparen. Hierdurch werden jedoch an den Compiler,

im Vergleich zu Compilern für nicht parallele oder superskalare Architekturen, sehr hohe Anforderungen gestellt [58, 77].

Die Anzahl der Verarbeitungseinheiten ist im Falle des CoreVA-Prozessors zur Entwurfszeit konfigurierbar. Jede Verarbeitungseinheit kann eine 32-Bit Instruktion verarbeiten. Obwohl die Verarbeitungseinheiten homogen sind, besteht die Möglichkeit ein heterogenes VLIW zu implementieren. Der Entwickler kann beispielsweise für jeden VLIW-Slot getrennt konfigurieren, ob die ALU durch dedizierte 32-Bit Multiplizier- und Dividiereinheiten unterstützt wird, oder ob sie mit Hilfe von Load/Store-Instruktionen Zugriffe auf den externen Datenspeicher initiieren kann.

3.3 Die SIMD-Technik

SIMD (Single Instruction Multiple Data) beschreibt eine weitere Form der parallelen Datenverarbeitung in Prozessoren. Bei der SIMD-Technik wird die Parallelität auf Datenebene benutzt, um mit einer einzelnen Instruktion mehrere parallele Berechnungen auf unterschiedlichen Datenworten auszuführen. Als Beispiel für eine Datenparallelität sei hier der Programmcode 3.1 genannt. Die Schleife schreibt zwar eine sequenzielle Verarbeitung vor, die einzelnen Elemente der Eingangs- und Ausgangsdaten sind jedoch voneinander vollkommen unabhängig, weshalb die einzelnen Additionen problemlos parallelisiert werden können. Die Parallelisierung mittels SIMD wird häufig in der digitalen Signalverarbeitung verwendet, da in diesem Umfeld meistens regelmäßige Datenstrukturen in Form von Vektoren beziehungsweise Arrays mit identischen Berechnungen verarbeitet werden (siehe Kapitel 2 und 6) [77].

```
1 for (i=0; i<n; i++) {
2   a[i] = b[i] + c[i];
3 }
```

Programmcode 3.1: Beispiel einer parallelisierbaren Vektorberechnung

Bei SIMD werden zwei Herangehensweisen unterschieden, die in Feld- oder Vektorrechnern Verwendung finden. Feldrechner sprechen mehrere gleichartige Verarbeitungseinheiten mit einem einzigen Befehl an. Die Verarbeitungseinheiten arbeiten hierbei auf unterschiedlichen Eingangsdaten und erzeugen jeweils ein Rechenergebnis. Die Größe und Komplexität der einzelnen Verarbeitungseinheiten reicht hierbei von ALUs mit minimalem Funktionsumfang bis hin zu vollständigen CPUs. Feldrechner sind bei der Verwendung kleinerer Verarbeitungseinheiten mit einem

VLIW-Prozessor vergleichbar, der in allen Slots die gleiche Instruktion ausführt. Im Unterschied zu einer VLIW-Implementierung wird durch SIMD jedoch der Aufwand für das Dekodieren der zusätzlichen Instruktionen vermieden [77].

Bei Vektorrechnern ist eine einzelne Verarbeitungseinheit in der Lage, ein Eingangsdatum aufzuspalten und die darin enthaltenen Elemente getrennt zu verarbeiten. Die Anzahl der parallel berechenbaren Elemente richtet sich bei Vektorrechnern direkt nach der Breite der Verarbeitungseinheiten und der Eingangs- und Ausgangsregister. Prozessorerweiterungen wie die Tensilica ConnX BBE DSP-Elemente besitzen beispielsweise einen speziellen Vektor-VLIW-Slot, der auf ein Vielfaches von 32 Bit erweiterbar ist (siehe Kapitel 2.3). Hierzu zählt sowohl die Verbreiterung der Verarbeitungseinheit auf bis zu 64 parallele 16-Bit ALU-Operationen und 64 parallele 16x16-Bit Multiplizieroperationen, als auch die dazu passende Vergrößerung der Eingangs- und Ausgangsregister [11, 77].

Der CoreVA-Prozessor arbeitet nach dem Prinzip der Vektorrechner und ermöglicht das Ausführen zweifacher SIMD-Instruktionen mit einer Datenwortbreite von jeweils 16 Bit. Es wurde bei der Implementierung bewusst auf eine heterogene VLIW-Struktur mit speziellen Vektor-Verarbeitungseinheiten verzichtet, damit weiterhin das 32-Bit Registerfile verwendet werden kann und die Pipelinestufen nicht maßgeblich erweitert werden müssen. Bei aktivierter SIMD-Unterstützung müssen lediglich die Verarbeitungseinheiten um zwei 16-Bit ALUs und zwei 16-Bit Multiplizierer erweitert werden, da SIMD-Berechnungen in den 32-Bit Verarbeitungseinheiten nicht effizient möglich sind.

Der CoreVA-Prozessor erreicht trotz der Begrenzung der Datenwortbreite auf 32 Bit eine hohe Parallelität, da die SIMD-Instruktionen in sämtlichen VLIW-Slots ausgeführt werden können. Durch diese Kombination aus SIMD und VLIW ist ein CoreVA-Prozessor mit vier Verarbeitungseinheiten in der Lage, acht 16-Bit Berechnungen parallel auszuführen.

3.4 Implementierte Register

Der CoreVA-Prozessor besitzt 32 Datenregister (General-Purpose-Register) mit einer Breite von jeweils 32 Bit. Diese Datenregister sind fester Bestandteil des Datenpfades, da sie die für die Berechnungen benötigten Eingangsdaten (Operanden) bereitstellen. Durch ihre direkte Integration in die Prozessorpipeline bilden sie des Weiteren einen sehr schnellen Zwischenspeicher für Rechenergebnisse aufeinander folgender Instruktionen.

Zu Beginn einer Berechnung werden die Operanden aus den Datenregistern gelesen und an die Verarbeitungseinheiten weitergeleitet. Nachdem die Verarbeitungseinheiten die Berechnung durchgeführt haben, werden die Rechenergebnisse wiederum in Datenregistern abgelegt und stehen dort nachfolgenden Instruktionen zur Verfügung. Da es keine direkte Verbindung zwischen den Eingängen der Verarbeitungseinheiten und dem Datenspeicher des Prozessors gibt, werden die Register zusätzlich als Zwischenspeicher bei der Verarbeitung von Werten des Datenspeichers genutzt. Hierzu werden die benötigten Speicherinhalte mit einer vorgelagerten Ladeinstruktion in ein Datenregister geschrieben und nach vollendeter Berechnung mit Hilfe einer Speicherinstruktion zurückgeschrieben [77].

Falls mehrere Verarbeitungseinheiten implementiert sind, ist jede dieser Einheiten in der Lage, auf alle verfügbaren Datenregister zuzugreifen. Das gleichzeitige Lesen mehrerer Verarbeitungseinheiten aus einem Register stellt hierbei kein Problem dar. Wenn mehrere Verarbeitungseinheiten jedoch gleichzeitig in dasselbe Register schreiben, werden die Daten bitweise durch eine Oder-Verknüpfung verbunden. Diese Funktionalität kann vom Compiler gezielt zum Verbinden von Teilergebnissen verschiedener Verarbeitungseinheiten desselben Taktes genutzt werden.

Die Datenregister sind in einem zentralen Registerspeicher, dem sogenannten Registerfile, angeordnet. Wie in Abbildung 3.2 zu sehen ist, erfolgt das Auslesen der Registerinhalte in der Register-Read-Stufe. Da die Verarbeitungseinheiten und gegebenenfalls angehängte Multiplizierer bis zu drei Eingangsdaten verarbeiten, muss das Registerfile drei 32-Bit Leseports pro Verarbeitungseinheit besitzen. Die Adressierung der Datenregister erfolgt über einen 5-Bit Adresseingang pro Leseport. Das Beschreiben des Registerfiles mit den Rechenergebnissen der Verarbeitungseinheiten oder mit Werten, die aus dem Datenspeicher gelesen wurden, erfolgt in der Register-Write-Stufe. Der CoreVA-Prozessor unterstützt hierbei auch Instruktionen, die neben dem Lesen oder Schreiben des Datenspeichers das gleichzeitige Ändern der Speicheradresse ermöglichen. Diese Pre- beziehungsweise Post-Modify genannten Instruktionen addieren hierzu entweder vor dem Speicherzugriff einen Offset auf die Speicheradresse, um sofort auf die geänderte Adresse zuzugreifen (Pre-Modify), oder erst nach dem Speicherzugriff (Post-Modify). Da hierdurch bei Leseoperationen der Speicherinhalt und die neue Speicheradresse in das Registerfile übernommen werden müssen, besitzt das Registerfile für jede Verarbeitungseinheit, die Zugriff auf den Datenspeicher hat, zwei 32-Bit Schreibeports und zwei 5-Bit Adresseingänge. Für Verarbeitungseinheiten, die keine Verbindung zum Datenspeicher besitzen, wird dementsprechend nur ein Schreib- und Adresseingang benötigt [89, 77].

Falls die Unterstützung von zweifachen SIMD-Instruktionen aktiviert ist, werden die 32-Bit Register des Registerfiles durch jeweils zwei 16-Bit Register mit eigenem Write-Enable-Anschluss realisiert. Hierdurch wird sichergestellt, dass beiden Hälften der SIMD-Instruktionen getrennt voneinander ausgeführt werden können. Die

Registerhälften sind entweder mit der oberen, oder mit der unteren Hälfte der Lese- oder Schreibeports verbunden. Die Adresssignale werden dupliziert.

Neben den Datenregistern besitzt der CoreVA-Prozessor Condition-Register (Bedingungsregister) mit einer Breite von 8 Bit, um die in Kapitel 3.12 und 3.13 beschriebene bedingte Ausführbarkeit der verschiedenen Instruktionstypen zu ermöglichen. Das unbedingte Ausführen dieser Instruktionen wird durch das Condition-Register C7 gekennzeichnet, welches dauerhaft auf 1 gesetzt ist. Zur Steuerung der bedingten Ausführbarkeit einzelner SIMD-Instruktionshälften werden dementsprechend zwei Condition-Register in den Prozessor integriert. Das Auslesen der Condition-Register erfolgt über zwei 8-Bit Signale, die in die Register-Read-Stufe eingebunden sind (siehe Abbildung 3.2). Die Extrahierung der einzelnen Bits erfolgt erst innerhalb der Pipeline-Struktur. Da die Condition-Register nicht direkt durch Inhalte des Datenspeichers verändert werden können, erfolgt das Schreiben dieser Register bereits in der Memory-Access-Stufe. Hierbei werden getrennt nach den SIMD-Instruktionshälften für jede Verarbeitungseinheit zwei 8-Bit Signale zum Übertragen der Werte und ebenfalls zwei 8-Bit Signale zur Adressierung der Register-Bits verwendet [89].

3.5 Die Instruction-Fetch-Stufe

Die Instruction-Fetch-Stufe (Instruktionsabrufungseinheit) bildet die erste Pipelinestufe des CoreVA-Prozessors. Mit Hilfe dieser Stufe werden die zu verarbeitenden Instruktionsworte abhängig vom Stand des Programmzählers aus dem Instruktionsspeicher ausgelesen und an den Prozessorkern übergeben [77].

Wenn mehrere Instruktionen parallel ausgeführt werden sollen, werden sie vom Compiler in einer Instruktionsgruppe zusammengefasst. Die Instruction-Fetch-Stufe muss die komplette Instruktionsgruppe einlesen, um die enthaltenen Instruktionsworte gleichzeitig an die verschiedenen Dekodiereinheiten weiterleiten zu können. Es ist daher sinnvoll, dass für einen Prozessor mit N parallelen Verarbeitungseinheiten eine Speicherbreite von $N \cdot 32$ Bit gewählt wird.

Falls der Compiler nicht in der Lage ist, alle Verarbeitungseinheiten auszulasten, ist die Instruktionsgruppe schmaler als die Speicherbreite. Um zu vermeiden, dass in diesem Fall Instruktionen des nachfolgenden Taktes eingelesen werden, müssten zu kleine Instruktionsgruppen durch 32 Bit breite NOP-Instruktionen aufgefüllt werden. Da hierdurch jedoch unnötig Speicherplatz verschwendet wird, wurde eine Instruktionskompression implementiert, die das letzte Instruktionswort einer Instruktionsgruppe durch das Setzen eines Stop-Bits (Bit 31) kennzeichnet. Durch

die Instruktionskompression entfällt zwar das Einfügen von NOP-Instruktionen, es kann jedoch passieren, dass die Instruktionsgruppen nicht mehr an den Blockgrenzen des Speicher ausgerichtet sind, sondern sich über mehrere Blöcke erstrecken (siehe Abbildung 3.3). Um Verzögerungen hierdurch zu vermeiden, besitzt die Instruction-Fetch-Stufe ein Alignment-Register, dass mindestens zwei Speicherblöcke vorhält und die Verfügbarkeit einer neuen Instruktionsgruppe in jedem Taktzyklus gewährleistet. Neben der Korrektur der Instruktionsausrichtung kann das Alignment-Register durch das Einfügen weiterer Registerbänke auch als Pufferspeicher (Level-0-Cache) zwischen cache-basierten Instruktionsspeichern und dem Prozessorkern genutzt werden [64, 77].

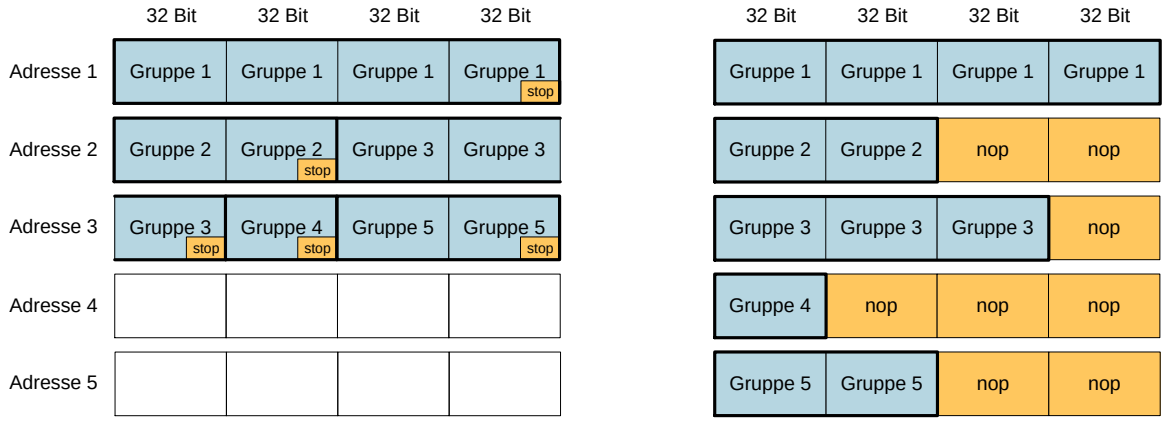

Abbildung 3.3: Exemplarischer Speicherinhalt mit und ohne Instruktionskompression bei einer Speicherbreite von 128 Bit (4 Verarbeitungseinheiten)

3.6 Die Instruction-Decode-Stufe

Die in der Instruction-Fetch-Stufe extrahierten Instruktionsworte werden im folgenden Prozessortakt in der Instruction-Decode-Stufe tiefergehend analysiert. Da der CoreVA-Prozessor Instruktionsworte mit unterschiedlichem Aufbau unterstützt, werden in den Dekodiereinheiten zuerst die übergebenen Typenkennungsbits analysiert und daraus der Instruktionstyp bestimmt (siehe Abbildung 3.4). Anschließend werden dem Instruktionswort die enthaltenen konstanten Operanden, Quell- und Zielregisteradressen (rn, rd, rm) sowie das Stop-Bit und das zugeordnete Condition-Register-Bit entnommen und auf verschiedene Signale aufgeteilt.

Bei der Analyse der Typenkennungsbits unterscheidet die Instruction-Decode-Stufe in erster Instanz, ob es sich bei dem Instruktionstyp um arithmetische oder logische Operationen (ALU), Operationen zum Lesen und Schreiben des Datenspei-

chers (LD/ST) oder Sprungoperationen (BR) handelt. Des Weiteren werden Spezialinstruktionen für Vektoroperationen (SIMD) oder für Multiplikationen (MAC) und Divisionen (DIV) gesondert behandelt (siehe Anhang).

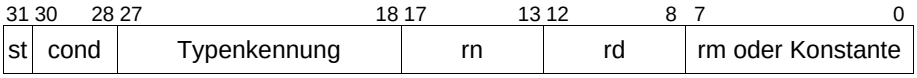

Abbildung 3.4: Aufbau eines Instruktionsworts

Bezüglich der Eingangsdaten unterscheidet die Instruktionsdekodierung, ob alle Operanden dem Registerfile entnommen werden, oder ob einzelne Operanden als konstante Datenworte in dem Instruktionswort hinterlegt sind. Sprunginstruktionen wird beispielsweise das Sprungziel durch Übergabe einer Registeradresse (Register-Sprung) oder durch das direkte Weiterleiten eines 22-Bit Wertes (Offset-Sprung) vorgegeben. Bei arithmetischen und logischen Operationen kann in den meisten Fällen gewählt werden, ob ihnen neben der Zielregisteradresse zwei Quellregisteradressen oder eine Quellregisteradresse und ein konstantes Datenwort übergeben werden. Da Multiply-Accumulate Instruktionen drei Operanden und eine Zielregisteradresse benötigen, wird bei diesem Instruktionstyp auf die Unterstützung von konstanten Werten verzichtet. Des Weiteren sei besonders auf die MVC-EXT-Instruktion hingewiesen, die dem Übergeben eines 32-Bit Datenwortes an das Registerfile dient. Da diese Instruktion neben dem Datenwort die Adresse des Registers sowie die Typenkennungs- und Stop-Bits enthalten muss, kann es nur mit Hilfe eines 64-Bit Instruktionsworts übertragen werden. Aus diesem Grund folgt auf eine 32-Bit MVC-Instruktion (move-constant) stets ein 32-Bit EXT-Instruktionswort (extension), welches die fehlenden 24 Bit des Datenworts nachliefert. Zuletzt wird bezüglich des Speicherns der Daten unterschieden, ob das Rechenergebnis wiederum im Registerfile hinterlegt werden soll, oder zur Änderung der Inhalte der Condition-Register dient [77].

Neben der Auswertung der Instruktionsworte dient die Instruction-Decode-Stufe der Ausführung von bedingten und unbedingten Sprüngen. Wie detailliert in Kapitel 3.12 beschrieben wird, unternimmt diese Stufe hierzu Sprungvorhersagen und besitzt Mechanismen zur Rückabwicklung falsch ausgeführter Sprünge.

3.7 Die Register-Read-Stufe

In der Register-Read-Stufe werden die für die Berechnung benötigten Eingangsdaten zusammengestellt und an die Execute-Stufe weitergeleitet. Standardmäßig werden hierbei jeder Verarbeitungseinheit zwei 32-Bit Operanden übergeben. Da

die in Kapitel 3.8 beschriebene Multiply-Accumulate Operation drei Operanden verarbeitet, müssen Verarbeitungseinheiten, die einen Multiplizierer besitzen, drei 32 Bit breite Eingangswerte geliefert bekommen.

Bei der Zusammenstellung der Eingangswerte wird wiederum unterschieden, ob die Operanden dem Datenregisterfile entnommen werden oder teilweise als Konstanten in dem Instruktionswort hinterlegt sind. Falls in der Instruction-Decode-Stufe eine Konstante extrahiert wurde, kann dieser Wert von der Register-Read-Stufe direkt übernommen und weitergeleitet werden. Falls eine Berechnung jedoch einen Operanden benötigt, der im Datenregister hinterlegt wurde, wird dieses von der Instruction-Decode-Stufe durch das Setzen des zugehörigen Register-Read-Enable-Signals signalisiert. Zum Lesen des Operanden legt die Register-Read-Stufe die in der Instruktion übergebene Registeradresse an das Datenregisterfile an und verbindet dessen Rückgabewerte mit den passenden Eingängen der Verarbeitungseinheiten. In ähnlicher Weise wird beim Auslesen der Condition-Register verfahren. Die Inhalte der Condition-Register werden jedoch zusätzlich an die Instruction-Decode-Stufe zurückgeführt, um die in Kapitel 3.12 beschriebene Sprungvorhersage zu überprüfen [77].

3.8 Die Execute-Stufe

Die Execute-Stufe besitzt in jedem VLIW-Slot eine arithmetisch-logische Verarbeitungseinheit (ALU), in der die eigentliche Berechnung der Instruktionen erfolgt. Die ALUs bestehen jeweils aus einer dedizierten 32-Bit Addier- und Subtrahiereinheit sowie aus einer arithmetisch-logischen Schiebeeinheit. Zur Berechnung der Instruktionen werden die in den vorausgegangenen Pipelinestufen bestimmten Instruktionstypen sowie die nun vorliegenden Operanden und Steuersignale an die Eingänge der Verarbeitungseinheiten angelegt. Die Verarbeitungseinheiten erzeugen anschließend in jedem Prozessortakt ein 32-Bit Ergebnis und leiten dieses an die Memory-Access-Stufe weiter. Neben der Verarbeitung der ALU-Instruktionen werden die Verarbeitungseinheiten auch zur Berechnung der Lese- und Schreibadressen für Datenspeicherzugriffe genutzt und um die Signale zur Manipulation der Condition-Register zu aktualisieren (carry- und zero-Signal) [77].

Wie detailliert in Kapitel 3.2 beschrieben wurde, kann die Execute-Stufe durch dedizierte Multiplizier- und Dividiereinheiten erweitert werden. Die Multiplizierereinheiten ermöglichen das Multiplizieren zweier 32-Bit Werte und sind in der Lage ein 32-Bit Ergebnis pro Rechentakt zu erzeugen. Gleichzeitig ermöglichen sie das Addieren eines dritten Operanden (Akkumulator), wodurch der Prozessor in der Lage ist, Multiply-Accumulate (MAC) Instruktionen der Form $A = B \cdot C + D$ auszuführen.

Die Multiplikationen erfolgen in zwei Teilschritten. Der in der Execute-Stufe implementierte partielle Multiplizierer liefert zwei 32 Bit breite Teilergebnisse im Carry-Save Format. Bei Multiplikationen ohne Akkumulation könnte das Multiplikationsergebnis bereits an dieser Stelle durch das Addieren beider Teilergebnisse ermittelt werden. Da jedoch auch Multiply-Accumulate Instruktionen verarbeiten werden sollen, werden die Teilergebnisse des partiellen Multiplizierers zuerst durch einen Carry-Save Addierer (CSA) mit dem Akkumulator verbunden. Erst nach diesem Schritt werden die zwei Ausgänge des Carry-Save Addierers durch einen Volladdierer zum Endergebnis zusammengefasst.

Um den kritischen Pfad des Prozessors zu verkürzen, sind der Carry-Save Addierer und der Volladdierer in die Memory-Access-Stufe ausgelagert worden (siehe Abbildung 3.5). Hierdurch ergibt sich für Multiplikationen oder MAC-Operationen eine Latenz von zwei Prozessortakten. Das Verschieben dieser Komponenten ist im CoreVA-Prozessor trotz der höheren Latenz problemlos möglich, da die Endergebnisse der Multiplikationen nicht zur Steuerung der Datenspeicherzugriffe genutzt werden und die Memory-Access-Stufe somit nicht auf diese Ergebnisse zugreifen muss [89].

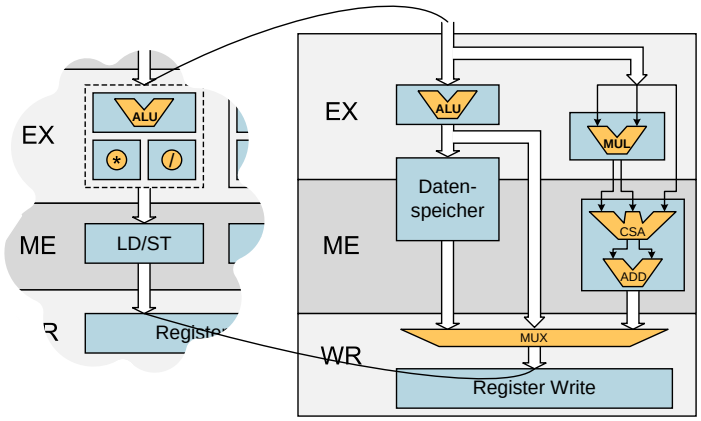

Abbildung 3.5: Verschaltung der Multiplizier- und LD/ST-Einheiten

Divisionen werden mit Hilfe einer dedizierten Divisionsschritteinheit[2] (DIV) beschleunigt. Diese Einheit basiert auf dem Radix-2-Divisionsprinzip und ist in der Lage eine 32-Bit Divisionen innerhalb von 32 Rechentakten durchzuführen [8].

[2]Division-Step-Unit

3.9 Die Memory-Access-Stufe

Die Memory-Access-Stufe ermöglicht dem CoreVA-Prozessor den lesenden oder schreibenden Zugriff auf Elemente des Datenspeichers. Die Anbindung des Speichers an die Prozessorpipeline erfolgt über dedizierte Load/Store-Einheiten (LD/ST), die eng mit den Verarbeitungseinheiten der Execute-Stufe zusammenarbeiten. Wie in Kapitel 3.15 dargestellt wird, kann der Datenspeicher entweder aus einem lokalen on-chip-Speicher oder aus einem externen Speicher mit Cache-Funktionalität bestehen. In beiden Fällen ist der Speicher mit einer little-endian Byte-Reihenfolge realisiert. Leseoperationen liefern Werte mit einer Breite von 32 Bit, die Datenbreite von Schreiboperationen kann 8 Bit, 16 Bit oder 32 Bit betragen [89].

Um die Latenz der Speicherzugriffe zu minimieren, wurden die LD/ST-Einheiten und der an sie angeschlossene Datenspeicher zwischen der Execute- und Memory-Access-Stufe parallel zu den EX-ME-Pipelineregistern platziert (siehe Abbildung 3.5). Hierdurch können die Speicheradressen für Lese- und Schreibzugriffe bereits in der Execute-Stufe an den Datenspeicher angelegt werden. Bei Schreibzugriffen muss neben der Adresse natürlich auch der zu speichernde Wert verfügbar sein. Durch dieses frühzeitige Aktivieren der betreffenden Datenspeicherstelle ist der CoreVA in der Lage, Daten mit einer Latenz von zwei Prozessortakten aus dem Datenspeicher zu lesen und an die Register-Write-Stufe weiterzuleiten. Das Schreiben in den Datenspeicher erfolgt mit einer Latenz von einem Prozessortakt. Im Bezug auf das Nutzen von Bypasspfaden bei aufeinander folgenden Instruktionen (siehe Kapitel 3.14) sei an dieser Stelle erwähnt, dass die gelesenen Daten nicht wie sonst üblich am Ende der Execute-Stufe, sondern erst am Ende der Memory-Access-Stufe zur Verfügung stehen.

Die Rechenergebnisse von logischen oder arithmetischen Berechnungen, die nicht direkt in den Datenspeicher übernommen werden können, werden in der Memory-Access-Stufe nicht weiter beachtet, sondern nur für die Endverarbeitung in der Register-Write-Stufe zwischengespeichert. Da die Condition-Register nicht durch Inhalte des Datenspeichers verändert werden können, müssen die Rechenergebnisse, die zur Aktualisierung dieser Register bestimmt sind, nicht an die Register-Write-Stufe weitergereicht werden, sondern können bereits in der Memory-Access-Stufe gespeichert werden.

Die hier beschriebenen Überschneidungen zwischen der Execute- und Memory-Access-Stufe zeigen die enge Verbundenheit dieser beiden Stufen. Aus diesem Grund finden sich in der Literatur häufig Systeme, bei denen die Berechnungen und die Speicherzugriffe innerhalb einer Stufe erfolgen oder in einer heterogenen VLIW-Anordnung nebeneinander realisiert sind [77]. Im CoreVA-Prozessor wurden diese Stufen bewusst zu Gunsten eines deutlich kürzeren kritischen Pfades getrennt [41].

3.10 Die Register-Write-Stufe

In der Register-Write-Stufe werden schließlich die Rechenergebnisse der Verarbeitungseinheiten und die aus dem Datenspeicher gelesenen Werte in die ausgewählten Datenregister geschrieben, um sie den nachfolgenden Instruktionen zur Verfügung stellen zu können [77].

Hierbei können jedoch durch Pre- und Post-Modify Leseinstruktionen gegebenenfalls zwei parallele Registerschreibzugriffe pro Verarbeitungseinheit erfolgen. Wie bereits in Kapitel 3.4 beschrieben wurde, initiieren diese Instruktionstypen nämlich nicht ausschließlich Speicherzugriffe, sie verändern auch die im Registerfile hinterlegten Speicheradressen, um in folgenden Rechenschritten auf anderen Speicherstellen weiterarbeiten zu können. Da hierdurch sowohl der aus dem Datenspeicher entnommene Wert als auch die aktualisierte Speicheradresse in den Datenregistern gespeichert werden müssen, müssen Verarbeitungseinheiten mit angeschlossener LD/ST-Einheit zwei Schreibports zugeordnet werden [58].

3.11 Das Resource-Sharing

Mit einer steigenden Anzahl von VLIW-Slots kann es aufgrund des erhöhten Ressourcenaufwands sinnvoll sein, nicht alle Verarbeitungseinheiten mit dedizierten Multiplizier-, Dividier- und Load/Store-Funktionseinheiten auszustatten. Um jedoch bei Konfigurationen mit einer begrenzten Anzahl an Funktionseinheiten eine möglichst hohe Auslastung zu gewährleisten, sollten die MAC- und LD/ST-Einheiten unterschiedlichen VLIW-Slots zugeordnet werden. Würden die Funktionseinheiten in den gleichen Slots platziert, könnten MAC- und LD/ST-Instruktionen nicht parallelisiert werden.

Eine mögliche Herangehensweise zur Vermeidung von Überschneidungen ist, die LD/ST-Einheiten den niederwertigsten VLIW-Slots und die MAC- und DIV-Einheiten den höherwertigsten VLIW-Slots zuzuordnen. In einem 4-Slot CoreVA-Prozessor mit zwei LD/ST-Einheiten ergibt sich demnach die in Tabelle 3.1 dargestellte Anordnung[3]. Diese Anordnung ermöglicht zwar eine hohe Auslastung der Funktionseinheiten, jedoch zu Lasten eines sehr großen Programmcodes. Falls beispielsweise in der Konfiguration mit einer MAC-Einheit ausschließlich Multiplikationen verarbeitet werden sollen, wird der Programmcode durch das Einfügen von drei NOP-Instruktionen pro Instruktionsgruppe unnötig vergrößert. Aus diesem Grund

[3] Die Anordnung der DIV-Einheiten entspricht der Anordnung der MAC-Einheiten

kann es mehreren VLIW-Slots mit Hilfe des Resource-Sharings ermöglicht werden, gemeinsam (jedoch nicht gleichzeitig) auf einzelne Funktionseinheiten zuzugreifen [89].

Anzahl FE	$Slot_1$	$Slot_2$	$Slot_3$	$Slot_4$
2 LD/ST, 1 MAC	LD/ST_1	LD/ST_2		MAC_1
2 LD/ST, 2 MAC	LD/ST_1	LD/ST_2	MAC_1	MAC_2
2 LD/ST, 3 MAC	LD/ST_1	LD/ST_2, MAC_1	MAC_2	MAC_3
2 LD/ST, 4 MAC	LD/ST_1, MAC_1	LD/ST_2, MAC_2	MAC_3	MAC_4

Tabelle 3.1: Zuordnung der Funktionseinheiten auf die Verarbeitungseinheiten eines 4-Slot CoreVA-Prozessors ohne Resource-Sharing

Im Falle des oben genannten Beispiels kann das Resource-Sharing so konfiguriert werden, dass alle Verarbeitungseinheiten auf die MAC-Einheit zugreifen können. Falls zwei Speicherzugriffe und eine Multiplikation parallel ausgeführt werden sollen, wird die Multiplikation in Slot 3 ausgeführt. Falls ein oder kein Speicherzugriff erfolgen soll, wird die Multiplikation vom Compiler zu Gunsten der Programmgröße in den Slot 2 oder 1 geschoben und die Instruktionsgruppe mit Hilfe der Instruktionskompression verkleinert (siehe Kapitel 3.5).

Um den Schaltungsaufwand für das Resource-Sharing möglichst gering zu halten und trotzdem eine sinnvolle Abdeckung zu erreichen, werden die LD/ST-Einheiten weiterhin exklusiv den niederwertigsten Verarbeitungseinheiten zugeordnet. Die Multiplizier- und Dividiereinheiten werden mittels Resource-Sharing so verschaltet, dass jede Verarbeitungseinheit auf genau eine Funktionseinheit zugreifen kann. Somit ist gewährleistet, dass sich die MAC-, DIV- und LD/ST-Instruktionen nicht überschneiden und dass durch das Wegfallen von LD/ST-Instruktionen keine NOP-Instruktionen eingefügt werden müssen. Da es jedoch in keinem Fall zu gleichzeitigen Zugriffen mehrerer Verarbeitungseinheiten auf eine Funktionseinheit kommen darf, muss der Compiler genaue Kenntnis über das Verhalten des Resource-Sharings besitzen. Des Weiteren ist darauf zu achten, dass die Funktionseinheiten den Verarbeitungseinheiten möglichst in gleichem Maße zugeordnet werden. Für einen CoreVA-Prozessor mit vier Verarbeitungseinheiten und zwei LD/ST-Einheiten ergibt sich dementsprechend die in Tabelle 3.2 dargestellte Zuordnung der Funktionseinheiten.

Anzahl FE	$Slot_1$	$Slot_2$	$Slot_3$	$Slot_4$
2 LD/ST, 1 MAC	LD/ST_1, MAC_1	LD/ST_2, MAC_1	MAC_1	MAC_1
2 LD/ST, 2 MAC	LD/ST_1, MAC_1	LD/ST_2, MAC_2	MAC_1	MAC_2
2 LD/ST, 3 MAC	LD/ST_1, MAC_1	LD/ST_2, MAC_2	MAC_1	MAC_3
2 LD/ST, 4 MAC	LD/ST_1, MAC_1	LD/ST_2, MAC_2	MAC_3	MAC_4

Tabelle 3.2: Zuordnung der Funktionseinheiten auf die Verarbeitungseinheiten eines 4-Slot CoreVA-Prozessors mit Resource-Sharing

3.12 Bedingte Ausführung von Sprüngen

Wie die meisten Prozessoren unterstützt der CoreVA-Prozessor das Ausführen bedingter und unbedingter Sprünge. Bedingte Sprünge sind dadurch gekennzeichnet, dass ihre Ausführung in Abhängigkeit bestimmter Faktoren (Conditions) erfolgt, die in vorgelagerten Berechnungen bestimmt werden. Durch bedingte Sprünge können beispielsweise Fallunterscheidungen oder Schleifen abgebildet werden [77].

Das folgende Assemblerbeispiel zeigt, dass der Inhalt des Register R1 von R2 subtrahiert werden soll, falls R2 größer oder gleich R1 ist. Diese Vorgabe wird durch einen bedingten Sprung realisiert, der die Subtraktion überspringt wenn das zugehörige Condition-Bit C1 auf 1 gesetzt ist. Ist das Condition-Bit jedoch 0 wird die Sprunginstruktion nicht ausgeführt und die Subtraktion findet statt. Das Condition-Bit wird mit Hilfe der vorangestellten Instruktion CMP SLT (compare, signed-lower-than) auf 1 gesetzt, falls R2 kleiner als R1 ist.

```
1 c7 cmp slt,c1,r2,r1 //c1=1 falls r2 < r1, c1=0 falls r2 >= r1
2 c1 br no_sub //Überspringe Subtraktion, wenn c1=1
3 c7 add r4,r6,r7 //delay-Slot wird immer ausgeführt
4 c7 sub r2,r2,r1 //r2=r2-r1, wird ggf. übersprungen
5 :no_sub
6 c7 add r5,r3,r4
```

Programmcode 3.2: Beispiel für die Verwendung bedingter Sprünge

Im Unterschied zu bedingten Sprüngen werden unbedingte Sprünge stets ausgeführt. Im Falle des CoreVA-Prozessors werden unbedingte Sprünge von bedingten abgegrenzt, indem sie in Abhängigkeit des Condition-Bits C7 ausgeführt werden, welches dauerhaft auf 1 gesetzt ist (siehe Kapitel 3.4). Unbedingte Sprünge kommen beispielsweise bei Aufrufen von Unterfunktionen zum Einsatz, die an anderer Stelle des Instruktionsspeichers hinterlegt sind [77].

Aufgrund der Pipeline-Struktur des CoreVA-Prozessors kommt es bei der Verarbeitung von Sprüngen jedoch zu einer Vielzahl von Problemen, die durch zusätzlichen Schaltungsaufwand vermieden werden müssen. Hierzu zählt beispielsweise, dass das Dekodieren der Instruktionen in der zweiten Pipelinestufe erfolgt (siehe Kapitel 3.6). Der Prozessor erfährt also erst im zweiten Takt, dass die zu verarbeitende Instruktion ein Sprung ist. Zu dieser Zeit hat die Instruction-Fetch-Stufe aber bereits fälschlicherweise damit begonnen, die Instruktionen einzulesen, die im Programmspeicher direkt hinter dem Sprung stehen. Um diesen Fehler nicht korrigieren zu müssen, gilt im CoreVA-Prozessor die Regel, dass die Instruktionsgruppe, die hinter einem Sprung steht, stets ausgeführt wird (siehe Programmbeispiel 3.2). In der

Literatur spricht man in diesem Fall von einer Instruktion im Verzögerungsschlitz (delay-Slot). Da der Compiler Kenntnis von dieser Regel besitzt, kann er den Verzögerungsschlitz gezielt für Berechnungen nutzen, die noch vor dem Sprung ausgeführt werden müssen [77].

Bei der Verarbeitung bedingter Sprünge kommt es zu weiteren Einschränkungen, da die Bedingungen erst zur Laufzeit eines Programms bestimmt werden. Bedingte Sprünge besitzen zwar ebenfalls einen Verzögerungsschlitz, die Bedingungen, die in den Condition-Registern hinterlegt sind, können aber erst in der Register-Read-Stufe ausgelesen werden. Um die Instruction-Fetch-Stufe bis zur Klärung der Bedingung nicht anhalten zu müssen, wird in der Instruction-Decode-Stufe eine statische Sprungvorhersage unternommen. Die Sprungvorhersage handelt nach dem Vorhersagemuster „Predict backward taken, forward not taken“[4].

Bei der Sprungvorhersage werden drei Sprungtypen unterschieden (siehe Tabelle 3.3). Sprünge, die um eine konstante Anzahl von Instruktionen zurück springen (Offset-Sprung rückwärts), werden häufig zur Realisierung von Schleifen genutzt. Da Schleifen deutlich häufiger durchlaufen als abgebrochen werden, geht der Prozessor davon aus, dass ein Rücksprung immer stattfinden wird. Dem entgegengesetzt beschreiben Vorwärtssprünge, um eine konstante Anzahl von Instruktionen (Offset-Sprung vorwärts), neben Fallunterscheidungen, die Ausstiege aus Schleifenkonstrukten, weshalb sie als nicht-genommen vorausgesagt werden. Sprünge zu Programmzählern, die in Registern hinterlegt sind (Register-Sprung), werden unter anderem genutzt, um Fehlerroutinen und Ausnahmebehandlungen auszulösen. Da dies ebenfalls nicht häufig vorkommt, wird die Ausführung von Registersprüngen auch standardmäßig als nicht-genommen vorhergesagt [77].

Sprungtyp	Regel
Offset-Sprung rückwärts	springen
Offset-Sprung vorwärts	nicht springen
Register-Sprung	nicht springen

Tabelle 3.3: Regeln der Sprungvorhersage

Das Vorhersageergebnis, beziehungsweise die Adresse des im nächsten Takt zu ladenden Speicherblocks, wird anschließend von der Instruction-Decode-Stufe an die Instruction-Fetch-Stufe weitergeleitet. Um eine etwaige falsche Vorhersage rückgängig machen zu können, speichert die Instruction-Decode-Stufe die Adresse der nicht gewählten Möglichkeit in einem Hilfsregister, um sie im Fehlerfall wieder zurückgewinnen zu können [77].

Im folgenden Takt lädt die Instruction-Fetch-Stufe die Instruktionsgruppe von der übergebenen Programmzähleradresse. Zeitgleich erfolgt die Überprüfung der

[4] Rückwärts genommen, Vorwärts nicht genommen

Sprungvorhersage in der Register-Read-Stufe durch Auswerten des Condition-Registers. War die Vorhersage erfolgreich, kann der Prozessor verzögerungsfrei weiterlaufen. Stellt sich heraus, dass die Vorhersage falsch war, muss der Sprung ungeschehen gemacht und bereits ausgeführte Instruktionen verworfen werden. Hierzu wird ein Pipeline-Flush ausgeführt, welches die Register zwischen der Register-Read- und der Execute-Stufe zurücksetzt, um die falsch eingefügten Operanden wieder zu löschen. Des Weiteren wird die im Hilfsregister gesicherte Adresse an den Instruktionsspeicher angelegt, um im nächsten Takt die richtige Instruktionsgruppe verarbeiten zu können [77].

3.13 Bedingte Ausführung arithmetisch-logischer Instruktionen

Im Unterschied zu den meisten am Markt erhältlichen Prozessoren unterstützen ARM-Prozessoren das bedingte Ausführen nahezu aller Instruktionen. Die Ausführung der Instruktionen erfolgt hierbei in Abhängigkeit der Statusbits, die von der ALU bei der Berechnung vorangegangener Instruktionen gesetzt wurden. Zu diesen Statusbits gehören beispielsweise das N-Bit, das gesetzt wird, wenn die Berechnung ein negatives Ergebnis liefert, das Z-Bit, das gesetzt ist, falls das Ergebnis 0 ist, und das V-Bit, das einen Überlauf kennzeichnet.

Zur Steuerung der bedingten Ausführbarkeit besitzt der ARM-Instruktionssatz eine Vielzahl von Instruktionskopien, deren Berechnungen identisch sind. Die Instruktionsworte dieser Kopien unterscheiden sich lediglich in den Teilen, in denen das zu nutzende Statusbit vorgeben wird. Der Assemblerbefehl ADDAL (add always) wird beispielsweise stets ausgeführt. Der Befehl ADDEQ (add equal) wird hingegen nur ausgeführt, wenn durch eine vorgelagerte Subtraktion festgestellt wurde, dass zwei Werte gleich groß sind. In diesem Fall ist das Z-Bit gesetzt und kann zur bedingten Ausführung analysiert werden [1, 16].

Da die Möglichkeit des bedingten Ausführens arithmetisch-logischer Instruktionen die Größe eines Programmcodes deutlich reduzieren kann, wurde sie ebenfalls in den Instruktionssatz des CoreVA-Prozessors übernommen. Der CoreVA wertet zur bedingten Ausführung jedoch nicht die Statusbits der ALU aus, sondern nutzt die bereits existierenden Condition-Register. Falls einer Instruktion ein Condition-Bit zugeordnet wurde, das nicht gesetzt ist, wird in der Register-Read-Stufe ein Cancel-Instruction-Signal generiert, woraufhin die Verarbeitungseinheiten der Execute-Stufe das Rechenergebnis dieser Instruktion verwerfen. Ein Vorteil der Condition-Register gegenüber den Statusbits der ARM-Prozessoren ist, dass die Register mehrere Bedin-

gungen speichern können, die auf dem selben Status basieren. In ARM-Prozessoren werden die jeweiligen Statusbits bei jeder erneuten Überprüfung überschrieben. Des Weiteren ermöglicht der Einsatz der zentralen Register, dass die Bedingungen in einem VLIW-Prozessor Slot-übergreifend genutzt werden können.

Das im Programmcode 3.2 angeführte Beispiel vereinfacht sich durch das bedingte Ausführen der Subtraktion wie im Programmcode 3.3 dargestellt. Es zeigt sich, dass ein Verarbeitungsschritt eingespart werden kann. Des Weiteren ist der Einsatz eines Verzögerungsslots nicht mehr nötig, weshalb die hier platzierte Instruktion vom Compiler gegebenenfalls verschoben werden kann.

```
1 c7 cmp slt,c1,r1,r2 //c1=1 falls r1 < r2, c1=0 falls r1 >= r2
2 c7 add r4,r6,r7 //früherer delay-Slot
3 c1 sub r2,r2,r1 //r2=r2-r1, wenn c1=1
4 c7 add r5,r3,r4
```

Programmcode 3.3: Beispiel für die Verwendung bedingter Instruktionen

Ein weiteres Beispiel für die Vorteile des bedingten Ausführens stellt das parallele Abbilden der verschiedenen Zweige von Fallunterscheidungen dar. Die Berechnungen des if- und des else-Zweiges können entweder in zwei VLIW-Slots in Abhängigkeit unterschiedlicher Condition-Register ausgeführt werden, oder durch die bedingte Ausführbarkeit einzelner SIMD-Instruktionsteile sogar in einer SIMD-Instruktion platziert werden.

3.14 Die verschiedenen Bypasssysteme

Der CoreVA-Prozessor besitzt ein differenziertes System von Bypasspfaden, um Datenkonflikte zwischen aufeinander folgenden Instruktionen zu vermeiden. Zu diesen Bypasspfaden gehören der Daten- und Conditionregisterbypass, der Kontrollbypass und der MLA[5]-Bypass. Die Gesamtbreite aller Pfade beläuft sich in einer Beispielkonfiguration mit vier VLIW-Slots und jeweils zwei MAC-, DIV- und LD/ST-Einheiten auf 11008 Bit [90].

Der Datenregisterbypass sorgt dafür, dass Rechenergebnisse, die zwar bereits in der Execute-Stufe berechnet wurden, aber erst in der Write-Register-Stufe in das Registerfile geschrieben werden, frühzeitig in die Register-Read-Stufe zurückgeführt werden können. Mit Hilfe dieses Bypasssystems können sowohl direkt aufeinander

[5]Multiply-Accumulate Instruktion des CoreVA-Instruktionssatzes

folgende Instruktionen verzögerungsfrei ausgeführt werden, als auch Instruktionen mit einem oder zwei Takten Abstand oder mit Abhängigkeiten, die sich über verschiedene VLIW-Slots erstrecken.

Zur Implementierung des Datenregisterbypasses werden Bypasspfade von den Ausgängen der Execute-, Memory-Access- und Write-Register-Stufe zu allen Operandenregistern der Register-Read-Stufe geleitet (siehe Abbildung 3.6). Die Auswahl der jeweiligen Bypasspfade erfolgt anhand eines Abgleichs zwischen den Lese-Registeradressen der einzelnen Operanden der Register-Read-Stufe und den Schreib-Registeradressen der unteren Pipelinestufen [74].

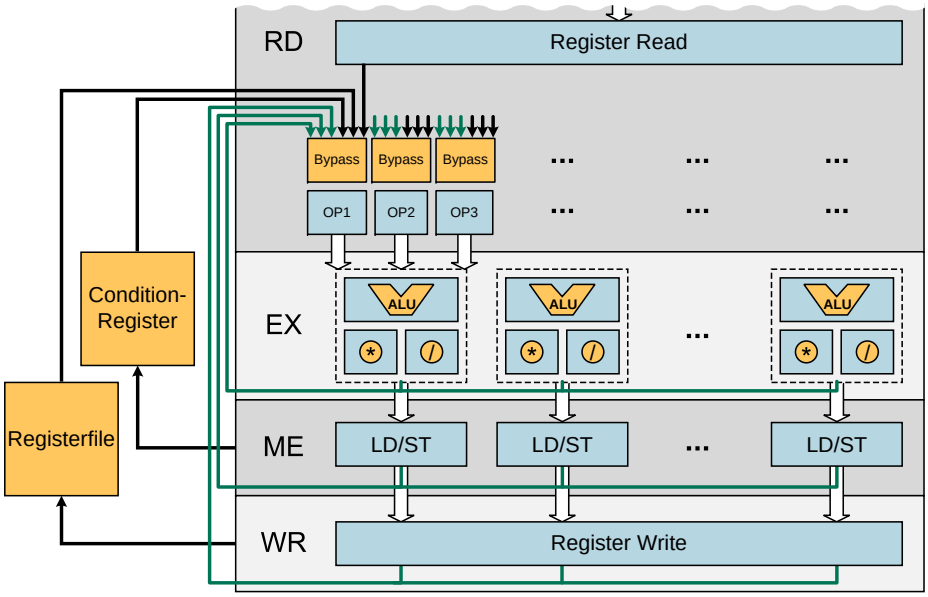

Abbildung 3.6: Ausschnitt der implementierten Registerbypasspfade

Im Unterschied zum Datenregisterbypass wird der Conditionregisterbypass genutzt, um die Zwischenergebnisse in die Register-Read-Stufe zurückzuführen, die in Condition-Registern gespeichert werden sollen. Der Kontrollbypass erweitert diese Funktionalität, indem er die Zwischenergebnisse sogar bis in die Instruction-Decode-Stufe zurückführt, um damit die Ausführung bedingter Sprünge beziehungsweise die Rückabwicklung einer falschen Sprungvorhersage zu steuern. Der MLA-Bypass dient schließlich dazu, Skalarprodukte verzögerungsfrei berechnen zu können. Da Skalarprodukte durch eine Aneinanderreihung mehrerer MAC-Instruktionen dargestellt werden, werden mit Hilfe dieses Bypasssystems die Endergebnisse der jeweiligen MAC-Einheiten an ihre Akkumulatoreingänge zurückgeführt [90].

Da sich die Anzahl der Bypasspfade bei einer steigenden Zahl von VLIW-Slots dramatisch erhöht und häufig nur ein Bruchteil der Pfade regelmäßig genutzt wird, ist die Verfügbarkeit jedes einzelnen Bypasspfades zur Entwurfszeit konfigurierbar.

Wenn ein Bypasspfad nicht implementiert werden soll, werden jedoch nur die jeweiligen Datensignale ausgelassen. Die Rückführung der Schreib-Registeradressen erfolgt auch bei deaktiviertem Bypasspfad, um Datenkonflikte weiterhin detektieren zu können. Falls im Verlauf der Programmausführung auf einen deaktivierten Bypasspfad zugegriffen werden soll, werden Wartezyklen eingefügt, um die Ausführung der nachfolgenden Instruktionen zu verzögern und somit den Konflikt aufzulösen. Das Deaktivieren von Bypasspfaden findet im CoreVA-Prozessor derzeit bei allen Conditionregister- und Kontrollbypasspfaden aus der Execute-Stufe Verwendung, da unsere Untersuchungen gezeigt haben, dass der kritische Pfad des CoreVA-Prozessors durch diese Rückführungen deutlich verlängert wird [90].

3.15 Die Speicheranbindung

Die Speicheranbindung des CoreVA-Prozessors basiert auf der Harvard-Architektur. Im Unterschied zu einer von-Neumann-Architektur ist bei der Harvard-Architektur der Instruktionsspeicher vom Datenspeicher getrennt. Diese Trennung ermöglicht eine effiziente Implementierung des Befehlspipelinings, da sie das gleichzeitige Zugreifen der Instruction-Fetch-Stufe auf den Instruktionsspeicher und der Memory-Access-Stufe auf den Datenspeicher unterstützt [77].

Die Instruktions- und Datenspeicher sind ebenfalls konfigurierbar. Sie können sowohl als externe Speicher ausgelegt werden, die über einen Level-1-Cache an die Prozessorpipeline angebunden werden, oder auch als lokale SRAM[6]-Speicher (on-chip Scratchpad-Speicher) konfiguriert werden, die direkt vom Prozessor adressiert werden können. Im Unterschied zu Cache-basierten Speichern, die im Falle eines Fehlzugriffes (Cache-Miss) den Prozessorkern über viele Takte anhalten, ist der Scratchpad-Speicher in der Lage, die angefragten Daten mit einer garantierten Latenz von zwei Taktzyklen zu liefern. Da der Scratchpad-Speicher jedoch keine Verbindung zu einem externen Arbeitsspeicher besitzt, ist die Größe des verfügbaren Speichers bei Scratchpad-Lösungen im Vergleich zu Cache-basierten Systemen begrenzt. Neben dieser grundlegenden Entscheidung können viele weitere Speichercharakteristika zur Entwurfszeit vorgegeben werden. So kann beispielsweise die Größe der Speicher an die jeweiligen Anwendungen angepasst werden und die Assoziativität, sprich die Anzahl der Cache-Zeilen (Blöcken) pro Cache-Eintrag (Satz), eingestellt werden [77].

Der Datenspeicher wird über dedizierte LD/ST-Einheiten angesprochen. Falls der Prozessor eine LD/ST-Einheit besitzt, wird der Datenspeicher als Single-Port Spei-

[6]Static Random Access Memory

cher mit einem Lese- und Schreibport implementiert. Wenn zwei LD/ST-Einheiten konfiguriert sind, kann der Scratchpad-Speicher entweder als Dual-Port Speicher mit zwei vollwertigen Lese- und Schreibports oder als Multi-Bank Speicher ausgelegt werden. Bei dem Multi-Bank Konzept werden zwei Single-Port Speicherbänke implementiert, die getrennt adressiert werden. Eine Speicherbank beinhaltet die Daten die in geraden Speicheradressen hinterlegt werden, die andere Bank beinhaltet die Daten der ungeraden Adressen. Durch diese Aufteilung ist ein Prozessor mit zwei LD/ST-Einheiten in der Lage, gleichzeitig auf Daten in unterschiedlichen Speicherbänken zuzugreifen. Bei gleichzeitigen Zugriffen auf die selben Speicherbänke kommt es jedoch zu einem Konflikt. In diesem Fall muss der Prozessorkern angehalten werden, um die Speicherzugriffe nacheinander ausführen zu können [89].

