TUKUTURI: Eine dynamisch selbstrekonfigurierbare Softcore Prozessorarchitektur

Von der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des akademischen Grades
Doktor-Ingenieur
(abgekürzt: Dr.-Ing.)
genehmigte Dissertation

von Herrn
M. Sc. Florian Giesemann

geboren am 27.03.1985
in Gehrden, Deutschland

2021

1. Referent:	apl. Prof. Dr.-Ing. Guillermo Payá Vayá
2. Referent:	Prof. Dr.-Ing. habil. Michael Hübner
Datum der Promotion:	22.03.2021

Danksagung

Diese Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter im Fachgebiet Architekturen und Systeme des Instituts für Mikroelektronische Systeme der Gottfried Wilhelm Leibniz Universität Hannover.

Mein herzlicher Dank gilt Herrn Prof. Dr.-Ing. Guillermo Payá Vayá für die hervorragende wissenschaftliche Betreuung meiner Arbeit, die anregenden und hilfreichen Diskussionen, die gute Zusammenarbeit in diversen Projekten, sowie für die Übernahme des ersten Referats. Für die Übernahme des zweiten Referats und sein Interesse an meiner Arbeit danke ich Prof. Dr.-Ing. habil. Michael Hübner.

Herrn Prof. Dr.-Ing. Holger Blume danke ich für die Möglichkeit zur Arbeit und Promotion am Institut, die ausgezeichneten Arbeitsbedingungen, die spannenden Projekte und die wissenschaftliche Betreuung, sowie für die Übernahme des Vorsitzes der Prüfungskommission.

Des Weiteren möchte ich mich bei meinen ehemaligen Kollegen am Institut für Mikroelektronische Systeme für die langjährige, sehr gute Zusammenarbeit und die angenehme Arbeitsumgebung bedanken. Besonderer Dank gilt Dipl.-Ing. Lukas Gerlach für die fachlichen Diskussionen, die wertvollen Anregungen und die gemeinsamen Debugging-Sessions. Ich danke M. Sc. Fabian Stuckmann und M. Sc. Moritz Weißbrich für die gründliche Durchsicht des Manuskripts und ihre wertvollen Anmerkungen.

Meinen Eltern und meiner Familie danke ich ganz herzlich für ihre Unterstützung und ermunternden Worte auf meinem Weg zur Promotion.

Stadthagen, 26. März 2021

Florian Giesemann

Kurzfassung

Der Entwurf von Systemen zur digitalen Signalverarbeitung stellt den Entwickler vor stetig wachsende Herausforderungen, die durch zunehmende Komplexität von Anwendungen und die dafür benötigte Steigerung der Leistungsfähigkeit eingebetteter Systeme verursacht werden. Ein weiterer Aspekt neben der Leistungsfähigkeit ist die Flexibilität, die es erlaubt, Anwendungen und Algorithmen auch nach Auslieferung eines Systems zu verändern. Diese kann zum einen durch Verwendung von FPGAs erreicht werden, die eine Rekonfiguration der Hardware ermöglichen. Zum anderen können prozessorbasierte Systeme verwendet werden, die Flexibilität durch Programmierbarkeit bereitstellen. Anwendungsspezifische Anpassungen der Prozessorarchitektur und ein hohes Maß an paralleler Datenverarbeitung, beispielsweise durch VLIW-Prozessoren, stellen dabei Mittel zum Erreichen hoher Leistungen dar.

Das Thema dieser Arbeit ist die Untersuchung eines Entwurfsprozesses für anwendungsspezifische Prozessorsysteme. Dieser basiert auf einem flexiblen SIMD-VLIW-Prozessor, der in großem Umfang konfiguriert und durch zusätzliche Hardwaremodule erweitert werden kann. Zur Exploration des Entwurfsraums werden Werkzeuge zur Analyse von Prozessorkonfigurationen in realen Anwendungen bereitgestellt sowie Methoden zur automatisierten Adaption der Architektur auf Basis dieser Analyseergebnisse untersucht. Die Kompilierung von Anwendungen für VLIW-Architekturen wird aufgrund der kombinatorischen Komplexität üblicherweise mittels statischer Heuristiken durchgeführt, wodurch eine optimale Adaption an flexible Architekturen erschwert werden kann. Daher werden hier dynamische Methoden zur Codegenerierung, die auf evolutionären Algorithmen basieren, untersucht. Die Umsetzung der Architektur als Softcore auf einem FPGA bietet zusätzlich die Möglichkeit der dynamischen Adaption der Hardware zur Laufzeit. Diese Möglichkeiten und deren Einfluss auf die Leistungsfähigkeit der Prozessorsysteme werden ebenfalls untersucht.

Die Analyse des Entwurfsprozesses in einer exemplarischen Anwendung der bildbasierten Objekterkennung und der Vergleich mit Implementierungen auf einem MIPS-Softcore bzw. VLIW-DSP zeigen die Eignung der Methoden zur Adaption von Softcore-Prozessoren und der EA-basierten Kompilierung von Anwendungen. Die dynamische Hardwarerekonfiguration zur Laufzeit kann bei reduziertem Flächenbedarf für die Hardware ohne Leistungsverlust eingesetzt werden.

Schlagworte: Entwurfsraumexploration, Codegenerierung, dynamische Rekonfiguration

Abstract

The design of digital signal processing systems poses increasingly difficult challenges caused by the ever-growing application complexity and the resulting demand on processing power. Another aspect besides performance is flexibility, which allows applications and algorithms to be changed even after a system has been delivered. On the one hand, this can be achieved by using FPGAs that allow reconfiguration of the hardware. On the other hand, processor-based systems provide flexibility through programmability. Application-specific adaption of processor architectures and a high degree of parallel data processing, e.g., in VLIW processors, provide the means to achieve high performance.

The topic of this thesis is the analysis of a design process for application-specific processor systems. It is based on a flexible SIMD-VLIW processor template that allows configuration on a large scale and can be extended with additional hardware modules. In order to assist design space exploration, tools for the analysis of processor configurations in real applications are provided and methods for the automated adaptation of the processor architecture based on analysis results are investigated. Due to combinatorial complexity, the compilation of applications for VLIW architectures is usually guided by static heuristics, which can prevent optimal adaption to flexible architectures. Therefore, in this thesis, dynamic methods based on evolutionary algorithms are examined for their use in code generation. The implementation of architectures as softcore processors on an FPGA also offers the possibility of dynamic adaptation of the hardware at runtime. These possibilities and their influence on the performance of processor systems are also examined.

Analysis and evaluation of the design process with an exemplary application from the field of image-based object detection and the comparison with implementations on a MIPS-based softcore processor and a VLIW-DSP show the suitability of the methods for the adaptation of softcore processors and the EA-based compilation of applications. The dynamic hardware reconfiguration during runtime can be used to reduce hardware area requirements without loss of performance.

Key words: design space exploration, code generation, dynamic reconfiguration

Inhaltsverzeichnis

Abkürzungsverzeichnis

ADL Architecture Description Language

ALU Arithmetic Logic Unit

ASIC Application-Specific Integrated Circuit

ASIP Application-Specific Instruction-Set Processor

CLB Configurable Logic Block

DFG Datenflussgraph

DLP Data-Level Parallelism

DPR dynamische partielle Rekonfiguration

DSP Digital Signal Processor

DtB Distance-to-Border

EA Evolutionärer Algorithmus

EDA Electronic Design Automation

FIR File Indirect Register

FPGA Field Programmable Gate Array

FPS Frames Per Second

FPU Floating Point Unit

FSM Final State Machine

FU funktionale Einheit

GA Genetischer Algorithmus

GPP General Purpose Processor

HDL Hardware Description Language

HLS High-Level Synthese

ICAP Internal Configuration Access Port

ILP Instruction-Level Parallelism

IPC Instructions Per Cycle

IR Intermediate Representation

ISA Instruction Set Architecture

LUT Look-Up Table

MO Mikrooperation

MOEA Multi-Objective Evolutionary Algorithm

PRR Partially Reconfigurable Region

RA Registerallokation

RCU Reconfigurable Coprocessor Unit

RFU rekonfigurierbare funktionale Einheit

RISC Reduced Instruction Set Computer

RPU Reconfigurable Processing Unit

SIMD Single Instruction Multiple Data

SoC System on Chip

SVM Support Vector Machine

VLIW Very Long Instruction Word

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

1 Einleitung

Systeme zur digitalen Signalverarbeitung sind heutzutage allgegenwärtig. Anwendungsgebiete umfassen neben mobilen Applikationen auch eingebettete Systeme zur Verarbeitung medizinischer Daten sowie zur Automatisierung von Produktionsprozessen und zunehmend auch Assistenzsysteme in Fahrzeugen [6], die in dieser Arbeit als Beispielanwendungsgebiet verwendet werden. Diese Assistenzsysteme dienen nicht nur dem Komfort des Fahrers, sondern können auch Warn- und Steuerfunktionen zur Erhöhung der Sicherheit von Verkehrsteilnehmern übernehmen. Verschiedene Richtlinien schreiben den Einsatz von Assistenzsystemen wie Spurhalteassistenten und automatischen Notbremsassistenten für zukünftige Fahrzeuge vor (z. B. EU-Verordnung (EU) 2019/2144). Um die Entscheidung für Steuereingriffe treffen zu können, muss die Fahrsituation und die Umgebung bekannt sein. Dazu werden unter Umständen eine Vielzahl von Sensorsystemen wie Kameras, Ultraschallsensoren, Radarsysteme, Lidar, usw. verwendet, die große Datenmengen produzieren. Gleichzeitig steigt die Komplexität der Datenverarbeitung, um Abstände und Positionen von Personen und anderen Verkehrsteilnehmern zuverlässig zu bestimmen. Verschiedene algorithmische Techniken zur Bildanalyse, des Computersehens und des maschinellen Lernens [31] tragen zum gesteigerten Bedarf an leistungsfähigen eingebetteten Rechensystemen bei.

Neben der Leistungsfähigkeit bzw. Effizienz der Systeme spielt auch die Flexibilität der Systeme eine große Rolle, die es erlaubt, die Anwendungen und Algorithmen zum Teil nach Auslieferung noch zu verändern und an neue Anforderungen anzupassen. Diese Flexibilität wird durch Verwendung programmierbarer Systeme erreicht, die Prozessoren zur Abarbeitung von kompiliertem Code enthalten. Mehrzweckprozessoren (General Purpose Processor (GPP)) bieten einen universellen Instruktionssatz und können für beinahe jede Anwendung eingesetzt werden, bieten häufig allerdings nicht die erforderliche Leistung bzw. Effizienz. Die Effizienz und Leistungsfähigkeit eines Prozessorsystems kann durch Spezialisierung gesteigert werden. Anwendungsspezifische Prozessoren (Application-Specific Instruction-Set Processor (ASIP)) sind programmierbare Prozessoren, die an eine spezifische Anwendung oder ein ganzes Anwendungsfeld (z. B. Bildverarbeitung) angepasst werden. Eine solche Anpassung kann beispielsweise durch Erweiterung des Befehlssatzes um anwendungsspezifische Operationen in Form von dedizierten Hardwareerweiterungen geschehen, wodurch die Vorteile der Flexibilität eines programmierbaren Prozessors mit der Leistungsfähigkeit und

Effizienz anwendungsspezifischer integrierter Schaltungen (Application-Specific Integrated Circuit (ASIC)) zu einem gewissen Grad kombiniert werden können. Oftmals erfolgt die Spezialisierung eines Prozessors durch Konfiguration einer bestehenden parametrisierten Architektur, die verschiedene Optionen zur Einstellung von Hardwareparametern und die Auswahl aus bereitgestellten Hardwareeinheiten erlaubt. Die Konfiguration der Architektur erfolgt statisch, bevor der Prozessor implementiert wird. Eine Implementierung eines konfigurierten Prozessors in einem ASIC ist mit hohen Kosten verbunden und daher nur für hohe Stückzahlen lohnend. FPGAs (Field Programmable Gate Array) bieten hier eine kostengünstige Alternative bei leicht geminderter Leistung und stellen zusätzlich Flexibilität auf einer weiteren Ebene bereit, indem sie die Rekonfiguration einer Prozessorarchitektur im Einsatzgebiet ermöglichen. Dadurch kann ein System an geänderte Anforderungen angepasst werden, indem eine neue statische Konfiguration einer Prozessorarchitektur erstellt wird.

Moderne FPGAs bieten weiterhin die Möglichkeit zur dynamischen Rekonfiguration [75], die eine Veränderung des abgebildeten Systems zur Laufzeit ermöglicht. Damit kann eine Anpassung der Prozessorarchitektur in gewissen Rahmen sogar an zeitlich- oder situationsbedingte Änderungen der Anforderungen einer Anwendung angepasst werden. Wird die Rekonfiguration des Prozessors von der Anwendung selbst, also bereits zur Kompilierungszeit, geplant, kann die Anpassung an nicht vorhergesehene Situationen zur Laufzeit nicht optimal vorgenommen werden. Daher ist ein System zur adaptiven dynamischen Rekonfiguration zur Laufzeit wünschenswert, das die momentane Rechenlast des Prozessors analysiert und gegebenenfalls eine dynamische Rekonfiguration auslöst, um den aktuellen Bedarf an Hardwareressourcen besser zu befriedigen.

Prozessorbasierte Systeme müssen programmiert werden. Dies kann auf unterschiedlichen Abstraktionsebenen geschehen, die von hardwarenahem Assemblercode bis zu abstrakten Hochsprachen reichen. Die Übersetzung des Programmtextes in ausführbaren Code wird durch einen Kompiler übernommen, dessen Leistung einen nicht unerheblichen Einfluss auf die Performanz der Anwendung hat. Insbesondere das Kompiler-Backend, in dem vor der Generierung des Binärcodes verschiedene Optimierungen am Code vorgenommen werden, spielt hierbei eine große Rolle.

1.1 Ziele der Arbeit

In dieser Arbeit soll die Möglichkeit zur Verwendung einer flexiblen, parametrisierten Architekturvorlage eines Prozessors in einer applikationsspezifischen Implementierung als Softcore auf einem FPGA untersucht werden. Zu diesem Zweck wird ein grundlegender Toolflow erstellt, mit dem die Konfiguration und die Analyse des Prozessors für Anwendungen vorgenommen werden kann.

Die Architekturvorlage stellt einen VLIW-SIMD-Prozessor bereit, der einen hohen Grad an Parallelität auf Instruktions- und auf Datenebene bietet und sich damit für intensive Datenverarbeitungsanwendungen, zum Beispiel in der Bildverarbeitung, eignet. Die Vorlage lässt sich feingranular und umfassend konfigurieren, um den Prozessor ideal an die Anwendung anpassen zu können.

Eine wichtige Komponente des Toolflows ist der Kompiler, mit dem Programme für die adaptierte Prozessorarchitektur übersetzt werden können. Gerade für VLIW-Architekturen spielen der Kompiler und die Codeoptimierungen im Backend eine wichtige Rolle, da das Instruction Scheduling, also das Parallelisieren der Operationen, vom Kompiler übernommen wird und nicht vom Prozessor zur Laufzeit durchgeführt wird. Die Aufgaben der Codegenerierung, also Instruction Scheduling, Registerallokation und Codeselektion, sind im Allgemeinen NP-vollständig [41, 50, 57], so dass optimale Lösungen üblicherweise nicht direkt berechnet werden können. Stattdessen werden Heuristiken eingesetzt, die den Schedulingalgorithmus lenken sollen, indem Codeeigenschaften geschätzt werden. Diese Heuristiken sind meist statisch und die Fähigkeit zur Adaption an bestimmte Situationen oder Gegebenheiten im Anwendungscode muss von den Entwicklern vorhergesehen werden. Weitere Aufgaben des Kompilerbackends sind die Allokation der Register und die Optimierung des Anwendungscodes durch Veränderung der verwendeten Operationen. Diese Schritte können nicht vollständig unabhängig durchgeführt werden, da sie unterschiedliche Einflüsse aufeinander ausüben. In dieser Arbeit soll untersucht werden, wie der Einsatz dynamischer Heuristiken auf Basis von evolutionären Algorithmen im Backend eines Kompilers zu einem besseren Binärprogramm führen kann, das weniger Taktzyklen benötigt als ein Programm, das mit statischen Heuristiken kompiliert wurde. Das Backend führt das Instruction Scheduling, die Registerallokation und Codeoptimierungen in kombinierter Form durch, was die Komplexität der Codegenerierung weiter erhöht, aber durch die Interaktion der verschiedenen Stufen auch bessere Ergebnisse liefern könnte.

Die Flexibilität der Architekturvorlage des Prozessors ist durch eine hohe Anzahl möglicher Parameter gegeben, wodurch die Exploration des Entwurfsraums aufwendig ist. Die Adaption des Prozessors wird durch den Toolflow insofern unterstützt, dass zunächst der Kompiler, der den Anwendungscode analysiert, eine auf die Anwendung maßgeschneiderte Minimalkonfiguration liefern kann, die minimalen Ressourcenaufwand bedeutet. Anschließend können Laufzeitanalysen (Profiling) der Anwendung aus einer Emulation des Prozessorsystems Informationen liefern, die eine unterstützte bzw. teilweise automatisierte Adaption des Prozessors ermöglicht, um maximale Taktfrequenz zu erreichen. Zu diesem Zweck soll der Einsatz des NSGA-II Algorithmus zur Optimierung der Prozessorkonfiguration untersucht werden.

Neben der statischen Konfiguration der Prozessorarchitektur kann bei der Verwendung als Softcore auf einem FPGA von der dynamischen partiellen Rekon-

figuration Gebrauch gemacht werden, um die Adaption des Prozessors dynamisch zur Laufzeit an die zeitlich veränderten Anforderungen einer Anwendung anzupassen. In dieser Arbeit sollen diese Möglichkeiten zur dynamischen Adaption und die Auswirkungen auf die Performanz einer Prozessorarchitektur untersucht werden.

1.2 Aufbau der Arbeit

Diese Arbeit ist wie folgt gegliedert: Kapitel 2 präsentiert Grundlagen von applikationsspezifischen Prozessoren, deren Designprozessen und Implementierungsmöglichkeiten. Weiterhin werden Grundlagen der Kompilertechnologie, insbesondere für Kompilerbackends, und die Codegenerierung für VLIW-Prozessoren beschrieben. Zu beiden Themenkomplexen wird ein Überblick über den aktuellen Stand der Technik gegeben. Kapitel 3 stellt die flexible Architekturvorlage für den Softprozessor vor und erklärt ausführlich die zur Verfügung stehenden Funktionen und Parameter. Es wird auf die Möglichkeiten zur Erweiterung des Befehlssatzes sowie die statische und dynamische Konfiguration der Prozessorarchitektur eingegangen. In Kapitel 4 werden die Kompilerstufen des Backends vorgestellt und die Unterschiede von statischen und dynamischen Heuristiken anhand einer exemplarischen Anwendung der Bildverarbeitung untersucht. Kapitel 5 gibt dann einen Überblick über den gesamten Toolflow zur Adaption der Prozessorvorlage an eine Anwendung, zur Laufzeitanalyse, unterstützten Anwendungsentwicklung und automatischen Adaption der Prozessorarchitektur. Kapitel 6 zeigt die Anwendung des Toolflows auf eine exemplarische Verkehrszeichendetektion. Zunächst werden die Algorithmen der Anwendung vorgestellt, anschließend werden Implementierungen auf verschiedenen Prozessorimplementierungen vorgestellt und analysiert. Zum Vergleich werden Implementierungen auf einem MIPS Softcore und einem TI DSP untersucht. Kapitel 7 gibt einen Überblick über mögliche Erweiterungen bzw. Fortführungen dieser Untersuchungen und Kapitel 8 schließt die Arbeit mit einer Zusammenfassung ab.

2 Grundlagen und Stand der Forschung

Zur Umsetzung von Anwendungen mit hohem Rechenaufwand werden Prozessoren an diese Anwendungen angepasst, wie im nächsten Abschnitt erläutert. Die Adaption der Prozessoren macht die Verwendung von flexiblen Kompilern erforderlich, um den Code für die Anwendung für den Prozessor umzusetzen, wie im zweiten Abschnitt dieses Kapitels erläutert. Der dritte Abschnitt erläutert Grundlagen evolutionärer Algorithmen, die in dieser Arbeit zur Optimierung einer Prozessorarchitektur und des dazugehörigen Kompilers eingesetzt werden.

2.1 Applikationsspezifische Instruktionssatz-Prozessoren

Eingebettete Systeme werden heutzutage in einem weiten Anwendungsfeld eingesetzt. Dies umfasst unter anderem Multimediageräte, Sensornetze, Steuersysteme in Fahrzeugen oder industriellen Anlagen, sowie bildgebende medizinische Systeme. Sie verarbeiten teils große Mengen an Daten, die von einer Vielzahl verschiedener Sensoren, wie Kameras, Ultraschall, Radar, u. v. m., geliefert werden. Die steigende Komplexität von Algorithmen und wachsende Datenmengen sorgen für höhere Anforderungen an die Rechenleistung der Systeme. Gleichzeitig unterliegen die Rechenplattformen verschiedenen Beschränkungen, z. B. im Bezug auf Leistungsaufnahme, Größe oder Portabilität. Zusätzliche Erschwernisse entstehen durch Niedrigpreisanforderungen und Forderungen nach kurzen Entwicklungszeiten (time-to-market).

Anforderungen an Rechenplattformen sind neben der *Leistungsfähigkeit* bzw. *Effizienz*, also der Rechenleistung im Bezug zu Flächenbedarf oder Leistungsaufnahme, außerdem die *Flexibilität*, die den Aufwand zur Umsetzung oder Änderung einer Implementierung auf der Plattform bewertet. Verschiedene Implementierungsvarianten für eingebettete Systeme unterscheiden sich hinsichtlich dieser Anforderungen, wie in Abbildung 2.1 qualitativ dargestellt. Auf der einen Seite besitzen *General Purpose Processors (GPPs)* aufgrund ihrer Programmierbarkeit eine hohe Flexibilität, wodurch Aktualisierungen und Anpassungen an geänderte Anforderungen der Anwendungen durch neue Software möglich werden. Auf der anderen Seite bieten *Application-Specific Integrated Circuits (ASICs)* durch Spezialisierung auf die Anwendung eine hohe Effizienz, haben allerdings hohe Initialkosten und bieten keine Flexibilität, da sie zur Anpassung an veränderte Anforderungen aufwendig und teuer neu entworfen und gefertigt werden müssen.

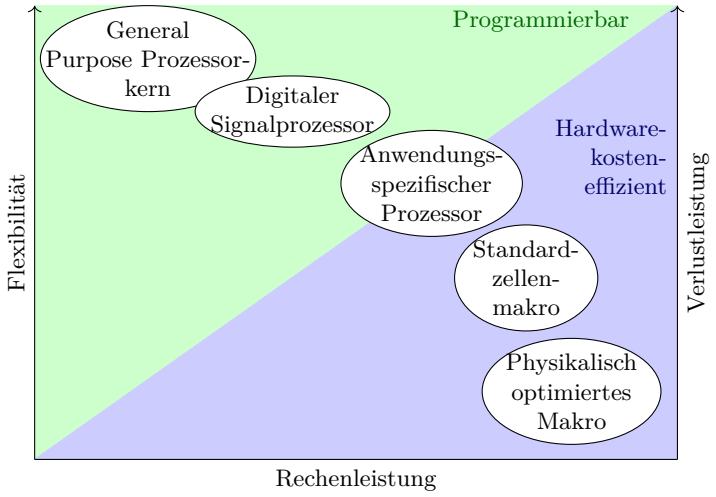


Abbildung 2.1: Qualitativer Vergleich verschiedener Hardwarearchitekturen bezüglich Flexibilität, Rechenleistung und Verlustleistung [69]

Eine Möglichkeit, die Vorteile dieser beiden Implementierungsvarianten zu kombinieren, stellen *Application-Specific Instruction-Set Processors (ASIPs)* dar. Ein ASIP ist ein Prozessor, der durch die Programmierbarkeit eine hohe Flexibilität bietet und der spezifisch an eine Anwendung oder eine Klasse von Anwendungen angepasst wird, wodurch diese effizienter ausgeführt werden kann. Die Anpassung des Prozessors erfolgt einerseits durch Spezialisierung verschiedener Parameter, wie der Wortbreiten im Datenpfad oder der Speicherhierarchie, andererseits durch Erweiterung des Instruktionssatzes um anwendungsspezifische Operationen und Beschleuniger, die als dedizierte Hardwaremodule realisiert werden und damit die Effizienz von ASICs bereitstellen.

2.1.1 ASIP-Designprozess

Der grundlegende Ablauf bei der Entwicklung eines ASIP für eine spezifische Anwendung ist nach [48] in zwei Phasen aufgeteilt, wie in Abbildung 2.2 dargestellt.

In der *Anwendungsanalyse* wird zunächst ein *Profiling* der Anwendung durchgeführt, um die am häufigsten ausgeführten und berechnungsintensivsten Programmteile (*hot-spots*) zu identifizieren. Aus der Analyse der hot-spots wird eine erste Mikroarchitektur für den ASIP ausgewählt und die Parameter, wie z. B. Wortbreiten und Speicherlayout, für eine effiziente Verarbeitung der hot-spots angepasst. Außerdem wird, z. B. aus der Auswertung der Häufigkeit einzelner Operationen, ein Basisinstruktionssatz sowie eine Menge von Erweiterungen und Beschleunigern abgeleitet.

In der *Optimierung* wird diese erste Architekturspezifikation schrittweise

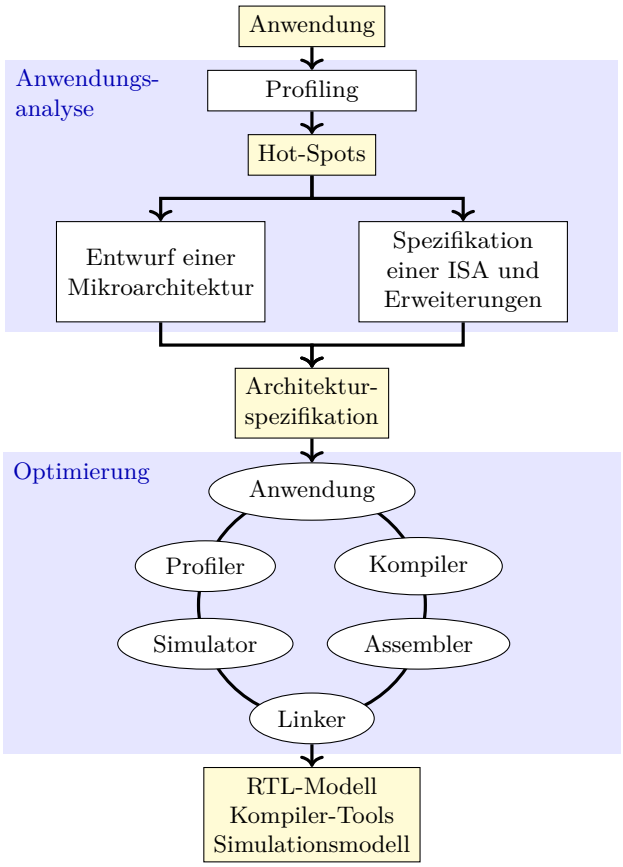


Abbildung 2.2: Grundlegender Ablauf des ASIP-Entwurfs nach [48]

optimiert. In einem iterativen Prozess werden inkrementelle Modifikationen an der Architektur vorgenommen und durch Simulation oder Emulation verifiziert und evaluiert. Modifikationen, die zu einer effizienteren Ausführung führen, werden in die Architektur integriert, und der Prozess wird wiederholt, bis alle Entwurfsziele erreicht werden.

Die hohe Komplexität des Entwurfsraums, verursacht durch die Vielzahl an Parametern der Architektur, macht eine vollständige Auswertung aller Parameterkombinationen unmöglich. Der Entwurfsraum wird daher beispielsweise durch Erfahrung der Entwickler oder Heuristiken eingeschränkt, wodurch sich im ungünstigen Fall die Zahl der Iterationen in der Optimierungsphase erhöht und der Optimierungsprozess verlängert wird. Der Einsatz geeigneter Entwurfswerkzeuge (Electronic Design Automation (EDA)) unterstützt den Entwickler nicht nur bei der Anwendungsanalyse und der Konfiguration einer Architektur, sondern kann auch die Entwurfsraumexploration und die Evaluation der inkrementellen Modifikationen unterstützen, indem Softwaretools (wie Kompiler und Linker)

mit Anpassungen an die modifizierte Architektur teilweise automatisch generiert werden.

Entwurfsmethoden von ASIPs werden in [48] in zwei Klassen unterteilt. In einem *specification-based design flow* wird ein ASIP aus einer Spezifikation des Verhaltens und der Funktionalität von Grund auf neu entwickelt. Die Spezifikation erfolgt in einer Form, die Details der Hardwareimplementierung abstrahiert und damit den Fokus stärker auf die Architekturbeschreibung legt, beispielsweise durch Beschreibungssprachen. Softwaretools der Entwicklungsumgebung übersetzen diese abstrakten Modelle in eine Hardwareimplementierung, z. B. ein RTL-Modell, und können teilweise die dazugehörige Software-Infrastruktur wie Kompiler und Simulatoren generieren. Im Laufe der Zeit sind verschiedene Formen der Spezifikation entwickelt worden, die Beschreibungen auf unterschiedlichen Abstraktionsebenen, z. B. auf Logikebene (Gatter) oder RTL-Ebene (Hardware Description Language (HDL), wie Verilog oder VHDL) ermöglichen. Die Beschreibung auf derart niedriger Abstraktionsebene kann sehr komplex sein und macht u. U. die automatische Generierung von Softwaretools schwierig. Heutzutage werden häufig Architecture Description Languages (ADLs) oder High-Level Synthese (HLS) eingesetzt. Eine ADL erlaubt zum einen die Beschreibung des Instruktionssatzes (Instruction Set Architecture (ISA)), also u. a. die Kodierung, Semantik und Latenzen der Operationen und zum anderen die Beschreibung der Mikroarchitektur, also des Speicherlayouts, der Pipeline, Registerbänke, und des Verhaltens der Operationen. Diese Abstraktionsebene eignet sich nicht nur zur Hardwarebeschreibung, sondern erlaubt auch die Generierung einer Software-Infrastruktur, wie Kompiler, Assembler, Linker, Instruktionssatzsimulator und Profiler. Eine High-Level Synthese verwendet eine Hochsprachenimplementierung einer Anwendung, beispielsweise in C++, um automatisiert eine Hardwarearchitektur für die Anwendung oder einzelne Bestandteile davon zu erzeugen. Eine Untersuchung HLS-basierter Implementierungen einer exemplarischen Anwendung zur Bewegungsschätzung in [86] zeigt, dass zur Generierung effizienter Hardware Anpassungen am Referenzcode der Anwendung vorgenommen werden müssen und der Ressourcenbedarf höher ausfällt als für eine manuelle Implementierung. Für Teile einer Anwendung, die größtenteils reguläre Datenverarbeitung durchführen, können akzeptable Ergebnisse mit annehmbarem Aufwand erzielt werden. Von Kontrollfluss dominierte Programmabschnitte erfordern größeren Aufwand und können häufig nur schlecht abgebildet werden.

Die spezifikationsbasierte Entwurfsmethode mit ADLs ist beispielsweise im EXPRESSION Projekt [34] umgesetzt worden. Die EXPRESSION ADL beschreibt Verhalten und Struktur eines Prozessors und erlaubt die Generierung eines C-Kompilers und eines zyklengenauen Simulators. Kommerzielle Umgebungen sind beispielsweise der CoWare Processor Designer, der auf der LISA (Language for Instruction Set Architectures) ADL basierte und mittlerweile ebenso von Synopsys übernommen wurde, wie die Firma Target Kompiler Technologies mit ihrem

auf der nML basierenden Entwurfswerkzeug. Der Synopsys ASIP Designer [98] verwendet die nML ADL, die viele Ähnlichkeiten zu LISA aufweist. Die Firma Codasip bietet mit CodAL eine Modellierungssprache und eine Entwicklungsumgebung zum spezifikationsbasierten ASIP-Entwurf und zusätzlich in CodAL beschriebene RISC-V Prozessormodelle, die an eine Anwendung angepasst werden können [19].

Im Gegensatz zum Entwurf durch Spezifikation erlaubt der *configuration-based design flow* die Konfiguration einer bestehenden Prozessorarchitektur zur Anpassung an eine Anwendung. Dem Entwickler wird ein vorab entwickelter und verifizierter *Basisprozessor* zur Verfügung gestellt, der bis zu einem gewissen Grad an die Anforderungen der Anwendung angepasst werden kann. Konfigurationsmöglichkeiten umfassen häufig die Auswahl vorgegebener funktionaler Einheiten (z. B. Dividierer, Floating Point Units (FPUs)), Operationsmodi (z. B. SIMD), Konfiguration des Speicherlayouts (Hierarchie, Caches), der Registerbänke und externer Schnittstellen (Bus, FIFOs). Sie können dem Entwickler außerdem erlauben, selbst definierte Erweiterungen des Instruktionssatzes zu integrieren. Die verschiedenen Optionen zur Konfiguration sind typischerweise vom Toolhersteller verifiziert, so dass der Entwickler nur die eigenen Erweiterungen verifizieren muss, wodurch die Entwicklung vereinfacht wird. Durch die eingeschränkte Flexibilität im Entwurf ist aber auch die Möglichkeit zur Entwurfsraumexploration eingeschränkt. Beispiele für den konfigurationsbasierten ASIP-Entwurf sind zum einen die Tensilica Customizable Processors [13], die in einer grafischen Entwicklungsumgebung konfiguriert und durch angepasste Instruktionen erweitert werden können. Ein weiteres Beispiel ist die CorExtend-Technologie [22] für MIPS-Prozessoren, die ebenfalls die Erweiterung des Instruktionssatzes erlaubt.

2.1.2 ASIP-Implementierung

Nach dem Entwurf eines ASIP kann dieser auf verschiedene Arten realisiert werden. Gängig ist eine Implementierung als *ASIC*, beispielsweise von Tensilica-Prozessoren [13]. Eine solche Implementierung kann günstig bezüglich Chip-Fläche und Leistungsaufnahme sein, allerdings werden viele Parameter des Systems fixiert, z. B. die Breite des Datenbusses, Speichergrößen, Konfiguration der Registerbänke usw., wodurch die Flexibilität des Systems wieder reduziert wird. Moderne Anwendungen und die verwendeten Algorithmen sind einer ständigen Weiterentwicklung unterworfen. Bei komplexen Anwendungen oder einem weiten Anwendungsgebiet des ASIP können deshalb nicht alle zukünftigen Anforderungen abgeschätzt und berücksichtigt werden, weshalb eine hohe Flexibilität auch für eine Hardwarerealisierung wünschenswert ist.

Um die Flexibilität einer Implementierung zu erhalten, können rekonfigurierbare Strukturen eingesetzt werden, die auch nach einer Implementierung veränderlich sind. Die Rekonfiguration der Hardware ist mit deutlich weniger

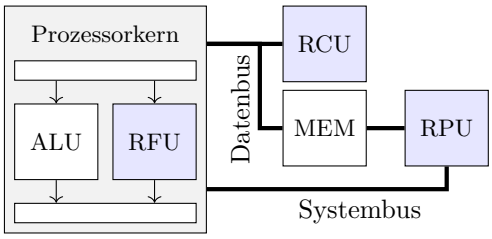


Abbildung 2.3: Verschiedene Arten der Kopplung eines Prozessorkerns mit rekonfigurierbaren Einheiten (in blau)

Aufwand und Kosten verbunden als ein erneutes Design und Implementierung eines ASIC.

Die Art der Kopplung von statischen und rekonfigurierbaren Bestandteilen eines Systems, beispielsweise des statischen Prozessorkerns als ASIC mit rekonfigurierbaren Erweiterungen in einem eFPGA, hat Einfluss auf die Kommunikation zwischen den Systemkomponenten. Es werden drei verschiedene Klassen der Kopplung unterschieden [65], die schematisch in Abbildung 2.3 dargestellt sind.

- Eine *attached Reconfigurable Processing Unit (RPU)* wird über den Systembus mit dem Prozessorkern verbunden und führt die Datenverarbeitung unabhängig vom Prozessor durch. Aufgrund der geringen Datenübertragungsrate des Systembusses eignet sich diese Anbindung nur für Fälle, in denen wenig Kommunikation zwischen der RPU und dem Prozessor stattfindet. Deshalb werden typischerweise große, komplexe Hardwarestrukturen als RPUs verwendet, die ganze Teilaufgaben übernehmen und möglichst unabhängig vom Prozessor bearbeiten. Die RPU kann für den Datenaustausch auch eine Verbindung zu einem gemeinsamen Speicher haben.

- Eine *Reconfigurable Coprocessor Unit (RCU)* ist auf den Datenbus des Prozessors abgebildet, so wie lokale Speicher oder Caches, oder durch eine ad-hoc Verbindung mit dem Prozessor gekoppelt. Diese Anbindung bietet eine höhere Datenrate als der Systembus, erhöht allerdings auch die Abhängigkeit zwischen Coprozessor und Prozessorkern durch intensivere Kommunikation.

- Eine *rekonfigurierbare funktionale Einheit (RFU)* ist sehr eng mit dem Prozessorkern verbunden, indem sie direkt als funktionale Einheit in die Pipeline integriert wird, beispielsweise zur Realisierung neuer komplexer Operationen zur Erweiterung des Instruktionssatzes. Diese Kopplung erlaubt der RFU unter Umständen den Zugriff auf die Registerbänke und bietet eine hohe Datenrate bei der Kommunikation zwischen Prozessorkern und RFU.

Durch die Rekonfiguration können beispielsweise Erweiterungen des Prozessors ausgetauscht werden, um Aktualisierungen und Anpassungen an geänderte Anforderungen der Anwendung auch auf Hardwareebene zu ermöglichen.

Es wird nach der Granularität der rekonfigurierbaren Einheiten unterschieden [73, 101]. *Feingranulare* (*fine-grained*) Strukturen werden für die Umsetzung komplexer Operationen auf Bit-Ebene verwendet. Derartige Strukturen basieren auf Look-Up Tables (LUTs) und Flipflops, beispielsweise in den Logikblöcken von Field Programmable Gate Arrays (FPGAs). Ein konfigurierbares Routing-Netzwerk erlaubt die flexible Verschaltung der einzelnen Elemente. Im Gegensatz dazu werden *grobgranulare* (*coarse-grained*) Strukturen für Operationen auf Wort-Ebene eingesetzt. Sie bieten typischerweise Arrays von arithmetischen Einheiten (Addierer, Multiplizierer) und auch Speichern. Da die Rekonfiguration auf einer höheren Ebene erfolgt, bieten grobgranulare Strukturen eine eingeschränkte Flexibilität. Die Zeit für eine Rekonfiguration sinkt aufgrund der geringeren Menge an rekonfigurierbaren Parametern. Die Elemente in grobgranularen Arrays sind häufig mit ihren direkten Nachbarn verbunden. Der gegenüber den feingranularen Strukturen verringerte Routingaufwand kann zu einer effizienteren Abbildung einer Rechenstruktur führen, da die Logikdichte größer wird. Diese beiden Paradigmen lassen sich auch kombinieren, wie Veröffentlichungen zu *FPGA Overlay-Architekturen* zeigen. Dabei wird eine grobgranulare Architektur, die für eine Klasse von Anwendungen entworfen ist und die Umsetzung verschiedener Anwendungen erlaubt, auf einem FPGA abgebildet [15, 45]. Diese Struktur verringert die Zeit für die Rekonfiguration signifikant [94], da weniger rekonfigurierbare Elemente enthalten sind, verringert aber auch die Flexibilität des FPGA, da nicht mehr auf Bitebene rekonfiguriert wird. Durch Verwendung partieller Rekonfiguration kann zwischen verschiedenen Overlays gewechselt werden, um die Flexibilität zu erhöhen [21].

Neben der Implementierung des Prozessors als ASIC gibt es auch die Möglichkeit, den Prozessor selbst auf einer rekonfigurierbaren Struktur, z. B. einem FPGA, zu implementieren. Dabei kann auch dieser Softcore-Prozessor zusätzliche Beschleuniger mit den oben beschriebenen Kopplungsmechanismen verwenden, die dann ebenfalls auf dem FPGA abgebildet werden. Die Konfiguration des FPGA erfolgt statisch vor der Inbetriebnahme des aus Softcore-Prozessor und Erweiterungen bestehenden Systems. Zur Anpassung an geänderte Anforderungen der Anwendung kann das FPGA mit einem adaptierten System rekonfiguriert werden, wobei der Betrieb unterbrochen wird (statische Rekonfiguration).

Untersuchungen in [10] zeigen, dass komplexe Anwendungen mehrere verschiedene rechenintensive Abschnitte (*hot-spots*) besitzen. Wird beim ASIP-Entwurf die Prozessorarchitektur für alle hot-spots erweitert, kann dies zu einer großen Zahl von Erweiterungen mit entsprechend großem Flächenbedarf führen. In einer FPGA-Implementierung kann dies aufgrund des erhöhten Routing-Aufwands zu einer ungünstigen Realisierung führen. Allerdings ist in vielen Fällen die

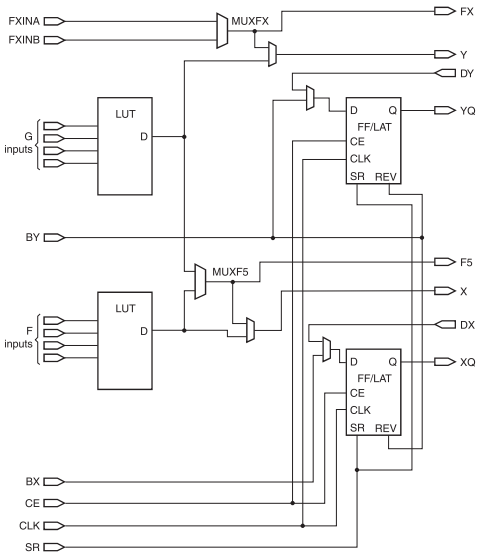


Abbildung 2.4: Look-Up Tables und Flipflops in Slices des Virtex-II FPGA [109]

Rechenlast so verteilt, dass die rechenintensiven Teile sequentiell bearbeitet werden, weshalb die Erweiterungen nicht gleichzeitig verwendet werden und die Effizienz aufgrund inaktiver Ressourcen sinkt. Dem kann durch *dynamische Rekonfiguration* entgegengewirkt werden. Im Gegensatz zur oben beschriebenen statischen Rekonfiguration, bei der das System vor dem Einsatz konfiguriert wird und die rekonfigurierbaren Strukturen im Betrieb nicht geändert werden, erlaubt die dynamische Rekonfiguration die Veränderung der gerade ungenutzten rekonfigurierbaren Strukturen. Sie kann daher verwendet werden, um nicht genutzte Erweiterungen durch aktuell (oder zukünftig) benötigte Erweiterungen zu ersetzen (Zeit-Multiplex der Hardwareressourcen). Xilinx war der erste Anbieter kommerzieller FPGAs, die dynamische Rekonfiguration unterstützten [110]. Mittlerweile wird eine vergleichbare Technologie auch von Altera (jetzt von Intel übernommen) angeboten [95].

Die folgenden Erläuterungen beziehen sich auf feingranulare Strukturen, wie sie in FPGAs zur Verfügung stehen. Ein FPGA besitzt eine konfigurierbare Logikebene, die in verschiedene Slices unterteilt ist. Diese enthalten Logikblöcke (LUTs und Flipflops), Speicher (Block RAM), oder komplexere Einheiten wie Digital Signal Processors (DSPs). Abbildung 2.4 zeigt eine Logik-Slice des Virtex-II FPGA von Xilinx [109], die zwei Look-Up Tabellen mit vier Eingängen, zwei Flipflops und Multiplexer enthält. Die 2^4 Einträge der LUTs können frei konfiguriert werden, so dass hier jede Funktion mit vier Eingängen realisiert werden kann. Zur Realisierung von Funktionen mit mehr Eingängen müssen mehrere LUTs zusammengeschaltet werden. Vier Slices werden in einem *Configurable Logic Block*

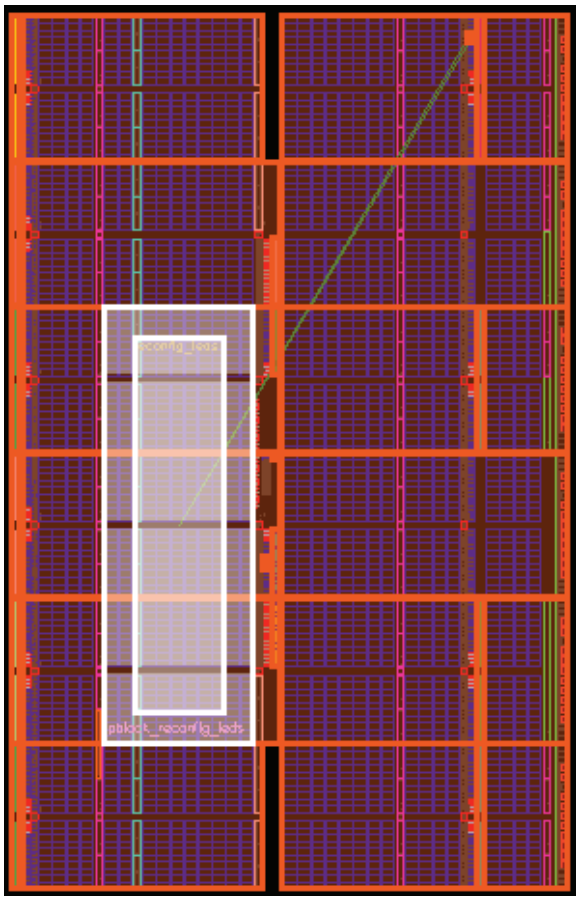


Abbildung 2.5: Floorplan eines Xilinx FPGA mit einer Region für dynamische partielle Rekonfiguration [123]

(CLB) zusammengefasst, wobei manche Slices direkt miteinander verbunden sind, um beispielsweise dedizierte Carry-Pfade zur Implementierung arithmetischer Einheiten zur Verfügung zu stellen. Programmierbare Verbindungselemente (*Connection Boxes* und *Routing Switches*) erlauben eine flexible Kombination der CLBs zu komplexen Schaltungen. Die Konfiguration der Elemente und der Verbindungen ist in einer zweiten Ebene, dem Konfigurationsspeicher, abgelegt und wird bei der statischen Konfiguration des FPGA aus einem *Bitstream* geladen, wodurch die Funktionalität des FPGA festgelegt wird. Eine Änderung des Konfigurationsspeichers führt zu einer veränderten Schaltung und damit einer geänderten Funktionalität des im FPGA abgebildeten Systems.

Die Rekonfiguration muss nicht den gesamten Konfigurationsspeicher überschreiben, sondern kann auch *partiell* erfolgen und nur einen Teil des FPGA verändern. Bei einer *dynamischen partiellen Rekonfiguration (DPRs)* muss das

Tabelle 2.1: Datendurchsatz von Schnittstellen zur Rekonfiguration von FPGAs [123]

Name	Bitbreite	Frequenz	Durchsatz
ICAP	32 bit	100 MHz	3,2 Gbit/s
SelectMAP	32 bit	100 MHz	3,2 Gbit/s
Serial Mode	1 bit	100 MHz	100 Mbit/s
JTAG	1 bit	66 MHz	66 Mbit/s

FPGA dazu nicht angehalten werden und die nicht rekonfigurierten Bereiche können weiter arbeiten. Schon während des Entwurfs des Systems müssen die dynamisch rekonfigurierbaren Bereiche (Partially Reconfigurable Region (PRR)) festgelegt werden, damit die Hardwaresynthese die Verbindungen mit dem statischen Teil der Architektur berücksichtigen kann. Dies ist exemplarisch für ein Xilinx-FPGA in Abbildung 2.5 dargestellt. Die verschiedenen Implementierungen unterscheiden sich dabei hinsichtlich der Granularität der rekonfigurierbaren Bereiche. Der Konfigurationsspeicher ist in *Frames* aufgeteilt, die die Einheiten der partiellen Rekonfiguration darstellen: die Konfigurationsdaten in einem Frame werden immer komplett überschrieben. In Virtex-II FPGAs ([110]) sind die Hardwareelemente (Logikblöcke, Block RAMs, Multiplizierer) in Spalten angeordnet. Die Frames sind 1 Bit breit und erstrecken sich über die gesamte Höhe des FPGA, betreffen also nicht nur einzelne Hardwareelemente. Daher sollten die PRRs entsprechend auch die ganze Höhe des FPGA umfassen. Diese Anordnung vereinfacht die Logiksynthese, da beispielsweise das Floor Planning auf ein eindimensionales Problem reduziert wird, kann sich aber negativ auf die Auslastung der verfügbaren Ressourcen auswirken. In späteren FPGAs wurden die Frames verkleinert (Virtex-4 [105]: 1 Bit breit, 16 Logikblöcke hoch), oder die Hardwareelemente in *Tiles* angeordnet, die jeweils Elemente eines Typs enthalten und die Einheit der Rekonfiguration sind (Virtex-5/6 [106]/[107]: Ein Tile enthält 20/40 Logikblöcke oder 8 DSPs oder 4/8 Block RAMs). Diese erhöhte Flexibilität erlaubt die Umsetzung komplexer Systeme und reduziert den Anteil nicht nutzbarer Hardwareelemente. Durch Auslesen eines Frames, Modifikation eines Teils und Zurückschreiben können mit zusätzlichem Aufwand auch Einheiten kleiner als ein Frame rekonfiguriert werden. Mit der Zync-Familie werden Systems On Chip (SoCs) angeboten, die ARM-Prozessoren mit rekonfigurierbaren Einheiten (basierend auf Virtex-7) verbinden [130].

Zur (Re-)Konfiguration des FPGA müssen die Konfigurationsdaten in den Konfigurationsspeicher geschrieben werden, wofür verschiedene Schnittstellen zur Verfügung stehen. Die Zeit für die Rekonfiguration hängt von der Datenmenge, also der Größe der PRR, sowie der Übertragungsgeschwindigkeit bzw. dem Durchsatz der verwendeten Schnittstelle ab. Der theoretisch erreichbare Durchsatz für verschiedene Schnittstellen ist in Tabelle 2.1 gegeben [123]. Die

Tabelle 2.2: Größe (Anzahl LUTs und rekonfigurierbarer Blöcke) und Zeit (ICAP Zyklen) für die Rekonfiguration verschiedener funktionaler Einheiten auf einem Virtex-5 LX 330 [79]. RPS steht für *Rekonfigurationen pro Sekunde.*

SIMD Einheit	LUTs	Blöcke	Zyklen	RPS
Clip/Max/Min	170	2	3080	32 468
Arithmetik	293	3	4620	21 645
Shift/Round	1540	12	17 920	5580

(Re-)Konfiguration eines FPGA kann über eine externe Beschaltung im seriellen Modus erfolgen. Parallele Interfaces sind z. B. JTAG oder SelectMap auf Xilinx-FPGAs [66]. Diese erlauben externen Komponenten, beispielsweise einem Hostprozessor, den Zugriff auf den Konfigurationsspeicher. Für einen internen Zugriff innerhalb des rekonfigurierbaren Systems stehen Schnittstellen, wie z. B. der Internal Configuration Access Port (ICAP) [55], zur Verfügung. Der ARM-Prozessor in einem Zync-SoC, auch *Processing System Processor (PS)* genannt, kann über den PS Processor Configuration Access Port (PCAP) eine (auch partielle) Rekonfiguration der programmierbaren Logik (PL) vornehmen. Weiterhin steht dort auch das ICAP-Modul für eine Selbstrekonfiguration der PL zur Verfügung.

Eine Möglichkeit, die Rekonfigurationszeit gering zu halten, ist die möglichst schnelle Übertragung der Bitstreams an die Schnittstelle zum Konfigurationsspeicher. Für Xilinx-FPGAs werden vom Hersteller Low-Level-Hardwaremodule (*Reconfiguration Controller*, z. B. XPS_HWICAP [120]) bereitgestellt, die für die Übertragung von Konfigurationsdaten an den ICAP zuständig sind. Diese Controller bieten allerdings geringen Durchsatz von bis zu 152,4 Mbit/s [53], wohingegen das ICAP-Modul selbst einen theoretischen Durchsatz von 3,2 Gbit/s unterstützt. Durch den Einsatz von DMA-Controllern für den Transfer der Konfigurationsdaten, wie beispielsweise auch im TUKUTURI-Prozessor, kann der laut Spezifikation maximale Durchsatz erreicht werden [79, 103]. Damit können typische Einheiten, wie in Tabelle 2.2 für ein Virtex-5 LX 330 gezeigt, zwischen 5580 und 32 470 mal pro Sekunde rekonfiguriert werden.

Ein höherer Durchsatz wurde beispielsweise in [35] durch Übertaktung des ICAP-Moduls auf bis zu 550 MHz gegenüber der spezifizierten Frequenz von 100 MHz erreicht. Die maximal erreichbare Frequenz ist aber von Fertigungstoleranzen und Routing im FPGA abhängig, was den breiten Einsatz über verschiedene Bauteile (selbst solche aus der gleichen FPGA-Familie) erschwert.

Für ein System mit rekonfigurierbaren Einheiten muss entschieden werden, wann welche Teile des Systems rekonfiguriert werden. Diese Entscheidung kann in Software oder in Hardware getroffen werden. Die Hersteller-Designprozesse erwarten das Vorhandensein eines Prozessors, der die Schnittstellen (z. B. ICAP)

verwendet [104]. Die Low-Level-Schnittstellen können aber auch mit Hardwaremodulen gekoppelt werden, die die Rekonfiguration steuern. Der TUKUTURI-Prozessor verwendet ein DMA-ICAP-Modul, um eine Selbstrekonfiguration anzustoßen [79]. Hier muss bereits zur Kompilierungszeit feststehen, welche Einheit zu welchem Zeitpunkt verwendet wird, da die Sequenz der Rekonfigurierungen im Anwendungsprogramm beschrieben wird. Andere Methoden entscheiden zur Laufzeit auf Basis von Taskgraphen, welche Einheiten rekonfiguriert werden [17]. In [73] wird die Entscheidung zur Rekonfiguration zur Laufzeit abhängig von der Häufigkeit der benutzten Einheiten bzw. der bereitgestellten Funktionalität bedarfsabhängig durchgeführt.

Bei Verwendung partieller Rekonfiguration werden verschiedene Schaltungen in den definierten PRRs abgelegt. Die Schnittstelle zwischen dem statischen und dem rekonfigurierbaren Teil der Architektur wird bei der Synthese festgelegt, so dass für alle Module, die in einer PRR verwendet werden sollen, die gleichen Verbindungen zur statisch konfigurierten Hardware bestehen. Daher muss bei der Definition der PRRs das jeweils größte dafür vorgesehene Modul berücksichtigt werden. Aufgrund der Heterogenität moderner FPGAs ist die Relokation einer Schaltung, also die Möglichkeit, eine einmal synthetisierte Schaltung in verschiedenen PRRs zu verwenden, so gut wie ausgeschlossen. Stattdessen wird die gleiche Schaltung dann mehrfach für verschiedene PRRs synthetisiert und es werden getrennte Bitstreams erzeugt. Die Altera/Intel Stratix 10 FPGA-Architektur ist in verschiedene Sektoren unterteilt, die jeweils einen eigenen Konfigurationsspeicher und einen Secure Digital Manager (SDM) besitzen, der die Rekonfiguration durchführt [43]. Da die Konfiguration der einzelnen Sektoren identisch ist, kann die Relokation von Schaltungen hier zumindest auf Ebene der Sektoren einfacher umgesetzt werden.

2.1.3 State-of-the-Art ASIP-Lösungen

Die verschiedenen Methoden und Technologien für den Entwurf und die Implementierung von ASIPs haben eine Vielzahl verschiedener Systeme hervorgebracht, von denen im Folgenden einige exemplarisch beschrieben werden.

Mit dem Tensilica Customizable Processor [13] bietet Cadence als führender Anbieter kommerzieller konfigurierbarer Prozessoren eine umfassende Entwicklungsumgebung (basierend auf der Eclipse-Plattform) zur Konfiguration und Erweiterung des so genannten Xtensa-Prozessors an. Die Konfiguration erlaubt die Auswahl aus einer Menge vorgegebener Architektureigenschaften, wie Daten- und Busbreiten, Größe/Tiefe der Registerbänke, Speicherhierarchie und Caches, usw. Optionale funktionale Einheiten, wie Dividierer, können genauso in den Prozessor integriert werden wie zusätzliche Registerbänke (z. B. für boolesche Werte). Mit der TIE-Sprache (Tensilica Instruction Extension) kann der Entwickler eigene angepasste Instruktionen oder auch spezialisierte Registerbänke

Tabelle 2.3: Vergleich von State-of-the-Art ASIP-Lösungen

Name	Jahr	Kern	Konfigurierbarkeit	Erweiterungen		Dyn. Rekonf.
Morphosys	2000 [90]	TinyRISC	nicht konfigurierbar	RPU	Array von ALUs, Multiplizierer	grob
DreAM	2000 [11]	FSM	nicht konfigurierbar	RPU	Array von Multiplizierern	grob
Xtensa LX	2002 [13]	VLIW	Pipeline, Register, optionale FUs, Speicher, Caches	RFU	TIE-Sprache ähnlich zu Verilog	keine
ADRES	2003 [63]	VLIW	nicht konfigurierbar	RFU	Array von konfigurierbaren FUs	grob
Chimaera	2004 [37]	RISC	nicht konfigurierbar	RFU	FPGA-ähnliche Struktur	fein
XiSystem	2006 [54]	VLIW	nicht konfigurierbar	RFU RPU	PiCoGA für RFUs; eFPGA für RPUs	fein
Stretch S6000	2008 [96]	VLIW	nicht konfigurierbar	RFU	Array von ALUs	grob
RMC	2010 [124]	RISC	nicht konfigurierbar	RPU	Array von CLBs	fein
i-Core	2011 [39]	RISC	nicht konfigurierbar	RFU	embedded FPGA	fein
RVPP	2013 [128]	RISC	nicht konfigurierbar	RPU	Array von Faltungskernen	grob
DIM-VEX	2018 [91]	VLIW	FUs, Issue-Slots, Speicher, Registerbänke	RPU	Array von ALUs, Multiplizierer	grob
MIPS	[112]	RISC	nicht konfigurierbar	RFU		keine

Tabelle 2.4: Vergleich von Softprozessoren

Name	Jahr	Kern	Statische Konfiguration	Dyn. Rekonfiguration
MicroBlaze	2002 [121]	RISC	Optionale FUs, Speicher, Caches	keine
PicoBlaze	2002 [16]	RISC	keine	keine
NIOS-II	2005 [44]	RISC	Caches, optionale FUs, Speicher, Pipeline, Branch-Prediction, Custom Instructions	keine
ρ-VEX	2008 [116]	VLIW	Issue-Slots, FUs, Speicher, Custom Instructions	Register, Issue-Slots, Caches
LEON4	2010 [18]	SPARC V8	Optionale FUs, Caches, Branch-Prediction	keine
VESPA	2012 [126]	MIPS + Vektor-coprozessor	Vektorlanes, Pipeline, FUs, ISA, Memory	keine
VectorBlox	2013 [87]	RISC + Vektor-coprozessor	Custom Instructions	keine
TILT	2013 [74]	VLIW	Optionale FUs, Register	keine
DFSC	2016 [102]	RISC + Dataflow	keine	Dataflow-Operationen (grob)
FPGU	2017 [47]	GPU	FUs, Caches	keine

oder externe Schnittstellen (z. B. FIFOs) beschreiben. Die Entwicklungsumgebung generiert automatisch die Kompiler-Toolchain sowie ein Simulations- und RTL-Hardwaremodell, das auf ASIC-Synthese optimiert ist.

MIPS-Prozessoren [112] bieten mit CorExtend eine Möglichkeit, den Instruktionssatz mit *User Defined Instructions (UDI)* zu erweitern. Da die Konfigurierbarkeit kein Ziel beim Entwurf der MIPS-Architektur war, sind die Möglichkeiten hier beschränkt.

Um auch nach einer Implementierung Flexibilität zu bieten, kombinieren einige Systeme einen ASIC-Prozessor mit rekonfigurierbaren Elementen. So wird bei DReAM [11] eine feste Final State Machine (FSM) mit einem Array grobgranularer RPUs gekoppelt. Morphosys [90] verbindet einen Reduced Instruction Set Computer (RISC) Prozessor mit einem grobgranularen Array von rekonfigurierbaren Zellen als RPU. Diese Zellen enthalten eine ALU, Multiplexer und Register. ADRES [63] verwendet einen festen Very Long Instruction Word (VLIW) Prozessor und ein Array mit funktionalen Einheiten und Registern. In der S6000-Prozessorfamilie von Stretch [96] wird ein vorkonfigurierter Tensilica VLIW-Prozessor mit einer grobgranular rekonfigurierbaren Struktur zur Implementierung von RFUs verbunden. Der XiRisc bzw. das XiSystem [14, 54] enthält einen XiRisc-VLIW-Prozessor, der ein PiCoGA (Pipelined Configurable Gate Array) als zusätzliche rekonfigurierbare RFU enthält. Es handelt sich dabei um ein Array feingranularer Zellen (LUTs, Register, Carry-Chain-Logik). Das XiSystem enthält außerdem noch ein verknüpftes eFPGA zur Realisierung von IO/Coprozessoren. Das Chimaera-System [37] ist eine Implementierung eines reduzierten FPGAs zur Integration auf einem Chip und wird mit einem festen RISC-Prozessor als RFU verbunden. Der RVPP (Reconfigurable Vision Pre-Processor) [128] ist ein Coprozessor, der rekonfigurierbare Hardware für Faltungsoperationen enthält, um Bildverarbeitung und Computer-Vision Anwendungen zu beschleunigen. Auch Multi-Core Systeme sind um rekonfigurierbare Strukturen erweitert worden, wie der RMC (Reconfigurable Multi-Core) [124], der feste homogene Multi-Core RISCs mit RPUs koppelt.

Im Gegensatz zum in dieser Arbeit vorgestellten TUKUTURI-Prozessor sind die hier genannten Prozessoren nicht konfigurierbar, sondern werden lediglich um anwendungsspezifische Module erweitert, die spezielle Operationen bereitstellen.

Der i-Core [39] ist ein Prozessor, der ein feingranulares eFPGA enthält, das nicht nur für Instruktionssatzerweiterungen, sondern auch für die Rekonfiguration der Mikroarchitektur zur Laufzeit verwendet werden kann. Die festen Anteile der Architektur sind als ASIC realisiert. Die Rekonfiguration erlaubt die Änderung der Pipelinestufen, der Branch-Prediction und des Caches. Der Satz von RFUs wird zur Kompilierungszeit einer Anwendung/eines Tasks festgelegt. Im Gegensatz dazu bietet das TUKUTURI-Architekturtemplate, da es vollständig in generischem VHDL beschrieben ist, deutlich mehr Parameter zur Konfiguration der Prozessorarchitektur. Im Sinne einer ressourcensparenden Implementierung

sind diese Parameter allerdings zur Laufzeit fixiert, beispielsweise um zusätzlichen Routing-Aufwand für eine dynamische Rekonfigurierbarkeit einzusparen.

Das Auto-SI System in [36] detektiert zur Laufzeit eines Programms Schleifen. Für den Schleifenkörper wird ein Datenflussgraph (DFG) erstellt und aus einer Bibliothek ein passender Beschleuniger ausgewählt und rekonfiguriert, um folgende Schleifendurchläufe zu beschleunigen. Im DIM-VEX [91] werden zur Laufzeit hot-spots detektiert und dynamisch Konfigurationen für ein grobgranulares rekonfigurierbares Array zur Beschleunigung erzeugt.

Anstatt eine ASIC-Komponente mit rekonfigurierbaren Einheiten zu koppeln, kann eine Prozessorarchitektur selbst auch in einer rekonfigurierbaren Struktur implementiert werden, z. B. auf einem FPGA. Die Rekonfigurierbarkeit des FPGA bietet ein hohes Maß an Flexibilität für eine anwendungsspezifische Ausprägung des Prozessors.

Die FPGA-Hersteller bieten für ihre FPGA-Architekturen angepasste Softprozessoren an, wie den NIOS-II [44] auf Altera/Intel-FPGAs oder den Microblaze [121] auf Xilinx-FPGAs. Dabei handelt es sich um 32-bit-RISC Prozessoren. Diese stehen in verschiedenen vordefinierten Konfigurationen zur Verfügung, mit denen zwischen hoher Performanz oder geringem Ressourcenverbrauch gewählt werden kann. Es können aber auch eigene Konfigurationen zusammengestellt werden. Der NIOS-II kann um anwendungsspezifische Instruktionen erweitert werden, der Microblaze bietet diese Möglichkeit nicht. Konfigurationsmöglichkeiten betreffen beispielsweise die Pipeline (Anzahl der Stufen) oder die Integration optionaler funktionaler Einheiten (Multiplizierer, Dividierer, ...). Für den Microblaze steht mit SecretBlaze [9] auch eine ISA-kompatible OpenSource-Implementierung zur Verfügung. Eine vereinfachte, ressourcenschonende Variante des Microblaze ist der 8-bit RISC PicoBlaze [16], der keine Konfigurationsmöglichkeiten bietet.

Der LEON4-Prozessor [18] ist ein in VHDL beschriebener SPARC V8 Prozessor, der für ASIC und FPGA synthetisiert werden kann. Der Prozessor bietet konfigurierbare Caches, Branch-Prediction und optionale funktionale Einheiten (Multiplizierer, Dividierer). Der komplexe Prozessor mit Unterstützung von Exceptions, virtuellem Speicher und Multiprozessorimplementierungen hat einen relativ hohen Flächen-/Ressourcenbedarf.

Die Konfigurierbarkeit dieser Prozessoren ist gegenüber dem TUKUTURI deutlich eingeschränkt und sie sind auch nicht für die Erweiterung mit rekonfigurierbaren Hardwareeinheiten ausgelegt.

Die VectorBlox-Architektur [87] bietet eine Vektorprozessor-Erweiterung, die in verschiedene Softprozessoren, wie NIOS-II oder Microblaze integriert werden kann. Sie ist in der Anzahl der Arithmetic Logic Units (ALUs) und der Vektorlanes konfigurierbar und um angepasste Instruktionen erweiterbar. VESPA [126] ist ein Vektorprozessor, der mit einem Softcore-MIPS gekoppelt wird. Er ist ebenfalls während des Entwurfs konfigurierbar, bietet allerdings keine Laufzeitrekonfiguration.

Der ρ-VEX [116] ist ein VLIW-Softprozessor, der u. A. in der Anzahl der Issue-Slots, den funktionalen Einheiten und der Speicherhierarchie konfigurierbar ist. Anwendungsspezifische Instruktionssatzerweiterungen können zur Entwurfszeit definiert werden. In darauf folgenden Arbeiten sind Methoden zur dynamischen Rekonfiguration des Prozessors integriert worden: So ist die Rekonfiguration der Registerbänke ([115]), der Anzahl der Issue-Slots ([4]) oder auch der Caches ([3]) möglich.

Im DFSC (Data-Flow Soft-Core) [102] gibt es neben dem Softprozessor noch extra ausgewiesene Bereiche für eine dynamische Rekonfiguration eines Beschleunigers. Die Datenflussoperationen, die auf dem Beschleuniger ausgeführt werden, enthalten Konfigurationsinformationen, um den Beschleuniger für die Ausführung anzupassen.

Mit TILT wurde in [74] ein Multi-Thread VLIW-Softprozessor vorgestellt. Die Instruktionen in einem langen Instruktionswort können aus unterschiedlichen Threads stammen. Die Architektur besitzt eine tiefe Pipeline. Das Thread-Scheduling erfolgt statisch zur Kompilierungszeit. Die Architektur bietet eine Konfiguration, jedoch keine dynamische Rekonfiguration.

In [47] wird mit FGPU eine 32-bit rekonfigurierbare GPU als Softprozessor auf einem FPGA beschrieben, die als Overlay-Architektur für Allzweck-Berechnungen (general purpose GPU, GPGPU) konzipiert ist. Sie bietet einen konfigurierbaren Instruktionssatz (Auswahl bzw. Einschränkung vorhandener Operationen) und Caches, sowie eine konfigurierbare Anzahl an IO-Schnittstellen.

2.2 Kompilertechnologie

2.2.1 Grundlagen

Die Programmierung von Prozessoren erfolgt in Programmiersprachen, die von sehr hardwarenahem Assemblercode bis zu abstrakten Hochsprachen reichen. Für die Ausführung auf einer Prozessorarchitektur müssen Programme von einem Übersetzer (Kompiler) in Binärcode übersetzt werden. Dieser Übersetzungsvorgang geschieht in mehreren Schritten, die typischerweise in die Phasen *Analyse* (auch *Front-End*) und *Synthese* (auch *Back-End*) gruppiert werden (vgl. [1, 114]).

Während der *Analyse* wird das Programm in der Programmiersprache eingelesen, die syntaktische Struktur analysiert (Lexing/Parsing) und das Programm in eine interne Darstellung (*Intermediate Representation (IR)*) konvertiert. Diese interne Repräsentation des Programms erlaubt weitere Analysen und Transformationen, zum Beispiel von der Zielplattform unabhängige Optimierungen, die die Programmstruktur (Schleifen, Sprünge, Funktionsaufrufe) betreffen. Das Ergebnis der Analysephase, also das geprüfte und optimierte Programm, wird dann an die Synthese weitergegeben.

Die *Synthese* wird gelegentlich in weitere Untereinheiten aufgeteilt. Im sogenannten *Middle-End* wird das Programm für die Zielplattform optimiert. Hierbei werden konkrete Eigenschaften und Fähigkeiten der Zielarchitektur, wie spezielle Operationen, ausgenutzt. Der letzte Abschnitt, das *Back-End*, übersetzt das Programm aus der internen Darstellung in lauffähigen Maschinencode. Diese auch als Codeerzeugung bekannte Phase kann in weitere Aufgaben aufgeteilt werden:

- Die *Code Selection* wählt die konkreten Maschinenbefehle für die Übersetzung der abstrakten Befehle aus der internen Darstellung des Programms aus. Hier entsteht in vielen Kompilern Assemblercode, der von den nachfolgenden Phasen optimiert und in Binärcode übersetzt wird.

- Das *Instruction Scheduling* legt die Reihenfolge fest, in der Befehle ausgeführt werden sollen bzw. in der sie im Binärcode erscheinen.

- Die *Registerallokation* wählt die Variablen im Programm aus, die in Registern gehalten werden sollen und ordnet ihnen Hardwareregister zu.

Abschließend erfolgt eine Binärkodierung des kompilierten Programms, um den ausführbaren Code zu erzeugen.

Die lexikalische und syntaktische Analyse von Programmen im Front-End eines Kompilers stellt im Allgemeinen keine besondere Herausforderung dar, da diese Phasen oftmals automatisiert aus einer einfachen Beschreibung der Programmiersprache (z. B. mittels einer Grammatik) generiert werden können. In dieser Arbeit liegt der Fokus auf der Synthesephase, nämlich der Codeoptimierung, dem Instruction Scheduling und der Registerallokation für die VLIW-SIMD-Softprozessorarchitektur namens TUKUTURI, die in Kapitel 3 vorgestellt wird.

2.2.2 Codegenerierung für VLIW-Architekturen

Ein Very Long Instruction Word Prozessor stellt Parallelität auf Instruktionsebene (*Instruction-Level Parallelism (ILP)*) bereit, da er mehrere Operationen parallel in verschiedenen *Issue-Slots* ausführen kann. Diese sind dazu in langen Instruktionswörtern angeordnet. Die Zusammenstellung, also die Festlegung der Reihenfolge und Parallelität der Operationen aus einem sequenziellen Eingabeprogramm, erfolgt statisch durch den Kompiler, im Gegensatz zu einem dynamischen Scheduling durch die Hardware zur Laufzeit des Programms (z. B. in superskalaren Prozessoren). Die Komplexität des Instruction Scheduling wird damit in VLIW-Architekturen von der Hardware in den Kompiler verschoben.

Die nachfolgenden Definitionen erläutern Begriffe, die in den darauf folgenden Beschreibungen der einzelnen oben erwähnten Teilaufgaben der Codegenerierung verwendet werden.

Definition 2.1. Eine *Mikrooperation (MO)* ist eine Operation, die in einem Issue-Slot eines VLIW-Prozessors ausgeführt wird. Der Satz von Operationen, die von der Architektur ausgeführt werden können, ist der *Befehlssatz* des Prozessors.

Definition 2.2. Ein *Straight-Line Microcode (SLM)* ist eine geordnete Sequenz von MOs mit dem einzigen Einsprungpunkt am Anfang und keinen Verzweigungen oder Sprüngen, außer möglicherweise an ihrem Ende.

$$\mathrm{SLM} = (\mathrm{MO}_1, \mathrm{MO}_2, \ldots, \mathrm{MO}_N).$$

Damit werden, sofern der Kontrollfluss den Einsprungpunkt der SLM erreicht, immer alle Operationen in der SLM ausgeführt, da es keine Möglichkeit gibt, die SLM vorzeitig zu verlassen. Die *Größe einer SLM* ist die Anzahl der enthaltenen MOs: $|\mathrm{SLM}| = N$. Beim Lesen des Eingabeprogramms wird die Folge der Operationen durch dem Kompiler in eine Folge von SLMs zerlegt.

Definition 2.3. Eine *Mikroinstruktion (MI)* ist eine Zusammenfassung von MOs die parallel ausgeführt werden und in einem Instruktionswort kodiert werden: $\mathrm{MI} = (\mathrm{MO}_1, \mathrm{MO}_2, \ldots, \mathrm{MO}_K)$. Die maximale Anzahl der MOs in einer MI ist durch die Anzahl K der Issue-Slots des Prozessors begrenzt. Benötigt die Kodierung einer MO zusätzliche Bits, z. B. zur Kodierung von langen Immediate-Werten, kann dies die Kodierung einer weiteren Operation in einem parallelen Issue-Slot verhindern. Beim Instruction Scheduling können unter Umständen aufgrund von Datenabhängigkeiten oder Konflikten nicht alle Issue-Slots aufgefüllt werden. Diese Slots werden in der Mikroinstruktion mit *no operation* (`NOP`) belegt.

Definition 2.4. Ein *Basic Block (BB)* ist eine Folge von MIs, die durch das Scheduling der MOs einer SLM entstanden ist:

$$\mathrm{BB} = (\mathrm{MI}_1, \mathrm{MI}_2, \ldots, \mathrm{MI}_P).$$

Die *Größe eines Basic Blocks* ist die Anzahl der enthaltenen Mikroinstruktionen $|\mathrm{BB}| = P$.

Definition 2.5. Ein *Mikroprogramm (MP)* ist eine Folge von basic blocks: $\mathrm{MP} = (\mathrm{BB}_1, \mathrm{BB}_2, \ldots, \mathrm{BB}_L)$. Als *Größe des Programms* wird die Gesamtzahl der Mikroinstruktionen, also die Summe der Größen der Basic Blocks, verstanden:

$$|\mathrm{MP}| = \sum_{i=1}^{L} |\mathrm{BB}_i|.$$

Die Codegenerierung für VLIW-Prozessoren ist der Prozess, der ein Programm in Form einer Folge von SLMs in ein funktional äquivalentes Mikroprogramm transformiert, das nach einer Binärkodierung auf dem Zielprozessor ausgeführt

werden kann. Dabei sind Eingabeprogramm und generiertes Mikroprogramm funktional äquivalent, wenn sie für gleiche Eingaben gleiche Ergebnisse produzieren.

Die wichtigste Aufgabe des Kompilers ist das Instruction Scheduling. Durch Festlegung einer geeigneten Reihenfolge und Parallelität für die Operationen bzw. Instruktionen kann großer Einfluss auf das Laufzeitverhalten des Programms genommen werden. Da das erzeugte Programm funktional äquivalent zum Eingabeprogramm sein muss, ist der Kompiler dabei verschiedenen Beschränkungen unterworfen. In Architekturen mit Pipeline treten z. B. folgende *Konflikte* auf [40]:

- *Strukturkonflikte* bezeichnen Konflikte, bei denen die Hardware nicht alle Zugriffe der überlappenden Operationen gleichzeitig erfüllen kann, wenn beispielsweise eine Ressource nicht ausreichend oft vorhanden ist.

- *Datenkonflikte* treten auf, wenn Abhängigkeiten zwischen Operationen bestehen und eine Operation auf Ergebnisse einer anderen Operation warten muss. Durch die Pipeline werden bestimmte Wartezeiten (*Latenzen*) eingeführt, die angeben, wann die Ergebnisse einer Operation für nachfolgende Operationen bereitstehen.

- *Steuerkonflikte* entstehen durch die Ausführung von Verzweigungen oder Sprungbefehlen in der Pipeline, da diese durch Änderung des *Program Counters* die Sequenz der Operationen ändern. Wenn der Sprungbefehl ausgeführt wird, sind nachfolgende Operationen bereits in die Pipeline geladen worden und müssen dann abgebrochen werden.

Konflikte können behoben werden, indem Operationen durch Anhalten der Pipeline (*stalls*) oder Einfügen von `NOP` (*bubbles*) verzögert werden. Eine weitere Möglichkeit bietet das *Forwarding*, mit dem Ergebnisse einer Operation nachfolgenden Operationen direkt zugeführt werden, ohne die restlichen Pipelinestufen abwarten zu müssen. Durch geeignetes Scheduling der Instruktionen können stalls und bubbles möglicherweise verhindert bzw. durch andere Operationen aufgefüllt werden. Dazu muss der Kompiler zunächst eine Datenabhängigkeitsanalyse durchführen, mit denen Beschränkungen beim Scheduling berücksichtigt werden können.

Eine weitere wichtige Aufgabe ist die Registerallokation (RA), die den im Programm verwendeten Variablen Hardwareregister zuordnet. Dabei muss die Lebensdauer der Variablen berücksichtigt werden, denn für jede noch lebende, d. h. von nachfolgenden Operationen als Operand verwendete Variable, muss ein Register bereitgehalten werden. Überlappen sich die jeweiligen Lebensdauern zu vieler Variablen, stehen möglicherweise nicht genügend Hardwareregister zur Verfügung. In einem solchen Fall kann der Kompiler *spill code* erzeugen, der

Werte aus Registern temporär in den Speicher auslagert und sie später wieder lädt. Dieser zusätzliche Aufwand kann durch geeignetes Instruction Scheduling verringert oder sogar verhindert werden, wenn das Überlappen der Lebensdauern verschiedener Register verringert werden kann.

Die Codegenerierung mit ihren Teilaufgaben kann als Optimierungsproblem aufgefasst werden. Neben einer Optimierung für minimale Programmlaufzeit (durch hohe Kompaktierung des Codes) können weitere Optimierungsziele, wie etwa Leistungsaufnahme oder auch gleichmäßige Alterung der Hardware, berücksichtigt werden. Im Allgemeinen sind diese Optimierungsprobleme, die ein Kompiler zu lösen hat, NP-schwer [41, 50, 57] und eine exakte Lösung ist nur für kleine Probleminstanzen möglich. Kompiler verwenden heuristische Funktionen, um den Lösungsraum einzuschränken, wodurch schneller Lösungen gefunden werden, die aber möglicherweise nicht mehr optimal sind. Wie nahe die gefundenen Lösungen der optimalen Lösung kommen, hängt von der Güte der Heuristik ab. Das Finden guter Heuristiken ist keine leichte Aufgabe, da sie von Eigenschaften der Zielarchitektur abhängt, und wird häufig manuell durchgeführt.

2.2.3 Stand der Technik

Die *Code Selection* wird häufig als Graphüberdeckungsproblem implementiert, bei denen die Operationen der Zielarchitektur in Form kleiner Datenflussgraphen beschrieben werden. Aus einer Überdeckung des Datenflussgraphen der Anwendung mit diesen kleineren Graphen wird dann die Sequenz der Operationen abgeleitet [51]. Eine komplexere Code Selection, die mehrere Operationen umfasste, wurde mit genetischen Algorithmen implementiert [60].

Für das *Instruction Scheduling* wird eine große Vielzahl verschiedener Techniken eingesetzt. Diese können eingeteilt werden in *lokales Scheduling*, bei dem das Programm in SLMs zerlegt wird und diese unabhängig voneinander verarbeitet werden, und *globales Scheduling*, das es dem Kompiler zusätzlich erlaubt, einzelne Operationen zwischen den SLMs zu verschieben.

Versuche eines optimalen Schedulings beinhalten beispielsweise diskrete Optimierung [118, 119], ganzzahlige lineare Optimierung [2, 24] und Constraintprogrammierung [58, 59]. Diese verwenden mathematische Modelle des Problems und Lösungsalgorithmen wie branch-and-bound oder Constraint-Löser, können aber aufgrund der Problemkomplexität nicht für alle Eingabeprogramme Lösungen finden. Der branch-and-bound-Algorithmus kann auch direkt zum Instruction Scheduling eingesetzt werden. Die MIs werden aus verfügbaren (d. h., alle Datenabhängigkeiten sind erfüllt) MOs zusammengesetzt. Jede MI wird zunächst vollständig mit MOs oder `NOP` gefüllt, bevor die nächste begonnen wird. Wenn dem Algorithmus mehrere MOs zur Auswahl stehen, verzweigt der Algorithmus und bildet so einen Suchbaum. Jeder Pfad von der Wurzel zu einem Blatt stellt ein vollständig scheduliertes Programm dar und der kürzeste dieser Wege liefert

das kompakteste Programm. Damit wird das Optimum zumindest theoretisch gefunden, in der Praxis ist der Aufwand zur Berechnung des kompletten Suchbaums aber zu groß, auch wenn der Algorithmus einzelne Teilbäume beschneidet, die als nicht optimal identifiziert werden können.

Approximative Techniken verwenden Heuristiken, um den Algorithmus zu steuern und das Scheduling zu beschleunigen. Das List Scheduling ist eine Variante des branch-and-bound-Verfahrens, bei der Zweige im Suchbaum aggressiv beschnitten werden, bis nur noch ein Pfad übrig ist. Das Verfahren wird in verschiedenen Arbeiten verwendet [50, 56, 76, 81, 89, 111]. Um zu entscheiden, welche Zweige im Suchbaum gekappt werden, ordnet die Heuristik den verschiedenen MOs Prioritäten zu. Ähnliche Methoden wie Modulo-Scheduling [7, 127] oder Rotation-Scheduling [111] sind speziell für die Optimierung von Schleifen entwickelt worden, da Optimierungen dort aufgrund der Wiederholungen lohnender sind. Heuristische Funktionen spielen eine große Rolle beim Entwurf eines Kompilers. Um tiefe Einsicht in das Scheduling-Problem zu liefern, muss die Funktion Eigenschaften sowohl der Zielarchitektur als auch des Programms berücksichtigen. Der Entwurf einer guten Heuristik ist eine komplexe Aufgabe, die häufig manuell durchgeführt wird.

Heuristiken werden auch in Standardkompilern wie dem GCC und LLVM eingesetzt. Beide Kompiler nutzen List Scheduling mit verschiedenen Heuristiken. Die Standardheuristik im LLVM-Kompiler berücksichtigt beispielsweise Registerdruck, Clustering und kritische Ressourcen [25]. GCC und LLVM stellen flexible Kompiler-Infrastrukturen bereit, mit denen Kompiler-Back-Ends für verschiedene Zielarchitekturen implementiert werden können. Dabei können bereitgestellte Scheduling-Verfahren eingesetzt oder eigene implementiert werden.

Genetische Algorithmen, eine Unterklasse von Evolutionären Algorithmen (EAs), sind ebenfalls für das Instruction Scheduling eingesetzt worden, entweder, um die Reihenfolge der Instruktionen direkt festzulegen [56, 57, 129], oder als eine dynamische Heuristik für andere Scheduling-Algorithmen (wie List Scheduling) in [24] und [28]. Die Autoren in [24] verglichen auf Genetischen Algorithmen basierende Heuristiken mit optimalem Scheduling mittels ganzzahliger Optimierung. Die Schlussfolgerung war, dass Genetische Algorithmen (GA) Lösungen nahe des Optimums (teilweise mit einem Längenunterschied von nur einer Instruktion) lieferten und auch dann mit vorhersagbarer Laufzeit erfolgreich endeten, wenn die Optimierung keine Lösung finden konnte. In [117] wurde ein GA zur Lösung eines chance-constrained rough-program eingesetzt und [129] kombiniert einen GA mit einer Tabu-Suche, um die Konvergenz gegen lokale Minima zu verhindern. Andere Methoden verwenden Partikelschwarmoptimierung [67] oder Reinforcement Learning [62].

Registerallokation ist eine weitere wichtige Aufgabe der Codegenerierung, für die verschiedene Algorithmen verwendet werden. Die Zuordnung von Variablen zu Hardwareregistern kann durch eine Graphfärbung durchgeführt werden [12].

In VLIW-Architekturen wird das Scheduling häufig durch Beschränkungen in den Registerbänken erschwert. Beispielsweise sind Registerbänke häufig geclustert und die Zahl der Lese-/Schreibports ist beschränkt. Die Registerallokation muss so durchgeführt werden, dass parallele Operationen auf alle benötigten Register gleichzeitig zugreifen können. Die Arbeit in [52] verwendet eine Heuristik zur Verteilung der Variablen auf Registerbänke und ordnet sie anschließend Registern über eine Graphfärbung zu. Registerallokation und Instruction Scheduling beeinflussen sich gegenseitig, wie in [68] beschrieben, wo eine Registerumbenennung benutzt wurde, um die ILP auf Kosten zusätzlicher Register zu erhöhen. Die Registerallokation wird zuerst ausgeführt und das Instruction Scheduling kann einzelne Variablen neu zuordnen. Andere Arbeiten kombinieren Registerallokation und Instruction Scheduling [97]. In [20] wird RA in Modulo-Scheduling integriert; [58, 59] verwenden Constraintprogrammierung mit kombinierter RA.

Die hier beschriebene Implementierung der Codegenerierung, die auf [28] basiert, verwendet das heuristische List Scheduling. Anstatt allerdings eine statische Heuristik zu verwenden, wird ein Evolutionärer Algorithmus (EA) verwendet, um die Prioritäten der Mikrooperationen (MOs) während des Scheduling festzulegen und zu evolvieren. Zusätzlich werden zwei weitere EAs verwendet, um die Registerallokation und eine Codeoptimierung (Operation Merging) durchzuführen. Die drei Verfahren sind miteinander verschachtelt und führen so eine kombinierte Optimierung aller drei Teilprobleme durch.

2.3 Evolutionäre Algorithmen

Optimierungsaufgaben werden in der Mathematik häufig als Zielfunktion (objective function) von Design-/Entscheidungsvariablen beschrieben. Das Ziel besteht darin, Werte für die Variablen zu finden, so dass die Zielfunktion möglichst kleine oder große Werte annimmt. Im Folgenden werden zur Vereinfachung der Erklärungen nur Maximierungsprobleme betrachtet, die gemachten Aussagen lassen sich aber leicht auf Minimierungsprobleme übertragen.

Evolutionäre Algorithmen sind Strategien zur Behandlung von Optimierungsproblemen, die an die natürliche Evolution angelehnt sind [38, 64, 125]. Erste Entwicklungen dieser Ideen stammen bereits aus den 1950er und 1960er Jahren, beispielsweise mit der Evolutionsstrategie von Rechenberg [85], und sind später von Holland [42] um zusätzliche Operationen erweitert worden. Heutzutage gibt es eine Vielzahl von Varianten der Methoden und die Grenzen sind nicht scharf definiert, weshalb in den folgenden Erläuterungen die Begriffe Evolutionäre Algorithmen (EA) und Genetische Algorithmen (GA) weitestgehend synonym verwendet werden.

2.3.1 Grundlagen Evolutionärer Algorithmen

Im Gegensatz zu vielen anderen Optimierungsmethoden können EA mit einer Vielzahl verschiedener Problemtypen umgehen, da sie wenig Anforderungen an die Zielfunktion stellen. Sie können unabhängig davon eingesetzt werden, ob die Zielfunktion statisch oder veränderlich, linear oder nichtlinear, stetig oder nicht stetig ist und benötigen auch keine Gradienten der Zielfunktion [125].

In Anlehnung an die natürliche Evolution verwendet ein EA eine Population von Individuen, die mögliche Lösungen des Problems darstellen und die Anhand einer Fitnessfunktion bewertet werden. Dann werden Individuen zur Rekombination ausgewählt und mittels der genetischen Operatoren *Crossover* und *Mutation* neue Individuen abgeleitet. Ein allgemeines Schema eines EA kann nach [38] wie folgt beschrieben werden (vereinfacht):

1. Vorgabe einer problemangepassten Zielfunktion (Fitnessfunktion).
2. Kodierung der Problemparameter in Genen der Individuen.
3. Erzeugen einer Initialpopulation.
4. Rekombination: Erzeugen neuer Individuen durch Anwendung der genetischen Operatoren.
5. Bewertung der neuen Individuen mit der Fitnessfunktion.
6. Erzeugen der neuen Generation durch Selektion der Individuen in Abhängigkeit ihrer Fitness.
7. Auswertung eines Abbruchkriteriums. Falls nicht erfüllt, Wiederholung ab Schritt 4.

Kodierung der Individuen

Ein Individuum wird durch eine Sequenz von Genen repräsentiert, die jeweils einen Parameter des Systems kodieren. Dabei kann es sich direkt um die Parameter der Zielfunktion handeln, es können aber auch zusätzliche Parameter kodiert werden, die beispielsweise Zusammenhänge zwischen einzelnen Genen kodieren oder die Einfluss auf die genetischen Operationen nehmen (Metaparameter, siehe z. B. [38]). In den ursprünglichen Beschreibungen genetischer Algorithmen wurden Gene ausschließlich binär codiert, später wurden auch größere Einheiten, wie ganze Zahlen (integer) oder reelle Zahlen (float) eingeführt, um den Einfluss der genetischen Operatoren, insbesondere der Mutation, besser steuern zu können (siehe dazu die Erläuterungen zur Mutation weiter unten). Die Sequenz der Gene wird auch als Chromosom bezeichnet. Typischerweise haben alle Chromosomen die gleiche Länge, und diese bleibt über die Laufzeit des Algorithmus konstant.

Fitnessbewertung

Die Fitnessfunktion eines EA dient der Bewertung der Lösungen, die durch die Individuen repräsentiert sind. Dies kann direkt die Zielfunktion sein, die Fitnessfunktion kann aber auch andere Aspekte des Problems und zusätzliche Beschränkungen an die Lösungen enthalten. Eine Fitnessfunktion kann absolute Fitnesswerte für die Individuen angeben, so dass die Differenz zwischen der Fitness zweier Individuen aussagekräftig ist, oder die Fitnessfunktion liefert nur relative Ergebnisse, die eine Rangfolge (ranking) der Individuen angeben. Dies hat insbesondere Auswirkungen auf die Selektionsoperationen.

Selektion und Crossover

Dieser Prozess simuliert die Fortpflanzung von Individuen, indem aus der Population zunächst Eltern ausgewählt werden und aus diesen dann durch Austausch des Genmaterials Kind-Individuen abgeleitet werden. Die Wahrscheinlichkeit für ein Individuum für die Reproduktion ausgewählt zu werden ist häufig mit der Fitness korreliert, wofür unterschiedliche Methoden verwendet werden können [84]:

- **Rouletteselektion:** Jedem Individuum wird ein Abschnitt einer Linie zugeordnet dessen Länge der (absoluten) Fitness des Individuums entspricht. Zur Selektion wird dann eine Zufallszahl gleichverteilt über die Gesamtlänge der Linie ermittelt. Die Selektion erfolgt damit fitnessproportional. Es werden so viele Zufallszahlen generiert, bis alle benötigten Individuen ausgewählt worden sind [8].
- **Stochastic universal sampling:** Wie bei der Rouletteselektion werden die Individuen gemäß ihrer Fitness auf einer Linie angeordnet. Für eine Anzahl N zu selektierender Individuen wird dann eine Menge von N Punkten gleichmäßig im Abstand $1/N$ auf der Linie verteilt, wobei der erste Punkt zufällig im Intervall $[0, 1/N]$ gewählt wird. Durch dieses Verfahren wird, im Gegensatz zur Rouletteselektion, eine größere Verteilung (*spread*) der selektierten Individuen garantiert. Die Selektion ist ebenfalls fitnessproportional [8].
- **Turnierselektion:** Die Turnierselektion [32] mit Turniergröße N wählt aus der Population gleichverteilt (also unabhängig von der Fitness) N Individuen aus. Der Gewinner eines Turniers ist das Individuum mit der größten Fitness unter diesen N Individuen. Der Selektionsdruck wird über die Größe der Turniere gesteuert. Diese Selektion verwendet nur die relative Fitness der Individuen und ignoriert den absoluten Betrag, ist also nicht fitnessproportional. Daher kann jede Fitnessfunktion verwendet werden, solange sie Individuen in die gleiche Reihenfolge wie die Zielfunktion sortiert.

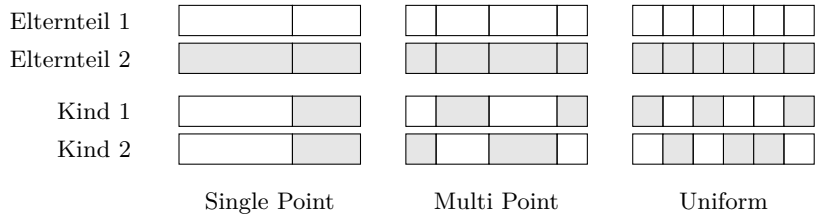


Abbildung 2.6: Verschiedene Varianten des Crossover zur Rekombination von Genmaterial. Für uniform crossover ist ein Beispiel für eine zufällige Verteilung der Gene dargestellt.

Zur Ableitung der Kindgenome aus zwei Elterngenomen, die sogenannte Rekombination, gibt es ebenfalls verschiedene Methoden, siehe [38, 84]. Das Crossover wird am häufigsten verwendet. Dabei werden die Chromosomen der Eltern an einer zufällig gewählten Stelle getrennt und die entstehenden Teile abwechselnd auf die Kinder verteilt. Wird mehr als eine Trennstelle verwendet, spricht man vom *multi-point crossover* im Gegensatz zum *single-point crossover* bei nur einer Trennstelle. Als Weiterführung kann die Auswahl des Elternteils auch für jedes Gen unabhängig getroffen werden (*uniform crossover*). Die verschiedenen Methoden sind schematisch in Abbildung 2.6 dargestellt.

Durch die Rekombination der elterlichen Gene liegen die beim Crossover entstehenden Individuen innerhalb eines Hyperwürfels, der durch die elterlichen Parameterwerte aufgespannt wird. Daher ist die Rolle der Rekombination eher die Exploitation der näheren Umgebung im Parameterraum und soll bestehende Lösungen verbessern.

Mutation

Die Mutation erzeugt zufällige Veränderungen an den Individuen. Typischerweise ist die Wahrscheinlichkeit für eine Mutation gering und die Veränderungen durch einen Mutationsoperator eher gering. Die Mutation kann auf die Individuen der aktuellen Population angewendet werden, kann aber beispielsweise auch nur die durch Rekombination gewonnenen Individuen verändern. Für binär kodierte Individuen bietet sich ein Operator an, der zufällig einzelne Bits kippt. Je nach gewählter Bitposition und der entsprechenden Wertigkeit des Bits kann dadurch eine große Veränderung in einem Parameterwert erzeugt werden. Bei ganzzahliger oder reelwertiger Kodierung wird häufig der Zahlenwert verändert indem eine normalverteilte Zufallszahl mit Mittelwert 0 addiert wird, wodurch größere Änderungen mit geringerer Wahrscheinlichkeit auftreten als kleinere Veränderungen [84].