Dual-Port Speicher weisen im Unterschied zu Single-Port Speichern aufgrund des zusätzlichen Schaltungsaufwands typischerweise einen sehr hohen Flächen- und Leistungsbedarf auf. Wie in den Kapitel 8.3 und 7 gezeigt wird, kann der Ressourcenaufwand durch den Einsatz des Multi-Bank Konzepts zwar deutlich verringert werden, jedoch zu Lasten einer höheren Laufzeit.

3.16 Erweiterbarkeit des CoreVA-Prozessors

Wie der Großteil der in Kapitel 2 untersuchten Prozessoren bietet auch der CoreVA-Prozessor die Möglichkeit, durch Instruktionssatzerweiterungen oder Hardwarebeschleuniger erweitert zu werden.

Da RISC-Prozessoren einen vergleichsweise eingeschränkten Instruktionsumfang besitzen, können Instruktionssatzerweiterungen genutzt werden, um mehrere Grundinstruktionen zu einer Spezialinstruktion zusammenzufassen. Um eine Instruktionssatzerweiterung in den CoreVA-Prozessor einzufügen, müssen die Merkmale der neuen Instruktion in die Instruktionsdekodierung aufgenommen und die Verarbeitungseinheiten um die gewünschten Funktionalitäten erweitert werden. Zusätzlich muss die Compiler-Werkzeugkette und der Instruktionssatzsimulator angepasst werden. Durch das Hinzufügen solcher Spezialinstruktionen kann die Leistungsfähigkeit des Prozessors bei der Verarbeitung bestimmter Algorithmen zwar gesteigert werden, aus der Erweiterung der Verarbeitungseinheiten resultiert jedoch ein zusätzlicher Schaltungsaufwand und somit ein Anstieg des Ressourcenbedarfs.

Hardwarebeschleuniger sind anwendungsspezifische Co-Prozessoren, die auf eine rechenintensive und häufig wiederkehrende Aufgabe spezialisiert sind. Bei einem Einsatz von Hardwarebeschleunigern werden die Rechenschritte einzelner Programmblöcke in einer kombinatorischen Logik abgebildet, die idealerweise in-

nerhalb eines Taktes ausgeführt werden kann. Hierdurch kann der Prozessor bei der Verarbeitung rechenintensiver Anwendungen maßgeblich entlastet werden, da er die Berechnungen nicht selber ausführen muss, sondern lediglich die jeweiligen Datenpakete an den Hardwarebeschleuniger übergibt und dessen Ergebnisse anschließend wieder zurückholt. Hardwarebeschleuniger können aufgrund der hohen Flächen- und Leistungsanforderungen jedoch meistens nur auf einen bestimmten Algorithmus ausgelegt werden und sind somit äußerst unflexibel in Bezug auf nachträgliche Programmanpassungen.

Um zu vermeiden, dass die Integration der Hardwarebeschleuniger über dedizierte Busleitungen erfolgen muss, arbeitet der CoreVA-Prozessor nach einem MMIO-Ansatz (Memory Mapped Input/Output). Bei diesem Ansatz werden die externen Komponenten einem bestimmten Adressbereich des Datenspeichers zugeordnet. Ein Adressdekoder, der zwischen die LD/ST-Einheiten des Prozessors und den Datenspeicher geschaltet ist, entscheidet, ob Speicherzugriffe an den Datenspeicher oder an die Hardwareerweiterungen weitergeleitet werden (siehe Abbildung 3.7) [77].

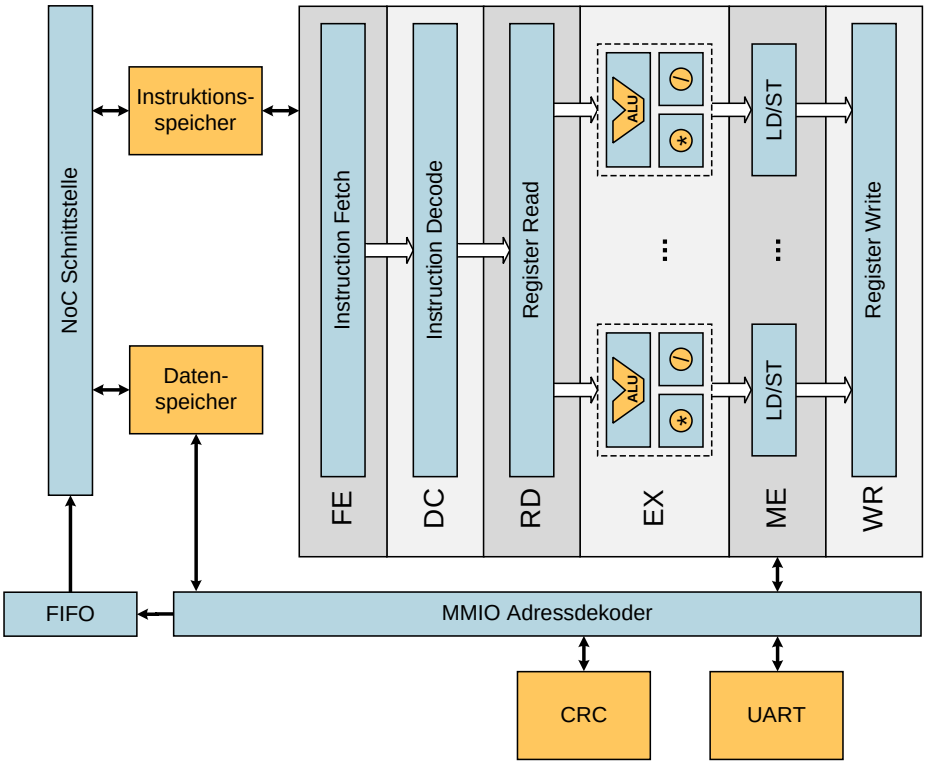

Abbildung 3.7: Anbindung der Hardwareschnittstellen

Damit der kritische Pfad des CoreVA-Prozessors durch das Hinzufügen eines Hardwarebeschleunigers nicht verlängert wird, sind die Beschleuniger durch Eingangs- und Ausgangsregister entkoppelt. Aus diesem Grund beträgt die Latenz bei Zugriffen auf einen Hardwarebeschleuniger, wie auch bei Zugriffen auf den lokalen

Datenspeicher, einen Prozessortakt für das Schreiben und zwei Prozessortakte für das Lesen eines 32-Bit Werts. Als Beispiel für einen Hardwarebeschleuniger des CoreVA-Systems sei hier der CRC[7]-Beschleuniger genannt, der innerhalb eines Taktes eine zyklische Redundanzprüfung auf einem 32 Bit breiten Eingangsdatenwort durchführt (siehe Kapitel 6.2). Neben den Hardwarebeschleunigern bietet der Adressdekoder die Möglichkeit auf externe Schnittstellen wie UART[8] oder Ethernet zuzugreifen, oder über eine NoC[9]-Schnittstelle in ein Multiprozessorsystem eingebunden zu werden [89].

3.17 Konfigurierbarkeit des CoreVA-Prozessors

In der Verhaltensbeschreibung des CoreVA-Prozessors kommt eine Vielzahl generischer Komponenten zum Einsatz, die eine feingranulare Konfiguration der Prozessorelemente zur Entwurfszeit ermöglichen. Die generischen Komponenten werden über Variablen gesteuert, die in zentralen Konfigurationsdateien hinterlegt werden. Mit Hilfe dieser Variablen ist der Entwickler unter anderem in der Lage, die Anzahl und Funktionalität der parallelen Verarbeitungseinheiten vorzugeben oder die Größe der Speicher anzupassen. Je nach Anforderung kann der Prozessor hierbei auch als heterogenes VLIW-System beschrieben werden, bei dem sich beispielsweise mehrere Verarbeitungseinheiten eine Multiplizier- und Dividiereinheit teilen, oder nur bestimmte Verarbeitungseinheiten Zugriff auf den Datenspeicher erhalten (siehe Kapitel 3.2 und 3.11) [89].

Durch den Einsatz der generischen Variablen werden die betroffenen Signalbreiten, Arraygrößen und Instanziierungen automatisch an die jeweiligen Konfigurationen angepasst. Ein weiterer Vorteil dieser Variablen ist, dass keine zusätzlichen Ressourcenanforderungen durch die Konfigurierbarkeit des Prozessors entstehen, da sämtliche Abhängigkeiten und allgemeinen Formulierungen durch die in Kapitel 4.2.2 beschriebenen Synthesewerkzeuge entfernt werden und nur die individuelle Konfiguration in eine Standardzellenbeschreibung abgebildet wird.

[7] Cyclic Redundancy Check, zyklische Redundanzprüfung
[8] Universal Asynchronous Receiver Transmitter
[9] Network on Chip

Die folgende Liste zeigt die variablen Konfigurationselemente des CoreVA-Prozessors.

- Anzahl der VLIW-Slots[10]
- Anzahl der Multiplizierereinheiten
- Anzahl der Dividiereinheiten
- Anzahl der Load/Store Einheiten
- Konfiguration des Resource-Sharing
- Verfügbarkeit zweifacher 16-Bit SIMD-Instruktionen
- Anzahl und Breite der Register im Registerfile
- Verfügbarkeit der Bypasspfade
- Größe und Typ des Instruktions- und Datenspeichers
- Datenbreite des Instruktionsspeichers
- Implementation von Instruktionssatzerweiterungen
- Implementation von Hardwarebeschleunigern

3.18 Zusammenfassung und eigene Beiträge

In diesem Kapitel wurde die Architektur des VLIW-Prozessors CoreVA beschrieben. Der CoreVA wurde ursprünglich entwickelt, um als eingebetteter Prozessor im Bereich der digitalen Signalverarbeitung eingesetzt werden zu können. Er basiert auf einer Harvard-Architektur mit einer sechsstufigen Prozessorpipeline und unterstützt einen ARM-ähnlichen 32-Bit RISC-Instruktionssatz. Der Prozessorkern besitzt eine variable Anzahl paralleler Verarbeitungseinheiten, die jeweils aus einer 32-Bit Addier- und Subtrahiereinheit sowie aus einer arithmetisch-logischen Schiebeeinheit bestehen. Zur Steigerung der Leistungsfähigkeit können die Verarbeitungseinheiten durch dedizierte 32-Bit Multiplizier- und Dividiereinheiten erweitert werden. Die Multiplizierereinheiten sind in der Lage, die in Signalverarbeitungsalgorithmen sehr häufig vorkommenden Multiply-Accumulate Berechnungen mit einem Durchsatz von einer Berechnung pro Takt und einer Latenz von zwei Taktzyklen durchzuführen. Des Weiteren kann der CoreVA zu einem SIMD-Prozessor ausgebaut werden, der zwei 16-Bit Operationen pro Verarbeitungseinheit ausführen kann.

Wie bereits in der Einleitung dieses Kapitels erwähnt wurde, basiert die vorgestellte Architektur im Wesentlichen auf den Entwürfen von Jungeblut [41]. Im Rahmen meiner Arbeit wurde neben zahlreichen Optimierungen, wie beispielsweise der Überarbeitung des Alignment-Registers und des Zusammenspiels zwischen ALU und Multiplizierer, hauptsächlich die Konfigurierbarkeit des Prozessors vorangetrieben. Die Verarbeitungseinheiten waren zwar bereits in ihrer ursprünglichen

[10] Im Falle des CoreVA-Prozessors entspricht die Anzahl der VLIW-Slots der Anzahl an parallelen Verarbeitungseinheiten (ALUs)

Implementierung durch generische Komponenten beschrieben und rudimentär konfigurierbar, in dieser Arbeit wurde der Prozessorkern jedoch so erweitert, dass die Anzahl der Verarbeitungs- und Funktionseinheiten skriptgesteuert und voneinander unabhängig und variiert werden kann (siehe Kapitel 3.17 und 4.2).

Die veränderte Verhaltensbeschreibung des CoreVA-Prozessors ermöglicht somit das Erstellen heterogener Konfigurationen mit bis zu acht Verarbeitungseinheiten, bei denen für jede dieser Einheiten auswählt werden kann, ob sie durch einen dedizierten Multiplizierer und Dividierer unterstützt wird und ob sie auf den Datenspeicher zugreifen kann. Neben diesen grundlegenden Konfigurationsmöglichkeiten kann der Entwickler nun bestimmen, ob der Datenspeicher als Single-Port Speicher mit einem Lese- und Schreibport, als Dual-Port Speicher mit zwei vollwertigen Lese- und Schreibports oder als Multi-Bank Speicher ausgelegt werden soll. Darüber hinaus lassen sich erstmals auch übergeordnete Einstellungen, wie das Aktivieren des Resource-Sharings oder der SIMD-Funktionalität, vornehmen. Da hierfür sowohl das zentrale Registerfile als auch die Beschreibung der SIMD-Instruktionen angepasst werden mussten, konnte in diesem Zuge die bis dahin fehlende bedingte Ausführbarkeit einzelner SIMD-Instruktionsteile hinzugefügt werden.

4 Die Entwicklungsumgebung des CoreVA-Prozessors

Die Entwicklungsumgebung des CoreVA-Prozessors unterteilt sich in die Werkzeugketten zur Hardware- und zur Anwendungsentwicklung. Die Werkzeugkette zur Hardwareentwicklung dient der Erstellung einer Maskenbeschreibung, mit deren Hilfe der CoreVA-Prozessor als Standardzellen-ASIC gefertigt werden kann. Bis zur Erstellung der Maskenbeschreibung werden mehrere Entwurfsschritte durchlaufen, zu denen neben der Synthese und dem Platzieren und Verdrahten umfangreiche Verifikationen der verschiedenen Entwicklungsstufen gehören. Das Abbilden von Programmen auf den CoreVA-Prozessor erfolgt mit Hilfe der Werkzeugkette zur Anwendungsentwicklung, die einen hardwarespezifischen LLVM[1]-Compiler sowie einen zyklenakkuraten Instruktionssatzsimulator beinhaltet.

4.1 Werkzeugkette zur Anwendungsentwicklung

4.1.1 Der LLVM-Compiler

Das LLVM-Projekt bietet eine modulare Compilerarchitektur, mit deren Hilfe der Programmcode einer höheren Programmiersprache wie beispielsweise C oder C++ in einen hardwarespezifischen Maschinencode übersetzt werden kann. Der LLVM-Compiler besteht aus drei Hauptkomponenten, der Vorverarbeitung (auch Precompiler oder Frontend genannt), dem Optimierer (Middleware) und der Endverarbeitung (Backend) (siehe Abbildung 4.1) [50, 85].

Die Vorverarbeitung wird genutzt, um die unterschiedlichen Programmiersprachen in das System einzulesen und nach einer syntaktischen und semantischen Überprüfung in eine plattformunabhängige und universelle LLVM-IR[2] Zwischensprache zu übertragen. Diese Zwischensprache zielt auf einen idealen virtuellen Prozessor

[1] Low Level Virtual Machine
[2] Intermediate Representation

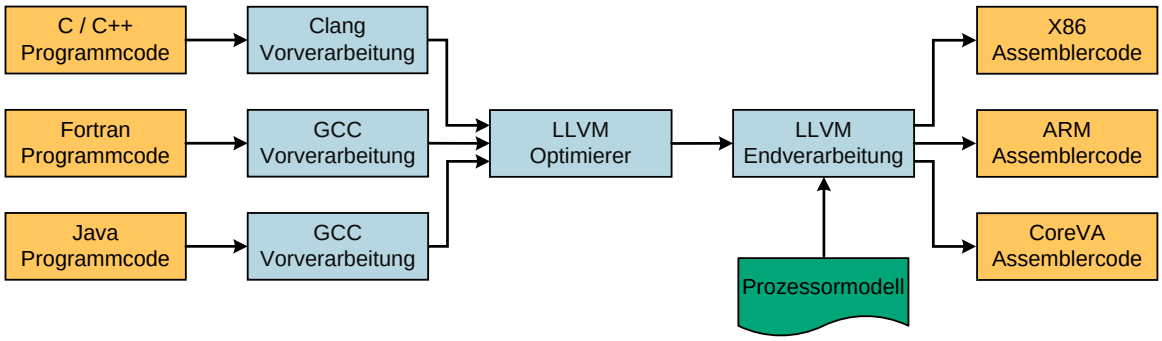

Abbildung 4.1: Hauptkomponenten des LLVM-Compilers

ab, der keinerlei Einschränkungen durch die Prozessorarchitektur erfährt und beispielsweise über eine unendliche Anzahl von Registern verfügt. Die Darstellung des IR-Programmcodes erfolgt in Form eines abstrakten Syntaxbaums[3] (siehe Programmcodebeispiel 7.4 in Kapitel 7). Bei der Erstellung dieses Syntaxbaums wird ein SSA[4]-Ansatz zu Grunde gelegt, bei dem jede Variable nur an einer einzigen Stelle des Programms beschrieben werden kann. Zur Beschreibung von Fallunterscheidungen müssen hierbei zwar virtuelle Phi-Knoten in den Programmcode eingefügt werden (Zeile 4 und 13), Datenabhängigkeiten zwischen Instruktionen sind jedoch eindeutig identifizierbar [50, 85].

In der Werkzeugkette des CoreVA-Prozessors wird das LLVM-Frontend Clang zur Vorverarbeitung des Programmcodes eingesetzt. Als Alternative ließe sich anstelle des Clang auch ein Frontend des GCC[5]-Projekts einbinden, das beispielsweise auch Programmiersprachen wie FORTRAN oder JAVA verarbeiten kann.

Der LLVM-Optimierer führt eine Reihe hardwareunabhängiger IR-zu-IR-Transformationen durch mit dem Ziel, den Programmcode in Bezug auf die Ausführungszeit und die Codegröße zu optimieren. Die Transformationen erfolgen in mehreren hintereinander geschalteten Blöcken, den sogenannten Passes. Diese Blöcke bilden abgeschlossene Optimierungs- oder Analyseschritte, die jeweils auf ein bestimmtes Ziel ausgerichtet sind. Zu diesen Zielen gehört beispielsweise das Entfernen redundanter Programmcodeteile oder das Abrollen von Schleifen. Die Grundform der LLVM-IR Darstellung bleibt während dieser Transformationen jedoch erhalten [50, 85].

Die Endverarbeitung erzeugt schließlich mit Hilfe des LLVM Static Compilers (LLC), beziehungsweise mit Hilfe des darin eingebundenen LLVM Code Generators, aus

[3] Abstract Syntax Tree, AST
[4] Static Single Assignment
[5] GNU Compiler Collection

dem IR-Code einen hardwarespezifischen Assemblercode. Hierbei kommen wiederum diverse Transformationsblöcke zum Einsatz. Zu Beginn der Endverarbeitung wird der IR-Code in einen gerichteten Graphen[6] umgewandelt. Die einzelnen Instruktionen werden durch Knoten repräsentiert, die Operanden und etwaige Datenabhängigkeiten werden durch die zugehörigen Kanten beschrieben. Anschließend werden die Knoten sukzessive durch Instruktionen des vorliegenden Instruktionssatzes ersetzt, wobei konkrete Eigenschaften der Prozessoren, wie beispielsweise die Anzahl vorhandener Funktionseinheiten, das Sprungverhalten oder ein endliches Registerfile berücksichtigt werden. Nachdem der Graph in eine linearisierte Liste von Instruktionsworten übertragen wurde, erfolgt mit Hilfe des CoreVA-spezifischen VLIW-Packetizer- und Reordering-Passes das Zusammenfassen der Instruktionsworte zu VLIW-Instruktionsgruppen [50, 85].

Ein Vorteil der LLVM-Werkzeugkette ist, dass nur die Endverarbeitung an die Eigenschaften des CoreVA-Prozessors angepasst werden muss, da nur an dieser Stelle hardwarespezifische Informationen verarbeitet werden. Die unterschiedlichen Konfigurationen des CoreVA-Prozessors werden hierzu in Prozessormodellen hinterlegt, wodurch der LLVM Code Generator bei seiner Erzeugung automatisch angepasst werden kann. Wie das im Codeausschnitt 4.1 dargestellte Prozessormodell eines CoreVA-Prozessors mit vier Verarbeitungseinheiten zeigt, wird in diesen Modellen die Anzahl der vorhandenen VLIW-Slots und die Zuordnung der MAC- und LD/ST-Funktionseinheiten festgelegt. Hierzu werden Instruktionsstufen (InstrStage) formuliert, die die einzelnen Verarbeitungsschritte einer Instruktion abbilden. Die verschiedenen Instruktionsstufen werden anschließend in einem Instruktionsablauf (InstrItin) verbunden. Da die Instruktionen des CoreVA-Prozessors unterschiedliche Verhalten aufweisen, werden hierbei mehrere Instruktionsabläufe angelegt.

Die Instruktionstypen ALU32, ALU32C, BRANCH, STORE, und MVCEXT können mit Hilfe einer einzelnen Instruktionsstufe modelliert werden. Das erste Argument bedeutet hierbei, dass diese Instruktionen innerhalb eines Taktes verarbeitet werden. Die Liste der Verarbeitungseinheiten legt fest, in welchen VLIW-Slots diese Instruktionstypen ausgeführt werden können. Die abschließende Zahlenliste beschreibt die Latenzen der Eingangs- und Ausgangsdaten, wobei die Latenzen der Ausgänge zuerst aufgeführt werden. Mit Hilfe dieser Liste wird die Verfügbarkeit von Bypasspfaden für das Weiterleiten von Rechenergebnissen bei aufeinander folgenden Instruktionen beschrieben. Zur einheitlichen Modellierung wird das Fehlen von Bypasspfaden hierbei ausschließlich durch das Erhöhen der Ausgangslatenz gekennzeichnet. Da die Rechenergebnisse der ALU32-Instruktionen mit Hilfe des Datenregisterbypasses bereits in der Execute-Stufe an die nachfolgenden Instruktionen weitergeleitet werden können, beträgt die Latenz dieses Instruktionstyps einen Takt. Die Weiterleitung von Rechenergebnissen der ALU32C-Instruktionen

[6] Directed Acyclic Graph, DAG

```
def CoreVAItineraries_4slot :
ProcessorItineraries<[SL0,SL1,SL2,SL3,ME0,ME1,ME2,ME3],[],
[
InstrItin<ALU32,[InstrStage<1,[SL0,SL1,SL2,SL3]>],[1,0,0]>,
InstrItin<ALU32C,[InstrStage<1,[SL0,SL1,SL2,SL3]>],[2,0,0,0]>,
InstrItin<BRANCH,[InstrStage<1,[SL0]>],[0,0]>,
InstrItin<MAC,[InstrStage<1,[SL2,SL3]>,
           InstrStage<1,[ME2,ME3]>],[2,0,0,0]>,
InstrItin<LOAD,[InstrStage<1,[SL0,SL1]>,
           InstrStage<1,[ME0,ME1]>],[2,0,0,0]>,
InstrItin<STORE,[InstrStage<1,[SL0,SL1]>],[0,0,0,0]>,
InstrItin<MVCEXT,[InstrStage<1,[SL0,SL1,SL2,SL3]>],[1,0],[],2>
]>;
```

Programmcode 4.1: Prozessormodell eines CoreVA-Prozessors mit vier Verarbeitungseinheiten

kann aufgrund des fehlenden Condition-Register-Bypasses jedoch frühestens in der Memory-Access-Stufe erfolgen, wodurch die Latenz des Ausgangs auf den Wert 2 gesetzt wird (siehe Kapitel 3.14).

Die Beschreibung der MAC- und LOAD-Instruktionen erfolgt in zwei Instruktionsstufen, da sich die Verarbeitung dieser Instruktionstypen über zwei Pipelinestufen erstreckt (siehe Kapitel 3.8 und 3.9). Anhand dieser Instruktionstypen lässt sich gut die Modellierung einer beschränkten Verfügbarkeit von Funktionseinheiten erläutern. MAC-Instruktionen können in dieser Prozessorkonfiguration ausschließlich in den VLIW-Slots 2 und 3 und LD/ST-Instruktionen ausschließlich in den Slots 0 und 1 ausgeführt werden.

4.1.2 Instruktionssatzsimulator

Der Instruktionssatzsimulator[7] des CoreVA-Prozessors ist ein C-basiertes Softwaremodell, dass das Verhalten des Prozessors bei der Anwendungsverarbeitung nachbildet. Der Simulator ist in der Lage, die Binärdateien eines auszuführenden Programms einzulesen und die darin enthaltenen Instruktionen so zu verarbeiten, wie es in den Spezifikationen des Instruktionssatzes vorgeschrieben ist. Das Registerfile und der Datenspeicher werden hierbei in internen Variablen abgebildet, die zu jeder Zeit sichtbar sind und daher gute Analysemöglichkeiten bieten. Als Basis für eine Verifikation der Hardware kann der Instruktionssatzsimulator aus

[7] Instruction Set Simulator, ISS

diesen Informationen Referenzdateien erzeugen, die die Inhalte der Daten- und Condition-Register sowie die Adressen und Werte von Speicherzugriffen nach jedem Prozessortakt beinhalten. Des Weiteren können Anwendungsentwickler mit Hilfe des Instruktionssatzsimulators die Ausführung ihrer Algorithmen bereits vor der Fertigstellung der Hardwarebeschreibung simulieren [68].

Neben der Verifikation ist der Instruktionssatzsimulator in der Lage, umfangreiche Statistiken über die Programmausführung zu erstellen. Diese Statistiken enthalten unter anderem Informationen über die benötigten Prozessortakte sowie über die Anzahl und den Typ der Instruktionen, die in den einzelnen Verarbeitungseinheiten ausgeführt werden.

4.2 Werkzeugkette zur Hardwareentwicklung

Die Hardwareentwicklung des CoreVA-Prozessors basiert auf dem gängigen Entwurfsablauf digitaler mikroelektronischer Systeme (siehe Abbildung 4.2). Dieser Ablauf wird genutzt, um eine abstrakte Verhaltensbeschreibung schrittweise auf eine gewünschte Zieltechnologie abzubilden [76].

Im Falle des CoreVA-Prozessors erfolgt die Verhaltensbeschreibung auf der Register-Transfer Ebene[8] in der Hardwarebeschreibungssprache VHDL[9]. Wie bereits detailliert in Kapitel 3.17 erläutert wurde, ist diese Hardwarebeschreibung durch den Einsatz generischer Komponenten zur Entwurfszeit hochgradig konfigurierbar. Das Ziel der Hardwareentwicklung ist die Abbildung des CoreVA-Prozessors auf eine 28 nm FD-SOI[10] Standardzellentechnologie der Firma STMicroelectronics. Als Vorgabe an die Werkzeugkette wird in einem Worst-Case Szenario der ungünstigste Fall angenommen, dass der Prozessor mit einer reduzierten Betriebsspannung von 1,0 V und einer Kerntemperatur von 125 °C betrieben wird.

Einen wichtigen Bestandteil der Hardwareentwicklung bildet die funktionale Verifikation der jeweiligen Entwicklungsstufen. Mit der Verifikation wird frühzeitig sichergestellt, dass bei der Erstellung der Verhaltensbeschreibung keine Fehler entstehen und dass die Funktionalität des Prozessors nach dem Synthetisieren und dem Platzieren und Verdrahten konsistent ist. Da eine formale funktionale Verifikation, also das mathematisch bewiesene vollständige Überprüfen aller Schaltzustände, in den meisten Entwicklungsebenen nicht praktikabel ist, wird im Rahmen der CoreVA-Entwicklungsumgebung eine simulationsbasierte funktionale Verifikation

[8] Register Transfer Level, RTL
[9] Very High Speed Integrated Circuit Hardware Description Language
[10] Fully Depleted - Silicon On Insulator

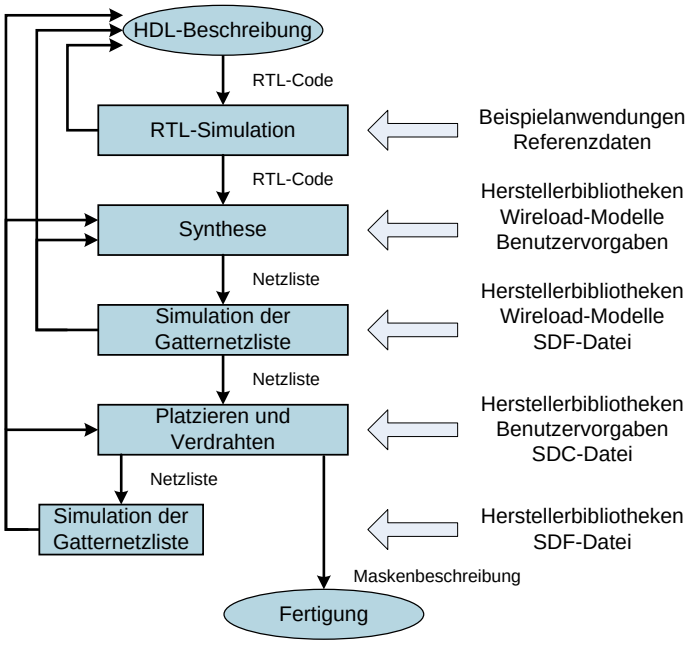

Abbildung 4.2: Entwurfsablauf digitaler mikroelektronischer Systeme

durchgeführt. Bei dieser Verifikation wird das Verhalten des Prozessors bei der Verarbeitung von Beispielanwendungen auf den verschiedenen Simulationsebenen analysiert. Die Korrektheit des Prozessors kann hierbei entweder durch eine Überprüfung der erzeugten Rechenergebnisse oder durch das Vergleichen der Daten- und Condition-Registerinhalte mit Soll-Vorgaben des Instruktionssatzsimulators bestimmt werden [20, 89].

In der Werkzeugkette kommt eine Vielzahl unterschiedlicher Skripte zum Einsatz, die die einzelnen Schritte des Entwurfsablaufs fast vollständig automatisieren. So lässt sich sowohl die Synthese als auch das Platzieren und Verdrahten durch einen einzelnen Befehlsaufruf durchführen. Die Verifikation kann mit Hilfe eines Regressionstests durch automatisiertes Kompilieren, Simulieren und Auswerten ebenfalls selbstständig erfolgen. Des Weiteren sind die einzelnen Werkzeuge der Hardwareentwicklung in der Lage, nach jedem Entwicklungsschritt Aussagen bezüglich der Leistungsfähigkeit und des Ressourcenbedarfs des späteren Prozessors zu treffen. Hierbei kommen unterschiedliche Abschätzungsverfahren zum Einsatz, die in den jeweiligen Entwicklungs- und Simulationsebenen integriert sind und Informationen über die benötigte Leistungsaufnahme und Chipfläche sowie die maximalen Betriebsfrequenz des Prozessors liefern [76].

4.2.1 Simulation auf Register-Transfer-Ebene

In dem ersten Schritt des Entwurfsablaufs erfolgt die funktionale Verifikation der Verhaltensbeschreibung auf der Register-Transfer-Ebene. Die Verifikation basiert auf Simulationen, die mit dem HDL-Simulationsprogramm ModelSim der Firma Mentor Graphics Corporation durchgeführt werden. Nach einer anfänglichen Überprüfung der VHDL-Syntax erstellt ModelSim eine Simulationsumgebung, die auf der RTL-Beschreibung des CoreVA-Prozessors basiert. Um das Ausführen verschiedener Anwendungen zu simulieren, wird der CoreVA-Prozessor in eine zentrale Testbench[11] eingebunden, die den Daten- und Instruktionsspeicher des Prozessors befüllt und die Programmausführung startet. Neben dem Beschreiben der Speicher ist die Testbench in der Lage, Eingabedateien einzulesen und Rechenergebnisse in Ausgabedateien zu schreiben. Des Weiteren kann sie zur Ermittlung der Laufzeit der jeweiligen Anwendung genutzt werden [55, 76].

Um die Inhalte der Daten- und Condition-Register nach jedem Prozessortakt überprüfen zu können, wird eine Schnittstelle zur Einzelschritt-Fehlersuche (Trace-Einheit) genutzt, die sich innerhalb des Prozessorkerns befindet und dem Zusammenfassen und Herausführen interner Prozessorsignale dient. In Kombination mit der FLI[12]-Schnittstelle von ModelSim können somit die benötigten Registerinhalte aus dem CoreVA-Prozessor ausgelesen und in einer Datei abgespeichert werden. Diese Datei kann anschließend direkt mit der Referenzdatei des Instruktionssatzsimulators verglichen werden [89, 55].

4.2.2 Die Logiksynthese

Nach der erfolgreichen RTL-Simulation überträgt die Logiksynthese die Verhaltensbeschreibung des CoreVA-Prozessors in eine Strukturbeschreibung. Hierzu überführt das Synthesewerkzeug Cadence Encounter RTL Compiler den RTL-Code in eine Gatternetzliste, die die verwendeten Standardzellen sowie deren Verschaltung beinhaltet. Die Synthese erfolgt unter Einbeziehung der Herstellerbibliotheken, die Informationen zum Zeitverhalten und zur Geometrie der Standardzellen beinhalten. Des Weiteren werden Benutzervorgaben berücksichtigt, die dem Synthesewerkzeug über Aufrufparameter und separate sdc[13]-Konfigurationsdateien übergeben werden. Zu diesen Benutzervorgaben gehören beispielsweise die anvisierte Periodendauer sowie die Zeitvorgaben zur Berücksichtigung der Eingangs- und Ausgangsverzögerungen der NoC-Schnittstelle [13, 76].

[11] In VHDL beschriebene Testumgebung
[12] Foreign Language Interface
[13] Synopsys Design Constraints

Zum Erzeugen der Gatternetzliste werden mehrere Arbeitsschritte durchlaufen (siehe Abbildung 4.3). In einem ersten Schritt wird der RTL-Code eingelesen und bezüglich seiner Semantik überprüft (Elaborieren). Anschließend werden die Benutzervorgaben angewendet und die Logiksynthese gestartet. Bei der Logiksynthese wird der RTL-Code kompiliert und in eine technologieunabhängige Netzliste abgebildet. Anschließend wird diese Netzliste tiefergehend optimiert und in eine temporäre Gatternetzliste übertragen. Hierbei versucht das Syntheseprogramms die Latenz (Register-zu-Register-Verzögerungszeit) aller Signalpfade an die vorgegebene Periodendauer anzupassen.

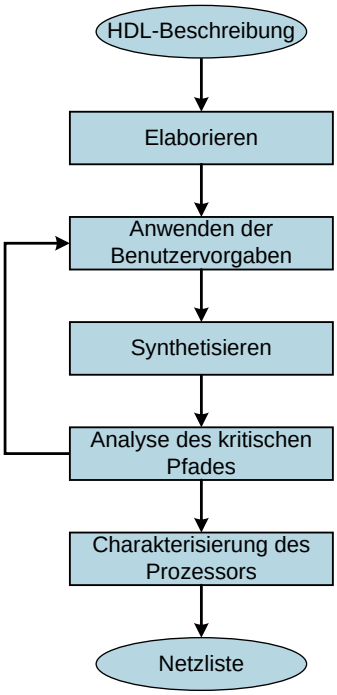

Abbildung 4.3: Ablauf der Logiksynthese

Auf Basis der temporären Gatternetzliste erfolgt eine erste Analyse des kritischen Pfades. Der kritische Pfad eines Prozessors ist der Signalpfad, der die größte Latenz aufweist und folglich die maximale Betriebsfrequenz des Prozessors vorgibt. Obwohl die Leitungslängen und die Größen der parasitären Kapazitäten während der Synthese noch nicht bekannt sind, kann die Latenz der einzelnen Signalpfade mit Hilfe von Wireload-Modellen abgeschätzt werden. Diese Modelle liefern statistische Durchschnittswerte über die Widerstände, parasitäre Kapazitäten und Flächen der Verbindungsnetze [76].

Falls die Latenz des kritischen Pfades die vorgegebene Periodendauer überschreitet und der Prozessor mit einer geringeren Taktfrequenz betrieben werden muss, weisen die Standardzellen die nicht im kritischen Pfad liegen gegebenenfalls eine nicht be-

nötigte Leistungsfähigkeit auf. Sie können dementsprechend durch Standardzellen mit geringerer Treiberleistung ersetzt werden, wodurch ihre Leistungsaufnahme und Fläche reduziert werden kann. Aus diesem Grund erfolgt nach der Analyse des kritischen Pfades gegebenenfalls ein weiterer Optimierungsschritt mit angepasster Periodendauer [13].

Nachdem die finale Gatternetzliste des CoreVA-Prozessors erzeugt wurde, generiert das Synthesewerkzeug eine Vielzahl von Dateien, die den Prozessorkern charakterisieren. Neben einer abschließenden Analyse des kritischen Pfades wird die Chipfläche und Leistungsaufnahme des CoreVA-Prozessors bestimmt. Die Chipfläche wird hierbei in die Fläche der verwendeten Standardzellen und in die Fläche zur Verschaltung der Standardzellen unterteilt. Bei der Bestimmung der Leistungsaufnahme wird zwischen der dynamischen und der statischen Verlustleistung unterschieden. Dynamische Verluste entstehen während eines Schaltvorgangs durch das Umladen von parasitären Kapazitäten und durch Querströme (Kurzschlussströme), die durch den p-Kanal- und n-Kanal-Teil einer CMOS[14]-Schaltung fließen. Statische Verluste entstehen durch Leckströme, die auch in Ruhephasen in Sperrrichtung über die p-n-Übergänge der Transistoren fließen [76, 84].

4.2.3 Simulation der Gatternetzliste und Aufnahme der Schaltaktivitäten

Nach der Synthese erfolgt die Überprüfung der Gatternetzliste durch eine erneute ModelSim-Simulation (pre-Layout Gatelevel-Simulation). Hierbei kommt eine mit der RTL-Simulation vergleichbare Testbench zum Einsatz, die jedoch nicht die VHDL-Beschreibungen des CoreVA-Prozessors einbindet sondern die Gatternetzliste verwendet. Neben der Gatternetzliste und den Herstellerbibliotheken wird eine vom Synthesewerkzeug erstellte sdf[15]-Datei benutzt, die Auskünfte über das Zeitverhalten der Standardzellen liefert. Die Verzögerungszeiten der Verbindungsnetze werden wiederum mit Hilfe von Wireload-Modellen abgeschätzt [76].

Die Verifikation des Systems erfolgt wiederum mit Hilfe von Ausgabendateien, die die Rechenergebnisse der Anwendungen beinhalten oder durch das Auswerten der Daten- und Condition-Registerinhalte nach jedem Prozessortakt. Im Vergleich zur RTL-Simulation lässt sich die Güte der Verifikation durch die Simulation der Gatternetzliste deutlich verbessern, da das Zeitverhalten der Standardzellenbibliotheken viel näher an der Realität liegt und das Verhalten des Prozessors bei der tatsächlichen Betriebsfrequenz simuliert werden kann.

[14] Complementary Metal Oxide Semiconductor
[15] Standard Delay Format

Die Simulation der Gatternetzliste kann neben der Verifikation auch zu einer verbesserten Abschätzung der durchschnittlichen Leistungsaufnahme eingesetzt werden. Wie bereits in Kapitel 4.2.2 erwähnt wurde, ist der dynamische Anteil der Leistungsaufnahme abhängig von den Schaltaktivitäten der jeweiligen Standardzellen. Die Logiksynthese geht bei der Abschätzung der Leistungsaufnahme von einer durchschnittlichen Schaltaktivität von 10% an den Eingängen aller Standardzellen aus. Die Charakteristik der auszuführenden Anwendungen wie beispielsweise der Grad ihrer Parallelität oder die Auslastung der MAC- und LD/ST-Einheiten wird hierbei nicht berücksichtigt. Die Simulation der Gatternetzliste ist hingegen in der Lage, die tatsächlichen Schaltaktivitäten der Standardzellen bei der Verarbeitung der Anwendungen aufzuzeichnen. Hierzu wird während der Simulation eine vcd[16]-Datei generiert, die eine Auflistung der Schaltzustände aller Netze beinhaltet. Mit Hilfe dieser Datei kann das Programm Encounter Power System der Firma Cadence anschließend die durchschnittliche Leistungsaufnahme des Prozessors anhand realistischer Anwendungsszenarien bestimmen [76, 84].

4.2.4 Platzieren und Verdrahten

Das Platzieren und Verdrahten (Place and Route) bildet den letzten Entwicklungsschritt des Entwurfsablaufs. In diesem Schritt wird die Strukturbeschreibung des CoreVA-Prozessors in eine physikalische Maskenbeschreibung (Layout) überführt, mit deren Hilfe ein Standardzellen-ASIC gefertigt werden kann [12, 76].

Wie der Name dieses Entwicklungsschritts suggeriert, basiert die Erstellung der Maskenbeschreibung auf dem Platzieren und Verdrahten sämtlicher Standardzellen auf der späteren Chipfläche. Hierbei kommt das Entwicklungsprogramm Cadence Encounter Digital Implementation System zum Einsatz. Dieses Programm verarbeitet wiederum die Herstellerbibliotheken und Benutzervorgaben bezüglich der Platzierungsdichte (Utilization) und der Anzahl der Optimierungsdurchläufe. Des Weiteren wird die Gatternetzliste der Logiksynthese und eine aktualisierte sdc-Datei eingelesen, über die das Synthesewerkzeug zusätzliche Zeit-, Flächen- und Leistungsvorgaben übermitteln kann.

Das Erstellen der Maskenbeschreibung beginnt mit einer manuellen Flächeneinteilung (Floorplan), bei der der Entwickler die Anordnung der Speicherblöcke, I/O-Zellen und Versorgungsleitungen festlegt. Anschließend erfolgt das automatisierte Platzieren der Standardzellen auf dem Teil der Chipfläche, der nicht durch die oben genannten Komponenten blockiert ist. Hierbei wird bereits im Vorfeld versucht, den späteren Verdrahtungsaufwand durch geschicktes Gruppieren der

[16] Value Change Dump

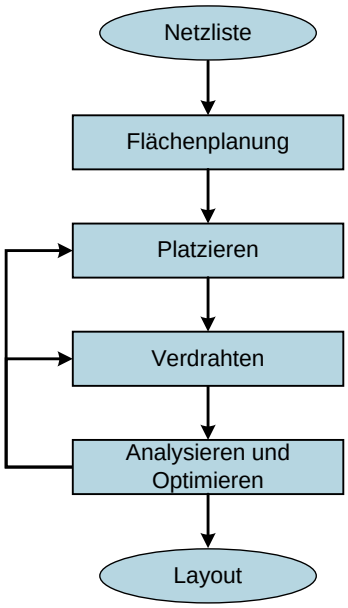

Abbildung 4.4: Ablauf des Platzierens und Verdrahtens

Zellen zu minimieren. Des Weiteren wird die vorgegebene Platzierungsdichte berücksichtigt, die vorschreibt, wie viel Freiraum zwischen den Standardzellen für die Verbindungsleitungen gelassen werden soll.

Nachdem alle Standardzellen platziert wurden, werden sie sowohl untereinander als auch mit Takt- und Versorgungsleitungen verbunden. Wie in Abbildung 4.4 ersichtlich ist, können die Verbindungen der Standardzellen in mehreren Durchläufen optimiert werden, um sicherzustellen, dass die entstandenen Signalpfade die vorgegebenen Randbedingungen einhalten. Zu diesen Randbedingungen zählt beispielsweise die Einhaltung der setup- und hold-Zeiten. Falls einzelne Pfade die Vorgaben nicht einhalten, werden diese Pfade entweder durch das Umsortieren der Standardzellen oder durch das Verändern der Treiberstärken korrigiert.

Nach der erfolgreichen Erstellung der Maskenbeschreibung und einer Aktualisierung der Gatternetzliste muss die Funktionalität des Prozessors durch erneute Simulationen sichergestellt werden (post-Layout Gatelevel-Simulation). Im Unterschied zur pre-Layout Simulation können nun die tatsächlichen parasitären Kapazitäten und Leitungslängen aus der Maskenbeschreibung extrahiert werden, wodurch der Einsatz von Wireload-Modellen entfallen kann und eine maximale Simulationsgenauigkeit erreicht wird. Des Weiteren stehen nun die tatsächlichen Werte für die maximale Betriebsfrequenz und die Chipfläche zur Verfügung [12, 76].

4.3 Zusammenfassung und eigene Beiträge

In diesem Kapitel wurde die Entwicklungsumgebung des CoreVA-Prozessors beschrieben. Die Entwicklungsumgebung besteht aus einer automatisierten Werkzeugkette zur Hardwareentwicklung und aus diversen Werkzeugen, die in der Anwendungsentwicklung Verwendung finden. Die Werkzeugkette zur Hardwareentwicklung bildet die Verhaltensbeschreibung des CoreVA-Prozessors schrittweise auf eine 28 nm Standardzellentechnologie der Firma STMicroelectronics ab. Hierbei kommen Programme der Firma Cadence zum Einsatz, die die Synthese und das Platzieren und Verdrahten der Standardzellen übernehmen. Die Werkzeugkette zur Anwendungsentwicklung besteht aus einem zyklenakkuraten Instruktionssatzsimulator und aus einem LLVM-Compiler, der um eine CoreVA-spezifische Endverarbeitung erweitert wurde.

Neben der eigentlichen Hardware- und Anwendungsentwicklung bildet das Zusammenspiel dieser Werkzeugketten eine Basis für funktionale Verifikationen, die sicherstellen, dass die Verhaltensbeschreibung keine Fehler beinhaltet und dass die Funktionalität des Prozessors nach dem Synthetisieren und dem Platzieren und Verdrahten konsistent ist. Hierzu simuliert der Instruktionssatzsimulator das Ausführen bestimmter Beispielanwendungen und erzeugt Referenzdateien, die die Registerinhalte jedes Prozessortaktes und die Adressen und Werte etwaiger Speicherzugriffe beinhalten. Die Referenzdateien dienen anschließend als Vergleichsgrundlage für Hardwaresimulationen, bei denen die Verhaltensbeschreibung des Prozessors oder eine Gatternetzliste zur Simulation der Anwendungsverarbeitung herangezogen wird.

Im Rahmen dieser Arbeit wurde ein skriptgesteuerter Regressionstest entwickelt, der die Funktionalität sämtlicher Prozessorkonfigurationen sicherstellen kann. Der Regressionstest passt hierzu den Compiler, den Instruktionssatzsimulator und die Verhaltensbeschreibung an die gewünschte Prozessorkonfiguration an. Für jede Anwendung und jede zu untersuchende Prozessorkonfiguration erfolgt anschließend ein Durchlauf des Instruktionssatzsimulators zur Ermittlung der Referenzwerte sowie eine automatisierte Synthese mit anschließender Hardwaresimulation. Da die funktionale Verifikation auf einem konsistenten Verhalten zwischen dem Instruktionssatzsimulator und den Hardwaresimulationen basiert, mussten der Instruktionssatzsimulator und der LLVM-Compiler im Laufe der Entwicklung vielfach an geplante Hardwareänderungen angepasst werden. So wurde beispielsweise ein Modell der Bypasspfade in die Endverarbeitung des Compilers hinzugefügt und die Auswertung etwaiger Wartezyklen überarbeitet (siehe Kapitel 4.1.1 und 7.1).

Neben der Verifikation werden die im Regressionstest genutzten Automatismen auch zur Ermittlung der in den Kapiteln 7 und 8 vorgestellten Werte eingesetzt.

Der Instruktionssatzsimulator wurde in dieser Arbeit dahingehend erweitert, dass er umfangreiche Statistiken über die Programmausführung erstellen kann. Hierzu gehören beispielsweise die Anzahl der benötigten Prozessortakte und die Auswertung der Instruktionstypen, die in den einzelnen Verarbeitungseinheiten ausgeführt werden. Der LLVM-Compiler wurde durch einen speziellen Transformationsblock ergänzt, der zum Einen die Instruktionsgruppen des Programmcodes analysiert und die enthaltenen Instruktionen und Instruktionstypen ermittelt, und zum Anderen Sprunginstruktionen auswertetet, um auf Schleifenzugehörigkeiten schließen zu können (siehe Kapitel 7.4.3 und 7.4.4.1). Die Werkzeugkette zur Hardwareentwicklung wurde schließlich so ausgebaut, dass sich die Ressourcenanforderungen der jeweiligen Prozessorkonfigurationen skriptgesteuert ermitteln und dokumentieren lassen. Die anwendungsspezifische Leistungsaufnahme wird hierbei auf Basis von Standardzellensynthesen und automatisierten Aufzeichnungen der Schaltaktivitäten bestimmt (siehe Kapitel 4.2.3).

5 Grundlagen der Entwurfsraumexploration

Die Entwurfsraumexploration[1] bildet einen entscheidenden Schritt in der Entwicklung anwendungsspezifischer Prozessoren. Das Ziel der Entwurfsraumexploration ist, einen Prozessor zu spezifizieren, der eine gewünschte Leistungsfähigkeit bei möglichst geringem Ressourcenaufwand erreicht. Hierzu wird ein Entwurfsraum aufgespannt, der die variablen Konfigurationselemente des Prozessors beinhaltet. Aus diesem Entwurfsraum wird eine Vielzahl von Prozessorkonfigurationen ermittelt, die in der sogenannten Konfigurationsmenge zusammengefasst werden. Die einzelnen Elemente der Konfigurationsmenge werden bezüglich ihrer Leistungsfähigkeit bei der Verarbeitung der Zielanwendungen untersucht. Im Anschluss hieran wird die Leistungsfähigkeit den Ressourcenanforderungen gegenübergestellt, die sich aus der Prozessorfläche und der Leistungsaufnahme ergeben. Da die Leistungsfähigkeit und die Ressourcenanforderungen häufig in Wechselwirkung zueinander stehen, werden schließlich verschiedene Lösungsverfahren angewandt, um die Prozessorkonfiguration auszuwählen, die die geforderte Charakteristik am effizientesten erfüllt [61, 73].

5.1 Ermittlung der Leistungsfähigkeit und der Ressourcenanforderungen

Bei der Entwurfsraumexploration wird die Leistungsfähigkeit eines Prozessors meistens durch die Anzahl der Berechnungen definiert, die innerhalb eines bestimmten Zeitraums durchgeführt werden. Als Maßeinheit kommt hierbei die durchschnittliche Anzahl der Instruktionen pro Sekunde[2] oder die Laufzeit einer Anwendung zum Einsatz (siehe Formel 5.1) [73].

[1] Exploration (lateinisch exploratio) bedeutet Erforschung oder Untersuchung (Duden)
[2] Instructions Per Second, IPS

$$Laufzeit = \frac{Prozessortakte}{Frequenz} = \frac{Instruktionen}{Parallelität \cdot Frequenz} \tag{5.1}$$

Die Leistungsfähigkeit eines Prozessors kann dementsprechend entweder durch die Erhöhung der Betriebsfrequenz oder durch die Erweiterung der Parallelität gesteigert werden. Eine Erhöhung der Frequenz hat hierbei einen direkten Einfluss auf die Anzahl der Prozessortakte, die pro Sekunde verarbeitet werden[3]. Die Parallelität erhöht hingegen die Menge der Instruktionen, die pro Prozessortakt berechnet werden können[4] [33].

Unter der durchschnittlichen Parallelität, beziehungsweise unter dem Grad der Parallelverarbeitung, ist im Folgenden die durchschnittliche Auslastung der Verarbeitungseinheiten (Slots) zu verstehen. Sie ist definiert als eine Division der Anzahl der Instruktionen durch die Anzahl der Prozessortakte inklusive etwaiger Wartezyklen (siehe Formel 5.2) [40].

Die aus einer N-fachen Parallelverarbeitung resultierende Leistungssteigerung ist hingegen durch einen Vergleich der Laufzeiten definiert (siehe Formel 5.3). Eine andere Möglichkeit der Definition ist nach Amdahl durch die Formel 5.4 gegeben. Im Unterschied zur vorherigen Betrachtungsweise basiert diese Definition nicht auf der Parallelität eines Programms, sondern auf dessen Parallelanteil (P), sprich dem Anteil des Programmcodes, der überhaupt parallelisierbar ist [5].

$$Parallelität = \frac{Instruktionen}{Prozessortakte} \tag{5.2}$$

$$Leistungssteigerung = \frac{Laufzeit_{1Slot}}{Laufzeit_{NSlot}} = \frac{Prozessortakte_{1Slot}}{Prozessortakte_{NSlot}} \tag{5.3}$$

$$Leistungssteigerung\ nach\ Amdahl = \frac{1}{1 - P + \frac{P}{N}} \tag{5.4}$$

Es existiert eine Vielzahl von Veröffentlichungen, in denen beschrieben wird, wie anwendungsspezifische Entwurfsraumexplorationen möglichst effizient durchgeführt werden können. Hierbei kristallisieren sich bei der Bestimmung der Laufzeit und der Ressourcenanforderungen zwei unterschiedliche Herangehensweisen heraus, die experimentelle und die modellbasierte Entwurfsraumexploration.

[3] Cycles Per Second, CPS
[4] Instructions Per Cycle, IPC

5.1.1 Experimentelle Entwurfsraumexplorationen

Bei der experimentellen Entwurfsraumexploration werden die Auswirkungen der unterschiedlichen Prozessorkonfigurationen auf die Betriebsfrequenz, Leitungsaufnahme und Prozessorfläche durch eine Vielzahl konfigurationsspezifischer Synthesen bestimmt. Da die Laufzeit der Anwendungsverarbeitung ebenfalls von den Prozessorkonfigurationen abhängt, muss diese sogar für jede Zielanwendung auf jeder zu untersuchenden Konfiguration ermittelt werden.