Durch die zufälligen Veränderungen können die Individuen neue Gebiete des Parameterraums erschließen, weshalb die Rolle der Mutation die Exploration des

Parameterraums und das Entkommen aus lokalen Optima ist.

Die Mutationswahrscheinlichkeit kann über die Laufzeit des Algorithmus konstant bleiben, es bietet sich allerdings auch die Möglichkeit einer adaptiven Veränderung. Zum Beispiel kann die Mutationsrate erhöht werden, wenn die Fitness vieler Individuen sehr ähnlich ist, oder der Algorithmus für mehrere Generationen nur geringe Verbesserung erreichen konnte.

Selektion für die nächste Generation

Typischerweise wird die Populationsgröße in einem EA konstant gehalten. Um nach Anwendung der genetischen Operatoren auf die Individuen die nächste Generation zu erzeugen, sind wiederum mehrere Varianten bekannt. So können die Individuen der vorherigen Generation komplett verworfen werden und die nächste Generation nur Individuen enthalten, die durch die Operatoren erzeugt wurden. Eine andere Möglichkeit ist, die Individuen der vorherigen Generation mit den daraus abgeleiteten Individuen gemeinsam nach ihrer Fitness zu sortieren und von diesen die benötigte Anzahl der besten Individuen für die nächste Generation auszuwählen. Eine als Elitismus bekannte Form kopiert die besten k Individuen der vorherigen Generation in die neue Generation und füllt diese dann mit abgeleiteten Individuen auf. Da in diesen Fällen Individuen über mehr als eine Generation existieren können, kann für die Individuen auch eine Lebensdauer eingeführt werden, mit der die maximale Anzahl an Generationen bezeichnet wird, die ein Individuum mitgeführt wird.

Die Selektionen für die Rekombination und für die nächste Generation sind die fitnessabhängigen Schritte eines EA und bilden damit die Triebfedern, die eine Verbesserung der Fitness in den Individuen vorantreiben, indem sie eine Verbreitung guten Genmaterials fördern.

2.3.2 Evolutionäre Algorithmen mit mehreren Zielfunktionen

Die bisher beschriebenen EA hatten eine einzige Zielfunktion die es zu optimieren galt. Wenn mehrere Zielfunktionen optimiert werden sollen (*multi-objective evolutionary algorithm*), ist das Ziel der Optimierung das Auffinden Pareto-optimaler Lösungen, da die Zielfunktionen häufig gegenläufig sind und die Verbesserung einer Zielfunktion auf Kosten der anderen Zielfunktionen erfolgt. Ein weit verbreiteter Algorithmus ist der NSGA-II [23], der eine Weiterentwicklung verschiedener früherer Algorithmen mit mehreren Zielfunktionen ist. NSGA steht für *Non-dominated Sorting Genetic Algorithm* und verwendet das Konzept der Dominanz: Ein Individuum dominiert ein anderes Individuum, wenn seine Bewertungen nach den Bewertungsfunktionen mindestens genauso gut sind (also größer oder gleich), wie für das andere Individuum, und mindestens eine Bewertung tatsächlich besser (also strikt größer) ist. Dann kann eine Pareto-Front aus den nicht-dominierten

Individuen der aktuellen Generation gebildet werden. Werden diese Individuen aus der Population entfernt, kann die nächste Pareto-Front aus den nun nicht-domnierten Individuen gebildet werden. So wird jedem Individuum ein Level zugeordnet, das beispielsweise bei der Selektion verwendet wird: Individuen auf niedrigerem Level werden den Individuen auf einem höheren Level vorgezogen. Zur Differenzierung zwischen Individuen des gleichen Levels (also der gleichen Pareto-Front) wird die sogenannte *crowding distance* herangezogen, mit der die Dichte der Individuen im Parameterraum bezüglich der Bewertungsfunktionen beschrieben wird. Individuen in nicht dicht besiedelten Bereichen des Parameterraums werden bevorzugt, da hier die Wahrscheinlichkeit für eine zielführende Verbesserung der Individuen höher ist als in dichter besiedelten Bereichen des Parameterraums. Die sich daraus ergebene Rangfolge der Individuen wird in einer Turnierselektion verwendet um Elternindividuen auszuwählen. Dann werden mit den bereits beschriebenen Operatoren für Rekombination und Mutation neue Individuen abgeleitet. Der NSGA-II generiert die nächste Generation von Lösungen, indem die Individuen der vorherigen Generation und die daraus abgeleiteten Individuen gemeinsam nach der oben beschriebenen nicht-dominierten Sortierung angeordnet werden und die besten Individuen in die nächste Generation übernommen werden.

3 TUKUTURI: Ein Applikationsspezifischer VLIW-SIMD Softprozessor

Dieses Kapitel beschreibt eine flexible Softprozessorarchitektur namens TUKUTURI. Diese basiert auf einem generischen Architekturtemplate, das im Rahmen des RAPANUI-Projektes [77] entstanden ist. Im Unterschied zum RAPANUI-Projekt, in dem die Architektur für eine ASIC-Implementierung optimiert wurde, ist die TUKUTURI-Prozessorarchitektur als Softprozessor für Xilinx Virtex 6 FPGAs optimiert. Die Prozessorarchitektur wird in Abschnitt 3.1 kurz erläutert. Die flexible Parametrisierung der Prozessorpipeline zur Optimierung der Architektur auf FPGAs wird in Abschnitt 3.2 vorgestellt.

Zur Umsetzung des ASIP-Konzeptes bietet der TUKUTURI neben der Parametrisierung der Pipeline außerdem Schnittstellen zur Erweiterung des Befehlssatzes. Zum einen können funktionale Einheiten direkt in die Pipeline integriert werden, um spezialisierte Operationen zur Verfügung zu stellen, zum anderen können komplexere Operationen in Coprozessoren implementiert werden, die über den Datenbus angebunden werden.

Die Nutzung rekonfigurierbarer Hardware erlaubt neben der statischen Konfiguration des Softprozessors auch die dynamische Rekonfiguration zur Laufzeit, um den Prozessor an zeitlich veränderliche Anforderungen der Anwendung anpassen zu können, wie in Abschnitt 3.4 beschrieben.

3.1 Eine generische VLIW-SIMD Prozessorarchitektur

Der TUKUTURI-Prozessor basiert auf einem flexiblen Architekturtemplate [77] wie in Abbildung 3.1 gezeigt, das als Basis eines konfigurationsbasierten ASIP-Entwurfs verwendet werden kann. Die nachfolgenden Abschnitte erläutern verschiedene Aspekte dieser Architektur.

3.1.1 Der VLIW-SIMD Datenpfad

Die Architektur bietet einen flexiblen Datenpfad, der durch ein 2-Issue-Slot VLIW gesteuert wird. Um den Parallelisierungsgrad auf Ebene der Instruktionen (Instruction-Level Parallelism (ILP)) zu erhöhen, können funktionale Einheiten

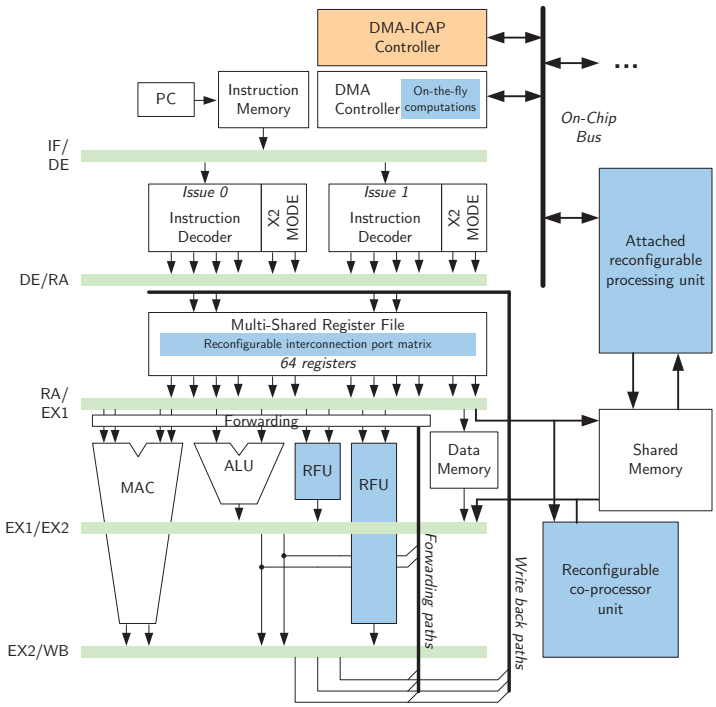


Abbildung 3.1: TUKUTURI: Eine generische VLIW-SIMD Softprozessorarchitektur [77]. Dynamisch rekonfigurierbare Anteile sind blau hinterlegt. Die Verbindung zum restlichen System (z. B. externer Speicher) ist durch die drei Punkte dargestellt.

dupliziert werden, so dass die bereitgestellten Operationen parallel verwendet werden können. Der 64-bit-Datenpfad kann in kleinere Subworte (8×8 bit, 4×16 bit, 2×32 bit) aufgeteilt werden, die von den funktionalen Einheiten (FUs) parallel verarbeitet werden (Single Instruction Multiple Data (SIMD)), wodurch die Parallelisierung auf Datenebene ermöglicht wird (Data-Level Parallelism (DLP)). Die Hardwarebeschreibung der SIMD-FUs ist für die FPGA-Implementierung hinsichtlich Größe und Geschwindigkeit optimiert worden [70], beispielsweise durch explizite Verwendung der vorhandenen Carry-Chain-Logik im FPGA.

Der Datenpfad ist in fünf Pipelinestufen aufgeteilt: *instruction fetch*, *instruction decode*, *register access*, *execution* und *write back*. Operationen können auch zusätzliche Ausführungsstufen verwenden, wie in Abbildung 3.1 für die MAC-Einheit (*multiply and accumulate*) sowie eine rekonfigurierbare Einheit (RFU) dargestellt. Durch Aufspalten der Ausführungsstufen in Einheiten, die im kritischen Pfad liegen, kann die Gesamtfrequenz der Pipeline erhöht werden.

Der TUKUTURI-Prozessor unterstützt die bedingte Ausführung von Operationen (*predication*). Dazu werden Operationen im *condition set*-Modus ausgeführt, um neben dem Ergebnis der Operation auch *Flags* zu generieren. Eine Opera-

tion im *condition read*-Modus liest die Flags und eine Bedingung aus einem Spezialregister und führt die Operation dann nur für die Subworte aus, für die die Bedingung erfüllt ist. Für die anderen Subworte wird der erste Operand übernommen.

Die Flags werden auch für bedingte Sprünge (*branch*) verwendet. In der Branch-Operation ist neben dem Sprungziel angegeben, welche Bedingung geprüft werden soll. Ist die Bedingung erfüllt, wird der Sprung ausgeführt, andernfalls wird die Ausführung des Programms mit der auf die Branch-Operation folgenden Operation fortgesetzt.

3.1.2 Speicheranbindung

Der TUKUTURI-Prozessor besitzt einen lokalen Speicher (*data memory*) und kann über den On-Chip-Bus auf externe Speicher zugreifen.

Zum Zugriff auf den lokalen Datenspeicher, also den Datentransfer zwischen Speicher und Registerbänken, können direkte Lade- und Schreibbefehle (`LOAD`, `STORE`) verwendet werden, bei denen die Speicheradressen als Immediate-Werte direkt in der Operation kodiert werden. Mittels der File Indirect Register (FIR) können auch berechnete Adressen in indirekten Speicherzugriffen verwendet werden. Die Adressen in den FIR-Registern können beim Speicherzugriff außerdem inkrementiert und dekrementiert werden (*post-increment/post-decrement*), entweder durch kleine Immediate-Werte in der Operation oder durch einen größeren Wert, der in einem Offset-Spezialregister abgelegt ist.

Der Datentransfer zwischen externem und lokalem Speicher wird durch ein DMA-Modul realisiert. Viele Algorithmen der Signalverarbeitung, insbesondere bei der Bildverarbeitung, operieren auf Datenblöcken (z. B. rechteckige Bildausschnitte). Daher bietet das DMA-Modul 1-D, 2-D und 3-D Blocktransfers [77] an. Die Programmierung der Parameter eines Datentransfers erfolgt über Kontrollregister des DMA-Moduls, die in den Adressraum des Prozessors abgebildet sind (*memory-mapping*) und mit `STORE`-Operationen beschrieben werden können. Der Datentransfer wird im Hintergrund ausgeführt, während der Prozessor mit der Bearbeitung des Programms fortfährt. Das DMA-Modul kann mehrere Transfers in einer Warteschlange speichern, die dann sequentiell abgearbeitet werden. Über eine spezielle Bedingung kann der Status eines Speichertransfers aus der Warteschlange in Branch-Operationen abgefragt werden und so das Ende des Transfers festgestellt werden.

3.1.3 X2-Modus: Operation Merging

Die Effektivität eines Prozessors kann für viele Anwendungen der digitalen Signalverarbeitung, insbesondere im Bereich der Bildverarbeitung, durch stärkere Parallelität erhöht werden. Das Duplizieren von FUs kann zwar zu einer höheren

```
ADD_16     V0R0, V0R2, V0R4
ADD_16     V0R1, V0R3, V0R4
// äquivalent:
ADD_16_X2  V0R0+V0R1, V0R2+V0R3, V0R4

SUBI_8     V0R0, V0R2, #5
SUBI_8     V0R1, V0R3, #5
// äquivalent:
SUBI_X2    V0R0+V0R1, V0R2+V0R3, #5

MIXL       V0R0, V0R2, V0R3
MIXR       V0R1, V0R2, V0R3
// äquivalent:
MIXL_X2    V0R0+V0R1, V0R2, V0R3
```

Listing 3.1: Beispiele für äquivalente Operationen mit und ohne X2-Modus

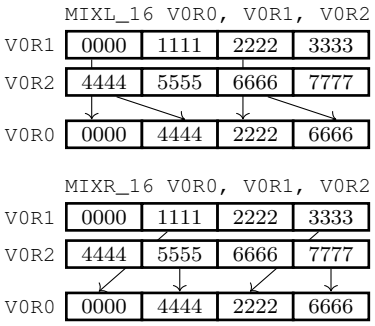


Abbildung 3.2: Mix-Operationen im TUKUTURI

Kompaktierung des Codes führen, doch um den Parallelisierungsgrad darüber hinaus zu erhöhen, sind weitere Maßnahmen erforderlich. In einem VLIW-Prozessor kann dies durch Bereitstellen weiterer Issue-Slots geschehen, womit eine höhere Anzahl von Operationen parallel ausgeführt werden kann, wodurch allerdings auch die Anbindung an den Instruktionsspeicher und insbesondere die Decodereinheit vergrößert wird. Aufgrund von erhöhtem Routing-Aufwand auf einem FPGA kann dies auch die Latenz der Schaltung erhöhen und damit die Gesamtleistung verringern.

Im TUKUTURI-Prozessor ist daher ein *Operation Merging* implementiert [80], wobei es sich um eine Methode handelt, mit der zwei gleiche Operationen, die auf unterschiedlichen Daten arbeiten, in einer Mikrooperation kodiert werden können und damit nur einen Issue-Slot belegen. Sind die benötigten Ressourcen vorhanden, können in jedem Issue-Slot solche fusionierten X2-Operationen ausgeführt werden, wodurch die effektive Anzahl parallel ausgeführter Operationen im günstigsten Fall verdoppelt werden kann, ohne die Anzahl zu dekodierender Operation zu erhöhen. Beispiele für äquivalente Operationen mit und ohne X2-Modus sind in Listing 3.1 gezeigt. In den Codebeispielen wird nach der Operation zunächst das Zielregister angegeben, danach folgen die Operanden. Register des Prozessors werden als `V`*r*`R`*n* geschrieben, wobei *r* die Registerbank angibt und *n* die Adresse des Registers in der entsprechenden Registerbank. Die Register eines X2-Operanden werden mit einem Plus verbunden (z. B. `V0R2+V0R3`). Immediate-Werte werden durch ein vorangestelltes Doppelkreuz (`#`) gekennzeichnet.

Um die Länge der Kodierung von X2-Operationen nicht zu erhöhen und die Auswirkung der Implementierung des X2-Modus gering zu halten, unterliegen die fusionierten Operationen verschiedenen Beschränkungen:

- Die Operationen müssen die gleichen Operationsmodi verwenden. Dies schließt den Subwortmodus, Vorzeichen (*signed*/*unsigned*) und die beding-

te Ausführung (*condition set*/*condition read*) ein. Falls die Operationen Immediate-Werte verwenden, müssen diese gleich sein. Dies hält den zusätzlichen Aufwand in der Decodereinheit gering.

- Die Daten, die von den Operationen verarbeitet werden, müssen in aufeinander folgenden Registern liegen, wobei die Registeradressen in der ersten Operation des fusionierten Paares gerade sein müssen, wodurch dann die Adressen der Register in der zweiten Operation ungerade sind. Dadurch wird zur Kodierung der Register in der Operation nur jeweils ein zusätzliches Bit benötigt, das angibt, ob der Operand den X2-Modus verwendet oder nicht. Im letzten Fall lesen beide Operationen des fusionierten Paares den Operanden aus demselben Register.

- Beim Speicherzugriff im X2-Modus müssen zwei aufeinander folgende Worte gelesen bzw. geschrieben werden, wobei die Adresse des ersten Wortes gerade sein muss. Dies kann sowohl mit direkten Speicheroperationen also auch über den indirekten Zugriff mit FIR-Registern geschehen. Der Speicherzugriff erfolgt in einem Taktzyklus, unabhängig davon, ob der X2-Modus verwendet wird, oder nicht.

Für manche Operationen ist auch ein erweiterter X2-Modus implementiert, in dem die beiden fusionierten Operationen die gleichen Operanden verwenden, dafür aber zwei verschiedene Varianten der Operation ausgeführt werden. Dies ist beispielsweise bei `MIX`-Operationen der Fall, die Subworte aus zwei Operanden in einem Register sammeln, und die in den Varianten `MIXL` und `MIXR` vorhanden ist, die jeweils die linke bzw. rechte Hälfte der Subworte verarbeitet (siehe Abbildung 3.2). Wird die `MIXL`-Operation im X2-Modus ohne X2-Operanden aufgerufen, enthält das erste Zielregister das Ergebnis von `MIXL` und das zweite das Ergebnis von `MIXR`.

Durch die höhere Anzahl an Operanden und Zielregistern steigt die Anzahl der von einer X2-Operation benötigten Ports zum Lesen und Schreiben in die Registerbänke. Die beschränkte Anzahl verfügbarer Ports erschwert durch Ressourcenkonflikte die Registerallokation, wie in Abschnitt 4.2.3 erläutert.

3.2 Statische Konfiguration der Softprozessorarchitektur

Das generische Architekturtemplate für den TUKUTURI-Prozessor erlaubt eine statische Anpassung des Prozessors an eine Anwendung zur Entwurfszeit. Die Möglichkeiten zur Konfiguration bezüglich minimalem Ressourcenaufwand und maximaler Frequenz werden in den nächsten Abschnitten beschrieben.

Tabelle 3.1: Parameter der TUKUTURI-Pipeline

Parameter	Beschreibung
DE	Anzahl zusätzlicher Decoderstufen
FWD	Zusätzliche Pipelinestufe für Forwarding
EX	Anzahl zusätzlicher Ausführungsstufen
FLOW	Zusätzliche Pipelinestufe für bedingte Sprünge
FIR_REG	Anzahl zusätzlicher Register zum Schreiben in FIR
BUSW	Zusätzliches Register beim Schreiben auf den Datenbus

3.2.1 Minimaler Ressourcenaufwand

Die Konfiguration des Prozessors erlaubt eine feingranulare Auswahl der funktionalen Einheiten (FUs), die in den Prozessor integriert werden. Die FUs selbst sind ebenfalls parametrisiert und können bezüglich der verfügbaren Modi (z. B. signed/unsigned, SIMD-Wortbreiten, konditionale Ausführung, Saturation) konfiguriert werden. Das Deaktivieren nicht benötigter Funktionalität verringert die Größe der FUs. In einer FPGA-Implementierung kann dies den Routing-Aufwand verringern und damit die Geschwindigkeit der Implementierung erhöhen. Die Möglichkeit der Konfiguration erlaubt die Verwendung einer anwendungsspezifischen Minimalkonfiguration des Prozessors, die vom Kompiler aus dem Anwendungscode abgeleitet werden kann. Somit kann der Entwickler bei der Optimierung des Assemblercodes gleichzeitig die Prozessorkonfiguration beeinflussen. Auch Befehlssatzerweiterungen (siehe Abschnitt 3.3) durch RFUs können als parametrisierte Implementierungen in der Minimalkonfiguration berücksichtigt werden, um dem Ressourcenbedarf der Prozessorimplementierung gering zu halten.

3.2.2 Maximale Frequenz

In dieser Arbeit wurde die Flexibilität der Pipeline durch weitere Parametrisierung erhöht. Die verfügbaren Parameter sind in Tabelle 3.1 aufgeführt. Durch die Verwendung zusätzlicher Pipelinestufen oder zusätzlicher Register in einzelnen Pfaden kann der kritische Pfad verkürzt und damit die Taktfrequenz des Prozessors erhöht werden. Wenn für den Code trotz erhöhter Latenzen zwischen den Operationen ein kompaktes Instruction Scheduling gefunden werden kann und so die negativen Auswirkungen der zusätzlichen Pipelinestufen auf die Platzierung der Operationen ausgeglichen werden können, erhöht dies die Gesamtperformanz des Systems.

Abbildung 3.3 zeigt eine detaillierte Darstellung der Pipeline des Prozessors zur Erläuterung der verschiedenen Parameter. Der exemplarische Assemblercode in Listing 3.2 wird verwendet, um die Auswirkungen der Parameter auf die

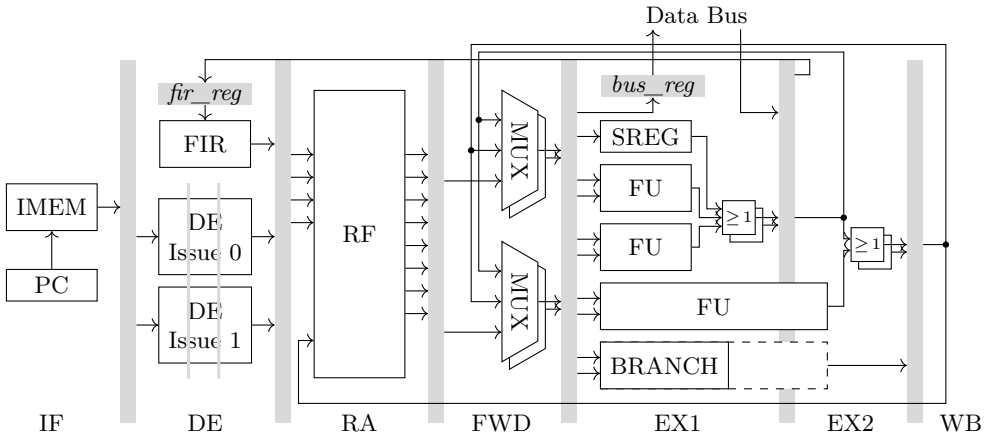


Abbildung 3.3: Parametrisierbare Pipeline der TUKUTURI-Architektur. Doppelt gezeichnete Elemente liegen jeweils pro Issue-Slot vor.

Ausführung der Operationen zu veranschaulichen. In Listing 3.3 ist der Instruction Schedule für die Basisarchitektur der Pipeline dargestellt, die keine zusätzlichen Stufen oder Register enthält. In Klammern eingeschlossene FIR-Register (z. B. `(FIR0)`) kennzeichnen einen indirekten Speicherzugriff. Steht ein Plus-Zeichen hinter der Klammer, wird die Speicheradresse im FIR-Register nach dem Zugriff inkrementiert (*post-increment*, siehe Abschnitt 3.1.2). Die `ELOOPR`-Operation bildet eine Schleife, deren Zähler hier in `V1R0` liegt. Es handelt sich um eine Verzweigung (*branch*), bei der zunächst der Wert des Zählers dekrementiert wird und anschließend ein Sprung zum angegebenen Label (`LBL`) erfolgt, sofern der Wert des Zählers größer oder gleich 0 ist. Da diese Sprungbedingung in der Ausführungsstufe (EX) ausgewertet wird, sind zu diesem Zeitpunkt bereits drei nachfolgende Instruktionen in die Pipeline geladen (in den Stufen IF, DE, RA). Diese drei als *Delay-Slots* bezeichneten Zyklen werden vom Instruction Scheduler mit der ELOOPR-Operation im sequentiellen Programmcode vorhergehenden, datenunabhängigen Operationen gefüllt, wenn dies möglich ist. In Listing 3.3 sind dies die Operationen aus den Zeilen 3, 7 und 8.

Die folgende Auflistung erklärt die Auswirkungen der verschiedenen Parameter auf die Ausführung der Operationen.

- Zusätzliche Decoderstufen (Parameter `DE`; in Abbildung 3.3 als zusätzliche Pipelineregister in den DE-Einheiten dargestellt) verringern den kritischen Pfad für die Dekodierung von Instruktionen, führen jedoch auch dazu, dass die Anzahl der Delay-Slots von Sprungbefehlen (bedingte und unbedingte Sprünge) steigt, wie in Listing 3.4 gezeigt.

- Die Pipeline kann mit einer zusätzlichen Stufe für das Forwarding konfigu-

```
1  :L_LBL
2  ADD     VxR0, V0R1, V0R2
3  MUL     VxR1, VxR0, V0R3
4  SUB     VxR2, V0R4, V0R5

6  SMVI    FIR0, #10
7  MV      (FIR0)+, VxR2
8  MV      VxR3, (FIR0)

10 ELOOPR V1R0, LBL, #0
```

Listing 3.2: Exemplarischer Assemblercode

```
:L_LBL
 6 SMVI   FIR0, #10     ; 4 SUB  V1R2, V0R4, V0R5
10 ELOOPR V1R0, LBL, #0 ; 2 ADD  V1R1, V0R1, V0R2
 0 NOP                  ; 3 MUL  V1R1, V1R1, V0R3
 7 MV     (FIR0)+, V1R2 ; 0 NOP
 8 MV     V1R1, (FIR0)  ; 0 NOP
```

Listing 3.3: Kompiliert für die Basiskonfiguration

```
:L_LBL
10 ELOOPR V1R0,    LBL, #0    ; 6 SMVI FIR0, #10
 2 ADD    V1R1,    V0R1, V0R2 ; 0 NOP
 3 MUL    V1R1,    V1R1, V0R3 ; 4 SUB  V1R2, V0R4, V0R5
 7 MV     (FIR0)+, V1R2       ; 0 NOP
 8 MV     V1R1,    (FIR0)     ; 0 NOP
```

Listing 3.4: Kompiliert für eine Architektur mit zusätzlicher Decoder-Stufe. Die Branch-Operation besitzt vier Delay-Slots.

riert werden (Parameter `FWD`). Hier wird entschieden, ob die Operanden einer Operation aus Registern oder von den Forwarding-Pfaden übernommen werden. Diese Stufe führt dazu, dass die Ergebnisse aller FUs um einen Taktzyklus verzögert werden. Dies ist in Listing 3.5 am Beispiel der `MUL`-Operation zu sehen, die zwei Zyklen nach der `ADD`-Operation platziert wurde, von der sie über das Register `V1R1` abhängt. Auch für Sprungbefehle erhöht sich die Anzahl der Delay-Slots. Zwischen einer `STORE`-Operation, die einen DMA-Transfer startet und einem Branch, der den Transferstatus abfragt (siehe Abschnitt 3.1.2), wird die Latenz ebenfalls erhöht.

- Die FUs des TUKUTURI können mit einer unterschiedlichen Anzahl von Zyklen zur Berechnung eines Ergebnisses konfiguriert werden, wie in Abbildung 3.3 exemplarisch für FUs mit 1 bzw. 2 Stufen gezeigt. Die Pipeline wird über den Parameter `EX` mit der benötigten Anzahl von Ausführungsstufen versehen. Für die entsprechenden Einheiten erhöht sich die Latenz der Operationen, wie in Listing 3.6 für die arithmetische Einheit gezeigt. Hier kann die `MUL`-Operation erst zwei Zyklen nach der `ADD`-Operation platziert werden, von deren Ergebnis sie abhängt. Da die arithmetische Einheit interne Pipelineregister besitzt, kann die `SUB`-Operation direkt nach der Addition gestartet werden und diese überlappen. Für jede Ausführungsstufe gibt es pro Issue-Slot einen Forwarding-Pfad, wodurch die Anzahl der Eingänge zu den Multiplexern in der Forwarding-Stufe und damit auch deren Größe steigt. Dadurch kann eine hohe Anzahl von Ausführungsstufen

```
:L_LBL
 6 SMVI   FIR0, #10     ; 2 ADD V1R1, V0R1, V0R2
 0 NOP                  ; 4 SUB V1R2, V0R4, V0R5
10 ELOOPR V1R0, LBL, #0 ; 3 MUL V1R1, V1R1, V0R3
 0 NOP                  ; 0 NOP
 7 MV     (FIR0)+, V1R2 ; 0 NOP
 8 MV     V1R1, (FIR0)  ; 0 NOP
 0 NOP                  ; 0 NOP
```

Listing 3.5: Kompiliert für eine Architektur mit zusätzlicher Forwarding-Stufe. Die Latenzen der Operationen und die Delay-Slots der Sprungoperationen werden erhöht.

```
:L_LBL
 6 SMVI   FIR0, #10     ; 2 ADD V1R1, V0R1, V0R2
10 ELOOPR V1R0, LBL, #0 ; 4 SUB V1R2, V0R4, V0R5
 0 NOP                  ; 3 MUL V1R1, V1R1, V0R3
 7 MV     (FIR0)+, V1R2 ; 0 NOP
 8 MV     V1R1, (FIR0)  ; 0 NOP
```

Listing 3.6: Kompiliert für eine Architektur mit zwei Ausführungsstufen in der arithmetischen Einheit (`ADD`, `SUB`).

die Latenz der Forwarding-Stufe in der FPGA-Implementierung aufgrund des komplizierteren Routings erhöhen.

- Die Auswertung der Bedingung in bedingten Sprungbefehlen kann auf zwei Pipelinestufen ausgeweitet werden (Parameter `FLOW`). Dies erhöht die Anzahl der Delay-Slots in bedingten Sprüngen (Branch, Schleifen), aber nicht in unbedingten Sprüngen.
- Beim Schreiben von Adressen in die FIR-Register kann ein zusätzliches Register eingefügt werden (Parameter `FIR_REG`; in Abbildung 3.3 als *fir_reg* eingezeichnet). Dies erhöht die Latenz entsprechender Schreibbefehle (`SMV`). Auf das Inkrementieren und Dekrementieren der Adressen beim indirekten Speicherzugriff hat dies keine Auswirkung.
- Beim Schreiben auf den internen Datenbus, also für Zugriffe auf den lokalen Speicher oder die *memory mapped* Kontrollregister von DMA oder Coprozessoren, kann ein zusätzliches Register eingefügt werden (Parameter `BUSW`; in Abbildung 3.3 als Register *bus_reg* dargestellt). Dieses erhöht die Latenz von `STORE`-Operationen oder indirekten Schreibzugriffen über FIR-Register.

3.3 Befehlssatzerweiterungen

Zur Anpassung des Softprozessors an die Anforderungen der Anwendung kann der Befehlssatz des Prozessors erweitert werden. Der TUKUTURI-Prozessor

bietet dazu zwei verschiedene Mechanismen an, die in den folgenden Abschnitten erläutert werden. Die Kopplung der Befehlssatzerweiterungen mit der Prozessorarchitektur ist in Abschnitt 2.1.2 erläutert.

3.3.1 Komplexe funktionale Einheiten

Der Softprozessor kann durch funktionale Einheiten erweitert werden, die von einer Anwendung häufig verwendete komplexe Operationen in einer effizienten Hardwareimplementierung bereitstellen. Dies führt zu einer schnelleren Ausführung des Programms im Vergleich zur Beschreibung komplexer Operationen durch Sequenzen von Operationen des Basisinstruktionssatzes.

Im Instruktionssatz werden die erweiterten funktionalen Einheiten analog zu den Basisoperationen kodiert. In der Standardkodierung stehen dafür 16 Instruktionen bereit. Dadurch können die damit beschriebenen Operationen zwei Operanden lesen und ein Ergebnis schreiben.

Die komplexen FUs werden in die Pipeline integriert und können 1 bis 3 Taktzyklen für die Berechnung der Operation verwenden. Es stehen alle Bits aus der Kodierung der Operationen zur Verfügung, so dass auch in diesen Operationen verschiedene Subwortmodi oder vorzeichenlose/-behaftete Modi verwendet werden können. Ebenso können die Flags sowohl gelesen als auch geschrieben werden, und die Operationen können damit im *condition read/condition set* Modus verwendet werden. Die Bits können in den Operationen aber auch auf andere Weise interpretiert werden, um spezielle Modi zu implementieren, beispielsweise zur Initialisierung oder zum Zurücksetzen einer FU, die einen internen Zustand besitzt.

Ein Beispiel für eine komplexe Operation, die durch eine funktionale Einheit bereitgestellt wird, ist die *Count Leading Zero, CLZ* Operation. Diese bestimmt für einen Eingabeoperanden die Anzahl der führenden Nullen und kann damit die Größenordnung einer Zahl abschätzen, was zum Beispiel bei bestimmten CORDIC-Implementierungen verwendet wird [71]. In Hardware kann diese Funktionalität in einem Taktzyklus realisiert werden, eine Implementierung mit dem Basisinstruktionssatz hingegen wäre deutlich länger.

3.3.2 Coprozessoren

Coprozessoren können für komplexe Operationen verwendet werden, die nicht in die Pipeline integriert werden, beispielsweise aufgrund der Anzahl benötigter Taktzyklen. Jeder Coprozessor besitzt Kontrollregister, die in den Addressraum des Prozessors abgebildet sind, so dass mit `LOAD`- und `STORE`-Operationen auf den Coprozessor zugegriffen werden kann.

Das Schreiben in bestimmte Kontrollregister kann die Verarbeitung der Daten anstoßen. Nach der vom Coprozessor benötigten Latenz können die Ergebnisse

dann aus Ergebnisregistern gelesen werden. Coprozessoren können ebenfalls eine interne Pipeline verwenden, so dass mehrere Berechnungen überlappt werden können. Dann ist eine präzise Anordnung der `LOAD`- und `STORE`-Operationen notwendig, um die Latenz des Coprozessors und die Reihenfolge der Zugriffe korrekt einzuhalten. Zu diesem Zweck sind verschiedene Pseudo-Operationen in dem Instruction Scheduler für den TUKUTURI-Prozessor (siehe Kapitel 4) implementiert, für die zum einen die Latenz flexibel im Assemblercode angegeben wird und zum anderen spezielle Abhängigkeiten zwischen aufeinander folgenden `STORE`-bzw. `LOAD`-Operationen für eine korrekte Einhaltung der Reihenfolge und eine korrekte Überlappung sorgen (siehe auch Abschnitt 4.2.1).

Ein Beispiel für einen Coprozessor ist das in [71] vorgestellte CORDIC-Modul, das verschiedene arithmetische Funktionen (Sinus/Kosinus, Wurzel, Exponentialfunktion) bereitstellt. Der Coprozessor startet die iterative Berechnung, sobald Operanden geschrieben werden. Nach einer Mindestlatenz von drei Taktzyklen kann das Ergebnis gelesen werden. Wird das Ergebnis erst später gelesen, führt der Coprozessor weitere Iterationen aus und erzielt dadurch eine höhere Genauigkeit. Um die Anforderungen der Anwendung an die Genauigkeit der Ergebnisse zu berücksichtigen, kann im Assemblercode eine Pseudo-`STORE`-Operation verwendet werden, bei der die Mindestlatenz bis zur nächsten `LOAD`-Operation direkt im Assemblercode festgelegt werden kann.

Ein weiteres Beispiel ist ein Coprozessor für Fixpunkt-Divisionen, der durch die interne Pipeline ein Überlappen mehrerer Divisionen erlaubt. Der Code in Listing 4.2 zeigt die Verwendung von `STORERCUPPL` Pseudo-Operationen (*store into pipelined RCU*). Durch die spezielle Definition von Abhängigkeiten von diesen Operationen mit nachfolgenden `LOAD`-Operationen kann der Instruction Scheduler die Divisionen überlappen und gleichzeitig die notwendige Latenz einhalten, wie in Listing 4.3 gezeigt.

3.4 Dynamische partielle Rekonfiguration

Wie in Abschnitt 2.1.2 beschrieben, beschleunigt die Erweiterung des Befehlssatzes eines Prozessors durch neue funktionale Einheiten oder Coprozessoren zwar die Ausführung eines Programms, führt aber auch zu einem erhöhten Hardwareaufwand. Eine mögliche Lösung stellt die *dynamische partielle Rekonfiguration* dar, mit der FUs und Coprozessoren nach Bedarf zur Laufzeit rekonfiguriert werden können, um nicht mehr benötigte Hardwareeinheiten zu ersetzen. Damit kann der Prozessor mit einer vergleichsweise kleinen Anzahl an RFUs und Coprozessoren konfiguriert werden, die als Platzhalter für konkrete FUs und Coprozessoren dienen. So wird trotz vieler spezialisierter Operationen zur beschleunigten Datenverarbeitung eine effiziente Abbildung auf das FPGA ermöglicht.

```
// Rekonfiguriere RFU1
STOREI  ADDR, #0x480c    // externe Speicheradresse
STOREI  LEN,  #0x600000  // Datenlänge
STOREI  DPR_START, #1    // Start der Rekonfiguration

// Subroutine, die RFU1 nicht verwendet
JLR     V0R0, SUBOUTINE_A

// Warte auf Ende der Rekonfiguration
:L_WAIT
BSR     WAIT, #DPR_FLAG, #0x1
```

Listing 3.7: Exemplarischer Assemblercode zur Verwendung des DMA-ICAP-Moduls

Die Dauer der Rekonfiguration hängt direkt von der Größe des rekonfigurierten Bereichs ab. Für einen Satz von FUs, die in einer Partially Reconfigurable Region (PRR) platziert werden sollen, muss diese Region so groß gewählt werden, dass sie die größte FU aufnehmen kann. Daher ist eine Optimierung der FUs sinnvoll, um deren Größe und die zur Rekonfiguration benötigte Zeit zu reduzieren.

3.4.1 Dynamische Selbstrekonfiguration: DMA-ICAP

Die Rekonfiguration eines Xilinx FPGA erfolgt über den ICAP [108], dem ein partieller Bitstream für einen Bereich des FPGA übergeben wird und der die notwendigen Änderungen an der Konfiguration des FPGA vornimmt. Der TUKUTURI greift indirekt auf dieses Modul zu, um eine *Selbstrekonfiguration* durchzuführen. Dazu ist ein DMA-ICAP-Modul integriert (siehe Abbildung 3.1 [79]), das über Konfigurationsregister angesteuert wird, und das die partiellen Bitstreams aus dem externen Speicher liest, um sie an das ICAP-Modul zu übertragen.

Ein exemplarischer Assemblercode zur Ansteuerung des DMA-ICAP-Moduls ist in Listing 3.7 gezeigt. Zunächst werden die Adresse und die Länge des partiellen Bitstreams im externen Speicher gesetzt. Danach wird die Rekonfiguration durch Schreiben in die Trigger-Adresse gestartet. Das DMA-ICAP-Modul rekonfiguriert die Einheit im Hintergrund, während der TUKUTURI-Prozessor beispielsweise eine Subroutine ausführt. Das Register `V0R0` wird hier zur Speicherung der Rücksprungadresse für den Sprung zur Subroutine (`JLR`) verwendet. Der Status der Rekonfiguration kann mithilfe einer speziellen Bedingung in einer Branch-Operation (`BSR`) abgerufen werden, so dass der Prozessor auf den Abschluss der Rekonfiguration einer Einheit warten kann, bevor sie verwendet wird. Das DMA-ICAP-Modul kann mehrere Transfers von Bitstreams an das ICAP-Modul in einer Warteschlange speichern, die dann nacheinander abgearbeitet werden.

Durch geschickte Programmierung des DMA-ICAP können die Rekonfigurationen im Hintergrund stattfinden, während der Prozessor parallel das Programm abarbeitet, da der TUKUTURI über zwei unabhängige Ports auf den externen

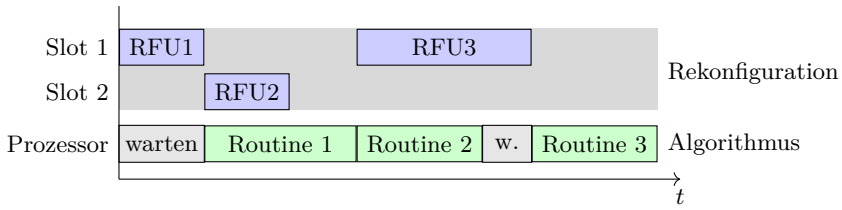


Abbildung 3.4: Zeitlicher Ablauf einer exemplarischen dynamischen Rekonfiguration von verschiedenen RFUs in zwei RFU Slots. Jede Routine verwendet eine spezifische RFU. Die Rekonfigurationen erfolgen parallel zur Datenverarbeitung im Prozessor. Durch lange Rekonfigurationen können Verzögerungen in der Datenverarbeitung entstehen.

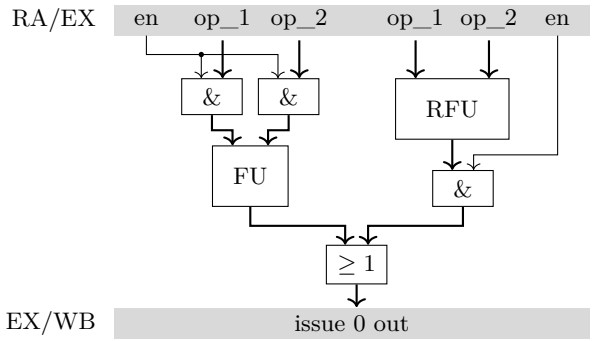


Abbildung 3.5: Isolierung von funktionalen Einheiten in der TUKUTURI-Pipeline [79]

Speicher zugreifen kann, in dem die partiellen Bitstreams und die zu verarbeitenden Daten liegen. Dazu ist in Abbildung 3.4 ein exemplarischer Ablauf gezeigt, bei dem der Prozessor drei Routinen abarbeiten muss, die jeweils eine spezifische RFU verwenden. Der Prozessor bietet dazu zwei rekonfigurierbare Slots an. Zunächst muss die erste RFU in den ersten Slot rekonfiguriert werden. Da diese RFU in Routine 1 verwendet wird, verzögert sich die Bearbeitung dieser Routine durch den Prozessor. Während Routine 1 abgearbeitet wird, kann der zweite Slot mit RFU 2 rekonfiguriert werden, die in Routine 2 benötigt wird. Kann diese Rekonfiguration schneller erfolgen als die Abarbeitung von Routine 1, kann im Anschluss Routine 2 ohne zusätzliche Wartezeit starten. Da nun der erste Slot nicht mehr verwendet wird, kann dort RFU 1 durch RFU 3 ersetzt werden, die in Routine 3 verwendet wird. Dauert diese Rekonfiguration länger als die Ausführung von Routine 2, muss vor Beginn von Routine 3 auf den Abschluss der Rekonfiguration gewartet werden. Die Programmierung der Rekonfigurationen erfolgt manuell durch den Programmierer und muss entsprechend platziert werden, damit sie möglichst vollständig im Hintergrund stattfinden kann, ohne zusätzliche Wartezeit einzufügen. Für eine exemplarische Anwendung auf dem TUKUTURI-Prozessor ist dies in Abschnitt 6.4 dargestellt und evaluiert.

Während der Rekonfiguration eines Bereichs des FPGA können die anderen Bestandteile des Prozessors weiter verwendet werden. Dazu müssen die RFUs vom Rest des Prozessors isoliert werden, damit während der Rekonfiguration keine ungültigen Daten in die Pipeline gelangen. Der Mechanismus basiert darauf, dass alle nicht aktiven FUs und RFUs eine Null als Ergebnis erzeugen, wie in Abbildung 3.5 gezeigt [79]. Für nicht rekonfigurierbare FUs wird eine UND-Verknüpfung mit einem *enable*-Signal für die Eingänge verwendet. Diese Struktur stammt aus der ASIC-Implementierung der generischen Architektur, wo sie zur Reduktion der Switching-Activity eingesetzt wird. Diese Möglichkeit der Isolation kann für rekonfigurierbare Einheiten nicht eingesetzt werden, da Signaländerungen während der Rekonfiguration auftreten können, die sich auf das Ausgangssignal der Einheit bzw. des rekonfigurierten Bereichs auswirken. Daher wird bei rekonfigurierbaren RFUs das *enable*-Signal mit dem Ausgang der Einheit UND-verknüpft, um den rekonfigurierbaren Bereich vom Rest der Pipeline zu trennen.

3.4.2 Adaptive dynamische Rekonfiguration (MIPS)

Die Rekonfiguration von Hardwareeinheiten im TUKUTURI-Prozessor wird von dem Anwendungsprogramm durch entsprechende Programmierung des DMA-ICAP-Moduls vorgegeben. Daher muss die Planung der Rekonfiguration statisch zur Entwicklungszeit der Anwendung erfolgen und ist daher für Anwendungen geeignet, deren Ablauf ebenfalls statisch ist und nicht datenabhängig.

Für eine bessere Adaption des Prozessors mit Befehlssatzerweiterungen an eine dynamische Anwendung kann auch die Rekonfiguration der Hardwareeinheiten dynamisch vom Prozessor vorgenommen werden. Dazu wurde in [73] die r-MIPS-Architektur vorgestellt, die einen MIPS-Softprozessor um eine *adaptive dynamische Rekonfiguration* erweitert.

Das *Dynamic Partial Reconfiguration System (DPR-System)* kontrolliert rekonfigurierbare Coprozessoren, die zur Laufzeit in vorhandene Partially Reconfigurable Regions (PRRs) geladen werden können. Dies wird durch eine *Software-Hardware-Substitution* ermöglicht. Dazu wird eine Bibliothek bereitgestellt, die sowohl Software- als auch Hardwareimplementierungen von rechenintensiven Funktionen enthält. Die Bibliothek kann jederzeit um neue Funktionen und passende Hardwaremodule erweitert werden. Der Entwickler nutzt bei der Programmierung des r-MIPS Wrapper-Funktionen, die standardmäßig die entsprechenden Softwarefunktionen aus der Bibliothek aufrufen. Sollte der Coprozessor einer Funktion bereits rekonfiguriert worden sein, substituiert das DPR-System den Funktionsaufruf zur Laufzeit, um statt der Softwareimplementierung eine Funktion zur Ansteuerung des Coprozessors aufzurufen. Die Substitution erfolgt automatisch anhand der Adressen der Wrapper-Funktionen und wird damit transparent für den Entwickler durchgeführt.

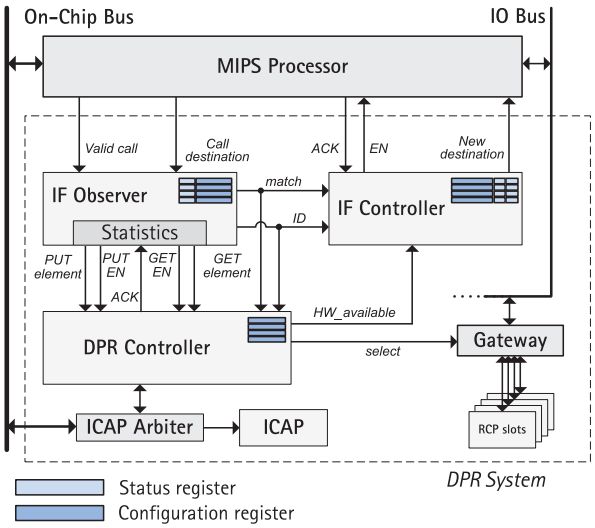


Abbildung 3.6: Observer-Constroller-Architektur des DPR-Systems für den rekonfigurierbaren MIPS Prozessor [73]

Da nur eine begrenzte Anzahl von PRRs zur Verfügung steht, ist die Zahl der parallel zur Verfügung stehenden Coprozessoren ebenfalls beschränkt. Daher muss das DPR-System eine Strategie zur Ersetzung rekonfigurierbarer Coprozessoren enthalten. Das DPR-System verwendet eine *observer-controller-Architektur*, wie in Abbildung 3.6 dargestellt. Ein *IF Observer* beobachtet Funktionsaufrufe in der Instruction-Fetch-Phase und führt eine Statistik über die Aufrufe der Wrapper-Funktionen. Bei jedem Aufruf eine Wrapper-Funktion, für die der entsprechende Coprozessor geladen ist, wird die SW-HW-Substitution durchgeführt. Dazu modifiziert der *IF Controller* die CPU-Pipeline, um die bereits begonnene Berechnung der reinen Softwareimplementierung abzubrechen und den Kontrollfluss auf die Funktion zur Ansteuerung der Hardware umzuleiten. Mithilfe der vom IF Observer generierten Statistiken entscheidet der *DPR Controller* über die Rekonfiguration der verfügbaren PRRs. Die Rekonfiguration kann im Hintergrund stattfinden, während der MIPS-Prozessor weiter arbeitet.