Ein Beispiel für dieses Vorgehen bilden die Entwurfsraumexplorationen von Wong [83], Seedorf [75] und Saptono [73], bei denen die unterschiedlichen Konfigurationen eines ρ-VEX-Prozessors bewertet werden. Zur Ermittlung des Laufzeitverhaltens kommen in diesen Arbeiten entweder Instruktionssatzsimulatoren und Hardwaresimulationen zum Einsatz [73, 83], oder die Anwendungsanalyse erfolgt auf einer prototypischen Abbildung des Prozessors auf einem FPGA-basierten System [75]. Die Ressourcenanforderungen der verschiedenen Konfigurationen werden bei Wong [83] und Seedorf [75] mit Hilfe von Xilinx ISE bestimmt.

Die Analyseergebnisse der experimentellen Entwurfsraumexplorationen weisen typischerweise nur sehr geringe Ungenauigkeiten auf. Eine Voraussetzung für die Korrektheit der Simulationsergebnisse ist jedoch, dass die Compiler-Werkzeugkette an die unterschiedlichen Konfigurationen angepasst werden kann und dass die jeweiligen Instruktionssatzsimulatoren oder Hardwaresimulationen zyklenakkurat arbeiten.

Das Ermitteln der Ressourcenanforderungen ist in den Arbeiten von Wong [83] und Seedorf [75] problemlos möglich, da die Konfigurationsmenge des ρ-VEX-Prozessors im Vorfeld auf drei bis vier Konfigurationen begrenzt wurde. Entwurfsraumexplorationen mit größeren Entwurfsräumen sind durch das häufige Synthetisieren und Simulieren jedoch äußerst zeitaufwändig und ab einer gewissen Anzahl variabler Elemente praktisch nicht mehr zu realisieren. Aus diesem Grund musste die Bestimmung der Ressourcenanforderungen in den Untersuchungen von Saptono [73] auf eine sehr ungenaue Flächenabschätzung reduziert werden, da in dieser Arbeit mit achtzehn verschiedenen Konfigurationselementen (9 Prozessorkern, 9 Speicher) ein deutlich größerer Entwurfsraum aufgespannt wird.

5.1.2 Modellbasierte Entwurfsraumexplorationen

Die zweite Herangehensweise verfolgt einen Ansatz auf abstrakterem Niveau und nutzt Software- und Hardwaremodelle, um die Leistungsfähigkeit und die Ressourcenanforderungen zu ermitteln. Die Softwaremodelle werden mit Hilfe von Anwendungsanalysen aufgestellt, die Aussagen über die erwartete Laufzeit der Zielanwendungen treffen, indem sie die durchschnittliche Parallelität bestimmen oder die Anzahl der verschiedenen Instruktionstypen auswerten. Hierbei kommen entweder statische Programmcodeanalysen oder einzelne konfigurationsunabhängige Simulationen zum Einsatz. Die Hardwaremodelle, die der Bestimmung der Ressourcenanforderungen dienen, werden häufig auf Basis weniger Probesynthesen oder anderen Abschätzungsverfahren aufgestellt.

Die Ergebnisse, die mit einem modellbasierten Ansatz gewonnen werden, können gegebenenfalls hohe Ungenauigkeiten aufweisen. Die Exploration eines großen Entwurfsraums ist bei einer modellbasierten Analyse jedoch bei Weitem nicht so zeitintensiv. Als Beispiele für modellbasierte Entwurfsraumexplorationen seien im Folgenden die Veröffentlichungen von Wang [81], Moseley [57], Kambadur [43], Jordans [40] und Ascia [7] genannt.

In der Arbeit von Wang [81] wird ein analytisches Modell vorgestellt, dass die Auswirkungen einer variablen VLIW-Länge, SIMD-Breite und Prozessorkernanzahl in einem Multiprozessorsystem beschreibt. Das vorgestellte Verfahren basiert auf dem Modell zur Beschreibung der Leistungsfähigkeit von Multiprozessoren von Hill und Marty [39], welches wiederum auf Amdahls Gesetz beruht (siehe Formel 5.4) [5]. Im Rahmen dieser Arbeit wurde das Modell von Hill und Marty um die VLIW- und SIMD-Funktionalitäten sowie um die Modellierung der Inter- und Intra-Core Kommunikation ergänzt. Anschließend wurde das Modell durch hardwarespezifische Vorgaben erweitert, um die Prozessorfläche und die Laufzeit ermitteln zu können. Die hier vorgestellte Form der Entwurfsraumexploration kommt nach anfänglichen Probesynthesen ohne weitere konfigurationsspezifische Synthesen oder Simulationen aus. Der Parallelanteil der jeweiligen Zielanwendungen, der einen maßgeblichen Einfluss auf die tatsächliche Auslastung der Prozessorkomponenten hat, wird an dieser Stelle jedoch nicht analysiert, sondern als konstanter Wert vorgegeben. Des Weiteren konnte zu der Zeit dieser Veröffentlichung die Leistungsaufnahme noch nicht ausgewertet werden.

Die Arbeiten von Moseley [57] und Kambadur [43] beschreiben, wie der Parallelanteil einer Anwendung mittels einmaliger hardwareunabhängiger Simulationen bestimmt werden kann. Dazu analysiert das von Moseley [57] vorgestellte Programm „LoopProf“ während der Programmausführung die Auftrittshäufigkeit bestimmter Basisblöcke. Diese Basisblöcke bilden Untergruppen des Programmcodes und

zeichnen sich dadurch aus, dass sie jeweils nur einen Eingangs- und Ausgangspunkt besitzen und neben den Sprüngen zum Verlassen der Blöcke keine weiteren Sprünge beinhalten (siehe Kapitel 7.3). Da hierdurch garantiert ist, dass bei dem Aufruf eines Basisblocks alle enthaltenen Instruktionen genau einmal ausgeführt werden, kann von der Auftrittshäufigkeit auf die Anzahl von Schleifendurchläufen und somit auf einen potentiell parallelisierbaren Programmanteil geschlossen werden. LoopProf arbeitet hierbei auf Basis einer hardware- und compilerunabhängigen Funktionsbibliothek, die eine Anwendungsanalyse auf handelsüblichen Linux-Systemen ermöglicht. In der Arbeit von Kambadur [43] werden mit Hilfe von „Harmony“, einer Erweiterung des LLVM-Compilers, spezielle Arrays[5] in den Programmcode eingefügt, die ebenfalls die Basisblockaufrufe zur Laufzeit aufzeichnen.

Im Gegensatz zu den vorangegangenen Methoden beschreibt Jordans [40] verschiedene Verfahren, um den Parallelitätsgrad durch statische Programmcodeanalysen abschätzen zu können. Die Analysen basieren auf den Ergebnissen eines zusätzlichen Transformationsblocks, der in den LLVM-Compiler eingefügt wurde, um zu prüfen, in welchem Maße der Scheduler die Berechnungen eines Basisblocks parallelisieren kann (siehe Kapitel 4.1.1). Durch diesen Ansatz ist Jordans [40] in der Lage, völlig ohne Simulationen auf die benötigte Anzahl paralleler Verarbeitungseinheiten zu schließen. Da der Transformationsblock jedoch auf der plattformunabhängigen LLVM-IR Zwischensprache arbeitet, können hardwarespezifischen Einschränkungen, wie beispielsweise eine begrenzte Anzahl von Multiplizierern oder Datenregistern, nicht berücksichtigt werden.

In der Literatur findet sich jedoch auch eine Fülle von Mischformen der oben genannten Herangehensweisen. In der Arbeit von Ascia [7] kommt mit dem open-source Projekt „Trimaran“ und dem darin enthaltenen IMPACT-Compiler beispielsweise eine konfigurierbare Compiler- und Simulationsumgebung zum Einsatz, die mit Hilfe eines Instruktionssatzsimulators konfigurationsspezifische Simulationen durchführt, um das Laufzeitverhalten der Zielanwendungen auf den verschiedenen Prozessorkonfigurationen zu bestimmen. Des Weiteren werden Statistiken erstellt, die die Anzahl und den Typ der ausgeführten Instruktionen beinhalten und neben den Wartezyklen auch Cache-Hit und Miss-Zugriffen protokollieren. Im Unterschied zu den vorangegangenen Beispielen zielt diese Arbeit jedoch nicht auf der Entwurfsraumexploration eines konkreten Prozessors ab, sondern auf der Erstellung des EPIC-Explorers, einer Arbeitsplattform zur Entwicklung effizienter Lösungsverfahren. Das zugrunde liegende Prozessorsystem besteht dementsprechend aus einem fiktiven VLIW-Prozessor, der einen Entwurfsraum mit einer Konfigurationsmenge von $1{,}47 \cdot 10^{13}$ Prozessorkonfigurationen aufspannt [7].

[5] Parallel Block Vector, PBV

Die Ressourcenanforderungen dieses Prozessors werden aus diesem Grund nur durch relativ abstrakte Modelle beschrieben. Die Prozessorfläche wird modelliert, indem zu der Grundfläche eines Prozessors mit minimalem Funktionsumfang die Flächen der konfigurationsabhängigen Funktionsblöcke hinzu addiert werden. Zu diesen Funktionsblöcken zählen beispielsweise die verschiedenen Fest- und Fließkomma-Funktionseinheiten, die konfigurierbare Sprungvorhersage, das Registerfile oder die Speicher. Das Modell zur Ermittlung der Leistungsaufnahme basiert ebenfalls auf einem Grundwert und den Mehraufwänden für die variablen Funktionsblöcke. Zur Modellierung der dynamischen Leistungsaufnahme werden zusätzlich die in der Anwendungsanalyse ermittelten Instruktionstypen berücksichtigt, um auf die Auslastung der Funktionsblöcke zu schließen.

5.2 Lösungsverfahren für Mehrzieloptimierungsprobleme

Da die Leistungsfähigkeit und die Ressourcenanforderungen häufig in direkter Wechselwirkung zueinander stehen, ist das Finden einer optimalen Prozessorkonfiguration durch ein Mehrzieloptimierungsproblem definiert. Zur Lösung dieses Mehrzieloptimierungsproblems gibt es verschiedene Ansätze, in denen die Bewertungskriterien zu einer gemeinsamen Zielfunktion zusammengefasst werden, um anschließend eine eindeutige Lösung finden zu können. So beschreibt beispielsweise eine Multiplikation aus Leistungsaufnahme und Laufzeit die Energie[6], die benötigt wird, um eine Zielanwendung zu verarbeiten (siehe Formel 5.5). Das Ziel der Entwurfsraumexploration sollte in diesem Fall die Minimierung der benötigten Energie sein. Die Energieeffizienz eines Systems wird hingegen durch den Kehrwert der Energie beschrieben (siehe Formel 5.6). Sie sollte folglich durch eine Entwurfsraumexploration gesteigert werden [52].

$$Energie = Leistung \cdot Laufzeit = \frac{Leistung \cdot Prozessortakte}{Frequenz} \tag{5.5}$$

$$Energieeffizienz = \frac{1}{Leistung \cdot Laufzeit} \tag{5.6}$$

Die Ressourceneffizienz berücksichtigt neben der Leistungsaufnahme und der Laufzeit zusätzlich die Fläche eines Prozessors (siehe Formel 5.7). Abhängig von den

[6] Power-Delay-Product, PDP

vorgegebenen Zielen können die einzelnen Bewertungskriterien durch die Exponenten x, y, z entweder besonders gewichtet, oder aus der Auswertung herausgenommen werden. Die Energieeffizienz kann durch diese Formel dargestellt werden, indem die Fläche mit dem Exponenten 0 gewichtet wird (x=1, y=1). Bei der Energie-Laufzeit-Effizienz, die aus dem Kehrwert des Energie-Laufzeit-Produkts[7] gebildet wird, nimmt die Laufzeit hingegen einen quadratischen Einfluss auf die Auswahl der Prozessorkonfiguration (x=1, y=2). Diese Gewichtung kann gewählt werden, um hohen Durchsatzanforderungen gerecht zu werden [52].

$$Ressourceneffizienz = \frac{1}{Leistung^x \cdot Laufzeit^y \cdot Fläche^z} \tag{5.7}$$

Ein ähnliches Verfahren wird in der Entwurfsraumexploration von Saptono [73] genutzt. Bei diesem Verfahren werden die Kosten der einzelnen Konfigurationselemente und die resultierende Laufzeit jedoch aufsummiert (siehe Formel 5.8). Das Aufsummieren ermöglicht im Unterschied zur Multiplikation eine feingranularere Gewichtung der einzelnen Bewertungskriterien. Diese Kostenfunktion beschreibt eine optimale Konfiguration, wenn die Summe der Kosten und der Laufzeit ein Minimum bildet.

$$Kostenfunktion = Laufzeit + Hardwarekosten = Laufzeit + \sum_i Anzahl_i \cdot Kosten_i \tag{5.8}$$

Eine andere Herangehensweise zur Lösung von Mehrzieloptimierungsproblemen bilden die Pareto-Optimierungen. Hierbei werden die Bewertungskriterien nicht zu einem Wert zusammengefasst, sondern können weiterhin getrennt betrachtet und optimiert werden. Da sich die Bewertungskriterien häufig gegenseitig beeinflussen, kann es zu mehreren gleichwertigen Lösungen in völlig unterschiedlichen Bereichen kommen. Ein System mit einer sehr begrenzten Leistungsfähigkeit kann aufgrund seiner geringeren Leistungsaufnahme beispielsweise eine genau so hohe Energieeffizienz wie ein Hochleistungssystem aufweisen. Da durch diese Wechselwirkungen keine optimale Lösung existiert, wird eine Menge von Pareto-optimalen Konfigurationen ermittelt [60, 61].

Der Begriff des Pareto-Optimums stammt ursprünglich aus der Volkswirtschaftslehre und bezeichnet einen Zustand „in dem durch eine Maßnahme kein Wirtschaftssubjekt mehr besser gestellt werden kann, ohne dass ein anderes Wirtschaftssubjekt

[7] Energy-Delay-Product, EDP

schlechter gestellt wird“ [15]. Die Pareto-optimalen Punkte zeichnen sich folglich dadurch aus, dass sie jeweils eine Lösung bilden, bei der eine Verbesserung eines Ziels nur noch durch die Verschlechterung eines anderen erreicht werden kann [60].

Die Pareto-optimalen Konfigurationen bilden die Pareto-Front (siehe Abbildung 5.1). Diese Front beschreibt die Menge von Konfigurationen, die alle für ein bestimmtes Anwendungsfeld optimal sind. Konfigurationen, die nicht auf dieser Front liegen, werden von den Pareto-optimalen Konfigurationen dominiert und sind deshalb als ineffizient einzustufen. Der Entwickler kann abhängig von den Zielvorgaben die Gewichtung der verschiedenen Teilziele abwägen und eine Konfiguration aus der Pareto-Front wählen. In den Entwurfsraumexplorationen von Palesi [61] und Ascia [7] wird der Entwurfsraum beispielsweise bezüglich der Wechselwirkungen zwischen Leistungsaufnahme und Laufzeit untersucht. Es werden Pareto-Optimale Konfigurationen gefunden, die anschließend entweder durch Vorgaben an die Laufzeit oder durch eine maximale Leistungsaufnahme eingeschränkt werden.

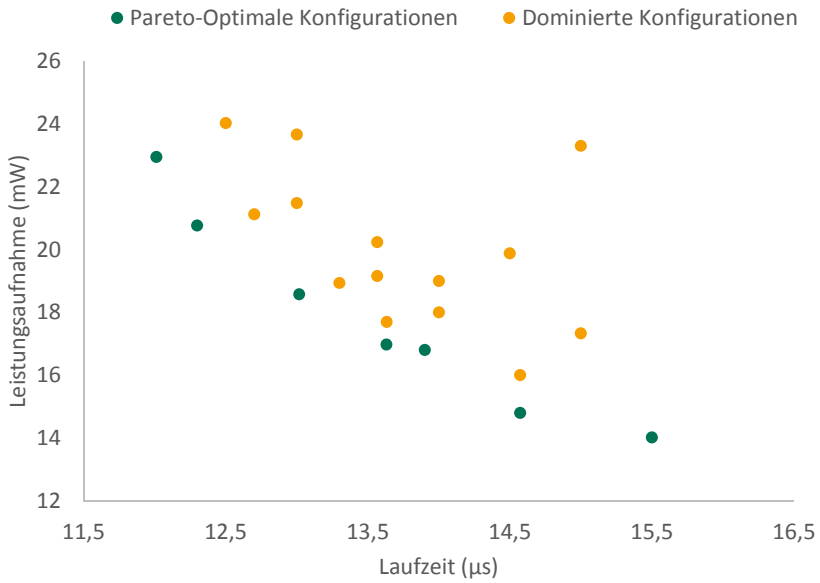

Abbildung 5.1: Fiktives Beispiel eines Pareto-Diagramms

Es finden häufig auch Kombinationen dieser zwei Herangehensweisen Verwendung, bei denen zuerst einige Bewertungskriterien zu einer Zielfunktion zusammengefasst werden, um anschließend eine Mehrzieloptimierung vorzunehmen. In der Arbeit von Palermo [60] wird beispielsweise eine Mehrzieloptimierung der Teilziele Energie und Laufzeit durchgeführt. Abhängig von Vorgaben an die Laufzeit wird anschließend eine Konfiguration mit einer möglichst hohen Energieeffizienz ausgewählt.

5.2.1 Mehrstufige Entwurfsraumexplorationen und effiziente Suchalgorithmen

Abhängig von der Größe des Entwurfsraums ist es selbst mit modellbasierten Verfahren nicht möglich, alle Prozessorkonfigurationen in einer angemessenen Zeit zu untersuchen und zu bewerten. Aus diesem Grund kommen bei der Exploration größerer Entwurfsräume häufig mehrstufige Entwurfsraumexplorationen zum Einsatz, bei denen die Konfigurationsmenge entweder sukzessive eingeschränkt wird oder mehrere Teilentwurfsräume aufgebaut werden [60, 61].

Das sukzessive Einschränken der Konfigurationsmenge wird durch ein wiederholtes Durchsuchen sämtlicher Prozessorkonfigurationen mit unterschiedlicher Genauigkeit erreicht. In der Arbeit von Mohanty [56] wird die Konfigurationsmenge in einem ersten Arbeitsschritt mit Hilfe schneller mathematischer Modelle eingeschränkt, indem frühzeitig geprüft wird, ob die jeweiligen Prozessorkonfigurationen die Vorgaben an die Leistungsfähigkeit einhalten können. In einem zweiten Arbeitsschritt kommen Abschätzungstechniken und Simulationen auf hoher Abstraktionsebene zum Einsatz, um die verbleibenden Prozessorkonfigurationen in Bezug auf die Laufzeit und Leistungsaufnahme zu überprüfen und die Anzahl der Konfigurationen nochmals zu verringern. Schließlich wird in einem dritten Schritt durch zeitintensive Simulationen auf einer niedrigeren Abstraktionsebene die beste Prozessorkonfiguration ausgewählt.

Bei dem Aufteilen des Entwurfsraums werden die Konfigurationselemente in sinnvolle Teilmengen zusammengefasst und getrennt ausgewertet. Hierbei sind die Elemente so zu gruppieren, dass keine Abhängigkeiten zwischen verschiedenen Teilmengen entstehen. Die Explorationen der einzelnen Teilmengen bilden lokale (Pareto-)optimale Lösungen, wie beispielsweise eine Lösung für den Prozessorkern und eine Lösung für das Speichersystem. Diese Lösungen können in einem abschließenden Schritt zu einer (Pareto-)optimalen Lösung für das Gesamtsystem zusammengefasst werden [61].

Da die Abhängigkeiten zwischen den Konfigurationselementen jedoch häufig sehr komplex sind und die resultierenden Teilmengen zu groß werden, zeigen die Arbeiten von Palermo [60] und Palesi [61] schließlich zwei vielversprechende Ansätze, um umfangreiche Entwurfsraumexplorationen mittels spezieller Suchalgorithmen zu beschleunigen. Palesi [61] nutzt zur Analyse des betrachteten Entwurfsraums[8] einen Algorithmus aus dem Bereich der Genetik. Dieser Algorithmus basiert auf der Evolution von Populationen, bei denen über mehrere Generationen hinweg jeweils nur die besten Individuen fortbestehen. Bezogen auf die Entwurfsraumexploration

[8] $5{,}97 \cdot 10^{12}$ unterschiedliche Prozessorkonfigurationen

werden hierbei die Generationswechsel durch mehrere Iterationsschritte abgebildet. Das Bewertungskriterium wird durch einen Fitness-Wert beschrieben, der aus der Laufzeit und der Leistungsaufnahme berechnet wird. Im Unterschied zu Verfahren die den Entwurfsraum vollständig durchsuchen, beginnt dieser Algorithmus mit einer zufällig gewählten Konfiguration und prüft nach jedem Iterationsschritt, ob das Ändern einzelner Konfigurationselemente den Fitness-Wert verbessert hat und somit diese Generation der vorangegangenen überlegen ist. Die Arbeit von Palermo [60] zeigt des Weiteren, dass das Finden der Pareto-optimalen Konfigurationen mit Hilfe heuristischer Algorithmen wie Random Search Pareto, Pareto Simulated Annealing und Pareto Reactive Tabu Search bei vergleichsweise geringer Ungenauigkeit um bis zu drei Größenordnungen beschleunigt werden kann.

5.3 Die Entwurfsraumexploration des CoreVA-Prozessors

Wie die nachfolgenden Überlegungen zeigen, ist es aufgrund der Vielzahl an unterschiedlichen Konfigurationselementen auch beim CoreVA-Prozessor praktisch unmöglich, alle Prozessorkonfigurationen in einem Schritt zu analysieren. Die Anzahl der unterschiedlichen Konfigurationsmöglichkeiten ist durch eine Zweierpotenz beschrieben. Soll beispielsweise untersucht werden, ob ein bestimmter VLIW-Slot durch eine MAC-Einheit erweitert werden kann, ergeben sich $2^1 = 2$ Prozessorkonfigurationen (M0 und M1). Soll stattdessen untersucht werden, ob neben einer potentiellen MAC-Einheit gegebenenfalls auch eine DIV-Einheit implementiert wird, ergeben sich $2^2 = 4$ Prozessorkonfigurationen (M0D0, M0D1, M1D0, M1D1). Die Anzahl der Konfigurationselemente, die Einflüsse auf einen einzelnen VLIW-Slot haben, bildet dementsprechend den Exponent der Zweierpotenz (siehe Formel 5.9). Die Variation der MAC-, DIV- und LD/ST-Funktionseinheiten führt bei einem CoreVA-Prozessor mit einer Verarbeitungseinheit demnach zu $2^3 = 8$ verschiedenen Konfigurationen.

$$Konfigurationen_{1Slot} = 2^{(Elemente\ pro\ Slot)} \tag{5.9}$$

Wird eine variable Anzahl von VLIW-Slots in die Untersuchung einbezogen, bei der für jeden Slot einzeln entschieden werden kann, ob er durch die oben genannten Funktionseinheiten erweitert wird, wird der Exponent der Zweierpotenz mit der maximalen Anzahl der VLIW-Slots multipliziert (siehe Formel 5.10). Bezogen auf das obige Beispiel steigt die Anzahl der möglichen Prozessorkonfigurationen in einem Prozessor mit bis zu vier Verarbeitungseinheiten auf $2^{(4 \cdot 3)} = 4096$.

$$Konfigurationen_{NSlot} = 2^{(Slots \cdot Elemente\ pro\ Slot)} \tag{5.10}$$

Übergeordnete Elemente, die die Funktionalität aller VLIW-Slots in gleicher Weise verändern, erhöhen den Exponenten der Zweierpotenz um einen Summanden und verdoppeln somit die Menge der Konfigurationen (siehe Formel 5.11). Beispiele hierfür sind das globale Aktivieren der SIMD-Berechnungen ($2^{(4\cdot3)+1}$= 8192) und das einmalige Verändern des Resource-Sharings ($2^{(4\cdot3)+2}$= 16384). Jede Konfigurationsänderung außerhalb des Prozessorkerns verdoppelt die Anzahl der möglichen Konfigurationen ebenfalls. Hierzu zählt beispielsweise die Entscheidung, ob der Speicher Cache-basiert oder als reiner on-chip Scratchpad-Speicher implementiert wird ($2^{(4\cdot3)+3}$= 32768), und ob der Speicher in einem Prozessor mit zwei LD/ST-Einheiten als Dual-Port oder als Multi-Bank Speicher ausgelegt werden soll ($2^{(4\cdot3)+4}$= 65536).

$$Konfigurationen = 2^{((Slots \cdot Elemente\ pro\ Slot) + übergeordnete\ Elemente)} \tag{5.11}$$

In der Praxis lässt sich diese mathematisch ermittelte Konfigurationsanzahl, die bei Weitem nicht vollständig ist sondern nur den exponentiellen Anstieg verdeutlichen soll, durch einige Benutzervorgaben wieder etwas eingrenzen. Es macht beispielsweise keinen Sinn, die erste Verarbeitungseinheit zu entfernen und nur die zweite, dritte und vierte Einheit zu betrachten. Außerdem muss mindestens eine MAC-, DIV- und LD/ST-Einheit implementiert werden, um alle Instruktionen des Instruktionssatzes unterstützen zu können. Des Weiteren erübrigen sich viele Konfigurationsmöglichkeiten durch den Einsatz des Ressource-Sharings.

Für eine vollständige Entwurfsraumexploration des CoreVA-Prozessors müssen jedoch trotz dieser Vorgaben mehrere tausend Konfigurationen synthetisiert und simuliert werden, da die Entwicklungsumgebung des CoreVA-Prozessors bisher nur eine experimentelle Ermittlung der benötigten Werte ermöglicht. Eine einzelne Synthese dauert hierbei etwa 20 Minuten, ein Simulationsdurchlauf auf dem Instruktionssatzsimulator eine Minute und eine Hardwaresimulation mit Aufzeichnung der Schaltaktivitäten ein bis drei Stunden. Aus diesem Grund wird im Rahmen dieser Arbeit eine mehrstufige Entwurfsraumexploration entwickelt, die sowohl die Aufteilung des Entwurfsraums nach Palesi [61], als auch das sukzessive Einschränken der Konfigurationsmenge nach Mohanty [56] nutzt.

Bei der Aufteilung des Entwurfsraums können die Konfigurationselemente, die keine oder nur geringe Abhängigkeiten untereinander aufweisen, getrennt betrachtet werden, indem die jeweils unberücksichtigten Elemente auf einen sinnvollen konstanten Wert gesetzt werden (siehe Kapitel 5.2.1). Im Falle des CoreVA-Prozessors

lässt sich hierdurch eine Entwurfsraumexploration über die Elemente des Prozessorkerns durchführen, ohne beispielsweise die Konfiguration der Speicher einbeziehen zu müssen. Das sukzessive Einschränken der Konfigurationsmenge erfolgt auf Basis einer modellbasierten Entwurfsraumexploration, mit deren Hilfe besonders ineffiziente Konfigurationen frühzeitig eliminiert werden können. Anschließend kommt in der verbleibenden Menge eine experimentelle Entwurfsraumexploration zum Einsatz, um mit Hilfe von konfigurationsspezifischen Simulationen und Synthesen eine endgültige Entscheidung treffen zu können.

5.4 Zusammenfassung

In diesem Kapitel wurden die Vor- und Nachteile verschiedener Verfahren zur Durchführung anwendungsspezifischer Entwurfsraumexplorationen beschrieben und das geplante Vorgehen für die Exploration des CoreVA-Prozessors dargelegt. Zu Beginn der Entwurfsraumexploration wird ein Entwurfsraum aufgespannt, der die variablen Konfigurationselemente des jeweiligen Prozessors beinhaltet. Im Falle des CoreVA-Prozessors besteht der Entwurfsraum aus der Anzahl der VLIW-Slots, der Anzahl der MAC-, DIV-, und LD/ST-Einheiten, der Verfügbarkeit von SIMD-Instruktionen, der Konfiguration des Resource-Sharings und der Größe und des Typs des Instruktions- und Datenspeichers (siehe Kapitel 3.17). Aus diesem Entwurfsraum wird eine Konfigurationsmenge abgeleitet, die anschließend bezüglich ihrer Leistungsfähigkeit und ihrer Ressourcenanforderungen bewertet wird.

Bei der Ermittlung dieser Bewertungskriterien werden zwei Herangehensweisen unterschieden, die experimentelle und die modellbasierte Exploration. Die experimentelle Entwurfsraumexploration bestimmt die Auswirkungen der unterschiedlichen Prozessorkonfigurationen durch eine Vielzahl konfigurationsspezifischer Synthesen und Simulationen. Hierdurch lassen sich sehr exakte Aussagen über die Leistungsaufnahme und Fläche eines Prozessors sowie über die Laufzeit der einzelnen Anwendungen treffen. Da die Untersuchungen jedoch für jede Anwendung und jede zu bewertende Prozessorkonfiguration durchgeführt werden müssen, sind experimentelle Explorationen äußerst zeitaufwändig. Modellbasierte Entwurfsraumexplorationen nutzen dagegen Verfahren, bei denen die Bewertungskriterien durch Software- und Hardwaremodelle beschrieben werden, die mit Hilfe einzelner Probesynthesen und Simulationen aufgestellt werden können. Diese Explorationen sind hierdurch zwar deutlich zeitsparender, sie können jedoch gegebenenfalls hohe Ungenauigkeiten aufweisen. Bei der Bewertung der Konfigurationen kommen ebenfalls verschiedene Ansätze in Betracht. So lassen sich beispielsweise die Energiewerte vergleichen, die zur Bearbeitung einer Anwendung auf den jeweiligen Prozessor-

konfigurationen benötigt werden. Alternativ können die Bewertungskriterien durch Pareto-Diagramme auch getrennt betrachtet und optimiert werden.

Da eine modellbasierte Entwurfsraumexploration mit hohen Fehlern behaftet sein kann und eine experimentelle Untersuchung sämtlicher Prozessorkonfigurationen aufgrund der vielfältigen Konfigurierbarkeit des CoreVA-Prozessors ebenfalls nicht in Frage kommt, wird in dieser Arbeit ein mehrstufiges Vorgehen verwendet. Hierbei wird die Konfigurationsmenge im Vorfeld durch modellbasierte Verfahren so weit eingeschränkt, dass im Anschluss experimentelle Entwurfsraumexplorationen zum Einsatz kommen können, um eine endgültige Entscheidung zu treffen. Da hierzu jedoch fundierte Modelle zur Beschreibung der Leistungsfähigkeit und der Ressourcenanforderungen benötigt werden, wird in den folgenden Kapiteln beschrieben, wie die Entwicklungsumgebung des CoreVA-Prozessors um verschiedene Software- und Hardwaremodelle erweitert wurde. Die Bewertung der jeweiligen Prozessorkonfigurationen erfolgt mit Hilfe der nun vorgestellten Beispielanwendungen.

6 Ausgewählte Beispielanwendungen

Dieses Kapitel beschreibt die Beispielanwendungen, die ausgewählt wurden, um ein Szenario für die anwendungsspezifischen Entwurfsraumexplorationen zu schaffen, dass dem späteren Einsatzgebiet des CoreVA-Prozessors möglichst nahe kommt. Hierbei wurde zum Einen berücksichtigt, dass der Prozessor sowohl als eigenständiger Prozessor in einem eingebetteten System, als auch als Teil eines Multiprozessorsystems einsetzbar sein soll. Zum Anderen wurde beachtet, dass der CoreVA-Prozessor nicht ausschließlich für die Verarbeitung einer einzelnen Anwendung ausgelegt werden soll, sondern einen großen Bereich der mobilen digitalen Signalverarbeitung abdecken muss. Ein mögliches Einsatzgebiet des CoreVA-Prozessors bildet folglich die Verarbeitung von Software-defined Radio Anwendungen (SDR), die es einem mobilen System ermöglichen, eine Vielzahl unterschiedlicher Übertragungsstandards zu unterstützen und somit eine weltweite Einsetzbarkeit zu garantieren. Zu den derzeit gebräuchlichen Übertragungsverfahren zählen beispielsweise Mobilfunkstandards wie UMTS[1], HSDPA[2] und LTE[3], Datenübertragungsverfahren wie Bluetooth und Wireless LAN oder digitale Rundfunkdienste wie DVB-T[4].

Im Unterschied zu herkömmlichen Hardwareplattformen, bei denen zahlreiche dedizierte Transceiverbausteine verwenden werden, um die verschiedenen Übertragungsverfahren abbilden zu können, wird die Signalverarbeitung in SDR-Systemen als flexible Softwareimplementierung realisiert. Da im alltäglichen Betrieb meist nur ein oder wenige Übertragungsverfahren zeitgleich genutzt werden und da die Softwareimplementierungen auf universellen Signalverarbeitungsprozessoren ausgeführt werden können, ergibt sich in SDR-Systemen eine deutlich bessere Auslastung der Hardware. Zusätzlich erlaubt die programmierbare Architektur das schnelle und somit kostengünstige Implementieren neuer Standards. Um die von den meisten Übertragungsverfahren geforderte Echtzeitfähigkeit zu gewährleisten, müssen die eingesetzten Prozessoren jedoch über eine vergleichsweise hohe Leistungsfähigkeit verfügen [3, 69].

[1] Universal Mobile Telecommunications System
[2] High Speed Downlink Packet Access
[3] Long Term Evolution
[4] Digital Video Broadcasting - Terrestrial

6.1 Der OFDM-Empfänger als Beispiel einer Software-defined Radio Anwendung

Das folgende Beispiel betrachtet die Basisbandsignalverarbeitung eines OFDM-Empfängers[5]. OFDM-Systeme sind dadurch charakterisiert, dass sie mit Hilfe von Multicarrier-Modulationen das zeitgleiche Übertragen mehrerer Datenströme unterstützen. Hierzu werden im Sender die einzelnen binären Datenströme mit Modulationsverfahren wie beispielsweise BPSK[6], QPSK[7] oder QAM[8] in den komplexen Symbolraum abgebildet. Anschließend werden diese Symbole durch inverse Fourier-Transformationen (IFFT) auf orthogonal zueinander stehende Trägersignale moduliert und mittels Addition zu einem Signal zusammengefasst [3, 19]. Aufgrund der gemeinsam genutzten Modulationsverfahren kann der im Folgenden beschriebene Teil des Empfängers sowohl für das Verarbeiten von LTE, DVB-T oder Wireless LAN Signalen genutzt werden [19, 59].

Wie bei SDR-Systemen üblich, besteht der OFDM-Empfänger aus mehreren einzelnen Blöcken, die über einen Datenpfad miteinander verbunden sind. Der Aufbau eines Empfängers ist jedoch nicht fest spezifiziert, sondern orientiert sich lediglich an den jeweiligen 3GPP[9]-Spezifikationen der zugehörigen Sender. Aus diesem Grund gibt es keine einheitlichen Darstellungen eines OFDM-Empfängers. Es finden sich jedoch zahlreiche Veröffentlichungen, wie beispielsweise die Arbeiten von Airoldi [3], Olsson [59], Berg [9] und May [53], in denen OFDM-Empfänger mit vergleichbarem Aufbau beschrieben werden. Exemplarisch wird im Folgenden der Aufbau eines LTE OFDM-Empfängers nach Olsson [59] erläutert (siehe Abbildung 6.1).

Bevor die empfangenen Signale mit Hilfe eines rein digitalen SDR-Systems verarbeitet werden können, müssen sie von einer analogen Empfangsschaltung aufbereitet und digitalisiert werden. Die analoge Empfangsschaltung besteht hierzu typischerweise aus einer Antenne mit mehreren analogen Filtern, Mischern und Verstärkern, sowie aus einem nachgeschalteten Analog-Digital-Wandler.

Der digitale Teil der Basisbandsignalverarbeitung gliedert sich in einen inneren Empfänger, der für die eigentliche Datenübertragung verantwortlich ist, und einen Dekodierer. Der innere Empfänger führt zuerst eine Frequenzkompensation durch, um eventuell aufgetretene Phasenverschiebungen zu erkennen und zu korrigieren. Hierbei kommen entweder Cordic-Algorithmen zum Einsatz oder die in SDR-Systemen

[5] Orthogonal Frequency Division Multiplexing, Orthogonales Frequenzmultiplexverfahren
[6] Binary Phase Shift Keying, Binäre Phasenmodulation
[7] Quadrature Phase Shift Keying, Quadraturphasenumtastung
[8] Quadrature Amplitude Modulation, Quadraturamplitudenmodulation
[9] 3rd Generation Partnership Project

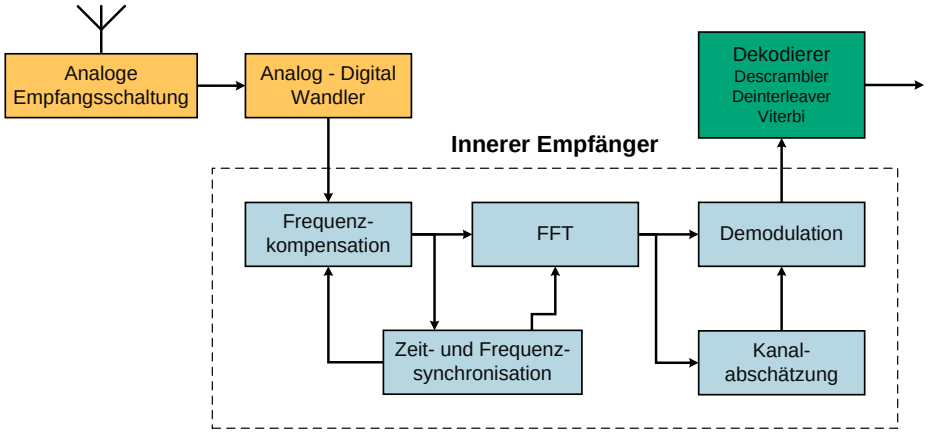

Abbildung 6.1: Die Komponenten eines OFDM-Empfängers nach [59]

verbreiteteren Kreuzkorrelationsverfahren [3, 9, 59]. Nach der Frequenzkompensation erfolgt eine Zeit- und Frequenzsynchronisation, mit deren Hilfe die Anfänge neuer OFDM-Symbole detektiert werden. Hierzu werden entweder speziell übertragene Synchronisationssignale analysiert[10] oder die in dem Datenstrom enthaltenen Cyclic-Prefix Daten ausgewertet. Die Synchronisation erfolgt in beiden Fällen ebenfalls auf Basis von Auto- oder Kreuzkorrelationsfunktionen [3, 59].

Das durch die Frequenzkompensation und die Synchronisationen bereinigte Signal wird an die diskrete Fourier-Transformation weitergeleitet, die die OFDM-Symbole mit speziellen Fast-Fourier-Transformationsalgorithmen (FFT) vom Zeitraum in den Frequenzraum überführt [59]. Anschließend erfolgt die Kanalabschätzung und das Demodulieren der Symbole. Die Kanalabschätzung ermittelt, wie das Ausbreitungsmedium (z.B. die Luft oder das Kabel) das Signal durch Rauschen oder punktuelle Störungen beeinflusst hat, indem spezielle Pilot-Signale aus den Nutzsignalen extrahiert und ausgewertet werden [3]. Hierbei kommen Verfahren wie die Methode der kleinsten Quadrate[11] oder Verfahren zur Minimierung des mittleren quadratischen Fehlers[12] zum Einsatz. Diese Verfahren basieren in SDR-Systemen häufig auf diskreten FIR-Filtern[13] [19, 72]. Bei der Demodulierung werden die im Sender angewandten Amplituden- oder Phasenmodulationsverfahren rückgängig gemacht und die binären Datenströme wiederhergestellt [3].

Zuletzt erfolgt das Dekodieren der physikalischen Kanäle und das Weiterleiten der darin enthaltenen Transportkanäle an die höher gelegenen Verarbeitungsschichten. Der Dekodierer vereint hierzu eine Vielzahl kleinerer Funktionsblöcke, die einen

[10] In LTE Systemen beispielsweise das primäre und sekundäre Synchronisationssignal PSS und SSS
[11] Method of Least Squares
[12] Minimum Mean Sqare Error Estimation
[13] Finite Impulse Response Filter, Filter mit endlicher Impulsantwort

korrekten Datenempfang gewährleisten. Der Descrambler revidiert beispielsweise die bitweise „Verwürfelung“ des Datenstroms, die der Sender durchführen musste, um eine Gleichanteilfreiheit im Sendesignal gewährleisten zu können. Diese Gleichanteilfreiheit wird benötigt, da der Empfänger die Signalflanken bei langen 0- beziehungsweise 1-Folgen nicht gesichert extrahieren kann, woraufhin die Taktsynchronisation mit dem Sender fehlschlägt. Der Viterbi-Dekoder ist schließlich in der Lage, einzelne Bitfehler zu korrigieren, da der Sender die Nutzdaten durch eine Faltung auf mehrere Stellen des Datenstroms verteilt hat und die ursprüngliche Sendesequenz aus den hierdurch entstehenden Redundanzen rekonstruiert werden kann. Um sogar mehrere aufeinander folgende Bitfehler korrigieren zu können, werden die durch die Faltung erzeugten Codewörter im Sender von einem Interleaver zusätzlich ineinander verschachtelt. Diese Verschachtelung muss der Empfänger vor dem Dekodieren mit Hilfe eines Deinterleavers ebenfalls wieder rückgängig machen [2, 9, 53].

Die hier genannten Veröffentlichungen haben gezeigt, dass auf die Kernalgorithmen des inneren Empfängers und insbesondere auf die hierin enthaltene Fourier-Transformation mit Abstand die meiste Rechenlast entfällt und dass die Verarbeitungsgeschwindigkeit dieser Algorithmen die Leistungsfähigkeit des ganzen SDR-Systems bestimmt [70]. Des Weiteren konnte festgestellt werden, dass diese Algorithmen in einer Vielzahl weiterer Signalverarbeitungssysteme Verwendung finden, wie beispielsweise in der Mustererkennung oder in der Sprach- und Bildverarbeitung. Da der Schwerpunkt dieser Arbeit auf der Entwurfsraumexploration einzelner Prozessoren liegt, soll in den folgenden Untersuchungen daher kein vollständiges SDR-System zu Grunde gelegt werden, sondern nur die in Tabelle 6.1 genannten Kernalgorithmen betrachtet werden. Diese Kernalgorithmen wurden aus verschiedenen SDR-Systemen extrahiert und können durch eigene Testprogramme separat betrieben werden. Bei der Auswahl der Algorithmen wurden bewusst Programme gewählt, die unterschiedlich gut an die Architektur des CoreVA-Prozessors angepasst sind und verschiedene Parallelitätsgrade aufweisen. Neben den Algorithmen der Basisbandsignalverarbeitung werden jedoch auch typische Anwendungen aus höheren Verarbeitungsschichten und einige Prozessorbenchmarks einbezogen, die im folgenden Abschnitt erläutert werden.

6.2 Beispielanwendungen höherer Verarbeitungsschichten

Neben der eigentlichen Datenübertragung entfällt ein großer Teil der Aufgaben eines SDR-Systems auf die Verschlüsselung und Kompression der Nutzdaten und auf die Erkennung von Übertragungsfehlern. Bei der Verschlüsselung kommen spezielle Kryptographiealgorithmen zum Einsatz, wie beispielsweise das symmetrische Verschlüsselungsverfahren AES[14]. Diese Verfahren ermöglichen es dem Sender, den Datenstrom mit Hilfe eines vorher vereinbarten Schlüsselworts so zu verschlüsseln, dass nur ein autorisierter Empfänger die enthaltenen Daten dekodieren kann [18, 26].

Um jedoch nicht nur sicherstellen zu können, dass die Nutzdaten vor dem Auslesen Dritter geschützt sind, sondern auch eine fehlerfreie Übertragung zu garantieren, müssen neben der Verschlüsselung diverse Prüfsummenverfahren ausgeführt werden. Ein Beispiel hierfür bildet die zyklische Redundanzprüfung CRC [65]. Bei diesem Verfahren wird aus den zu übertragenden Nutzdaten eine Prüfsumme berechnet, die anschließend als zusätzliche Redundanz an den Datenstrom anfügt wird. Mit Hilfe dieser Prüfsumme kann der Empfänger eine fehlerhafte Datenübertragung detektieren und das Datenpaket gegebenenfalls erneut anfordern. Da die meisten Prüfsummenverfahren jedoch reversibel sind und somit keinen Schutz vor bewussten Manipulationen bieten, sollten sie stets in Kombination mit Verschlüsselungsverfahren eingesetzt werden. Als Alternative hierzu können kryptographische Hash-Funktionen wie der SHA-3-Algorithmus[15] eingesetzt werden. Bei diesen Funktionen (auch Streuwertfunktionen genannt) werden ebenfalls Prüfsummen erzeugt, um Übertragungsfehler erkennen zu können. Da bei der Erstellung der Prüfsummen jedoch komplexe Verschlüsselungsalgorithmen zum Einsatz kommen, kann die geforderte Manipulationssicherheit auch ohne das Verschlüsseln des kompletten Datenstroms erreicht werden [26].

Als Beispielanwendung für eine Datenkompression in den höheren Verarbeitungsschichten wird in den folgenden Analysen der SATD[16]-Algorithmus betrachtet. Dieser Algorithmus ist Teil der Videokompressionen des ITU[17]-Standards H.264 und des ISO/IEC[18]-Standards MPEG-4/AVC[19]. Der Grundgedanke dieser Datenkompression ist, dass bei bewegten Bildern nicht die Änderungen einzelner Pixel übertragen

[14] Advanced Encryption Standard
[15] Secure Hash Algorithm
[16] Sum of Absolute Transformed Differences
[17] International Telecommunication Union
[18] International Organization for Standardization, International Electrotechnical Commission
[19] Moving Picture Experts Group, Advanced Video Coding

werden, sondern die Verschiebungen ganzer Pixelblöcke. Um diese Verschiebungen jedoch gesichert detektieren zu können, müssen die Pixel des ursprünglichen Bildausschnitts mit den Pixeln des potentiell neuen Standorts verglichen werden. Hierbei kommt der SATD-Algorithmus zum Einsatz, der die Gleichheit zweier Pixelblöcke bewertet, indem er die Differenzen zwischen den Bildbereichen durch eine Hadamard-Transformation in den Frequenzbereich überträgt und anschließend aufaddiert [71].

Mit den Anwendungsprogrammen Dhrystone und Coremark werden schließlich zwei industriell anerkannte Benchmarkprogramme in die Untersuchungen aufgenommen, die einem herstellerunabhängigen Vergleich verschiedener Prozessorsysteme dienen. Der Dhrystone-Benchmark gehört zur Klasse der synthetischen Benchmarks. Er besteht aus einer Vielzahl arithmetisch-logischer Rechenoperationen und Speicherzugriffen, die bewusst gewählt wurden, um die Leistungsfähigkeit von Prozessoren zu messen. Im Unterschied hierzu basiert die Bewertung des Coremarks auf den Algorithmen tatsächlicher Anwendungsszenarien. So führt er Matrizenmultiplikationen, Steuerungsprogramme von Zustandsautomaten, Listenverarbeitungen und schließlich zyklische Redundanzprüfungen durch. Da beide Benchmarks auf den Vergleich eingebetteter Systeme abzielen, wird auf eine Verarbeitung von Fließkommaoperationen verzichtet [37, 82].

Zur Ermittlung der Leistungsfähigkeit werden die Programmcodes der Benchmarks auf den jeweiligen Zielsystemen ausgeführt. Das Testergebnis wird schließlich aus der Laufzeit gebildet, die die Prozessoren zur Verarbeitung eines Testdurchlaufs (Iteration) benötigen. Die Coremark-Ergebnisse werden hierbei in Iterationen pro Sekunde angegeben. Die Dhrystone-Resultate werden ebenfalls in Relation zur Zeit angegeben. In der Praxis wird dieser Wert jedoch durch eine zusätzliche Division auf die Referenzmaschine VAX 11/780 normiert, die mit 1757 Iterationen pro Sekunde als eine 1 MIPS[20]-Maschine angesehen wird. Das normierte Ergebnis des Dhrystone-Benchmarks trägt dementsprechend den Namen Dhrystone-MIPS (DMIPS) (siehe Formel 6.1) [37, 82].

$$DMIPS = \frac{\frac{Iterationen}{Sekunde}}{1757} \tag{6.1}$$

[20] Millionen Instruktionen pro Sekunde

Anwendungsname	Algorithmus	Zuordnung
AES	Symmetrische Datenverschlüsselung	Kryptographie
Coremark	EEMBC Coremark	Prozessorbenchmark
CRC	Zyklische Redundanzprüfung	Prüfsummenberechnung
Deinterleaver	Deinterleaver eines LTE-Empfängers	Basisbandsignalverarbeitung
Descrambler	Descrambler eines LTE-Empfängers	Basisbandsignalverarbeitung
Dhrystone	Dhrystone 2.1	Prozessorbenchmark
FFT	Fast-Fourier-Transformation	Basisbandsignalverarbeitung
FIR	FIR-Filter	Basisbandsignalverarbeitung
Korrelation	Kreuzkorrelation	Basisbandsignalverarbeitung
SATD	Bewertung der Unterschiedlichkeit zweier Bilder	Videokompression
SHA-3	Kryptographische Hash-Funktion	Kryptographie
Viterbi	Korrektur einzelner Bitfehler	Basisbandsignalverarbeitung

Tabelle 6.1: Beispielanwendungen für die Entwurfsraumexploration des CoreVA-Prozessors

6.3 Komplexität der Algorithmen

Eine erste Methode, die zur Charakterisierung der Beispielanwendungen herangezogen werden kann, bildet die Analyse ihrer Komplexität. Die Komplexität einer Anwendung beschreibt ihren maximalen Ressourcenbedarf in Abhängigkeit von der Länge der Eingangsdaten (n). Zu den Ressourcen zählen hierbei zum Einen die Anzahl der Rechenschritte (Zeitkomplexität) und zum Anderen die Größe des Speichers, den der Algorithmus während seiner Verarbeitung belegt (Platzkomplexität) [42, 66].

Im Unterschied zu den in Kapitel 7 behandelten Analyseverfahren liegt der Fokus bei der Bestimmung der Komplexität jedoch nicht auf der Ermittlung des tatsächlichen Ressourcenbedarfs, sondern lediglich auf der Bewertung des relativen Zuwachses der Laufzeit und des Speicherbedarfs. Aus diesem Grund wird die Komplexität der Anwendungen in der Regel in einer vergleichsweise abstrakten O-Notation angegeben, die das Wachstum mit Hilfe einfacher mathematischer Funktionen wie beispielsweise O(1), O(log n), O(n), O(n^2) beschreibt. Konstante Faktoren, die Implementierungsdetails wie die Programmiersprache, die Qualität des Compilers oder die Hardwareeigenschaften des Prozessors widerspiegeln, werden hierbei vernachlässigt. Durch die hierarchische Anordnung der verschiedenen Notationsfunktionen bilden sich Komplexitätsklassen, in die sich die zu untersuchenden Anwendungen einordnen lassen [42, 66].

Ein Großteil der Beispielanwendungen (AES, CRC, Deinterleaver, Descrambler, SHA-3) fügt sich in eine Klasse mit linearer Zeit- und Platzkomplexität ein (O(n)). Dieses Verhalten rührt daher, dass die Algorithmen ihre Eingangsdaten in Abschnitte mit

einer fixen Blockgröße unterteilen und diese dann getrennt verarbeiten. Bei einer Verdopplung der Eingangsdaten muss dementsprechend ein doppelter Speicherplatz und eine doppelte Laufzeit veranschlagt werden [2, 48]. Die Zeitkomplexität des Coremark- und Dhrystone-Benchmarks weist ebenfalls ein lineares Verhalten auf, da diese Programme auf geschlossenen Tests basieren, die mehrmals ausgeführt werden. Da hierbei jedoch stets die gleichen Eingangsdaten verwendet werden und die Ergebnisse der vorangegangenen Testdurchläufe verworfen werden, zeigt die Platzkomplexität einen konstanten Zuwachs (O(1)).

Algorithmen die auf Matrizenmultiplikationen basieren, besitzen meistens eine quadratische oder kubische Komplexität. So hat die Kreuzkorrelation, die im Wesentlichen aus einer n x n-Matrizenmultiplikation besteht, eine Zeitkomplexität zwischen $O(n^2)$ und $O(n^3)$ und eine Platzkomplexität von $O(n^2)$ [62, 66]. Unter Ausnutzung von Symmetrien lassen sich die Komplexitäten einiger Anwendungen so weit reduzieren, dass beispielsweise der FFT- und der SATD-Algorithmus eine Zeit- und Platzkomplexität von O(n log n) und $O(2^n\, n)$ erreichen [67, 80]. Die Komplexitäten des FIR-Algorithmus können sogar auf O(n) und O(n) gesenkt werden, da hier der variable Eingangsvektor mit einer konstanten Anzahl an Koeffizienten multipliziert wird (siehe Formel 7.1 und Programmcode 7.1) [47].

Die Komplexität des Viterbi-Algorithmus wird schließlich durch zwei unabhängige Variablen charakterisiert. Die Variable n kennzeichnet hierbei wiederum die Länge der Eingangsdaten, die Variable s beschreibt die Menge der in diesen Eingangsdaten verborgenen Zustände. Es ergibt sich eine Zeit- und Platzkomplexität von $O(n\, s^2)$ und $O(n^2\, s^2)$. Unter der Annahme, dass die Menge der verborgenen Zustände konstant ist, verringern sich die Komplexitäten auf O(n) und $O(n^2)$ [51].

Anwendungsname	Zeitkomplexität	Platzkomplexität
AES	O(n)	O(n)
Coremark	O(n)	O(1)
CRC	O(n)	O(n)
Deinterleaver	O(n)	O(n)
Descrambler	O(n)	O(n)
Dhrystone	O(n)	O(1)
FFT	O(n log n)	$O(2^n\, n)$
FIR	O(n)	O(n)
Korrelation	$O(n^2)$	$O(n^2)$
SATD	O(n log n)	$O(2^n\, n)$
SHA-3	O(n)	O(n)
Viterbi	$O(n\, s^2)$	$O(n^2\, s^2)$

Tabelle 6.2: Komplexität der betrachteten Algorithmen

6.4 Zusammenfassung

In diesem Kapitel wurden die Zielanwendungen beschrieben, die als Basis für die Entwurfsraumexploration des CoreVA-Prozessors dienen. Die Anwendungen wurden so gewählt, dass sie einen großen Bereich der mobilen digitalen Signalverarbeitung abdecken. Hierbei wurde der Fokus zum Einen auf die Software-defined Radio Implementierung eines OFDM-Empfängers gelegt, da dieser eine Vielzahl unterschiedlicher Übertragungsstandards unterstützt und alle relevanten Algorithmen der Basisbandsignalverarbeitung beinhaltet. Zum Anderen wurden typische Anwendungen aus höheren Verarbeitungsschichten sowie einige Prozessorbenchmarks einbezogen (siehe Tabelle 6.1).

Die Analyse der Beispielanwendungen und die Recherche ihrer Zeit- und Platzkomplexität haben gezeigt, dass die gewählten Anwendungen sämtliche Komplexitätsklassen abdecken (siehe Tabelle 6.2). Es werden sowohl rechenintensive Algorithmen wie Fouriertransformationen und Matrizenmultiplikationen untersucht, als auch Algorithmen, die häufige Speicherzugriffe initiieren. Zur maximalen Parallelität der Anwendungen werden in der Literatur jedoch keine Aussagen getroffen. Aus diesem Grund wurde bei der Auswahl der Algorithmen bewusst darauf geachtet, das Programmversionen aufgenommen werden, die unterschiedlich gut an die Architektur des CoreVA-Prozessors angepasst sind.

7 Leistungsfähigkeit verschiedener Prozessorkonfigurationen

In diesem Kapitel werden verschiedene Analyseverfahren vorgestellt, um die Leistungsfähigkeit der jeweiligen Prozessorkonfigurationen zu ermitteln. Da die Leistungsfähigkeit durch die Laufzeit der Anwendungen und durch die Anzahl der benötigten Prozessortakte definiert ist (siehe Formel 5.1), ist sie sowohl von der Anzahl der verfügbaren Verarbeitungs- und Funktionseinheiten, als auch von der tatsächlichen Auslastung dieser Einheiten abhängig. Aus diesem Grund können neben den Prozessortakten auch abstraktere Anwendungscharakteristika betrachtet werden, um auf die Leistungsfähigkeit zu schließen. So lässt sich beispielsweise aus der durchschnittlichen Parallelität und aus den Anteilen der MAC- und LD/ST-Instruktionen abschätzen, in welchem Umfang die Anwendungen von zusätzlichen Verarbeitungs- und Funktionseinheiten profitieren können und inwiefern sich hierdurch ihre Laufzeiten reduzieren lassen.

Die Analyseverfahren, die zur Charakterisierung der Anwendungen eingesetzt werden, unterteilen sich in die Gruppen der statischen und dynamischen Verfahren. Statische Analyseverfahren, wie sie beispielsweise von Jordans [40] angewendet werden, basieren ausschließlich auf der Auswertung des Programmcodes der jeweiligen Zielanwendung und nutzen beispielsweise die Ergebnisse eines Compiler-Schedulers, um Aussagen über die Parallelität eines Programms treffen zu können. Dynamische Verfahren basieren hingegen auf einer simulationsbasierten Analyse des Laufzeitverhaltens. Im Unterschied zu den statischen Verfahren können dynamische Verfahren mit Hilfe von Instruktionssatzsimulatoren oder Hardwaresimulationen die tatsächliche Laufzeit der Anwendungen bestimmen (siehe Kapitel 5.1).

Die Werkzeugkette des CoreVA-Prozessors hat zur Ermittlung der Prozessortakte bisher lediglich einen experimentellen Ansatz unterstützt, bei dem die Anwendungsverarbeitung für sämtliche Prozessorkonfigurationen simuliert werden musste. Da dieses Vorgehen bei einer großen Anzahl unterschiedlicher Prozessorkonfigurationen jedoch sehr zeit- und rechenintensiv ist, wurden im Rahmen dieser Arbeit zwei modellbasierte Ansätze erarbeitet, die die Leistungsfähigkeit der Prozessorkonfigurationen ohne konfigurationsspezifische Simulationen ermitteln. Zum Einen wird in Kapitel 7.3 ein approximationsbasiertes dynamisches Analyseverfahren vorgestellt,

das die Prozessortakte und die Auslastung der Verarbeitungseinheiten auf Basis einer einzelnen konfigurationsunabhängigen Simulation bestimmt. Zum Anderen kommt in Kapitel 7.4 eine statische Programmcodeanalyse zum Einsatz, die die durchschnittliche Parallelität und die Anteile der verschiedenen Instruktionstypen mit Hilfe des LLVM-Compilers bestimmt. In einem ersten Schritt wird jedoch nochmals eine experimentelle Anwendungsanalyse durchgeführt, um Referenzwerte für die modellbasierten Verfahren zu erhalten und um die Arbeitsweise des Compilers zu verdeutlichen.

Die Gegenüberstellung der verschiedenen Analyseverfahren erfolgt am Beispiel des diskreten FIR-Filters. Wie in Abschnitt 6 gezeigt wurde, ist dieser Filter ein typischer Bestandteil der digitalen Signalverarbeitung. Er wird durch die Faltung der einzelnen Signalwerte mit einer definierten Anzahl von Koeffizienten realisiert und besteht folglich aus einer Vielzahl zusammenhängender Multiplikationen (siehe Formel 7.1) [72].

$$Ergebnis\ [i] = \sum_{k=0}^{m} Koeffizient\ [k] \cdot Signalwert\ [k+i] \tag{7.1}$$

Der Codeausschnitt 7.1 zeigt die Implementierung des FIR-Filters in der Programmiersprache C. Ein Durchlauf dieses Algorithmus verarbeitet demnach 80 diskrete Signalwerte und 17 Koeffizienten, um hieraus 64 Ergebniswerte zu erzeugen. Während der Verarbeitung eines FIR-Durchlaufs müssen 1088 Multiply-Accumulate Berechnungen durchgeführt werden. Unter der Annahme, dass die Koeffizienten jeweils nur einmal geladen werden, müssen für diese MAC-Operationen insgesamt 2193 Operanden geladen und 1088 Zwischenergebnisse gespeichert werden. Die Summe der LD/ST-Instruktionen einer FIR-Iteration würde dementsprechend 3281 betragen. Weitgehend unbekannt ist jedoch die Anzahl der zusätzlichen ALU- und LD/ST-Instruktionen, die beispielsweise zur Abbildung der Schleifensteuerung benötigt werden. Des Weiteren können mit dieser manuellen Programmcodeanalyse keine fundierten Aussagen darüber getroffen werden, wie gut der Compiler das hier vorgeschlagene Vorgehen tatsächlich umsetzten kann. Zur Parallelität des Programmcodes kann lediglich ausgesagt werden, dass es dem Compiler theoretisch möglich sein sollte, die Kernberechnung in Zeile 6 zu parallelisieren, da die Berechnungen der inneren Schleife keine Abhängigkeiten zu Zwischenergebnisse des gleichen Schleifendurchlaufs aufweisen.

Anhand dieses Programmcodes lässt sich nochmals gut die in Kapitel 6.3 beschriebene Komplexität verdeutlichen. Obwohl der FIR-Filter auf einer Matrizenmultiplikation basiert, die aufgrund verschachtelter Schleifkonstrukte normalerweise quadratische oder kubische Komplexitäten aufweist, besitzt dieser Algorithmus

eine lineare Zeit- und Platzkomplexität (O(n)). Dieses Verhalten rührt daher, dass der variable Eingangsvektor bei einem FIR-Filter mit einer konstanten Anzahl an Koeffizienten multipliziert wird. Die äußere Schleife wird somit stets siebzehnmal durchlaufen. Da die innere Schleife ausschließlich von der Anzahl der Eingangswerte abhängt, wird sich die Laufzeit und der Speicherbedarf für die Eingangs- und Ausgangssignale bei einer Verdopplung der Eingangsdaten ebenfalls verdoppeln.

```
1 short signal[80];
2 short coeff[17];
3 short result[64];
4 for (k=0; k<17; k++) {
5   for (i=0; i<64; i++) {
6     result[i] += coeff[k] * signal[k+i];
7   }
8 }
```

Programmcode 7.1: C-Programmcode des FIR-Algorithmus

7.1 Bewertung von Wartezyklen und NOP-Instruktionen

Wie bereits in den jeweiligen Grundlagenkapiteln des CoreVA-Prozessors angedeutet wurde (siehe Kapitel 3.5, 3.9, 3.12 und 3.14), kann die kontinuierliche Ausführung eines Programmcodes durch Wartezyklen und NOP-Instruktionen unterbrochen werden. Da der Instruktionssatzsimulator das Auswerten dieser Unterbrechungen in der bisherigen Version nicht unterstützt hat, musste er im Rahmen dieser Arbeit um einige Funktionalitäten erweitert werden.

Wartezyklen werden von der Prozessorhardware während der Programmausführung in den Programmablauf eingefügt, um das Ausführen von Instruktionen zu unterbinden, für die die benötigten Eingangsdaten noch nicht vorliegen. Dies tritt beispielsweise auf, falls ein Datenkonflikt zwischen zwei aufeinander folgenden Instruktionen nicht über einen Pipelinebypass aufgelöst werden kann, oder falls die Latenz der Ladeinstruktionen nicht berücksichtigt wurde. Durch das Einfügen von Wartezyklen werden die Instruction-Fetch- und die Instruction-Decode-Stufe so lange angehalten, bis die Datenkonflikte aufgehoben werden konnten. Die hinteren Pipelinestufen laufen währenddessen weiter, um die darin enthaltenen Berechnungen fertig stellen zu können [77]. Das Detektieren von Datenkonflikten und das Einfügen von Wartezyklen konnte im Instruktionssatzsimulator mit Hilfe einer virtuellen Pipeline nachgebildet werden, die die Zielregister der vorausgegangenen

Instruktionen speichert und Datenabhängigkeiten zwischen aufeinander folgenden Instruktionen erkennt.

Im Unterschied zu den Wartezyklen werden die vorderen Pipelinestufen bei der Verarbeitung von NOP-Instruktionen nicht angehalten. Da diese Instruktionen beim Durchlaufen der Prozessorpipeline jedoch keine Berechnungen ausführen und keine Register- oder Speicherschreibzugriffe initiieren, können sie ebenfalls zur Unterbrechung des Programmablaufs genutzt werden, falls sie als einzige Instruktion in einer Instruktionsgruppe stehen. In den meisten Prozessorsystemen werden NOP-Instruktionen bereits vom Compiler in den Programmcode eingefügt, um Datenkonflikte aufzulösen oder Instruktionen in einen speziellen VLIW-Slot zu verschieben [74]. Da die Größe des benötigten Instruktionsspeichers durch das Einfügen von NOP-Instruktionen jedoch unverhältnismäßig stark ansteigt, ist die Funktionalität des CoreVA-Prozessors so ausgelegt, dass die meisten Konflikte durch Wartezyklen oder durch das Ressource Sharing unterbunden werden. Der CoreVA-Compiler verwendet NOP-Instruktionen derzeit ausschließlich zur Reservierung des Verzögerungsschlitzes nach Sprungbefehlen. Diese Instruktionen konnten bereits von der ursprünglichen Version des Instruktionssatzsimulators ausgewertet werden.

In manchen Fällen können NOP-Instruktionen jedoch auch zur Laufzeit in den Programmablauf eingefügt werden. Dies geschieht, falls das Alignment-Register nach einem bedingten Sprung mit falscher Sprungvorhersage noch nicht vollständig wiederhergestellt werden konnte oder falls sich die Instruktionsgruppe des Sprungziels über zwei Instruktionsblöcke erstreckt. Da diese nachträglich eingefügten NOP-Instruktionen bisher nicht berücksichtigt werden konnten, wurde das Überprüfen, ob sich eine Instruktionsgruppe über zwei Blöcke erstreckt, durch einen vorausschauenden Zugriff auf den Instruktionsspeicher realisiert. Falsche Sprungvorhersagen werden hingegen erst rückwirkend detektiert, da der Instruktionssatzsimulator keine Sprungvorhersagen durchführen muss, um bedingte Sprünge korrekt ausführen zu können. In Anlehnung an die Vorgaben aus Tabelle 3.3 in Kapitel 3.12 wird im Instruktionssatzsimulator von einer falsch getroffenen Sprungvorhersage ausgegangen, falls Register-Sprünge stattfinden oder falls Offset-Sprünge mit positivem Offset auftreten.

Da der CoreVA-Prozessor NOP-Instruktionen derzeit ausschließlich zur Unterbrechung der Programmverarbeitung verwendet, werden sie im Folgenden den Wartezyklen zugeordnet. Bei der Ermittlung der Instruktionsanzahl werden sie folglich nicht mitgezählt.

7.2 Experimentelle Anwendungsanalyse

Die experimentelle Anwendungsanalyse ist dadurch gekennzeichnet, dass für jede zu untersuchende Prozessorkonfiguration und jede Anwendung ein Simulationsdurchlauf mit anschließender Auswertung erfolgen muss. Aufgrund des konfigurationsspezifischen Schedulings des LLVM-Compilers müssen die Zielanwendungen hierbei im Vorfeld für jede Konfiguration erneut kompiliert werden [73, 83].

Diese Form der simulationsbasierten Anwendungsanalyse kann auf verschiedenen Ebenen der Entwicklungsumgebung erfolgen. Sowohl der Instruktionssatzsimulator des CoreVA-Prozessors, als auch die Hardwaresimulationen auf Register-Transfer- und Gatternetzlistenebene ermöglichen eine zyklenakkurate Simulation des Laufzeitverhaltens. Da die Simulationen hierbei sowohl statische als auch datenabhängige Schleifendurchläufe berücksichtigen, sind sie in der Lage, die tatsächliche Anzahl der Instruktionen und Prozessortakte zu bestimmen. Weil sie die ermittelten Werte jedoch nicht nach Funktionen unterteilen können, kommen spezielle Makros (defines) zum Einsatz, die den auszuwertenden Programmausschnitt markieren. Hierdurch wird vermieden, dass Initialisierungsphasen und das Einlesen und Zurückschreiben der Nutzdaten einen Einfluss auf die Auswertung der Kernalgorithmen nehmen.

Die in Tabelle 7.1 gelisteten Werte zeigen die Ergebnisse der experimentellen Anwendungsanalyse des FIR-Filters. Wie in den ersten beiden Spalten der Tabelle zu sehen ist, beginnt die Analyse mit einer RISC-ähnlichen 1-Slot Prozessorkonfiguration, die jeweils eine Multiplizier- und LD/ST-Einheit besitzt (S1M1L1) und endet bei einer Konfiguration mit 4 VLIW-Slots und einer maximalen Anzahl an Funktionseinheiten (S4M4L2). Da der Instruktionssatzsimulator neben der Anzahl der Prozessortakte auch detaillierte Informationen zur Auslastung der einzelnen Verarbeitungs- und Funktionseinheiten liefern kann, werden diese Simulationen im folgenden Abschnitt verwendet, um die generelle Arbeitsweise des Compilers zu verdeutlichen.

Einleitend muss noch erwähnt werden, dass der Compiler so konfiguriert ist, dass er Instruktionsgruppen immer linksbündig anordnet. Falls eine Instruktionsgruppe nur aus einer Instruktion besteht, wird diese folglich im ersten VLIW-Slot ausgeführt. Falls mehrere Instruktionen parallel ausgeführt werden sollen, werden sie vom niederwertigsten zum höchstwertigsten VLIW-Slot in der Reihenfolge BR, LD/ST, MAC und ALU angeordnet. Sprünge können dementsprechend nur im ersten Slot ausgeführt werden. LD/ST-Instruktionen werden abhängig von der Anzahl der LD/ST-Einheiten entweder im Slot 1 oder in den Slots 1 und 2 platziert. Divisionen werden prinzipiell wie MAC-Instruktionen behandelt. Da der Compiler die diversen DIV-Instruktionen aufgrund ihrer hohen Latenz jedoch in keiner der betrachteten Zielanwendungen verwendet, werden sie in diesem Kapitel nicht weiter betrachtet (siehe Kapitel 3.8).

Name	Slots	MAC	LD/ST	Prozessortakte	Parallelität
S1M1L1	1	1	1	13061	0,99
S2M1L1	2	1	1	10271	1,26
S2M1L2	2	1	2	10254	1,26
S2M2L1	2	2	1	10271	1,26
S2M2L2	2	2	2	10254	1,26
S3M1L1	3	1	1	8030	1,61
S3M1L2	3	1	2	8013	1,61
S3M2L1	3	2	1	8030	1,61
S3M2L2	3	2	2	8013	1,61
S3M3L1	3	3	1	8030	1,61
S3M3L2	3	3	2	8013	1,61
S4M1L1	4	1	1	9161	1,41
S4M1L2	4	1	2	8011	1,61
S4M2L1	4	2	1	9161	1,41
S4M2L2	4	2	2	8011	1,61
S4M3L1	4	3	1	9161	1,41
S4M3L2	4	3	2	8011	1,61
S4M4L1	4	4	1	9161	1,41
S4M4L2	4	4	2	8011	1,61

Tabelle 7.1: Ergebnisse der experimentellen Anwendungsanalyse des FIR-Filters

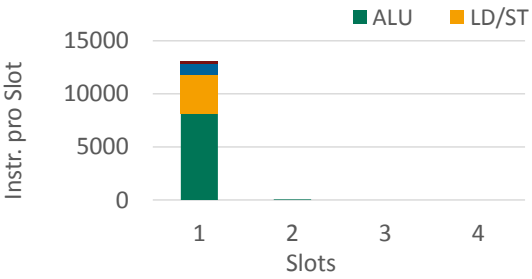

Abbildung 7.1: Instruktionsverteilung S1M1L1

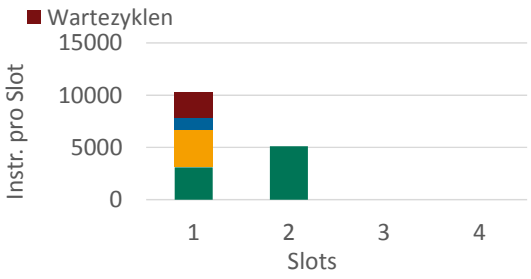

Abbildung 7.2: Instruktionsverteilung S2M1L1

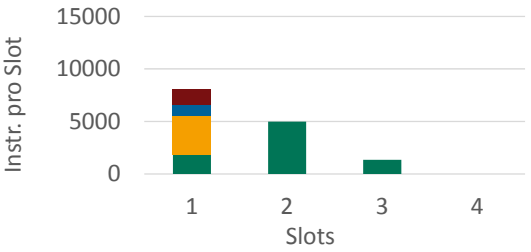

Abbildung 7.3: Instruktionsverteilung S3M1L1

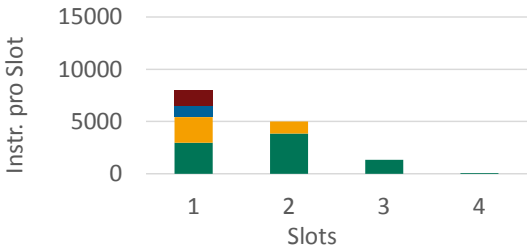

Abbildung 7.4: Instruktionsverteilung S4M4L2

Die Prozessorkonfiguration S1M1L1 benötigt in der experimentellen Simulation 13061 Prozessortakte zur Verarbeitung eines FIR-Durchlaufs (siehe Tabelle 7.1 und Abbildung 7.1). In dieser Zeit werden 8207 ALU-, 1088 MAC- und 3637 LD/ST-Instruktionen verarbeitet. Aufgrund von Datenkonflikten treten jedoch auch 197 Wartezyklen auf. Es ergibt sich somit ein Anteil von MAC- und LD/ST-Instruktionen an der Summe aller Instruktionen von 8,46% und 28,27%. Der Anteil der Wartezyklen an den Prozessortakten ist 1,51%. Obwohl diese Konfiguration nur eine Verarbeitungseinheit besitzt, werden 68 der 8207 ALU-Instruktionen in dem zweiten VLIW-Slot gezählt. Dies rührt daher, dass das in Kapitel 3.6 beschriebene 64-Bit MVC-EXT-Instruktionswort selbst in Prozessoren mit einer Verarbeitungseinheit parallel eingelesen wird.

Durch das Hinzufügen einer zweiten Verarbeitungseinheit (S2M1L1, Abbildung 7.2) können 5116 ALU-Instruktionen in den zweiten Slot verschoben werden und parallel zu anderen Berechnungen ausgeführt werden. Das Verhältnis zwischen ALU-, MAC- und LD/ST-Instruktionen bleibt gleich, die Anzahl der Wartezyklen steigt jedoch auf 2454 Takte. Durch den hohen Anteil an Wartezyklen reduziert sich die Zahl der Prozessortakte lediglich um 21,36%. Mit Hilfe der in Kapitel 5 beschriebenen Formel 5.2 ergibt sich hieraus eine durchschnittliche Parallelität von 1,26. Der zweite VLIW-Slot ist also zu 26% ausgelastet.

Der Anstieg der Wartezyklen resultiert aus der Vorgehensweise des LLVM-Compilers, bei der ein deutlich höherer Fokus auf dem Parallelisieren des Programmcodes, als auf dem Vermeiden von Datenkonflikten liegt. Der Compiler hat bei steigender Parallelität in vielen Fällen nicht mehr die Möglichkeit, Konflikte durch das Umsortieren von Instruktionen aufzulösen, ohne hierdurch an anderer Stelle Konflikte zu erzeugen. Beispiele für das Auftreten dieser Konflikte finden sich in dem Programmcode 7.5 in den Zeilen 18 und 20 (bedingter Sprung folgt direkt auf das Ermitteln der Bedingung) sowie in den Zeilen 14 und 16 (gelesener Wert wird im folgenden Takt weiterverarbeitet). Der Compiler ist sich dieser Konflikte zwar bewusst und versucht sie bestmöglich zu umgehen, da es in diesen Fällen jedoch keine Möglichkeit gibt, die Konflikte zu lösen ohne zusätzliche Prozessortakte einzufügen, verlässt sich der Compiler auf die Konfliktlösungsstrategien der Hardware. Würde er die Konflikte durch das Einfügen zusätzlicher Prozessortakte auflösen, wäre das Resultat zwar eine geringere Anzahl an Wartezyklen, die Anzahl der Prozessortakte und die durchschnittliche Parallelität wäre jedoch identisch.

Da der Anteil der LD/ST- und MAC-Instruktionen vergleichsweise gering ist, verringert das Hinzufügen einer zweiten LD/ST- und MAC-Einheit (S2M2L2) die Anzahl der Prozessortakte lediglich um 0,17%. Das Hinzufügen einer dritten Verarbeitungseinheit (S3M1L1, Abbildung 7.3) reduziert die Anzahl der Prozessortakte im Vergleich zur Konfiguration S2M1L1 hingegen um weitere 21,82% auf 8030 Takte. Diese Verbesserung resultiert aus einer deutlichen Reduzierung der Wartezyklen, da

der Compiler durch den hinzukommenden VLIW-Slot eine Instruktionsanordnung finden kann, die eine Vielzahl von Datenkonflikten auflösen kann.

Es zeigt sich, dass jedoch auch bei der Anzahl der Verarbeitungseinheiten an diesem Punkt eine Sättigung erreicht wird. Ein Prozessor mit vier Verarbeitungseinheiten und einer maximalen Anzahl dedizierter Funktionseinheiten (S4M4L2, Abbildung 7.4) verringert die benötigten Prozessortakte im Vergleich zur Konfiguration S3M1L1 lediglich um 19 Takte beziehungsweise 0,24%. In einigen Fällen ist die Anzahl der Prozessortakte in Konfigurationen mit vier VLIW-Slots sogar höher als bei einer Konfiguration mit drei Verarbeitungseinheiten. Wie im folgenden Abschnitt gezeigt wird, kann das Hinzufügen weiterer Verarbeitungs- und Funktionseinheiten auch bei den anderen Zielanwendungen keine wesentliche Reduzierung der Prozessortakte erwirken, da der Compiler aufgrund von Datenabhängigkeiten zwischen den Instruktionen nicht in der Lage ist, die Parallelität weiter zu erhöhen.

7.2.1 Gegenüberstellung der Zielanwendungen

Die Gegenüberstellung der experimentellen Analyseergebnisse aller Zielanwendungen erfolgt in graphischer Form. Der in Abbildung 7.5 dargestellte Graph zeigt die jeweiligen Anteile der ALU-, LD/ST- und MAC-Instruktionen. Da die Anzahl der vorhandenen Verarbeitungs- und Funktionseinheiten keinen Einfluss auf die Anzahl der auszuführenden Instruktionen hat, gelten diese Werte für alle Prozessorkonfigurationen.

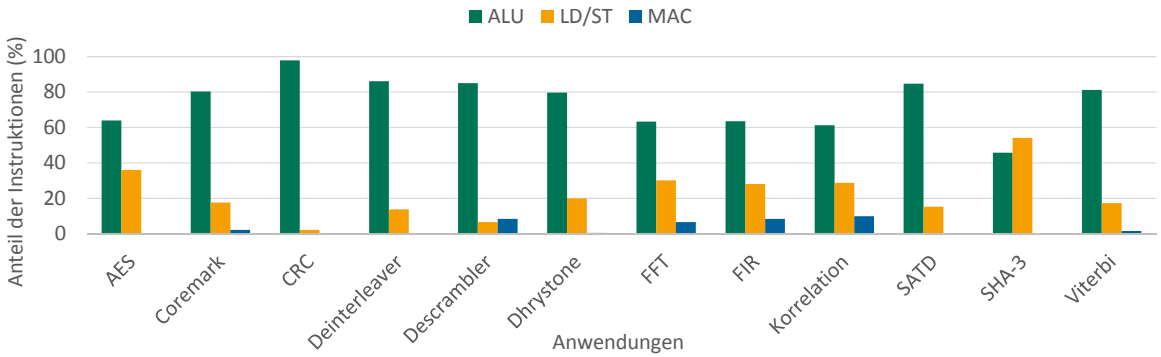

Abbildung 7.5: Anteile der Instruktionstypen bei verschiedenen Zielanwendungen

Es zeigt sich, dass der Anteil der LD/ST-Instruktionen starken Schwankungen unterliegt. Beim CRC-Algorithmus und beim Descrambler liegt der Anteil der LD/ST-Instruktionen mit 2,13% und 6,53% beispielsweise deutlich unter dem durchschnittlichen Anteil dieses Instruktionstyps von 22,48%. Dies erklärt sich dadurch, dass die Anzahl der Rechenoperationen pro Eingangsdatum bei diesen Anwendungen

sehr hoch ist. Beim CRC wird beispielsweise ein einzelnes Eingangsdatum in einer Schleife mit acht Durchläufen und jeweils fünf logischen Operationen verarbeitet (siehe Programmcode 7.2). Aus den resultierenden 40 ALU-Instruktionen pro LD-Instruktion ergibt sich ein rechnerischer LD/ST-Anteil von 2,44%, der sehr gut zu dem gemessenen Wert passt.

Der SHA-3-Algorithmus weist mit 54,20% den höchsten Anteil an LD/ST-Instruktionen auf. Es müssen somit mehr Lade- und Speicheroperationen durchgeführt werden, als tatsächliche Berechnungen stattfinden. Dieser sehr hohe Anteil kommt dadurch zustande, dass der Kernalgorithmus auf einer Matrix mit 50 Werten arbeitet, deren Elemente durch vielfältige logische Operationen miteinander verbunden werden. Da diese Matrix aufgrund der begrenzten Anzahl an Datenregistern nicht vollständig in das Registerfile geladen werden kann, müssen die einzelnen Werte ständig aus dem Datenspeicher nachgeladen werden.

```
1 ...
2 crc_accum = -1;
3 char d = *data_ptr++;
4 for (i=0; i<8; i++) {
5   if ((crc_accum & 0x80000000L) != ((d & 0x80) << 24)) {
6     crc_accum = (crc_accum << 1) ^ POLYNOMIAL;
7   } else {
8     crc_accum <<= 1;
9   }
10 d <<= 1;
11 }
12 ...
```

Programmcode 7.2: Kernalrogithmus der CRC-Berechnung

Zur Gegenüberstellung der durchschnittlichen Parallelität und des resultierenden parallelen und seriellen Anteils der Zielanwendung kommen die Simulationsergebnisse der Konfiguration S4M4L2 sowie die Formeln 5.2, 7.2 und 7.3 zum Einsatz. Die Parallelität ergibt sich aus einer Division der Anzahl aller Instruktionen durch die Prozessortakte. Der parallele und serielle Anteil des Programms wird bestimmt, indem entweder die Summe aller Instruktionen in den VLIW-Slots 2 bis 4 oder die Summe aller Instruktionen im VLIW-Slot 1 inklusive der Wartezyklen durch die Summe aller Instruktionen und Wartezyklen geteilt wird.

$$Parallelanteil = \frac{Instruktionen_{Slot_{2,3,4}}}{Instruktionen_{Slot_{1,2,3,4}} + Wartezyklen} \tag{7.2}$$

$$Seriellanteil = \frac{Instruktionen_{Slot_1} + Wartezyklen}{Instruktionen_{Slot_{1,2,3,4}} + Wartezyklen} \tag{7.3}$$

Wie in Abbildung 7.6 zu sehen ist, ergeben sich bei der durchschnittlichen Parallelität der verschiedenen Zielanwendungen ebenfalls deutliche Unterschiede. Der Grad der Parallelität reicht bei der Prozessorkonfiguration S4M4L2 von 0,83 (Coremark) bis 2,43 (SATD). Der CRC-Algorithmus weist beispielsweise trotz der geringen Anzahl an LD/ST-Instruktionen mit 1,46 eine vergleichsweise schlechte Parallelität auf, da ein Großteil der Berechnungen auf der zentralen Variable crc_accum arbeitet, wodurch viele Datenabhängigkeiten entstehen.

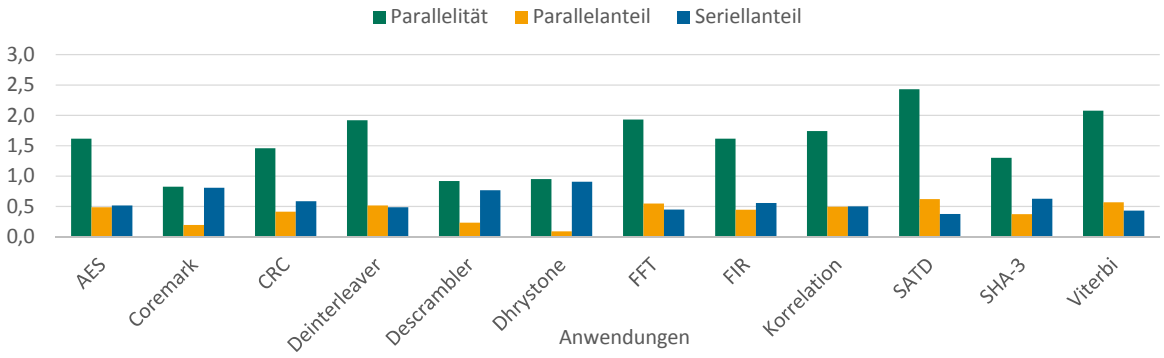

Abbildung 7.6: Parallelität, Parallel- und Seriellanteil der Zielanwendungen in der Konfiguration S4M4L2

Der SATD-Algorithmus besitzt mit 2,43 die höchste durchschnittliche Parallelität, da er hauptsächlich aus einer diskreten Hadamard-Transformation besteht, die auf Additionen in einer 2x2 Matrix basiert (siehe Programmcode 7.3). Die ersten vier Berechnungen weisen keine Datenabhängigkeiten auf und können folglich problemlos parallelisiert werden. Die darauf folgenden Berechnungen besitzen zwar Abhängigkeiten zu den vorangegangenen Instruktionen, diese Abhängigkeiten können jedoch mit Hilfe des Datenregisterbypasses verzögerungsfrei aufgelöst werden, wodurch auch diese Berechnungen parallelisiert werden können. Da die Matrix aufgrund ihrer Größe komplett in das Registerfile eingelesen werden kann, ist die Anzahl der benötigten LD/ST-Instruktionen beim SATD-Algorithmus im Unterschied zum SHA-3-Algorithmus deutlich geringer.

```
1 #define HADAMARD4(d0,d1,d2,d3,s0,s1,s2,s3) {\
2   int16_t t0 = s0 + s1;\
3   int16_t t1 = s0 - s1;\
4   int16_t t2 = s2 + s3;\
5   int16_t t3 = s2 - s3;\
6   d0 = t0 + t2;\
7   d2 = t0 - t2;\
8   d1 = t1 + t3;\
9   d3 = t1 - t3;\
10 }
```

Programmcode 7.3: Kernalgorithmus der SATD-Anwendung

7.3 Anwendungsanalyse mit Approximationsverfahren

Da experimentelle Anwendungsanalysen aufgrund der derzeitigen Größe des Entwurfsraums nicht effizient zur Ermittlung der Leistungsfähigkeit genutzt werden können, beschreibt dieses Kapitel die Charakterisierung der Anwendungen mit Hilfe einer einzelnen konfigurationsunabhängigen Simulation. Das hier gewählte Vorgehen basiert auf den Arbeiten von Moseley [57] und Kambadur [43], die durch einmalige hardwareunabhängige Simulationen die Auftrittshäufigkeit bestimmter Basisblöcke ermitteln und hieraus auf den Parallelanteil einer Anwendung schließen. Zur Ermittlung der Basisblockaufrufe werden in diesen Arbeiten spezielle Makros oder Array-Elemente in den Programmcode eingefügt und während der Simulation ausgewertet. Da der Instruktionssatzsimulator des CoreVA-Prozessors jedoch auch ohne Manipulation des Programmcodes in der Lage ist, die Anzahl und den Typ der Instruktionen in den einzelnen Verarbeitungseinheiten zu bestimmen, wird im Folgenden eine etwas andere Herangehensweise gewählt.

Um konfigurationsunabhängige Ergebnisse zu erhalten, wird die Auslastung der VLIW-Slots bei der Verarbeitung der jeweiligen Zielanwendungen auf einem Prozessor mit maximaler Parallelität ermittelt. Die maximale Parallelität kann hierbei erreicht werden, indem dem Compiler eine Konfiguration mit acht Verarbeitungseinheiten und jeweils acht MAC- und LD/ST-Funktionseinheiten vorgegeben wird. Hierdurch ist gewährleistet, dass der Compiler praktisch keine hardwarespezifischen Einschränkungen erfährt und alle Freiheitsgrade der Parallelverarbeitung voll ausschöpfen kann. In der anschließenden Simulation lässt sich erneut feststellen, dass die Auslastung der Verarbeitungseinheiten bei der Berechnung des FIR-Filters nach dem dritten Slot stark nachlässt. Die MAC- und LD/ST-Funktionseinheiten werden sogar im zweiten Slot praktisch nicht mehr genutzt (siehe Abbildung 7.7).

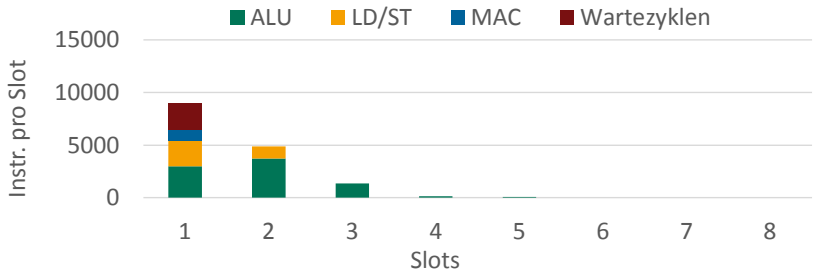

Abbildung 7.7: Instruktionsverteilung in der Prozessorkonfiguration S8M8L8

7.3.1 Approximation der Instruktionsverteilung und der Rechentakte

Auf Basis dieses Simulationsergebnisses lässt sich durch ein Approximationsverfahren die Instruktionsverteilung in beliebigen Prozessorkonfigurationen berechnen. Die Approximation erfolgt hierbei unter der Annahme, dass sich die Instruktionen der wegfallenden Verarbeitungs- oder Funktionseinheiten zu gleichen Teilen auf die verbleibenden Einheiten aufteilen.

Die Formel 7.4 bestimmt aus den Ergebnissen der 8-Slot Simulation die Anzahl der Instruktionen in Slot X eines Prozessors mit N Verarbeitungs- und Funktionseinheiten. Hierzu bildet sie die Summe über die Instruktionen der wegfallenden VLIW-Slots (Slot N bis Slot 8), dividiert diese durch die Anzahl der verbleibenden VLIW-Slots (N), und addiert sie zu der Anzahl der Instruktionen, die ursprünglich in der 8-Slot Konfiguration in Slot X waren. Da der Compiler die Instruktionen vorrangig in den niederwertigen Verarbeitungseinheiten anordnet, beschreibt die Anzahl der Instruktionen in Slot 1 gleichzeitig die Anzahl der benötigten Rechentakte (siehe Formel 7.5). Zu erwähnen ist hierbei, dass dieser Approximationsansatz davon ausgeht, dass alle VLIW-Slots mit MAC- und LD/ST-Einheiten bestückt sind. Des Weiteren dürfen Wartezyklen nicht mitgezählt werden.

$$Instruktionen_{K_N\,Slot_X} = Instruktionen_{K_8\,Slot_X} + \frac{\sum_{i=N+1}^{8} Instruktionen_{K_8\,Slot_i}}{N} \tag{7.4}$$

$$Rechentakte_{K_N} = Instruktionen_{K_N\,Slot_1} \tag{7.5}$$

Da jedoch auch Konfigurationen berücksichtigt werden müssen, bei denen nicht alle VLIW-Slots mit dedizierten MAC- und LD/ST-Einheiten ausgestattet sind, muss die Approximation getrennt nach den unterschiedlichen Instruktionstypen erfolgen.

Die in 7.6 gezeigten Formeln berechnen dementsprechend die Anzahl der ALU-, MAC-, und LD/ST-Instruktionen in den einzelnen VLIW-Slots eines Systems mit N Verarbeitungseinheiten, M MAC-Einheiten und O LD/ST-Einheiten. Die Anzahl der benötigten Rechentakte wird schließlich durch eine Addition der ALU-, MAC- und LD/ST-Instruktionen in Slot 1 berechnet (siehe Formel 7.7).

$$\begin{aligned} ALU_{K_N\,Slot_X} &= ALU_{K_8\,Slot_X} + \frac{\sum_{i=N+1}^{8} ALU_{K_8\,Slot_i}}{N} \\ MAC_{K_M\,Slot_X} &= MAC_{K_8\,Slot_X} + \frac{\sum_{i=M+1}^{8} MAC_{K_8\,Slot_i}}{M} \\ LD/ST_{K_O\,Slot_X} &= LD/ST_{K_8\,Slot_X} + \frac{\sum_{i=O+1}^{8} LD/ST_{K_8\,Slot_i}}{O} \end{aligned} \tag{7.6}$$

$$Rechentakte_{K_{NMO}} = ALU_{K_N\,Slot_1} + MAC_{K_M\,Slot_1} + LD/ST_{K_O\,Slot_1} \tag{7.7}$$

Die Güte dieser Approximationen wird im Folgenden durch einen Vergleich der Rechentakte mit den Ergebnissen der experimentellen Simulationen aus Kapitel 7.2 bestimmt. Wie in Abbildung 7.8 und Tabelle 7.2 zu sehen ist, entsteht durch die Approximation der Rechentakte des FIR-Filters ein durchschnittlicher relativer Fehler von 6,52% und ein maximaler relativer Fehler von 17,44%. Die Approximationen für Konfigurationen mit zwei Verarbeitungseinheiten und für Konfigurationen mit drei oder vier Verarbeitungseinheiten und einer LD/ST-Einheit stechen hierbei besonders hervor.

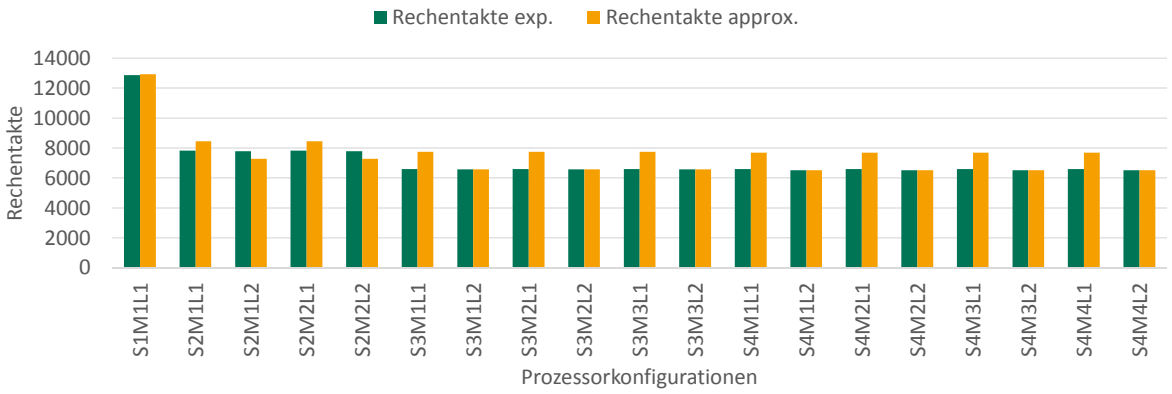

Abbildung 7.8: Vergleich der approximierten und experimentell ermittelten Rechentakte des FIR-Filters

Für diese Abweichungen gibt es zwei verschiedene Ursachen. Zum Einen ist der Compiler bei Konfigurationen mit zwei LD/ST-Einheiten aufgrund von Datenabhängigkeiten häufig nicht in der Lage die zweite LD/ST-Einheit voll auszulasten,

Name	Slots	MAC	LD/ST	Rechentakte experimentell	Rechentakte approximiert	Fehler absolut	Fehler relativ (%)
S1M1L1	1	1	1	12864	12933	69	0,54
S2M1L1	2	1	1	7817	8456	639	8,17
S2M1L2	2	1	2	7800	7287	-514	-6,58
S2M2L1	2	2	1	7817	8456	639	8,17
S2M2L2	2	2	2	7800	7287	-514	-6,58
S3M1L1	3	1	1	6598	7749	1151	17,44
S3M1L2	3	1	2	6581	6580	-1	-0,02
S3M2L1	3	2	1	6598	7749	1151	17,44
S3M2L2	3	2	2	6581	6580	-1	-0,02
S3M3L1	3	3	1	6598	7749	1151	17,44
S3M3L2	3	3	2	6581	6580	-1	-0,02
S4M1L1	4	1	1	6596	7700	1104	16,74
S4M1L2	4	1	2	6515	6531	16	0,25
S4M2L1	4	2	1	6596	7700	1104	16,74
S4M2L2	4	2	2	6515	6531	16	0,25
S4M3L1	4	3	1	6596	7700	1104	16,74
S4M3L2	4	3	2	6515	6531	16	0,25
S4M4L1	4	4	1	6596	7700	1104	16,74
S4M4L2	4	4	2	6515	6531	16	0,25

Tabelle 7.2: Vergleich der approximierten und experimentell ermittelten Rechentakte des FIR-Filters

wodurch die Approximation zu gering ausfällt (z.B. -6,58% bei S2M1L2). Zum Anderen ist es dem Compiler bei Konfigurationen mit mehreren Verarbeitungseinheiten und einer LD/ST-Einheit wider Erwarten möglich, überproportional viele ALU-Instruktionen in die hinteren Verarbeitungseinheiten zu verschieben und somit den VLIW-Slot mit der LD/ST-Einheit zu entlasten (siehe Instruktionsverteilung in den Abbildungen 7.2 und 7.3). Die Approximation der Rechentakte ist hierdurch in manchen Fällen deutlich zu hoch (z.B. +8,17% bei S2M1L1 und +17,44% bei S3M1L1). In den Konfigurationen mit drei oder vier Verarbeitungseinheiten und zwei LD/ST-Einheiten heben sich diese Fehler gegenseitig auf, wodurch Approximationen mit geringer Ungenauigkeit entstehen.

Der Graph 7.9 zeigt schließlich die Auswertung der Approximationsfehler bei der Analyse aller in Kapitel 6 vorgestellten Anwendungen. Zur Verbesserung der Übersichtlichkeit kommen in dieser Darstellung der bereits zuvor genutzte arithmetische Mittelwert und die Standardabweichung zum Einsatz. Da sich die absolute Anzahl der Rechentakte zwischen den verschiedenen Anwendungen stark unterscheidet, beziehen sich die folgenden Analysen jeweils auf einen relativen Fehler, der die prozentuale Abweichung zu den experimentell ermittelten Werten angibt (siehe Formel 7.8). Der arithmetische Mittelwert (Formel 7.9) zeigt dementsprechend den durchschnittlichen relativen Fehler bei der Approximation der verschiedenen Prozessorkonfigurationen. Die Standardabweichung (Formel 7.10) gibt die Streuung der relativen Approximationsfehler an, indem sie die quadratischen Abweichungen der einzelnen Werte von ihrem arithmetischen Mittelwert betrachtet. Bei der Berech-

nung der Standardabweichung ist zu beachten, dass die verwendeten Werte keine Stichproben sind, sondern bereits der Grundgesamtheit entsprechen. Im Nenner steht folglich die Anzahl aller Approximationen (n und nicht n-1) [63].

$$Relativer\ Fehler : x(\%) = \frac{Wert_{Approximation} - Wert_{Experimentell}}{Wert_{Experimentell}} \cdot 100\% \tag{7.8}$$

$$Arithmetischer\ Mittelwert : \overline{x} = \frac{x_1 + x_2 + \cdots + x_n}{n} \tag{7.9}$$

$$Standardabweichung : s = \sqrt{\frac{(x_1 - \overline{x})^2 + (x_2 - \overline{x})^2 + \cdots + (x_n - \overline{x})^2}{n}} \tag{7.10}$$

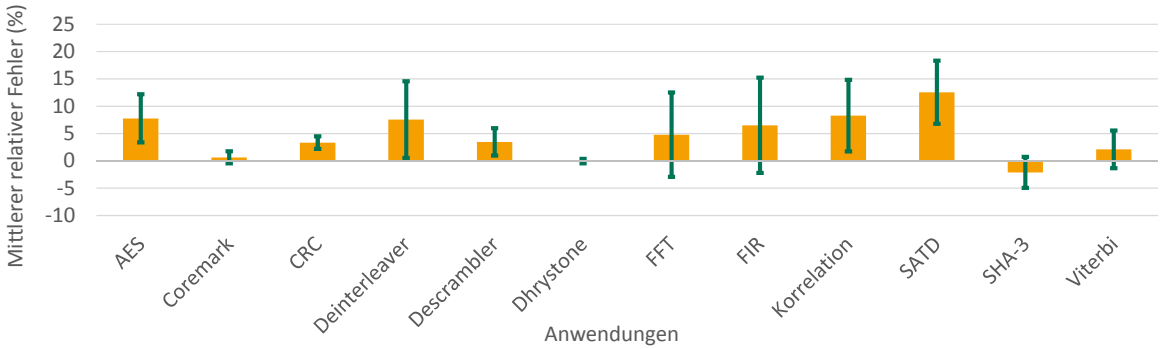

Abbildung 7.9: Mittlerer relativer Fehler und Standardabweichung der Approximation der Rechentakte

Die Auswertung der verschiedenen Anwendungen zeigt, dass die Approximation der Rechentakte einen mittleren relativen Fehler zwischen -2,11% (SHA-3) und 12,57% (SATD) aufweist. Da der Großteil der arithmetischen Mittelwerte positiv ist, lässt sich darauf schließen, dass die Approximationen meistens zu pessimistisch sind. Der Compiler kann die verbleibenden Verarbeitungseinheiten besser auslasten als angenommen. Lediglich bei dem SHA-3-Algorithmus überwiegt das bereits in der Fehleranalyse des FIR-Filters beschriebene Problem, dass der Compiler die zweite LD/ST-Einheit nicht so gut auslastet, wie von der Approximation angenommen.

In Kombination mit der Auswertung der durchschnittlichen Parallelität in Abbildung 7.6 lässt sich die eindeutige Tendenz erkennen, dass die Approximationsfehler in direkter Abhängigkeit zur Parallelität der jeweiligen Anwendungen stehen. Die Anwendungen Coremark und Dhrystone weisen beispielsweise sowohl die geringste Parallelität (<1) als auch den kleinsten mittleren Fehler (<1%) auf. Im Gegenzug besitzt der SATD-Algorithmus mit 2,43 und 12,57% sowohl die höchste Parallelität

als auch den größten Approximationsfehler. Dieser Zusammenhang lässt sich dadurch erklären, dass bei der Approximation der Instruktionsverteilung bei hoher Parallelität deutlich mehr Instruktionen aus den hinteren VLIW-Slots verschoben werden müssen.

7.3.2 Approximation der Prozessortakte

Da die kontinuierliche Ausführung eines Programmcodes häufig durch Wartezyklen unterbrochen wird, müssen diese bei einer Approximation der Prozessortakte ebenfalls betrachtet werden. Einen ersten Approximationsansatz bildet daher eine Addition der Wartezyklen der 8-Slot Konfiguration zu den zuvor ermittelten Rechentakten (siehe Formel 7.11).

$$Prozessortakte = Rechentakte + Wartezyklen \tag{7.11}$$

Weil hierbei jedoch die Annahme getroffen wird, dass die Anzahl der Wartezyklen konfigurationsunabhängig konstant ist, was in den experimentellen Analysen des FIR-Filters in Kapitel 7.2 deutlich widerlegt werden konnte, wurde die Auftrittshäufigkeit der Wartezyklen im Vorfeld genauer untersucht (siehe Abbildung 7.10).

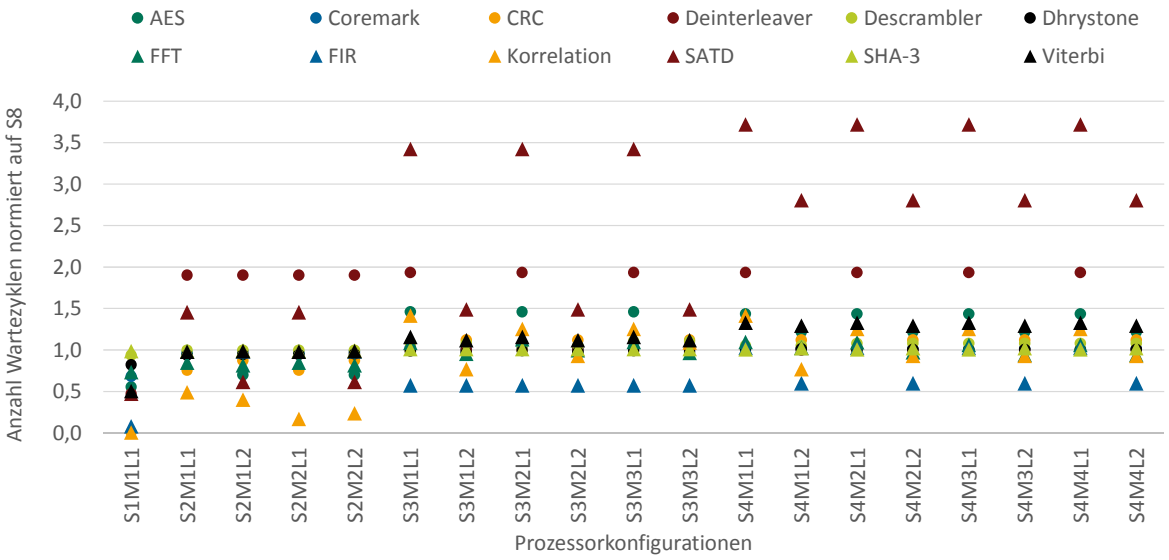

Abbildung 7.10: Anzahl der Wartezyklen normiert auf die Wartezyklen der Konfiguration S8M8L8

Bei der Betrachtung der Wartezyklen zeichnete sich die eindeutige Tendenz ab, dass die tatsächliche Anzahl der zusätzlichen Takte in Konfigurationen mit einer Verarbeitungseinheit etwa der Hälfte der approximierten Anzahl entspricht. Falls mehr als eine Verarbeitungseinheit implementiert ist, unterliegt die Anzahl der Wartezyklen zwar weiterhin starken Schwankungen, es ist jedoch kein weiterer anwendungsunabhängiger Sprung auszumachen. Aus diesem Grund wurde die zuvor entwickelte Approximation um einen Spezialfall für Konfigurationen mit einer Verarbeitungseinheit erweitert (siehe Formel 7.12).

$$Prozessortakte = \begin{cases} Rechentakte + Wartezyklen \cdot 0,5 & \text{, falls } N = 1 \\ Rechentakte + Wartezyklen & \text{, falls } N > 1 \end{cases} \tag{7.12}$$

Die Gegenüberstellung der experimentell ermittelten und approximierten Prozessortakte des FIR-Filters in Tabelle 7.3 und Abbildung 7.11 zeigt, dass sich der Approximationsfehler durch das Hinzufügen der Wartezyklen deutlich erhöht. Im Vergleich zur Approximation der Rechentakte steigt der durchschnittliche relative Fehler von 6,52% auf 12,22% und der maximale relative Fehler von 17,44% auf 27,64%. Die Approximation der Prozessortakte verschlechtert sich, da sie in den Konfigurationen mit zwei LD/ST-Einheiten von deutlich mehr Wartezyklen ausgeht als tatsächlich auftreten. Wie in Abbildung 7.10 zu sehen ist, liegt die tatsächliche Anzahl der Wartezyklen in diesen Konfigurationen lediglich bei 60% der angenommenen Wartezyklen aus der 8-Slot Konfiguration.