Verschiedene Ersetzungsstrategien für Coprozessoren sind untersucht worden. Einige von Speicher-Caches bekannte Methoden, wie First-In First-Out (FIFO), Random (RND) oder Least-Recently-Used (LRU), führen eine Rekonfiguration durch, sobald eine Wrapper-Funktion aufgerufen wird, für die der Coprozessor nicht aktiv ist. Der Coprozessor, der auf eine neue Funktion rekonfiguriert wird, ist der älteste (FIFO), ein zufällig ausgewählter (RND) oder der am längsten nicht verwendete (LRU). Bei dieser Rekonfiguration nach Bedarf (on-demand) wird eine Rekonfiguration auch dann durchgeführt, wenn der Coprozessor nur ein einziges Mal verwendet wird.

Eine verbesserte Strategie ist die *gewichtete Ersetzungsstrategie mit Hysterese (WRP)*, deren Statistik den theoretisch zu erwartenden Gewinn einer Rekonfiguration ermittelt. Bezeichnet N_x bzw. N_y die Anzahl der Aufrufe einer Funktion x bzw. y, und sei der Coprozessor für Funktion x geladen, der für Funktion y jedoch nicht und sei weiterhin SW_x bzw. HW_y die Anzahl an Taktzyklen zur Berechnung von x bzw. y in Software bzw. Hardware, dann kann die Bedingung, ob der Coprozessor für y den für x ersetzen soll angegeben werden als

$$N_x(SW_x - HW_x) < N_y(SW_y - HW_y).$$

Mit einem Gewichtungsfaktor W_{xy}, berechnet als

$$W_{xy} = \frac{SW_x - HW_x}{SW_y - HW_y},$$

kann die Bedingung vereinfacht werden zu

$$N_x W_{xy} < N_y.$$

Um das abwechselnde Austauschen von Coprozessoren zu verhindern, kann ein Hystereseparameter H hinzugefügt werden, der die Häufigkeitszähler N für Funktionen virtuell erhöht/verringert. Durch positive H wird die Rekonfiguration einer Einheit erschwert, durch negative H gefördert. Die Bedingung zur Rekonfiguration hat dann diese Gestalt:

$$N_x W_{xy} < N_y - H.$$

Damit die Verwendungsstatistiken der Einheiten nur die *nähere Vergangenheit* beschreiben, werden die Zählerstände nach Ablauf eines Timers halbiert. Eine Erweiterung des Systems verwendet einen modifizierten *Leat-Mean-Squares* Algorithmus (LMS), um die Laufzeit des Timers und den Wert H der Hysterese dynamisch anzupassen. Abbildung 3.7 zeigt, dass für eine exemplarische Anwendung aus der Robotik die WRP Strategie im Vergleich zu anderen Ersetzungsstrategien zu deutlich weniger Rekonfigurationen führt. Dadurch kann sogar die Anzahl der Zyklen zur Datenverarbeitung gesenkt werden, da die häufig verwendeten Coprozessoren weniger häufig ersetzt werden, sondern zur Benutzung bereitstehen [73]. Im gezeigten Beispiel stehen sechs verschiedene Einheiten und zwei Slots für die Rekonfiguration zur Verfügung.

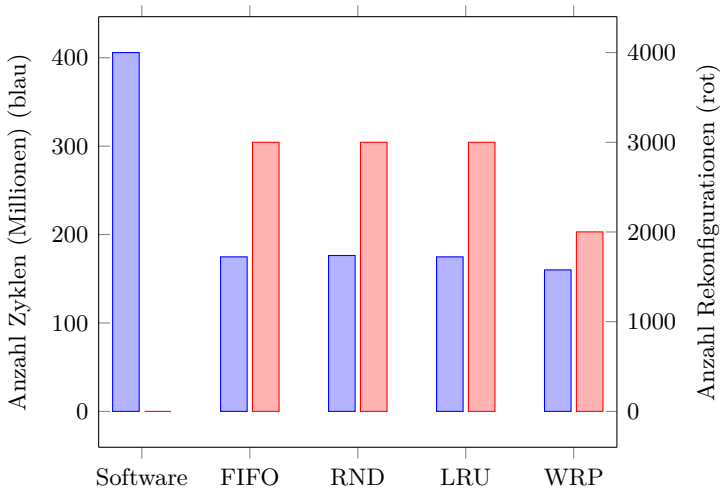


Abbildung 3.7: WRP-Strategie im Vergleich zu anderen Ersetzungsstrategien am Beispiel einer Anwendung der Robotik. Durch seltenere Ersetzung der verwendeten Coprozessoren kann die Zyklenzahl für die Anwendung gesenkt werden. [73]

4 Kompilierung für TUKUTURI basierend auf evolutionären Algorithmen

In Abschnitt 2.2.1 wurde der grundlegende Ablauf der Kompilierung eines Programms in ausführbaren Binärcode in den Phasen Analyse und Synthese vorgestellt. Das Hauptaugenmerk in diesem Kapitel liegt auf der Synthesephase im so genannten Back-End (auch Codegenerierung genannt) des Kompilers für die in Kapitel 3 vorgestellte generische VLIW-Architektur.

Der folgende Abschnitt liefert Definitionen und Beschreibungen, die in den darauf folgenden Abschnitten zur Erläuterung der Implementierungen der Codegenerierung mittels statischen und dynamischen EA-basierten Heuristiken verwendet werden.

4.1 Generische Codegenerierung für die TUKUTURI-Architektur

Da die TUKUTURI-Architektur, die für diesen Codegenerator als Zielarchitektur dient, flexibel konfiguriert werden kann, muss auch der Codegenerator diese Flexibilität aufweisen. Dazu werden die für die Codegenerierung wesentlichen Parameter der Architektur in einer Konfigurationsdatei beschrieben, die vor der Kompilierung vom Codegenerator eingelesen wird. Dies umfasst die Registerkonfiguration, also die Anzahl der Registerbänke, die zur Verfügung stehenden Schreib- und Leseports pro Registerbank, sowie die Anzahl der Register in den Registerbänken. Weiterhin werden die zur Verfügung stehenden funktionalen Einheiten (auch deren Anzahl) und die darin bereitgestellten Operationen, sowie die vorhandenen Spezialregister beschrieben. Die Konfigurationsdatei enthält keine detaillierte Beschreibung der Funktionalität, gibt aber an, welche Operanden oder Ressourcen gelesen bzw. geschrieben werden, um daraus Datenabhängigkeiten und Konflikte ableiten zu können. Ebenso wird die Binärkodierung der Operationen beschrieben, die der Codegenerator für die Ausgabe des Binärprogramms benötigt. Schließlich definiert die Prozessorkonfiguration auch die lokalen Speicher der Architektur, also die Anzahl der Speicher und der Schreib- und Leseports.

Die Transformationen in der Codegenerierung zielen häufig auf eine Kompaktierung des Programms in möglichst wenige Mikroinstruktionen ab, da die Größe des Mikroprogramms direkten Einfluss auf die Ausführungsgeschwindigkeit hat. Da eine Mikroinstruktion pro Taktzyklus gestartet wird, wirkt sich insbesondere die Kompaktierung von basic blocks, die innerhalb von Schleifen mehrfach ausgeführt werden, positiv auf die Ausführungszeit aus. Um die Möglichkeiten zur Parallelisierung zu verbessern, können Optimierungen an den MOs vorgenommen werden, wie in Abschnitt 4.2 erläutert. So können beispielsweise die meisten Operationen nicht nur auf Registeroperanden arbeiten, sondern auch Immediate-Werte verarbeiten, die direkt in der Mikrooperation kodiert werden. Dazu stehen, je nach Operation, nur wenige Bits im Operationswort zur Verfügung. Sollen größere Werte verarbeitet werden, für die die vorhandenen Bits nicht zur Kodierung ausreichen, können lange Immediate-Werte verwendet werden, die 32 bit umfassen. Die Kodierung dieser Immediate-Werte erfolgt dann in dem auf die Operation folgenden Issue-Slot, so dass dort keine weitere Operation kodiert werden kann. Der Kompiler kann vor dem Instruction Scheduling eine Analyse der Immediate-Werte durchführen und lange Immediates, die auch in wenigen Bits kodiert werden können, entsprechend umwandeln, so dass der folgende Issue-Slot zur Aufnahme einer weiteren Operation frei wird. Komplexere Transformationen werden in Abschnitt 4.2.2 gezeigt.

Der Kompiler für die TUKUTURI-Architektur erlaubt die Verwendung von *virtuellen Registern*, die in beliebig großer Zahl vorhanden sind. Im Assemblercode werden virtuelle Register in der Form `VxR`z angegeben, wobei z eine beliebige Zahl ist. Die Gültigkeit bzw. Lebensdauer der virtuellen Register ist auf einzelne SLMs beschränkt. Bei der Registerallokation in diesem Codegenerator wird kein *spill code* generiert, um Werte in den lokalen Speicher auszulagern, wenn nicht genügend Hardwareregister vorhanden sind. Die Behandlung dieses Falls wird in den späteren Abschnitten dieses Kapitels erläutert.

4.2 Codegenerierung mit klassischen statischen Heuristiken

Die Codegenerierung für VLIW-Architekturen umfasst verschiedene Aufgaben wie Codeoptimierungen, Instruction Scheduling und Registerallokation, wie in Abschnitt 2.2.1 beschrieben. Dabei handelt es sich um derart komplexe Optimierungsaufgaben, dass eine exakte Lösung meist nur für sehr kleine Probleminstanzen gefunden werden kann, für die die Auflistung aller möglichen Lösungen (eine erschöpfende Suche) durchführbar ist. Auch andere Optimierungsverfahren, die keine erschöpfende Suche durchführen, sind für große Probleminstanzen häufig nicht erfolgreich. Daher werden üblicherweise Algorithmen eingesetzt, die die Anzahl der auszuwertenden Lösungen mithilfe von Heuristiken reduzieren.

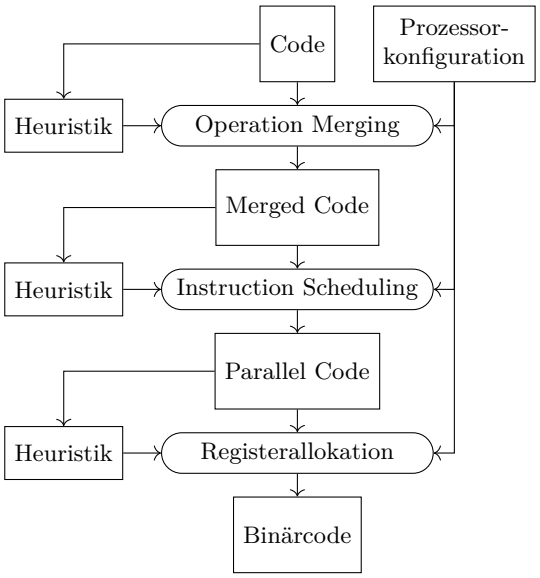


Abbildung 4.1: Codegenerierung mittels statischer Heuristiken

Eine Heuristik ist eine Funktion, die durch Analyse der Probleminstanz abschätzt, ob eine mögliche Lösung für das Optimierungsproblem günstig ist, ohne sie tatsächlich vollständig auszuwerten. Beispielsweise werden Informationen aus der Abhängigkeitsanalyse der Operationen abgeleitet, die den Schedulingalgorithmus dirigieren, oder es fließen Eigenschaften der Registerverteilung in die Registerallokation ein. Die Güte der Heuristik hat damit großen Einfluss auf die Qualität der Lösung und die Performanz des Algorithmus. Ist die Heuristik zu eng gewählt und schließt viele mögliche Lösungen als ungünstig aus, wird die Optimierung beschleunigt, kann aber möglicherweise keine guten Ergebnisse finden. Werden zu wenig ungünstige Lösungen von der Heuristik ausgeschlossen, steigt damit die Rechenzeit während der Optimierung. Das Finden guter Heuristiken ist selbst eine komplexe Aufgabe, die häufig manuell durchgeführt wird.

Die hier vorgestellte Codegenerierung ist schematisch in Abbildung 4.1 dargestellt. Sie umfasst das Instruction Scheduling und die Registerallokation für die TUKUTURI-Architektur, wonach die Binärkodierung und Ausgabe des Programms erfolgt. Als optionale Vorverarbeitung kann vor dem Instruction Scheduling ein automatisches Operation Merging durchgeführt werden. Die Algorithmen für Operation Merging, Instruction Scheduling und Registerallokation lesen das Eingabeprogramm bzw. die Ergebnisse der vorherigen Kompilerstufe und verwenden die Beschreibung der Prozessorkonfiguration und statische Heuristiken, um schlussendlich ein Binärprogramm auszugeben. Die folgenden Abschnitte beschreiben die einzelnen Algorithmen und die verwendeten statischen Heuristiken.

4.2.1 Instruction Scheduling

Nach dem Parsen des Eingabeprogramms liegt eine Sequenz von Mikrooperationen (MOs) vor, wie sie der Programmierer geschrieben hat. Das Instruction Scheduling erzeugt daraus einen parallelen (horizontalen) Code, indem mehrere MOs in einem Instruktionswort (Mikroinstruktion, MI) zusammengefasst werden. Das in diesem Abschnitt behandelte Verfahren zum Instruction Scheduling ist ein *lokales* Instruction Scheduling, bei dem die Sequenz der Operationen in SLMs (nach Definition 2.2) aufgeteilt wird, die unabhängig voneinander verarbeitet werden. Der Kompiler verwendet einen List Scheduling Algorithmus [26] mit einer statischen Heuristik zur Auswahl der Scheduling-Kandidaten und basiert auf [83], das wiederum eine Weiterentwicklung von [82] ist.

Vorverarbeitung des Codes

Bevor Mikrooperationen in Instruktionswörtern (MIs) zusammengefasst werden können, müssen zunächst Abhängigkeiten zwischen den MOs analysiert werden, da diese die möglichen Anordnungen (Reihenfolge) der MOs im schedulierten Programm einschränken. Abhängigkeiten entstehen, wenn verschiedene MOs nacheinander auf gleiche Ressourcen, z. B. Register, zugreifen. Dabei können folgende Fälle unterschieden werden:

- *Read-After-Write (RAW)*: Eine MO benötigt Daten, die von einer früheren MO geschrieben werden. Die zweite MO kann erst ausgeführt werden, nachdem die erste MO ihr Ergebnis berechnet und beispielsweise in einem Register abgelegt hat. Um die Wartezeit zu verkürzen, kann *forwarding* verwendet werden, um der zweiten MO das Ergebnis der ersten direkt im Datenpfad zur Verfügung zu stellen, ohne den Weg über die Register zu wählen.

- *Write-After-Read (WAR)*: Eine MO überschreibt Daten, die von einer früheren MO gelesen werden. Die zweite MO darf ihr Ergebnis erst dann in das Zielregister schreiben, wenn die erste MO das vorherige Ergebnis gelesen hat.

- *Write-After-Write (WAW)*: Zwei MOs beschreiben die gleiche Ressource. Die korrekte Reihenfolge der MOs muss erhalten bleiben, damit das korrekte Ergebnis nicht überschrieben wird.

Die letzten beiden Abhängigkeitstypen beschreiben so genannte *falsche Abhängigkeiten*, da sie in manchen Fällen durch den Kompiler aufgelöst werden können, beispielsweise durch eine Umbenennung von Zielregistern. Das *register renaming* wird in der hier vorgestellten Codegenerierung für die virtuellen Register auf Ebene der SLMs durchgeführt. Dazu wird ein virtuelles Zielregister einer Operation

```
 1 :L_LOOP

 3 SMVI     V0CONDSEL, #2
 4 MAX      VxR1, V0R0, V0R1
 5 MAX      VxR1, VxR1, V0R2
 6 SUBCS    VxR2, VxR1, V0R3
 7 ADD      V0R3, V0R4, V0R5
 8 MACPLZ   VxR4+VxR5, V0R6, V0R7
 9 SUB      VxR2, V0R3, VxR4
10 MVCR     V0R4, VxR2

12 ELOOPR   V1R10, LOOP, #0
```

Listing 4.1: Exemplarischer Assemblercode für eine SLM

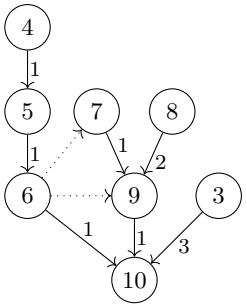


Abbildung 4.2: DDG zu nebenstehendem Code

umbenannt zu einem in der SLM bisher nicht verwendeten virtuellen Register. In allen nachfolgenden Operationen, die dieses Register als Argument verwenden, wird die Umbenennung ebenfalls durchgeführt. Eine Registerumbenennung für physische Register ist in dieser lokalen Vorgehensweise nicht möglich, da der Kompiler dazu den gesamten Datenfluss über alle SLMs hinweg analysieren müsste. Ist eine solche Umbenennung nicht möglich, können WAR-Abhängigkeiten auch *schwache Abhängigkeiten* sein. Dies bedeutet, dass die schreibende MO mit der lesenden MO gleichzeitig platziert werden kann, da das Lesen in einer früheren Pipelinestufe als das Schreiben erfolgt. Damit kann folgende Definition für Abhängigkeiten gegeben werden:

Definition 4.1. Eine MO m_2 ist *direkt abhängig* von einer MO m_1, wenn sie das Ergebnis von m_1 für die eigene Ausführung benötigt. Die Menge aller MOs, die direkt von einer MO m abhängen, heißt *Nachfolger* oder auch *Kinder* von m und wird mit children(m) bezeichnet. Umgekehrt bezeichnet parents(m) die *Vorgänger* einer MO m, also die Menge aller MOs, von denen m direkt abhängt.

Bei der Abhängigkeitsanalyse wird ein *Datenabhängigkeitsgraph* (*Data Dependency Graph*, *DDG*) aufgebaut, in dem Knoten die einzelnen MOs darstellen. Eine gerichtete Kante von MO m_1 zu MO m_2 stellt eine direkte Abhängigkeit der MO m_2 von m_1 dar. Die Kante (m_1, m_2) trägt als zusätzliche Information die Latenz der Operation m_1. Der TUKUTURI Codegenerator unterstützt Operationen mit flexibler Latenz, für die im Eingabeprogramm die Latenz für die konkrete Instanz der Operation festgelegt wird.

Als Beispiel ist der Code für eine SLM in Listing 4.1 gegeben. Die SLM hat den explizit durch das Label `LOOP` angegebenen Einsprungpunkt und endet mit der Schleifenoperation `ELOOPR`, die sich ähnlich zu einem Branch verhält. Für die MOs bezeichnet das erste Argument das Zielregister, die folgenden Argumente bezeichnen die Operanden. Die Argumente können Immediate-Werte sein, die durch ein Doppelkreuz (#) gekennzeichnet werden. Register der Prozessorarchitektur

werden als `VrRn` bezeichnet, wobei r die Registerbank angibt und n die Adresse des Registers innerhalb der Registerbank. Der Programmierer kann außerdem Variablen benutzen, die durch *virtuelle Register* `VxRn` bezeichnet werden, wobei n eine beliebige Zahl sein kann. Diese virtuellen Register werden in der Registerallokation durch den Kompiler Hardwareregistern zugeordnet. Eine Besonderheit stellt die `MAC`-Operation dar: das *multiply and accumulate* schreibt das Ergebnis in zwei Zielregister. Außerdem werden die Zielregister als Akkumulator benutzt und daher zusätzlich gelesen. In dieser Variante (`MACPLZ`) wird der Akkumulator allerdings mit 0 initialisiert und daher nicht gelesen. Das zweite Argument der `ELOOPR`-Operation ist das Label, das den Beginn der Schleife kennzeichnet, die durch das `ELOOPR` beendet wird. In der Binärkodierung der Operation wird der Kompiler hier die Adresse der ersten Schleifenoperation im schedulierten Programm eintragen.

Für den Beispielcode ist in Abbildung 4.2 der Datenabhängigkeitsgraph gezeigt, wobei die Knoten die MOs darstellen und die Zahl in den Knoten die Zeilennummer der MOs in Listing 4.1 angibt. Die Zahlen an den Kanten des Graphen stellen die Latenzen der Operationen dar, also die Anzahl von Zyklen, die die Operation zur Generierung eines Ergebnisses benötigt. Diese wird für eine Operation m mit latency(m) bezeichnet. MO 3 beschreibt das Spezialregister `V0CONDSEL`, das für die Speicherung der Bedingung *zero* für die konditionale Ausführung (siehe Abschnitt 3.1.1) in MO 10 benutzt wird. Daher gibt es eine Abhängigkeit zwischen diesen MOs, die mit der Latenz von MO 3 gekennzeichnet ist. Zwischen MO 4 und MO 5 besteht eine RAW-Abhängigkeit über das virtuelle Register `VxR1`, das von MO 4 geschrieben und in MO 5 gelesen wird. Ebenso besteht eine RAW-Abhängigkeit zwischen MO 5 und MO 6. Die MOs 6 und 9 verwenden das gleiche Zielregister, so dass hier eine WAW-Abhängigkeit besteht. Dabei handelt es sich um eine falsche Abhängigkeit (gestrichelt im Graphen), die vom Kompiler durch die Umbenennung von `VxR2` in den MOs 9 und 10 in ein nicht verwendetes Register aufgelöst werden kann. Der gleiche Fall tritt auch schon bei den MOs 4 und 5 auf, doch hier besteht die WAW-Abhängigkeit zusätzlich zur RAW-Abhängigkeit. Das Zielregister der MO 5 wird zwar umbenannt (um Registerabhängigkeiten zu verringern), doch die Abhängigkeit von MO 5 von MO 4 kann dadurch nicht aufgelöst werden. Die WAR-Abhängigkeit zwischen MO 6 und MO 7 über Register `V0R3` ist eine schwache Abhängigkeit: MO 7 kann parallel zu MO 6 platziert werden, da MO 6 das Register bereits gelesen hat, wenn MO 7 das Ergebnis schreibt. Dies ist keine falsche Abhängigkeit, da ein physikalisches Register verwendet wird, für das keine Umbenennung erfolgen kann, da es möglicherweise in vorherigen oder nachfolgenden SLMs verwendet wird und die Umbenennung lokal auf Ebene der SLMs durchgeführt wird.

Während dieser Vorbereitung des Instruction Scheduling werden neben dem *register renaming* weitere Optimierungen vorgenommen, die den Parallelisierungsgrad des Codes erhöhen können. Der Prozessor/Kompiler unterstützt beispiels-

```
STORERCUPPL  0x4000, V0R0, #3
LOADRCU      V0R1, 0x4001

STORERCUPPL  0x4002, V0R2, #3
LOADRCU      V0R3, 0x4002
```

Listing 4.2: Exemplarischer Code mit STORERCUPPL-Pseudooperation

```
1 STORE 0x4000, V0R0
4 STORE 0x4002, V0R2
  NOP
2 LOAD  V0R1, 0x4001
5 LOAD  V0R3, 0x4002
```

Listing 4.3: Exemplarischer Code nach Scheduling

weise zwei Arten von *immediate* Werten: *short immediates*, die aufgrund ihrer geringen Bitlänge direkt in der MO kodiert werden können, und *long immediates*, die 32 Bit umfassen und in dem auf die MO folgenden Issue-Slot kodiert werden, so dass dort keine weitere MO parallel platziert werden kann. Der Kompiler kann die immediate-Werte analysieren und long immediates, die klein genug sind, in short immediates transformieren, um so den parallelen Issue-Slot für eine weitere MO freizugeben.

List Scheduling

Der grundlegende Ablauf des List Scheduling [26] kann wie folgt beschrieben werden: Nach der Abhängigkeitsanalyse werden alle MOs, deren Abhängigkeiten erfüllt sind, in eine Menge (*data ready set*) aufgenommen. Der Scheduler versucht nun, ein Instruktionswort (eine MI) nach dem anderen zu füllen, indem er aus dem data ready set MOs auswählt. Ist die MI gefüllt, oder sind keine weiteren Kandidaten vorhanden, wird die MI an den aktuellen basic block angehängt und eine neue begonnen. Die in dieser MI platzierten MOs werden vom data ready set in die Liste der laufenden MOs verschoben. Diese Liste dient der Verwaltung der Latenzen der platzierten Operationen: Jede weitere MI, die von Algorithmus gefüllt und an den basic block angehängt wird, stellt einen Taktzyklus dar, der die Wartezyklen der platzierten Operationen verringert. Sind die Wartezyklen für eine platzierte Operation abgelaufen, gilt diese Operationen für all ihre Nachfolger im DDG als erledigt und das data ready set kann aktualisiert werden, indem alle Operationen, deren Abhängigkeiten nun erfüllt sind, in das data ready set aufgenommen werden. Zur Behandlung von schwachen Abhängigkeiten ist die Aktualisierung des data ready sets nach der Platzierung jeder einzelnen MO nötig, damit eine MO m_2, die schwach von einer MO m_1 abhängt, als Kandidat zur Verfügung steht, sobald m_1 in der aktuellen MI platziert wurde (sofern keine weiteren Abhängigkeiten bestehen), und parallel in der gleichen MI platziert werden kann.

Für die meisten Operationen bestimmt die Latenz die Anzahl der Taktzyklen nach dem Start der Operation, bis das Ergebnis vorliegt. Da die funktionalen Einheiten des Prozessors die Ergebnisse in internen Registern speichern, können die Ergebnisse frühestens nach dem Ablauf dieser Taktzyklen gelesen werden,

die Ergebnisse sind aber auch in späteren Zyklen noch vorhanden, sofern nicht eine erneute Benutzung der funktionalen Einheit das frühere Ergebnis überschreibt. Die Latenz kann also als eine *Mindestwartezeit* aufgefasst werden. Bei der Benutzung spezieller Coprozessoren (siehe Abschnitt 3.3.2), die eine interne Pipeline besitzen, kann es nötig sein, die zur Kommunikation verwendeten `STORE` und `LOAD` Operationen zu exakt vorgegebenen Zyklen (also mit genau bestimmtem Abstand) zu platzieren. Dazu wurde das List Scheduling dahin gehend erweitert, dass Operationen für diese Kommunikation nicht mit einer Mindestlatenz, sondern mit einer exakten Latenz platziert werden können: Im Assemblercode werden für diese Art der Kommunikation Pseudooperationen verwendet, die später in `STORE` und `LOAD` Operationen umgewandelt werden, für die aber die Abhängigkeitsanalyse so umgestaltet ist, dass nur Abhängigkeiten von einem `STORERCUPPL` zum nächsten `LOADRCU` berücksichtigt werden. Dadurch kann der Kompiler mehrere aufeinander folgende Blöcke von `STORE`-`LOAD`-Operationen überlappen. In Listing 4.2 ist ein Beispielcode gegeben, der zwei Benutzungen eines Coprozessors über diese Pseudooperationen zeigt. Nach dem Schreiben eines Datenwortes in MO 1 kann das Ergebnis der Berechnung nach drei Wartezyklen von MO 2 abgerufen werden. Der Coprozessor besitzt eine interne Pipeline, so dass die nächste Berechnung, die in MO 4 gestartet wird, direkt nach dem Schreiben des ersten Operanden begonnen werden kann. Dies zeigt der kompaktierte Code in Listing 4.3, in dem die beiden Benutzungen des Coprozessors überlappt sind. Das `NOP` in MI 3 kann durch andere Operationen aus der SLM gefüllt werden, sofern Abhängigkeiten dies nicht verhindern. Wurde eine solche `STORERCUPPL` Pseudooperation während des Scheduling platziert und ist die festgelegte Latenz abgelaufen, muss die abhängige `LOADRCU` Operation in der nächsten MI platziert werden. Ist dies nicht möglich, zum Beispiel aufgrund von Konflikten zwischen anderen MOs, wird das Scheduling als ungültig abgebrochen. Da die statische Heuristik nur eine Scheduling-Variante untersucht, schlägt damit die Codegenerierung für diese SLM fehl. Der später beschriebene EA-basierte Algorithmus zum Instruction Scheduling (Abschnitt 4.3.1) kann in diesem Fall ein anderes Instruction Scheduling evaluieren, um eine kompakte Lösung zu finden.

Während des Instruction Scheduling wird die Entscheidung, welche MOs aus dem data ready set zur Platzierung in der aktuellen MI gewählt werden, über *Gewichte* getroffen, die den MOs von einer statischen Heuristik nach der Datenabhängigkeitsanalyse zugewiesen werden. Typische Heuristiken werden dabei aus dem Datenabhängigkeitsgraphen abgeleitet. In dieser Implementierung berechnet sich das Gewicht einer MO m aus dem DDG als Summe der eigenen Latenz plus dem größten Gewicht der Kind-Operationen, das rekursiv auf die gleiche Weise berechnet wird:

$$\text{weight}(m) = \text{latency}(m) + \max_{c \in \text{children}(m)} \text{weight}(c).$$

```
 1 ADD       VxR0, V0R2, V0R0
 2 SRI       VxR1, VxR0, #1
 3 MAX       VxR2, VxR1, V0R2
 4 PERMREG0  V0R6, VxR0, VxR2

 6 MAX       VxR0, V1R0, V1R1
 7 ADD       V1R5, VxR0, V0R0

 9 ADD       VxR2, VxR0, V0R1
10 PERMREG0  V0R7, VxR0, VxR2

12 PERMREG1  V1R6, VxR0, V0R2
```

Listing 4.4: Exemplarischer Code für die Codegenerierung

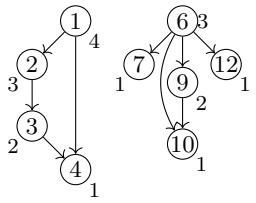


Abbildung 4.3: DDG zu nebenstehendem Code

Bei der Platzierung von MOs in den MIs müssen zusätzlich noch Konflikte zwischen den MOs berücksichtigt werden. Konflikte entstehen, wenn verschiedene MOs gleichzeitig auf gleiche Ressourcen (z. B. Register, Speicher) zugreifen, diese Ressourcen aber nicht in ausreichender Zahl vorhanden sind oder die geforderte Zahl von Zugriffen nicht parallel erfolgen kann (z. B. die Anzahl von Lese-/Schreibzugriffen auf eine Registerbank). Der hier implementierte List Scheduler prüft für jeden MO Kandidaten, ob Konflikte mit bereits in der MI platzierten MOs bestehen. Im Konfliktfall wird ein anderer Kandidat gesucht. Es findet kein Backtracking statt, bei dem der Scheduler die bereits platzierten MOs wieder entfernt und neue Kandidaten sucht.

Eventuell in der SLM vorhandene Branches oder Jumps werden speziell behandelt, da diese immer nahe dem Ende des basic blocks platziert werden müssen. Die Anzahl der Delay-Slots für diese Operationen ist abhängig von der Pipeline-Konfiguration des Prozessors. Diese Delay-Slots können mit anderen Operationen aus der SLM aufgefüllt werden. Dazu findet zunächst ein Scheduling aller Operationen der SLM bis auf die Branches/Jump statt. Anschließend werden die letzten n MIs und die darin enthaltenen MOs wieder freigegeben, wobei n die Anzahl der Delay-Slots ist. Der Branch/Jump wird nun in der nächsten MI platziert und das Scheduling mit den freigegebenen MOs fortgesetzt. Sollten die Delay-Slots nicht aufgefüllt werden können, zum Beispiel wegen Datenabhängigkeiten oder Konflikten zwischen den MOs, wird eine MI mit den freigegebenen MOs gefüllt und der Branch/Jump in die nächste MI platziert. Dies wiederholt sich so lange, bis eine Platzierung aller restlichen MOs inklusive dem Branch/Jump gefunden wurde.

Ein exemplarischer Assemblercode, an dem die Codegenerierung in diesem und den folgenden Abschnitten gezeigt wird, ist in Listing 4.4 gegeben. Der dazugehörige DDG ist als Referenz in Abbildung 4.3 gezeigt. Mit den oben beschriebenen Techniken ermittelt das List Scheduling für den Beispielcode das kompaktierte Mikroprogramm aus Listing 4.5 mit 6 Mikroinstruktionen.

```
 1 ADD       V1R2, V0R2, V0R0   ;   6 MAX  V0R3, V1R0, V1R1
 2 SRI       V1R3, V1R2, #1     ;   9 ADD  V0R4, V0R3, V0R1
 3 MAX       V1R3, V1R3, V0R2   ;   7 ADD  V1R5, V0R3, V0R0
 4 PERMREG0  V0R6, V1R2, V1R3   ;   0 NOP
10 PERMREG0  V0R7, V0R3, V0R4   ;   0 NOP
12 PERMREG1  V1R6, V0R3, V0R2   ;   0 NOP
```

Listing 4.5: Beispielcode nach Instruction Scheduling mit statischer Heuristik

Da die verwendete Prozessorarchitektur jede funktionale Einheit nur einmal enthält, können die PERMREG-Operationen nicht parallel ausgeführt werden. Die Allokation der virtuellen Register ist hier ebenfalls durchgeführt worden, das Verfahren dazu wird in Abschnitt 4.2.3 beschrieben.

Bei Speicheroperationen kommt es zu Konflikten, wenn die Anzahl paralleler Lese-/Schreibzugriffe zu hoch ist. Daher wird der Kompiler im Normalfall lesende oder schreibende Operation nicht parallel platzieren. Für den Fall, dass der Prozessor parallele Speicherzugriffe auf unterschiedliche Adressbereiche erlaubt (siehe Abschnitt 3.1.3), ist daher ein Mechanismus eingeführt worden, der die parallele Nutzung des Speichers bei der indirekten Adressierung ermöglicht. Bei dieser Adressierung wird eine Speicheradresse in einem FIR-Spezialregister gespeichert. Dieses kann in anderen Operationen zum Zugriff auf den Speicher verwendet werden. Ist der Prozessor mit n dieser FIR-Register konfiguriert, dann erlaubt der Kompiler auch die Verwendung von mehr als n solcher Register. Dabei werden aber die Register mit Nummern größer als n auf die vorhandenen FIR-Register abgebildet:

$$FIR_k = FIR_{k \bmod n}$$

und die Register so auf Gruppen von jeweils n Elementen verteilt. Abhängigkeiten werden vom Kompiler nun nur noch zwischen Registern innerhalb einer Gruppe gebildet. Damit können in einer Konfiguration mit 4 FIR-Registern die Register FIR_0 bis FIR_3 wie gewohnt nicht parallel verwendet werden. Das Register FIR_4 entspricht FIR_0 und kann parallel zu den Registern FIR_1 bis FIR_3 verwendet werden. Datenabhängigkeiten bestehen für dieses Register dann nur mit den Registern FIR_5 bis FIR_7.

4.2.2 Operation Merging mittels statischer Heuristik

Die TUKUTURI-Architektur unterstützt einen besonderen Modus zur parallelen Ausführung von Operationen des gleichen Typs innerhalb eines Issue-Slots, wie in Abschnitt 3.1.3 erläutert. Dabei werden zwei MOs mit gleichem Typ, also gleichem Opcode und gleichen Modi, die auf unterschiedlichen Daten (Registern, Immediates) arbeiten, in einer X2-Operation zusammengefasst und kodiert. Da die Kodierung der X2-Operation nicht länger ist als die Kodierung einer einzelnen

MO, müssen bestimmte Bedingungen an die verwendeten Register gestellt werden, wie unten beschrieben.

Der Kompiler unterstützt automatisches *Operation Merging*, mit dem passende MOs fusioniert werden können, wie im nächsten Abschnitt erläutert wird. Dieses *Operation Merging* kann als ein Optimierungsschritt vor dem Instruction Scheduling aktiviert werden: Der Kompiler durchsucht die SLM nach MOs, die fusioniert werden können, und ersetzt sie durch entsprechende X2-Operationen. Anschließend wird die modifizierte SLM dem Instruction Scheduling und der Registerallokation übergeben.

Da es zu jeder MO in einer SLM mehrere Kandidaten zur Kombination geben kann, muss der Algorithmus eine Auswahl treffen. Dazu ist in diesem Kompiler für die TUKUTURI-Architektur eine statische Heuristik implementiert und erweitert worden.

Operation Merging

Ob zwei Operationen fusioniert werden können, hängt in erster Linie von der Konfiguration der Prozessorarchitektur ab. Die funktionale Einheit zur Ausführung einer Operation, beispielsweise einer Addition, muss doppelt im Prozessor vorhanden sein, damit eine X2-Operation zwei Additionen gleichzeitig ausführen kann. Weitere Einschränkungen ergeben sich in den Registerbänken durch die konfigurierte Anzahl von Schreib- bzw. Leseports. Eine X2-Operation schreibt immer zwei Register und liest aus bis zu vier Registern und benötigt entsprechend viele Ports.

Ein weiterer Aspekt bei der Frage der Kombinierbarkeit von Operationen ist die Kodierung der X2-Operation. Da diese vom Prozessor in einem Issue-Slot ausgeführt werden soll, muss sie in einem 32-Bit-Wort kodiert werden. Daher gelten die Beschränkungen, dass kombinierte Operationen den gleichen Opcode, gleiche Modi (wie etwa Subwordlänge, konditionale Ausführung, Vorzeichen, usw.) und gleiche Immediate-Argumente verwenden müssen. Da eine X2-Operation immer zwei Zielregister verwendet, ist für dieses keine weitere Markierung in der Kodierung nötig. Das Operationswort bietet nicht genügend Bits, um vier verschiedene Quellregister aufzunehmen. Daher müssen für jeden Operand jeweils aufeinander folgende Register gewählt werden und das erste Register eines Paares eine gerade Adresse haben. Dann reicht jeweils ein Bit aus, um einen X2-Parameter zu kennzeichnen und der Decoder kann die Adresse des zweiten Registers eines Paares durch das Setzen des niederwertigsten Bits bestimmen.

Natürlich hängt die Möglichkeit zur Kombinierung außerdem vom Programm selbst ab. So können MOs, zwischen denen eine Datenabhängigkeit besteht, nicht fusioniert werden, da sie nicht gleichzeitig ausgeführt werden können.

Für Operationen, die auf den Speicher lesend oder schreibend zugreifen, können X2-Operationen ebenfalls verwendet werden, wenn die Speicheranbindung mit

```
  6 MAX       V0R3,       V1R0,       V1R1 ; 0 NOP
1+7 ADD_X2    V1R4+V1R5,  V0R2+V0R3,  V0R0 ; 0 NOP
  9 ADD       V0R4,       V0R3,       V0R1 ; 2 SRI       V1R3, V1R4, #1
 12 PERMREG1  V1R6,       V0R3,       V0R2 ; 3 MAX       V0R5, V1R3, V0R2
 10 PERMREG0  V0R7,       V0R3,       V0R4 ; 4 PERMREG0  V0R6, V1R4, V0R5
```

Listing 4.6: Beispielcode nach Operation Merging und Instruction Scheduling mit statischer Heuristik

doppelter Wortbreite erfolgt, so dass zwei Worte aus dem Speicher gelesen werden können (siehe Abschnitt 3.1.3). Dazu müssen die Worte allerdings direkt aufeinander folgen und die Adresse des ersten Worts muss gerade sein. Der Programmierer muss den Kompiler bei Speicheroperationen unterstützen, da dieser die Bedingungen an die (dynamisch berechneten) Adressen nicht überprüfen kann. Die am Ende von Abschnitt 4.2.1 beschriebenen virtuellen FIR-Register für die indirekte Adressierung können vom Programmierer eingesetzt werden, um dem Kompiler eine automatische Fusion solcher Zugriffe zu ermöglichen.

Statische Heuristik für das Operation Merging

Das automatische Operation Merging verarbeitet die MOs einer SLM sequenziell und sucht für jede MO nach möglichen Partnern zur Kombinierung. Nach der statischen Heuristik wird dabei für jede MO der Kandidat gewählt, der in der Folge der MOs als erster nach der aktuellen MO auftritt. Dabei werden MOs, die bereits mit vorher verarbeiteten MOs fusioniert wurden, übersprungen. Diese Methode zielt auf bestimmte Muster im Anwendungscode ab, bei denen gleiche oder ähnliche Codeblöcke wiederholt auftreten, was beispielsweise durch *loop unrolling* entstehen kann. Allerdings können dadurch auch Operationen fusioniert werden, die nicht genau diesen Mustern entsprechen, wodurch ungünstige Paarungen entstehen können, die besseres *Operation Merging* verhindern. Daher wurde in Abschnitt 4.3.2 eine flexiblere Methode zum *Operation Merging* implementiert.

Wird eine MO m_1 mit einer (späteren) MO m_2 kombiniert, entsteht eine neue MO m_{1+2}. Bevor diese an der Stelle von m_1 in die SLM eingefügt werden kann, müssen zunächst alle Vorgänger von m_2 aus dem DDG (siehe Definition 4.1) in die SLM aufgenommen werden, sofern sie nicht bereits enthalten sind, damit die Datenabhängigkeiten von m_2 erfüllt sind, wenn diese in der fusionierten Operation an früherer Stelle platziert wird. Dieser Prozess läuft rekursiv, es werden also die Vorgänger aller MOs aufgenommen, bevor die MO selbst in die neue kombinierte SLM aufgenommen wird.

Für den Beispielcode aus Listing 4.4 wird mit dieser statischen Heuristik und anschließendem Instruction Scheduling der Code aus Listing 4.6 mit 5 Mikroinstruktionen generiert. Dazu wurde die Anzahl aller funktionalen Einheiten in der Prozessorarchitektur verdoppelt, so dass beispielsweise zwei Additionen

parallel ausgeführt werden können. Dies kann durch parallele Verwendung von ADD-Operationen in jeweils einem Issue-Slot, oder durch Verwendung einer ADD_X2-Operation in einem Issue-Slot geschehen. Auch in diesem Beispiel sind die virtuellen Register mit dem Verfahren aus Abschnitt 4.2.3 allokiert worden.

Nicht immer sind alle möglichen Kombinationen auch günstig, und es kann besser für das nachfolgende Scheduling und die Registerallokation sein, manche MOs nicht zu kombinieren. Die statische Heuristik, wie bisher beschrieben, ist dafür nicht ausgelegt. Es wird beispielsweise kein Backtracking benutzt, um verschiedene Kombinationen von MOs zu evaluieren. Abschnitt 4.3.2 beschreibt ein flexibleres Verfahren basierend auf einem evolutionären Algorithmus.

4.2.3 Registerallokation mit statischer Heuristik

Nach dem Instruction Scheduling einer SLM werden die darin vorkommenden virtuellen Register durch den Kompiler den Hardwareregistern des TUKUTURI Prozessors zugeordnet.

Definition 4.2. Die *Lebensdauer (live range)* eines Registers beginnt mit der MO, die das Register beschreibt. Sie endet mit der letzten MO, die aus dem Register liest. Ein Register kann innerhalb einer SLM mehrere nicht überlappende Lebensdauern besitzen.

Zunächst analysiert der Kompiler das gesamte Eingabeprogramm, also alle SLMs, auf die darin verwendeten Hardwareregister, und sperrt diese für die Allokation der virtuellen Register. Mit jeder weiteren SLM können also weitere Hardwareregister ausgeschlossen werden. Da der Kompiler ein lokales Instruction Scheduling mit Registerallokation durchführt und den Kontrollfluss des Programms nicht vollständig untersucht, kann er nicht feststellen, wann verwendete Hardwareregister das Ende ihrer Lebensdauer erreichen und diese automatisch für eine Allokation freigeben. Deshalb ist im Kompiler eine Pseudooperation FREEREG implementiert, mittels der Register vom Programmierer für die SLM freigegeben werden können, sobald sie nicht mehr benötigt werden. Virtuelle Register werden vom Kompiler automatisch freigegeben, wenn sie nicht mehr benötigt werden, wie unten beschrieben.

Anschließend werden Abhängigkeiten zwischen Registern untersucht, die beispielsweise durch die Verwendung der Register in X2-Operationen entstehen. Register eines Paares in X2-Operationen müssen konsekutiv sein und das erste Register muss eine gerade Adresse erhalten. Die gleichen Bedingungen gelten auch für Operationen, die zwei Ausgaberegister beschreiben, wie zum Beispiel die MAC Operation.

Die Allokation von virtuellen Registern erfolgt in zwei Schritten: Zunächst wählt der Algorithmus für jedes Register bzw. Registerpaar die Registerbank aus, in der die Allokation stattfinden soll, anschließend werden darin ein oder

zwei Register ausgewählt. Dieser zweite Schritt ist, sofern die oben genannten Beschränkungen eingehalten werden, relativ frei in der Wahl der Register. Die Auswahl, die im ersten und zweiten Schritt getätigt wird, wird durch eine Heuristik gesteuert, die für eine ausgeglichene Belegung der verfügbaren Registerbänke sorgt. Dazu wird zunächst bestimmt, in welcher Registerbank bisher die wenigsten Register allokiert sind. Zur Allokation eines einzelnen Registers durchsucht der Algorithmus dann die gewählte Registerbank nach einem Registerpaar, von dem bereits ein Register allokiert ist und wählt das freie Register für die Allokation. Sollte kein solches Registerpaar existieren, wählt der Algorithmus das erste freie Register aus. Dadurch werden Paare von Registern für spätere Allokation frei gehalten. Soll ein Registerpaar allokiert werden, wählt der Algorithmus das erste freie Paar in der Registerbank. Sollte die Allokation auf diese Weise nicht möglich sein, weil keine Register oder Registerpaare verfügbar sind, wird die Registerbank mit der nächstgeringeren Anzahl allokierter Register ausgewählt.

Für den Fall, dass keine Register zur Allokation verfügbar sind, generiert der Kompiler keinen Code für ein Register-Spilling, bei dem Werte aus allokierten Registern in den Speicher kopiert werden, um diese verfügbar zu machen. Stattdessen bricht der Kompiler die Kompilierung mit einer Meldung ab.

Eine einmal erfolgte Allokation eines virtuellen Registers blockiert das gewählte Hardwareregister nicht unbedingt bis zum Ende des basic blocks. Stattdessen analysiert der Kompiler vor der Allokation die Lebensdauer (live ranges) der virtuellen Register innerhalb des basic blocks. Das Hardwareregister, das einem virtuellen Register zugeordnet wurde, wird für die Allokation wieder freigegeben, wenn das virtuelle Register in einer späteren MI das Ende seiner Lebensdauer erreicht. So kann dasselbe Hardwareregister sogar noch in der selben MI als Zielregister einer Operation benutzt werden, in der es letztmalig als Operand verwendet wird, wodurch die Lebensdauer endet.

4.3 Dynamische Heuristiken mit evolutionären Algorithmen

Die Codegenerierung für VLIW-Architekturen umfasst mit Instruction Scheduling, Operation Merging und Registerallokation komplexe Optimierungsprobleme, die häufig mit heuristischen Methoden behandelt werden, wie in Abschnitt 4.2 gezeigt.