Name	Slots	MAC	LD/ST	Prozessortakte experimentell	Prozessortakte approximiert	Fehler absolut	Fehler relativ (%)
S1M1L1	1	1	1	13061	14184	1123	8,59
S2M1L1	2	1	1	10271	10957	686	6,67
S2M1L2	2	1	2	10254	9788	-467	-4,55
S2M2L1	2	2	1	10271	10957	686	6,67
S2M2L2	2	2	2	10254	9788	-467	-4,55
S3M1L1	3	1	1	8030	10250	2220	27,64
S3M1L2	3	1	2	8013	9081	1068	13,32
S3M2L1	3	2	1	8030	10250	2220	27,64
S3M2L2	3	2	2	8013	9081	1068	13,32
S3M3L1	3	3	1	8030	10250	2220	27,64
S3M3L2	3	3	2	8013	9081	1068	13,32
S4M1L1	4	1	1	9161	10201	1040	11,35
S4M1L2	4	1	2	8011	9032	1021	12,74
S4M2L1	4	2	1	9161	10201	1040	11,35
S4M2L2	4	2	2	8011	9032	1021	12,74
S4M3L1	4	3	1	9161	10201	1040	11,35
S4M3L2	4	3	2	8011	9032	1021	12,74
S4M4L1	4	4	1	9161	10201	1040	11,35
S4M4L2	4	4	2	8011	9032	1021	12,74

Tabelle 7.3: Vergleich der approximierten und experimentell ermittelten Prozessortakte des FIR-Filters

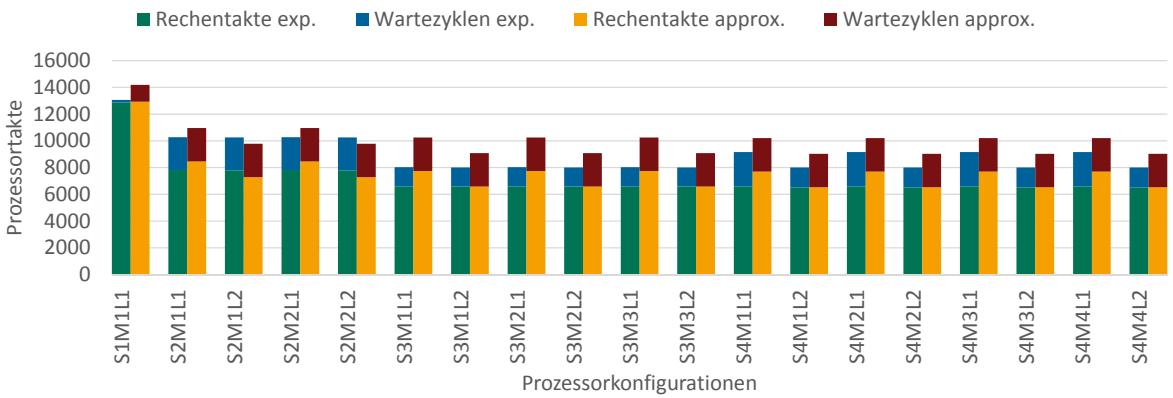

Abbildung 7.11: Vergleich der approximierten und experimentell ermittelten Prozessortakte des FIR-Filters

Im Unterschied zum FIR-Filter sind die Approximationen der Wartezyklen für die anderen Zielanwendungen in den Konfigurationen mit drei oder vier Verarbeitungseinheiten entweder sehr genau oder sogar zu optimistisch. Es kommen folglich mehr Wartezyklen hinzu, als angenommen wird. Da die Approximation der Rechentakte tendenziell zu pessimistisch ist, lässt sich in einem Vergleich der Fehler aller Zielanwendungen feststellen, dass die meisten Approximationen von der Einbeziehung der Wartezyklen profitieren (siehe Abbildung 7.12). Die durchschnittlichen relativen Fehler der Anwendungen AES, Coremark, Deinterleaver, Descrambler und SATD sinken beispielsweise auf unter 1%. Da durch die Wartezyklen ein weiterer Ungenauigkeitsfaktor hinzukommt, steigt jedoch die Standardabweichung in einigen Fällen.

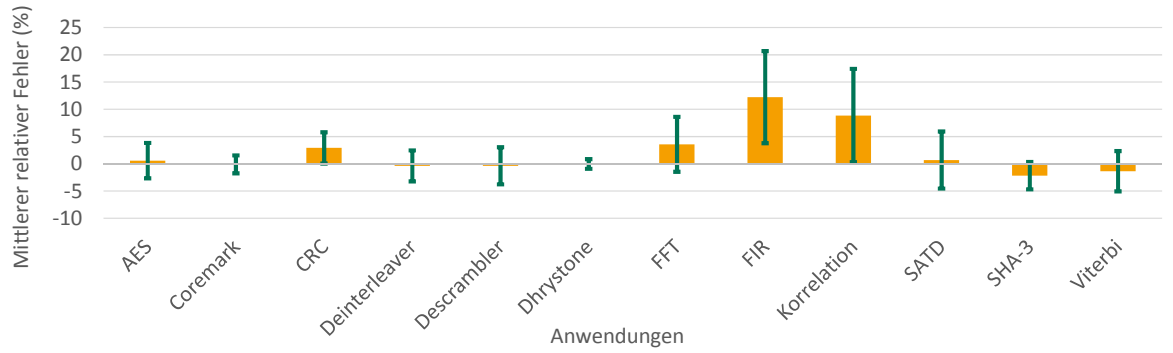

Abbildung 7.12: Mittlerer relativer Fehler und Standardabweichung der Approximation der Prozessortakte

7.4 Statische Anwendungsanalyse

Im Unterschied zur dynamischen Anwendungsanalyse basiert die statische Anwendungsanalyse nicht auf der Auswertung des Laufzeitverhaltens der jeweiligen Anwendungen, sondern ausschließlich auf der Analyse des Programmcodes. Obwohl die statische Anwendungsanalyse nicht in der Lage ist, die tatsächliche Anzahl der Instruktionen und Prozessortakte zu bestimmen, kann sie durch verschiedene Verfahren ebenfalls zur Abschätzung der zu erwartenden Leistungsfähigkeit genutzt werden.

Das folgende Kapitel beschreibt eine statische Programmcodeanalyse auf Basis des in Kapitel 4.1.1 beschriebenen LLVM-Compilers. In Anlehnung an die Arbeiten von Ascia [7] erfolgt die Charakterisierung der Anwendungen anhand ihrer durchschnittlichen Parallelität und der Anzahl der verschiedenen Instruktionstypen. Aus diesen Werten lässt sich abschätzen, in welchem Umfang die Anwendungen von zusätzlichen Verarbeitungs- und Funktionseinheiten profitieren können und inwiefern sich hierdurch ihre Laufzeiten reduzieren lassen.

7.4.1 Der LLVM-IR Programmcode

Der Codeausschnitt 7.4 zeigt den zu Beginn dieses Kapitels vorgestellten Programmcode des FIR-Filters in der LLVM-IR Darstellung nach der Vorverarbeitung durch das LLVM-Frontend Clang. Der LLVM-IR Programmcode zeichnet sich dadurch aus, dass selbst einfache Berechnungen in einem Instruktionsbaum dargestellt werden (siehe Kapitel 4.1.1). Die Instruktionsbäume basieren jeweils auf einer Grundinstruktion, die in diesem Fall aus der Instruktion zum Speichern des Rechenergebnisses in Zeile 27 besteht. Die Grundinstruktion wird in vorgelagerten Definitionen mit einer Vielzahl hardwareunabhängiger Instruktionen verbunden (durch % gekennzeichnet), so dass der Instruktionsbaum sowohl die MAC-Berechnung, als auch das Laden der hierzu benötigten Operanden beinhaltet [50, 85].

Eine Form der statischen Anwendungsanalyse könnte dementsprechend aus der Auswertung der Grundinstruktionen und allen zugehörigen Definitionen bestehen. Ein Vorteil dieser Auswertung wäre, dass der Programmcode den Optimierer und die Endverarbeitung der LLVM-Werkzeugkette nicht durchlaufen müsste und somit keine hardwarespezifischen Komponenten entwickelt werden müssten. Eine Analyse des LLVM-IR Programmcodes wäre dementsprechend für verschiedene Prozessoren oder Prozessorkonfigurationen universell einsetzbar.

```
1 //Kopf der äußeren Schleife
2 for.cond6.preheader: ; preds = %for.body, %for.inc18
3   //Definition: lade kernel
4   %k.066 = phi i32 [%inc19,%for.inc18], [0,%for.body]
5   %arrayidx11 = getelementptr inbounds [17 x i16]* @kernel
6   %2 = load i16* %arrayidx11, align 2, !tbaa !1
7   %conv12 = sext i16 %2 to i32
8   br label %for.body8
9
10 //Kopf der inneren Schleife
11 for.body8: ; preds = %for.body8, %for.cond6.preheader
12   //Definition: lade state
13   %i.165 = phi i32 [0,%for.cond6.preheader], [%inc16,%for.body8]
14   %add9 = add nsw i32 %i.165, %k.066
15   %arrayidx10 = getelementptr inbounds [80 x i16]* @state
16   %3 = load i16* %arrayidx10, align 2, !tbaa !1
17   %conv = sext i16 %3 to i32
18
19   //Definition: mul = kernel * state
20   %mul = mul nsw i32 %conv12, %conv
21   //Definition: lade result
22   %arrayidx13 = getelementptr inbounds [64 x i32]* %result
23   %4 = load i32* %arrayidx13, align 4, !tbaa !5
24   //Definition: add14 = mul + result
25   %add14 = add nsw i32 %mul, %4
26   //Ausführen der add14 Instruktion, Speichern des Ergebnisses
27   store i32 %add14, i32* %arrayidx13, align 4, !tbaa !5
28   //Definition: Schleifenzähler erhöhen und prüfen
29   %inc16 = add nsw i32 %i.165, 1
30   %exitcond68 = icmp eq i32 %inc16, 64
31   //Ausführen Schleifensteuerung (innere Schleife)
32   br i1 %exitcond68, label %for.inc18, label %for.body8
33
34 for.inc18: ; preds = %for.body8
35   //Definition: Schleifenzähler erhöhen und prüfen
36   %inc19 = add nsw i32 %k.066, 1
37   %exitcond69 = icmp eq i32 %inc19, 17
38   //Ausführen Schleifensteuerung (äußere Schleife)
39   br i1 %exitcond69, label %for.inc18, label %for.cond6.preheader
```

Programmcode 7.4: LLVM-IR-Repräsentation des FIR-Algorithmus

Da die hardwareunabhängigen Instruktionen jedoch sehr allgemein gehalten sind, wird das Analyseergebnis stark verzerrt. Wie das obige Beispiel zeigt, kann eine automatisierte Auswertung beispielsweise nicht erkennen, dass die MAC-Instruktionen auf dieser Ebene durch separate Multiplikations- und Additionsinstruktionen beschrieben werden. Des Weiteren geht der hardwareunabhängige Programmcode von einem unendlich großen Registerfile aus, wodurch viele Speicherzugriffe entfallen. Ein weiteres Manko der Auswertung des LLVM-IR Codes ist, dass zu diesem Zeitpunkt noch kein hardwarespezifisches Scheduling erfolgt ist und dementsprechend keine Aussagen zur Parallelität des Programmcodes getroffen werden können. Aus diesen Gründen wird die Analyse der Instruktionen nach der Vorverarbeitung in dieser Arbeit nicht weiter verfolgt.

7.4.2 Der Assemblercode

Der Assemblercode 7.5 zeigt den Kernalgorithmus des FIR-Filters nach der Endverarbeitung durch den LLVM Static Compiler. Wie zu erkennen ist, wurden die hardwareunabhängigen Instruktionen in der Endverarbeitung durch CoreVA-spezifische Instruktionsworte ersetzt. Aus diesem Grund sind jetzt beispielsweise die Multiplikation und Addition in der CoreVA-spezifischen MAC- beziehungsweise MLA-Instruktion zusammengefasst.

Im Unterschied zum vorangegangenen Abschnitt lassen sich nach der Endverarbeitung deutlich verlässlichere Aussagen bezüglich der auszuführenden Instruktionen treffen, da nun beispielsweise aufgrund des endlichen Registerfiles eine Vielzahl zusätzlicher LD/ST-Instruktionen einfügt wurden. Des Weiteren können nun erstmals Instruktionsgruppen analysiert werden, um Erkenntnisse zum Grad der Parallelität und zur Verteilung der Instruktionen auf die verschiedenen Verarbeitungseinheiten zu gewinnen (siehe Kapitel 4.1.1). Das Hauptproblem bei der Analyse der Instruktionsgruppen bildet jedoch die unbekannte Häufigkeit, mit der die einzelnen Gruppen ausgeführt werden. Abhängig von statischen und dynamischen Schleifen oder Fallunterscheidungen kann es sein, dass bestimmte Instruktionsgruppen um ein Vielfaches häufiger ausgeführt werden als andere Gruppen. Wie auch schon bei der LLVM-IR Darstellung kann die Anzahl der Schleifendurchläufe jedoch nur unter sehr großem Aufwand aus dem Assemblercode extrahiert werden. Es kann aus den Abbruchbedingungen der Schleifensteuerung (Zeile 18 und 26) zwar im Klartext abgelesen werden, dass die Schleifen 64- beziehungsweise 17-mal durchlaufen werden. Da die Abbruchbedingungen jedoch nicht direkt mit den Köpfen der Schleifensteuerung verbunden sind, müsste die Anzahl der Instruktionen rückwirkend mit der Anzahl der Durchläufe multipliziert werden. Dies sollte zwar selbst bei mehrfach verschachtelten Schleifenkonstrukten automatisiert möglich sein, dy-

namische Schleifen, bei denen die Ausführungshäufigkeit von den Eingangsdaten abhängt, können mit diesen Mitteln jedoch nicht berücksichtigt werden.

```
1 //Kopf der äußeren Schleife
2 .LBB1_3: //for.cond6.preheader
3  //Addressberechnung
4  c7 lsl r8,r5,1 || c7 mov r9,0
5  c7 add r8,r7,r8
6  //lade Kernel
7  c7 ldw r1,[r8,0]
8  c7 mvsh r8,r1,[r8,0] || c7 mov r1,r6
9
10 //Kopf der inneren Schleife
11 .LBB1_4: //for.body8
12  //lade state und result
13  c7 ldw r2,[r1,0]
14  c7 ldw r3,[r4,r9 lsl 2] || c7 mvsh r2,r2,[r1,0]
15  //MAC und Addressberechnung
16  c7 mla r2,r8,r2,r3 || c7 add r3,r9,1 || c7 add r1,r1,2
17  //Speichern des Ergebnisses und Schleifensteuerung
18  c7 stw r2,[r4,r9 lsl 2] || c7 sub ne,c0,r3,64 || c7 mov r9,r3
19  //Schleifensteuerung (innere Schleife)
20  c0 br .LBB1_4
21  c7 nop //delay-slot
22
23 .LBB1_5: //for.inc18
24  //Schleifensteuerung (äußere Schleife)
25  c7 add r5,r5,1 || c7 add r6,r6,2
26  c7 sub eq,c0,r5,17
27  c0 br .LBB1_6
28  c7 nop //delay-slot
29  c7 br .LBB1_3
30  c7 nop //delay-slot
```

Programmcode 7.5: Assemblercode des FIR-Algorithmus

Abhilfe könnte das automatisierte Abrollen statischer Schleifen schaffen. Da durch das Abrollen der Schleifen jedoch zahlreiche Instruktionen der Schleifensteuerung wegfallen, wird das Auswertungsergebnis hierdurch wiederum verfälscht. Aus diesem Grund müsste der Transformationsblock, der für das automatisierte Abrollen verantwortlich ist, so erweitert werden, dass er anstelle des tatsächlichen Abrollens lediglich die Anzahl der abzurollenden Schleifendurchläufe protokolliert. Hierdurch würden jedoch nur die Schleifen berücksichtigt, die der Compiler als solche erkennt. Die Trefferquote des Compilers ist hierbei jedoch verhältnismäßig schlecht, da diese Funktionalität nicht darauf ausgelegt ist, möglichst viele Schleifen

zu detektieren, sondern die Ausführung geeigneter Schleifen zu optimieren. Dynamische Schleifen können mit diesen Mitteln ebenfalls nicht vollständig erfasst werden. Aus diesem Grund basiert die folgende Anwendungsanalyse wie auch die Arbeiten von Jordans [40], Moseley [57] und Kambadur [43] auf der Auswertung einzelner Basisblöcke.

7.4.3 Analyse einzelner Basisblöcke

Wie man anhand des LLVM-IR und des Assemblercodes sieht, wurden die Instruktionen bereits während der Vorverarbeitung in einzelnen Untergruppen zusammengefasst. Diese sogenannten Basisblöcke (LLVM Basic Blocks) beginnen stets mit einer Sprungmarke[1] und enden mit Sprüngen zum Verlassen oder erneuten Durchlaufen dieses Blocks. Da die Basisblöcke keine weiteren Sprünge beinhalten und somit einen einzigen Eingangs- und Ausgangspunkt besitzen, ist garantiert, dass bei dem Aufruf eines Basisblocks alle enthaltenen Instruktionen genau einmal ausgeführt werden. Aus einer Analyse auf Basisblockebene lassen sich dementsprechend korrekte Informationen zur Instruktionsverteilung und zur durchschnittlichen Parallelität innerhalb eines Basisblocks gewinnen.

Zur Auswertung der einzelnen Basisblöcke wurde in Anlehnung an die Arbeit von Jordans [40] ein zusätzlicher Transformationsblock in den LLVM-Compiler eingefügt. Dieser Analyseblock (Profiling-Pass) untersucht die einzelnen Instruktionsgruppen eines Basisblocks bevor sie in den Assemblercode übertragen werden und ermittelt die Anzahl der enthaltenen Instruktionen und Instruktionstypen (siehe Programmcode 7.6). Ein besonderer Augenmerk musste hierbei auf die korrekte Ermittlung der NOP-Instruktionen und Wartezyklen gelegt werden (siehe Kapitel 7.1). NOP-Instruktionen, die vom Compiler zur Reservierung des Verzögerungsschlitzes gesetzt werden, können vom Analyseblock bei der Auswertung der Instruktionsgruppen problemlos detektiert werden. Wartezyklen, die erst während der Programmausführung zur Auflösung von Datenkonflikten durch die fehlenden Condition-Register Bypasspfade oder durch die erhöhte Latenz der Ladeinstruktionen eingefügt werden, müssen gesondert ermittelt werden. Dies könnte geschehen, indem der Compiler für eine Prozessorhardware konfiguriert würde, die keine Mechanismen zum automatischen Einfügen dieser Wartezyklen besitzt. In diesem Fall würde der Compiler die vermuteten Wartezyklen durch NOP-Instruktionen ersetzen, die mit den oben genannten Mitteln gezählt werden könnten.

Da dieses Vorgehen jedoch vergleichsweise aufwändig ist, wurde die Funktionalität der in Kapitel 3.14 beschriebenen Detektion von Datenkonflikten direkt in den

[1] z.B. for.cond6.preheader in der LLVM-IR Darstellung oder LBB1_3 im Assemblercode

```
1 actual_BB:LBB1_3;label:for.cond6.preheader;
2 profiling:groups:4;instr:6;ldst:1;mac:0;alu:5;br:0;nop:1;
3 ...
4 actual_BB:LBB1_4;label:for.body8;
5 profiling:groups:5;instr:10;ldst:3;mac:1;alu:5;br:1;nop:2;
6 ...
7 actual_BB:LBB1_5;label:for.inc18;
8 profiling:groups:4;instr:5;ldst:0;mac:0;alu:3;br:2;nop:3;
```

Programmcode 7.6: Basisblock-Analysedatei des FIR-Filters

Analyseblock übertragen. Der Analyseblock erfasst hierzu die Zielregisteradressen der Ladeinstruktionen und sämtlicher Instruktionen, die die Inhalte der Condition-Register manipulieren. Falls in der direkt darauffolgenden Instruktionsgruppe eines dieser Zielregister ausgelesen werden soll, wird die Anzahl der Wartezyklen automatisch erhöht. NOP-Instruktionen, die von der Fetch-Stufe zur Rückabwicklung falscher Sprungvorhersagen eingefügt werden, können in der statischen Programmcodeanalyse nicht vorhergesagt werden.

7.4.4 Approximation der Ausführungshäufigkeit der Basisblöcke

Obwohl durch die Analyse auf Basisblockebene vermieden wird, dass es zu Ungenauigkeiten innerhalb eines Basisblocks kommt, sind die Zusammenhänge zwischen den einzelnen Basisblöcken und ihre jeweiligen Ausführungshäufigkeiten weiterhin unbekannt. Der manuellen Analyse des C-Programmcodes in Kapitel 7 und des resultierenden Assemblercodes kann beispielsweise entnommen werden, dass die innere Schleife des FIR-Filters (for.body8) und der Kopf- und Fußteil der äußeren Schleife (for.cond6.preheader und for.inc18) jeweils einen eigenen Basisblock bilden. Der Basisblock der inneren Schleife muss dementsprechend 1088-mal, die Blöcke der äußeren Schleife jedoch jeweils nur 17-mal durchlaufen werden. Da diese Zahlen durch eine automatisierte statische Programmcodeanalyse in manchen Fällen jedoch nicht direkt ermittelt werden können (beispielsweise bei dynamischen Schleifen, siehe Kapitel 7.4.2), kommt zur Bestimmung der Ausführungshäufigkeit ein Verfahren zum Einsatz, das in Anlehnung an die Arbeiten von Wagner [79] und Alba [4] entwickelt wurde.

Im Unterschied zu den Ansätzen von Moseley [57] und Kambadur [43], bei denen von der Anzahl der Basisblockaufrufe auf die Schleifenzugehörigkeiten geschlossen wurde, wird in der Arbeit von Wagner [79] von der Anzahl gefundener Schleifen

auf die Ausführungshäufigkeit der Basisblöcke geschlossen. Hierzu wird mit Hilfe eines GCC-Compilers ein Kontrollflussgraph und ein abstrakter Syntaxbaum des jeweiligen Programmcodes erzeugt, die anschließend in einem separaten Programm zur Ermittlung der Schleifenzugehörigkeit herangezogen werden. Der erste Schritt dieses Verfahrens erfolgt auf intraprozeduraler (funktionsinterner) Ebene, indem der Syntaxbaum der jeweiligen Funktionen nach Schleifen und Fallunterscheidungen durchsucht wird. Falls Schleifen gefunden werden, wird die vergleichsweise simple Voraussage getroffen, dass sie im Durchschnitt fünfmal durchlaufen werden. Für Fallunterscheidungen wird anhand gängiger Programmcodemuster (Idiome) eine Sprungvorhersage durchgeführt. Falls beispielsweise in einem Zweig der Fallunterscheidung ein Programmabbruch erfolgt (assert, return, break), wird für diesen Zweig eine sehr geringe Durchlaufwahrscheinlichkeit angenommen. Falls in einem Zweig jedoch Variablen beschrieben werden, die in darauf folgenden Programmregionen weiterverwendet werden, wird dieser Zweig mit einer deutlich höheren Wahrscheinlichkeit durchlaufen. In einem nachfolgenden Arbeitsschritt werden bei Wagner [79] sämtliche Funktionsaufrufe untersucht, um interprozedurale (funktionsübergreifende) Zusammenhänge berücksichtigen zu können. Hierbei wird anhand der Ausführungshäufigkeit der aufrufenden Basisblöcke auf die Ausführungshäufigkeit der Unterfunktionen geschlossen. Die Basisblöcke, die Teil dieser Unterfunktion sind, werden anschließend zusätzlich mit dieser Ausführungshäufigkeit gewichtet.

Die Arbeit von Alba [4] untersucht ebenfalls die Vorhersagbarkeit von Schleifendurchläufen. Hierbei wird jedoch das Verhalten der Schleifen in gängigen Beispielprogrammen der SPECint2000 Benchmarkserie mittels dynamischer Anwendungsanalysen untersucht. Die Kernaussage dieser Untersuchungen ist, dass ein Großteil der Schleifen bis zu 50-mal durchlaufen wird, sobald sie einmal ausgeführt werden. Bei dieser Beobachtung gab es jedoch auch deutliche Ausreißer, bei denen die durchschnittliche Anzahl der Schleifendurchläufe im fünfstelligen Bereich lag. Diese Ausreißer traten besonders bei Schleifen auf, die eine vergleichsweise große Anzahl an Instruktionen beinhalteten. Aus diesem Grund wurde in weiteren Analysen festgestellt, dass die Anzahl der Schleifendurchläufe mit der Anzahl der enthaltenen Instruktionen steigt.

7.4.4.1 Extraktion der Sprungziele

Um die Ausführungshäufigkeiten der einzelnen Basisblöcke bestimmen zu können, müssen in einem vorgelagerten Arbeitsschritt zuerst die Zusammenhänge zwischen den Basisblöcken ermittelt werden. Dies geschieht wiederum mit Hilfe des bereits beschriebenen Analyseblocks des LLVM-Compilers, der um eine Funktionalität zur Bewertung der jeweiligen Sprunginstruktionen ergänzt wurde. Bei der Analyse

der Sprunginstruktionen müssen ebenfalls zwei verschiedene Arten von Sprüngen unterschieden werden. Zum Einen werden intraprozedurale Sprünge ausgewertet, die zu beliebigen Basisblöcken innerhalb der gleichen Funktion springen, um Schleifenkonstrukte und Fallunterscheidungen abzubilden. Zum Anderen werden Funktionsaufrufe analysiert, die als interprozedurale Sprünge zum jeweils ersten Basisblock der Zielfunktionen beschrieben werden.

Der Analyseblock unterscheidet die verschiedenen Sprungtypen anhand ihrer zugehörigen Instruktionsworte. Intraprozedurale Sprünge werden durch bedingte oder unbedingte Einweg-Sprunginstruktionen (branch) beschrieben. Funktionsaufrufe werden hingegen durch sogenannte branch-and-link Instruktionen beschrieben, die die aktuelle Programmzähleradresse vor dem Ausführen des Sprungs in einem speziellen Register zwischenspeichern, um am Ende der Funktionsverarbeitung an die Adresse des aufrufenden Basisblocks zurückkehren zu können.

Die Sprungziele intraprozeduraler Sprünge können mit Hilfe LLVM-eigener Funktionen ermittelt werden, die beispielsweise durch successor- oder predecessor-Abfragen alle nachfolgenden oder vorausgegangenen Basisblöcke bestimmen können. Da Sprünge stets auf den Beginn eines Basisblocks zeigen, können die einzelnen Blöcke einer Funktion anhand ihrer Startadresse im Instruktionsspeicher eindeutig identifiziert werden. Die Zuordnung der Sprungziele von Funktionsaufrufen ist hingegen aufwändiger, da der Compiler die einzelnen Dateien des Programmcodes separat verarbeitet und deshalb keine Übersicht über die Anordnung der Basisblöcke in anderen Unterfunktionen besitzt. Aus diesem Grund extrahiert der Analyseblock die Operanden der jeweiligen Funktionsaufrufe und ermittelt den Namen der Unterfunktionen. Die so gesammelten Informationen werden anschließend zusammen mit den Informationen über die in den jeweiligen Basisblöcken enthaltenen Instruktionstypen in Textdateien gespeichert. Actual_BB kennzeichnet hierbei den untersuchten Basisblock, succ_BB listet die zugehörigen Sprungziele. Um Doppelbenennungen von Blöcken in verschiedenen Unterfunktionen zu vermeiden, werden die Basisblöcke durch eine Kombination des Funktionsnamens und der derzeitigen Speicheradresse identifiziert (siehe Programmcode 7.7).

7.4.4.2 Ermittlung der Schleifenzugehörigkeiten

Um das Vorgehen von Wagner [79] nachbilden zu können, wurde ein Auswertungsprogramm entwickelt, dass die Analysedateien aller zugehörigen Funktionen einliest und die ermittelten Basisblöcke als Knoten in einen gerichteten Graphen überträgt. Sobald alle Knoten angelegt sind, werden die Analyseergebnisse erneut durchlaufen, um alle intraprozeduralen Sprünge auszuwerten und entsprechende Kanten zu erzeugen. Abbildung 7.13 zeigt die hierdurch entstehenden intraprozeduralen Teil-

```
1 function:fir;begin_BB:fir_82501872;end_BB:fir_82503232
2 ...
3 actual_BB:fir_82502320;label:for.cond6.preheader;
4 profiling:groups:4;instr:6;ldst:1;mac:0;alu:5;br:0;nop:1;
5 succ_BB:fir_82502688
6 ...
7 actual_BB:fir_82502688;label:for.body8;
8 profiling:groups:5;instr:10;ldst:3;mac:1;alu:5;br:1;nop:2;
9 succ_BB:fir_82503056,fir_82502688
10 ...
11 actual_BB:fir_82503056;label:for.inc18;
12 profiling:groups:4;instr:5;ldst:0;mac:0;alu:3;br:2;nop:3;
13 succ_BB:fir_82676400,fir_82502320
14 ...
```

Programmcode 7.7: Durch Sprungziele erweiterte Analysedatei des FIR-Filters

graphen. Wie zu erkennen ist, wurde der FIR-Algorithmus zu Erklärungszwecken und zur Überprüfung der Funktionalität testweise um einen weiteren Verarbeitungsschritt ergänzt. Dieser Schritt wird durch eine zusätzliche Schleife beschrieben, die in die Unterfunktion functest ausgelagert wurde. Der Funktionsaufruf erfolgt in der inneren Schleife des FIR-Algorithmus.

In einem dritten Verarbeitungsschritt werden die aufgefundenen Funktionsaufrufe verarbeitet, um die verschiedenen Teilgraphen zu einem interprozeduralen Graphen zu verbinden. Hierbei kommen einige Hilfsfunktionalitäten zum Einsatz. Zum Einen wird die Unterfunktion genauer untersucht, um den Namen des ersten und letzten Basisblocks zu erhalten und somit Verbindungen über Funktionsgrenzen hinweg zu ermöglichen. Zum Anderen wird der aufrufende Basisblock in zwei Teile zerlegt, um sicherzustellen, dass Funktionsaufrufe in Schleifen zusätzlich gewichtet werden können. Diese Aufteilung vermeidet auch das im Folgenden beschriebene Problem der Querkanten. Wie die Abbildung 7.14 zeigt, wird in diesem Fall der Knoten fir_for.body8 in die Knoten fir_for.body8_begin und fir_for.body8_end aufgeteilt. Anschließend wird der Funktionsaufruf eingefügt, indem eine Kante vom führenden Knoten zum Startbasisblock der Unterfunktion erzeugt wird. Der Rücksprung aus der Unterfunktion wird durch eine zusätzliche Kante vom Endbasisblock der Unterfunktion zum unteren Knoten des aufrufenden Basisblocks abgebildet. Die Kanten, die in den ursprünglichen Knoten hinein gingen, werden an den neuen Eingangsknoten angelegt.

Auf Basis dieses Graphen wird schließlich eine Zyklensuche durchgeführt, die Wege von Knoten zu sich selbst detektieren kann und somit sämtliche Schleifen einer Funktion findet. Die Zyklensuche basiert auf einer Tiefensuche, die alle Knoten eines

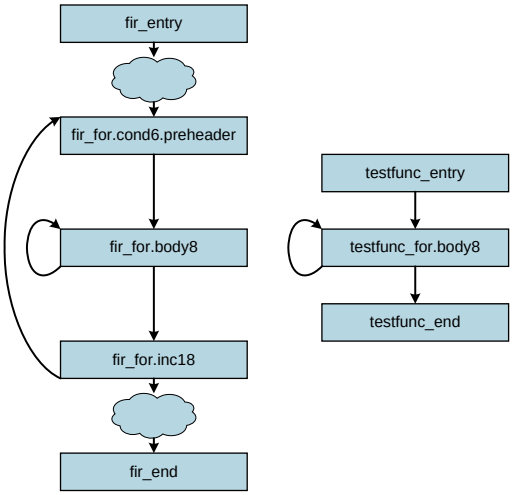

Abbildung 7.13: Teilgraph des FIR-Algorithmus

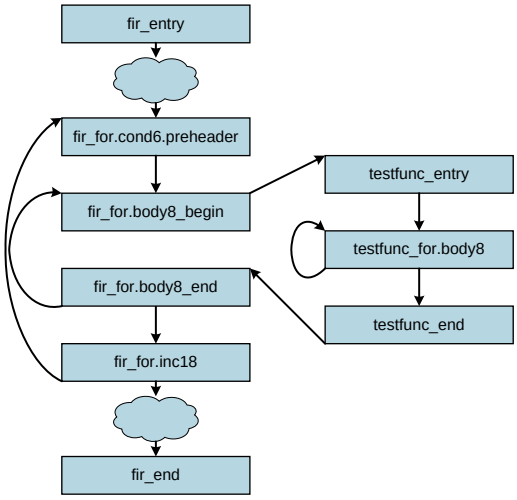

Abbildung 7.14: Vollständiger Graph des FIR-Algorithmus

Graphen durchsucht. Wie der unter 7.8 dargestellte Pseudocode einer rekursiven Tiefensuche zeigt, beginnt der Suchalgorithmus, indem er den aktuellen Knoten als „besucht“ markiert. Anschließend untersucht der Algorithmus die angeschlossenen Knoten, indem er alle Kanten entlang geht, um weitere unbesuchte Knoten zu finden. Findet er einen unbesuchten Nachfolgeknoten, startet er die Tiefensuche bei diesem Knoten erneut [78].

```
1 Tiefensuche(Knoten) {
2   besucht(Knoten) = 1 //markiere Knoten als besucht
3   for (alle Nachfolger in Kanten) { //prüfe angeschlossene Knoten
4     if (besucht(Nachfolger)==0) { //falls Nachfolger nicht besucht
5       Tiefensuche(Nachfolger) //Tiefensuche mit Nachfolger starten
6     }
7   }
8 }
```

Programmcode 7.8: Tiefensuche

Zur Zyklensuche wurde dieser Algorithmus durch die rot dargestellten Programmzeilen so erweitert (siehe Programmcode 7.9), dass er nicht nur erkennt ob der aktuelle Knoten ein Teil eines Zyklus ist, sondern dass er alle darin enthaltenen Knoten wiedergeben kann. Hierzu merkt sich der Algorithmus bei jedem rekursiven Aufruf von welchem Vorgänger der neue Knoten angesprochen wurde. Da der Kontrollflussgraph gerichtet ist, kann der Algorithmus beim Erreichen eines bereits besuchten Knotens davon ausgehen, dass eine Rückwärtskante gefunden wurde, die einen Zyklus erzeugt. Zur Auswertung dieses Zyklus muss der Algorithmus

anschließend den gespeicherten Suchpfad zurückgehen und die enthaltenen Knoten ausgeben [78].

```
1 Zyklensuche(Knoten) {
2   besucht(Knoten) = 1 //markiere Knoten als besucht
3   for (alle Nachfolger in Kanten) { //prüfe angeschlossene Knoten
4     if (besucht(Nachfolger)==0) { //falls Nachfolger nicht besucht
5       Suchpfad[Nachfolger] = Knoten //speichere Schritt zu Nachf.
6       Zyklensuche(Nachfolger) //Tiefensuche mit Nachfolger starten
7     }
8     else { //besuchter Knoten erreicht, Zyklus gefunden
9       while (Knoten!=Nachfolger) { //gehe Suchpfad zurück
10        print(Knoten) //gebe Knoten des Suchpfad aus
11        Knoten = Suchpfad[Knoten]
12      }
13      print(Knoten) //Spezialfall, Zyklus mit 1 Knoten
14    }
15  }
16 }
```

Programmcode 7.9: Zyklensuche

Bei der Analyse des zu Testzwecken erweiterten FIR-Filters stellt die Zyklensuche richtigerweise fest, dass die Basisblöcke for.cond6.preheader und for.inc18 jeweils in einem Zyklus enthalten sind. Dieser Zyklus bildet folglich die äußere for-Schleife ab. Der Basisblock fir.body8 sowie die Eingangs- und Ausgangsbasisblöcke der Unterfunktion testfunc_entry und testfunc_end sind in zwei Schleifen enthalten, nämlich der äußeren und inneren for-Schleife. Der Basisblock testfunc_for.body8 ist schließlich in drei Schleifen enthalten.

7.4.4.3 Berücksichtigung von Querkanten

Solange ein Graph wie im Beispiel des FIR-Filters nur Vorwärts- und Rückwärtskanten besitzt, wird die Anzahl der Schleifen durch das oben genannte Vorgehen korrekt ermittelt [78]. Falls sich ein Graph zur Abbildung von Fallunterscheidungen jedoch zeitweilig in Teilgraphen aufteilt, kann es dazu kommen, dass einzelne Zyklen doppelt gezählt werden. Dieses Problem wird anhand des Beispiels in Abbildung 7.15 verdeutlicht, bei dem eine Fallunterscheidung in den Programmcode der Funktion testfunc eingefügt wurde, die das Ausführen der inneren Schleife gegebenenfalls überspringt. Diese Fallunterscheidung erzeugt den Teilgraphen 1, der von testfunc_entry über testfunc_for.body8 nach testfunc_end führt, und den

Teilgraphen 2, der direkt von testfunc_entry nach testfunc_end führt. Falls der Algorithmus nun durch Zufall bei der Suche nach Zyklen vom Knoten testfunc_entry zuerst den Teilgraphen 1 durchläuft, werden die Basisblöcke testfunc_for.body8 und testfunc_end als besucht markiert. Bei der anschließenden Suche im Teilgraphen 2 würde der Algorithmus erneut zum Knoten testfunc_end gelangen und aufgrund der Markierung fälschlicherweise eine weitere Schleife detektieren.

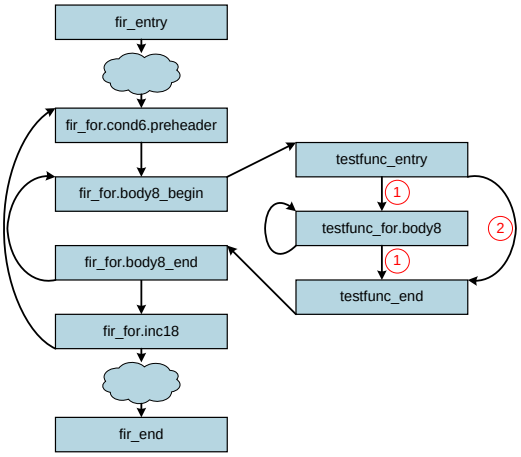

Abbildung 7.15: Graph des FIR-Algorithmus mit Querkante

Um dieses Problem zu vermeiden, wurde der oben genannte Algorithmus nochmals erweitert, so dass er zusätzlich prüft, ob ein Knoten vollständig abgeschlossen wurde (siehe Programmcode 7.10). Falls nun der Algorithmus zuerst den Teilgraphen 1 bearbeitet, wird der Knoten testfunc_end als abgeschlossen markiert, da er nur einen Ausgang besitzt, über den bereits ein Zyklus gefunden werden konnte. Bei der anschließenden Untersuchung des Teilgraphen 2 würde folgerichtig eine Querkante detektiert, woraufhin allen enthaltenen Knoten die Schleifenzugehörigkeit des Knotens zugeordnet wird, bei dem sich die Teilbäume wieder vereinen. Für die Zweige einer Fallunterscheidung wird somit immer die gleiche Ausführungshäufigkeit angenommen [78].

Die Graphentheorie und die Mechanismen zum Abfangen spezieller Schleifenkonstellationen gehen weit über den hier dargestellten Ansatz hinaus. Es sind diverse Sonderfälle denkbar, die durch den Algorithmus nicht erkannt werden. So könnten beispielsweise Graphen entstehen, bei denen verschiedene Schleifenebenen durch Querkanten miteinander verbunden werden. In diesem Fall würden die innere Schleife nicht korrekt erkannt und bliebe gegebenenfalls unberücksichtigt. Die in dieser Arbeit angestellten Untersuchungen haben jedoch ergeben, dass der Compiler solche Schleifenkonstrukte in der Praxis nicht einsetzt. Des Weiteren wurde bei der Entwicklung dieses Algorithmus sichergestellt, dass er trotz etwaiger Fehlentscheidungen kontrolliert weiterläuft und nicht in einen undefinierten Zustand gerät.

```
1 Zyklensuche(Knoten) {
2   besucht(Knoten) = 1 //markiere Knoten als besucht
3   for (alle Nachfolger in Kanten) { //prüfe angeschlossene Knoten
4     if (besucht(Nachfolger)==0) { //falls Nachfolger nicht besucht
5       Suchpfad[Nachfolger] = Knoten //speichere Schritt zu Nachf.
6       Zyklensuche(Nachfolger) //Tiefensuche mit Nachfolger starten
7     }
8     elseif (besucht(Nachfolger)==1) { //besuchter Knoten erreicht,
9                                       //Zyklus gefunden
10      while (Knoten!=Nachfolger) { //gehe Suchpfad zurück
11        print(Knoten) //gebe Knoten des Suchpfad aus
12        Knoten = Suchpfad[Knoten]
13      }
14      print(Knoten) //Spezialfall, Zyklus mit 1 Knoten
15    } else { //abgeschlossener Knoten erreicht,Querkante gefunden
16      print(Knoten) //gebe Querkante aus
17    }
18  }
19  besucht(Knoten) = 2 //markiere Knoten als abgeschlossen
20 }
```

Programmcode 7.10: Zyklensuche mit Querkantenerkennung

7.4.5 Gewichtung der Analyseergebnisse

Um die Analyseergebnisse der einzelnen Basisblöcke zu einem sinnvollen Gesamtergebnis zusammenfassen zu können, werden die Teilergebnisse mit der approximierten Ausführungshäufigkeit gewichtet und anschließend aufsummiert. Die Gewichtung erfolgt, indem die Ergebnisse für jede gefundene Schleifenzugehörigkeit mit einer durchschnittlichen Anzahl an Schleifendurchläufen multipliziert werden. In Anlehnung an Wagner [79] wird in dieser Arbeit davon ausgegangen, dass Schleifen im Durchschnitt fünfmal durchlaufen werden. Die Formel 7.13 zeigt die exemplarische Approximation der Gesamtanzahl aller auszuführenden Instruktionen. Auf die gleiche Weise lässt sich jedoch auch die Anzahl der einzelnen Instruktionstypen und der Instruktionsgruppen bestimmen.

$$Instruktionen = \sum_{BB=0}^{N} Instruktionen_{BB} \cdot 5^{Schleifenzugehörigkeiten_{BB}} \tag{7.13}$$

Im Falle des ursprünglichen FIR-Algorithmus werden die Analyseergebnisse der Basisblöcke for.cond6.preheader und for.inc18 mit dem Faktor 5 multipliziert, da sie

nur in der äußeren Schleife vorhanden sind. Die Ergebnisse des Basisblocks fir.body8 werden mit dem Faktor 25 multipliziert, da dieser Block in der inneren und äußeren Schleife enthalten ist (siehe Programmcode 7.11).

```
1 function:fir;begin_BB:fir_82501872;end_BB:fir_82503232
2 ...
3 actual_BB:fir_82502320;label:for.cond6.preheader;
4 weight:5;
5 profiling:groups:20;instr:30;ldst:5;mac:0;alu:25;br:0;nop:5;
6 succ_BB:fir_82502688
7 ...
8 actual_BB:fir_82502688;label:for.body8;
9 weight:25;
10 profiling:groups:125;instr:250;ldst:75;mac:25;alu:125;br:25;
11 nop:50;
12 succ_BB:fir_82503056,fir_82502688
13 ...
14 actual_BB:fir_82503056;label:for.inc18;
15 weight:5;
16 profiling:groups:20;instr:25;ldst:0;mac:0;alu:15;br:10;nop:15;
17 succ_BB:fir_82676400,fir_82502320
18 ...
```

Programmcode 7.11: Gewichtete Analysedatei des FIR-Filters

Da die tatsächliche Anzahl der Schleifendurchläufe jedoch weiterhin unbekannt ist, ist es im Folgenden nicht sinnvoll, mit absoluten Zahlen zu arbeiten. Aus diesem Grund werden die ermittelten Ergebnisse ausschließlich genutzt, um die relativen Anteile der ALU-, MAC- und LD/ST-Instruktionen zu bestimmen und die durchschnittliche Parallelität sowie den Parallelanteil zu berechnen. Bei der Berechnung der Parallelität kommt wiederum die Formel 5.2 zum Einsatz. Zur Bestimmung des Parallelanteils wird wie in der dynamischen Anwendungsanalyse in Kapitel 7.2 die Summe der Instruktionen in den zusätzlichen Verarbeitungseinheiten durch die Summe aller Instruktionsworte und Wartezyklen geteilt (siehe Formel 7.2). Da die Summe der Instruktionen in den hinteren Verarbeitungseinheiten bei dem hier verwendeten Vorgehen jedoch unbekannt ist, wird stattdessen die Anzahl der Instruktionsgruppen von der Summe aller Instruktionen abgezogen (siehe Formel 7.14).

$$Instruktionen_{Slot_{2,\ldots,N}} = Instruktionen - Instruktionsgruppen \tag{7.14}$$

Die abschließende Analyse der verschiedenen Zielanwendungen erfolgt jeweils für die Prozessorkonfiguration S8M8L8. Die Auswertung des FIR-Filters ergibt bei

dieser Konfiguration eine durchschnittliche Parallelität von 1,46 und einen Parallelanteil von 0,40. Der Anteil der LD/ST- und MAC-Instruktionen liegt bei 26,76% und 4,88%. Ein Vergleich dieser Werte mit den Ergebnissen einer experimentellen Anwendungsanalyse zeigt bei der statisch ermittelten Parallelität und dem Parallelanteil einen relativen Fehler von 1,69% und -3,07%. Die Anteile der verschiedenen Instruktionstypen weisen einen mittleren Fehler von -23,42% auf. Da die Ergebnisse der statischen Anwendungsanalyse für die einzelnen Basisblöcke nahezu korrekt sind, entstehen diese Abweichungen ausschließlich durch die ungenaue Approximation der Basisblockdurchläufe.

Bei der Analyse der Approximationsfehler lassen sich jedoch keine weiteren Tendenzen ausmachen, die in die Modellierung einfließen könnten. Wie die Gegenüberstellung der statisch und experimentell ermittelten Anteile der LD/ST- und MAC-Instruktionen sowie der durchschnittlichen Parallelität und des Parallelanteils aller Zielanwendungen in den Graphen 7.16 und 7.17 zeigen, weist die statische Programmcodeanalyse sowohl positive als auch negative Fehler auf. Im Schnitt ergibt sich für die hier untersuchten Anwendungen ein mittlerer relativer Fehler von 9,40% und eine Standardabweichung von ±43,74%.

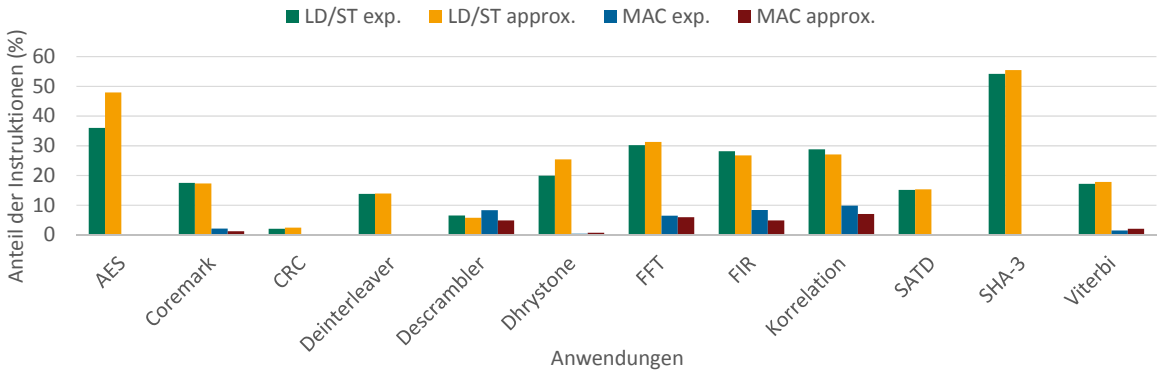

Abbildung 7.16: Gegenüberstellung der statisch und experimentell ermittelten Anteile der LD/ST- und MAC-Instruktionen

7.5 Zusammenfassung

In diesem Kapitel wurden zwei modellbasierte Analyseverfahren vorgestellt, die entwickelt wurden, um Anwendungsprogramme zu charakterisieren und hiermit auf die Leistungsfähigkeit verschiedener Prozessorkonfigurationen zu schließen. Die Analysen bestimmen die Anzahl der benötigten Prozessortakte, den Parallelitätsgrad, die verwendeten Instruktionstypen und gegebenenfalls sogar die Auslastung

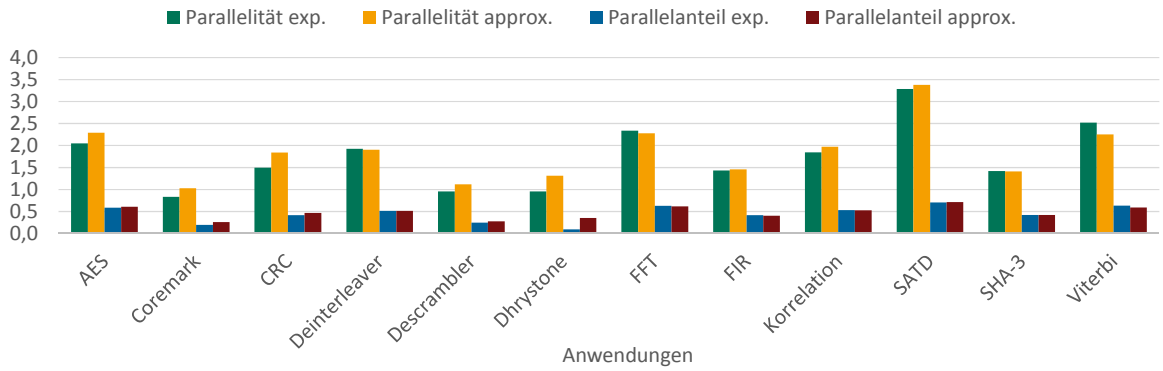

Abbildung 7.17: Gegenüberstellung der statisch und experimentell ermittelten durchschnittlichen Parallelität und des Parallelanteils

der einzelnen Verarbeitungs- und Funktionseinheiten für unterschiedliche Prozessorkonfigurationen.

Zunächst wurden zahlreiche konfigurationsspezifische Anwendungsanalysen durchgeführt, um die generelle Arbeitsweise des Compilers zu verdeutlichen und eine Referenz für die Entwicklung der modellbasierten Verfahren zu generieren. Hierbei wurde die Ausführung der Zielanwendungen auf neunzehn verschiedenen Prozessorkonfigurationen simuliert und ausgewertet. Die Anzahl der VLIW-Slots und der MAC-Einheiten erhöhte sich schrittweise von eins auf vier, die Anzahl der LD/ST-Einheiten wechselte zwischen eins und zwei. Die Analysen ergaben, dass die Eigenschaften der Zielanwendungen deutlich variieren. So reicht die durchschnittliche Parallelität von 0,83 (Coremark) bis 2,43 (SATD). Die Anteile der LD/ST-Instruktionen schwanken zwischen 2,44% (CRC) und 54,20% (SHA-3).

Da eine vollständige Entwurfsraumexploration auf Basis konfigurationsspezifischer Analysen wegen der Größe des Entwurfsraums nicht praktikabel ist, wurde in dieser Arbeit ein modellbasiertes Verfahren entwickelt, dass die gesuchten Werte mit Hilfe einer einzelnen konfigurationsunabhängigen Simulation approximiert. Bei dieser Simulation wird die Auslastung der einzelnen Verarbeitungs- und Funktionseinheiten eines Prozessors mit maximaler Parallelität aufgezeichnet. Aus den Simulationsergebnissen wird anschließend auf die Instruktionsverteilung beliebiger Prozessorkonfigurationen geschlossen, indem angenommen wird, dass sich die Instruktionen der wegfallenden Verarbeitungseinheiten zu gleichen Teilen auf die verbleibenden Einheiten verteilen. Die Auswertung der verschiedenen Zielanwendungen zeigte, dass die hieraus abgeleitete Approximation der benötigten Prozessortakte einen mittleren relativen Fehler von 2,03% bei einer Standardabweichung von ±6,24% aufweist.