Gute Heuristiken, die qualitativ hochwertige Ergebnisse liefern, sind im Allgemeinen schwer zu finden [81]. Statische Heuristiken, die durch die Analyse eines Programms Informationen zur Steuerung der Codeerzeugung ableiten, werden häufig an die verwendete Prozessorarchitektur angepasst, um eine hohe Qualität zu erreichen. Da die TUKUTURI-Architektur allerdings flexibel konfigurierbar ist, erhöht dies die Komplexität statischer Heuristiken zusätzlich und dynamische

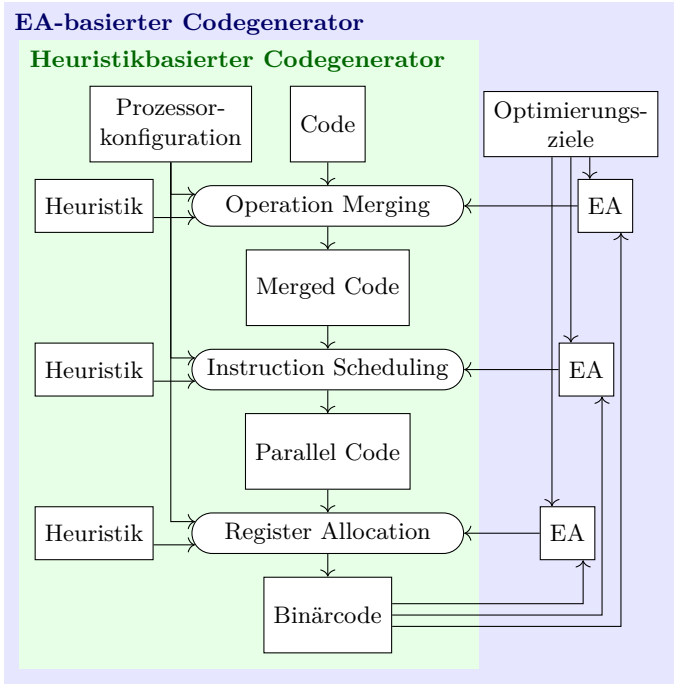


Abbildung 4.4: Codegenerierung mittels EA-basierter dynamischer Heuristiken

Heuristiken, die sich an die Zielarchitektur anpassen können, sollten in diesem Fall bessere Ergebnisse erzielen können. Die Adaptivität dynamischer Heuristiken erlaubt außerdem eine Anpassung an die zu kompilierende Applikation, wodurch manuelle Vorgaben an die Codegenerierung durch den Anwendungsprogrammierer (z. B. durch Pragmas im Anwendungscode) verringert oder gänzlich unnötig werden. Daher wird in den folgenden Abschnitten eine Erweiterung der Codegenerierung um dynamische Heuristiken, die auf evolutionären Algorithmen beruhen, vorgestellt.

Evolutionäre Algorithmen (EA) sind erfolgreich für eine Vielzahl von Optimierungsproblemen eingesetzt worden. Anstatt direkt nach einer Lösung des Problems zu suchen, verwenden EA eine *Population von Individuen*, die mögliche Lösungen für das Problem in jeweils einem so genannten Chromosom kodieren. Eine *Fitnessfunktion* bewertet die Individuen anhand der Lösungen, die sie kodieren. Verschiedene Operatoren, wie *Crossover* und *Mutation*, kombinieren anhand der Fitness ausgewählte Individuen zu neuen Lösungen in einer neuen Generation. Diese simulierte Evolution soll die Lösungen von Generation zu Generation verbessern und so das Optimierungsproblem lösen. Die Verwendung randomisierter Operatoren (typischerweise die Mutation) ermöglicht dabei das Überwinden von lokalen Optima.

Der erweiterte Codegenerator ist schematisch in Abbildung 4.4 gezeigt. Der

grün hinterlegte Teil zeigt die Codegenerierung mittels statischer Heuristiken, wie in Abschnitt 4.2 vorgestellt. Der blau hinterlegte Teil stellt die Erweiterung mittels EA-basierter dynamischer Heuristiken dar. Die evolutionären Algorithmen ersetzen die statischen Heuristiken für die drei Kompilerstufen Operation Merging, Instruction Scheduling und Registerallokation. Sie verwenden Informationen aus dem generierten Binärprogramm (insbesondere die Größe) zur Bewertung der aktuellen Generation, um daraus die nächste Generation möglicher Lösungen zu ermitteln. Dadurch fließen neben den Parameten der Prozessorkonfiguration auch Eigenschaften des Anwendungscodes selbst in die Optimierung mit ein.

Die Optimierungsprobleme sind auch nicht völlig isoliert voneinander zu betrachten, sondern beeinflussen sich gegenseitig. So kann ein ungünstiges Instruction Scheduling die Lebensdauer vieler Register so weit erhöhen und überlappen, dass eine Registerallokation sehr schwierig oder gar unmöglich wird. Auch das Operation Merging kann durch die erzeugten zusätzlichen Abhängigkeiten zwischen Registern die weitere Optimierung im Instruction Scheduling und der Registerallokation erschweren oder verhindern. Umgekehrt unterstützt ein gutes Instruction Scheduling die Registerallokation von Code nach dem Operation Merging, weil es Einfluss auf die Lebensdauer der Register nehmen kann. Daher sind die im Folgenden beschriebenen evolutionären Algorithmen miteinander verschachtelt und können sich gegenseitig beeinflussen, wie ebenfalls in Abbildung 4.4 zu erkennen. In den folgenden Abschnitten werden die implementierten evolutionären Algorithmen für das Operation Merging, das Instruction Scheduling und die Registerallokation beschrieben. Dabei wird zunächst das Instruction Scheduling beschrieben, um den generellen Ablauf vorzustellen. Das optionale Operation Merging, das vor dem Scheduling stattfindet, wird im Anschluss erklärt.

4.3.1 Instruction Scheduling

Für das Instruction Scheduling wird das List Scheduling eingesetzt, wie in Abschnitt 4.2.1 beschrieben. Dieses verwendet Gewichte zur Auswahl der zu platzierenden MOs während des Schedulings. Diese Gewichte werden statisch, einmalig zu Beginn des Scheduling aus dem DDG für die SLM abgeleitet und legen damit eine einzelne Lösung für das Schedulingproblem fest. Der im folgenden vorgestellte evolutionäre Algorithmus implementiert eine dynamische Heuristik, mit der die Gewichte für die einzelnen MOs flexibel vom Algorithmus vorgegeben werden können, so dass verschiedene mögliche Lösungen evaluiert und durch Rekombination verbessert werden können.

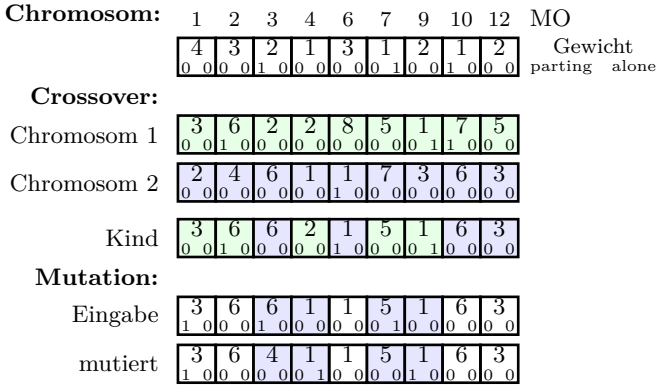


Abbildung 4.5: Schematische Darstellung eines Chromosoms für das Instruction Scheduling.

Evolutionärer Algorithmus für das Instruction Scheduling

Der evolutionäre Algorithmus für das Instruction Scheduling erweitert das in Abschnitt 4.2.1 beschriebene List Scheduling Verfahren. List Scheduling verwendet zur Auswahl der MOs bei der Platzierung in den MIs Gewichte, die beispielsweise aus dem Datenabhängigkeitsgraph (DDG) ermittelt werden. Im Gegensatz zu dieser statischen Gewichtung der MOs erweitert der EA das Verfahren, indem die Gewichte der MOs nicht mehr aus dem DDG abgeleitet werden, sondern direkt vom Chromosom vorgegeben werden. Dadurch ist die Verteilung der Gewichte und die sich daraus ergebende Reihenfolge der MOs während des Instruction Scheduling flexibler. Der Algorithmus kann anstatt einer einzigen Lösung nun verschiedene Gewichtsverteilungen evaluieren und über mehrere Generationen hinweg durch Rekombination und Mutation optimieren.

Das Instruction Scheduling hat durch flexible Anordnung von MOs auch Einfluss auf die Möglichkeiten zur Allokation von virtuellen Registern. Insbesondere kann ein ungünstiges Scheduling die Registerallokation sogar unmöglich machen. Dies ist insbesondere dann der Fall, wenn die Lebensdauer von Registern dadurch verlängert wird, dass die erzeugende MO weit vor der verbrauchenden MO platziert wird und sich die Lebensdauern mehrerer Register überschneiden, wodurch sich die Anzahl der gleichzeitig benötigten Registern erhöht.

Das Chromosom für das EA-basierte Instruction Scheduling enthält ein Gen für jede MO der SLM, wie in Abbildung 4.5 beispielhaft gezeigt. Jedes Gen trägt ein *Gewicht*, mit dem die Reihenfolge der Operationen während des Scheduling kontrolliert wird. Weiterhin sind in jedem Gen zwei Flags *parting* und *alone* angegeben, die Einfluss auf die möglichen Kombinationen von MOs nehmen: Ist das *parting*-Bit für eine MO gesetzt, wird die SLM an dieser MO virtuell geteilt. Dann können MOs aus dem Teil nach der parting-MO nicht mit MOs aus dem

Teil vor der parting-MO platziert werden, wodurch eine starke Verlängerung der Lebensdauer von Registern und auch die Überlappung von Lebensdauern unterschiedlicher Register verringert wird. Dadurch kann die Anzahl gleichzeitig verwendeter Register verringert werden, was die Registerallokation vereinfacht. Wenn das *alone*-Bit für eine MO gesetzt ist, kann diese MO nur alleine in einer MI platziert werden. Wird also diese MO für das Scheduling in der aktuellen MI ausgewählt, kann sie dort nur platziert werden, wenn nicht bereits andere MOs in der MI platziert sind. Damit sind außerdem die anderen Issue-Slots der MI blockiert und werden mit `NOPs` aufgefüllt. Dies kann Konflikte in der Registerallokation auflösen, wodurch das Scheduling insgesamt verbessert werden kann. Da allerdings die Auswirkungen auf die Kompaktierung des Codes groß sind, wird dieses Bit mit geringer Wahrscheinlichkeit gesetzt.

Die Populationsgröße S für das evolutionäre Instruction Scheduling leitet sich aus der Größe der SLM ab und kann durch ein *Optimierungslevel* o, das als Kommandozeilenparameter angegeben werden kann, beeinflusst werden:

$$S = \max\{|\text{SLM}| \cdot 2^{o-2}/5, 10\}.$$

Der Wert $o = 0$ deaktiviert das Instruction Scheduling, in diesem Fall muss der Programmierer das Scheduling manuell vornehmen. Der Wert $o = 1$ verwendet das Instruction Scheduling mit statischer Heuristik. Für Werte $o \geq 2$ wird der EA mit der angegebenen Populationsgröße verwendet. Die Anzahl der MOs in einer SLM ermöglicht eine Abschätzung der Komplexität für die Kompaktierung dieser SLM, da sie die Anzahl der möglichen Kombinationen von MOs in MIs beeinflusst. Daher geht diese Anzahl in die Berechnung der Populationsgröße ein, um den Kompaktierungsaufwand auf die lohnenden SLMs zu konzentrieren. Eine Mindestgröße von 10 Individuen ist vorgegeben. Der Programmierer kann über Kompilerdirektiven (pragmas) das globale Optimierungslevel für einzelne SLMs überschreiben, um so einen feingranularen Einfluss auf die Verteilung der Rechenzeit zu nehmen. Außerdem bietet der Kompiler ein Kommandozeilenargument, um die Populationsgröße unabhängig von der Größe der SLMs vorzugeben.

Zur Initialisierung der ersten Generation von Individuen wird zunächst ein Chromosom erzeugt, das die gleichen Gewichte enthält, wie sie die statische Heuristik für das Instruction Scheduling aus dem DDG berechnet hat. Für dieses Chromosom werden die *parting*- und *alone*-Flags nicht gesetzt. Dieses Chromosom stellt in vielen Fällen eine brauchbare Grundlage für die folgende Evolution bereit, kann aber von EA auch ignoriert werden, wenn die Qualität des erzeugten Mikroprogramms hinter der der anderen Chromosomen steht. Außerdem garantiert dieses Individuum, dass die vom EA gefundene Lösung immer mindestens so gut ist, wie die Lösung der statischen Heuristik, wenn überhaupt eine Lösung gefunden wird. Für die restlichen Chromosomen der ersten Generation werden die Gewichte für die einzelnen MOs zufällig aus dem

Bereich $[1, |\text{SLM}|]$ gewählt. Die Flags *parting* und *alone* werden für jede MO unabhängig mit voreingestellten Wahrscheinlichkeiten 0,01 bzw 0,001 gesetzt, die über Kommandozeilenparameter verändert werden können.

Zur Bewertung der Individuen wird zunächst das Instruction Scheduling und auch die Registerallokation für die SLM mit den Parametern aus dem Chromosom durchgeführt. Die Registerallokation kann dabei je nach Vorgabe durch den Programmierer mit der statischen Heuristik (Abschnitt 4.2.3) oder mit dem später erläuterten evolutionären Algorithmus (Abschnitt 4.3.3) durchgeführt werden. Als Fitness der Individuen wird die Größe des kompaktierten Mikroprogramms verwendet. Sollte die Registerallokation für den Code fehlschlagen, wird ein Strafterm auf die Fitness addiert. Zur Auswahl der Individuen für die Rekombination ist kein absoluter Fitnesswert für die Individuen nötig. Stattdessen wird eine relative Ordnung der Individuen verwendet, die durch die folgende Sortierung der Population erzeugt wird.

Die Population wird in verschiedene Bereiche aufgeteilt: Der erste Bereich enthält die Individuen, für die das Instruction Scheduling und die Registerallokation erfolgreich waren. Innerhalb dieses Bereichs werden die Individuen nach der Größe des generierten Mikroprogramms sortiert. Der zweite Bereich enthält die Individuen, für die keine Registerallokation durchgeführt wurde (siehe die nachfolgende Diskussion zum Überspringen der Registerallokation), oder für die die Registerallokation nicht erfolgreich war. In diesem Bereich werden die Individuen nach der Größe des Mikroprogramms sortiert. Falls diese identisch ist, wird zur Sortierung die Fitness der Registerallokation verwendet: Wurde die genetische Registerallokation verwendet, wird deren Fitness benutzt, andernfalls die Fitness der Registerallokation mit statischer Heuristik.

In der nächsten Generation wird eine Population mit der gleichen Anzahl an Individuen verwendet. Um diese zu erzeugen, werden die besten drei Individuen nach der oben angegebenen Sortierung aus der vorigen Generation kopiert (*Elite*) und 1 % der neuen Individuen wird zufällig erzeugt. Der Rest der Population wird durch *Crossover* und *Mutation* erzeugt. Diese Zusammensetzung der neuen Generation ist empirisch als günstig ermittelt worden. Da diese Parameter aber von der zu kompilierenden Anwendung abhängen, können sie durch Kommandozeilenparameter vom Benutzer überschrieben werden.

Die zufällige Erzeugung neuer Individuen entspricht dabei dem Vorgehen während der Initialisierung der Individuen für die erste Generation: Die Gewichte werden zufällig zwischen 1 und der Anzahl der MOs in der SLM gewählt, die Flags *parting* und *alone* werden mit den jeweiligen Wahrscheinlichkeiten unabhängig für jedes Gen gesetzt. Gewichte, wie sie die statische Heuristik aus dem DDG berechnet, werden in diesem Schritt nicht mehr verwendet.

Beim Crossover werden zwei Individuen aus der aktuellen Generation ausgewählt und zu einem neuen Individuum für die nächste Generation kombiniert. Bei der Auswahl der Individuen sollten solche, die bezüglich der Größe des er-

```
 1 ADD       V0R3, V0R2, V0R0   ;   6 MAX   V1R2, V1R0, V1R1
12 PERMREG1  V1R6, V1R2, V0R2   ;   2 SRI   V1R3, V0R3, #1
 3 MAX       V0R4, V1R3, V0R2   ;   7 ADD   V1R5, V1R2, V0R0
 4 PERMREG0  V0R6, V0R3, V0R4   ;   9 ADD   V1R3, V1R2, V0R1
10 PERMREG0  V0R7, V1R2, V1R3   ;   0 NOP
```

Listing 4.7: Beispielcode aus Listing 4.4 nach Instruction Scheduling mit EA-basierter Heuristik (o8)

reichten Mikroprogramms besser abschneiden, mit höherer Wahrscheinlichkeit ausgewählt werden. Dazu dient das oben beschriebene Kriterium zur Sortierung der Individuen. Für die in diesem Algorithmus verwendete *tournament selection* werden aus der aktuellen Generation mehrere Individuen (die *tournament size* ist standardmäßig auf 3 gesetzt) zufällig ausgewählt. Das Individuum, das bezüglich der Sortierung am weitesten vorne steht, wird selektiert. Mit dieser Methode können prinzipiell alle Individuen gewählt werden, doch Individuen mit besseren Ergebnissen werden mit einer höhere Wahrscheinlichkeit gewählt als Individuen mit schlechteren Ergebnissen. Zur Kombination zweier Individuen wird dann jedes Gen (also die Kombination aus Gewicht, *parting* und *alone*) zufällig und unabhängig von den anderen Genen aus einem der beiden Elternindividuen übernommen. Dieses neu erzeugte Chromosom wird anschließend der Mutation unterzogen: Für jedes Gen wird unabhängig mit einer Wahrscheinlichkeit von 0,008 ein zufälliges Gewicht bestimmt. Die Flags *parting* und *alone* werden jeweils mit einer Wahrscheinlichkeit von 0,01 wie in der Initialisierung neu bestimmt (also mit den dort angegebenen Wahrscheinlichkeiten gesetzt).

Die Anzahl der Generationen, die der Instruction Scheduler verwendet, kann über einen Kommandozeilenparameter fest vorgegeben werden. Im Normalfall verwendet der EA allerdings ein *dynamisches Abbruchkriterium*: Der Algorithmus stoppt nach einer einstellbaren Anzahl von Runden, in denen die Fitness des besten Individuums nicht verbessert werden konnte. Sollte der Algorithmus in den ersten Runden keine Lösung finden, versucht der Algorithmus für eine ebenfalls einstellbare Anzahl von Generationen eine Lösung zu finden. Erst wenn diese erste Lösung gefunden wurde, tritt das oben beschriebene Abbruchkriterium in Kraft und der Generationenzähler wird zurückgesetzt. Der Algorithmus berechnet zudem aus dem DDG den kritischen Pfad, d. h., die mindestens nötige Anzahl von MIs für die aktuelle SLM. Über die Anzahl der MOs und die Anzahl der Issue-Slots im Prozessor kann zusätzlich eine weitere untere Schranke für die Größe des basic blocks geschätzt werden. Sollte während des Schedulings eine dieser unteren Schranken erreicht werden, bricht der EA vorzeitig ab, da keine weitere Verbesserung möglich ist. Die Evaluation dieses Abbruchkriteriums ist in Abschnitt 4.4.1 gegeben.

Mit dieser dynamischen Heuristik wird der Beispielcode aus Listing 4.4 zu dem

Code aus Listing 4.7 mit nur 5 Mikroinstruktionen kompaktiert, im Vergleich zum Scheduling mit statischer Heuristik, die 6 Instruktionen generierte (Listing 4.5). Dadurch, dass nach der Platzierung der MOs 1 und 6 die MO 12 vorgezogen wurde, konnte MO 9 später parallel zu einem `PERMREG` platziert werden und es wurde keine weitere Instruktion für das `PERMREG` benötigt.

Um die Ausführung des EA zu beschleunigen, kann die Registerallokation, die als Teil der Bewertung der Individuen aufgerufen wird (siehe Abbildung 4.4), übersprungen werden. Dazu wird die Größe des aktuell besten Scheduling herangezogen: Wenn das Mikroprogramm für ein Individuum um mindestens ein einstellbares Offset größer ist als diese Grenzgröße, wird die Registerallokation für das Individuum übersprungen. Das Offset zum Vergleich mit der Grenzgröße kann auch negative Werte annehmen. In dem Fall muss ein Chromosom zu einem Mikroprogramm führen, das um den entsprechenden Betrag kleiner ist als die Grenzgröße, damit eine Registerallokation durchgeführt wird. Der Einfluss dieses Parameters auf die Laufzeit und die Größe der erzielten Mikroprogramme ist in Abschnitt 4.4.1 evaluiert worden.

Da der Algorithmus an verschiedenen Stellen Zufallselemente enthält, besteht die Möglichkeit, dass innerhalb einer Generation oder über mehrere Generationen hinweg identische Chromosomen wiederholt erzeugt werden. Da außerdem die Chromosomen über die Gewichte nur die relative Reihenfolge der MOs festlegen, mit der diese während des List Scheduling ausgewählt werden, können Chromosomen trotz unterschiedlicher Gewichte zum gleichen Verhalten während des Schedulings und damit zu gleichen Mikroprogrammen führen. Eine Analyse hat gezeigt, dass beim Scheduling größerer SLMs einer Beispielanwendung (siehe Kapitel 6) etwa 0,8 % der generierten Chromosomen identisch waren und dass nur etwa 0,62 % unterschiedlicher Chromosomen zu gleichen Mikroprogrammen geführt haben. Diese geringen Anteile rechtfertigen den zusätzlichen Aufwand zur Erkennung und Filterung von Duplikaten nicht, weshalb dieser Mechanismus in den Standardeinstellungen des Kompilers deaktiviert ist.

Parallelisierung des genetischen Algorithmus

Genetische Algorithmen verarbeiten die Individuen einer Generation häufig unabhängig voneinander, weshalb eine Parallelisierung auf dieser Ebene ohne erhöhten Aufwand zur Synchronisierung umgesetzt werden kann. In dieser Implementierung des Kompilers für die TUKUTURI-Architektur ist eine Parallelisierung mit OpenMP[1] durchgeführt worden.

Die Population für eine SLM hat über alle Generationen eine feste Größe, es werden also keine dynamischen Datenstrukturen dafür verwendet. Die Individuen können ohne Synchronisierung unabhängig voneinander manipuliert werden.

[1] `https://www.openmp.org`

Die Initialisierung der ersten Population und auch die Erzeugung der nächsten Generation aus der aktuellen Generation geschieht parallel für jedes Individuum. Dann wird wiederum für jedes Individuum parallel das Instruction Scheduling und die auf der statischen Heuristik basierende Registerallokation durchgeführt. Sollte die EA-basierte Registerallokation aktiviert sein, wird bereits in diesem Schritt der dafür verwendete RDG (siehe Abschnitt 4.3.3) berechnet. Dann kann die Fitness für jedes Individuum bestimmt werden. Sollte die erweiterte Registerallokation mit genetischem Algorithmus (siehe Abschnitt 4.3.3) aktiviert sein, wird diese anschließend für jedes Individuum im Scheduling sequentiell ausgeführt, da innerhalb der genetischen Registerallokation eine Parallelisierung stattfindet und eine verschachtelte Parallelisierung schlechtere Performanz gezeigt hat.

4.3.2 Operation Merging

Wie in Abschnitt 4.2.2 beschrieben wurde, gibt es oftmals für eine MO mehrere mögliche Partner für das *Operation Merging*. Die Auswahl der Paarungen kann großen Einfluss auf die erreichbare Kompaktierung und die Registerallokation des kombinierten Codes haben. Da die Anzahl möglicher Kombinationen in größeren SLMs zu groß für eine vollständige Evaluation (erschöpfende Suche) ist, wird auch beim Operation Merging ein heuristisches Verfahren verwendet. Die statische Heuristik in Abschnitt 4.2.2 verfolgt dabei einen *greedy*-Ansatz und evaluiert nur eine einzige mögliche Lösung, weshalb hier ein dynamischer Ansatz zum Operation Merging auf Basis eines evolutionären Algorithmus vorgestellt wird.

Evolutionärer Algorithmus für automatisches Operation Merging

Der evolutionäre Algorithmus für das automatische Operation Merging analysiert zunächst die SLM und ermittelt für jede MO eine Liste aller Kandidaten, die mit dieser Operation kombiniert werden können. Dabei werden wiederum nur die MOs als mögliche Kandidaten untersucht, die in der Folge der MOs nach der aktuell betrachteten MO stehen und es kommen wieder die Kriterien aus Abschnitt 4.2.2 zur Anwendung, um festzustellen, ob MOs kombiniert werden können.

Das Chromosom für den EA zum automatischen Operation Merging enthält ein Gen für jede MO in der SLM. Das Gen speichert zu der MO die Liste aller Kandidaten, die mit dieser MO kombiniert werden können, wie in Abbildung 4.6 gezeigt. Das Chromosom aus der Abbildung zeigt die Merge-Kandidaten für den Beispielcode aus Listing 4.4. Die Reihenfolge, in der die Kandidaten in einer solchen Liste stehen, ist auch die Reihenfolge, in der sie während des Kombinierungsschritts ausgewählt werden. In jede Kandidatenliste wird ein zusätzlicher Stoppeintrag eingefügt, der in der Abbildung durch ein ‘X’ dargestellt

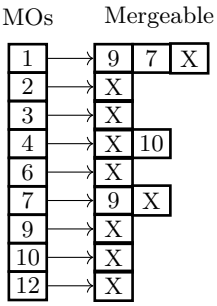


Abbildung 4.6: Exemplarisches Chromosom für das automatische Operation Merging

ist und mit dem festgelegt wird, dass die entsprechende MO nicht kombiniert werden soll. Während der Fusionierung werden die MOs der SLM der Reihe nach verarbeitet. Im Beispiel würde MO 1 mit MO 9 fusioniert. Die MOs 2 und 3 werden nicht fusioniert, da sie keine Kandidaten besitzen. MO 4 wird nicht fusioniert, da das Chromosom dies verbietet. Schließlich kann MO 7 nicht fusioniert werden, da MO 9 bereits verarbeitet ist und keine weiteren Kandidaten vorliegen.

Bei der Initialisierung eines Chromosoms werden die Listen der Kandidaten innerhalb jedes Gens permutiert, also die Einträge in eine zufällige Reihenfolge gebracht. Dadurch ergeben sich unterschiedliche Paarungen von MOs zur Fusionierung und die Chromosomen stellen unterschiedliche Lösungen für das Optimierungsproblem dar. In der ersten Generation werden spezielle Chromosomen erzeugt. Für das erste Chromosom werden die Kandidatenlisten nicht permutiert, sondern bleiben in der Reihenfolge, wie sie durch die Kandidatensuche generiert wurden. Dies bedeutet, dass jede MO mit dem ersten möglichen Kandidaten fusioniert werden soll, so wie es auch die statische Heuristik vorgeben würde. Die Stoppeinträge für dieses Chromosom werden an das Ende der Kandidatenliste angehängt. Im zweiten Chromosom werden alle Stoppeinträge an den Anfang der Kandidatenlisten gesetzt, so dass dieses Chromosom keine MOs fusioniert. Damit enthält die erste Generation beide Extremfälle. Der Rest der Population wird mit zufällig initialisierten Chromosomen aufgefüllt, bei denen die Kandidatenlisten in den einzelnen Genen zufällig permutiert werden.

Um den Druck auf die Registerbänke verringern zu können, verwendet der Algorithmus eine *no merge probability* p_X, die vom Anwender vorgegeben werden kann und während des Kompiliervorgangs automatisch adaptiert wird. Bei der Erzeugung eines neuen Chromosoms wird für jedes Gen mit Wahrscheinlichkeit p_X der Stoppeintrag an den Anfang der Kandidatenliste verschoben, so dass die entsprechende MO nicht fusioniert wird. Dies erlaubt dem Operation Merging, anfangs größere Lösungen mit geringeren Anforderungen an die Registerbänke zu finden, die später optimiert werden können. Zur Adaption dieser Wahrscheinlichkeit ermittelt der Algorithmus nach jeder Generation den Anteil r_X an Individuen,

für die keine Registerallokation möglich war. Dann wird die Wahrscheinlichkeit wie folgt angepasst:

$$p_X \leftarrow \begin{cases} 0, & r_X = 0 \\ p_X + (1 - p_X)(r_X - 0{,}5), & r_X > 0{,}5 \\ p_X - p_X(0{,}5 - r_X), & r_X < 0{,}5 \end{cases}$$

Die Populationsgröße S für das automatische Operation Merging wird von der Anzahl n der MOs, die mindestens einen Kandidaten haben, abgeleitet und kann durch einen Kommandozeilenparameter x (*merge level*) und einen Divisor D beeinflusst werden:

$$S = \max\{n \cdot 2^{x-2}/D, 3\}.$$

Die minimale Populationsgröße für eine SLM, in der mindestens eine Kombination von Operationen möglich ist, ist 3. Der Divisor D ist in den Standardeinstellungen des Kompilers auf 4 gesetzt, kann aber über einen Kommandozeilenparameter festgelegt werden. Der Programmierer kann im Assemblercode mittels Kompilerdirektiven (Pragmas) das merge level für einzelne SLMs überschreiben, insbesondere das *Operation Merging* für SLMs auch deaktivieren (*merge level* $x = 0$).

Zur Evaluation der Individuen wird zunächst das *Operation Merging* nach den im Chromosom vorgegebenen Paarungen durchgeführt. Dazu werden die MOs in der SLM sequenziell verarbeitet. Für jede MO wertet der Algorithmus die Liste der Kandidaten in der im Gen vorliegenden Reihenfolge aus. Der erste noch verfügbare Kandidat für die aktuelle MO wird zur Kombination gewählt. Kandidaten, die bereits vorher kombiniert wurden oder aus anderen Gründen in die kombinierte SLM aufgenommen wurden, werden übersprungen. Trifft der Algorithmus bei der Suche nach einem Kandidaten auf den Stoppeintrag, wird die aktuelle MO ohne Kombination in die neue SLM aufgenommen. Wie schon bei Verwendung der statischen Heuristik werden die Vorgänger aus dem DDG der MO, die durch die Kombination an eine frühere Stelle gesetzt wird, in die neue SLM aufgenommen, bevor die neue X2-Operation aufgenommen wird. Da im Gegensatz zur statischen Heuristik mit dem EA der Abstand dieser Paarungen häufig größer ausfällt, tritt häufiger der Fall ein, dass viele MOs über die Datenabhängigkeiten aufgenommen werden, die später durch das Chromosom kombiniert worden wären. Daher wird für die MOs, die als Vorgänger der aktuell kombinierten MO übernommen werden, das *Operation Merging* rekursiv durchgeführt.

Nach dem *Operation Merging* wird die neue SLM dem Instruction Scheduling und der Registerallokation übergeben. Die Bewertung der Individuen erfolgt durch Kombination der Informationen vom Instruction Scheduling und der Registerallokation, um eine relative Ordnung der Individuen zu definieren. Dazu wird die Population in verschiedene Abschnitte unterteilt, von denen der erste die Individuen enthält, für die Instruction Scheduling und Registerallokation erfolgreich

durchgeführt wurden. Innerhalb dieses Abschnitts werden die Individuen nach der Größe des erzeugten Mikroprogramms sortiert. Ein weiterer Abschnitt enthält die Individuen, für die eine Registerallokation gescheitert ist. In diesem Fall werden die Individuen nach der Fitness der Registerallokation sortiert, wobei die EA-basierte Registerallokation Vorrang vor der Registerallokation mit statischer Heuristik hat.

Um die neue Generation von Individuen zu erstellen, werden die besten drei Individuen der vorherigen Generation nach der beschriebenen Sortierung kopiert (Elite). Etwa 99 % der neuen Population entstehen durch Crossover und Mutation (dies ist ein einstellbarer Parameter), der Rest wird durch zufällig gebildete Individuen gefüllt. Für das Crossover werden zunächst zwei Elternindividuen durch Tournament Selection (siehe Abschnitt 4.3.1) ausgewählt, die tournament size hat einen Standardwert von 3 Individuen. Das Crossover erzeugt nun zwei Kindindividuen aus den Eltern. Dazu wird für jedes Gen zufällig und unabhängig ausgewählt, ob es aus dem ersten oder zweiten Elternindividuum kopiert wird. Für das zweite Kind wird das Gen dann aus dem jeweils anderen Elternteil übernommen. Die anschließende Mutation verwendet eine adaptive Mutationsrate, mit der für jedes Gen entschieden wird, ob die Kandidatenliste zufällig permutiert wird oder nicht. In den Standardeinstellungen des Kompilers wird eine Mutationsrate von $m = 0{,}01$ verwendet, die aber durch den Anwender geändert werden kann. Von dieser globalen Mutationsrate wird nun für jedes Individuum i eine individuelle Mutationsrate m_i abgeleitet. Dazu wird die mittlere Fitness der Elternindividuen f_p in Relation gesetzt zur Fitness des besten Individuums aus der vorherigen Generation f^*:

$$m_i = m \cdot (1 + \alpha \cdot \min\{f_p - f^*, 10\}) .$$

Der Parameter α bestimmt, wie stark der Einfluss der Fitness auf die Mutationsrate sein soll und ist in den Standardeinstellungen auf 1 gesetzt.

Die Erzeugung von neuen, zufälligen Individuen geschieht wie bei der Initialisierung, indem die Kandidatenlisten für alle MOs zufällig permutiert werden. Hierbei wird, wie oben beschrieben, auch die *no merge probability* verwendet, um auf die Stärke des Operation Merging Einfluss zu nehmen und dem Algorithmus die Möglichkeit zu geben, in schwierigen Situationen mit hohen Anforderungen an die Registerallokation überhaupt eine Lösung zu finden. Hat der Algorithmus dann mehrere Lösungen gefunden, für die eine Registerallokation erfolgreich durchgeführt wurde, sinkt die *no merge probability* durch die Adaption wieder, wodurch der Algorithmus die Bemühungen im Operation Merging wieder steigert. Dadurch können im weiteren Verlauf die gefundenen Lösungen verbessert werden.

Der Algorithmus kann mit einer festen Anzahl an Runden ausgeführt werden, die als Parameter vorgegeben werden kann. Üblicherweise wird jedoch ein dynamisches Abbruchkriterium verwendet, bei dem der Algorithmus stoppt, wenn für

```
   6 MAX          VxR4,       V1R0,       V1R1       ; 1 ADD VxR0, V0R2, V0R0
  12 PERMREG1     VxR8,       VxR4,       V0R2       ; 2 SRI VxR1, VxR0, #1
   3 MAX          VxR2,       VxR1,       V0R2       ; 9 ADD VxR6, VxR4, V0R1
4+10 PERMREG0_X2 VxR3+VxR7, VxR0+VxR4, VxR2+VxR6 ; 7 ADD VxR5, VxR4, V0R0
```

Listing 4.8: Beispielcode aus Listing 4.4 nach Operation Merging und Instruction Scheduling mit EA-basierten Heuristiken (o8, x8)

eine einstellbare Anzahl von Generationen keine Verbesserung in der Fitness des besten Individuums erreicht werden konnte. Dieses Abbruchkriterium wird erst ab der Generation verwendet, in der die erste gültige Lösung gefunden wurde. Für den Fall, dass in der ersten Generation kein Scheduling oder Registerallokation möglich war, versucht der Algorithmus für eine einstellbare Zahl von Runden eine Lösung zu finden. Gelingt dies, startet danach das beschriebene Abbruchkriterium, andernfalls bricht der Algorithmus mit einer Fehlermeldung ab.

Mit diesem Verfahren für das Operation Merging kann der Beispielcode aus Listing 4.4 noch stärker kompaktiert werden als mit der statischen Heuristik, wie in Listing 4.8 zu sehen. Hier werden nur 4 Mikroinstruktionen verwendet, im Gegensatz zu Listing 4.6 mit 5 Mikroinstruktionen. Die statische Heuristik fusioniert die MOs 1 und 7. Da MO 7 von MO 6 abhängt, bedeutet dies, dass die fusionierte MO erst in der zweiten Mikroinstruktion platziert werden kann. Dies verlängert aber den kritischen Pfad (MOs 1, 2, 3 und 4). Die dynamische EA-basierte Heuristik findet eine andere Fusionierung, die den kritischen Pfad nicht beeinträchtigt und so zu einem kompakteren Code führt. In diesem Beispiel ist die Allokation der virtuellen Register mittels der statischen Heuristik erfolgreich durchgeführt worden. Trotzdem zeigt der Beispielcode die virtuellen Register, da diese Informationen im nächsten Abschnitt zur EA-basierten Registerallokation verwendet werden.

Da die Evaluation eines Individuums in der Fitnessfunktion das Instruction Scheduling und die Registerallokation der fusionierten SLM umfasst, wobei für diese Schritte auch wieder die auf genetischen Algorithmen basierenden Implementierungen zum Einsatz kommen können, bedeutet das eine erhöhte Laufzeit des Kompilers. Um diese zu reduzieren, bietet der Operation Merging Algorithmus die Möglichkeit, das Scheduling und die Registerallokation für solche Individuen zu überspringen, für die die Größe der SLM nach dem Merging um ein einstellbares Offset größer ist als die bisher beste gefundene Lösung. Dadurch werden Individuen, die zu einem schwachen Operation Merging führen, frühzeitig ausgeschlossen, was den Zeitbedarf reduziert, wodurch allerdings auch die Qualität der Kompaktierung der SLM reduziert werden kann. Eine Auswertung dieses Features findet sich in Abschnitt 4.4.2.

Wie schon beim Instruction Scheduling werden auch im automatischen Opera-

tion Merging Duplikate in den Chromosomen analysiert. Da zusätzlich die Gene in einem Chromosom nicht vollständig unabhängig voneinander sind, weil ein Gen ein Merging durchführen kann, dass ein anderes Merging in einem anderen Gen verhindert, können auch unterschiedliche Chromosomen zu gleichem Operation Merging führen. Analysen haben gezeigt, dass dies für die größeren Kernel der Applikation nicht auftritt. In den mittleren bis kleinen SLMs, die häufig Kontrollflussoperationen enthalten, ist das Merging aber derart eingeschränkt, dass bis zu 80 % der Chromosomen zu Duplikaten im Merging führen. Daher wurde der Mechanismus zum Erkennen und Überspringen dieser Merges aktiviert, da der Aufwand im Vergleich zur Ersparnis durch Überspringen von Instruction Scheduling und Registerallokation gering ist.

Parallelisierung des evolutionären Algorithmus

Wie schon das auf einem EA basierte Instruction Scheduling (Abschnitt 4.3.1) ist auch der evolutionäre Algorithmus für das Operation Merging mittels OpenMP nach der gleichen Struktur parallelisiert. Die Erzeugung von Individuen in der Initialisierung oder durch evolutionäre Operatoren (Crossover, Mutation) von einer Generation zur nächsten geschieht für alle Individuen parallel. Da die Individuen unabhängig voneinander erzeugt werden können, ist hier keine Synchronisation zwischen den Threads nötig.

Die Berechnung der Fitness der Individuen wird sequentiell durchgeführt, da das Instruction Scheduling und die Registerallokation intern parallelisiert sind und sich eine Verschachtelung der Parallelisierung als ungünstig erwiesen hat.

4.3.3 Evolutionärer Algorithmus für die Registerallokation

Die Registerallokation ist ein wichtiger Aspekt, um hohe Kompaktierung des Codes zu erreichen, da ein starkes Instruction Scheduling und Operation Merging nur dann erfolgreich sein können, wenn auch die Register der SLMs allokiert werden können. Je stärker der Code kompaktiert wird, desto schwieriger wird in den meisten Fällen auch die Registerallokation, da mehr Register gleichzeitig benötigt werden. Die auf der statischen Heuristik basierende Registerallokation (Abschnitt 4.2.3) kann in vielen Fällen keine Allokation aller virtuellen Register finden. Daher wurde ein EA-basiertes Verfahren untersucht.

Die statische Heuristik verteilt die virtuellen Register auf die vorhandenen Registerbänke und versucht dabei, die Auslastung gleichmäßig zu verteilen. Anstatt dieser statischen Zuordnung verwendet der EA-basierte Algorithmus eine flexible Zuordnung der Register zu Registerbänken, die in den Chromosomen kodiert ist und so im Verlauf optimiert werden kann. Durch eine Analyse von Abhängigkeiten zwischen Registern wird die Anzahl der zu kodierenden Register verringert.

Registerabhängigkeiten

Bevor die eigentliche Registerallokation beginnt, analysiert der Algorithmus Abhängigkeiten zwischen Registern und erzeugt einen *Registerabhängigkeitsgraphen* (*register dependency graph, RDG*). Abhängigkeiten zwischen Registern entstehen beispielsweise durch Operation Merging, da die Register, die in einem X2-Registerpaar verwendet werden, Beschränkungen unterliegen, wie in Abschnitt 3.1.3 erklärt. Insbesondere müssen die beiden Register eines Paares in derselben Registerbank allokiert werden. Daher enthält der RDG Knoten für die virtuellen Register und Kanten verbinden Knoten, wenn es eine Abhängigkeit zwischen den entsprechenden Registern gibt. Dabei trägt jede Kante eine Markierung, die angibt, ob die Register in derselben Registerbank liegen müssen, oder ob sie in unterschiedliche Registerbänke allokiert werden müssen. Der letzte Fall tritt beispielsweise auf, wenn zwei X2-Operationen parallel ausgeführt werden, da dann vier Schreibzugriffe auf die Register durchgeführt werden. Bieten die Registerbänke aber jeweils nur zwei Schreibports, müssen die beiden Registerpaare in unterschiedlichen Registerbänken allokiert werden. Den RDG zum Beispielcode in Listing 4.8 zeigt Abbildung 4.7. Register `VxR3` und `VxR7` werden als Zielregister in einem Registerpaar verwendet, daher müssen beide in der gleichen Registerbank allokiert werden. Im parallelen Issue-Slot wird `VxR5` ebenfalls als Zielregister verwendet. Da das Registerpaar bereits die zwei Schreibports einer Registerbank belegt, muss dieses Register also in einer anderen Registerbank allokiert werden. Dies ist im RDG durch eine Kante mit $\neq$-Markierung dargestellt.

Weitere Abhängigkeiten können entstehen, wenn die Registerbänke nicht symmetrisch sind. Als Beispiel sei eine Registerkonfiguration mit zwei Registerbänken gegeben, bei der die erste 4 Lese- und 2 Schreibports bietet, die zweite Registerbank aber nur 2 Lese- und 1 Schreibport. Für derart eingeschränkte Registerbänke gäbe es eine bedingte Abhängigkeit zwischen den Registern `VxR1` und `VxR8`. Wenn nämlich eines dieser Register in Registerbank 1 (mit wenigen Ports) allokiert würde, müsste das andere Register zwingend in Registerbank 0 allokiert werden. Dann enthält der RDG eine entsprechende Kante von `VxR1` zu `VxR8`, die mit $\neq$ markiert ist und an die Bedingung geknüpft ist, dass `VxR1` in Registerbank 1 allokiert wurde.

Das Ergebnis dieser Abhängigkeitsanalyse wird im Folgenden verwendet, um die Anzahl der Freiheitsgrade in der Registerallokation mit dem evolutionärem Algorithmus zu verringern. Für jeden Cluster aus untereinander abhängigen Registern muss im Chromosom nur ein Repräsentant kodiert werden. Diese sind in Abbildung 4.7 als *Wurzelknoten* bezeichnet. In diesem Beispiel reduziert sich die Anzahl der Freiheitsgrade bereits von 9 auf 5. In größeren SLMs mit einer größeren Zahl von X2-Operationen fällt die Reduktion noch größer aus.

Im Falle bedingter Abhängigkeiten, wie im Beispiel von `VxR1` und `VxR8` für die eingeschränkten Registerbänke, müssen beide Register im Chromosom kodiert

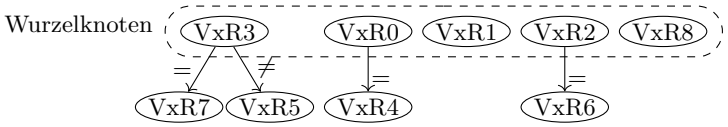


Abbildung 4.7: Registerabhängigkeitsgraph (RDG) für den exemplarischen Code.

werden. Der Eintrag für `VxR8` wird aber nur dann verwendet, wenn `VxR1` in Registerbank 0 allokiert wird. Sollte `VxR1` in Registerbank 1 allokiert werden, dann wird `VxR8` unabhängig von der Belegung im Chromosom in Registerbank 0 allokiert.

Sollten bei der Registerabhängigkeitsanalyse Konflikte erkannt werden, zum Beispiel ein Register, dass aufgrund der Abhängigkeiten in keiner Registerbank allokiert werden darf, kann die Registerallokation noch vor Beginn des aufwändigen evolutionären Algorithmus abgebrochen werden. Außerdem wird der RDG verwendet, um die Anzahl lebendiger Variablen im Verlauf des Mikroprogramms zu analysieren. Sollte dabei die Anzahl der Variablen die Anzahl der verfügbaren Register überschreiten, ist keine Registerallokation möglich und das Verfahren wird ebenfalls abgebrochen.

Evolutionärer Algorithmus für die Registerallokation

Der evolutionäre Algorithmus für die Registerallokation verwendet ein Chromosom, das ein Gen für jeden Wurzelknoten im RDG, also für jeweils einen Repräsentanten eines Cluster untereinander abhängiger Register enthält. Dieses Gen enthält die Zuordnung des entsprechenden Registers zu einer Registerbank des Prozessors. In der Initialisierung eines neuen Chromosoms werden die Register zufällig auf Registerbänke verteilt. Im Gegensatz zum Instruction Scheduling oder dem Operation Merging wird hier kein Chromosom erzeugt, das der Verteilung mittels der statischen Heuristik entspricht. Die statische Heuristik berechnet die Verteilung der Register während der Registerallokation (on-the-fly) und die Kosten für die Berechnung zur Generierung eines Chromosoms sind nicht gerechtfertigt. Allerdings wird zur Registerallokation für ein Mikroprogramm immer zunächst die statische Registerallokation ausgeführt und nur im Falle, dass diese fehlschlägt, die EA-basierte Registerallokation gestartet, so dass durch den Einsatz des EA die Registerallokation nur verbessert werden kann.

Die Populationsgröße S für die EA-basierte Registerallokation hängt von der Anzahl n der unabhängigen Register (Wurzelknoten im RDG) sowie von einem Optimierungslevel P ab:

$$S = \max\{n \cdot 2^{P/8-5}, 5\}.$$

Die minimale Populationsgröße für eine SLM, die virtuelle Register enthält, ist 5. Da der Einsatz der EA-basierten Registerallokation die Laufzeit der Gesamt-

kompilierung wegen der verschachtelten Ausführung (siehe Abbildung 4.4) stark beeinflusst, geht das Optimierungslevel P hier feiner in die Populationsgröße ein als im Vergleich zum Instruction Scheduling oder Operation Merging. Typische Werte für das Optimierungslevel sind $P = 10$ oder $P = 30$.

Wenn der EA eine gültige Registerallokation für den gegebenen Basic Block findet, endet er sofort mit einem Fitnesswert von 0. Andernfalls berechnet sich die Fitness eines Chromosoms aus verschiedenen Faktoren, die die verschiedenen Gründe für das Fehlschlagen der Registerallokation beschreiben. Diese Faktoren sind, angegeben nach absteigender Priorität:

1. Die Anzahl von zusätzlich benötigten Registern, wenn die Anzahl der lebenden Variablen (*live variables*) die Anzahl der zur Verfügung stehenden Register übersteigt.

2. Die Unausgeglichenheit der Verteilung der Register auf die verschiedenen Registerbänke. Hier wird die Differenz der Füllstände der Registerbänke verwendet.

3. Die Anzahl an Konflikten in den Schreib-/Leseports der Registerbänke nach der Registerallokation.

Diese Faktoren fließen gewichtet in eine Summe ein, die den Fitnesswert des entsprechenden Chromosoms angibt. Diese Fitness wird während der tournament selection und der Auswertung des Abbruchkriteriums für den EA verwendet und außerdem, wie in Abschnitt 4.3.1 beschrieben, vom EA-basierten Instruction Scheduling ausgewertet.

Die Populationsgröße bleibt über den Verlauf der Registerallokation konstant. Zur Erzeugung der nächsten Generation wird das beste Individuum der vorherigen Generation kopiert (*Elite*). Etwa 10 % der neuen Individuen werden zufällig erzeugt, der Rest der Population wird durch Rekombination und Mutation erstellt. Zur Rekombination zweier Elternindividuen werden diese durch tournament selection mit einer Größe des tournament sets von 3 Individuen ausgewählt. Das Crossover generiert ein neues Chromosom, bei dem jedes Gen zufällig und gleichverteilt aus einem der beiden Elternchromosomen gewählt wird. Die Mutation ordnet die einzelnen Register jeweils mit einer Wahrscheinlichkeit von 1 % einer neuen Registerbank zu.

Wie oben erwähnt, stoppt der Algorithmus, sobald eine gültige Registerallokation für den gegebenen Basic Block gefunden wurde. Dieses Abbruchkriterium ist geeignet für die Fälle, in denen das Ziel bei der Kompilierung die Kompaktierung des Codes ist, da eine gültige Registerallokation in dieser Hinsicht so gut ist wie eine andere. Sollte das Ziel aber beispielsweise die Verringerung der Verlustleistung (durch verringerte Switching-Activity auf den Adressbussen) sein, kann der Algorithmus eine Registerallokation im weiteren Verlauf unter Umständen verbessern und muss ein anderes Abbruchkriterium auf Basis einer Fitnessbewertung

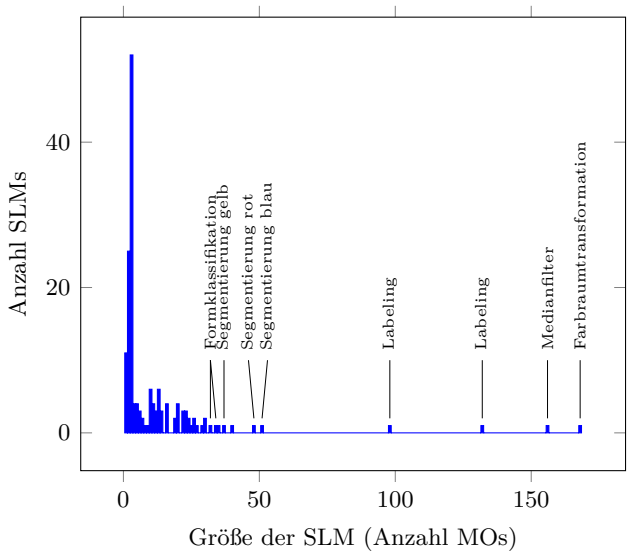


Abbildung 4.8: Verteilung der SLM-Größen in der exemplarischen Anwendung. Die großen, rechenintensiven Kernel der Anwendung sind markiert.

der Allokation verwenden. Im hier beschriebenen Fall stoppt der Algorithmus für nicht erfolgreiche Registerallokationen nach einer vorgegebenen Anzahl von Runden ohne Verbesserung der Fitness.

Für die Registerallokation findet keine Analyse der Chromosomen auf Duplikate statt, da diese selten auftreten und es keine weiteren Stufen im Algorithmus gibt, die im Falle von Duplikaten übersprungen werden könnten. Damit ist der Aufwand zur Erkennung von Duplikaten größer als der Nutzen durch das Überspringen der Bewertung der entsprechenden Individuen.

Parallelisierung des genetischen Algorithmus

Auch die EA-basierte Registerallokation ist mit OpenMP parallelisiert worden. Da keine weitere Verschachtelung von Algorithmen mehr stattfindet (siehe Abbildung 4.4), ist nicht nur die Generierung der einzelnen Individuen, sondern hier auch die Durchführung der Registerallokation und die Berechnung der Fitness für alle Chromosomen unabhängig voneinander parallelisiert.

4.4 Evaluation

Dieser Abschnitt zeigt Ergebnisse aus der Evaluation der Codegenerierung für die TUKUTURI-Architektur anhand der in Kapitel 6 beschriebenen exemplarischen videobasierten Straßenschilderkennung. Abbildung 4.8 zeigt die Häufigkeitsver-

teilung der verschiedenen SLM-Größen in dieser Anwendung. Die große Anzahl kleiner SLMs (mit Größen 1–3 MOs) ist durch Kontrollfluss verursacht. Diese SLMs sind nicht Teil innerer Schleifen und werden daher nicht häufig ausgeführt. Die verschiedenen Subroutinen des Programms, die in der Abbildung markiert sind, stellen die größten SLMs und die rechenintensivsten Teile der Anwendung dar. Mit 168 MOs stellt die innere Schleife der Farbraumtransformation die größte SLM dar.

Zunächst werden Experimente vorgestellt, mit denen die Parameter der evolutionären Algorithmen festgelegt wurden. Danach erfolgt ein Vergleich der Codegenerierung mittels statischer Heuristiken gegen die EA-basierten dynamischen Heuristiken. Zuletzt wird eine Analyse der Codegenerierung auf den einzelnen Subroutinen der Anwendung vorgestellt, die unterschiedliche algorithmische Charakteristiken aufweisen und eine Verallgemeinerung der Ergebnisse auf verschiedene Anwendungen ermöglichen.

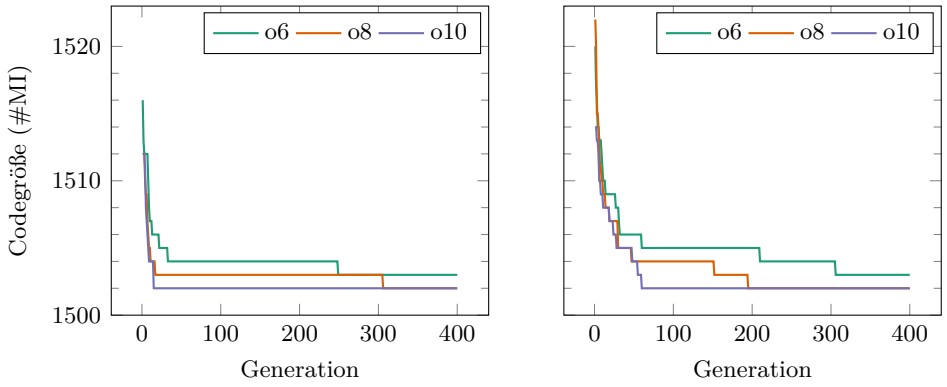
Die Experimente in diesem Abschnitt sind auf dem Rechencluster des Rechenzentrums der Leibniz Universität Hannover[2] durchgeführt worden. Die verwendeten Rechenknoten besitzen zwei Intel Westmere-EP Xeon X5670 Prozessoren mit jeweils 6 Kernen bei 2,93 GHz und insgesamt 48 GB Arbeitsspeicher. Während der Evaluation des Instruction Scheduling und der Registerallokation ohne Operation Merging kam eine Prozessorarchitektur zum Einsatz, die jede funktionale Einheit nur einmal enthält. Für die Evaluation des Operation Merging wurde die Anzahl aller funktionalen Einheiten mit Ausnahme der `MAC`-Unit verdoppelt.

4.4.1 EA-basiertes Instruction Scheduling

Abbruchkriterium

Das EA-basierte Instruction Scheduling kann mit einer festen Anzahl von Generationen verwendet werden, die vom Benutzer vorgegeben werden kann. Üblicherweise verwendet der Algorithmus allerdings ein dynamisches Abbruchkriterium, das mehrere Faktoren berücksichtigt. Aus der Anzahl der MOs in der SLM und der Anzahl verfügbarer Issue-Slots im Prozessor kann eine untere Schranke für die Größe der kompaktierten SLM bestimmt werden, indem die Anzahl der MOs durch die Anzahl der Issue-Slots geteilt wird. Diese minimale Größe kann nur erreicht werden, wenn Datenabhängigkeiten zwischen den MOs dies nicht verhindern. Daher wird zusätzlich aus dem Datenabhängigkeitsgraph (DDG) für die SLM ein *kritischer Pfad* bestimmt, also eine Folge von MOs, die aufgrund von Abhängigkeiten sequentiell ausgeführt werden müssen und für die die Summe der Latenzen maximal ist. Dieser kritische Pfad ist immer mindestens so lang wie die eben beschriebene untere Schranke. Sollte während des Instruction Scheduling die Länge des kritischen Pfades oder die minimale Größe der SLM

[2] `https://www.luis.uni-hannover.de/scientific_computing.html`



(a) Registerkonfiguration RF64 **(b)** Registerkonfiguration RF44

Abbildung 4.9: Größe des kompaktierten Codes (Anzahl der MIs) für das jeweils beste Individuum über 400 Generationen für verschiedene Registerkonfigurationen.

erreicht werden, bricht der Algorithmus ab. Andernfalls nutzt der Algorithmus ein dynamisches Abbruchkriterium, bei dem der Algorithmus stoppt, wenn für eine gewisse Anzahl an Generationen keine Verbesserung in der Fitness des besten Individuums auftritt. Um dieses Abbruchkriterium zu bestimmen, wurden verschiedene Kompilierungen mit einer festen Anzahl von 400 Generationen und unterschiedlichen Populationsgrößen durchgeführt. Diese Anzahl von Generationen ist ausreichend groß, so dass nahezu alle Kompilierungen zu einer gemeinsamen finalen Programmgröße konvergieren. Um außerdem den Einfluss von Druck im Registerfile zu untersuchen, wurden die Kompilierungen mit zwei verschiedenen Registerkonfigurationen ausgeführt. Das Standardregisterfile RF64 enthält zwei Registerbänke mit jeweils 32 Registern und jeweils 4 Lese- und 2 Schreibports pro Registerbank. Um den Registerdruck zu erhöhen, wurde eine zweite, kleinere Registerkonfiguration RF44 mit zwei Registerbänken von jeweils 22 Registern und der gleichen Anzahl von Lese- und Schreibports wie die RF64 Konfiguration, untersucht. Der Verlauf der Fitness des jeweils besten Individuums über die 400 Generationen ist in Abbildung 4.9 gezeigt.

Für größere Populationen, also höhere Optimierungslevel (o), konvergieren die Kurven schneller gegen die minimale Programmgröße, die nach 400 Runden von fast allen Läufen erreicht wird. Auch die Größe des Registerfiles hat Einfluss auf die Konvergenz: Für das eingeschränkte Registerfile RF44 ist die Konvergenz langsamer als für das größere Registerfile RF64. Die Plateaus in den Kurven zeigen eine Reihe von Generationen, in denen die Fitness des besten Individuums nicht verbessert werden konnte. Das Abbruchkriterium wird nun so gewählt, dass die kürzeren Plateaus noch nicht zu einem Abbruch des Algorithmus führen, sondern der Algorithmus diese überwinden und weitere Verbesserungen in der Fitness und damit kleinere Mikroprogramme erreichen kann. Tabelle 4.1 zeigt die erreichbare

Tabelle 4.1: Abbruchkriterium für Instruction Scheduling mit evolutionärem Algorithmus für verschiedene Populationsgrößen und Registerfilekonfigurationen. Gezeigt ist die Anzahl an Generationen ohne Verbesserung der Fitness des besten Individuums, die der Algorithmus tolerieren muss, um den angegebenen Prozentsatz an Kompaktierung zu erreichen.

	RF44 / RF64					
Kompakt.	o6	o8	o10	o12	o14	o16
95 %	26/10	30/9	25/8	24/7	23/5	21/5
96 %	30/11	31/9	26/8	25/7	25/7	21/6
97 %	45/11	38/10	28/8	42/9	37/7	27/6
98 %	75/12	75/10	37/9	62/13	50/8	36/9
99 %	191/67	132/12	83/10	101/23	88/12	41/14

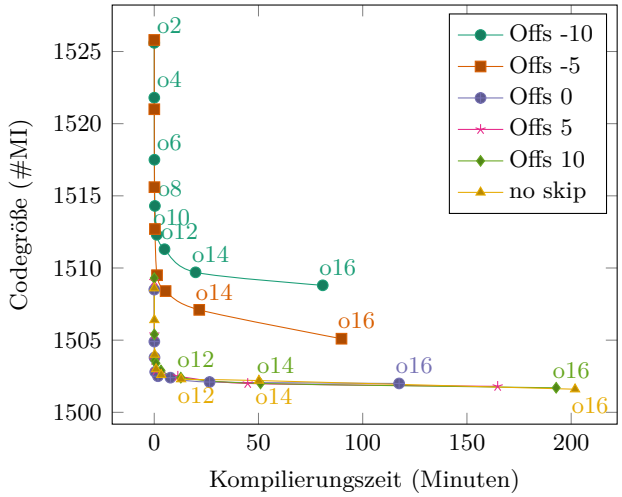
Kompaktierung in Prozent, für verschiedene Abbruchkriterien, bezogen auf die größtmögliche Kompaktierung nach 400 Generationen. Ein Abbruchkriterium von 25 Generationen ohne Verbesserung wurde gewählt, so dass etwa 95 % der Kompaktierung für die größeren Populationen erreicht werden.

Überspringen der Registerallokation

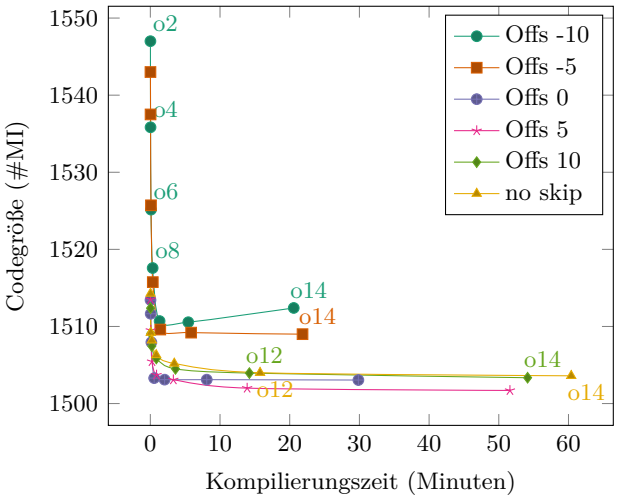
Zur Berechnung der Fitness eines Individuums in dem evolutionären Algorithmus für das Instruction Scheduling wird nach dem Scheduling der Operationen, wie es durch das Chromosom vorgegeben wird, die Registerallokation durchgeführt, wie in Abbildung 4.4 gezeigt. Dieser Schritt ist notwendig um festzustellen, ob das generierte Scheduling überhaupt sinnvoll ist. Dabei kann sowohl die Registerallokation mittels statischer Heuristiken als auch die EA-basierte Registerallokation zum Einsatz kommen.

Die verschachtelte Ausführung der Registerallokation für jedes Scheduling-Individuum führt zu einer erhöhten Kompilierungszeit. Daher ist im Schedulingalgorithmus eine Methode zum Überspringen der Registerallokation für einzelne Individuen implementiert. Nachdem das Instruction Scheduling für alle Individuen der aktuellen Generation ausgeführt wurde, werden die Individuen nach der erreichten Codegröße (absteigend) sortiert. Als Grenzgröße wird die Größe des n-ten Codes gewählt, wobei n die Größe der Elite-Gruppe ist (siehe Erläuterungen in Abschnitt 4.3.1). Für alle Individuen, für die der Code nach Instruction Scheduling größer als ein gegebenes Offset von dieser Grenzgröße ist, wird die Registerallokation übersprungen. In der Berechnung der Fitness für diese Individuen wird ein Strafterm auf die Codegröße addiert. Das Offset kann sogar negativ gewählt werden, wodurch dann die Registerallokation nur für Individuen durchgeführt wird, deren Code nach Scheduling um den gewählten Wert *kleiner* als die Grenzgröße ist.

Abbildung 4.10 zeigt Ergebnisse für unterschiedliche Optimierungslevel (o)



(a) Registerkonfiguration RF64



(b) Registerkonfiguration RF44

Abbildung 4.10: Überspringen der Registerallokation für Individuen, deren kompaktierter Code größer als ein angegebenes Offset von der kompaktesten Lösung ist.

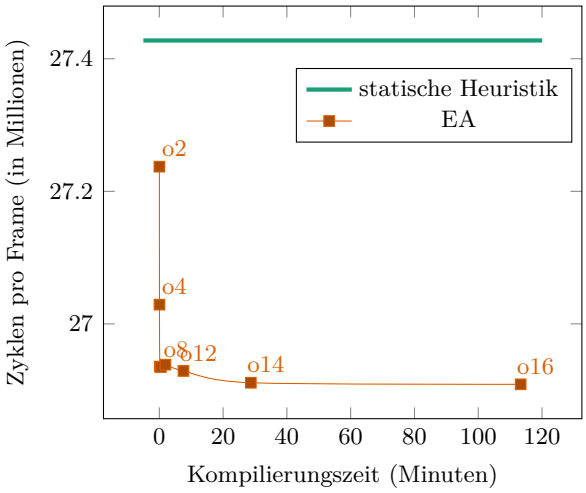
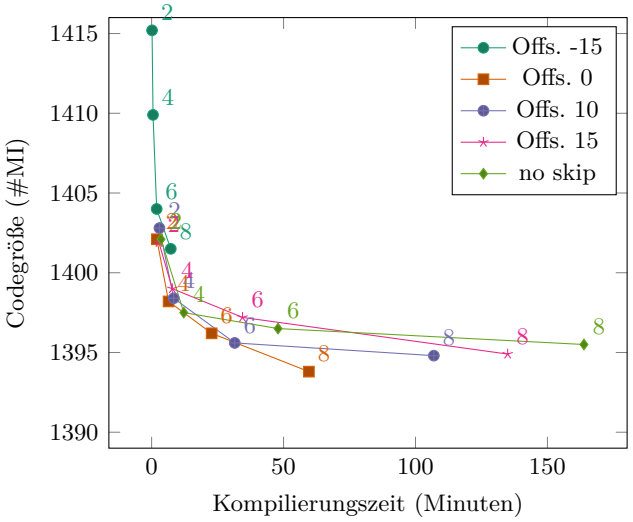


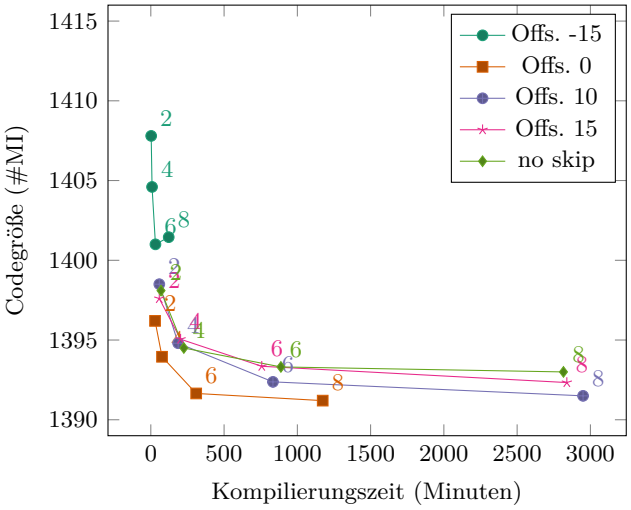
Abbildung 4.11: Prozessorzyklen und Kompilierungszeit in Abhängigkeit von der Populationsgröße im Instruction Scheduling. Zum Vergleich ist außerdem die Zyklenzahl für den Code nach Scheduling mit statischer Heuristik gegeben, der 27 427 607 Zyklen pro Frame benötigt.

im Instruction Scheduling, verschiedene Offsets und Registerkonfigurationen. Je kleiner das Offset gewählt wird, desto mehr Registerallokationen werden übersprungen und die Kompilierungszeit sinkt. Für Offsets größer oder gleich 0 ist dabei kein Einfluss auf die erreichte Kompaktierung des Codes zu erkennen. Wird ein negatives Offset gewählt, nimmt die Kompilierungszeit weiter ab, allerdings erreicht der Algorithmus dann auch nicht mehr die gleiche Kompaktierung wie mit größeren Offsets. Daher wurde in den Standardeinstellungen des Codegenerators ein Offset von 0 gewählt.

Die verringerte Codegröße des kompaktierten Programms hat auch Einfluss auf die Ausführungsgeschwindigkeit des Programms, wie in Abbildung 4.11 gezeigt. Hier ist die Anzahl an Prozessorzyklen zur Verarbeitung eines Frames der Verkehrszeichenerkennung für den Code nach Scheduling mit verschiedenen Populationsgrößen und auch für den Code nach Scheduling mit der statischen Heuristik gezeigt. Der EA-basierte Scheduler konnte die Anzahl der Zyklen auf 26 908 817 senken. Im Vergleich zu 27 427 607 Zyklen mit der statischen Heuristik ist dies eine Reduktion um 2 % einzig durch das verbesserte Instruction Scheduling.



(a) Codegröße gegen Kompilierungszeit für Merging Level x2.



(b) Codegröße gegen Kompilierungszeit für Merging Level x6.

Abbildung 4.12: Überspringen des Instruction Scheduling und der Registerallokation für verschiedene Populationsgrößen im Operation Merging (x) und Instruction Scheduling (o) bei verschiedenen Offsets.

Tabelle 4.2: Anzahl der Generationen im Operation Merging ohne Verbesserung in der Fitness des besten Individuums, die der Algorithmus tolerieren muss, um einen gegebenen Prozentsatz an Kompaktierung zu erreichen.

Kompakt.	o2 x2	o4 x4	o6 x6	o8 x6
95 %	24	13	16	12
96 %	29	19	19	12
97 %	32	24	23	12
98 %	34	31	31	31
99 %	55	43	38	31

4.4.2 Automatisches Operation Merging

Abbruchkriterium

Der evolutionäre Algorithmus für das Operation Merging kann mit einer festen Anzahl von Generationen verwendet werden, die vom Benutzer vorgegeben wird. Typischerweise wird jedoch ein dynamisches Abbruchkriterium eingesetzt, das den Algorithmus stoppt, wenn für eine vorgegebene Anzahl an Generationen die Fitness des besten Individuums nicht verbessert werden konnte. Zur Evaluation dieses Abbruchkriteriums sind verschiedene Kompilierungen mit unterschiedlichen Populationsgrößen im Instruction Scheduling und dem Operation Merging für eine Registerkonfiguration RF64 durchgeführt worden. Das Operation Merging erzeugt durch die Verlängerung der Lebensdauer verschiedener Variablen selbst Druck in den Registerbänken, so dass dieser nicht durch kleinere Registerkonfigurationen erzeugt werden muss. Für das Instruction Scheduling werden die Parameter gewählt, wie sie im vorigen Abschnitt ermittelt wurden. Tabelle 4.2 zeigt die Anzahl der Generationen ohne Verbesserung in der Fitness des besten Individuums, die der Algorithmus tolerieren muss, um verschiedene Grade an Kompaktierung zu erreichen. Ein Abbruchkriterium von 15 Generationen ohne Verbesserung wurde gewählt, um für die größeren Populationen etwa 95 % der möglichen Kompaktierung zu erreichen.

Überspringen des Instruction Scheduling

Durch die verschachtelte Ausführung des Instruction Scheduling und der Registerallokation in der Fitnessfunktion des Operation Merging, wie in Abbildung 4.4 gezeigt, ist der Zeitbedarf der Kompilierung relativ hoch. Um dem entgegenzuwirken, kann der Algorithmus das Instruction Scheduling und damit auch die Registerallokation für einzelne Individuen im evolutionären Algorithmus für das Operation Merging überspringen. Nach dem Operation Merging einer SLM mit den Informationen aus einem Chromosom liegt eine fusionierte SLM vor. Ist deren Größe größer als die beste bisher gefundene fusionierte SLM, wobei ein durch den Benutzer vorgegebenes Offset addiert wird, überspringt der Algorithmus das

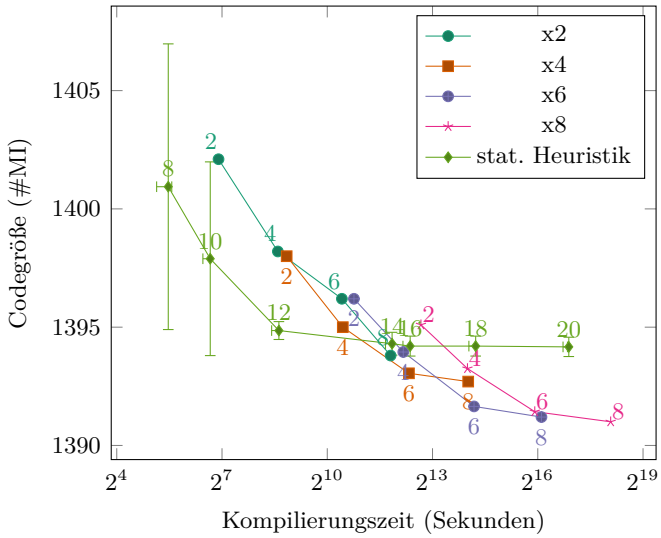


Abbildung 4.13: Vergleich von Operation Merging mit statischer und EA-basierter dynamischer Heuristik für unterschiedliche Populationsgrößen in Merging und Instruction Scheduling.

Instruction Scheduling für diese fusionierte SLM. Die Fitness ist dann die Größe dieser fusionierten SLM und wird mit einem zusätzlichen Strafterm versehen.

Die Ergebnisse sind in Abbildung 4.12 gezeigt. Für kleinere Offsets werden Instruction Scheduling und Registerallokation häufiger übersprungen, wodurch die Kompilierungszeit sinkt. Für Offsets größer oder gleich 0 ist die erreichte Kompaktierung nahezu gleich. Für Offsets kleiner als 0 konvergiert der Algorithmus schneller, aber zu größeren Programmen. Daher wurde für das Offset ein Wert von 0 gewählt.

Abbildung 4.13 zeigt einen Vergleich des Operation Merging mit statischer Heuristik und mit EA-basierter dynamischer Heuristik mit unterschiedlichen Populationsgrößen im Merging und Scheduling. Die Zahlen an den Datenpunkten geben das Optimierungslevel des EA-basierten Schedulings an. Das aggressive Merging der statischen Heuristik erhöht die Schwierigkeit im Scheduling und der Registerallokation, indem es die Anzahl der aktiven Variablen im Programm (live variables) erhöht. Daher kann der Instruction Scheduler für Optimierungslevel o2 keine Lösung finden und auch für Levels o6 und o8 wird in über 90 % der Fälle keine Lösung gefunden. Außerdem ist für diese kleinen Populationsgrößen die Varianz in der erreichten Codegröße hoch. Das starke Merging erhöht auch die minimale Codegröße, die von der statischen Heuristik erreicht werden kann.

Tabelle 4.3: Abbruchkriterium für die EA-basierte Registerallokation für verschiedene Populationsgrößen und verschiedene Registerkonfigurationen. Die Parameter o, x und P sind die entsprechenden Optimierungslevel für Instruction Scheduling, Operation Merging und Registerallokation.

(a) Registerkonfiguration RF64_4r2w4r2w

	o2 x2			o4 x4		
Kompakt.	P10	P20	P30	P10	P20	P30
95 %	58	47	25	57	49	25
96 %	65	54	32	65	57	32
97 %	75	64	45	74	67	44
98 %	89	79	69	88	82	69
99 %	111	106	109	111	108	111

(b) Registerkonfiguration RF52_4r2w4r2w

	o2 x2			o4 x4		
Kompakt.	P10	P20	P30	P10	P20	P30
95 %	60	47	33	63	53	38
96 %	68	56	43	71	61	50
97 %	79	68	58	82	72	66
98 %	94	87	80	97	88	90
99 %	117	117	114	120	115	125

(c) Registerkonfiguration RF64_4r2w2r1w

	o2 x2			o4 x4		
Kompakt.	P10	P20	P30	P10	P20	P30
95 %	42	24	14	27	18	11
96 %	49	29	18	31	21	13
97 %	58	38	26	38	25	17
98 %	71	54	40	48	22	25
99 %	95	87	72	69	53	48

4.4.3 EA-basierte Registerallokation

Abbruchkriterium

Die Registerallokation mit evolutionärem Algorithmus stoppt, sobald eine gültige Registerallokation für den kompaktierten Basic Block gefunden wurde, da im Hinblick auf Kompaktierung eine Registerallokation so gut ist wie eine andere. Falls das Optimierungsziel nicht Kompaktierung des Programms, sondern Beispielsweise eine Reduktion der Leistungsaufnahme ist, muss der Algorithmus möglicherweise verschiedene mögliche Registerallokationen evaluieren. Solange noch keine gültige Allokation gefunden wurde, verwendet der Algorithmus ein dynamisches Abbruchkriterium, bei dem der Algorithmus stoppt, wenn für eine vorgegebene Anzahl von Generationen keine Verbesserung der Fitness des besten Individuums stattfindet.

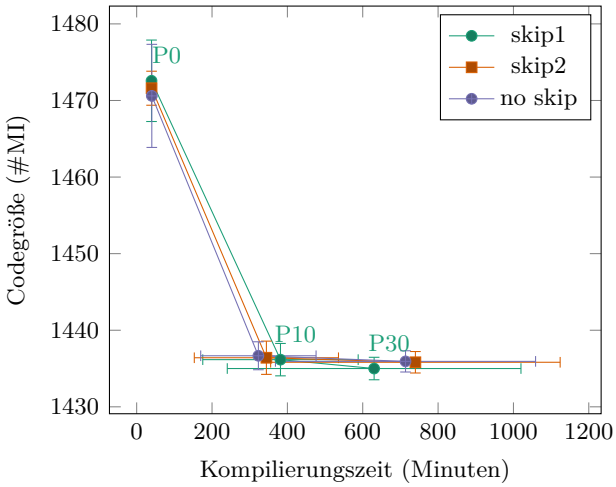


Abbildung 4.14: Überspringen der EA-basierten Registerallokation für Scheduling und Merging Levels o4 x4 und Registerkonfiguration RF64_4r2w2r1w. Die Kurven zeigen Datenpunkte für P0, P10, und P30

Um dieses Abbruchkriterium festzulegen, wurden Kompilierungen mit verschiedenen Populationsgrößen im Instruction Scheduling, Operation Merging und der Registerallokation durchgeführt. Die Parameter für Operation Merging und Instruction Scheduling wurden wie in den vorherigen Abschnitten gewählt. Für die Registerallokation wurden 150 Generationen vorgegeben. Dabei wurden drei verschiedene Registerkonfigurationen untersucht. Tabelle 4.3 zeigt für verschiedene Optimierungslevel der Registerallokation (Parameter P, siehe Abschnitt 4.3.3) die Anzahl der nötigen Generationen ohne Verbesserung der Fitness des besten Individuums, um verschiedene Prozentsätze der maximalen Kompaktierung zu erreichen. Ein Abbruchkriterium von 50 Generationen wurde gewählt, um etwa 95 % der maximalen Kompaktierung zu erreichen.

Die Anzahl der Register in der Registerkonfiguration hat wenig Einfluss auf die Registerallokation (Konfigurationen (a) und (b)), da die Anzahl der aktiven Variablen, die in Registern gehalten werden müssen, im Instruction Scheduling beeinflusst wird. Eine Reduktion der Lese-/Schreibports in der Registerbank hingegen reduziert die Freiheitsgrade bei der Registerallokation und führt zu einer schnelleren Konvergenz.

Überspringen der Registerallokation

Nach dem Instruction Scheduling wird die Registerallokation zunächst für alle Individuen mittels der statischen Heuristik durchgeführt. Nur in den Fällen, in denen dies nicht erfolgreich ist, wird der evolutionäre Algorithmus gestartet. Da dieser Algorithmus die Kompilierungszeit mitunter drastisch erhöht,

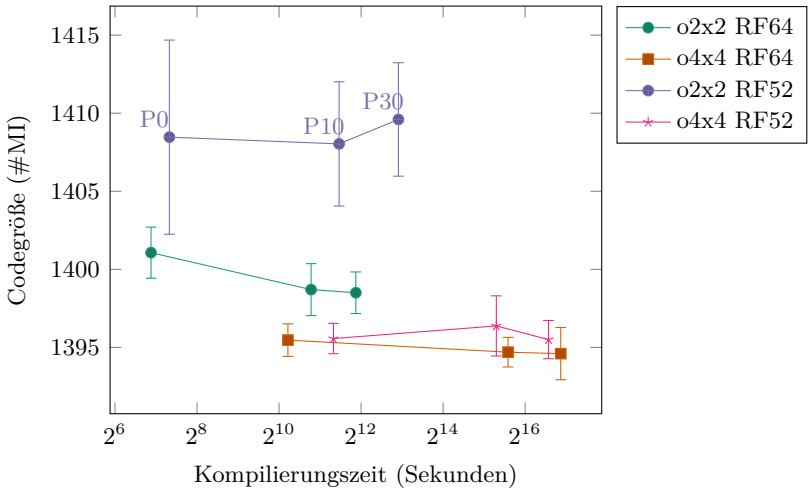


Abbildung 4.15: EA-basierte Registerallokation für zwei Registerkonfigurationen RF64_4r2w4r2w und RF52_4r2w4r2w. Die Standardabweichung an jedem Datenpunkt ist eingezeichnet.

kann die Anzahl der Scheduling-Individuen, für die eine EA-basierte Registerallokation durchgeführt wird, begrenzt werden. Allerdings zeigen die Kurven in Abbildung 4.14, dass die Reduktion der Anzahl erlaubter EA-basierter Registerallokationen auf 1 pro Generation die Kompilierungszeit nicht reduziert. Stattdessen wird dadurch die Zahl der Individuen, die dem EA für das Scheduling zur Verfügung stehen reduziert, was zu langsamerer Konvergenz führt. Die hohe Varianz in den Ergebnissen und der minimale Effekt führten dazu, dass dieser Mechanismus in den Standardeinstellungen nicht aktiviert ist.

Einen Vergleich der Registerallokation mit statischer Heuristik mit der EA-basierten Registerallokation zeigen die Abbildungen 4.15 und 4.16 für Registerkonfigurationen RF64, RF52 und RF64 mit eingeschränkten Ports. Sind genügend Ports zum Zugriff auf die Registerbänke verfügbar, kann die Registerallokation nur fehlschlagen, wenn mehr aktive Variablen als freie Register zur Verfügung stehen. Die Anzahl der aktiven Variablen wird aber maßgeblich vom Instruction Scheduling beeinflusst und nicht durch die Registerallokation. Daher ist der Effekt der EA-basierten Registerallokation (P10, P30) im Vergleich zur statischen Registerallokation (P0) minimal.

Im Fall einer Registerbank mit eingeschränkter Anzahl an Lese-/Schreibports übertrifft der EA-basierte Algorithmus zur Registerallokation die statische Heuristik, wie in Abbildung 4.16 zu sehen. In einer solchen Registerkonfiguration ist die Anzahl der verwendeten Ports der einschränkende Faktor, der direkt durch die Registerallokation behandelt wird. Das Instruction Scheduling hat hier wenig Einfluss. Für kleine Populationsgrößen in Scheduling und Merging (o2 x2) ist

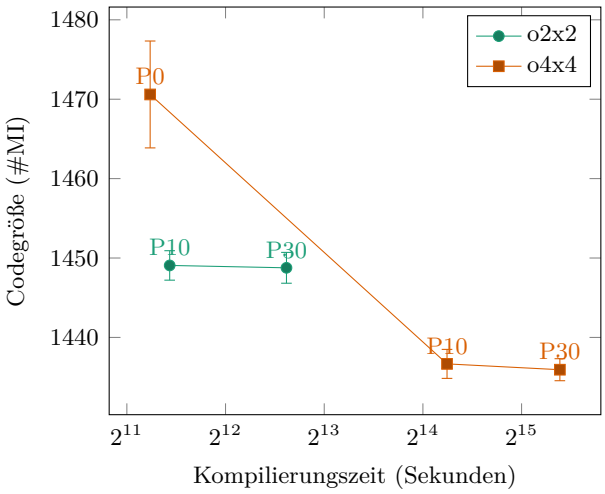


Abbildung 4.16: EA-basierte Registerallokation für Scheduling- und Merging-Level o4 x4 für Registerkonfiguration RF64_4r2w2r1w. Die Standardabweichung an jedem Datenpunkt ist eingezeichnet.

Abbildung 4.17: Parallelisierungsschema des Codegenerators.

die Registerallokation auf Basis der statischen Heuristik nicht erfolgreich. Für größere Populationen kann schließlich eine Lösung gefunden werden, doch die EA-basierte Registerallokation verringert die Codegröße um etwa 3 % gegenüber der statischen Heuristik.

4.4.4 Parallelisierung

Der Einsatz der evolutionären Algorithmen für das Operation Merging, Instruction Scheduling und die Registerallokation liefert bessere Ergebnisse im Vergleich zu den statischen Heuristiken, doch die Komplexität erhöht auch die Kompilierungszeit. Um die Kompilierungszeit zu verringern, ist der Codegenerator durch den Einsatz von OpenMP parallelisiert worden. Wie bereits in den Abschnitten zu den einzelnen evolutionären Algorithmen erklärt, wird keine verschachtelte Parallelisierung verwendet, da sich dies als ungünstig herausgestellt hat. Stattdessen ist jeder Algorithmus intern parallelisiert, wie in Abbildung 4.17 gezeigt.

Das Operation Merging findet parallel für alle Individuen einer Generation statt.

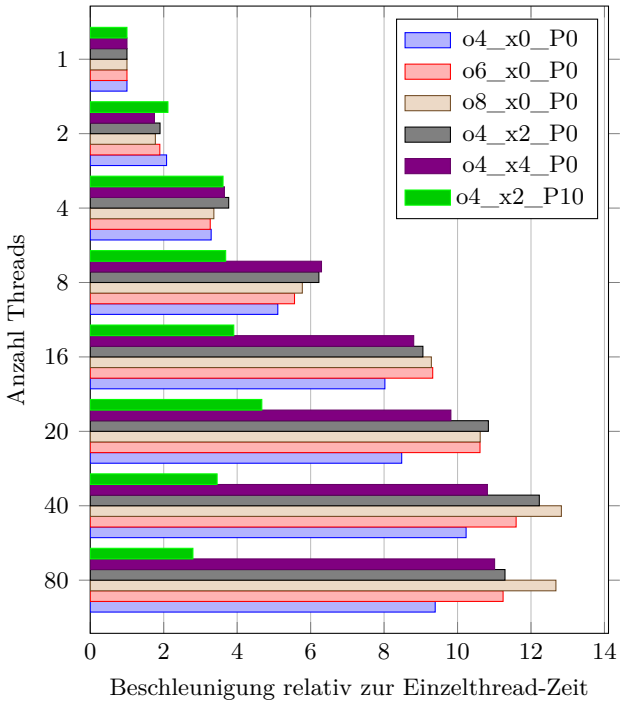


Abbildung 4.18: Ausführungszeit normiert auf die jeweiligen Einzel-Thread Läufe.

Auch die Berechnung der folgenden Generation aus der aktuellen Generation ist parallelisiert. Am Synchronisierungspunkt (1) liegen nach jeder Generation die fusionierten SLMs vor. Ist das automatische Operation Merging nicht aktiviert, liegt an diesem Punkt einzig die ursprüngliche SLM vor. Anschließend werden die fusionierten SLMs zur Berechnung der Fitnessfunktion sequentiell dem Instruction Scheduling übergeben.

Das Instruction Scheduling wird nach den in den Chromosomen gegebenen Gewichten wieder parallel für alle Individuen ausgeführt und produziert einen Basic Block für jedes Individuum. Außerdem wird die Registerallokation mit statischer Heuristik direkt im Anschluss ebenfalls parallel ausgeführt. Sollte außerdem die Registerallokation mit evolutionärem Algorithmus aktiviert sein, wird parallel für die Basic Blocks noch die Datenabhängigkeitsanalyse durchgeführt und jeweils ein RDG berechnet. Diese Informationen liegen dann für jeden kompaktierten Basic Block an Synchronisierungspunkt (2) vor.

Die Registerallokation wird dann für die Basic Blocks sequentiell ausgeführt und ist intern parallelisiert. Die Berechnung der Chromosomen und die eigentliche Registerallokation findet parallel statt. An Synchronisierungspunkt (3) liegt das fertig kompilierte Programm vor, das dann als Binärcode geschrieben werden kann.

Abbildung 4.18 zeigt die Beschleunigung des Codegenerators für verschiedene Parameterkombinationen, jeweils bezogen auf die Ausführungszeit mit nur einem Thread. Diese Experimente wurden auf einem Server mit zwei Intel Xeon 6148 CPUs mit 40 Kernen bei 3,7 GHz und 377 GB Arbeitsspeicher ausgeführt.

Für Kompilierungen, die ausschließlich EA-basiertes Instruction Scheduling verwenden (x und P sind 0), skaliert die Beschleunigung bis zu einer Threadanzahl von 40 und flacht danach ab. Dies kann damit begründet werden, dass bei einer höheren Threadanzahl Hyperthreading eingesetzt wird, was zusätzlichen Overhead mit sich bringt. Es ist auch zu erkennen, dass die Kompilierungen mit größerer Population (also höherem o) bei hoher Anzahl von Threads stärker von der Parallelisierung profitieren als die Läufe mit kleinerer Population, da sie mehr Potential zur Parallelisierung bieten.

Wird das Operation Merging aktiviert (x2, x4), ergibt sich ein ähnlicher Speed-Up wie beim Instruction Scheduling allein. Für eine Threadanzahl größer als 8 zeigen die Kompilierungen mit kleinerer Population (x2) etwas stärkere Beschleunigung als die Kompilierungen mit größerer Population (x4). Dies ist durch eine unbalancierte Arbeitsverteilung bei höheren Populationen begründet, da manche Chromosomen im Operation Merging schneller verarbeitet werden können als andere.

Kompilierungen mit EA-basierter Registerallokation (P10) zeigen keine Beschleunigung für Threadanzahlen größer als 20, da durch die geringe Populationsgröße die Parallelisierung auf weitere Threads nicht sinnvoll ist. Außerdem ist auch hier die Verteilung der Laufzeit der einzelnen Threads wieder unausgeglichen, wenn zum Beispiel die Registerallokation aufgrund von Konflikten frühzeitig abgebrochen wird, wodurch eine starke Parallelisierung behindert wird.

4.4.5 Evaluation der EA-basierten Codegenerierung

Für die folgenden Experimente wurde eine TUKUTURI-Konfiguration verwendet, in der alle funktionalen Einheiten bis auf die MAC-Einheit verdoppelt wurden, auch wenn der Code bzw. Kompiler kein Operation Merging verwendet.

Für Registerkonfigurationen RF64 und RF52 mit 4r2w Lese-/Schreibports in jeder Partition wird die Registerallokation mit statischer Heuristik (P0) verwendet. Aus Abbildung 4.19 können die folgenden Schlüsse gezogen werden:

- Das Instruction Scheduling mit statischer Heuristik ist unabhängig von der Anzahl der Register in den Registerbänken, da die Anzahl aktiver Variablen im Programm gleich bleibt, wenn kein Operation Merging eingesetzt wird. Daher ist in beiden Registerkonfigurationen die Kompilierung (o1 x0) gleich. Das Instruction Scheduling mit EA-basierter Heuristik (o2 x0, o4, x0, ...) kann die Codegröße um etwa 3,7 % und die Anzahl der Zyklen pro Frame um etwa 0,3 % gegenüber der statischen Heuristik verbessern. Für größere

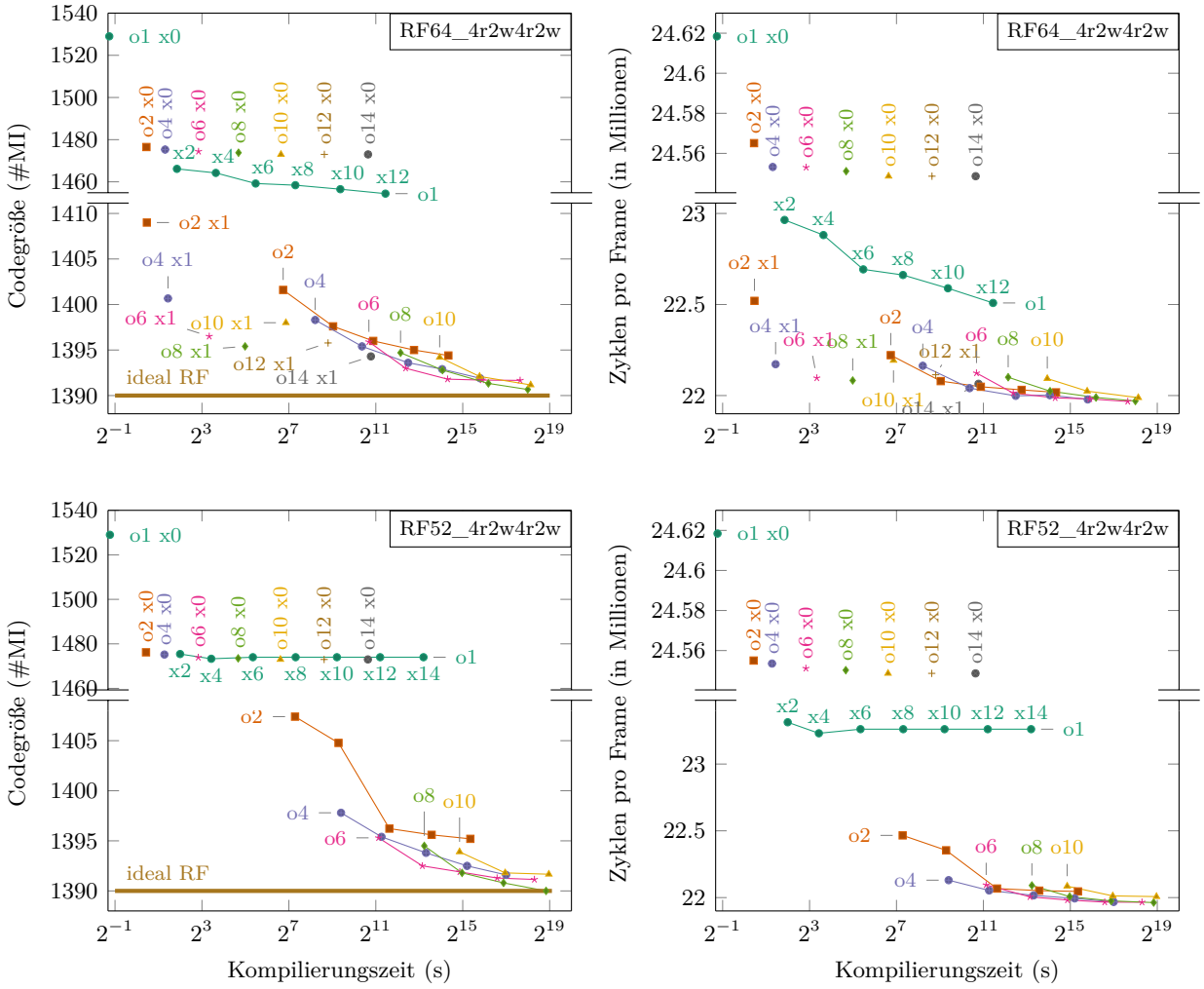


Abbildung 4.19: Evaluation der EA-basierten Codegenerierung für Registerbänke RF64/52 mit 4 Lese- und 2 Schreibports pro Registerpartition. Verbundene Kurven zeigen Datenpunkte für x2, x4, ..., wie an der o1-Kurve angegeben.

Optimierungslevel (o) wird eine Sättigung erreicht. Der kompaktierte Code enthält dann durchschnittlich 1,54 Instruktionen pro Taktzyklus (*instructions per cycle, IPC*). Gewichtet mit der Anzahl der Ausführungen der einzelnen Mikroinstruktionen ergibt sich ein dynamischer IPC von 1,98, mit einem theoretischen Maximum von 2.

- Instruction Scheduling mit statischer Heuristik ist nicht erfolgreich für Code nach Operation Merging mit statischer Heuristik (o1 x1), da das starke Merging die Anzahl der aktiven Variablen zu stark erhöht und keine

Registerallokation möglich ist. EA-basiertes Operation Merging kann auch weniger Operationen fusionieren, um so eine erfolgreiche Registerallokation zu ermöglichen. In der Registerkonfiguration RF64 sinkt die Codegröße mit steigendem Merginglevel (o1 x2, o1 x4, ...). In der Konfiguration RF52 ist der Registerdruck zu groß, um weitere Verbesserungen zu erreichen und schon die kleinen Merginglevels sind ausreichend.