In einem zweiten Schritt wurde eine statische Anwendungsanalyse entwickelt, die ausschließlich auf der Auswertung des Programmcodes der jeweiligen Zielanwendungen basiert. Bei diesem Verfahren werden die im Programmcode enthaltenen Instruktionsgruppen während des Kompiliervorgangs durch einen zusätzlichen LLVM-Transformationsblock eingelesen und bezüglich ihrer Parallelität und ihrer Instruktionstypen untersucht. Da bei diesem Verfahren auf Simulationen verzichtet wird, kann die statische Programmcodeanalyse das Laufzeitverhalten der Anwendungen nicht berücksichtigen und dementsprechend nicht ermitteln, wie oft die einzelnen Instruktionsgruppen ausgeführt werden. Abhängig von statischen und dynamischen Schleifen kann es jedoch sein, dass einzelne Instruktionsgruppen um ein Vielfaches häufiger ausgeführt werden als die anderen Gruppen. Um die hieraus resultierenden Fehler zu reduzieren, wurde die Analyse um ein Verfahren zur Approximation der Ausführungshäufigkeiten ergänzt. Der LLVM-Compiler extrahiert hierzu neben den zuvor genannten Informationen auch die enthaltenen Sprungziele. Anschließend wird mit Hilfe dieser Sprungziele ein gerichteter Graph aufgebaut und eine Zyklensuche durchgeführt, die die Analyseergebnisse der Instruktionsgruppen mit der Anzahl ihrer Schleifenzugehörigkeiten gewichtet. Ein Vergleich der hierdurch ermittelten Parallelitäten und der Anteile der MAC- und LD/ST-Instruktionen mit den Ergebnissen der experimentellen Anwendungsanalysen zeigte schließlich einen relativen Fehler von 9,40% (±43,74%).

8 Ressourcenanforderungen des Prozessorkerns

Dieses Kapitel untersucht die Ressourcenanforderungen der verschiedenen Konfigurationselemente des CoreVA-Prozessors mit dem Ziel, ein Hardwaremodell des Prozessorkerns zu erstellen. Um die Anzahl der hierzu benötigten Synthesen auf einen praktikablen Wert einzugrenzen, ist der Entwurfsraum bei dieser Analyse auf die Elemente beschränkt, die einen direkten Einfluss auf die Prozessorpipeline haben. Zu diesen Konfigurationselementen gehört die Anzahl der VLIW-Slots, die Anzahl der zugehörigen MAC- und LD/ST-Einheiten und die Aktivierung der SIMD-Unterstützung und des Resource-Sharings. Im Unterschied zum vorangegangenen Kapitel wird in diesem Fall auch die Anzahl der Dividiereinheiten variiert, da sie ebenfalls einen deutlichen Einfluss auf die Ressourcenanforderungen hat. Für Systeme mit zwei LD/ST-Einheiten wird ferner die Implementierung des Datenspeichers als Dual-Port oder Multi-Bank Speicher unterschieden (siehe Kapitel 3.15).

Die Einflüsse der Konfigurationselemente werden durch das sukzessive Variieren der einzelnen Parameter bestimmt. Da diese Untersuchungen dem Bilden eines Modells dienen, ist die maximale Anzahl paralleler Verarbeitungseinheiten auf vier VLIW-Slots begrenzt. Die Implementierungen des Datenspeichers sehen einen maximalen Einsatz von zwei LD/ST-Einheiten vor. Es ergeben sich dementsprechend die in Tabelle 8.1 beschriebenen Prozessorkonfigurationen.

In einem ersten Schritt wird der Einfluss der jeweiligen Prozessorkonfigurationen auf die Latenz des kritischen Pfades bestimmt. Anschließend werden Untersuchungen zu Ermittlung der durchschnittlichen Schaltaktivitäten durchgeführt und die Konfigurationen bezüglich ihrer Leistungsaufnahme und Prozessorfläche bewertet. Dies geschieht zuerst bei deaktivierter SIMD-Unterstützung und ohne Resource-Sharing, da die Ressourcenanforderungen dieser übergeordneten Funktionalitäten in einem nachfolgenden Schritt getrennt betrachtet werden.

Die Werte für diese Analysen werden mit Hilfe von Standardzellensynthesen mit dem Synthesewerkzeug Cadence Encounter RTL Compiler 14.11 erzeugt (siehe Kapitel 4.2.2). Hierbei kommen Standardzellen und Speicherblöcke in einer 28 nm FD-SOI Technologie der Firma STMicroelectronics zum Einsatz. Die Randbedingungen an

Name	Slots	MAC	DIV	LD/ST	Breite Instruktions-speicher (Bit)	Breite Daten-speicher (Bit)
S1M1D1L1	1	1	1	1	64	32
S2M1D1L1	2	1	1	1	64	32
S3M1D1L1	3	1	1	1	128	32
S4M1D1L1	4	1	1	1	128	32
S4M2D1L1	4	2	1	1	128	32
S4M3D1L1	4	3	1	1	128	32
S4M4D1L1	4	4	1	1	128	32
S4M1D2L1	4	1	2	1	128	32
S4M1D3L1	4	1	3	1	128	32
S4M1D4L1	4	1	4	1	128	32
S4M1D1L2 MB	4	1	1	2	128	64
S4M1D1L2 DP	4	1	1	2	128	64

Tabelle 8.1: Untersuchte Prozessorkonfigurationen

das Synthesewerkzeug sind in dem ungünstigsten Fall (Worst-Case Szenario) mit einer Versorgungsspannung von 1,0 V und einer Betriebstemperatur von 125 °C gegeben. Es werden jeweils Prozessorkerne inklusive Daten- und Instruktionsspeicher sowie einer Netzwerkschnittstelle synthetisiert (siehe Abbildung 3.7).

Da die Anzahl und Funktionsweise der oben genannten Prozessorelemente zur Entwurfszeit einstellbar ist, können die Synthesen der folgenden Untersuchungen weitgehend automatisiert erfolgen. Hierzu liest ein Skript eine Liste mit den geforderten Prozessorkonfigurationen ein, beschreibt die Variablen zur Steuerung der generischen Komponenten, und startet eine Synthese. Nach dem erfolgreichen Durchlauf der Synthesen werden die jeweiligen Ergebnisse durch ein weiteres Skript aus den Ergebnisdateien extrahiert und weiterverarbeitet. Die Ermittlung der Leistungsaufnahme erfolgt in anschließenden Arbeitsschritten mit Hilfe des Programms Cadence Encounter Power System (siehe Kapitel 4.2.3).

Die konstanten Konfigurationselemente werden wie folgt vorgegeben:

- Maximal vier Verarbeitungseinheiten und zwei LD/ST-Einheiten.
- Daten- und Instruktionsspeicher werden als reine on-chip Scratchpad-Speicher implementiert.
- Die Größe des Daten- und Instruktionsspeichers beträgt jeweils 16 KByte.
- Die Breite des Instruktionsspeichers beträgt 64 Bit (1, 2 Slot), 128 Bit (3, 4 Slot) oder 256 Bit (5 bis 8 Slot).
- Die Breite des Datenspeichers ist 32 Bit (1 LD/ST) oder 64 Bit (2 LD/ST).
- Der Kontroll- und Condition-Register-Bypass zwischen der Execute- und der Instruction-Decode-Stufe ist deaktiviert (siehe [90]).
- Das Registerfile besteht aus 32 Registern mit einer Breite von jeweils 32 Bit.
- Die Condition-Register werden aus zwei 8-Bit Registern gebildet.
- Der Prozessor besitzt eine Trace-Einheit (siehe Kapitel 4.2.1) und eine Netzwerk- und UART-Schnittstelle.

8.1 Analyse des kritischen Pfades

Wie bereits in Kapitel 4.2.2 erläutert wurde, ist die maximale Betriebsfrequenz des Prozessors direkt an die Latenz des kritischen Pfades gebunden. Die Latenz dieses Signalpfades kann bestimmt werden, indem das zuvor beschriebene Syntheseskript um eine Rückführung von Zwischenergebnissen erweitert wird. Das Skript beginnt mit einer Synthese bei einer vorgegebenen Periodendauer und prüft anschließend, ob diese Vorgabe eingehalten wurde. Konnte die Periodendauer eingehalten werden, verkürzt das Skript die Vorgabe um einen gewissen Prozentsatz und startet eine erneute Synthese. Falls die vorgegebene Periodendauer nicht eingehalten werden konnte, wird eine Synthese mit entsprechend größerer Zeitvorgabe gestartet. Auf diese Weise nähert sich das Skript von beiden Seiten an die Latenz des kritischen Pfades an. Nach dem Beenden des Syntheseskripts lässt sich neben der Latenz des kritischen Pfades auch sein konkreter Verlauf bestimmen, indem die entsprechenden Ausgabedateien analysiert werden. Da das Syntheseprogramm in einem nachgelagerten Optimierungsschritt jedoch sämtliche Pfade an die Latenz des kritischen Pfades anpasst, kann der Verlauf des kritischen Pfades nur bestimmt werden, indem Synthesen betrachtet werden, bei denen die Zeitvorgaben nicht eingehalten werden konnten.

Aufgrund des nicht deterministischen Verhaltens des Synthesewerkzeugs und seiner inkrementellen Optimierungsschritte verändert sich der Verlauf des kritischen Pfades bereits bei kleinsten Änderungen der Prozessorkonfiguration. Die Auswertung der Syntheseergebnisse des CoreVA-Prozessors zeigt jedoch die eindeutige Tendenz, dass der kritische Pfad bis auf wenige Ausreißer stets durch die Execute-Stufe verläuft. Der Pfad beginnt in diesen Fällen in einem der Pipelineregister zwischen der Register-Read- und Execute-Stufe (z.B. im RD-EX-Operandenregister oder RD-EX-Instruktionswortregister) und verläuft anschließend auf unterschiedlichen Wegen durch die verschiedenen Verarbeitungs- und Funktionseinheiten (z.B. ALU, Multiplizierer oder Dividierer). Der Pfad endet schließlich am Adresseingang des Datenspeichers oder in der Kontrolllogik, die signalisiert, dass der Speicher aufgrund einer Prozessoranfrage beschäftigt ist.

Erfahrungen bei der Analyse von Syntheseergebnissen zeigen, dass dem konkreten Verlauf des kritischen Pfades aufgrund der zuvor beschriebenen Streuung nicht zu viel Beachtung geschenkt werden sollte. Es sollte vielmehr betrachtet werden, ob sich die Latenz des kritischen Pfades und somit die spätere maximale Betriebsfrequenz durch einzelne Konfigurationselemente maßgeblich verändert. Wie der Graph 8.1 zeigt, weisen die Konfigurationen S1M1D1L1, S2M1D1L1, S4M3D1L1, S4M4D1L1, S4M1D3L1 und S4M1D4L1 mit 833 MHz jeweils die gleiche maximale Betriebsfrequenz auf. Die Betriebsfrequenzen der Konfigurationen S3M1D1L1, S4M1D1L1, S4M2D1L1 und S4M1D2L1 liegen mit 820 MHz und 826 MHz nur

knapp unter dem Wert der vorgenannten Konfigurationen. Es lässt sich dementsprechend feststellen, dass sich die maximale Betriebsfrequenz durch das Hinzufügen zusätzlicher Verarbeitungs- und Funktionseinheiten trotz der hinzukommenden Querverbindungen und Bypasspfade praktisch nicht verändert (Verringerung < 2%). Deutlich gravierendere Änderungen ergeben sich durch das Hinzufügen einer zweiten LD/ST-Einheit und durch die Variation der Speichertypen. Die maximale Betriebsfrequenz der Konfiguration mit zwei LD/ST-Einheiten und Dual-Port Speicher liegt bei 813 MHz. Durch den Einsatz des Multi-Bank Speichers sinkt die maximale Betriebsfrequenz sogar auf 760 MHz. Dieser deutliche Rückgang entsteht, da am Adresseingang dieses Speichertyps deutlich mehr Logik durchlaufen werden muss, um die Speicheradresse zu analysieren und den passenden Speicherblock zu aktivieren. Um die Einflüsse des kritischen Pfades auf die Ermittlung der Ressourcenanforderungen zu minimieren, werden die folgenden Untersuchungen jeweils bei einer vorgegebenen Periodendauer von 1,33 ns durchgeführt, was einer Betriebsfrequenz von 750 MHz entspricht.

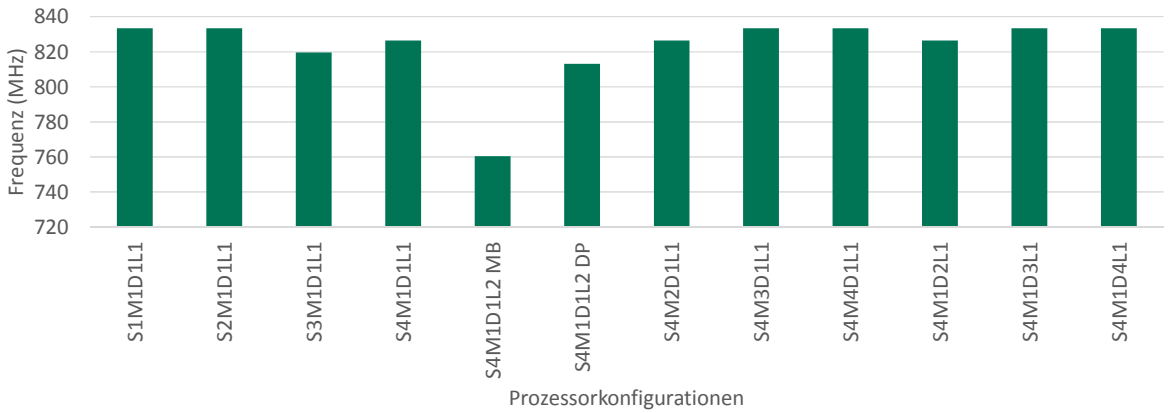

Abbildung 8.1: Maximale Betriebsfrequenz des CoreVA-Prozessors bei unterschiedlichen Prozessorkonfigurationen

8.2 Ermittlung der durchschnittlichen Schaltaktivitäten

In der Arbeit von Meixner [54] wurde gezeigt, dass die von den Synthesewerkzeugen standardmäßig genutzten Verfahren zur Bestimmung der Leistungsaufnahme mit sehr hohen Fehlern behaftet sind. Um bei der Bewertung der verschiedenen Prozessorkonfigurationen auf möglichst realistische Werte zurückgreifen zu können, muss das tatsächliche Verhalten des Prozessors bei der Ausführung der Zielanwendungen berücksichtigt werden. Hierzu kommen Hardwaresimulationen zum Einsatz, die

die Schaltaktivitäten der einzelnen Standardzellen bei der Anwendungsverarbeitung aufzeichnen. Auf Basis dieser Schaltaktivitäten kann das Programm Cadence Encounter Power System anschießend deutlich akkuratere anwendungsspezifische Leistungsaufnahmen ermitteln (siehe Kapitel 4.2.3).

Da das Hardwaremodell jedoch keine Anwendungsabhängigkeiten aufweisen soll und das simulationsbasierte Aufzeichnen der Schaltaktivitäten sehr zeitaufwändig ist, werden im Folgenden durchschnittliche Schaltaktivitäten bestimmt. Weil sich die Durchschnittswerte jedoch nicht direkt aus den aufgezeichneten Werten ableiten lassen, erfolgt ihre Ermittlung durch exemplarische Vergleiche verschiedener Leistungsaufnahmen. Hierzu werden die Leistungsaufnahmen, die sich aus durchschnittlichen Schaltaktivitäten ergeben, den Leistungsaufnahmen gegenübergestellt, die auf Basis tatsächlicher Schaltaktivitäten gewonnen wurden. Anschließend werden die mittleren relativen Fehler ausgewertet, die durch unterschiedliche Durchschnittswerte entstehen (siehe Graph 8.2 und 8.3). Die Referenz für die Fehlererhebungen bilden jeweils die anwendungsspezifischen Leistungsaufnahmen aller in Kapitel 6 betrachteten Algorithmen. Hierbei ist noch zu erwähnen, dass die Schaltaktivitäten nur während der Verarbeitung der eigentlichen Kernalgorithmen aufgezeichnet werden.

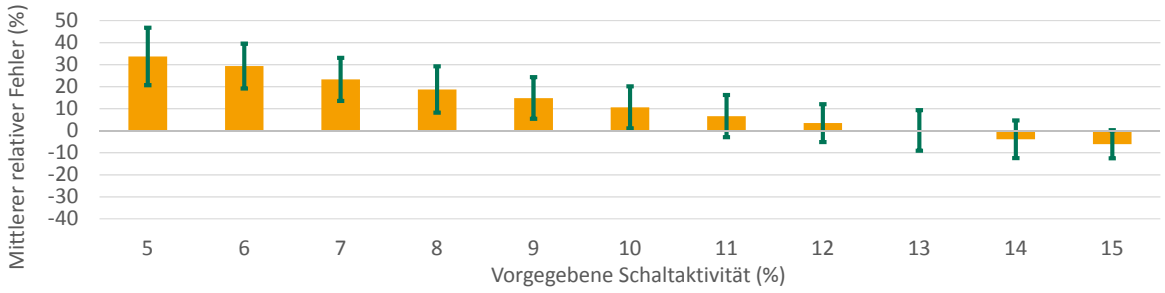

Abbildung 8.2: Mittlerer relativer Fehler und Standardabweichung der Leistungsaufnahme mit unterschiedlichen Schaltaktivitäten, S1M1D1L1

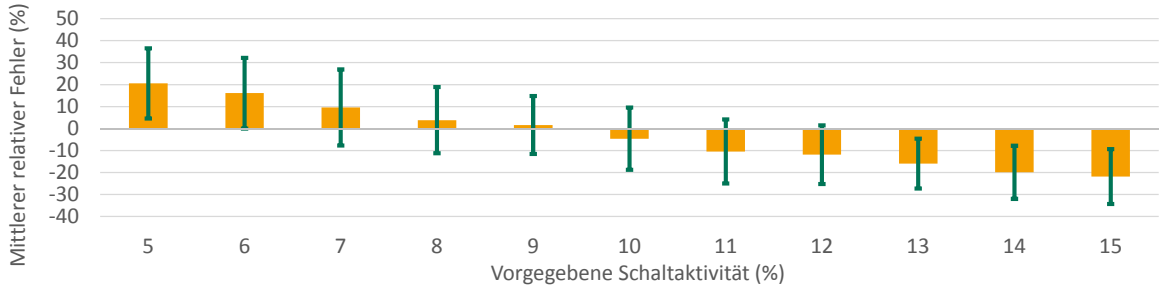

Abbildung 8.3: Mittlerer relativer Fehler und Standardabweichung der Leistungsaufnahme mit unterschiedlichen Schaltaktivitäten, S4M4D4L2 DP

Es zeigt sich, dass die Leistungsaufnahme der Konfiguration S1M1D1L1 den kleinsten mittleren Fehler bei einer vorgegebenen durchschnittlichen Schaltaktivität von 13% aufweist. Die Prozessorkonfiguration S4M4D4L2 DP zeigt hingegen den geringsten Fehler bei einer durchschnittlichen Schaltaktivität von 9%. Dieses Verhalten lässt sich dadurch erklären, dass die höherwertigen VLIW-Slots durch die begrenzte Parallelität der Zielanwendungen häufig nicht ausgelastet sind und dementsprechend deutlich geringere dynamische Verluste generieren. Auf Basis dieser Untersuchung lassen sich die durchschnittlichen Schaltaktivitäten schließlich in Abhängigkeit von der Anzahl der implementierten Verarbeitungseinheiten beschreiben (siehe Tabelle 8.2).

Slots	Schaltaktivität (%)
1	13
2	11,66
3	10,33
4	9

Tabelle 8.2: Durchschnittliche Schaltaktivität in Abhängigkeit von der Anzahl der Verarbeitungseinheiten

8.3 Ressourcenanforderungen zusätzlicher Verarbeitungs- und Funktionseinheiten

Die folgenden Untersuchungen beschreiben die Einflüsse der in Tabelle 8.1 genannten Konfigurationselemente auf die Ressourcenanforderungen des Prozessorkerns. Die Prozessorflächen werden hierbei den jeweiligen Syntheseergebnissen des CoreVA-Prozessors entnommen. Die Leistungsaufnahmen entstammen einer nachträglichen Auswertung der Gatternetzlisten unter Berücksichtigung der durchschnittlichen Schaltaktivitäten in Tabelle 8.2. Die vorgegebene Periodendauer beträgt 1,33 ns, die SIMD-Unterstützung und das Resource-Sharing sind deaktiviert.

Wie in den Abbildungen 8.4 bis 8.7 zu sehen ist, werden die Flächen und Leistungsaufnahmen der verschiedenen Prozessorkonfigurationen getrennt nach den Werten des Instruktions- und Datenspeichers und nach den Hauptkomponenten der Prozessorpipeline aufgetragen. Dies geschieht, indem dem Synthesewerkzeug das Aufbrechen der Hierarchieebenen für diese Komponenten verboten wird. Da die Register-Read- und Register-Write-Stufe sowie die verschiedenen Pipelineregister und Bypasspfade einzeln betrachtet eine vergleichsweise geringe Fläche aufweisen, ist der Synthese das Aufbrechen dieser Hierarchieebenen erlaubt. Die Anteile dieser Schaltungselemente sind dementsprechend in der Sektion „Andere“ zusammengefasst.

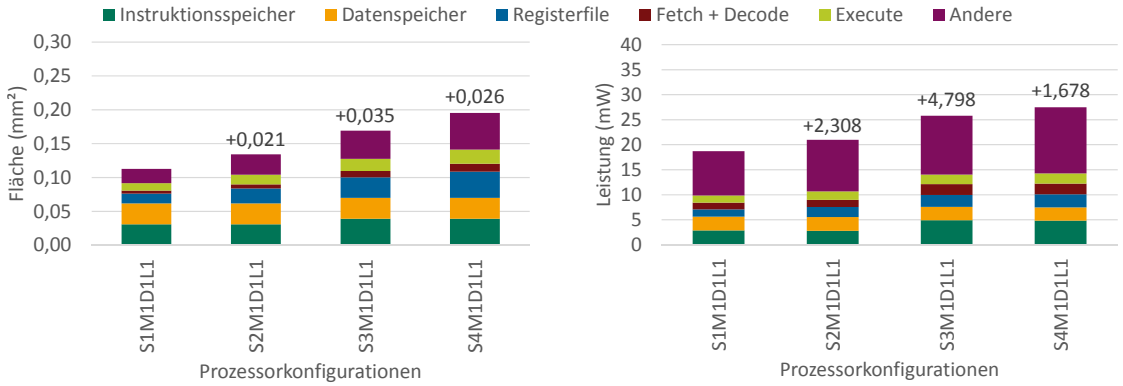

Abbildung 8.4: Flächen- und Leistungsanstieg für zusätzliche Verarbeitungseinheiten

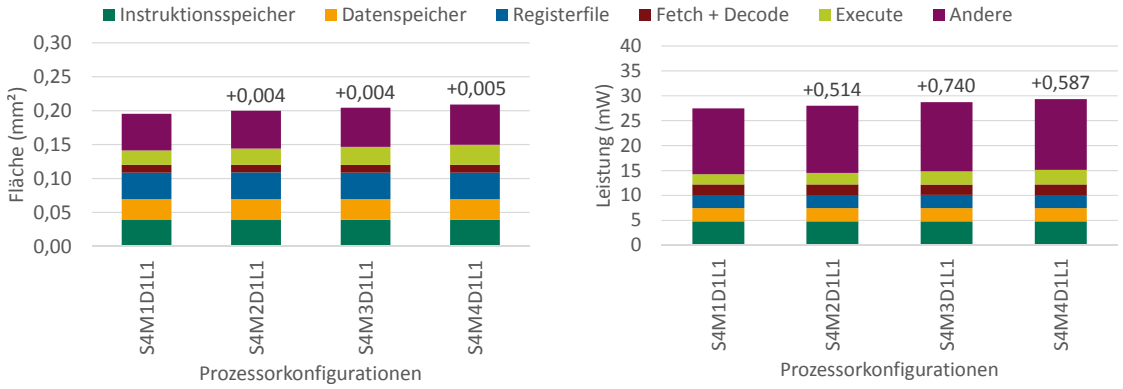

Abbildung 8.5: Flächen- und Leistungsanstieg für zusätzliche Multiplizierereinheiten

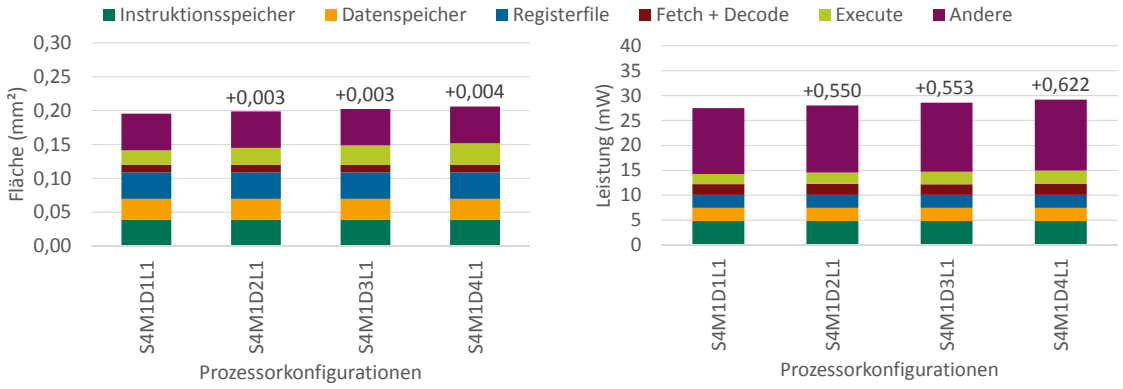

Abbildung 8.6: Flächen- und Leistungsanstieg für zusätzliche Dividiereinheiten

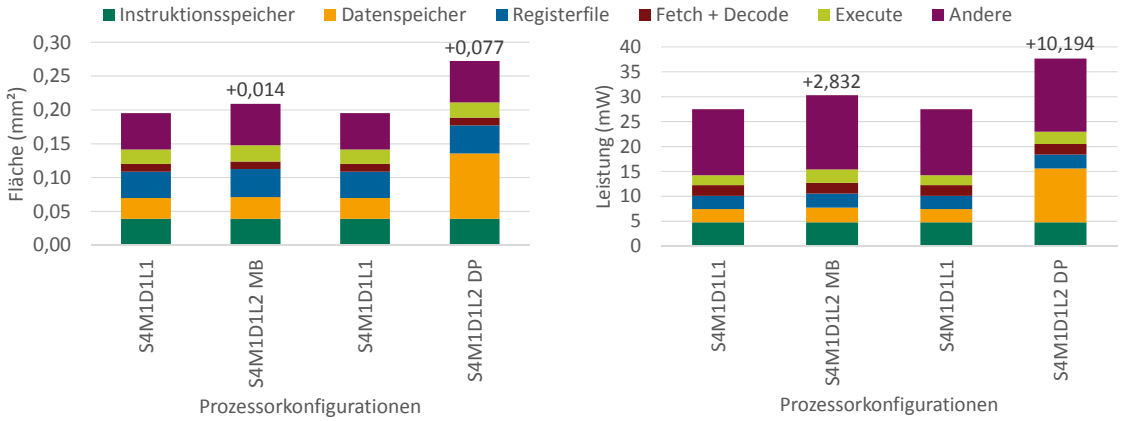

Abbildung 8.7: Flächen- und Leistungsanstieg für zusätzliche LD/ST-Einheiten

Die Konfigurationen S1M1D1L1 bis S4M1D1L1 zeigen den Mehraufwand für zusätzliche VLIW-Slots. Die RISC-ähnliche 1-Slot Konfiguration weist eine Fläche von 0,113 mm^2 und eine Leistungsaufnahme von 18,700 mW auf. Wie in der Abbildung 8.4 zu sehen ist, steigt die Prozessorfläche durch das Hinzufügen einer zweiten Verarbeitungseinheit um 0,021 mm^2. Durch das Hinzufügen einer dritten und vierten Verarbeitungseinheit steigt die Fläche im Vergleich zur jeweils vorangegangenen Konfiguration um 0,035 mm^2 und 0,026 mm^2. Die Leistungsaufnahme steigt um 2,308 mW, 4,798 mW und 1,678 mW. Diese Anstiege erklären sich hauptsächlich durch das Hinzufügen zusätzlicher Recheneinheiten in die Execute-Stufe, durch das Einfügen weiterer Schreib- und Leseports in das Registerfile und durch die automatische Verbreiterung der verbleibenden Pipelinestufen. Zwischen den Konfigurationen S2M1D1L1 und S3M1D1L1 zeigt sich jedoch auch eine deutliche Erhöhung der Ressourcenanforderungen des Instruktionsspeichers und der Instruction-Fetch-Stufe. Dieser Anstieg rührt daher, dass die Breite des Instruktionsspeichers aufgrund der derzeitigen Implementation der Speicherschnittstelle an dieser Stelle von 64 Bit auf 128 Bit springt (siehe Tabelle 8.1). Aus diesem Sprung resultiert nicht nur eine Verdoppelung des Alignment-Registers und seiner zugehörigen Funktionalitäten, sondern auch eine Verdoppelung der benötigten Leseports. Aus diesem Grund kann der Instruktionsspeicher in den Konfigurationen S3M1D1L1 und S4M1D1L1 nicht mehr wie gehabt aus zwei 2048x32-Bit Speicherblöcken aufgebaut werden, sondern muss mit vier 1024x32-Bit Blöcken realisiert werden, die eine geringere Ressourceneffizienz aufweisen.

Die Abbildungen 8.5 und 8.6 veranschaulichen den Mehraufwand zusätzlicher Multiplizier- und Dividiereinheiten auf Basis einer Grundkonfiguration mit vier Verarbeitungseinheiten (S4M1D1L1). Das Hinzufügen einer zweiten, dritten und vierten Multiplizierereinheit erhöht die Prozessorfläche im Vergleich zum jeweiligen Vorgänger um 0,004 mm^2, 0,004 mm^2 und 0,005 mm^2. Die Leistungsaufnahme steigt entsprechend um 0,514 mW, 0,740 mW und 0,587 mW. Die Mehraufwände für zusätz-

liche Dividiereinheiten liegen mit 0,003 mm^2, 0,003 mm^2, 0,004 mm^2 und 0,550 mW, 0,553 mW, 0,662 mW in einer ähnlichen Größenordnung. Diese Erhöhungen resultieren aus einer allgemeinen Vergrößerung der Execute-Stufe und aus dem Hinzufügen weiterer Registerfile-Leseports, um den Multiply-Accumulate-Operationen drei Operanden zur Verfügung stellen zu können.

Die in der Abbildung 8.7 dargestellten Vergleiche zeigen schließlich sowohl die Mehraufwände für eine zweite LD/ST-Einheit, als auch die Einflüsse der verschiedenen Speicherimplementierungen. Da in Konfigurationen mit einer LD/ST-Einheit und in Konfigurationen mit zwei LD/ST-Einheiten und Multi-Bank Speicher die gleichen Speicherblöcke verwendet werden (2 x 2048 x 32 Bit, low-leakage Speicher) und der Mehraufwand der Arbitrationslogik zur Vermeidung von Zugriffskonflikten bei den Multi-Bank Speichern vernachlässigbar klein ist, zeigt der Vergleich der Konfigurationen S4M1D1L1 und S4M1D1L2 MB ausschließlich den Mehraufwand der zweiten LD/ST-Einheit. Der Flächenanstieg für eine zusätzliche LD/ST-Einheit liegt demnach bei 0,014 mm^2, die Leistungsaufnahme steigt um 2,832 mW.

Der Vergleich zwischen den Konfigurationen S4M1D1L1 und S4M1D1L2 DP bestimmt hingegen sowohl den Mehraufwand der zweiten LD/ST-Einheit, als auch die Einflüsse des Dual-Port Speichers. Diese deutlich höheren Ressourcenanforderungen rühren daher, dass der Dual-Port Speicher derzeit nur durch einen auf Geschwindigkeit optimierten Speichertyp (4096 x 32 Bit, high-performance Speicher) realisiert werden kann. Des Weiteren weist der Dual-Port Speicher im Vergleich zum Multi-Bank Speicher einen deutlich höheren Schaltungsbedarf für den zweiten Lese- und Schreibport und die Arbitrationslogik auf. Aus der Kombination dieser Einflüsse ergibt sich für die Konfiguration mit Dual-Port Speicher ein Flächenanstieg von 0,077 mm^2 und eine Steigerung der Leistungsaufnahme von 10,194 mW.

8.4 Ressourcenanforderungen des Resource-Sharings und der SIMD-Unterstützung

Wie bereits in Kapitel 3.11 erläutert wurde, ermöglicht das Resource-Sharing das gemeinsame jedoch nicht gleichzeitige Verwenden von MAC- und DIV-Einheiten in mehreren VLIW-Slots. Im Unterschied zu den vorangegangenen Untersuchungen, wird der Mehraufwand für das Aktivieren des Resource-Sharings nicht durch Vergleiche verschiedener Prozessorkonfigurationen bestimmt, sondern indem die Ressourcenanforderungen gleicher Konfigurationen bei aktiviertem und deaktiviertem Resource-Sharing gegenübergestellt werden (siehe Abbildung 8.8 und 8.9).

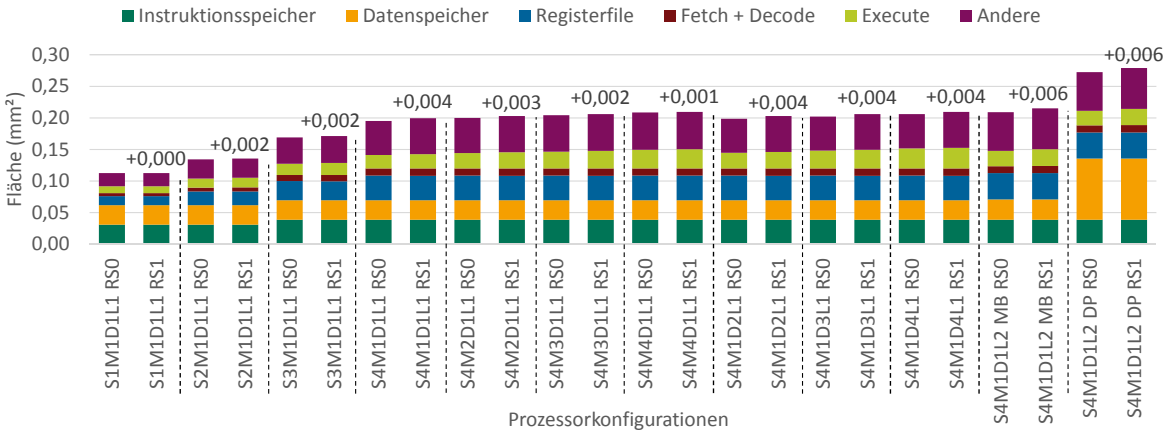

Abbildung 8.8: Fläche der verschiedenen Prozessorkonfigurationen mit deaktiviertem und aktiviertem Resource-Sharing

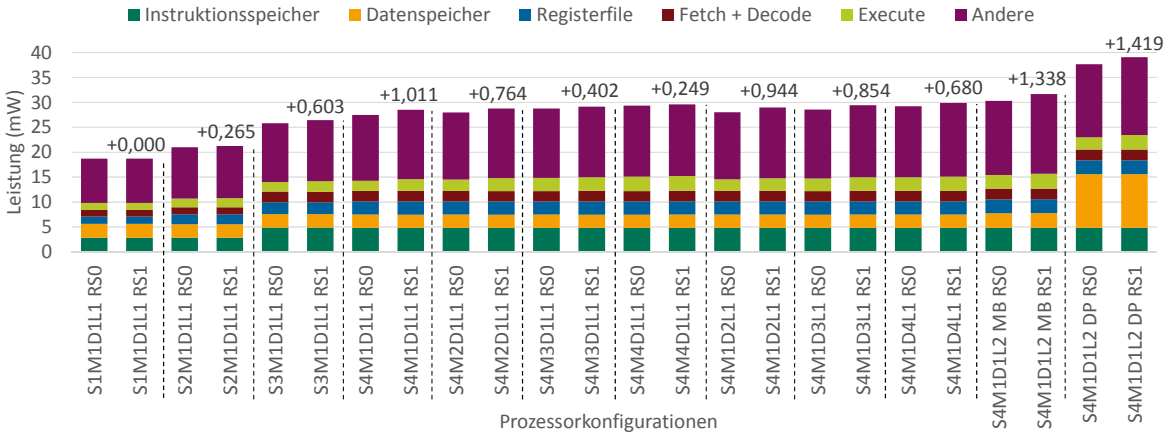

Abbildung 8.9: Leistungsaufnahme der verschiedenen Prozessorkonfigurationen mit deaktiviertem und aktiviertem Resource-Sharing

Es zeigt sich, dass sich die Fläche und Leistungsaufnahme des CoreVA-Prozessors durch das Aktivieren des Resource-Sharings um bis zu 0,006 mm^2 beziehungsweise 1,419 mW erhöht. Dieser Anstieg resultiert hauptsächlich aus zusätzlichen Kontrolllogiken am Eingang und Ausgang der Execute-Stufe, die benötigt werden, um die Operanden, Steuersignale und Rechenergebnisse zwischen den Verarbeitungseinheiten zu verschieben. Des Weiteren muss die Register-Read-Stufe insofern erweitert werden, dass alle VLIW-Slots einen dritten Operanden für MAC-Instruktionen bereitstellen können.

Eine detailliertere Betrachtung der einzelnen Vergleiche zeigt, dass der Mehraufwand für Konfigurationen, bei denen wenige MAC- beziehungsweise DIV-Einheiten vielen Verarbeitungseinheiten zugängig gemacht werden müssen, am höchsten ist. Aus diesem Grund steigen die Fläche und Leistungsaufnahme von Systemen mit je-

weils einer MAC- und DIV-Einheit und zwei, drei oder vier VLIW-Slots (S2M1D1L1 bis S4M1D1L1) um 0,002 mm^2, 0,002 mm^2, 0,004 mm^2 und 0,265 mW, 0,603 mW, 1,011 mW. Im Umkehrschluss zeigt die Betrachtung der Konfigurationen S4M2D1L1 bis S4M4D1L1 beziehungsweise S4M1D2L1 bis S4M1D4L1, dass der Mehraufwand für das Resource-Sharing in einem Prozessor mit vier Verarbeitungseinheiten durch das Hinzufügen zusätzlicher Multiplizier- und Dividiereinheiten wieder sinkt. Da in den untersuchten Konfigurationen jedoch entweder nur die Anzahl der MAC- oder der DIV-Einheiten angehoben wird, bleibt der Mehraufwand der jeweils anderen Funktionseinheit bestehen. Aus dem Anstieg der Ressourcenanforderungen der Konfiguration S4M4D1L1 lässt sich dementsprechend der Aufwand des Resource-Sharings für eine DIV-Einheit auf vier Verarbeitungseinheiten bestimmen. Er beträgt in diesem Fall 0,001 mm^2 und 0,249 mW. Auf der anderen Seite weist das Resource-Sharing einer MAC-Einheit auf vier Verarbeitungseinheiten wegen der höheren Anzahl zu verteilender Operanden einen Mehraufwand von 0,004 mm^2 und 0,680 mW (S4M1D4L1) auf.

Die in den Abbildungen 8.10 und 8.11 gezeigte Gegenüberstellung der Ressourcenanforderungen von Prozessorkonfigurationen mit aktivierter und deaktivierter SIMD-Unterstützung zeigt schließlich, dass der Mehraufwand für das Aktivieren der SIMD-Funktionalität um ein Vielfaches über dem Aufwand des Resource-Sharings liegt.

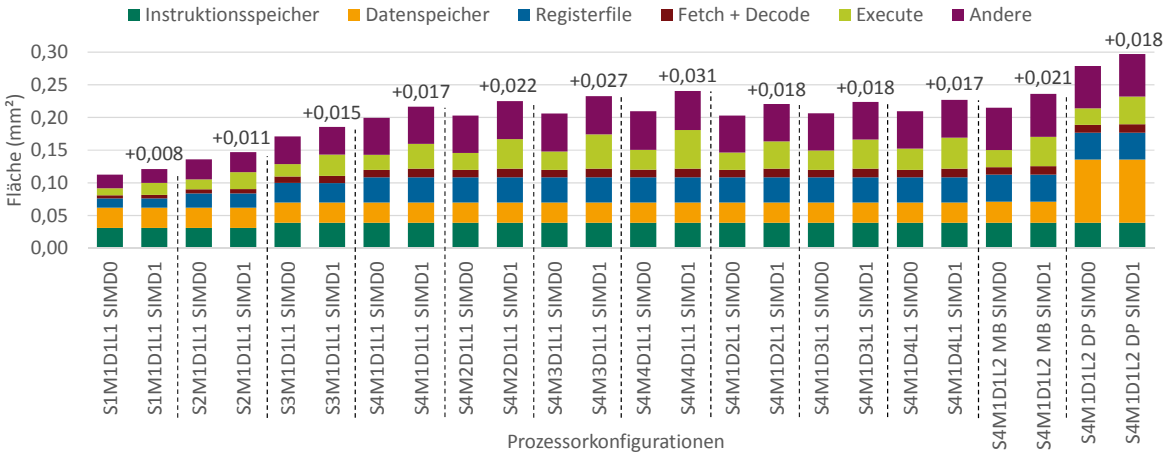

Abbildung 8.10: Fläche der verschiedenen Prozessorkonfigurationen mit deaktivierter und aktivierter SIMD-Unterstützung

Der Mehraufwand resultiert zum Einen aus einer deutlichen Vergrößerung der Execute-Stufe (S4M4D1L1: Fläche +94%, Leistung +40%), da durch das Aktivieren der SIMD-Funktionalität jeweils zwei 16-Bit ALUs pro Verarbeitungseinheit und zwei 16-Bit Multiplizierern pro MAC-Funktionseinheit hinzugefügt werden. Zum Anderen steigen die Ressourcenanforderungen der Instruction-Fetch- und Instruction-Decode-Stufe an, da dreizehn zusätzliche SIMD-Instruktionen dekodiert

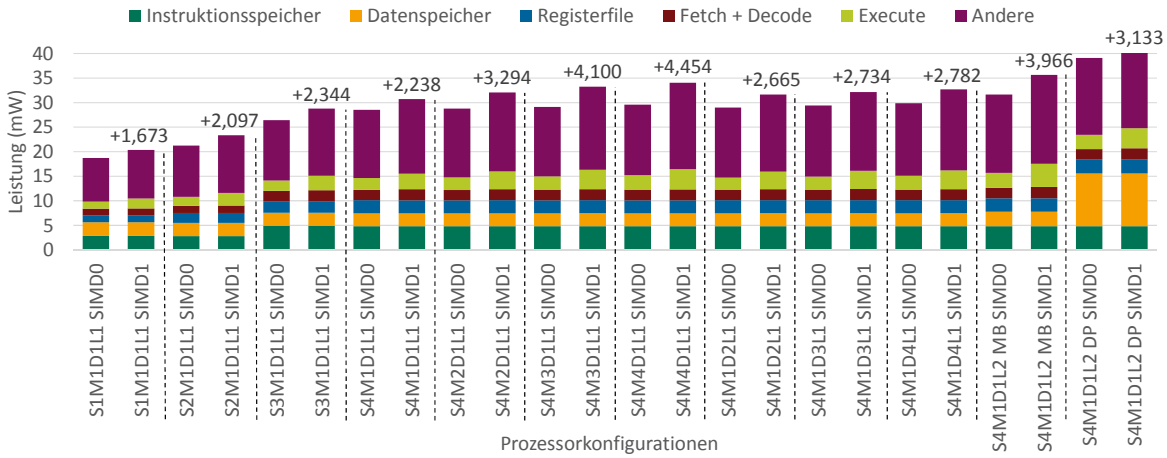

Abbildung 8.11: Leistungsaufnahme der verschiedenen Prozessorkonfigurationen mit deaktivierter und aktivierter SIMD-Unterstützung

werden müssen (siehe Kapitel 3.6). Da das Registerfile für das bedingte Ausführen einzelner SIMD-Instruktionsteile aufgrund der geringen zusätzlichen Ressourcenanforderungen auch bei deaktivierter SIMD-Unterstützung durch jeweils zwei 16-Bit Register mit eigenem Write-Enable-Anschluss realisiert ist, hat die SIMD-Funktionalität für die verbleibenden Pipelinestufen keine Auswirkungen (siehe Kapitel 3.4).

8.5 Modellbeschreibung des Prozessors

Aus den Erkenntnissen der vorangegangenen Auswertungen lässt sich nun ein Hardwaremodell formulieren, das die Fläche und Leistungsaufnahme eines Prozessors bei einer vorgegebenen Anzahl von Verarbeitungs- und Funktionseinheiten approximiert. Das hier entwickelte Hardwaremodell basiert auf der Vorgehensweise von Ascia [7], bei der die Prozessorfläche und Leistungsaufnahme bestimmt wird, indem zu den Ressourcenanforderungen eines Prozessors mit minimalem Funktionsumfang die Werte der konfigurationsabhängigen Funktionsblöcke hinzu addiert werden. Wie in Formel 8.1 ersichtlich ist, beginnt auch das Hardwaremodell des CoreVA-Prozessors mit einen konfigurationsunabhängigen Startwert (S1M1D1L1 RS0 SIMD0) und addiert für jede zusätzliche Verarbeitungseinheit und jede zusätzliche MAC-, DIV- und LD/ST-Einheit einen relativen Mehraufwand hinzu. Es folgen die Einflüsse des Resource-Sharings und der SIMD-Funktionalität, die ebenfalls über einen konstanten Startwert und konfigurationsspezifische Mehraufwände beschrieben werden.

$$
\begin{aligned}
Fläche_{Approx}/Leistung_{Approx} = {} & Startwert_{S1M1D1L1\ RS0\ SIMD0} \\
& + Mehraufwand_{Slots} \cdot (Slots - 1) \\
& + Mehraufwand_{MAC} \cdot (MAC - 1) \\
& + Mehraufwand_{DIV} \cdot (DIV - 1) \\
& + Mehraufwand_{LD/ST} \cdot (LD/ST - 1) \\
& + Startwert_{RS} \cdot RS \\
& + Mehraufwand_{Slots_{RS}} \cdot (Slots - 1) \cdot RS \\
& + Mehraufwand_{MAC_{RS}} \cdot (MAC - 1) \cdot RS \\
& + Mehraufwand_{DIV_{RS}} \cdot (DIV - 1) \cdot RS \\
& + Mehraufwand_{LD/ST_{RS}} \cdot (LD/ST - 1) \cdot RS \\
& + Startwert_{SIMD} \cdot SIMD \\
& + Mehraufwand_{Slots_{SIMD}} \cdot (Slots - 1) \cdot SIMD \\
& + Mehraufwand_{MAC_{SIMD}} \cdot (MAC - 1) \cdot SIMD \\
& + Mehraufwand_{DIV_{SIMD}} \cdot (DIV - 1) \cdot SIMD \\
& + Mehraufwand_{LD/ST_{SIMD}} \cdot (LD/ST - 1) \cdot SIMD
\end{aligned}
\tag{8.1}
$$

Um die Anzahl der Synthesen auf ein Minimum zu beschränken, werden bei der Ermittlung der benötigten Werte nur noch die Randbereiche der vorangegangenen Untersuchungen berücksichtigt. Die Mehraufwände für das Einfügen zusätzlicher Verarbeitungseinheiten errechnen sich dementsprechend aus der Differenz der Ressourcenanforderungen zwischen der 4-Slot und der 1-Slot Konfiguration geteilt durch die Anzahl der zusätzlichen Slots (siehe Formel 8.2). Die Mehraufwände der zusätzlichen Funktionseinheiten werden, wie auch schon in den Abbildungen 8.5, 8.6 und 8.7, auf Basis einer Konfiguration mit vier Verarbeitungseinheiten bestimmt (siehe Formel 8.3).

Die Auswirkungen, die sich durch das Aktivieren des Resource-Sharings und der SIMD-Unterstützung ergeben, werden nach dem gleichen Ansatz berechnet. Zur Ermittlung der jeweiligen Startwerte und Mehraufwände kommen nun jedoch nicht mehr die absoluten Flächen- und Leistungswerte zum Einsatz, sondern die Differenzen zwischen den jeweiligen Konfigurationen mit und ohne aktivierter Funktionalität. Der Startwert des Resource-Sharings errechnet sich dementsprechend durch Subtraktion der Ressoucenaufwände der Konfigurationen S1M1D1L1 RS1 SIMD0 und S1M1D1L1 RS0 SIMD0 (siehe Formel 8.4). Der Mehraufwand für zusätzliche Verarbeitungseinheiten berechnet sich schließlich nach der Formel 8.5.

$$Mehraufwand_{Slots} = (S4M1D1L1\ RS0\ SIMD0 - S1M1D1L1\ RS0\ SIMD0)/3 \quad (8.2)$$

$$Mehraufwand_{MAC} = (S4M4D1L1\ RS0\ SIMD0 - S4M1D1L1\ RS0\ SIMD0)/3 \quad (8.3)$$

$$Startwert_{RS} = S1M1D1L1\ RS1\ SIMD0 - S1M1D1L1\ RS0\ SIMD0 \quad (8.4)$$

$$\begin{aligned} Mehraufwand_{Slots_{RS}} = ((&S4M1D1L1\ RS1\ SIMD0 - S4M1D1L1\ RS0\ SIMD0) \\ -(&S1M1D1L1\ RS1\ SIMD0 - S1M1D1L1\ RS0\ SIMD0))/3 \end{aligned} \quad (8.5)$$

Aus den oben genannten Formeln ergeben sich die in Tabelle 8.3 hinterlegten Werte. Wie zu erwarten war, zeigt das Aktivieren des Resource-Sharings für Prozessorkonfigurationen mit einer Verarbeitungseinheit keinerlei Mehraufwand, da es bei dieser Konfiguration nicht möglich ist, MAC- oder DIV-Zugriffe auf mehrere Slots zu verteilen. Der Startwert für die Modellierung des Resource-Sharings ist folglich 0. Für das Hinzufügen zusätzlicher MAC- und DIV-Einheiten ergibt sich bezüglich des Resource-Sharings ein negatives Ergebnis, da sich der Mehraufwand durch weitere Multiplizier- und Dividiereinheiten verringert (siehe Kapitel 8.4). Zusammenfassend lässt sich sagen, dass zum Aufstellen des Hardwaremodells einschließlich der Bewertung des Resource-Sharings und der SIMD-Unterstützung siebzehn Synthesen und siebzehn anwendungsunabhängige Analysen der Leistungsaufnahme durchgeführt werden müssen (siehe Tabelle 8.4). Mit Hilfe dieses Modells lassen sich dann für die Entwurfsraumexploration eines Prozessors mit bis zu vier Verarbeitungseinheiten die Ressourcenanforderungen von 352 unterschiedlichen Prozessorkonfigurationen approximieren. Falls ein Entwurfsraum mit acht VLIW-Slots untersucht werden soll, lassen sich sogar die Werte von 2440 unterschiedliche Konfigurationen ermitteln.

Die Überprüfung des Hardwaremodells erfolgt wiederum anhand eines Vergleichs der approximierten Prozessorflächen und Leistungsaufnahmen mit experimentell ermittelten Referenzwerten. Die Referenzwerte entstammen hierbei einer Stichprobe aus 34 Synthesen und 420 Hardwaresimulationen, die zur Ermittlung der anwendungsspezifischen Leistungsaufnahmen bei der Ausführung der zwölf Zielanwendungen benötigt werden. Bei der Überprüfung der Approximationen werden zuerst die bereits in Kapitel 7 untersuchten Prozessorkonfigurationen betrachtet.

Mehraufwand für zusätzliche Verarbeitungs- und Funktionseinheiten

	Fläche (mm^2)	Leistung (mW)
Startwert (S1M1D1L1)	0,113	18,700
Mehraufwand Slot	0,028	2,928
Mehraufwand MAC	0,005	0,614
Mehraufwand DIV	0,004	0,575
Mehraufwand LD/ST MB	0,014	2,832
Mehraufwand LD/ST DP	0,077	10,194

Mehraufwand für Resource-Sharing

	Fläche (mm^2)	Leistung (mW)
Startwert RS	0,000	0,000
Mehraufwand Slot	0,001	0,337
Mehraufwand MAC	- 0,001	-0,254
Mehraufwand DIV	- 0,000	-0,110
Mehraufwand LD/ST MB	0,002	0,327
Mehraufwand LD/ST DP	0,002	0,408

Mehraufwand für SIMD

	Fläche (mm^2)	Leistung (mW)
Startwert SIMD	0,008	1,673
Mehraufwand Slot	0,003	0,188
Mehraufwand MAC	0,005	0,739
Mehraufwand DIV	0,000	0,181
Mehraufwand LD/ST MB	0,004	1,728
Mehraufwand LD/ST DP	0,001	0,895

Tabelle 8.3: Startwerte und Mehraufwände zur Approximation der Fläche und Leistungsaufnahme

Name	Slots	MAC	DIV	LD/ST	RS	SIMD
S1M1D1L1 RS0 SIMD0	1	1	1	1	0	0
S4M1D1L1 RS0 SIMD0	4	1	1	1	0	0
S4M4D1L1 RS0 SIMD0	4	4	1	1	0	0
S4M1D4L1 RS0 SIMD0	4	1	4	1	0	0
S4M1D1L2 MB RS0 SIMD0	4	1	1	2 MB	0	0
S4M1D1L2 DP RS0 SIMD0	4	1	1	2 DP	0	0
S4M1D1L1 RS1 SIMD0	4	1	1	1	1	0
S4M4D1L1 RS1 SIMD0	4	4	1	1	1	0
S4M1D4L1 RS1 SIMD0	4	1	4	1	1	0
S4M1D1L2 MB RS1 SIMD0	4	1	1	2 MB	1	0
S4M1D1L2 DP RS1 SIMD0	4	1	1	2 DP	1	0
S1M1D1L1 RS0 SIMD1	1	1	1	1	0	1
S4M1D1L1 RS0 SIMD1	4	1	1	1	0	1
S4M4D1L1 RS0 SIMD1	4	4	1	1	0	1
S4M1D4L1 RS0 SIMD1	4	1	4	1	0	1
S4M1D1L2 MB RS0 SIMD1	4	1	1	2 MB	0	1
S4M1D1L2 DP RS0 SIMD1	4	1	1	2 DP	0	1

Tabelle 8.4: Zur Berechnung der Startwerte und Mehraufwände untersuchte Prozessorkonfigurationen

Darüber hinaus werden auch einzelne Prozessoren mit sechs oder acht Verarbeitungseinheiten und mehreren Dividiereinheiten synthetisiert, um die Skalierbarkeit des Modells zu überprüfen.