- Mit Operation Merging basierend auf statischer Heuristik berücksichtigt das EA-basierte Instruction Scheduling nicht nur die Kompaktierung des Codes, sondern hilft außerdem bei der Verringerung des Registerdrucks, indem es die Anzahl aktiver Variablen reduziert. Dadurch wird es möglich, Lösungen in der RF64-Konfiguration zu finden (o2 x1, o4 x1, ...). Dabei reicht die Verringerung in der Codegröße von 7,8 % (o2) bis 8,8 % (o14) und in der Anzahl der Taktzyklen pro Frame von 8,5 % (o2) bis 10,4 % (o14) im Vergleich zum Instruction Scheduling mit statischer Heuristik (o1 x0). In der Konfiguration RF52 ist der Registerdruck zu hoch, da das statische Merging zu viele aktive Variablen erzeugt, so dass hier kein Scheduling und Registerallokation erfolgen kann.

- In der Kombination beider EA-basierter Algorithmen unterstützt das Instruction Scheduling das Operation Merging durch eine Reduktion des Registerdrucks, so dass mehr Operationen fusioniert werden können, was zu kompakterem Code führt. Im Vergleich von (o1 x2) mit (o2 x2) kann die Codegröße um 4,4 % bzw. 4,6 % in Registerkonfiguration RF64 bzw. RF52 verringert werden. Dies resultiert in einer Verringerung der Taktzyklen um 3,2 % bzw. 3,6 %. Für (o1 x2) im Vergleich zu (o10 x2) reduziert sich die Codegröße um 4,9 % bzw. 5,5 % und die Zahl der Taktzyklen um 3,8 % bzw. 5,3 %. Weder das Instruction Scheduling noch das Operation Merging kann allein eine hohe Kompaktierung erreichen. Die Abbildung zeigt eine Pareto-Front, auf der die Levels für Instruction Scheduling und Operation Merging ähnlich oder gleich sind (o2 x2, o4 x2, o4 x4, o6 x4, o6 x6, ...).

- EA-basiertes Operation Merging und Instruction Scheduling mit Registerkonfigurationen RF64/RF52 finden Lösungen mit der gleichen Größe wie für unbeschränkte Registerbänke (RF ideal, 1024 Register, 8/4 Lese-/Schreibports pro Partition). Dies ist eine Reduktion von 9,1 % in Codegröße und 10,8 % in Taktzyklen pro Frame verglichen mit (o1 x0). Im Vergleich mit einer Prozessorkonfiguration, in der die funktionalen Einheiten jeweils nur einfach vorhanden sind, entspricht dies einer Reduktion in Taktzyklen um 19,8 %.

Für eine Registerkonfiguration mit eingeschränkter Anzahl von Ports in der zweiten Partition kommt zusätzlich zur Registerallokation mit statischer Heuristik

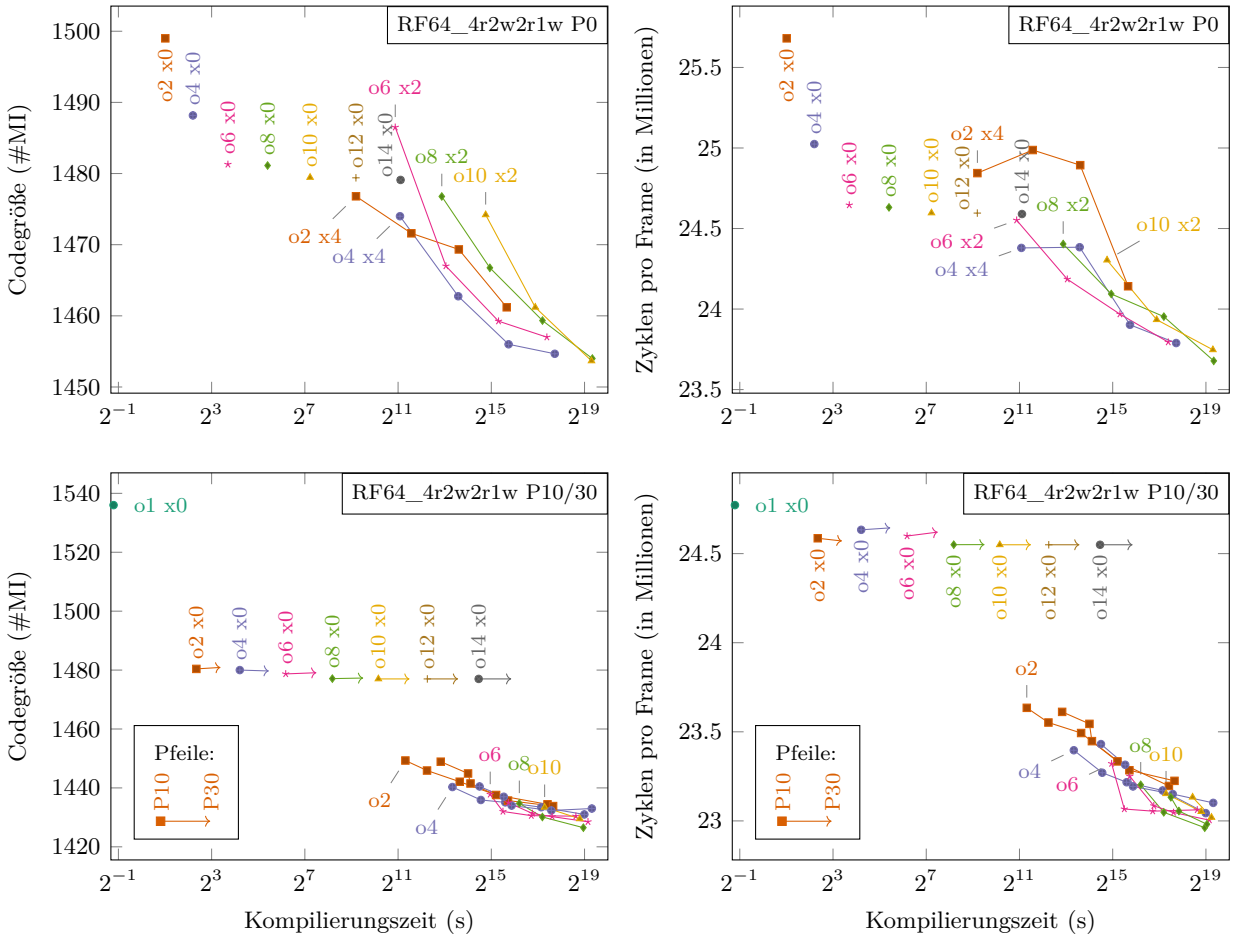


Abbildung 4.20: Evaluation der EA-basierten Codegenerierung für Registerkonfiguration RF64 und 4r2w2r1w Portkonfiguration. Durchläufe mit P10 und P30 erreichen nahezu identische Ergebnisse, nur benötigt P30 etwa doppelt so lange. Dies ist durch Pfeile dargestellt, die die Lage des P30 Datenpunktes relativ zum P10 Datenpunkt zeigen.

(P0) die EA-basierte Registerallokation (P10/30) zum Einsatz. Ergebnisse für die Konfiguration RF64 mit 4r2w2r1w Ports sind in Abbildung 4.20 dargestellt. Folgende Schlüsse können daraus gezogen werden:

- Das Instruction Scheduling mit statischer Heuristik (o1 x0) schlägt fehl, wenn die Registerallokation mit statischer Heuristik (P0) verwendet wird, da Konflikte in den Lese-/Schreibports erzeugt werden, die von der statischen Heuristik nicht behandelt werden können. Mit EA-basierter Registerallokation (P10/30) können diese Konflikte gelöst werden und das Scheduling

kann erfolgreich abgeschlossen werden. Im Vergleich zur nicht beschränkten Registerkonfiguration steigt hier die Codegröße um 7 Instruktionen und die Anzahl der Taktzyklen um etwa 150 000. Für das Instruction Scheduling mit EA-basierter Heuristik (o2 x0, o4 x0, ...) können Konflikte in den Lese-/Schreibports durch das flexible Scheduling umgangen werden. Größere Populationen sind darin erfolgreicher, was zu einer geringeren Codegröße führt. Wird die EA-basierte Registerallokation verwendet, werden die Konflikte durch die verbesserte Allokation gelöst, weshalb auch schon kleine Schedulingpopulationen den kompakten Code finden und steigende Populationsgrößen nur wenig zusätzlichen Gewinn erzielen. Im Vergleich zum Instruction Scheduling mit statischer Heuristik (o1 x0) wird die Codegröße um 3,8 % und die Zahl der Taktzyklen um 0,9 % verringert.

- Die erreichte Codegröße und die Anzahl der Taktzyklen ist für die Populationsgrößen P10 und P30 in der EA-basierten Registerallokation nahezu identisch. Allerdings benötigt die Allokation mit P30 etwa doppelt so lange für die Kompilierung wie die Allokation mit P10. Dies ist in der Abbildung durch Pfeile dargestellt, die die Verschiebung der Datenpunkte für P10 angeben.

- Bei Verwendung von Operation Merging werden keine Lösungen gefunden, solange Instruction Scheduling oder Merging mit statischer Heuristik ausgeführt werden. Die aggressive statische Heuristik für das Merging (x1) fusioniert zu viele Operationen, wodurch Konflikte in den Registerbänken entstehen, die vom Instruction Scheduling nicht mehr kompensiert werden können. Das Instruction Scheduling mit statischer Heuristik (o1) ist nicht flexibel genug, um die Konflikte in den Registerbänken zu lösen und schlägt daher für fusionierten Code fehl. Nur die Kombination von EA-basiertem Scheduling und Merging bietet genügend Flexibilität, um die Einschränkungen in den Registerbänken zu umgehen und die Kompilierung abzuschließen. Allerdings ist für kleine Populationen im Scheduling (o2, o4) die Wahrscheinlichkeit für eine erfolgreiche Kompilierung für Merginglevel x2 sehr gering. In diesem Experiment war Merginglevel x4 für diese Populationen nötig. Mit größeren Populationen (o6, ...) werden Beschränkungen der Registerallokation noch besser behandelt, so dass auch die kleinen Populationen (x2) erfolgreich sind. Die Abbildung zeigt Pareto-optimale Parameterkombinationen, bei denen Scheduling und Merging Level gleich oder ähnlich sind (o2 x2, x2 x4, o4 x2, o4 x4, o6 x4, ...)

- Die Ergebnisse für Schedulinglevel o2 mit EA-basiertem Merging und Registerallokation mit statischer Heuristik zeigen, dass eine verringerte Gesamtgröße des Programms nicht zwangsläufig zu einer verringerten Ausführungszeit führt. Dies ist damit zu begründen, dass für kleine Populations-

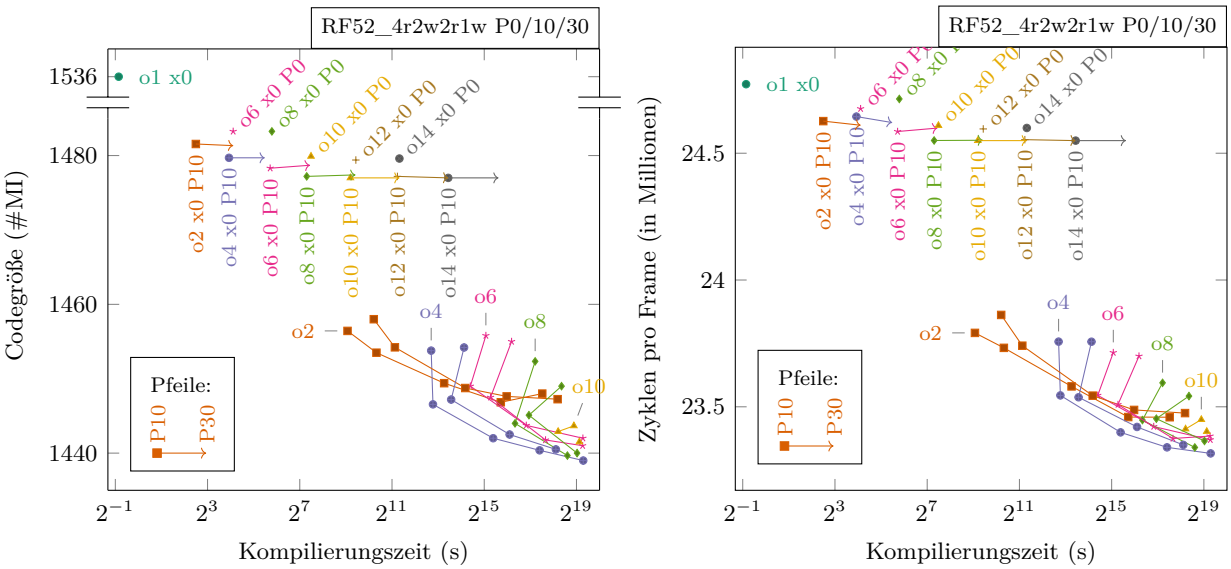


Abbildung 4.21: Evaluation der EA-basierten Codegenerierung für Registerkonfiguration RF52 und 4r2w2r1w Portkonfiguration. Durchläufe mit P10 und P30 erreichen nahezu identische Ergebnisse, nur benötigt P30 etwa doppelt so lange. Dies ist durch Pfeile dargestellt, die die Lage des P30 Datenpunktes relativ zum P10 Datenpunkt zeigen.

größen im Merging die Reduktion der Codegröße hauptsächlich in kleinen SLMs, die selten ausgeführt werden, erfolgte. Erst für größere Populationen werden die häufig ausgeführten SLMs ausreichend kompaktiert.

- Die Codegröße und die Anzahl der Taktzyklen erreichen eine Sättigung bei etwa 1431 MIs und etwa 23,28 Millionen Zyklen. Dies ist eine Reduktion um 6,8 % in Codegröße und 6,0 % in der Zahl der Taktzyklen verglichen mit dem Instruction Scheduling mit statischer Heuristik (o1 x0).

Ergebnisse für eine Registerkonfiguration mit Einschränkungen sowohl in der Anzahl der Register als auch in der Anzahl der Lese-/Schreibports in der zweiten Registerpartition sind in Abbildung 4.21 dargestellt.

- Instruction Scheduling mit statischer Heuristik (o1) und Registerallokation mit statischer Heuristik (P0) sind aufgrund der beschränkten Registerbänke nicht erfolgreich. Nur mit EA-basierter Registerallokation (P30) gelingt das Instruction Scheduling (o1 x0). Mit 1536 Instruktionen ist der Code um 7 Instruktionen größer als in unbeschränkten Registerkonfigurationen. Mit dem EA-basierten Scheduler gelingt die Codegenerierung für Populationsgrößen o6 und höher (o6 x0 P0, o8 x0 P0, ...), da das flexible

Scheduling mit den Einschränkungen in den Registerbänken umgehen kann. Die Codegröße reicht von 1483 bis 1479 und die Zahl der Taktzyklen von 24,68 Millionen bis 24,60 Millionen. Mit EA-basierter Registerallokation ist die Codegenerierung bereits für kleinere Populationsgrößen im Scheduling erfolgreich (o2 x0 P10/30, o4 x0 P10/30, ...). Eine steigende Populationsgröße führt zu sinkender Codegröße und Anzahl von Taktzyklen bis zu einer Sättigung für Populationsgröße o8 und darüber.

- Wird Operation Merging verwendet, findet das Instruction Scheduling mit statischer Heuristik (o1) keine Lösung, da die Einschränkungen in den Registerbänken zu stark sind. Nur das EA-basierte Scheduling (o2, o4, ...) ist flexibel genug, um die Einschränkungen zu umgehen. Manche Durchläufe mit Merginglevel x4 sind schneller als die entsprechenden Durchläufe mit Merginglevel x2, da aufgrund der größeren Populationen kompaktere Lösungen bereits in früheren Generationen gefunden werden und die Konvergenz beschleunigt wird. Für noch größere Populationen steigt die benötigte Rechenzeit so stark an, sodass der Vorteil der schnelleren Konvergenz nicht mehr zum Tragen kommt.

- Durch die starken Beschränkungen in den Registerbänken liegt die minimal erreichte Codegröße bei 1439 Instruktionen und die Zahl der Taktzyklen bei 23,32 Millionen, erreicht von o4 x10. Größere Populationen erreichen keine weitere Verbesserung, benötigen aber länger. Deshalb besteht die Pareto-Front hier aus den o2- und o4-Durchläufen.

4.4.6 Vergleich von automatischem und manuellem Operation Merging

Der Programmierer kann im Assemblercode direkt X2-Operationen verwenden und damit das Operation Merging manuell durchführen. Dies wurde für die Straßenschilderkennung durchgeführt. Ein Vergleich von automatischem und manuellem Operation Merging ist in Tabelle 4.4 gezeigt. Kompilierungen für Registerkonfigurationen RF64 und RF52 mit 4r2w4r2w Portkonfiguration verwenden die Registerallokation mit statischer Heuristik. Für die Registerkonfiguration mit eingeschränkter Portanzahl wird die EA-basierte Registerallokation (P10) verwendet. Angegeben ist der Zuwachs in der Codegröße bzw. Anzahl der Taktzyklen in Prozent, jeweils bezogen auf den besten Durchlauf mit EA-basiertem Scheduling und Merging in der entsprechenden Registerkonfiguration. In allen Fällen erzielt das automatische Operation Merging bessere Ergebnisse als die Compilierung des Codes nach manuellem Merging.

- Das Instruction Scheduling des manuell fusionierten Codes gelingt nur für Populationsgrößen von o4 und darüber, da durch die Fusion die Anzahl

Tabelle 4.4: Vergleich von automatischem und manuellem Operation Merging. Einträge in der Tabelle zeigen den prozentualen Unterschied der Codegröße bzw. der Anzahl der Taktzyklen bezogen auf den besten Durchlauf mit EA-basiertem Scheduling und Merging eines nicht manuell fusionierten Codes in der gleichen Registerkonfiguration. Für die ideale Registerkonfiguration kann keine Emulation durchgeführt werden, daher ist hier nur die Codegröße angegeben.

	RF64		RF52		RF64_4r2w2r1w		RF ideal	
	x0	x4	x0	x4	x0	x4	x0	x4
	MIs / Zyklen	MIs / Zyklen	MIs / Zyklen	MIs / Zyklen	MIs / Zyklen	MIs / Zyklen	MIs	MIs
o4	1.82 / 3.98	0.97 / 1.35	– / –	– / –	– / –	3.14 / 7.77	1.29	0.59
o6	1.49 / 3.32	0.65 / 0.51	1.85 / 4.26	0.92 / 0.84	– / –	2.87 / 7.07	1.18	0.43
o8	1.46 / 3.38	0.53 / 0.51	1.38 / 2.58	0.78 / 0.79	– / –	2.32 / 6.12	1.01	0.35
o10	1.42 / 2.40	0.51 / 0.54	1.15 / 1.49	0.76 / 0.88	– / –	2.02 / 5.44	0.91	0.35
o12	1.39 / 2.34	0.54 / 0.61	1.15 / 1.49	0.79 / 0.96	3.26 / 8.47	1.78 / 5.46	0.89	0.34
o14	1.15 / 1.77	0.52 / 0.57	1.15 / 1.49	*timeout*	3.19 / 8.43	*timeout*	0.86	0.30
o16	1.01 / 1.45	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*	0.86	0.29
o18	0.97 / 1.38	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*

der aktiven Variablen erhöht wird, was nur von größeren Populationen kompensiert werden kann. Für die Registerkonfiguration RF52 ist dieser Effekt noch stärker und Lösungen werden erst für o6 und größer gefunden.

- Für Registerkonfigurationen mit eingeschränkter Anzahl an Ports werden bei Verwendung der statischen Heuristik in der Registerallokation keine Lösungen gefunden. Eine Codegenerierung ist nur bei Verwendung der EA-basierten Registerallokation möglich. Die Kompilierung gelingt ohne zusätzliches Merging wegen der verschärften Beschränkungen erst für Schedulinglevel o12. Wird zusätzlich automatisches Operation Merging aktiviert (x4), findet der Algorithmus auch schon für o4 Lösungen, da das Instruction Scheduling für alle Merging-Individuen ausgeführt wird, wodurch eine deutlich größere Anzahl von Schedulingversuchen durchgeführt wird, was einer erhöhten Populationsgröße ähnelt.
- Das automatische Operation Merging kann in dem bereits fusionierten Code weitere Operationen fusionieren. Die Ergebnisse für Merginglevel x2 und x4 sind dabei nahezu identisch, da bereits kleine Populationen ausreichen, um die verbliebenen möglichen Fusionen zu finden.
- Mit steigender Populationsgröße im Instruction Scheduling sinkt auch die erreichte Codegröße bzw. Anzahl an Taktzyklen. Trotzdem sind die Ergebnisse größer bzw. langsamer als die besten Durchläufe mit EA-basiertem Scheduling und Merging des nicht manuell fusionierten Codes in der gleichen Registerkonfiguration. Dies gilt sogar, wenn zusätzliches Operation Merging erlaubt wird. Dieser Effekt kann nicht ausschließlich durch Beschränkungen in den Registerbänken erklärt werden, da er auch für eine ideale Registerkonfiguration ohne jede Einschränkung auftritt. Vielmehr verhindern einige manuell durchgeführte Fusionen die Fusion und das Scheduling anderer Operationen, die größere Auswirkungen auf Codegröße und Zyklenzahl haben.

Der Vergleich zeigt, dass in dieser Anwendung, ein manuelles Merging durch den Programmierer nicht sinnvoll ist, da dies die Möglichkeiten der EA-basierten Algorithmen beschränkt. Die Algorithmen können durch die explorative Natur der EAs bessere Lösungen aus dem Raum aller Möglichkeiten identifizieren.

4.4.7 Algorithmische Charakteristika der Verkehrszeichendetektion

Die exemplarische Verkehrszeichendetektion, die zur Evaluation des Codegenerators verwendet wird, besteht aus verschiedenen Routinen, deren genaue Funktion bzw. Arbeitsweise in Abschnitt 6.1 detailliert erläutert wird. Durch unterschiedliche algorithmische Charakteristika der Routinen kann die Anwendung als eine Menge verschiedener Benchmarks für typische DSP-Anwendungen

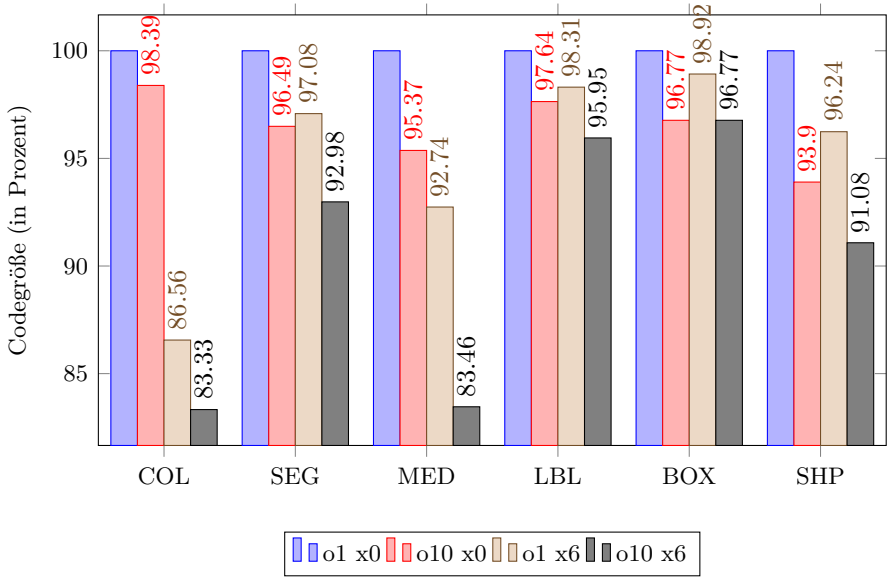


Abbildung 4.22: Vergleich der Codegröße verschiedener Routinen für verschiedene Optionen des Codegenerators für eine TUKUTURI-Konfiguration mit zwei Instanzen jeder funktionalen Einheit und einem RF64 Registerfile. Die Codegröße ist relativ zur Größe nach Scheduling mit statischer Heuristik (o1 x0) angegeben.

gesehen werden. Daher soll in diesem Abschnitt ein Überblick über die unterschiedlichen algorithmischen Charakteristika der Subroutinen gegeben werden, und der Codegenerator für die einzelnen Routinen evaluiert werden, wodurch eine Verallgemeinerung auf andere Signalverarbeitungsanwendungen ermöglicht wird. Da eine starke Korrelation zwischen der Codegröße und der Anzahl der Taktzyklen für die Anwendung besteht, beschränken sich die Analysen hier auf die Codegröße.

Abbildung 4.22 zeigt die Codegröße für verschiedene Routinen der Verkehrszeichendetektion nach Scheduling mit verschiedenen Parametern in der Codegenerierung für eine TUKUTURI-Architektur mit zwei Instanzen jeder funktionalen Einheit (mit Ausnahme der MAC) und einer Registerkonfiguration mit zwei Partitionen mit jeweils 32 Registern und 4 Lese- und 2 Schreibports (RF64).

Der größte Effekt des EA-basierten Schedulings (o10 x0) tritt in der Median-Routine (MED) und der Formklassifikation (SHP) auf. Die Berechnung des Filters in MED bzw. eines Skalarprodukts für die SVM-Klassifikation in SHP sind rechenintensive Algorithmen, die wenig Kontrollfluss enthalten und daher eine Reduktion der Codegröße um 4,6 % bzw. 6,1 % ermöglichen. Die Farbraumtransformation (COL) profitiert mit nur 1,6 % nicht so stark vom Scheduling, da hier durch die Verwendung des Divisions-Coprozessors mit einer hohen Latenz die Möglichkeiten zur Parallelisierung verringert werden. Die Labeling Routine

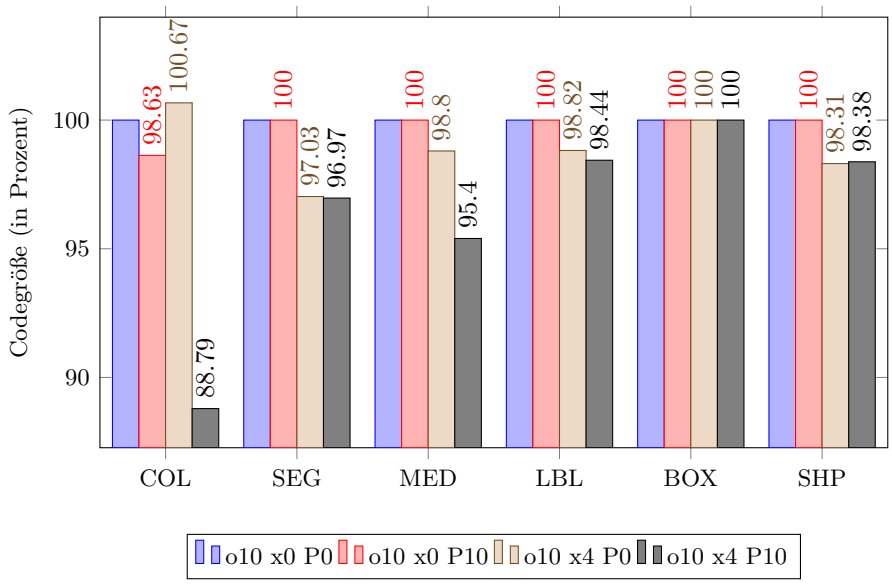


Abbildung 4.23: Vergleich der Codegröße verschiedener Routinen für verschiedene Optionen der Codegenerierung für eine TUKUTURI-Konfiguration mit zwei Instanzen jeder funktionalen Einheit und einem Registerfile mit eingeschränkter Portanzahl (RF64_4r2w2r1w). Angaben sind relativ zum EA-basierten Scheduling (o10 x0 P0).

(LBL) enthält Kontrollabhängigkeiten über die Flags und konditionale Ausführung (CS/CR, siehe Abschnitt 3.1.1) wodurch die Parallelisierung erschwert wird. Dies führt zu einer Codereduktion um 2,4 %. Die Ermittlung der umfassenden Rechtecke (bounding box, BOX) der gelabelten Einheiten ist ein speicherintensiver Algorithmus, da für jedes gelabelte Pixel ein Eintrag aus der Tabelle der Bounding Boxes aktualisiert werden muss. Diese kann aufgrund ihrer Größe nicht in Registern gehalten werden, so dass die Einträge aus dem Speicher geladen werden müssen. Die dadurch entstehenden Abhängigkeiten erschweren die Parallelisierung, so dass eine Codereduktion um 3,2 % erreicht werden kann.

Mit EA-basiertem Operation Merging (o1 x6) profitieren die Farbraumtransformation (COL) und der Medianfilter (MED) aufgrund der hohen Anzahl gleicher Operationen (hauptsächlich ALU) besonders, was zu Codereduktionen von 13,4 % bzw. 7,3 % führt. Segmentierung (SEG) und Labeling (LBL) enthalten zu viele Datenabhängigkeiten, um viele Operationen fusionieren zu können, wodurch die Codegröße nur um 2,9 % bzw. 1,7 % verringert werden kann. Die Speicherzugriffe in BOX müssen sequentiell erfolgen, so dass hier nur um 1 % in der Codegröße reduziert werden kann.

Werden EA-basiertes Scheduling und Operation Merging kombiniert (o10 x6), können COL und MED besonders kompaktiert werden (16,7 % bzw. 16,5 %), da sie eine regelmäßige Struktur aufweisen und eher rechenlastig sind als dass sie

Daten- oder Kontrollabhängigkeiten enthalten. Datenabhängigkeiten in SEG und SHP erschweren Scheduling und Merging, so dass Reduktionen von 7 % und 8,9 % erreicht werden. Die BOX Routine profitiert hauptsächlich vom Scheduling, da die Speicherzugriffe, wie erwähnt, nicht parallelisiert werden können.

Um auch die Effekte der EA-basierten Registerallokation zu untersuchen, wurde eine Prozessorkonfiguration mit eingeschränkter Zahl von Ports in den Registerbänken verwendet (RF64_4r2w2r1w). Die Ergebnisse sind in Abbildung 4.23 dargestellt. Die Codegröße ist hierbei relativ zum Scheduling mit EA-basierter Heuristik ohne Operation Merging oder EA-basierter Registerallokation (o10 x0 P0) angegeben.

Ohne die Verwendung von Operation Merging (o10 x0 P10) hat die EA-basierte Registerallokation keine Auswirkungen, da das Scheduling von den Einschränkungen in der Portanzahl in den Registerbänken nicht beeinflusst wird (siehe Abschnitt 4.4.5). Für Operation Merging mit Registerallokation mittels statischer Heuristik (o10 x4 P0), kann SEG mit einer Reduktion der Codegröße um 3 % stärker kompaktiert werden als die rechenintensiveren MED und LBL (mit Reduktionen um 1,2 %), da in SEG deutlich weniger Register verwendet werden und damit die Anzahl paralleler Zugriffe auf die Registerbänke ebenfalls geringer ist. Die BOX Routine profitiert aufgrund der Speicherzugriffe, wie oben bereits beschrieben, nicht vom Operation Merging.

Bei Verwendung der EA-basierten Heuristik in der Registerallokation nach Operation Merging (o10 x4 P10) können die rechenintensiven und repetitiven Routinen COL und MED stark kompaktiert werden (11,2 % bzw. 4,6 %), wohingegen die anderen Routinen nur wenig profitieren, da sie von der Einschränkung der Portanzahl weniger stark betroffen sind.

Zusammengefasst zeigen die einzelnen Routinen der Verkehrszeichendetektion unterschiedliche algorithmische Charakteristika, die für VLIW-Anwendungen typisch sind. Bei rechenintensiven und insbesondere bei repetitiven Strukturen (beispielsweise durch *loop unrolling*) sind EA-basiertes Instruction Scheduling und Operation Merging besonders effektiv. Daten- und Kontrollabhängigkeiten, wie auch bedingte Ausführung, z. B. in Fallunterscheidungen, können die Parallelisierung und Kompaktierung genauso beeinträchtigen wie intensive Speicherzugriffe.

5 Entwicklungsumgebung für den TUKUTURI

5.1 Überblick über die Entwicklungsumgebung

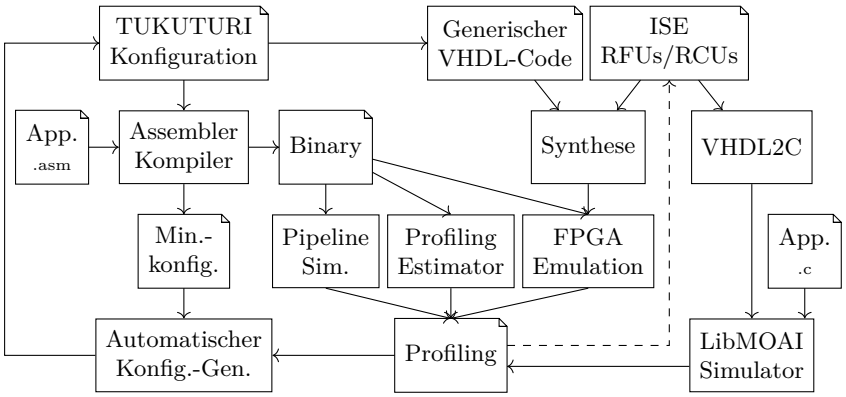


Abbildung 5.1: Überblick über die TUKUTURI-Entwicklungsumgebung

Die Entwicklungsumgebung für den TUKUTURI-ASIP folgt der Struktur des konfigurationsbasierten ASIP-Entwurfs (vgl. Abschnitt 2.1.1), wie in Abbildung 5.1 dargestellt. Ausgehend von einer Prozessorkonfiguration werden Hardware- und Software-Implementierungen generiert und evaluiert. Aus den Profilingergebnissen können Verbesserungen für die Architektur abgeleitet werden, was für eine iterative Adaption des Prozessors an die Anwendung verwendet werden kann. Dies wird zum Teil automatisch durchgeführt, muss aber an manchen Stellen manuell durchgeführt werden, wie durch den gestrichelten Pfeil dargestellt. Die folgenden Abschnitte erläutern die einzelnen Komponenten der Entwurfsumgebung eingehend.

5.2 Statische TUKUTURI-Konfiguration

Zur statischen Konfiguration des TUKUTURI-Prozessors werden die Parameter (siehe Abschnitt 3.2) in einer XML-Datei beschrieben, die von verschiedenen Teilen der Entwicklungsumgebung gelesen wird. Listing 5.1 zeigt einen exemplarischen Ausschnitt aus dieser XML-Datei. Darin werden beispielsweise zwei

```
<processor>
  <register-file>
    <num>2</num> <size>32</size>
    <w-ports>2</w-ports> <r-ports>4</r-ports>
    <special-regs>
      <sreg binary="1010100">CONDSEL</sreg>
      <sreg binary="10100XX">PERMREG</sreg>
    </special-regs>
  </register-file>

  <memory>
    <num>2</num> <r-ports>2</r-ports> <w-ports>2</w-ports>
  </memory>

  <instruction>
    <firwritelat value="3" />
    <forwarding value="0" />
  </instruction>

  <vectorUnit numIssueSlots="2" registers="0,1">
    <FU name="NOP" />
    <FU name="ALU" />
    <FU name="PER" />
    <FU name="RFU1" />
    <FU name="MV" />
    <FU name="MV" />
  </vectorUnit>

  <FU name="ALU">
    <lat value="1" />
    <size name="_8">XXXXX X001XXX XXXXXXX XXXXXX XXXXXXX</size>
    <op name="ADD"> 01000 XXXXXXX BBBBBBB CCCCCC AAAAAAA</op>
    <op name="SUB"> 01010 XXXXXXX BBBBBBB CCCCCC AAAAAAA</op>
    <suf name="_X2">XXXXX 1XXXXXX XXXXXXX XXXXXX XXXXXXX</suf>
  </FU>
</processor>
```

Listing 5.1: Exemplarischer Auszug aus der XML-Datei zur Konfiguration des TUKUTURI-Prozessors

Registerbänke mit jeweils 32 Registern und jeweils 2 Schreib- und 4 Leseports beschrieben (`<register-file>`-Block). Der Block listet außerdem die vorhandenen Spezialregister mit ihrer Binärkodierung auf. Hier ist zunächst das `CONDSEL` zu nennen, das die Bedingung für konditionale Ausführung (condition read/set) speichert. Die Kodierung für die `PERMREG`-Register enthält zwei variable Bits (`XX`), woduch bestimmt wird, dass vier verschiedene `PERMREG`-Register vorhanden sind. Ähnlich zu den Registerbänken wird im `<memory>`-Block auch die Speicherkonfiguration beschrieben. Hier werden zwei lokale Speicher mit jeweils 2 Lese- und Schreibports definiert.

Der `<instruction>`-Block enthält Angaben zu Pipelinestufen und sich daraus ergebenen Latenzen. Die Latenz beim Beschreiben eines FIR-Registers mittels `SMV` ist explizit angegeben, da sich diese von der Latenz des `SMV` beim Schreiben in andere Spezialregister unterscheidet. Im Beispiel ist außerdem angegeben, dass

keine zusätzliche Stufe für Forwarding verwendet wird.

Der `<vectorUnit>`-Block legt zum einen die Zahl der Issue-Slots und die erreichbaren Registerbänke fest, zum anderen listet er die funktionalen Einheiten auf, die in den Issue-Slots zur Verfügung stehen.

Es folgt die Beschreibung der einzelnen funktionalen Einheiten. Über die generische Beschreibung der funktionalen Einheiten in VHDL kann die Anzahl der Ausführungsstufen festgelegt werden, die hier durch die Latenz (`lat`) abgebildet ist. Zusätzlich beschreibt die Konfiguration die zur Verfügung stehenden Operationsmodi und deren Umsetzung in der Kodierung der Operation. So wird mit `size` hier die Kodierung der 8-Bit-Subwortbreite im Operationswort spezifiziert. Auf ähnliche Weise werden auch die anderen Subwortgrößen (16, 32, 64-Bit), sowie signed/unsigned oder saturation-Modi beschrieben. Es folgt die Auflistung der verschiedenen Operationen, die von dieser funktionalen Einheit angeboten werden, sowie die zugehörige Binärkodierung. Abschließend wird der X2-Modus für diese Operation definiert, der als *Suffix* bezeichnet wird (da er nach der Syntax des Assemblercodes durch Anhängen von `_X2` an die Operation ausgewählt wird). Die Binärkodierung der gesamten Operation mit allen Modi wird dann durch eine ODER-Verknüpfung der angegebenen Kodierungen erstellt, wobei die `X`-Einträge wie *don't care* behandelt werden.

Die Prozessorkonfiguration in der XML-Datei stellt eine zentrale Komponente der Entwicklungsumgebung dar, da sie den gesamten Entwurf und die Verwendung des TUKUTURI steuert. Einerseits wird sie von anderen Teilen als Eingabe verarbeitet, um die verschiedenen Komponenten an den Prozessor zu adaptieren. Andererseits wird die Konfiguration während des Entwurfs des ASIP teilweise automatisch angepasst, um den Prozessor an die Anwendung anzupassen.

5.3 Verifikation des TUKUTURI Kompilers

Der Kompiler für die TUKUTURI-Architektur wird auf verschiedene Arten verifiziert. Zum einen können vom Kompiler generierte Programme auf korrekte Funktion getestet werden, beispielsweise in einer Emulation, wie im nächsten Abschnitt beschrieben. Dazu stehen verschiedene Testprogramme zur Verfügung, die jeweils bestimmte Funktionen der Hardware und des Kompilers verwenden. Zusätzlich werden Anwendungen, wie auch die in Kapitel 6 beschriebene Verkehrszeichendetektion, für den TUKUTURI kompiliert und emuliert. Die Verifikation erfolgt dann durch Vergleich der Ausgaben dieser Programme gegen Referenzdaten.

Eine spezifischere Methode der Verifikation verwendet ein in Python geschriebenes Checker-Programm, das Assemblercode sowohl vor als auch nach der Kompilierung mit dem TUKUTURI-Kompiler lesen kann. Auf den eingelesenen Programmen wird eine Abhängigkeitsanalyse der Operationen und eine Analyse

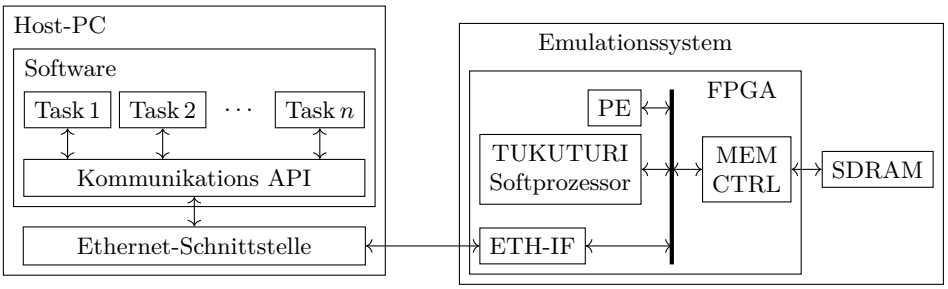


Abbildung 5.2: UEMU Emulationsframework für die Abbildung von Processing Elements (PE) und Kommunikation mit einem Host-PC [49].

des Datenflusses durchgeführt, wobei etwas andere Verfahren zum Einsatz kommen als in der C++-Implementierung des Kompilers. Ergibt der Vergleich der Abhängigkeiten und des Datenflusses eines Programms vor und nach der Kompilierung Abweichungen, so liegt ein Fehler entweder im Kompiler oder im Checker vor, der dann näher untersucht wird. Sind keine Abweichungen vorhanden, ist dies kein Beweis für die Korrektheit des Kompilers, die im Kompiler eingesetzten Methoden sind dann jedoch mit weiteren unterschiedlichen Implementierungen und Methoden getestet worden.

5.4 Synthese und Emulation des TUKUTURI

Der TUKUTURI ist als flexibles Architekturtemplate in generischem VHDL beschrieben. Nach der Festlegung aller Parameter durch die Prozessorkonfiguration (XML-Datei, siehe Abschnitt 5.2), kann diese VHDL-Beschreibung für eine Application-Specific Integrated Circuit (ASIC)- oder FPGA-Implementierung synthetisiert werden. In dieser Arbeit wird der TUKTURUI-Prozessor als Softprozessor auf einem FPGA verwendet. Dazu wird das ML605 Evaluationsboard von Xilinx verwendet, das einen Virtex-6 XC6VLX240T-1FFG1156 FPGA, eine Ethernet-Schnittstelle und DDR3-Speicher enthält [122].

Bei der Synthese wird eine spezifische Konfiguration des Prozessors in das UEMU-Framework integriert, das eine einheitliche Schnittstelle zur Simulation (z. B. in Modelsim) und Emulation (auf FPGAs) von Hardwaremodulen bereitstellt und Komponenten zur Kapselung der FPGA-Peripherie wie z. B. Speicher- und Ethernetschnittstelle enthält [49].

Mittels UEMU-Framework kann ein System aus verschiedenen *Processing Elements (PE)*, von denen eines oder mehrere Softprozessoren sein können, aufgebaut werden. Außerdem wird eine Schnittstelle zur Kommunikation eines Host-PC mit dem emulierten System über eine Ethernetverbindung bereitgestellt, wie in Abbildung 5.2 dargestellt. Darüber kann einerseits das Emulationssystem vom Host-PC gesteuert werden, andererseits erlaubt diese Kopplung auch die

```
moai_condsel = COND_Z;
moai_reg_8 reg0, reg1, reg2;
reg1 = init_8(0x54,0x12,0xb3,0xa2,
              0x26,0x77,0xf1,0x12);
reg2 = init_8(0x23,0x12,0xb3,0x32,
              0x26,0x36,0x8a,0x12);
SUB_CS(reg1, reg2);
reg0 = ADD_CR(reg1, reg2);
```

Listing 5.2: Exemplarischer Code in C mit Intrinsics

```
SMVI  CONDSEL, #COND_Z

MVIL_32 V0R1, #0xf0, #0x5412b3a2
MVIL_32 V0R1, #0x0f, #0x2677f112
MVIL_32 V0R2, #0xf0, #0x2312b332
MVIL_32 V0R2, #0x0f, #0x26368a12
SUBCS_8 V0R0, V0R1, V0R2
ADDCR_8 V0R0, V0R1, V0R2
```

Listing 5.3: Exemplarischer Code in MOAI-Assembler

Auslagerung rechenintensiver Teilaufgaben einer komplexen Anwendung von dem Host-PC auf das FPGA, wo sie durch Hardwarebeschleuniger umgesetzt werden können [30]. Dadurch werden verschiedene Arbeitsabläufe ermöglicht, wie in [30] erläutert:

- Durch die einfache Austauschbarkeit von Software- und funktionsgleichen Hardwaremodulen und den Vergleich der Ergebnisse beider Implementierungen können die Hardwaremodule verifiziert werden (*hardware-in-the-loop*).
- Durch Beschleunigung rechenintensiver Teilaufgaben einer Anwendung in dedizierten Hardwarebeschleunigern kann das Prototyping eines komplexen Systems beschleunigt und vereinfacht werden, da beispielsweise Parameterkombinationen in Algorithmen schneller evaluiert werden können.
- Mittels einer Bibliothek verschiedener Hardwareimplementierungen einer Aufgabe kann eine Entwurfsraumexploration bezüglich verschiedener Entwurfsziele wie Geschwindigkeit oder Flächenbedarf durchgeführt werden.

In der TUKUTURI-Implementierung erlaubt das UEMU-Framework dem Host-PC über eine Ethernet-Verbindung das Schreiben und Lesen von Daten in bzw. aus dem externen Speicher und weiterhin den Zugriff auf spezielle Register im Softprozessor. Dies wird genutzt, um den Prozessor mit Daten zu versorgen, die im externen Speicher abgelegt werden, und zur Synchronisierung des TUKUTURI mit dem Host-PC über Semaphoren in den Spezialregistern des TUKUTURI.

5.5 Anwendungsentwicklung

Die Entwicklung von Anwendungen für den TUKUTURI-Softprozessor erfolgt in Assemblercode, der dann vom Assembler Kompiler in ausführbare Binaries übersetzt wird. Der Assembler Kompiler, wie in Kapitel 4 beschrieben, muss die Konfiguration des TUKUTURI-Prozessors kennen, um Abhängigkeiten und Konflikte im Code zu erkennen und gültigen Binärcode zu generieren. Daher wird die XML-Datei mit der TUKUTURI-Konfiguration eingelesen, aus der

```
[ SLM 57]
  143700 | 143700 -> (0x246) STORE  0x4001, V0R3
  143700 | 143700 -> (0x247) STOREI 0x4000, #1
  143700 | 143700 -> (0x248) STOREI 0x4006, #1
[ SLM 58]
11520000 | 877312 -> (0x249) MV     V1R1, (V0FIR3)+
11520000 | 874130 -> (0x24a) MVI_8  V1R3, #0xff
11520000 | 872826 -> (0x24b) MV     V1R6  (V0FIR12)+
         ...
11520000 | 806641 -> (0x288) MV     (V0FIR5)+, V1R1
11520000 | 752131 -> (0x289) NOP
11520000 | 741178 -> (0x28a) NOP
[ SLM 59]
  144000 |      0 -> (0x28b) XORIL  V0R13, V0R13, #0xf0
  144000 |      0 -> (0x28c) ELOOPR V0R1, LBL
  144000 |      0 -> (0x28d) ADDIL  V0R3, V0R2, #0x177000
```

Listing 5.4: Exemplarischer Auszug aus einem Profiling. Die erste Spalte zeigt den run count der Instruktion, die zweite Spalte ist der DMA-Zähler.

Informationen über die verfügbaren funktionalen Einheiten, die Operationen, die Register-/Speicherkonfiguration, und die Pipeline entnommen werden.

Da der Kompiler den Anwendungscode analysiert, kann er eine dazu passende Minimalkonfiguration erzeugen. In dieser sind funktionale Einheiten und Operationsmodi deaktiviert, die im Anwendungscode nicht verwendet werden. Eine Adaption der Pipelinekonfiguration und der sich daraus ergebenden Latenzen findet an dieser Stelle nicht statt, da dies auf Basis eines Profilings der Anwendung erfolgt (siehe nächster Abschnitt).