Es zeigt sich, dass die Approximation der Prozessorflächen einen mittleren Fehler von 0,11% und eine Standardabweichung von ±3,31% aufweist (siehe Abbildung 8.12). Dieser Fehler resultiert aus der Annahme, dass die Ressourcenanforderungen beim Hinzufügen zusätzlicher Verarbeitungseinheiten stets linear ansteigen. Da dies aufgrund der sprunghaften Verdoppelung der Instruktionsspeicherbreite (siehe Kapitel 8.3) und aufgrund des überproportionalen Anstiegs der benötigten Querverbindungen und Bypasspfade jedoch nicht gewährleistet ist, liegt die approximierte Fläche der Konfigurationen mit sechs und acht VLIW-Slots bis zu 10% unter den experimentell ermittelten Werten.

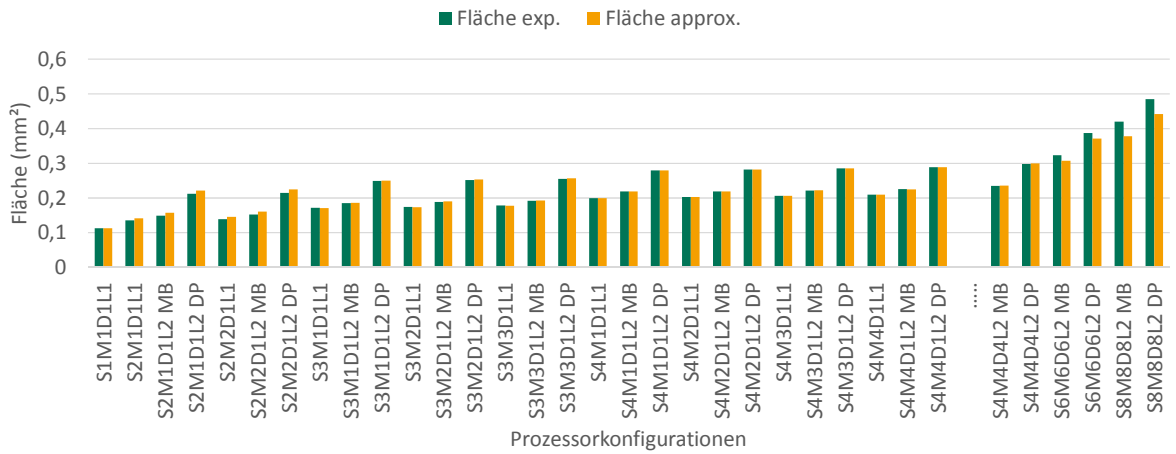

Abbildung 8.12: Vergleich der approximierten und experimentell ermittelten Prozessorflächen

Der Vergleich der approximierten und experimentell ermittelten Leistungsaufnahmen zeigt bei der Verarbeitung des FIR-Filters einen mittleren Fehler von -17,81% und eine Standardabweichung von ±8,97%. Dieser Fehler rührt daher, dass die Approximation auf den durchschnittlichen Schaltaktivitäten basiert, die in Kapitel 8.2 festgelegt wurden. Da der CoreVA-Prozessor bei der Ausführung des FIR-Filters jedoch eine überdurchschnittlich hohe Schaltaktivität aufweist, fällt die approximierte Leistungsaufnahme für alle Konfigurationen zu gering aus (siehe Abbildung 8.13).

Die in Abbildung 8.14 dargestellte Gegenüberstellung der Approximationsfehler der anderen Zielanwendungen ergibt schließlich, dass dieses Problem bei allen Approximationen auftritt. Abhängig von den Abweichungen der tatsächlichen Schaltaktivitäten weisen die Leistungsaufnahmen stets einen positiven oder negativen Fehler auf. Ein Vergleich der Abbildungen 7.6 und 8.14 zeigt weiterhin, dass zwischen der Parallelität der Zielanwendungen und den Fehlern der approximierten Leis-

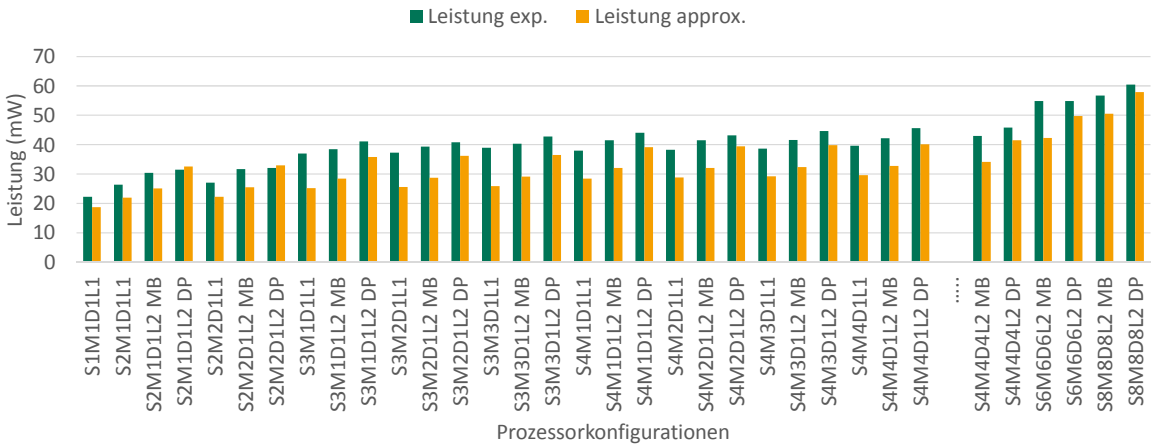

Abbildung 8.13: Vergleich der approximierten und experimentell ermittelten Leistungsaufnahme bei der Verarbeitung des FIR-Filters

tungsaufnahme eine gegenläufige Beziehung besteht. Algorithmen, die eine geringe Parallelität aufweisen, haben einen positiven Approximationsfehler (Coremark, Descrambler und Dhrystone). Algorithmen mit einer überdurchschnittlich hohen Parallelität haben einen negativen Fehler (AES, FFT, FIR, Korrelation, SATD und Viterbi). Dieses Verhalten lässt sich dadurch erklären, dass die hinteren VLIW-Slots bei einer geringen Parallelität nur sehr selten ausgelastet sind, was zu einer unterdurchschnittlichen Schaltaktivität und folglich zu einer unterdurchschnittlichen Leistungsaufnahme führt.

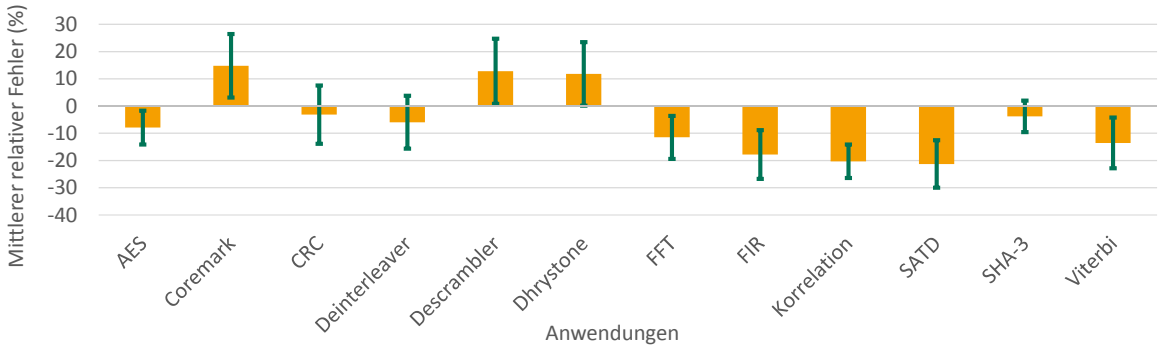

Abbildung 8.14: Mittlerer relativer Fehler und Standardabweichung der Approximation der Leistungsaufnahme

8.6 Zusammenfassung

In diesem Kapitel wurde die Entwicklung eines Hardwaremodells des CoreVA-Prozessors beschrieben. Das Hardwaremodell approximiert die Leistungsaufnahmen und die Chipflächen verschiedenster Prozessorkonfigurationen, indem es mit den konfigurationsunabhängigen Ressourcenanforderungen eines 1-Slot CoreVAs beginnt und für jeden zusätzlichen Slot und jede zusätzliche MAC-, DIV- oder LD/ST-Einheit einen relativen Mehraufwand hinzuaddiert.

Zum Aufstellen dieses Modells wurde eine Reihe von Probesynthesen durchgeführt, bei denen die einzelnen Konfigurationsparameter sukzessive variiert wurden. Zuerst wurden die Syntheseergebnisse von Konfigurationen mit ein bis vier Verarbeitungseinheiten, ein bis vier MAC- oder DIV-Einheiten und ein oder zwei LD/ST-Einheiten gegenübergestellt und die jeweiligen Mehraufwände ermittelt. In einer ähnlichen Weise wurde anschließend mit den Ressourcenanforderungen für das Aktivieren des Resource-Sharings und der SIMD-Funktionalität verfahren. Die präsentierten Ergebnisse basieren auf Standardzellensynthesen bei einer vorgegebenen Betriebsfrequenz von 750 MHz. Um ein anwendungsunabhängiges Modell zu erzeugen, wurden die Leistungsaufnahmen mit Hilfe vorher ermittelter durchschnittlicher Schaltaktivitäten bestimmt.

Die Güte des Hardwaremodells wurde wiederum durch einen Vergleich mit experimentell ermittelten Referenzwerten erfasst. Im Unterschied zu den vorangegangenen Untersuchungen wurden hierbei auch Konfigurationen mit bis zu acht Verarbeitungseinheiten einbezogen, um die Skalierbarkeit des Modells zu überprüfen. Die Approximation der Prozessorflächen wies hierbei einen mittleren relativen Fehler von 0,11% und eine Standardabweichung von ±3,31% auf. Die Gegenüberstellung der approximierten und experimentell ermittelten Leistungsaufnahmen zeigte schließlich einen durchschnittlichen Fehler von -5,49% (±15,33%).

9 Modellbasierte Entwurfsraumexploration des CoreVA-Prozessors

Mit Hilfe der in den vorangegangenen Kapiteln aufgestellten Hard- und Softwaremodelle können nun modellbasierte Entwurfsraumexplorationen durchgeführt werden, um ineffiziente Konfigurationen zu eliminieren und eine Vorauswahl für eine anschließende experimentelle Entwurfsraumexploration zu treffen.

Wie bereits in Kapitel 5.2 beschrieben wurde, gibt es hierbei verschiedene Herangehensweisen, um die Wechselwirkungen zwischen einer Steigerung der Leistungsfähigkeit und einer Erhöhung der Leistungsaufnahme bewerten zu können. So lässt sich die Konfigurationsmenge durch das Zusammenfassen der Bewertungskriterien oder durch das Aufstellen und Lösen eines Mehrzieloptimierungsproblems mittels Pareto-Optimierung einschränken. Eine Voraussetzung hierfür ist jedoch, dass simulationsbasierte Anwendungsanalyseverfahren genutzt werden, da nur diese in der Lage sind, die tatsächliche Anzahl der Prozessortakte zu bestimmen. Falls statische Analyseverfahren eingesetzt werden, können ineffiziente Konfigurationen in einer ersten Näherung auch durch das direkte Gegenüberstellen der Leistungssteigerung und des Mehraufwands gefunden werden.

Im Unterschied zu den Untersuchungen der verschiedenen Anwendungsanalyseverfahren in Kapitel 7 werden in den folgenden Auswertungen auch Prozessorkonfigurationen mit bis zu acht Verarbeitungseinheiten und mehreren Dividiereinheiten betrachtet. Des Weiteren wird in Konfigurationen mit zwei LD/ST-Einheiten nun auch der Typ des Datenspeichers variiert (Dual-Port oder Multi-Bank Speicher, siehe Kapitel 3.15). Obwohl diese Variationen bei dem Aufstellen der Softwaremodelle nicht explizit erwähnt wurden, ist der Instruktionssatzsimulator selbstverständlich in der Lage, die Wartezyklen, die durch Konflikte bei gleichzeitigen Zugriffen auf den Multi-Bank Speicher entstehen, zu detektieren und getrennt auszuwerten. Die approximationsbasierte Anwendungsanalyse kann somit weiterhin alle benötigten Informationen auf Basis eines einzelnen Simulationsdurchlaufs ermitteln. Die statische Programmcodeanalyse ist hingegen nicht in der Lage, die Einflüsse des

Multi-Bank Speichers vorherzusagen, da die Adressen der meisten Speicherzugriffe erst zur Laufzeit ermittelt werden. Aus diesem Grund werden bei dem in Kapitel 9.3 beschriebenen Verfahren nur Prozessorkonfigurationen mit Single- oder Dual-Port Speicher betrachtet.

9.1 Entwurfsraumexploration auf Basis der Energie

Ein Beispiel für eine modellbasierte Entwurfsraumexploration auf Basis eines einzelnen Bewertungskriteriums bildet die Ermittlung der Energie, die benötigt wird, um die Zielanwendungen auf den jeweiligen Prozessorkonfigurationen zu verarbeiten. Sie errechnet sich, indem die approximierte Leistungsaufnahme mit der Anzahl der erwarteten Prozessortakte multipliziert wird und durch die Betriebsfrequenz von 750 MHz dividiert wird (siehe Formel 5.5). Das Ergebnis dieser Berechnung könnte nun direkt als Bewertungsmaß für die Effizienz der Prozessorkonfigurationen genutzt werden. Um Fehlentscheidungen zu vermeiden, wird die approximierte Energie wie in der Arbeit von Mohanty [56] hier jedoch lediglich zur Einschränkung der Konfigurationsmenge genutzt, bevor die verbleibenden Konfigurationen in einem zweiten Schritt durch konfigurationsspezifische Synthesen und Hardwaresimulationen erneut untersucht werden.

Die in der Abbildung 9.1 dargestellten Gegenüberstellungen der approximierten und experimentell ermittelten Energiewerte zur Ausführung des FIR-Filters zeigt, dass sich die Fehler der beiden Modelle in einem gewissen Maße kompensieren. Im Vergleich zur Approximation der Leistungsaufnahme in Abbildung 8.13 hat sich der mittlere relative Fehler um die Hälfte verringert. Die Standardabweichung ist mit ±8,14% nahezu gleich geblieben. Dieses Verhalten trifft jedoch nicht für alle Zielanwendungen zu. Wie ein Vergleich der Abbildungen 9.2 und 8.14 zeigt, überwiegt bei einem Großteil der betrachteten Zielanwendungen die Ausprägung des teilweise sehr hohen Fehlers des Hardwaremodells, so dass sich die Approximationsfehler der Energie auf einem ähnlichen Niveau befinden. Aus diesem Grund wird im Folgenden anhand einer Suche nach der jeweils energieeffizientesten Prozessorkonfiguration überprüft, wie die approximierte Energie trotz ihrer Abweichungen zur Einschränkung der Konfigurationsmenge genutzt werden kann.

Um die Verlässlichkeit der modellbasierten Entwurfsraumexploration zu bestimmen, listet die Tabelle 9.1 die Prozessorkonfigurationen auf, die durch den experimentellen und den modellbasierten Ansatz ermittelt werden konnten. Falls die Konfigurationen nicht übereinstimmen und die modellbasierte Entwurfsraumexploration folglich zu einer Fehlentscheidung führen würde, wird zusätzlich die

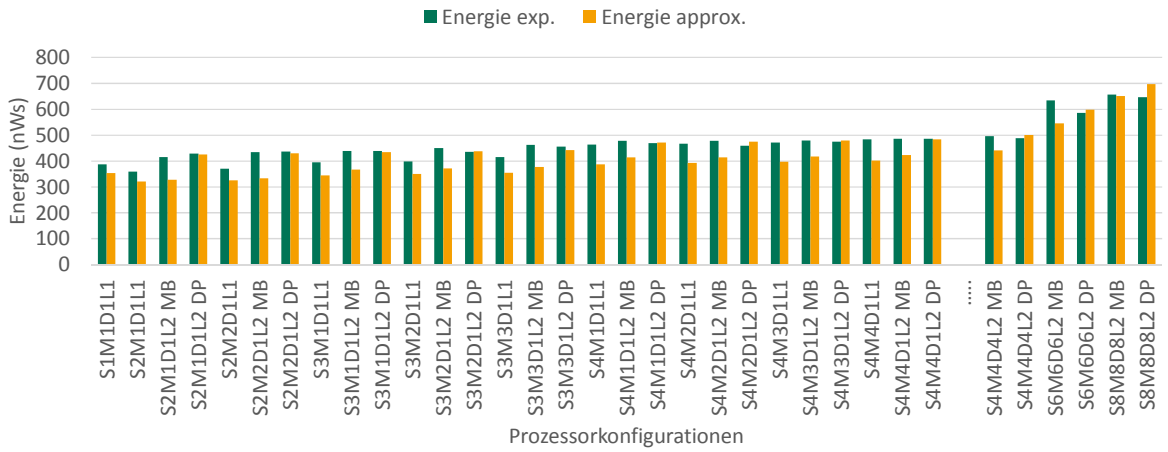

Abbildung 9.1: Vergleich der approximierten und experimentell ermittelten Energie für die Verarbeitung des FIR-Filters

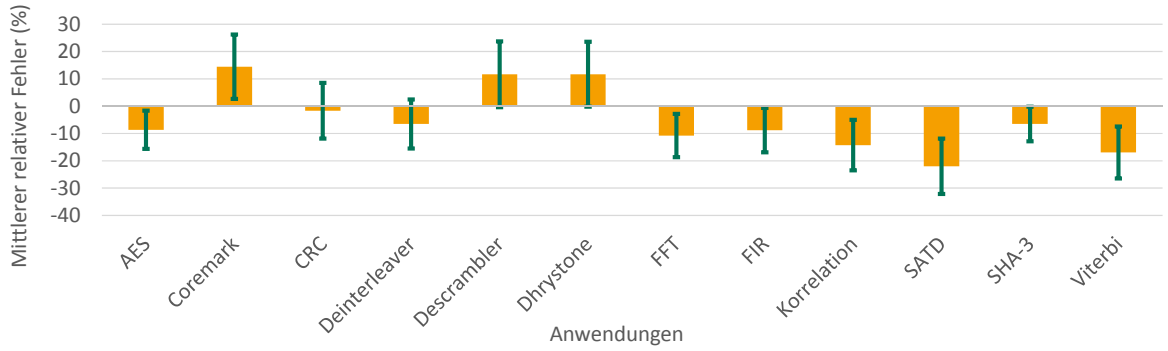

Abbildung 9.2: Mittlerer relativer Fehler und Standardabweichung der Approximation der Energie

prozentuale Abweichung zu der Konfiguration angegeben, die fälschlicherweise als die energieeffizienteste Konfiguration eingestuft wurde.

Obwohl die absoluten Energiewerte teilweise mit hohen Fehlern behaftet sind, besitzen diese Fehler eine vergleichsweise geringe Streuung und beeinflussen die relativen Vergleiche zwischen den einzelnen Prozessorkonfigurationen kaum. Die modellbasierte Entwurfsraumexploration ist bei der Suche nach der energieeffizientesten Prozessorkonfiguration dementsprechend in acht von zwölf Fällen in der Lage, das richtige Ergebnis zu liefern. In den verbleibenden vier Fällen sind die Abweichungen jedoch so gravierend, dass die relativen Vergleiche zwischen den verschiedenen Konfigurationen nicht mehr übereinstimmen und die Approximation eine falsche Konfiguration auswählt. Um in der nachfolgenden experimentellen Entwurfsraumexploration sichergehen zu können, dass die energieeffizienteste Konfiguration von der modellbasierten Entwurfsraumexploration nicht frühzeitig

herausgenommen wurde, kann die Konfigurationsmenge bei den hier untersuchten Anwendungen folglich nur auf die Konfigurationen eingeschränkt werden, die in einem Bereich von ±7% um die vom Modell ermittelte Prozessorkonfiguration liegen. Im Fall des FIR-Filters wären das neben der Konfiguration S2M1D1L1 auch die Konfigurationen S2M2D1L1 (326,14 nWs), S2M1D1L2 MB (328,20 nWs) und S2M2D1L2 MB (332,90 nWs).

Anwendung	Experimentelle Entwurfsraumexpl.		Modellbasierte Entwurfsraumexpl.		Abweichung
	Konfiguration	Energie (nWs)	Konfiguration	Energie (nWs)	(%)
AES	S2M1D1L1	691,43	S2M1D1L1	700,00	
Coremark	S1M1D1L1	26525,30	S1M1D1L1	27045,93	
CRC	S2M1D1L1	53,88	S2M1D1L1	50,87	
Deinterleaver	S2M1D1L1	83,27	S2M1D1L1	74,99	
Descrambler	S1M1D1L1	3636,63	S1M1D1L1	3653,01	
Dhrystone	S1M1D1L1	2576,02	S1M1D1L1	2495,48	
FFT	S2M1D1L1	11313,46	S3M2D1L1	10741,78	+3,78
FIR	S2M1D1L1	359,49	S2M1D1L1	320,88	
Korrelation	S2M2D1L1	129,99	S2M2D1L1	129,60	
SATD	S2M1D1L1	38,47	S3M1D1L1	31,55	+6,43
SHA-3	S2M1D1L1	687,52	S1M1D1L1	631,89	-2,57
Viterbi	S2M1D1L1	57340,21	S3M1D1L1	49048,41	+6,97

Tabelle 9.1: Energieeffizienteste Prozessorkonfiguration aus experimenteller und modellbasierter Entwurfsraumexploration

9.2 Entwurfsraumexploration mit Pareto-Optimierungen

Um die Konfigurationsmenge des CoreVA-Prozessors mit Hilfe von Pareto-Optimierungen einzuschränken, werden die approximierten Laufzeiten aus Kapitel 7.3 und die approximierten Leistungsaufnahmen aus Kapitel 8.5 wie in den Arbeiten von Palesi [61] und Ascia [7] in ein Pareto-Diagramm übertragen. Wie die Abbildung 9.3 zeigt, bildet sich eine Pareto-Front, die aus mehreren Pareto-optimalen Konfigurationen besteht (grüne Punkte). Diese Konfigurationen stellen jeweils Lösungen dar, die einen für ein bestimmtes Anwendungsfeld optimalen Kompromiss aus Leistungsfähigkeit und Leistungsaufnahme beschreiben.

Eine detailliertere Betrachtung des Pareto-Diagramms zeigt, dass sich mehrere Konfigurationsgruppen bilden, in denen die Werte sehr nah beieinander liegen. Diese Gruppen entstehen, da das Hinzufügen einer zweiten oder dritten Verarbeitungseinheit die Leistungsfähigkeit des Prozessors zwar deutlich anhebt, das Hinzukommen weiterer MAC-Einheiten im Falle des FIR-Filters jedoch keine gravierenden Verbesserungen hervorruft. Die Auswertung des Pareto-Diagramms geht somit über die

Analyse der Energieeffizienz hinaus, da sich nun die effizientesten Konfigurationen in verschiedenen Leistungsbereichen ermitteln lassen.

Ein Vergleich des modellbasierten Pareto-Diagramms mit dem in Abbildung 9.4 dargestellten Diagramm auf Basis experimentell ermittelter Werte zeigt jedoch, dass sich die Anordnung der Konfigurationen durch die Approximationsfehler geringfügig verschiebt. Da die Konfigurationen in den einzelnen Gruppen so nah beieinander liegen, reichen diese Verschiebungen aus, um falsche Konfigurationen als Pareto-optimal zu definieren. Bei der Bewertung der untersuchten Zielanwendungen kann aus diesem Grund festgestellt werden, dass die modellbasierte Entwurfsraumexploration im Durchschnitt nur 72,30% der Konfigurationen als Pareto-optimal ansieht, die auch von der experimentellen Entwurfsraumexploration detektiert werden. Es sollte deshalb bei der Einschränkung der Konfigurationsmenge wiederum ein gewisser Unschärfebereich um die Pareto-Front gelegt werden, um den Approximationsfehlern entgegenzuwirken und keine Pareto-optimalen Konfiguration auszulassen. Auf Basis der hier untersuchten Stichproben ist dieser Bereich ausreichend groß gewählt, wenn auch die Konfigurationen eingeschlossen werden, deren Leistungsaufnahme 5% über den detektierten Konfigurationen liegt. Die energieeffizientesten Konfigurationen können mit Hilfe des modellbasierten Pareto-Diagramms bereits ohne diesen Unschärfebereich für jede Zielanwendung gefunden werden.

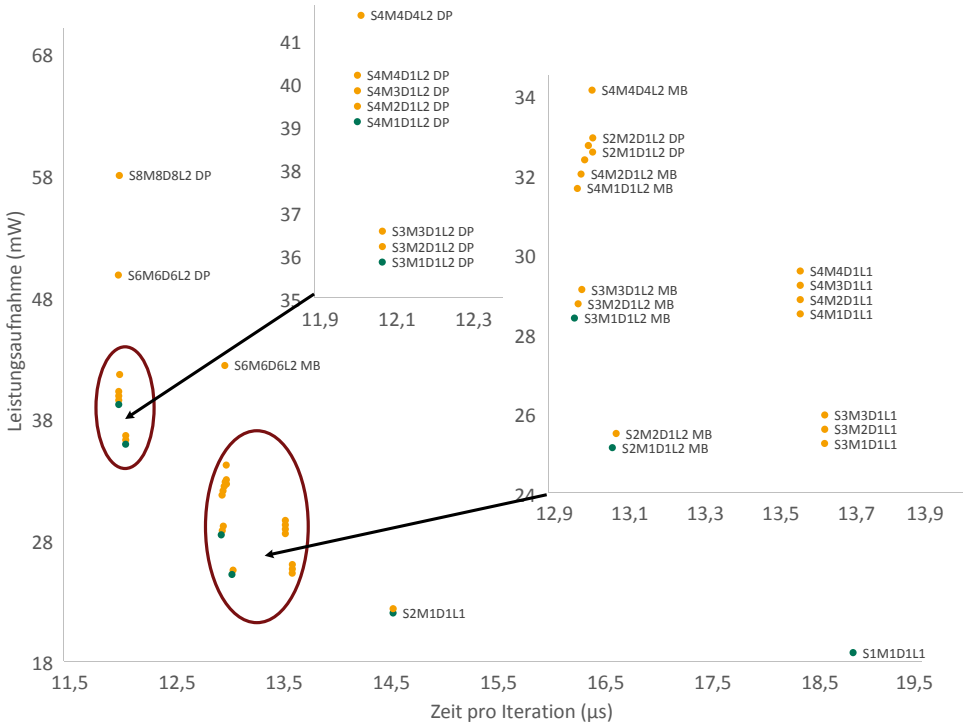

Abbildung 9.3: Approximationsbasiertes Pareto-Diagramm des FIR-Filters

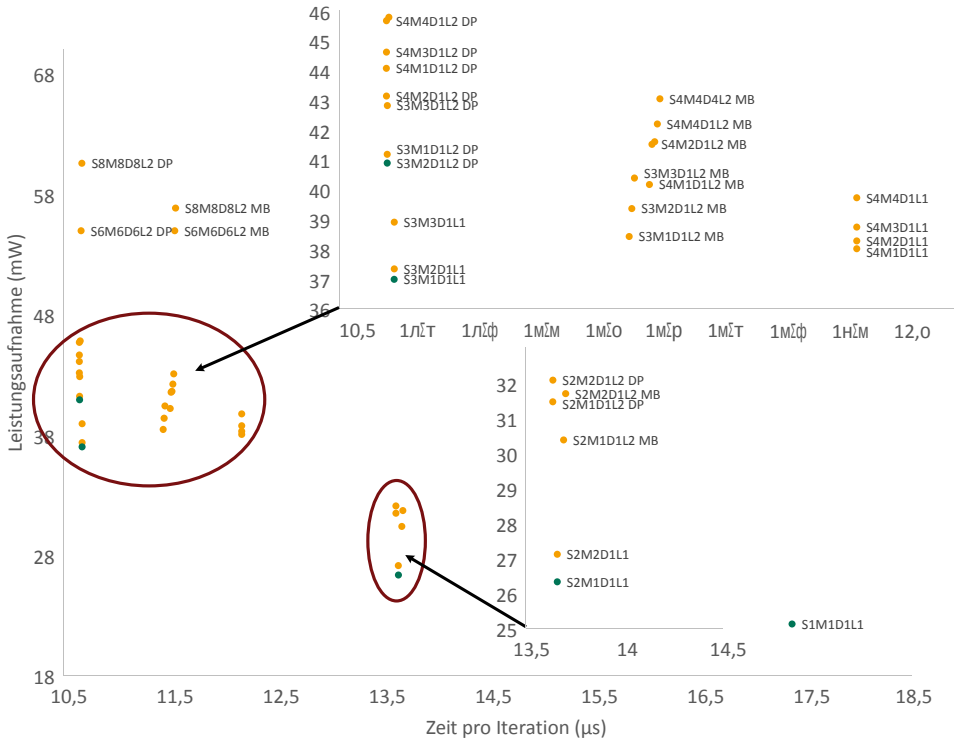

Abbildung 9.4: Experimentell ermitteltes Pareto-Diagramm des FIR-Filters

9.3 Gegenüberstellen der Leistungssteigerungen und des Mehraufwands

Da die statische Anwendungsanalyse aus Kapitel 7.4 nicht in der Lage ist, die tatsächliche Anzahl der Prozessortakte zu approximieren, wird im Folgenden ein Ansatz vorgestellt, bei dem die Änderungen der Leistungsfähigkeit und des Ressourcenaufwands direkt gegenübergestellt werden. Dieser Ansatz basiert wiederum auf dem Ziel, eine möglichst energieeffiziente Konfiguration zu ermitteln. Es wird daher festgelegt, dass die Leistungssteigerungen, die sich aus dem Hinzufügen zusätzlicher Verarbeitungs- und Funktionseinheiten ergeben, die jeweiligen Erhöhungen der Leistungsaufnahme unmittelbar kompensieren müssen.

Die Leistungssteigerungen werden auf Basis der in Kapitel 5 vorgestellten Formel nach Amdahl approximiert. Da diese Formel jedoch nur eine N-fache Parallelverarbeitung betrachtet und dementsprechend nur für Prozessorkonfigurationen mit einer maximalen Anzahl an MAC- und LD/ST-Einheiten gilt, wurde sie in Anlehnung an die Arbeit von Wang [81] durch das getrennte Erfassen der Funktionseinheiten erweitert (siehe Formel 9.1). Im Unterschied zu Wang [81], bei dem zur Ermittlung der Leistungssteigerung ein fiktiver Parallelanteil vorgegeben wurde, kann in dieser Ar-

beit auf den Parallelanteil der jeweiligen Zielanwendungen zurückgegriffen werden, der in der statischen Programmcodeanalyse bestimmt wurde. Die Auswirkungen der verschiedenen Konfigurationselemente können anhand der dort ebenfalls ermittelten Verhältnisse zwischen den ALU-, MAC- und LD/ST-Instruktionen gewichtet werden. Die Höhe der Leistungsaufnahme wird schließlich wiederum mit Hilfe des in Kapitel 8.5 vorgestellten Hardwaremodells bestimmt. In der direkten Gegenüberstellung wird nun jedoch nicht mehr der absolute Wert der Leistungsaufnahme zu Grunde gelegt, sondern ihr relativer Anstieg verwendet (siehe Formel 9.2).

$$\begin{aligned} Leistungssteigerung\ nach\ Amdahl &= \frac{1}{1 - P + \frac{P}{N}} \\ Leistungssteigerung &= \frac{1}{1 - P + \frac{P}{Anteil_{ALU} \cdot Slots + Anteil_{MAC} \cdot MAC + Anteil_{LD/ST} \cdot LD/ST}} \end{aligned} \tag{9.1}$$

$$Anstieg\ \ der\ \ Leistungsaufnahme = \frac{Leistungsaufnahme_{SwMxDyLz}}{Leistungsaufnahme_{S1M1D1L1}} \tag{9.2}$$

Bei der Gegenüberstellung der approximierten Leistungssteigerungen und der jeweiligen Mehraufwände zeigt sich für die Verarbeitung des FIR-Filters, dass die Konfigurationen S1M1L1, S2M1L1 und S2M2L1 das Potential besitzen, die Energieeffizienz des CoreVA-Prozessors zu erhöhen, da die Differenz zwischen der Leistungssteigerung und dem relativen Anstieg der Leistungsaufnahme einen positiven Wert annimmt (siehe Abbildung 9.5). Diese Aussage deckt sich sehr gut mit den Ergebnissen aus der Betrachtung der Energie und des Pareto-Diagramms, da diese drei Konfigurationen sowohl die energieeffizienteste Konfiguration, als auch zwei der vier Pareto-optimalen Konfigurationen darstellen. Analysiert man wiederum die in Kapitel 6 vorgestellten Zielanwendungen, lässt sich feststellen, dass dieses Verfahren in der Lage ist, die Konfigurationsmenge so einzuschränken, dass sie im Durchschnitt aus sechs Prozessorkonfigurationen besteht, die energieeffizienteste Konfiguration enthält und ein Drittel aller Pareto-optimalen Konfigurationen einschließt.

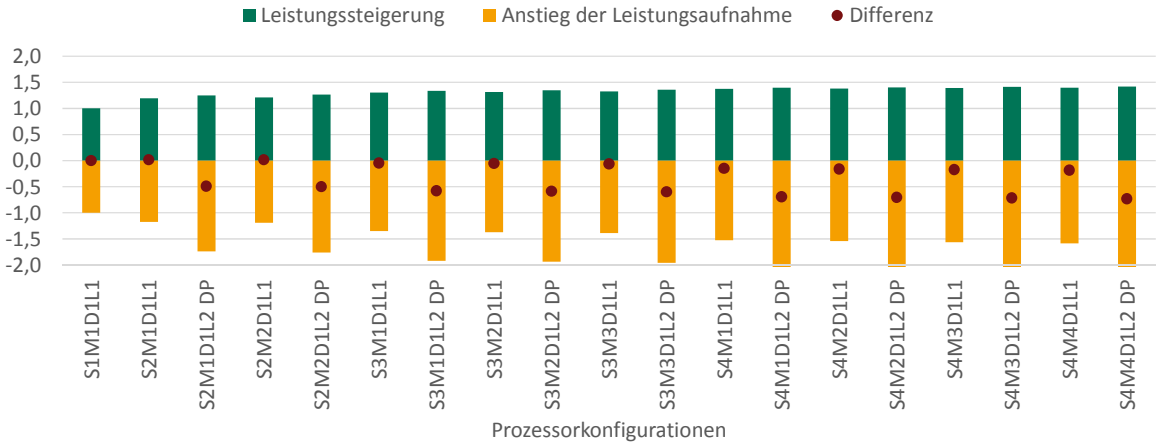

Abbildung 9.5: Gegenüberstellung der Leistungssteigerung und des Anstiegs der Leistungsaufnahme

9.4 Zusammenfassung

In diesem Kapitel wurden drei Verfahren zur modellbasierten Entwurfsraumexploration des CoreVA-Prozessors vorgestellt. Die Verfahren basieren jeweils auf der Gegenüberstellung eines Software- und eines Hardwaremodells, wodurch die Wechselwirkungen zwischen der Leistungsfähigkeit und der Leistungsaufnahme der einzelnen Prozessorkonfigurationen ausgewertet werden können.

Wie bereits in den Kapiteln 7.3 und 7.4 beschrieben wurde, basieren die Softwaremodelle entweder auf einer dynamischen oder einer statischen Anwendungsanalyse. Bei der dynamischen Anwendungsanalyse müssen die zu untersuchenden Anwendungen zuerst für einen Prozessor mit maximaler Parallelität kompiliert werden, um anschließend in einem einmaligen Durchlauf des Instruktionssatzsimulators ausgewertet werden zu können (siehe Abbildung 9.6). Die Simulation ermittelt die durchschnittliche Auslastung der einzelnen Verarbeitungs- und Funktionseinheiten, woraufhin das Softwaremodell die Auslastungen auf allen Prozessorkonfigurationen approximiert und hieraus auf die Laufzeit der Anwendungen schließt. Bei der statischen Anwendungsanalyse müssen die Programme ebenfalls kompiliert werden. Hierbei werden jedoch die in dem Kapitel 7.4.3 beschriebenen Compilererweiterungen aktiviert, die die Instruktionsgruppen der einzelnen Basisblöcke auswerten. Da die Schleifenzugehörigkeiten der Basisblöcke in diesen Auswertungen jedoch noch nicht bekannt sind, müssen sie mit Hilfe einer Zyklensuche bestimmt werden, bevor das zugehörige Softwaremodell den durchschnittlichen Parallelanteil und das durchschnittliche Verhältnis zwischen ALU-, MAC- und LD/ST-Instruktionen approximieren kann. Zur Aufstellung des Hardwaremodells müssen schließlich sieb-

zehn Probesynthesen durchgeführt werden, um die Startwerte und die jeweiligen Mehraufwände für die Approximation der konfigurationsspezifischen Leistungsaufnahmen und Prozessorflächen errechnen zu können.

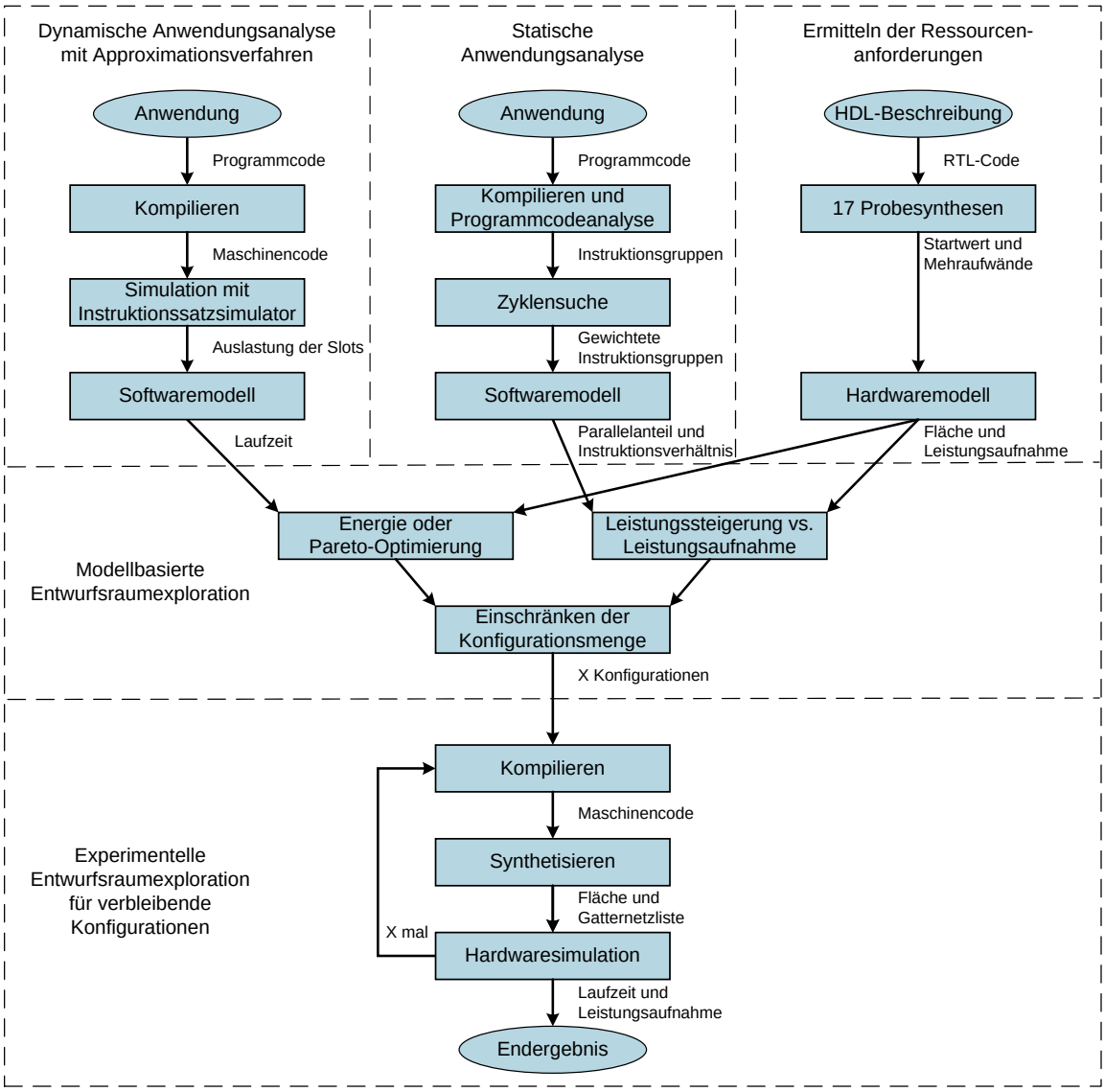

Abbildung 9.6: Ablauf der modellbasierten Entwurfsraumexploration

Da die Modelle mit gewissen Approximationsfehlern behaftet sind, werden mit ihrer Hilfe nun keine endgültigen Entscheidungen getroffen, sondern lediglich besonders ineffiziente Konfigurationen aus der Konfigurationsmenge herausgenommen, bevor die verbleibenden Konfigurationen in einer experimentellen Entwurfsraumexploration erneut untersucht werden. Auf Basis des Hardwaremodells und des Modells der dynamischen Anwendungsanalyse kann hierzu die Energie berechnet

werden, die benötigt wird, um die Zielanwendungen auf den jeweiligen Prozessorkonfigurationen auszuführen. Alternativ lassen sich die Ergebnisse dieser Modelle auch in Pareto-Optimierungen einsetzen. Falls bei der Einschränkung der Konfigurationsmenge anstelle des dynamischen Analyseverfahrens das Softwaremodell der statischen Programcodeanalyse genutzt werden soll, können die hiermit ermittelten Leistungssteigerungen den Ressourcenanforderungen der zusätzlichen Verarbeitungs- und Funktionseinheiten auch direkt gegenübergestellt werden.

In einer exemplarischen Entwurfsraumexploration für einen CoreVA-Prozessor mit bis zu vier Verarbeitungseinheiten waren die modellbasierten Verfahren bei allen Zielanwendungen in der Lage, die Konfigurationsmenge auf weniger als zehn Konfigurationen zu reduzieren. Aus diesem Grund mussten für die Exploration dieses Prozessors nun nicht mehr 352 Synthesen und Hardwaresimulationen durchgeführt werden, sondern lediglich siebzehn Probesynthesen zur Aufstellung des Hardwaremodells, eine Simulation zur dynamischen Anwendungsanalyse und maximal zehn konfigurationsspezifische Hardwaresimulationen und Synthesen zur Auswahl der effizientesten Konfiguration. Weil eine Synthese etwa 20 Minuten dauert und die einzelnen Hardwaresimulationen für die Aufzeichnung der Schaltaktivitäten jeweils ein bis drei Stunden benötigen, dauert eine experimentelle Untersuchung des kompletten Entwurfsraums im Durchschnitt 34 Tage[1]. Da das Aufstellen der Modelle und die Untersuchung der eingeschränkten Konfigurationsmenge jedoch lediglich 29 Stunden benötigt, ergibt sich durch den Einsatz der modellbasierten Verfahren eine Zeitersparnis von 96%.

[1] Die Zeitangaben gelten unter der Annahme, dass die Synthesen und Simulationen nicht parallel durchgeführt werden.

10 Vergleich verschiedener Prozessorsysteme

Als Ergänzung zu dem bereits in Kapitel 2 dargestellten Stand der Technik, werden nun die Ressourcenanforderungen und die Leistungsfähigkeiten der Prozessoren in den Tabellen 10.1 und 10.2 nochmals zusammengefasst. Wie bei Prozessorvergleichen üblich, werden hierbei sowohl die Betriebsfrequenzen, Chipflächen und Leistungsaufnahmen betrachtet, als auch die häufig verwendeten Coremark- und Dhrystone-Benchmarks einbezogen. Die folgende Analyse beschränkt sich dementsprechend auf die ASIC-basierten Systeme, für die ein zusammenhängender Datensatz mit all diesen Bewertungskriterien recherchiert werden konnte. Die Daten der ARM-, MIPS- und ARC-Prozessoren entstammen hierbei verschiedenen Artikeln der Zeitschrift Microprocessor Report (siehe Quellenangaben in den Tabellen 10.1 und 10.2). Die Werte der Cadence Xtensa-Prozessoren basieren auf den Simulationen und Abschätzungen der Entwicklungsumgebung Xtensa Xplorer. Die Prozessoren, die für eine Implementierung auf FPGA-Plattformen ausgelegt sind, können mangels vergleichbarer Kenngrößen nicht einbezogen werden.

Name	Technologie	Frequenz (MHz)	Fläche (mm^2)	Leistungsaufnahme (μW/MHz)	Quelle
Cortex M0+	90nm LP	50	0,04	9,80	[14, 30]
Cortex A5	TSCM 28nm HPM	600	0,40	90	[45, 46]
Cortex A7	TSCM 28nm HKMG	1500	0,58	120	[21, 45, 46]
Cortex A9	TSCM 28nm HKMG	2000	1,50	350	[21, 46]
Cortex A12	TSCM 28nm HKMG	2000	2,00	400	[21, 44]
Cortex A15	TSCM 28nm HKMG	2000	2,80	600	[21, 45]
Warrior M5100	28nm HPM	496	0,23	39	[23]
Xtensa LX 6	28nm HPL	976	0,13	53,90	Xplorer
Xtensa LX 6 FLIX3	28nm HPL	929	0,24	72,56	Xplorer
ARC-EM4	40nm GP	885	0,04	8,40	[30]
ARC-HS34	TSCM 28nm HPM	1100	0,12	33	[38]
S1M1D1L1	STM 28nm LP	750	0,11	23,03[1] / 24,85[2]	Synthese, HW-Sim
S4M4D4L2 DP	STM 28nm LP	750	0,29	42,01 / 43,11	Synthese, HW-Sim

[1]Coremark
[2]Dhrystone

Tabelle 10.1: Ressourcenanforderungen verschiedener Prozessorsysteme

Name	Instruktions-satz	Pipeline-stufen	Verarb.-einh.	Speicher (kByte)	Coremark (Cmark/Mhz)	Dhrystone (DMIPS/MHz)	Quelle
Cortex M0+	ARMv6M	2	1	n/a	2,46	0,93	[14, 30]
Cortex A5	ARMv7	8	1	2 x 32	2,30	1,57	[45, 46]
Cortex A7	ARMv7	8	2	2 x 32	2,60	1,95	[21, 45, 46]
Cortex A9	ARMv7	10	2	2 x 32	2,90	2,50	[21, 46]
Cortex A12	ARMv7	11	2	2 x 32	3,50	3,00	[21, 44][3]
Cortex A15	ARMv7	18	3	2 x 32	3,30	3,50	[21, 45][4]
Warrior M5100	MIPS32 R5	5	1	2 x 32	3,40	1,57	[23]
Xtensa LX6	RF2014.0	7	1	2 x 16	1,67	1,09	Xplorer
Xtensa LX6 FLIX3	RF2014.0	7	3	2 x 16	1,99	1,21	Xplorer
ARC-EM4	ARCv2	3	1	n/a	2,29	1,52	[30]
ARC-HS34	ARCv2	10	1	2 x 16	3,40	1,93	[38]
S1M1D1L1	CoreVA	6	1	2 x 16	1,73	0,55	ISS[5]
S4M4D1L2 DP	CoreVA	6	4	2 x 16	1,90	0,58	ISS

[3] Dhrystone aus Herstellerangaben
[4] Benchmarks aus Herstellerangaben
[5] Instruktionssatzsimulator

Tabelle 10.2: Leistungsfähigkeit verschiedener Prozessorsysteme

Die Datensätze werden mit den Werten zweier CoreVA-Prozessoren verglichen. Zum Einen mit der Prozessorkonfiguration S1M1D1L1, die der energieeffizientesten Konfiguration bei der Verarbeitung der Benchmarkprogramme entspricht, und zum Anderen mit der Konfiguration S4M4D1L2 DP, die den höchsten Durchsatz garantieren kann. Für die Bewertung der folgenden Vergleiche sei jedoch erwähnt, dass die Ressourcenanforderungen der Referenzprozessoren auf unterschiedlichen Standardzellentechnologien und teilweise sogar auf unterschiedlichen Strukturgrößen basieren. Des Weiteren weisen die Prozessoren verschiedene Speichergrößen auf. Da die Anteile der Speicher an der gesamten Prozessorfläche und Leistungsaufnahme nicht bekannt sind, können die Speicher nicht auf eine einheitliche Größe skaliert werden.

Zur besseren Visualisierung der Ressourcenanforderungen wird in Abbildung 10.1 ein Graph dargestellt, bei dem die normierte Leistungsaufnahme gegen die Betriebsfrequenz aufgetragen ist. Die Größe der Punkte spiegelt die jeweiligen Prozessorflächen wider. Bei der Betrachtung dieses Graphen zeichnen sich drei Prozessorklassen ab, in die sich die vorgestellten Prozessoren eingruppieren. Die erste Klasse bilden Mikrocontroller, die einen sehr eingeschränkten Funktionsumfang besitzen und in tief eingebetteten Systemen eingesetzt werden. Zu diesen Mikrocontrollern zählen der ARM Cortex M0+ und der Synopsys ARC-EM4, die weder über leistungsfähige Multiplizierereinheiten, noch über SIMD-Erweiterungen oder Caches verfügen. Die Multiplizierereinheit des Cortex M0+ benötigt beispielsweise 32 Takte zur Berechnung einer 32x32-Bit Multiplikation. Durch diesen Minimalismus können die Prozessoren zwar nicht für rechenintensive Arbeiten eingesetzt werden, ihre Chipflächen und

Leistungsaufnahmen liegen mit jeweils 0,04 mm^2 und mit 9,80 $\mu W/MHz$ beziehungsweise 8,40 $\mu W/MHz$ jedoch weit unter dem Durchschnitt der hier gezeigten Prozessoren.

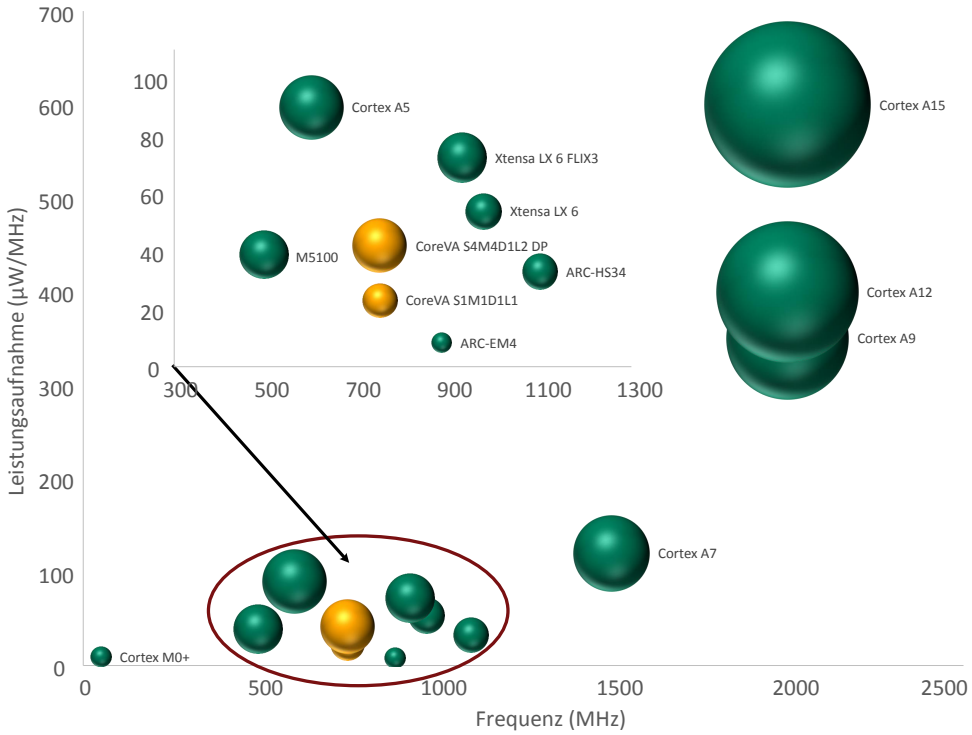

Abbildung 10.1: Ressourcenanforderungen verschiedener Prozessorsysteme

Die zweite Klasse ist durch kleinere Signalverarbeitungsprozessoren bestimmt. Diese Prozessoren zählen zwar ebenfalls zu den Mikrocontrollern, sie können jedoch durch parallele Verarbeitungseinheiten oder durch spezielle Multiplizier- und Vektorrecheneinheiten erweitert werden, die direkt auf eine Beschleunigung digitaler Signalverarbeitungsalgorithmen ausgelegt sind. Zu den Prozessoren dieser Klasse gehören der MIPS Warrior-M5100, der Cadence Xtensa LX 6 und der CoreVA-Prozessor. Bei der Betrachtung der einzelnen Prozessoren lässt sich erkennen, dass die Funktionseinheiten des M5100 und des LX 6 im Vergleich zum CoreVA etwas umfangreicher ausgestattet sind, was sich in leicht höheren Ressourcenanforderungen widerspiegelt. So besitzen der M5100 und der LX 6 Multiplizierereinheiten, die eine 32x32-Bit MAC-Berechnung innerhalb eines Taktes durchführen können. Beim CoreVA-Prozessor wurde diese Berechnung zu Gunsten einer höheren Betriebsfrequenz auf zwei Prozessortakte aufgeteilt (siehe Kapitel 3.8). Des Weiteren haben der M5100 und der LX 6 SIMD-Recheneinheiten, die das Ausführen zweifacher 16-Bit Instruktionen ermöglichen. Der CoreVA-Prozessor kann zwar auch durch eine solche Recheneinheit erweitert werden, dies ist in den hier dargestellten Konfigurationen aufgrund mangelnder Auslastung jedoch nicht geschehen. Der LX 6 und der CoreVA können

schließlich zu einem VLIW-Prozessor mit mehreren Verarbeitungseinheiten ausgebaut werden. Um dies zu berücksichtigen, werden in der Gegenüberstellung jeweils eine Konfiguration mit einem VLIW-Slot (Xtensa LX 6 und CoreVA S1M1D1L1) und eine Konfiguration mit drei (Xtensa LX 6 FLIX3) oder vier (CoreVA S4M4D4L2 DP) VLIW-Slots untersucht. Es zeigt sich, dass sich die Ressourcenaufwände beim LX 6 durch das Aktivieren der dreifachen VLIW-Unterstützung in einem ähnlichen Maße erhöhen, wie bei einem CoreVA-Prozessor mit drei Verarbeitungseinheiten. So steigt die Fläche und die Leistung beim LX 6 um 85% und 35%, beim CoreVA steigen die Werte um 63% und 48%.

Die dritte Klasse bilden schließlich die superskalaren Anwendungsprozessoren der ARM Cortex A-Reihe, die in Smartphones und Tabletcomputern eingesetzt werden. Die Prozessoren haben bis zu drei parallele Verarbeitungseinheiten und sind standardmäßig mit Instruktions- und Datencaches, 128-Bit SIMD-Erweiterungen und Fließkommarecheneinheiten ausgestattet. Hierdurch ist der Ressourcenaufwand um ein Vielfaches höher als bei den anderen Prozessoren, insbesondere da die Cortex A-Prozessoren aufgrund ihrer Superskalarität deutlich aufwändigere Steuerlogiken zur Realisierung des Schedulings besitzen als VLIW-Prozessoren. Auch das Implementieren verhältnismäßig vieler Pipelinestufen wirkt sich aufgrund der einzufügenden Pipelineregister negativ auf die Ressourcenanforderungen aus. Im Vergleich zum 4-Slot CoreVA hat der Cortex A15 eine fast zehnmal größere Chipfläche und eine vierzehnfache Leistungsaufnahme.

Zur Beschreibung der Leistungsfähigkeit werden für eingebettete Prozessoren in der Literatur meistens die Ergebnisse der Coremark- und Dhrystone-Benchmarks angegeben. Wie bereits in Kapitel 6.2 beschrieben wurde, werden die Benchmarkergebnisse aus der Laufzeit errechnet, die die Prozessoren zur Verarbeitung einzelner Testdurchläufe benötigen. Da die Ergebnisse somit in direkter Abhängigkeit zur Betriebsfrequenz stehen, werden sie in der Regel durch eine Normierung auf ein vergleichbares Maß umgerechnet. Hierdurch sind die Vorteile einer höheren Betriebsfrequenz zwar nicht mehr zu erkennen, die Anzahl und die Ausstattung der einzelnen Verarbeitungseinheiten bekommt jedoch eine größere Bedeutung.

Der in Tabelle 10.2 dargestellte Vergleich der normierten Benchmarkergebnisse zeigt, dass die Leistungsfähigkeit der Anwendungsprozessoren erwartungsgemäß hoch ist. Der Cortex A5 erreicht beispielsweise ein Coremark-Ergebnis von 2,30 Coremarks/MHz und einen Dhrystone-Wert von 1,57 DMIPS/MHz. Durch das Hinzufügen zusätzlicher Verarbeitungseinheiten können diese Werte mit dem Cortex A15 sogar auf 3,30 Coremarks/MHz und 3,50 DMIPS/MHz gesteigert werden. Die Benchmarkergebnisse der Mikrocontroller Cortex M0+, ARM-EM4 und M5100 bewegen sich in einem ähnlichen Bereich wie die Werte des Cortex A5. Das Coremark-Ergebnis des M5100 liegt sogar zwischen den Werten des Cortex A12 und A15. Dies bedeutet zwar nicht, dass die Mikrocontroller den Anwendungsprozessoren

ebenbürtig sind, da die tatsächlichen Programmlaufzeiten aufgrund der deutlich geringeren Betriebsfrequenzen um ein Vielfaches höher liegen. Die Werte zeigen jedoch, dass die Mikrocontroller eine vergleichsweise hohe Rechenleistung pro Prozessortakt erzeugen, weil sie beispielsweise nur lokale Speicher besitzen und somit nicht auf das Nachladen des Caches warten müssen.

Die Prozessoren Xtensa LX 6 und CoreVA S1M1D1L1 zeigen hingegen mit 1,67 Coremarks/MHz und 1,09 DMIPS/MHz beziehungsweise 1,73 Coremarks/MHz und 0,55 DMIPS/MHz deutlich schlechtere Benchmarkergebnisse. Beim CoreVA resultieren die geringen Werte zu einem gewissen Teil aus dem Fehlen bestimmter Bypasspfade und aus den hierdurch hinzukommenden Wartezyklen (siehe Kapitel 3.14 und 7.1). In der aktuellen Konfiguration des CoreVA-Prozessors sind sämtliche Conditionregister- und Kontrollbypasspfade der Execute-Stufe deaktiviert, da in vorangegangenen Untersuchungen gezeigt werden konnte, dass die maximale Betriebsfrequenz hierdurch um 26% angehoben wird. Da die Anzahl der auszuführenden Prozessortakte durch das Deaktivieren dieser Bypässe jedoch nur um 16% (Coremark) oder 8% (Dhrystone) steigt, wirkt sich diese Bypasskonfiguration insgesamt positiv auf die Laufzeit der Benchmarks aus [90].

Neben dieser Einschränkung hat die Analyse der Prozessorhardware jedoch keine gravierenden Defizite aufgezeigt, die diese hohen Abweichungen erklären würden. So scheidet das Fehlen einer Fließkommarecheneinheit als Argument aus, da die Benchmarks ausschließlich ganzzahlige Berechnungen durchführen. Wartezeiten, die durch das Lesen der Instruktions- und Datenspeicher entstehen können, kommen ebenfalls nicht in Betracht, weil die lokalen Scratchpad-Speicher die benötigten Inhalte garantiert nach zwei Takten zur Verfügung stellen. Da die untersuchten Prozessoren vergleichbare Instruktionssätze unterstützen und folglich ähnliche Instruktionsfolgen zur Verarbeitung der Benchmarks benutzen sollten, lässt sich die geringe Leistungsfähigkeit nur dadurch erklären, dass der Compiler das Potential des CoreVA-Prozessors nicht voll ausschöpfen kann. Dies zeigt sich auch in einer Gegenüberstellung baugleicher Prozessoren mit unterschiedlicher Anzahl von Verarbeitungseinheiten. Während der Vergleich des Cortex A5 und des Cortex A15 zeigt, dass die Coremark-Ergebnisse durch das Hinzufügen zweier Verarbeitungseinheiten eine Steigerung von 43,48% erfahren und die Dhrystone-Werte sogar verdoppelt werden, lässt sich beim CoreVA zwischen den Konfigurationen S1M1D1L1 und S4M4D1L2 DP nur eine Leistungssteigerung von 9,83% und 5,45% verzeichnen. Wie auch schon die Analyse der Parallelität in Kapitel 7.2.1 gezeigt hat, ist der Compiler bei diesen Anwendungen also praktisch nicht in der Lage, von den zusätzlichen Verarbeitungseinheiten zu profitieren.

Betrachtet man jedoch die Energie, die zur Verarbeitung eines Coremark- oder Dhrystone-Durchlaufs erforderlich ist, lässt sich feststellen, dass der CoreVA-Prozessor in diesem Bereich durchaus wettbewerbsfähig ist (siehe Abbildung 10.2). So

benötigt der 1-Slot CoreVA bei der Verarbeitung der Benchmarks deutlich weniger Energie als die Xtensa-Prozessoren und als sämtliche Prozessoren der Cortex A-Serie. Die Energie, die der M5100 und der ARC-HS34 zur Verarbeitung des Coremarks benötigen, liegt nur 13,76% und 27,07% unter der Energie des CoreVA-Prozessors. Die Energieeffizienz des ARC-EM4 und des Cortex M0+ ist aufgrund ihrer sehr geringen Leistungsaufnahme jedoch zwischen 3,6- und 8-mal besser. Diese Werte sind jedoch mit einer gewissen Skepsis zu betrachten, da die letztgenannten Prozessoren in einer anderen Strukturgröße gefertigt werden.

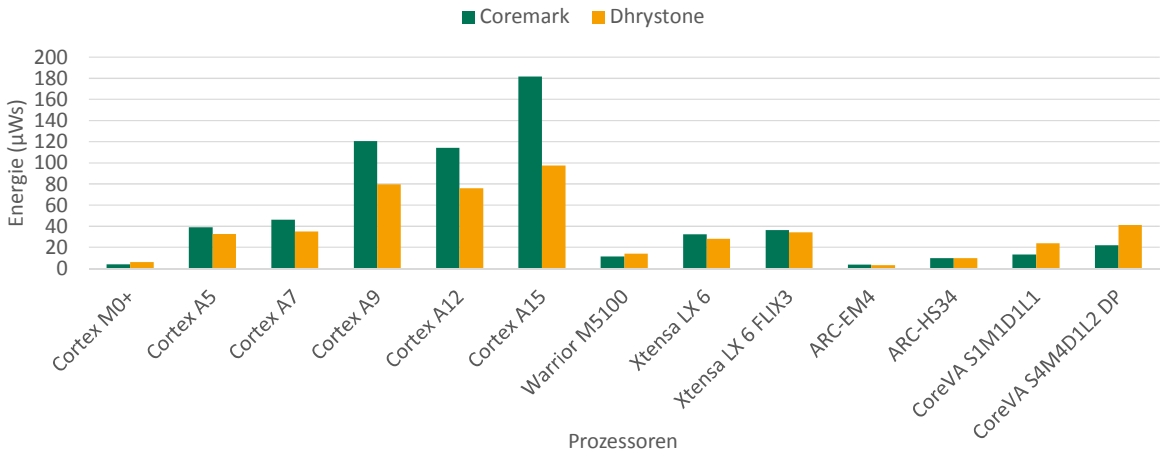

Abbildung 10.2: Energie für einen Coremark- und 1000 Dhrystone-Durchläufe

11 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurden nach einer anfänglichen Vorstellung des CoreVA-Prozessors und seiner zugehörigen Werkzeugkette die verschiedenen Verfahren der anwendungsspezifischen Entwurfsraumexploration gegenübergestellt. Die Entwurfsraumexploration durchsucht die unterschiedlichen Konfigurationsmöglichkeiten eines Prozessors und bewertet jeweils das Zusammenspiel der Leistungsfähigkeit und der Ressourcenanforderungen, um hieraus die Effizienz einer Konfiguration zu bestimmen. Bei der Ermittlung der benötigten Werte kommen zwei verschiedene Vorgehensweisen zum Einsatz. Zum Einen können experimentelle Entwurfsraumexplorationen genutzt werden, bei denen die Kenndaten der einzelnen Prozessorkonfigurationen durch eine Vielzahl konfigurationsspezifischer Synthesen und Simulationen bestimmt werden. Zum Anderen finden häufig modellbasierte Verfahren Verwendung, bei denen die oben genannten Werte mit Hilfe einzelner Probesynthesen und Simulationen ermittelt werden.

11.1 Modellbasierte Entwurfsraumexploration des CoreVA-Prozessors

Da der Entwurfsraum des CoreVA-Prozessors aufgrund seiner vielseitigen Konfigurierbarkeit nicht auf einem experimentellen Weg durchsucht werden kann, wurden in dieser Arbeit verschiedene Ansätze entwickelt, um die Konfigurationsmenge mit Hilfe modellbasierter Verfahren einzuschränken und somit eine Vorauswahl für eine anschließende experimentelle Entwurfsraumexploration zu treffen. Die Charakterisierung der Anwendungen erfolgt hierbei mit Hilfe einer einzelnen konfigurationsunabhängigen Simulation, indem von der Auslastung der VLIW-Slots bei maximaler Parallelität auf die Laufzeit der anderen Prozessorkonfigurationen geschlossen wird. Ein alternativer Ansatz approximiert die durchschnittliche Parallelität und die Anteile der verschiedenen Instruktionstypen sogar ausschließlich auf Basis einer statischen Programmcodeanalyse. Um die Anzahl der Synthesen und Hardwaresimulationen auf ein Minimum zu reduzieren, wurde in einem zweiten Schritt ein Hardwaremodell entwickelt, dass mit Hilfe von siebzehn Probesynthesen

aufgestellt werden kann und 352 experimentelle Synthesen und 352 anwendungsspezifische Hardwaresimulationen ersetzt. Mit Hilfe dieser Modelle konnten schließlich verschiedene Explorationsverfahren durchgeführt werden. Zum Einen konnte mit Hilfe der approximierten Laufzeiten und Leistungsaufnahmen die Energie ermittelt werden, die die jeweiligen Prozessorkonfigurationen zur Ausführung der verschiedenen Zielanwendungen benötigen. Zum Anderen konnten durch das Aufstellen von Pareto-Diagrammen verschiedene Leistungsgruppen ausgemacht werden, die anschließend detaillierter untersucht wurden. In einem dritten Schritt konnte durch das direkte Gegenüberstellen der Leistungssteigerungen und der Ressourcenmehraufwände ein Verfahren entwickelt werden, dass die Konfigurationsmenge auch ohne Kenntnis der tatsächlichen Laufzeit einschränkt.

Nach jedem dieser Entwicklungsschritte wurden umfangreiche Vergleiche mit experimentell ermittelten Werten unternommen, um die Güte der Verfahren sicherzustellen. Es zeigte sich, dass die Laufzeit der untersuchten Anwendungen einen durchschnittlichen relativen Approximationsfehler von 2,03% und eine Standardabweichung von ±6,24% aufweist. Die Überprüfung der modellbasierten Ermittlung der Leistungsaufnahme ergab einen durchschnittlichen Fehler von -5,49% (±15,33%). Bei der Approximation der Energie ergab sich schließlich ein mittlerer Fehler von -4,85% (±14,83%). Da die relativen Vergleiche zwischen den Prozessorkonfigurationen durch diese Abweichungen jedoch nur geringfügig beeinflusst wurden, konnte der modellbasierte Ansatz die energieeffizienteste Konfiguration in acht von zwölf Fällen direkt detektieren. Um sicherstellen zu können, dass diese Konfiguration in den anderen Fällen nicht vorzeitig aus der Konfigurationsmenge entfernt wird, wurde für die untersuchten Stichproben ein Unschärfebereich von ±7% eingeführt. Die Ergebnisse der statischen Programmcodeanalysen wiesen aufgrund der sehr abstrakten Approximation der Basisblockdurchläufe einen durchschnittlichen relativen Fehler von 9,40% (±43,74%) auf. Trotz dieses hohen Fehlers war es bei den betrachteten Anwendungen durch das direkte Gegenüberstellen der Bewertungskriterien möglich, die Konfigurationsmenge so einzuschränken, dass im Durchschnitt nur sechs Prozessorkonfigurationen pro Zielanwendung verbleiben und die energieeffizienteste Konfiguration garantiert enthalten ist.

Obwohl die modellbasierte Entwurfsraumexploration aufgrund ihrer geringen Approximationsfehler bereits eine Zeitersparnis von 96% ermöglicht, werden im Folgenden einige Vorschläge genannt, mit denen die vorgestellten Modelle gegebenenfalls noch genauere Werte liefern können. Bei der Entwicklung des Hardwaremodells wurde die bewusste Entscheidung getroffen, dass das Modell keine Anwendungsabhängigkeiten aufweisen soll und dass dementsprechend von einer durchschnittlichen Schaltaktivität auszugehen ist. Eine detaillierte Analyse der Approximationsfehler zeigte daraufhin, dass zwischen der Parallelität der Zielanwendungen und dem Fehler bei der Approximation der Leistungsaufnahme eine direkte Beziehung besteht. Dieses Verhalten erklärt sich dadurch, dass die hinteren VLIW-Slots bei

einer schlechten Parallelität vergleichsweise selten ausgelastet sind, was zu einer geringeren dynamischen Verlustleistung führt. Der Approximationsfehler konnte durch das Einfügen eines Unschärfebereichs zwar außer Acht gelassen werden, falls die Werte des Hardwaremodells jedoch in Zukunft auch als finales Entscheidungskriterium genutzt werden sollen, könnte die Güte der Approximation durch Einbeziehen der Parallelität maßgeblich verbessert werden. Dies könnte geschehen, indem das Modell um einen Faktor erweitert würde, der die einzelnen Verarbeitungseinheiten mit ihrer jeweiligen Auslastung gewichtet. Zusätzlich könnte bei der Gewichtung der MAC- und LD/ST-Einheiten sogar der Anteil der auszuführenden Instruktionstypen berücksichtigt werden [7].

Wie in Kapitel 7.4.5 beschrieben wurde, resultiert der Fehler der statischen Programmcodeanalyse hauptsächlich aus der ungenauen Approximation der Basisblockaufrufe. Dies kann zum Einen an der hier verwendeten Annahme von Wagner [79] liegen, die aussagt, dass gefundene Schleifen in Schnitt fünfmal durchlaufen werden. Zum Anderen können die Abweichungen auch aus der stets gleichen Gewichtung der Fallunterscheidungen resultieren. Um die Annahme von Wagner zu überprüfen, könnte der Instruktionssatzsimulator des CoreVA-Prozessors um eine Funktionalität zum Zählen der Basisblockdurchläufe erweitert werden. Hierdurch würde auch ein probates Werkzeug geschaffen, das zur Verifikation des Zyklensuchalgorithmus eingesetzt werden könnte. Die Ausführungshäufigkeit der einzelnen Fallunterscheidungszweige könnte genauer abgeschätzt werden, indem die enthaltenen Instruktionen genauer analysiert würden. So ließe das Auffinden von assert-, return- oder break-Befehlen darauf schließen, dass ein Zweig einen Programmabbruch darstellt und dementsprechend nur selten aufgerufen wird. Zweige, in denen mehrere Berechnungen stattfinden, würden hingegen häufiger durchlaufen [79].

Im Rahmen zukünftiger Arbeiten könnte untersucht werden, ob eine feinere Aufschlüsselung der Ressourcenaufwände die Modellierung des Prozessors gegebenenfalls noch weiter verbessern kann. So ließe sich beispielsweise die Energie bestimmen, die zur Ausführung einer einzelnen Instruktion aufgewendet werden muss. Anschließend könnten diese Werte sogar in Bezug zu den jeweiligen Eingangsdaten gesetzt werden. Neben einer potentiellen Verbesserung der Modellierung könnten hierdurch gegebenenfalls Schwachstellen im Instruktionssatz und in der Implementierung des Prozessorkern aufgezeigt werden. Zuletzt müsste noch eine Entwurfsraumexploration des Speichers durchgeführt werden, da der Speicherbedarf und hieraus resultierende Notwendigkeit einer Cache-Implementierung sowie die Vor- und Nachteile der verfügbaren Speichermodule bisher nur stichprobenartig bewertet wurde.

11.2 Energieeffizienz des CoreVA-Prozessors

Im letzten Teil dieser Arbeit wurde schließlich ein Vergleich der zuvor ermittelten Prozessorkonfigurationen mit diversen kommerziellen Prozessoren der Firmen ARM, Imagination Technologies, Cadence und Synopsys durchgeführt. Es zeigte sich, dass der 1-Slot CoreVA bei der Verarbeitung eines Coremark- oder Dhrystone-Durchlaufs eine Energieeffizienz erreicht, die zwischen der Effizienz der Anwendungsprozessoren und der Mikrocontroller liegt. Die Prozessoren der ARM Cortex A-Reihe schneiden wegen ihrer überdurchschnittlich hohen Leistungsaufnahmen bei den Energievergleichen bis zu 14-mal schlechter als der CoreVA-Prozessor ab. Die Mikrocontroller Synopsys ARC-EM4 und ARM Cortex M0+ besitzen trotz ihrer eingeschränkten Funktionalität aufgrund ihrer sehr geringen Ressourcenanforderungen eine 3,6- oder 8-mal bessere Energieeffizienz.

Bei der Analyse der Benchmarkergebnisse des 4-Slot CoreVAs wurde jedoch deutlich, dass das Hinzufügen zusätzlicher Verarbeitungseinheiten einen vergleichsweise geringen Anstieg der Leistungsfähigkeit nach sich zieht, wodurch die Energieeffizienz dieser Konfiguration deutlich geringer ausfällt. Dieses Problem erklärt sich dadurch, dass beim LLVM-Compiler das Parallelisieren der Instruktionen erst in den Transformationsblöcken der Endverarbeitung erfolgt (siehe Kapitel 4.1.1). Da an dieser Stelle jedoch nur noch auf Ebene einzelner Basisblöcke gearbeitet wird, lassen sich keine blockübergreifenden Optimierungen durchführen. Bei genauer Betrachtung des erzeugten Assemblercodes lässt sich erkennen, dass der Compiler aufgrund der geringen Basisblockgrößen kaum Möglichkeiten hat, auftretende Datenabhängigkeiten aufzulösen und hierdurch die Parallelität der Programme zu erhöhen. Demgegenüber hat eine manuelle Assemblercodeoptimierung des FIR-Filters gezeigt, dass es für einen 4-Slot CoreVA durchaus möglich ist, eine Parallelität von 3,24 zu erreichen, wenn die Beschränkung durch die Blockgrenzen wegfällt[1] [89]. Um dem Compiler mehr Freiheiten bei der Anordnung der Instruktionen einzuräumen, müsste die Größe der Basisblöcke somit im Vorfeld erhöht werden. Dies könnte beispielsweise durch das automatisierte Zusammenfassen kleinerer Funktionen (Inline-Ersetzung) oder durch das Abrollen von Schleifen geschehen, was vom Programmierer durch spezielle Precompiler-Direktiven[2] initiiert werden könnte. Hierdurch würde auch eine Vielzahl von Sprüngen eingespart, die die Entscheidungsmöglichkeiten des Compilers durch ihre delay-Slots ebenfalls einschränken [50].

Ein zusätzliches Manko der aktuellen Entwicklungsumgebung ist, dass der Compiler bei der Übersetzung der untersuchten Anwendungen keine SIMD-Instruktionen einsetzt, obwohl dies möglich wäre. Die Spezifikation der Instruktionen wurde

[1] Der LLVM-Compiler erreicht in derselben Konfiguration eine Parallelität von 1,61 (siehe Tabelle 7.1)
[2] In den Quelltext eingefügte Steueranweisung für den Compiler, auch Pragma oder Makro genannt

zwar vollständig in die Endverarbeitung des LLVM-Compilers übertragen, die vorderen Compilerstufen erkennen die Einsatzmöglichkeit der SIMD-Instruktionen jedoch nur in speziell konstruierten Testszenarien. Hier müsste gegebenenfalls auf eine neuere LLVM-Version gewechselt werden oder wiederum versucht werden, dem Compiler das Platzieren der SIMD-Instruktionen durch manuell eingefügte Precompiler-Direktiven zu erleichtern. Außerdem könnte die Einsetzbarkeit der SIMD-Recheneinheiten durch die Implementation einer Permutationseinheit erhöht werden, die den SIMD-Instruktionsteilen auch Zugriffe auf Operanden in unterschiedlichen Datenregistern ermöglicht. Ein weiterer Ansatz zur Steigerung der Leistungsfähigkeit könnte auch die Implementierung größerer Vektorrecheneinheiten sein. Diese Einheiten könnten in einem speziellen VLIW-Slot Verwendung finden und beispielsweise SIMD-Instruktionen mit einer Breite von 128-Bit ausführen. Zur effizienten Anbindung der Recheneinheiten müssten hierzu jedoch die Pipeline- und Datenregister sowie die Schreib- und Leseports der Speicher an die jeweiligen Datenbreiten angepasst werden. Alternativ könnten die Vektorrecheneinheiten, wie die DSP-Elemente der Cadence Xtensa-Prozessoren, auch als externe Hardwarebeschleuniger eingebunden werden (siehe Kapitel 3.16 und 2.3).

Zuletzt seien die Überarbeitungen des Registerfiles und der Sprungsteuerung als potentielle Möglichkeiten zur Verbesserung der Energieeffizienz genannt. Die Analyse der Anwendungsverarbeitung hat gezeigt, dass die implementierten Datenregister vergleichsweise schlecht auslastet werden. Bei den hier betrachteten Anwendungen wird beispielsweise maximal die Hälfte der 32 Register genutzt. Zur Steigerung der Leistungsfähigkeit wäre zu prüfen, ob der Compiler gegebenenfalls mehr Operanden und Zwischenergebnisse in den Datenregistern vorhalten müsste, um die Anzahl der Load/Store-Instruktionen zu reduzieren. Falls der Compiler die Register bereits bestmöglich auslastet, könnte das Registerfile im Umkehrschluss verkleinert werden, wodurch die Ressourcenanforderungen des CoreVA-Prozessors reduziert würden. Des Weiteren wurde festgestellt, dass die Anzahl der auszuführenden Sprunginstruktionen bei einigen Anwendungen einen nicht unerheblichen Anteil der Gesamtinstruktionen ausmacht. Da bei der Ausführung eines Sprungs jeweils eine vollwertige Verarbeitungseinheit blockiert wird, könnte die Sprungverarbeitung gegebenenfalls in einen dedizierten VLIW-Slot ausgelagert werden.

Zusammenfassend lässt sich sagen, dass der CoreVA-Prozessor durch seine Energieeffizienz und seine vielseitige Konfigurierbarkeit sehr gut in bestehenden und zukünftigen Anwendungsszenarien einsetzbar ist. Da die Entwurfsraumexploration kein einmaliger Vorgang ist, sondern nach jeder Änderung in der Funktionalität des Prozessors und in den verwendeten Werkzeugen und Standardzellentechnologien erneut durchgeführt werden muss, wird die modellbasierte Entwurfsraumexploration die Weiterentwicklung des CoreVA-Prozessors maßgeblich beschleunigen.

Abkürzungsverzeichnis

3GPP	3rd Generation Partnership Project
AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
AHB	Advanced High-performance Bus
ARC	Argonaut RISC Core
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instructionset Processor
AST	Abstract Syntax Tree
AVC	Advanced Video Coding
BPSK	Binary Phase Shift Keying
BR	Branch
CEA	Commissariat à l'Énergie Atomique et aux énergies alternatives
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal Oxide Semiconductor
CMP	Compare
CoreVA	Configurable resource-efficient VLIW Architecture
CPS	Cycles Per Second
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSA	Carry-Save Addierer
DAG	Directed Acyclic Graph
DC	Instruction-Decode
DIV	Division
DMIPS	Dhrystone-MIPS
DP	Dual-Port
DSP	Digital Signal Processor
DVB-T	Digital Video Broadcasting - Terrestrial

EDP	Energy Delay Product
EEMBC	Embedded Microprocessor Benchmark Consortium
ESA	European Space Agency
ESTEC	European Space Research and Technology Centre
EX	Execute
EXT	Extension
FD-SOI	Fully Depleted - Silicon On Insulator
FE	Instruction-Fetch
FFT	Fast Fourier Transformation
FIR	Finite Impulse Response
FLI	Foreign Language Interface
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GPL	General Public License
GRLIB	Gaisler Research IP Library
HDL	Hardware Description Language
HSDPA	High Speed Downlink Packet Access
I/O	Input/Output
IEC	International Electrotechnical Commission
IFFT	Inverse Fast Fourier Transformation
IPC	Instructions Per Cycle
IP	Intellectual Property
IPS	Instructions Per Second
IR	Intermediate Representation
ISO	International Organization for Standardization
ISS	Instruktionssatzsimulator
ITU	International Telecommunication Union
LD	Load
Leti	Laboratoire d'Electronique des Technologies de l'Information
LLVM	Low Level Virtual Machine
LTE	Long Term Evolution
MAC	Multiply-Accumulate
MB	Multi-Bank
ME	Memory-Access
MIPS	Microprocessor without Interlocked Pipeline Stages Million Instructions Per Second
MLA	Multiply-Accumulate Instruktion im CoreVA-Instruktionssatz

MMIO	Memory Mapped Input/Output
MPEG	Moving Picture Experts Group
MPSoC	Multiprocessor System on Chip
MVC	Move-Constant
NAS	Network Attached Storage
NoC	Network on Chip
NOP	No-Operation
OFDM	Orthogonal Frequency Division Multiplexing
PBV	Parallel Block Vector
PDP	Power Delay Product
PSS	Primary Synchronisation Signal
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Keying
RD	Register-Read
ρ-VEX	Reconfigurable VEX
RISC	Reduced Instruction Set Computer
RS	Resource-Sharing
RTL	Register Transfer Level
SATD	Sum of Absolute Transformed Differences
SDC	Synopsys Design Constraints
SDF	Standard Delay Format
SDR	Software-defined Radio
SHA	Secure Hash Algorithm
SIMD	Single Instruction Multiple Data
SLT	Signed-Lower-Than
SmMnDoLp	Konfiguration mit m Slots, n MAC-, o DIV- und p LD/ST-Einheiten
SPARC	Scalable Processor ARCitecture
SRAM	Static Random Access Memory
SSA	Static Single Assignment
SSD	Solid State Drive
SSS	Secondary Synchronisation Signal
ST	Store
UART	Universal Asynchronous Receiver Transmitter
UMTS	Universal Mobile Telecommunications System

VAX	Virtual Address Extension
VCD	Value Change Dump
VEX	VLIW-Example
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
WR	Register-Write

Literaturverzeichnis

[1] ARM Limited. *ARM Developer Suite Assembler Guide 1.2*. 2000.

[2] Syed Ameer Abbas, Angel Joybell Sheeba und S. J. Thiruvengadam. "Design of Downlink PDSCH Architecture for LTE Using FPGA". In: *2011 International Conference on Recent Trends in Information Technology (ICRTIT)*. IEEE. 2011, S. 947–952.

[3] Roberto Airoldi, Fabio Garzia, Omer Anjum und Jari Nurmi. "Homogeneous MPSoC as Baseband Signal Processing Engine for OFDM Systems". In: *2010 International Symposium on System-on-Chip (SoC)*. 2010, S. 26–30. DOI: 10.1109/ISSOC.2010.5625562.

[4] Marcos R. de Alba und David R. Kaeli. "Runtime Predictability of Loops". In: *2001 IEEE International Workshop on Workload Characterization*. 2001, S. 91–98.

[5] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, S. 483–485.

[6] Jan Andersson, Jiri Gaisler und Roland Weigand. "Next Generation Multipurpose Microprocessor". In: *International Conference on Data Systems in Aerospace (DASIA 2010)*. 2010.

[7] Giuseppe Ascia, Vincenzo Catania, Maurizio Palesi und Davide Patti. "EPIC-Explorer: A Parameterized VLIW-based Platform Framework for Design Space Exploration". In: *ESTImedia*. Citeseer. 2003, S. 65–72.

[8] Daniel E. Atkins. "Higher-Radix Division Using Estimates of the Divisor and Partial Remainders". In: *IEEE Transactions on Computers* C-17.10, 1968, S. 925–934. ISSN: 0018-9340.

[9] Heikki Berg, Claudio Brunelli und Ulf Lücking. "Analyzing Models of Computation for Software Defined Radio Applications". In: *2008 International Symposium on System-on-Chip (SOC)*. 2008, S. 1–4. DOI: 10.1109/ISSOC.2008.4694886.

[10] Christian Bernard und Fabien Clermidy. "A low-power VLIW processor for 3GPP-LTE Complex Numbers Processing". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*. IEEE, 2011. DOI: 10.1109/DATE.2011.5763048.

[11] Joseph Byrne. "Xtensa Extends Instructions to 128 Bits". In: *Microprocessor Report* 25.5, 2011, S. 20–22.

[12] Cadence. *EDI System User Guide, Product Version 14.12*.

[13] Cadence. *Encounter RTL Compiler Synthesis Flows, Product Version 14.1*.

[14] Loyd Case. "Casting Off the Pipeline". In: *Microprocessor Report* 28.11, 2014, S. 10–12.

[15] Wolfgang Cezanne. *Allgemeine Volkswirtschaftslehre*. Oldenbourg Verlag, 2005.

[16] Alan Clements. *Computer Organization and Architecture: Themes and Variations, First Edition*. Cengage Learning, 2013. ISBN: 978-1111987046.

[17] Lucian Codrescu, Willie Anderson, Suresh Venkumanhanti, Mao Zeng, Erich Plondke. "Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications". In: *IEEE Micro* 34.2, 2014, S. 34–43.

[18] Joan Daemen und Vincent Rijmen. *The design of Rijndael: AES - the Advanced Encryption Standard*. Springer Verlag, 2002.

[19] Erik Dahlman, Stefan Parkvall und Johan Sköld. *4G: Lte/Lte-Advanced for Mobile Broadband*. Academic Press, 2013. ISBN: 0124199852.

[20] Werner Damm. *Entwurf und Verifikation mikroprogrammierter Rechnerarchitekturen*. Springer Verlag, 1987. ISBN: 978-3540183204.

[21] Mike Demler. "Cortex-A12 Fills the Middle". In: *Microprocessor Report* 27.6, 2013, S. 10–12.

[22] Mike Demler. "Cortex-M7 Doubles Up on DSP". In: *Microprocessor Report* 28.10, 2014, S. 22–24.

[23] Mike Demler. "Smallest Warrior gets virtualization". In: *Microprocessor Report* 28.3, 2014, S. 5,22.

[24] Mike Demler. "Xtensa 10 Plays Well With ARM". In: *Microprocessor Report* 27.10, 2013, S. 25–27.

[25] Joseph A. Fisher, Paolo Faraboschi und Clifford Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2004.

[26] Borko Furth. *Encyclopedia of Multimedia*. Springer Verlag, 2008.

[27] Aeroflex Gaisler. *GRLIB IP Core User's Manual*. 2013.

[28] Jiri Gaisler und Edvin Catovic. "Multi-core Processor based on LEON3-FT IP Core (LEON3-FT-MP)". In: *Data Systems in Aerospace (DASIA 2006)*. Bd. 630. 2006, S. 76.

[29] J. Scott Gardner. "Cadence Buys Tensilica for CPU IP". In: *Microprocessor Report* 27.4, 2013, S. 4.

[30] J. Scott Gardner. "Synopsys Build a Better ARC". In: *Microprocessor Report* 26.10, 2012, S. 25–28.

[31] J. Scott Garnder. “MIPS Aptiv Cores hit the Mark”. In: *Microprocessor Report* 26.5, 2012, S. 1,6–10.

[32] J. Scott Garnder. “MIPS Release 5 extends SIMD”. In: *Microprocessor Report* 26.12, 2012, S. 19–22.

[33] Linley Gwennap. “How Cortex-A15 Measures Up”. In: *Microprocessor Report* 27.5, 2013, S. 1,6–11.

[34] Linley Gwennap. “Qualcomm Extends Hexagon DSP”. In: *Microprocessor Report* 27.8, 2013, S. 13–15.

[35] Tom R. Halfhill. “ARM’s 64-Bit Makeover”. In: *Microprocessor Report* 26.12, 2012, S. 23–29.

[36] Tom R. Halfhill. “Cortex-M0+ Simplifies 32-Bit MCUs”. In: *Microprocessor Report* 26.3, 2012, S. 29.

[37] Tom R. Halfhill. “EEMBC’s Dhrystone Killer”. In: *Microprocessor Report* 23.6, 2009, S. 1–5.

[38] Tom R. Halfhill. “Synopsys Accelerates ARC CPUs”. In: *Microprocessor Report* 27.11, 2013, S. 19–23.

[39] Mark D. Hill und Michael R. Marty. “Amdahl’s Law in the Multicore Era”. In: *IEEE Computer* 41.7, 2008, S. 33–38.

[40] Roel Jordans, Rosilde Corvino, Lech Jóźwiak und Henk Corporaal. “Exploring Processor Parallelism: Estimation Methods and Optimization Strategies”. In: *16th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2013)*. 2013.

[41] Thorsten Jungeblut. *Entwurfsraumexploration ressourceneffizienter VLIW-Prozessoren*. Dissertation. Exzellenzcluster Kognitive Interaktionstechnologie (CITEC), Universität Bielefeld, 2011.

[42] Manas Ranjan Kabat. *Design and Analysis of Algorithms*. Prentice-Hall of India, 2013. ISBN: 978-8120348066.

[43] Melanie Kambadur, Kui Tang und Martha A. Kim. “Harmony: Collection and Analysis of Parallel Block Vectors”. In: *ACM SIGARCH Computer Architecture News*. Bd. 40. 3. IEEE, 2012, S. 452–463.

[44] Kevin Krewell. “ARM Opens Up on Cortex-A12”. In: *Microprocessor Report* 27.7, 2013, S. 1,6–8.

[45] Kevin Krewell. “ARM Pairs Cortex-A7 With A15”. In: *Microprocessor Report* 25.11, 2011, S. 23–26.

[46] Kevin Krewell. “Cortex-A53 Is ARM’s Next Little Thing”. In: *Microprocessor Report* 26.11, 2012, S. 9–11.

[47] Erik Larsen und Ronald Aarts. *Audio Bandwidth Extension: Application of Psychoacoustics, Signal Processing and Loudspeaker Design*. John Wiley & Sons, 2004. ISBN: 978-0470858646.

[48] Baiming Liu und Weiwei. "CRC algorithm in computer network communication". In: *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013)*. 2013.

[49] Ivan Llopard, Albert Cohen, Christian Fabre, Jérôme Martin, Henri-Pierre Charles. "Code Generation for an Application-Specific VLIW Processor with Clustered, Addressable Register Files". In: *Proceedings of the 10th Workshop on Optimizations for DSP and Embedded Systems*. ODES '13. ACM, 2013, S. 11–19. DOI: 10.1145/2443608.2443612.

[50] Bruno C. Lopes und Rafael Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014. ISBN: 978-1782166924.

[51] Robert Luk und Robert Damper. "Computational Complexity of a Fast Viterbi Decoding Algorithm for Stochastic Letter-Phoneme Transduction". In: *IEEE Transactions on Speech and Audio Processing* 6.3, 1998, S. 217–225.

[52] Dejan Markovic und Robert W. Brodersen. *DSP Architecture Design Essentials (Electrical Engineering Essentials)*. Springer Verlag, 2012. ISBN: 978-1441996596.

[53] Thomas May, Hermann Rohling und Volker Engels. "Performance Analysis of Viterbi Decoding for 64-DAPSK and 64-QAM Modulated OFDM Signals". In: *IEEE Transactions on Communications* 46.2, 1998, S. 182–190.

[54] Michael Meixner und Tobias G. Noll. "Limits of gate-level power estimation considering real delay effects and glitches". In: *2014 International Symposium on System-on-Chip (SoC)*. 2014, S. 1–7. DOI: 10.1109/ISSOC.2014.6972437.

[55] Mentor Graphics. *ModelSim SE User's Manual, Product Version 10.1*.

[56] Sumit Mohanty, Viktor K. Prasanna, Sandeep K. Neema und James R. Davis. "Rapid Design Space Exploration of Heterogeneous Embedded Systems Using Symbolic Search and Multi-granular Simulation". In: *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems (LCTES/SCOPES '02)*. ACM, 2002, S. 18–27. DOI: 10.1145/513829.513835.

[57] Tipp Moseley, Daniel A. Connors, Dirk Grunwald und Ramesh Peri. "Identifying Potential Parallelism via Loop-centric Profiling". In: *Proceedings of the 4th International Conference on Computing Frontiers (CF '07)*. ACM, 2007, S. 143–152. DOI: 10.1145/1242531.1242554.

[58] Walter Oberschelp und Gottfried Vossen. *Rechneraufbau und Rechnerstrukturen*. Oldenbourg Verlag, 2006.

[59] Thomas Olsson, Anders Carlsson, Leif Wilhelmsson, Johan Eker, Carl von Platen. "A Reconfigurable OFDM Inner Receiver Implemented in the CAL Dataflow Language". In: *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2010, S. 2904–2907. DOI: 10.1109/ISCAS.2010.5538042.

[60] Gianluca Palermo, Cristina Silvano und Vittorio Zaccaria. "Multi-objective Design Space Exploration of Embedded Systems". In: *Journal of Embedded Computing* 1.3, 2005, S. 305–316. ISSN: 1740-4460.

[61] Maurizio Palesi und Tony Givargis. "Multi-objective Design Space Exploration Using Genetic Algorithms". In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES 2002)*. 2002, S. 67–72. DOI: 10.1109/CODES.2002.1003603.

[62] Victor Pan. "Complexity of parallel matrix computations". In: *Theoretical Computer Science* 54.1, 1987, S. 65–85.

[63] Lothar Papula. *Mathematische Formelsammlung: Für Ingenieure und Naturwissenschaftler*. Springer Vieweg, 2014. ISBN: 978-3834819130.

[64] Jongsoo Park, James Balfour und William J. Dally. "Fine-grain Dynamic Instruction Placement for L0 Scratch-pad Memory". In: *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '10)*. ACM, 2010, S. 137–146. DOI: 10.1145/1878921.1878943.

[65] William W. Peterson und David T. Brown. "Cyclic Codes for Error Detection". In: *Proceedings of the IRE* 49.1, 1961, S. 228–235.

[66] Gustav Pomberger und Heinz Dobler. *Algorithmen und Datenstrukturen: Eine systematische Einführung in die Programmierung*. Pearson Studium, 2008. ISBN: 978-3827372680.

[67] Piotr Porwik. "Efficient algorithm of affine form searching for weakly specified Boolean function". In: *Fundamenta Informaticae* 77.3, 2007, S. 277–291.

[68] Dhiraj K. Pradhan und Ian G. Harris. *Practical Design Verification*. Cambridge University Press, 2009. ISBN: 978-0030055386.

[69] Ulrich Ramacher. "Software-Defined Radio Prospects for Multistandard Mobile Phones". In: *Computer* 40.10, 2007, S. 62–69. DOI: 10.1109/MC.2007.362.

[70] Deepak Revanna, Omer Anjum, Manuele Cucchi, Roberto Airoldi und Jari Nurmi. "A scalable FFT Processor Architecture for OFDM Based Communication Systems". In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*. 2013, S. 19–27. DOI: 10.1109/SAMOS.2013.6621101.

[71] Iain Richardson. *The H.264 Advanced Video Compression Standard*. John Wiley & Sons, 2010.

[72] Narayanaswamy Sankarayya, Kaushik Roy und Debashis Bhattacharya. "Algorithms for Low Power and High Speed FIR Filter Realization Using Differential Coefficients". In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 44.6, 1997, S. 488–497. DOI: 10.1109/82.592582.

[73] Debyo Saptono, Vincent Brost, Fan Yang und Eri Prasetyo. "Design Space Exploration for a Custom VLIW Architecture: Direct Photo Printer Hardware Setting Using VEX Compiler". In: *2008 IEEE International Conference on Signal Image Technology and Internet Based Systems*. IEEE, 2008, S. 416–421. DOI: 10.1109/SITIS.2008.69.

[74] Wolfram Schiffmann und Robert Schmitz. *Technische Informatik 2. Grundlagen der Computertechnik*. Springer Verlag, 2005.

[75] Roël Seedorf, Fakhar Anjam, Anthony Brandon und Stephan Wong. "Design of a Pipelined and Parameterized VLIW Processor: ρ-VEX v.2". In: *Proceedings of the 6th HiPEAC Workshop on Reconfigurable Computing*. 2012.

[76] Michael J. S. Smith. *Application-Specific Integrated Circuits*. Addison-Wesley, 2001. ISBN: 0201500221.

[77] Andrew S. Tanenbaum und James Goodman. *Rechnerarchitektur: Von der digitalen Logik zum Parallelrechner*. Pearson Studium, 2014. ISBN: 978-3868942385.

[78] Berthold Vöcking, Helmut Alt, Martin Dietzfelbinger, Rüdiger Reischuk, Christian Scheideler. *Taschenbuch der Algorithmen*. Springer Verlag, 2008. ISBN: 3540763937.

[79] Tim A. Wagner, Vance Maverick, Susan L. Graham und Michael A. Harrison. "Accurate Static Estimators for Program Optimization". In: *SIGPLAN* 29.6, 1994, S. 85–96. DOI: 10.1145/773473.178251.

[80] Ruye Wang. *Introduction to Orthogonal Transforms: With Applications in Data Processing and Analysis*. Cambridge University Press, 2012.

[81] Yaohua Wang, Shuming Chen, Kai Zhang, Hu Chen und Xiaowen Chen. "Architecture Design Trade-offs among VLIW SIMD and Multi-core Schemes". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*. 2012, S. 1649–1658. DOI: 10.1109/IPDPSW.2012.206.

[82] Reihnold P. Weicker. "Dhrystone: A synthetic systems programming benchmark". In: *Communications of the ACM* 27.10, 1984, S. 1013–1030.

[83] Stephan Wong, Thijs van As und Geoffrey Brown. "ρ-VEX: A reconfigurable and extensible softcore VLIW processor". In: *2008 International Conference on Field-Programmable Technology*. IEEE, 2008, S. 369–372. DOI: 10.1109/FPT.2008.4762420.

[84] Vittorio Zaccaria, Mariagiovanna Sami, Donatella Sciuto und Cristina Silvano. *Power Estimation and Optimization Methodologies for VLIW-based Embedded Systems*. Springer Verlag, 2003.

[85] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin und Steve Zdancewic. "Formalizing the LLVM Intermediate Representation for Verified Program Transformations". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, 2012, S. 427–440. DOI: 10.1145/2103656.2103709.

Eigene Arbeiten

[86] Johannes Ax, Boris Hübener, Jan Lachmair und Ronnie Zurmöhle. *Entwicklung einer Prototypingumgebung für Software-Defined Radio Anwendungen*. Projektarbeit. Heinz Nixdorf Institut, Universität Paderborn, 2009.

[87] Boris Hübener. *Analyse und Optimierung der Forwarding-Architekturen eines eingebetteten VLIW-Prozessors*. Diplomarbeit. Heinz Nixdorf Institut, Universität Paderborn, 2010.

[88] Boris Hübener. *Optimierung eines 802.11b Algorithmus für einen VLIW-Prozessor*. Studienarbeit. Heinz Nixdorf Institut, Universität Paderborn, 2009.

[89] Boris Hübener, Gregor Sievers, Thorsten Jungeblut, Mario Porrmann und Ulrich Rückert. "CoreVA: A Configurable Resource-Efficient VLIW Processor Architecture". In: *12th IEEE International Conference on Embedded and Ubiquitous Computing (EUC '14)*. 2014, S. 9–16. DOI: 10.1109/EUC.2014.11.

[90] Thorsten Jungeblut, Boris Hübener, Mario Porrmann und Ulrich Rückert. "A Systematic Approach for Optimized Bypass Configurations for Application-specific Embedded Processors". In: *ACM Transactions on Embedded Computing Systems (TECS) - Special issue on application-specific processors* 13.2, 2013, 18:1–18:25. DOI: 10.1145/2514641.2514645.

[91] Thorsten Jungeblut, Johannes Ax, Gregor Sievers, Boris Hübener, Mario Porrmann. "Resource Efficiency of Scalable Processor Architectures for SDR-based Applications". In: *Proceedings of the Radar, Communication and Measurement Conference (RADCOM)*. 2011.

Betreute Arbeiten

[92] Adriana-Victoria Dreyer. *LTE Signalverarbeitung auf dem CoreVA-Prozessor*. Bachelorarbeit. Exzellenzcluster Kognitive Interaktionstechnologie (CITEC), Universität Bielefeld, 2013.

[93] Philippe Geisler. *Evaluierung des LEON3 Mikroprozessors für die Verwendung in einem Multiprozessorsystem*. Bachelorarbeit. Exzellenzcluster Kognitive Interaktionstechnologie (CITEC), Universität Bielefeld, 2013.

[94] Jonas Gerlach. *QR Code processing on the CoreVA-MPSoC*. Bachelorarbeit. Exzellenzcluster Kognitive Interaktionstechnologie (CITEC), Universität Bielefeld, 2014.

[95] Jan Hendrik Krekeler. *Implementierung einer Hardware/Software-Schnittstelle für das DB-SDR*. Projektarbeit. Heinz Nixdorf Institut, Universität Paderborn, 2012.

[96] Leonardo Sabino dos Santos. *High-level analysis of requirements for upgrading an LTE baseband modem to LTE-Advanced*. Masterarbeit. Heinz Nixdorf Institut, Universität Paderborn, 2013.

[97] Leonardo Sabino dos Santos. *Integration des DB-SDR in die Gnuradio-Umgebung*. Projektarbeit. Heinz Nixdorf Institut, Universität Paderborn, 2012.

Abbildungsverzeichnis

Programmcodeverzeichnis

Tabellenverzeichnis

Instruktionssatz des CoreVA-Prozessors

ALU-Instruktionen	
ABS	Absolutwertberechnung
ADC	Addition mit Übertragbit
ADD	Addition
AND	Logische Und-Verknüpfung
ANDN	Logische Und-Verknüpfung mit Negation des zweiten Operanden
ASR	Arithmetisches Rechtsschieben
CAN	Logische Und-Verknüpfung für Condition-Register
CEO	Logische Exklusiv-Oder-Verknüpfung für Condition-Register
CLZ	Bestimmung führender Nullstellen
CMP	Vergleich zweier Werte
COR	Logische Oder-Verknüpfung für Condition-Register
DEC	Dekrementieren und Vergleichen (für Schleifensteuerung)
DEC4	Dekrementieren und Vergleichen (für vierfach abgerollte Schleifen)
EOR	Logische Exklusiv-Oder-Verknüpfung
EXT	Erweiterung der MVC Instruktion (liefert die oberen 24 Bit eines Worts)
LSL	Logisches Linksschieben
LSR	Logisches Rechtsschieben
MCR	Condition-Register in Datenregister kopieren
MOV	Wort zwischen Datenregistern kopieren
MRC	Datenregister in Condition-Register kopieren
MVAH	Variablen Ausschnitt zwischen Datenregistern kopieren und an oberer 16-Bit-Grenze ausrichten
MVAL	Variablen Ausschnitt zwischen Datenregistern kopieren und an unterer 16-Bit-Grenze ausrichten
MVB	Byte zwischen Datenregistern kopieren
MVC	Konstante in Datenregister schreiben (liefert die unteren 8 Bit eines Worts)
MVH	Halbwort zwischen Datenregistern kopieren
MVSB	Byte zwischen Datenregistern kopieren (Zweierkomplement)
MVSH	Halbwort zwischen Datenregistern kopieren (Zweierkomplement)
NOP	Keine Berechnung ausführen
OR	Logische Oder-Verknüpfung
RSB	Subtraktion mit vertauschten Operanden
RSC	Subtraktion mit vertauschten Operanden mit Übertragbit
SBC	Subtraktion mit Übertragbit
SUB	Subtraktion
LD/ST-Instruktionen	
LDW	Wort aus Speicher lesen
STB	Byte in Speicher schreiben
STH	Halbwort in Speicher schreiben
STW	Wort in Speicher schreiben
MAC-Instruktionen	
MLA	Multiplizieren und Akkumulieren
MLAS	Multiplizieren und Akkumulieren (Zweierkomplement)

DIV Instruktionen	
DVI	Divisionsschrittberechnung initialisieren
DVQ	Quotient der Divisionsschrittberechnung auslesen
DVR	Rest der Divisionsschrittberechnung auslesen
DVS	Divisionsschrittberechnung
BR Instruktionen	
BR	Springen (absolut oder relativ)
BRL	Springen und Rücksprungadresse sichern
SIMD Instruktionen	
VABS	Absolutwertberechnung
VADD	Addition
VASR	Arithmetisches Rechtsschieben
VCMP	Vergleich der einzelnen Halbwörter
VDEC4	Dekrementieren und Vergleichen (für zwei separate Schleifensteuerungen)
VLSL	Logisches Linksschieben
VLSR	Logisches Rechtsschieben
VMLA	Multiplizieren und Akkumulieren
VMLAS	Multiplizieren und Akkumulieren (Zweierkomplement)
VMLS	Multiplizieren und Subtrahieren
VMLSS	Multiplizieren und Subtrahieren (Zweierkomplement)
VPACK	Konvertierung zwischen Skalar und SIMD-Vektoren
VSUB	Subtraktion
VRSB	Subtraktion mit vertauschten Operanden
VSAS	Halbwörter vertauschen und Addieren/Subtrahieren
VSMLA	Multiplizieren und Akkumulieren mit vertauschten Operanden
VSMLAS	Multiplizieren und Akkumulieren mit vertauschten Operanden (Zweierkomplement)
VSMLS	Multiplizieren und Subtrahieren mit vertauschten Operanden
VSMLSS	Multiplizieren und Subtrahieren mit vertauschten Operanden (Zweierkomplement)
VSSA	Halbwörter vertauschen und Subtrahieren/Addieren