Da die Entwicklung und das Debugging von Assemblercode aufwändig sind, wird mit der LibMOAI eine C/C++-Bibliothek bereitgestellt, die ein funktionales Modell des TUKUTURI in Software verfügbar macht [29]. Die Bibliothek bietet Datentypen für SIMD-Register, intrinsische Funktionen für die Operationen des TUKUTURI sowie ein Speichermodell inklusive simulierter DMA-Transfers zwischen externem und internem Speicher. Damit kann eine Anwendung von einem C/C++-Referenzcode schrittweise mittels Intrinsics in Assembler ähnlichen Code transformiert werden, und es können Software-Werkzeuge für das Debugging eingesetzt werden. Listing 5.2 zeigt einen exemplarischen C-Code mit Intrinsics, der zwei Register (mit 8-Bit-Subwords) initialisiert und anschließend eine Subtraktion mit *condition set* und eine Addition mit *condition read* ausführt. Der gleiche Code in MOAI-Assembler ist zum Vergleich in Listing 5.3 dargestellt.

5.6 Profiling

Eine wichtige Komponente im ASIP-Entwurfsprozess ist das Profiling der Anwendung mit der adaptierten Prozessorarchitektur. Zu diesem Zweck kann der

TUKUTURI-Prozessor mit einem Profiling-System ausgestattet werden, das während der FPGA-Emulation Profilingdaten sammelt und diese über das UEMU-Framework (siehe Abschnitt 5.4) abrufbar macht. Diese Profilingdaten umfassen Ausführungszähler für jede Operation (run counts), mit denen die Gesamtlaufzeit eines Programms und die Verteilung der Rechenzeit über die einzelnen Codeabschnitte untersucht werden kann. Des Weiteren wird zu jeder Instruktion im Instruktionsspeicher gezählt, wie oft der DMA aktiv war, während die Instruktion ausgeführt wurde. Damit können die Datentransfers zwischen dem externen und dem internen Speicher ausgewertet werden, um beispielsweise festzustellen, ob diese vollständig im Hintergrund ausgeführt werden, oder ob die Datenverarbeitung auf den Abschluss von Transfers warten musste. Der gleiche Mechanismus steht auch für den DMA-ICAP zur Verfügung, um die dynamische partielle Rekonfiguration von Hardwaremodulen zu untersuchen.

Listing 5.4 zeigt einen Ausschnitt aus einem exemplarischen Profiling. Die erste Spalte gibt die Anzahl der Ausführungen für die entsprechende Instruktion an (run count), die zweite Spalte zählt die DMA-Transfers. Nach dem Pfeil ist die Adresse und die Instruktion selbst angegeben (hier aus Platzgründen nur der erste Issue-Slot). Die Instruktionen in SLM 57 werden jeweils 143 700-mal ausgeführt. Bei jeder Ausführung war im Hintergrund ein DMA-Transfer aktiv, weshalb der zweite Zähler den gleichen Wert zeigt. SLM 58 ist ein Schleifenkörper und wird entsprechend häufiger ausgeführt. Der DMA-Transfer im Hintergrund endet, bevor die Schleife vollständig abgelaufen ist, weshalb die DMA-Zähler für diese Instruktionen geringer sind. Nach Abschluss der Schleife finden keine weiteren DMA-Transfers in SLM 59 statt, weshalb dieser Zähler dort 0 ist.

Wird das gleiche Eingabeprogramm für unterschiedliche TUKUTURI-Konfigurationen kompiliert, bleibt die Struktur des Programms, also die Abfolge der SLMs, gleich, auch wenn sich das Scheduling innerhalb der SLMs ändert. Daher können Profilingergebnisse aus einer Emulation auf anders kompilierten Code übertragen werden. Der *Profiling Estimator* überträgt dazu die run counts aus einem Profiling auf die entsprechenden SLMs in einem zweiten Code. Da die Schätzung eines Profilings deutlich schneller ausgeführt wird als eine Emulation einer Anwendung auf dem Prozessor, können unterschiedliche Konfigurationen des TUKUTURI-Prozessors mit Hilfe des Profiling Estimators deutlich schneller bewertet werden. Eine Auswertung des Schätzfehlers ist in Abschnitt 6.2.2 gegeben.

5.7 Manuelle Erweiterung des TUKUTURI

Dem ASIP-Entwurfsmodell aus Abschnitt 2.1.1 folgend, können Profilingergebnisse genutzt werden, um rechenintensive Abschnitte der Anwendung zu identifizieren und die Prozessorarchitektur, z. B. durch Instruktionssatzerweiterungen, an die

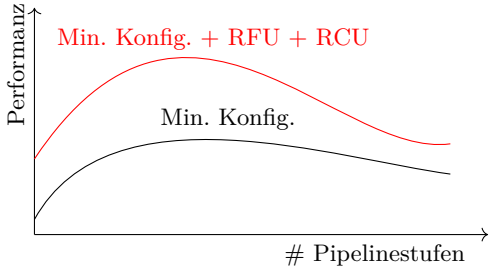


Abbildung 5.3: Qualitativer Zusammenhang zwischen der Zahl der Pipelinestufen und der Gesamtperformanz einer Implementierung

Anwendung anzupassen und damit einen Kreis zu schließen und die Architektur iterativ zu adaptieren. Für den TUKUTURI-Prozessor erfolgt dies manuell durch Beschreibung von neuen RFUs oder RCUs in VHDL. Die Eigenschaften dieser neuen Hardwaremodule werden in der Prozessorkonfiguration beschrieben, damit sie vom Kompiler berücksichtigt und in der Synthese verwendet werden können.

Zur Verifikation der Hardwaremodule wurde ein *VHDL-2-C*-Konverter auf Basis der FreeHDL-Bibliothek[1] implementiert [29], der die VHDL-Beschreibung in C-Code übersetzt, der dann mit der LibMOAI zur funktionalen Simulation, Verifikation und Debugging der Module im Softwaremodell des TUKTURI verwendet werden kann. Die Module werden dazu in Form von Funktionen bereitgestellt, die ähnlich wie die Intrinsics des Prozessors genutzt werden können, um die Funktionalität der Module in der Softwareimplementierung zu nutzen. Der Assemblercode der Anwendung muss für die Verwendung der neuen Hardwaremodule manuell angepasst werden. Dies bedeutet in den meisten Fällen das Ersetzen von Assemblercode durch den Aufruf einer RFU oder durch `STORE`- und `LOAD`-Operationen zur Ansteuerung einer RCU.

5.8 Automatische Adaption des TUKUTURI

Die Anpassung der Prozessorarchitektur erfolgt nicht nur durch Erweiterung des Befehlssatzes, sondern betrifft auch die Auswahl der Operationen und der verfügbaren Operationsmodi. Wie in Abschnitt 5.5 beschrieben, kann eine zu einer Anwendung passende Minimalkonfiguration der Prozessorarchitektur vom Kompiler generiert werden. Zusätzlich zu der Konfiguration der Operationen umfasst die statische Prozessorkonfiguration auch die Auswahl der Pipelinekonfiguration, mit der Einfluss sowohl auf die Taktfrequenz des Prozessors als

[1] `http://freehdl.seul.org/`

auch das Instruction Scheduling des Assemblercodes genommen wird (vgl. Abschnitt 3.2). Diese Parameter können automatisch durch eine automatisierte Suche im Parameterraum an eine gegebene Anwendung adaptiert werden: Ein *automatischer Konfigurationsgenerator* erzeugt verschiedene Prozessorkonfigurationen. Aus der Hardwaresynthese für die FPGA-Implementierung kann die maximale Taktfrequenz dieser Konfiguration bestimmt werden. Nach der Kompilierung der Anwendung für diese Konfiguration kann durch eine Emulation oder den oben beschriebenen *Profiling Estimator* ein Profiling ermittelt werden und die Prozessorkonfiguration z. B. hinsichtlich der erreichten Performanz bewertet werden.

Werden weitere Pipelinestufen in den Prozessor eingefügt, kann damit die maximal erreichbare Taktfrequenz erhöht werden. Gleichzeitig steigt auch die Latenz der Operationen, wodurch das Programm eine höhere Zahl von Zyklen benötigt. Solange es gelingt, die höheren Latenzen durch geeignetes Instruction Scheduling auszugleichen, kann die Gesamtperformanz der Implementierung gesteigert werden. Werden zu viele Pipelinestufen eingefügt, als dass der Kompiler ein kompaktes Scheduling finden kann, kompensiert die gesteigerte Taktfrequenz die steigende Zahl von Taktzyklen in der Anwendung nicht mehr und die Gesamtperformanz sinkt. Dieser Zusammenhang ist in Abbildung 5.3 qualitativ dargestellt.

Die Anzahl der Parameter in der TUKUTURI-Prozessorkonfiguration, die neben der Anzahl der verschiedenen funktionalen Einheiten auch die Latenzen der Einheiten sowie zusätzliche Pipelinestufen enthält (siehe Abschnitt 3.2.2), erhöht den Aufwand für eine vollständige Evaluation des Parameterraumes. Daher wurde ein Algorithmus zur multidimensionalen Optimierung eingesetzt: Zum einen soll die Taktfrequenz, die von einer Implementierung erreicht werden kann, maximiert werden. Zum anderen soll die Anzahl der Pipelinestufen gering gehalten werden, um sowohl den Ressourcenbedarf als auch die Anzahl der Ausführungszyklen in der Anwendung möglichst klein zu halten. Diese beiden Kriterien sind gegensätzlich, daher werden mehrere Pareto-optimale Konfigurationen erwartet. Für die Implementierung wird statt der Taktfrequenz die minimal mögliche Taktperiode betrachtet, da dann eine Minimierung beider Parameter erfolgen kann. Der hier zum Einsatz kommende Algorithmus ist der populäre NSGA-II (non-dominated search genetic algorithm) [23], siehe Abschnitt 2.3.2, der im Folgenden kurz erläutert wird.

Der NSGA-II Algorithmus ist eine Weiterentwicklung früherer Multi-Objective Evolutionary Algorithms (MOEAs) (speziell des NSGA [92]), der eine bessere Laufzeitkomplexität aufweist und außerdem Elitismus bietet. Die Chromosomen des Algorithmus, die neben den Problemvariablen auch die Ergebnisse einer oder mehrerer Bewertungsfunktionen enthalten, werden nach einer Dominanz-Beziehung sortiert. Ein Chromosom x_1 dominiert ein Chromosom x_2, wenn in x_2 keine Bewertungsfunktion kleiner ist als in x_1 und mindestens ein Wert einer

Bewertungsfunktion in x_2 größer ist, als in x_1 (Das Ziel ist hier die Minimierung aller Bewertungsfunktionen. Für den Fall einer Maximierung werden die Begriffe kleiner und größer entsprechend vertauscht). Die Menge der nicht-dominierten Chromosomen in einer Population bildet also die erste Pareto-Front. Der Algorithmus entfernt dann diese Individuen aus der Population und bestimmt wiederum die nicht-dominierten Individuen in der restlichen Population zur Bestimmung der nächsten nicht-dominierten Front. Dadurch wird jedem Individuum ein Level zugeordnet. Um die Individuen im Parameterraum und auf den Pareto-Fronten gleichmäßig zu verteilen, setzt der Algorithmus ein zusätzliches Distanzmaß, die *crowding distance*, ein. Für ein Individuum beschreibt die crowding distance den mittleren Abstand der benachbarten Individuen auf der gleichen nicht-dominierten Front bezüglich der verschiedenen Bewertungsfunktionen. Zusammengenommen können Individuen verglichen werden, indem zunächst das Level herangezogen wird. Individuen auf einem niedrigeren Level werden solchen auf höheren Leveln vorgezogen. Wenn Individuen das gleiche Level besitzen, also in der gleichen nicht-dominierten Front liegen, wird das Individuum mit der höheren crowding distance, also in dünner besiedelten Bereichen des Paramterraumes, bevorzugt. Dieses Sortierkriterium wird dann für die binäre Tournament Selection verwendet. Aus der Population wird durch Rekombination und Mutation eine gleich große Nachkommen-Population gebildet. Beide Populationen zusammen werden dann der nicht-dominierten Sortierung unterworfen, wodurch Elitismus automatisch sichergestellt ist. Anschließend werden aus dieser Gesamtpopulation die besten Individuen (nach Level und crowding distance) ausgewählt, um die nächste Generation zu erzeugen.

In der Implementierung für die Optimierung des TUKUTURI-Prozessors, die auf Code von A. Seshadri[2] basiert, werden die Konfigurationsparameter für die Operationen und die Pipeline in den Chromosomen gespeichert und die minimal mögliche Taktperiode nach einer FPGA-Synthese sowie die Gesamtlatenz der Konfiguration als Bewertungsfunktion verwendet. Die Gesamtlatenz ergibt sich dabei aus der Summe der Latenzen der einzelnen Operationen sowie des Forwarding-Parameters, der mit einem Gewicht von 7 eingeht, da er eine zusätzliche Pipelinestufe für alle Operationen darstellt. Des Weiteren wird eine zusätzliche Decoder-Stufe mitgezählt. Schließlich wird noch der BUSW-Parameter addiert, der eine zusätzliche Latenz für Speicheroperationen darstellt.

Um die Konvergenz des Algorithmus zu beschleunigen, wurde er durch problemspezifische Operatoren erweitert, die in jeder Generation zusätzliche Nachkommen generieren, und die nach der Evaluation bei der Erzeugung der Folgegeneration wie die anderen Nachkommen behandelt werden.

Für die Generierung zusätzlicher Nachkommen wird die Population je nach

[2] `https://de.mathworks.com/matlabcentral/fileexchange/10351-multi-objective-optimizaion-using-evolutionary-algorithm`

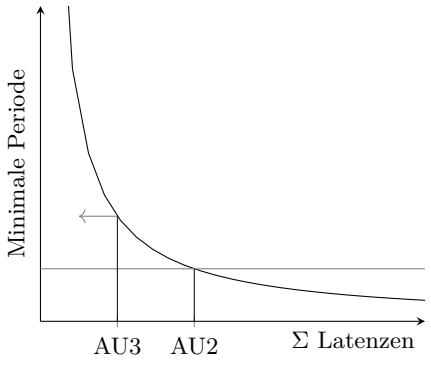


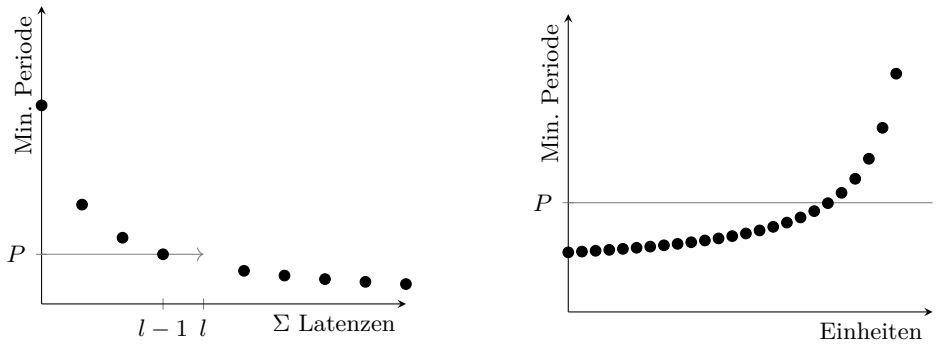
Abbildung 5.4: Reduktion der Gesamtlatenz einer Konfiguration durch Verringern der Latenz einer funktionalen Einheit. Hier ist AU3 eine AU-Einheit mit Latenz 2, AU3 ist die Einheit mit Latenz 2.

Kombination der DE- und FWD-Parameter in neun Teilpopulationen aufgeteilt (beide Parameter können jeweils drei Werte annehmen), die unabhängig voneinander verarbeitet werden.

Verringerung der Gesamtlatenz Für die erste Gruppe von Nachkommen wird versucht, die Gesamtlatenz einer Konfiguration zu verringern, ohne dabei die Periode zu beeinträchtigen. Dazu werden die Konfigurationen auf der aktuellen Pareto-Front betrachtet. Generell sollte eine höhere Latenz in einer funktionalen Einheit aufgrund der damit verbundenen zusätzlichen Pipelinestufen zu einer geringeren Periode der Konfiguration führen, wenn diese nicht durch andere Einheiten bestimmt wird. Sollte also eine Konfiguration mit einer höheren Anzahl von Pipelinestufen in einer Einheit eine höhere Periode aufweisen als eine andere Konfiguration mit geringerer Latenz in der gleichen Einheit, wie schematisch in Abbildung 5.4 gezeigt, sollte die Latenz der Einheit in der langsameren Konfiguration verringert werden können, ohne dadurch die Periode negativ zu beeinflussen. Dadurch wird die Gesamtlatenz der Konfiguration verringert, was in der Grafik einer Verschiebung der Konfiguration nach links entspricht, wodurch diese Konfiguration der wahren Pareto-Front näher rückt.

Erhöhung der Latenz Mit dem zweiten zusätzlichen Operator wird versucht, „Lücken“ in der Pareto-Front zu schließen, wie in Abbildung 5.5a dargestellt. Ist in der aktuellen Pareto-Front keine Konfiguration mit einer Latenz l enthalten[3], versucht der Operator, die Latenz einer Einheit in der Pareto-optimalen Konfiguration mit Latenz $l-1$ zu inkrementieren. Die Taktperiode P dieser

[3] es existieren Konfigurationen mit Latenz l, doch deren Periode ist zu groß, so dass sie nicht Teil der Pareto-Front sind



(a) Schließen von Lücken in der Pareto-Front durch Erhöhen der Latenz in einer Konfiguration

(b) Auswahl der funktionalen Einheit, deren Latenz erhöht wird

Abbildung 5.5: Erhöhen der Latenz in Konfigurationen zum Schließen von Lücken in der Pareto-Front.

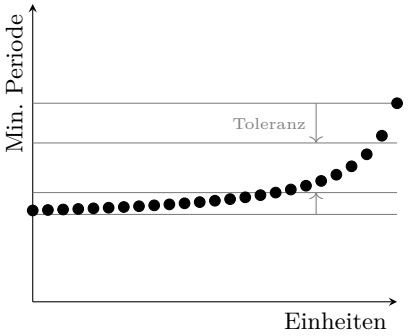


Abbildung 5.6: Verschiebung der Latenz zwischen Einheiten.

Konfiguration sollte dadurch nicht beeinträchtigt werden, da eine zusätzliche Pipelinestufe diese höchstens verringert.

Zur Auswahl der funktionalen Einheit, deren Latenz inkrementiert werden kann, ermittelt der Algorithmus die minimal erreichte Periode für alle Einheiten in allen Konfigurationen mit Latenz l, wie in Abbildung 5.5b dargestellt. Keine der Einheiten, die unterhalb der Periode P liegen, trägt zum kritischen Pfad der Konfiguration bei. Unter diesen Einheiten sucht der Algorithmus nun nach einer, deren Latenz um 1 größer ist als in der Konfiguration mit Latenz $l-1$. Die Latenz dieser Einheit wird dann in der Konfiguration mit Latenz $l-1$ erhöht, wodurch sie in Abbildung 5.5a nach rechts, aber keinesfalls nach oben verschoben wird, und damit die Lücke in der Pareto-Front bei Latenz l schließt.

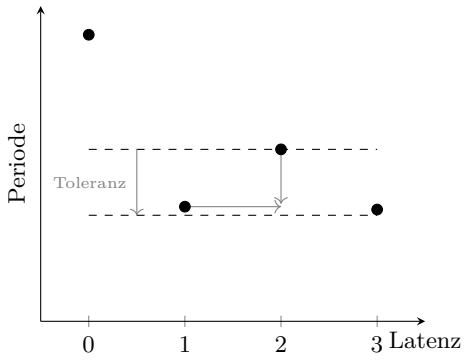


Abbildung 5.7: Inkorrekte Sortierung von Konfigurationen einer funktionalen Einheit.

Spezielle Mutation Der dritte Operator ist eine spezialisierte Mutation mit dem Ziel, aus einer Konfiguration eine neue Konfiguration abzuleiten, die die gleiche Latenz, allerdings eine geringere Periode besitzt. Der Operator ist nicht auf die Pareto-Front beschränkt, sondern betrachtet zu jeder Gesamtlatenz die Konfiguration mit minimaler Periode. Dazu wird wieder die Information über die minimal erreichbare Periode für verschiedene Latenzen der funktionalen Einheiten wie aus Abbildung 5.6 verwendet. Die Einheiten mit maximaler Periode liegen im kritischen Pfad der Prozessorkonfiguration. Kann in einer Konfiguration die Latenz einer Einheit im kritischen Pfad inkrementiert werden, sollte die dadurch entstehende Konfiguration eine verringerte Periode erreichen können. Um die Gesamtlatenz der Konfiguration unverändert zu halten, muss im Gegenzug die Latenz einer anderen Einheit verringert werden. Dazu sucht der Operator eine Einheit, die möglichst weit vom kritischen Pfad entfernt ist, und dekrementiert deren Latenz. In der Abbildung sind Schwellenwerte eingetragen, die wie folgt verwendet werden: Sollte die Latenz der Einheit mit maximaler Periode nicht weiter inkrementiert werden können, verwendet der Algorithmus die Einheit mit nächst kleinerer Periode, bis der obere Schwellenwert unterschritten wird. Sollte bis dahin keine Einheit verwendbar sein, generiert der Operator für diese Konfigurationen keine Individuen. Der untere Schwellenwert gibt auf ähnliche Weise vor, welche Einheiten der Algorithmus untersuchen darf, um deren Latenz zu verringern. Beide Schwellenwerte werden bei Verwendung des Operators als Prozentwerte der maximalen bzw. minimalen Periode als Parameter übergeben.

Detektion des kritischen Pfads Für die eben genannten Operatoren, die auf die Informationen zur minimalen Periode für verschiedene Latenzen der funktionalen Einheiten zurückgreifen, ist die Korrektheit dieser Informationen entscheidend. Wird eine bestimmte funktionale Einheit X betrachtet, dann sollte die minimale Periode für die Konfigurationen der Einheit X mit höherer Latenz kleiner sein

als die Periode für Konfigurationen mit geringerer Latenz. Sollte allerdings der umgekehrte Fall eintreten, wie in Abbildung 5.7 für $l = 2$ gezeigt, dann sollten in den entsprechenden Prozessorkonfigurationen andere Einheiten im kritischen Pfad liegen und der Algorithmus sollte andere Konfigurationen analysieren. Daher wird in der ersten Variante des Operators versucht, die Latenzen der Einheiten im kritischen Pfad zu inkrementieren, damit die Konfigurationen mit Latenz l in Einheit X geringere Perioden erreichen können. In der zweiten Variante des Operators wird in die schnelleren Konfigurationen, hier also Latenz $l' = 1$ für Einheit X die höhere Latenz l eingesetzt, da dadurch eine Konfiguration entstehen sollte, die die kleinere Periode besitzt. Auch für diesen Operator wird eine Toleranz verwendet, so dass der Operator nur arbeitet, wenn der Unterschied in den Perioden für verschiedene Konfigurationen derselben Einheit groß genug ist.

Diese Informationen über den kritischen Pfad der Prozessorkonfigurationen in Abhängigkeit von den funktionalen Einheiten wird auch als Abbruchkriterium für den evolutionären Algorithmus verwendet. Der Algorithmus zählt, wie häufig eine funktionale Einheit in einer Konfiguration mit höherer Latenz langsamer ist als in Konfigurationen mit geringerer Latenz. Dabei wird ein Toleranzparamter verwendet, so dass nur Fälle berücksichtigt werden, in denen der Unterschied in der Taktperiode ausreichend groß ist. Diese Zahlen werden über alle funktionalen Einheiten aufsummiert und der Algorithmus endet, wenn die Zahl 0 erreicht.

6 Fallstudie Verkehrszeichendetektion

In diesem Kapitel wird die Entwicklungsumgebung und die TUKUTURI-Prozessorarchitektur anhand einer exemplarischen Verkehrszeichendetektion evaluiert und mit Implementierungen auf einem kommerziellen DSP von Texas Instruments sowie einem Softcore MIPS Prozessor verglichen.

6.1 Der Verkehrszeichendetektionsalgorithmus

Algorithmen zur Erkennung von Objekten in Bildern lassen sich häufig in zwei Teilaufgaben unterteilen:

1. **Detektion** von Objekten, d. h., die Identifikation von Bildbereichen, die möglicherweise ein Objekt enthalten, und die

2. **Klassifikation** von Bildbereichen, die im ersten Schritt gefunden wurden, um das darin enthaltene Objekt zu identifizieren. Der Algorithmus sollte auch erkennen, wenn der Bildbereich im ersten Schritt fälschlicherweise identifiziert wurde und tatsächlich kein Objekt enthält.

Der in dieser Arbeit eingesetzte Algorithmus zur Detektion von Verkehrszeichen in Kamerabildern ist in Abbildung 6.1 schematisch gezeigt. Er folgt der Methode aus [61] und verwendet eine farbbasierte Segmentierung und eine formbasierte Klassifikation. Der Algorithmus führt zunächst eine Farbraumtransformation durch, um gezielt nach Farben zu segmentieren, die in Verkehrsschildern eingesetzt werden. Ein Medianfilter entfernt kleine Segmente und schließt gegebenenfalls kleine Lücken in größeren Segmenten. Zusammenhängende Pixel werden mittels eines Connected Component Labeling zu Blobs zusammengefasst und die umschließenden Rechtecke (Bounding Boxes) ermittelt. Anschließend erfolgt die Formklassifikation zur Identifikation von geometrischen Formen der Kandidaten Blobs. Der Algorithmus kann im Anschluss die identifizierten Blobs analysieren, um die abgebildeten Verkehrszeichen zu bestimmen. Diese Identifikation ist in dieser Arbeit jedoch nicht auf dem TUKTURI umgesetzt worden. Die folgenden Abschnitte geben eine detailliertere Beschreibung der einzelnen Routinen des Algorithmus.

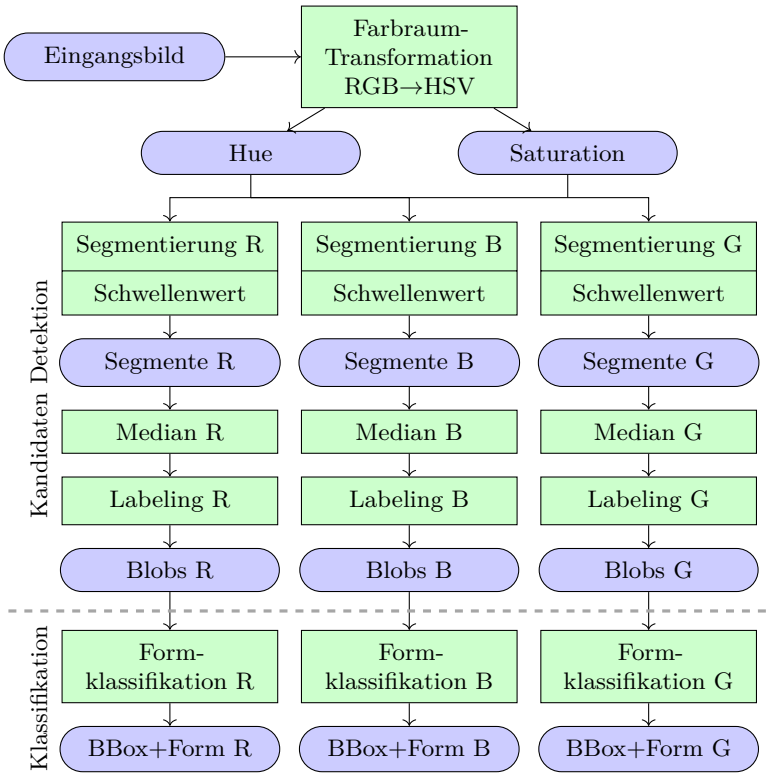


Abbildung 6.1: Überblick über den Algorithmus zur Detektion von Verkehrszeichen in Farbbildern nach [61]

6.1.1 Farbraumtransformation

Die Eingabe für den Algorithmus ist in dieser Implementierung ein Farbbild im RGB-Farbraum mit 24 Bit pro Pixel (also 8 Bit pro Farbkanal). Für die Suche nach bestimmten Farben in der Segmentierung wird das Bild zunächst in den HSV-Farbraum konvertiert, in dem eine Farbe durch die drei Koordinaten Farbton (*Hue*), Sättigung (*Saturation*) und Helligkeit (*Value*) beschrieben wird. Die Helligkeitsinformation wird von den nachfolgenden Stufen nicht verwendet, da sie zahlreichen Störungen wie unterschiedlichen Beleuchtungen (auch nach Tageszeit) oder Schattenwurf unterliegt. Durch Verzicht auf diese Komponente kann die Segmentierung weitestgehend unabhängig von diesen Einflüssen arbeiten. Zusätzlich wird die Berechnung der Helligkeitskomponente eingespart.

Die Transformation basiert auf Formeln aus [33] und wurde hier für den 8-Bit-Wertebereich der Pixel angepasst. Der Algorithmus ist in Listing 6.1 gezeigt. Es werden zunächst die größte und kleinste R-, G- und B-Komponente der Farbe und deren Differenzen bestimmt. Daraus kann dann der Farbwinkel (Hue, h) und die Sättigung (s) berechnet werden, wobei eine Division angewendet werden

```
function color_transform(r, g, b)
  M := max(r, g, b); m := min(r, g, b); d := M - m

  s := 255 * d / M

  if r = M
    h := 42.5 * (g-b) / d
  else if g = M
    h := 85 + 42.5 * (b-r) / d
  else
    h := 170 + 42.5 * (r-g) / d
  if h < 0: h := h + 255

  return (h, s)
```

Listing 6.1: Farbraumkonvertierung von RGB nach HSV

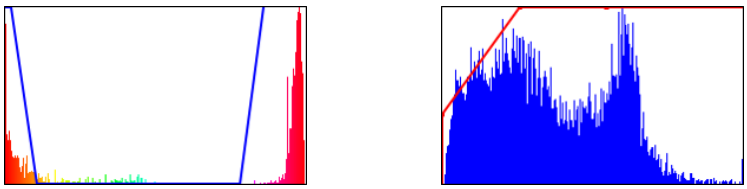


Abbildung 6.2: Histogramm der Hue- (links) und Saturation- (rechts) Werte von Pixeln, die in Bildern mit Verkehrszeichen manuell markiert wurden. Eingezeichnet ist außerdem eine daraus abgeleitete Transferfunktion als stückweise lineare Funktion.

muss, die nicht im Instruktionssatz des TUKUTURI enthalten ist. Anstelle eines iterativen Algorithmus wurde ein Coprozessor für die Division bereitgestellt, wie in Abschnitt 6.2.1 beschrieben.

6.1.2 Farbsegmentierung

Die Segmentierung des Bildes zur Detektion von Verkehrszeichen erfolgt durch Auswertung der Farbinformationen, indem Pixel mit den Farben Rot, Blau und Gelb hervorgehoben und andere Farben unterdrückt werden. Dies geschieht getrennt für jede Farbe, wie in Abbildung 6.1 dargestellt.

Für jede Farbe werden jeweils zwei Transferfunktionen verwendet, mit denen die H- und S-Komponente transformiert werden. Diese Transferfunktionen wurden aus Histogrammen abgeleitet, die ihrerseits aus Bildern generiert wurden, in denen Pixel mit den gesuchten Farben in Verkehrsschildern manuell markiert wurden. Ein Beispiel für Histogramme mit abgeleiteter Transferfunktion für rote Pixel ist in Abbildung 6.2 gezeigt. Zur vereinfachten Auswertung der Transferfunktionen sind diese durch stückweise lineare Funktionen angenähert, wie schematisch in Abbildung 6.3 gezeigt. Die zwei Formen für die Hue-Komponente sind damit begründet, dass die rote Farbe an den beiden Enden des Wertebereichs für den Farbwinkel (hier $[0, 255]$) liegt. Für die Hue-Komponente ist die Kurve durch vier

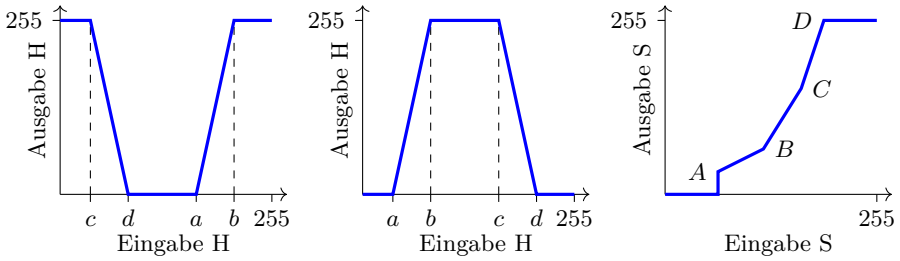
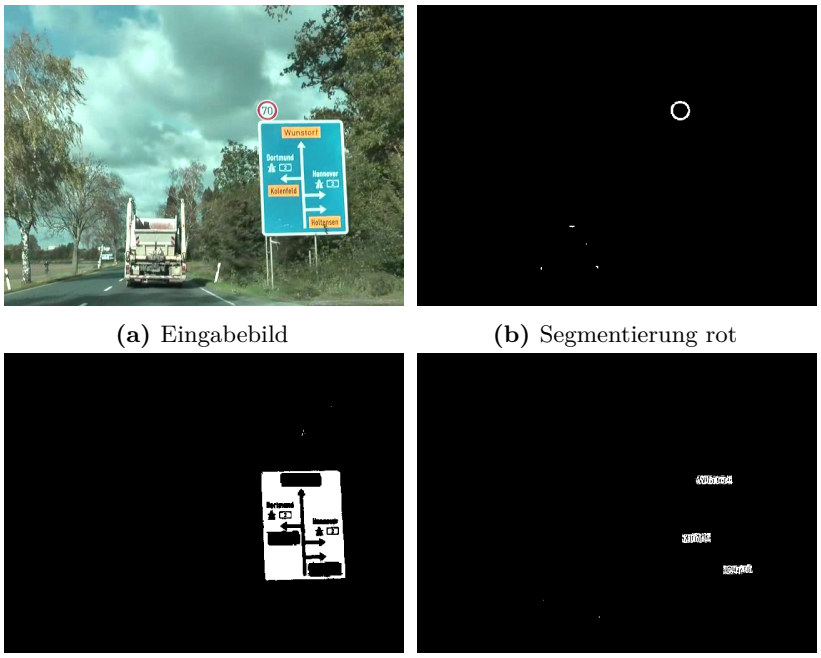


Abbildung 6.3: Schematische Darstellung der verwendeten Transferfunktionen für Hue und Saturation. Jede Funktion ist aus fünf linearen Abschnitten zusammengesetzt.



(a) Eingabebild **(b)** Segmentierung rot

(c) Segmentierung blau **(d)** Segmentierung gelb

Abbildung 6.4: Farbsegmentierung eines exemplarischen Eingangsbildes

Parameter beschrieben: Bei a steigt sie vom Wert 0 aus an, bis sie bei b den Wert 255 erreicht. Diesen hält sie, bis sie bei c sinkt und schließlich ab d wieder den Wert 0 annimmt. Die Kurve für die Saturation ist anfangs 0, und interpoliert dann die Punkte A bis D. Der letzte Punkt liegt immer bei einem Ausgabewert von 255, der nach D konstant gehalten wird. Nach der separaten Transformation der Hue- und Saturation-Komponente werden die beiden Ausgabewerte miteinander multipliziert. Ein Schwellenwert (je einer pro Farbe) wird verwendet, um die Bilder zu binarisieren. Segmentierungsergebnisse für ein Beispielbild sind in Abbildung 6.4 dargestellt.

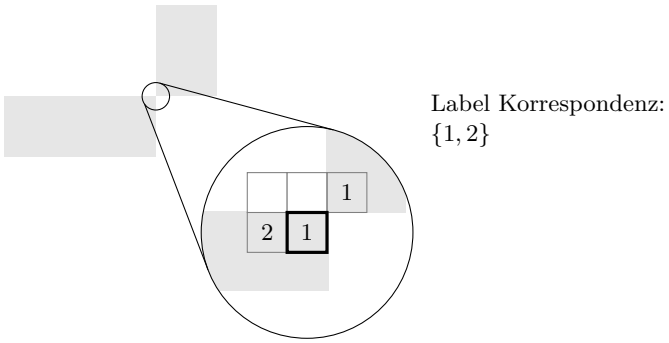


Abbildung 6.5: Erster Durchlauf des Connected Component Labeling für zwei exemplarische Blobs (graue Rechtecke), die durch ein Pixel diagonal verbunden sind. Der Algorithmus muss hier die Labels 1 und 2 vereinigen.

6.1.3 Medianfilter

Zur Filterung der Pixel nach der Segmentierung wird ein Medianfilter mit einem 5×5-Filterkern verwendet. Die Berechnung des Median-Filters für Graustufenbilder erfordert die Sortierung der Pixel nach Helligkeitswerten. Da die Segmentierung hier jedoch Binärbilder liefert, kann der Medianfilter für ein Pixel stattdessen berechnet werden, indem die Nachbarpixel unter der Filtermaske akkumuliert werden. Sind 13 oder mehr Nachbarpixel gesetzt, muss auch das Ausgabepixel gesetzt sein.

6.1.4 Connected Component Labeling

Das Connected Component Labeling ordnet jedem gesetzten Pixel im segmentierten Bild ein Label zu, sodass in einem zusammenhängenden Bereich von Pixeln, der auch *Blob* genannt wird, jedes Pixel das gleiche Label trägt und nicht verbundene Blobs unterschiedliche Label tragen. Der hier verwendete Algorithmus basiert auf einer Beschreibung in [88] und ist schematisch in Abbildung 6.5 dargestellt. Im ersten Schritt wird das Bild zeilenweise durchlaufen und Pixel anhand der Label ihrer Nachbarpixel verarbeitet. Da das Bild zeilenweise von oben nach unten und von links nach rechts durchlaufen wird, können nur die drei Pixel aus der vorherigen Bildzeile, sowie das linke Nachbarpixel in der aktuellen Bildzeile bereits mit einem Label versehen sein, wie in der Abbildung gezeigt. Trägt noch keines der Nachbarpixel ein Label, weist der Algorithmus dem aktuellen Pixel ein noch nicht vergebenes Label zu. Sind mehrere Nachbarpixel mit Labeln versehen, gehören sie alle zum gleichen Blob wie das aktuelle Pixel. Sollten die Label der Nachbarpixel unterschiedlich sein, muss der Algorithmus daher diese Label vereinigen. Diese Situation ist in Abbildung 6.5 exemplarisch für zwei Segmente (als graue Rechtecke dargestellt) gezeigt, die über ein Pixel diagonal verbunden

sind: Das rechte obere Segment wurde als erstes identifiziert und mit dem Label 1 versehen. Das linke untere Segment wurde später entdeckt. Da die linke obere Ecke dieses Segments weit von dem anderen Segment entfernt ist, kann der Algorithmus den Zusammenhang dieser Segmente nicht erkennen und weist den Pixeln dieses unteren Segments daher das Label 2 zu. Die Detailansicht zeigt das Pixel, das die beiden Segmente verbindet. Es sind bereits zwei Nachbarpixel mit unterschiedlichen Labels vorhanden. Daher wählt der Algorithmus das kleinere der beiden Labels für das aktuelle Pixel und notiert die Korrespondenz der beiden Labels. In der Beschreibung in [88] wird zur Verwaltung der Korrespondenzen eine *union-find* Datenstruktur verwendet, die eine Menge disjunkter Mengen (*disjoint set*) verwaltet. In einem zweiten Durchlauf werden dann die Labels gemäß der gespeicherten Korrespondenzen vereinheitlicht, so dass Segmente, deren Zusammenhang erst im Verlauf des ersten Durchlaufs erkannt wurde, ein einheitliches Label tragen.

Für die TUKUTURI-Implementierung wurde der Algorithmus zu Gunsten einer verringerten Komplexität modifiziert. Im ersten Durchlauf durch das Bild wird die Nachbarschaft des aktuellen Pixels, die in Abbildung 6.5 dargestellt ist, in der vorherigen Zeile um ein Pixel nach rechts erweitert (dies ist bei der Beschreibung der Hardwareeinheit in Abbildung 6.19 dargestellt). Dadurch können kleinere Lücken zwischen eng beieinander liegenden Blobs überbrückt werden. Weiterhin wird hier auf die Verwaltung einer union-find-Struktur verzichtet. Stattdessen wird der zweite Durchlauf in umgekehrter Richtung (also von rechts unten nach links oben) wiederum zeilenweise durchgeführt. Dabei wird für jedes Pixel das kleinste Label aus seiner 8er-Nachbarschaft gewählt. Dies löst für viele auftretende Blobs das Problem der Vereinigung von Labels ohne den Aufwand durch die Verwaltung der Labelkorrespondenzen in einer zusätzlichen Datenstruktur. Es können mit diesem vereinfachten Verfahren nicht alle Korrespondenzen aufgelöst werden, d. h. manche Segmente zerfallen inkorrekt in Bereiche mit unterschiedlichen Labels, die auftretenden Fehler sind jedoch für die weitere Verarbeitung der gefundenen Blobs vernachlässigbar. Der im segmentierten Beispielbild für rote Pixel gefundene und korrekt mit einem einheitlichen Label versehene Blob ist in Abbildung 6.6 eingezeichnet. Hier ist außerdem zu erkennen, dass die kleinen Segmente im unteren linken Bereich des Bildes durch die Medianfilterung entfernt wurden.

6.1.5 Formklassifikation

Nachdem Pixel zu Blobs zusammengefasst wurden, kann nun die geometrische Form analysiert werden, um mögliche Kandidaten für Straßenschilder zu identifizieren. In [61] werden dazu lineare Support Vector Machines (SVMs) eingesetzt, die auf den sogenannten *Distance-to-Border (DtB)* Merkmalen arbeiten. Dazu wird in einem ersten Schritt für jeden Blob das umfassende Rechteck (*bounding*

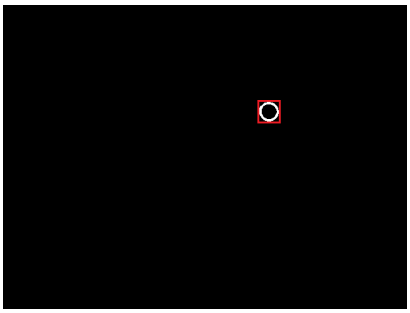


Abbildung 6.6: Blobs für rote Pixel im Beispielbild

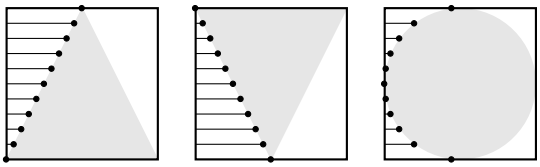


Abbildung 6.7: Darstellung der Distance-to-Border-Vektoren von der linken Kante des umschließenden Rechtecks bis zum Blob für verschiedene Formen

box) ermittelt. Anhand der Größe und des Seitenverhältnisses der Rechtecke können Blobs gefiltert werden, die keine Straßenschilder enthalten können. Im nächsten Schritt gibt dann ein DtB-Vektor den Abstand des Blobs von einer Kante des umschließenden Rechtecks an mehreren Stellen an, wie in Abbildung 6.7 exemplarisch für verschiedene Formen gezeigt. Der so erhaltene Vektor wird auf die Gesamtbreite des umschließenden Rechtecks normiert, um das Merkmal unabhängig von der Skalierung des Blobs zu machen. In der Beschreibung in [61] werden DtB-Vektoren aus den vier verschiedenen Richtungen (von links, rechts, oben und unten) mit jeweils 16 Messpunkten verwendet. In der TUKUTURI-Implementierung wird nur der DtB-Vektor von der linken Kante verwendet, da er genügend Informationen zur Identifikation der Form bereitstellt, selbst wenn die Straßenschilder im Kamerabild leicht rotiert oder perspektivisch verzerrt sind.

Die Formklassifikation erfolgt dann im letzten Schritt über vier lineare binäre SVMs, die jeweils über eine Form (aufwärts gerichtetes Dreieck, abwärts gerichtetes Dreieck, Kreis, Raute) entscheiden. Diese wurden mit Hilfe von Daten aus dem GTSRB-Datensatz [93] trainiert. Achteckige Stopp-Schilder werden von dieser Methode als Kreise identifiziert, da die Form aufgrund der geringen Skalierung der Schilder in den Eingangsbildern nicht eindeutig von einem Kreis getrennt werden kann. Für das Ausfiltern von falsch positiven Detektionen ist dies ausreichend. Zur Auswertung einer linearen SVM wird das Skalarprodukt des DtB-Merkmalsvektors mit dem Supportvektor der SVM berechnet und ein Schwellenwert addiert.

6.2 TUKUTURI-Implementierung mit Basisinstruktionssatz

6.2.1 Assemblerimplementierung

Ausgehend von einer Referenzimplementierung in C++ mit OpenCV[1] wurde die Assemblerimplementierung für den TUKUTURI entwickelt. Die einzelnen Routinen des Algorithmus werden auf dem TUKUTURI sequenziell abgearbeitet. Grundsätzlich sind die Eingangsdaten einer Stufe im externen SRAM abgelegt. Sie werden mittels DMA-Transfers in kleineren Blöcken in den lokalen Speicher des TUKUTURI übertragen. Von dort aus erfolgt die Verarbeitung der Daten und das Schreiben der Ergebnisdaten ebenfalls in den lokalen Speicher, von wo sie wiederum per DMA-Transfer in den externen SRAM abgelegt werden. Da das DMA-Modul Daten unabhängig vom TUKUTURI-Prozessor übertragen kann und eine Warteschlange von Transfers verwaltet (siehe Abschnitt 3.1.2), kann ein double-buffering-Schema eingesetzt werden, um Datenblöcke im Hintergrund parallel zur Datenverarbeitung durch den TUKUTURI zu übertragen und so Wartezeiten durch Transfers größtenteils zu verstecken.

Sowohl in der Farbraumtransformation als auch bei der Berechnung der DtB Merkmale wird eine Division benötigt, die im Standardinstruktionssatz des TUKUTURI nicht enthalten ist. Da eine reine Assemblerimplementierung eines (iterativen) Divisionsverfahrens zu ineffizient wäre, wurde der TUKUTURI mit einem Coprozessor erweitert, der Divisionen mit zwei unterschiedlichen Fixpunktformaten (Q7.9 signed, Q5.11 unsigned) auf vier 16-Bit SIMD-Subworten gleichzeitig durchführt. Die Farbraumtransformation gibt die Hue- und Saturation-Komponenten jeweils als 8-Bit Werte aus, so dass 8 Pixel in einem 64-Bit Register abgelegt werden können. Diese Fixpunkt-Formate werden auch für die Auswertung der SVMs zur Formklassifikation verwendet.

Der Coprozessor basiert auf einer *non-restoring Division*, einem iterativen Verfahren. Die Zahl der Iterationen pro Zyklus ist konfigurierbar, wodurch Auswirkungen auf die maximal erreichbare Taktfrequenz und die Latenz des Coprozessors entstehen. Durch internes Pipelining ist die Überlappung mehrerer Berechnungen möglich. Dann müssen die `STORE`- und `LOAD`-Operationen zur Ansteuerung durch geeignetes Instruction Scheduling einen bestimmten Abstand einhalten. Dazu wurden im Kompiler spezielle Pseudo-Instruktionen bereitgestellt (siehe Abschnitt 4.2.1).

Die Berechnung der **Farbraumtransformation** folgt der Beschreibung in Abschnitt 6.1.1 und beinhaltet die Berechnung des Minimums und Maximums der drei Farbkanäle (R, G, B), die Division der Werte zur Berechnung des Farbwinkels und der Saturierung. Die Fallunterscheidung in der Berechnung

[1] `https://opencv.org`

der Hue-Komponente wird mit konditionaler Ausführung (CS/CR, siehe Abschnitt 3.1.1) umgesetzt.

Zur **Segmentierung** nach Farben werden die Pixelwerte mit den Steigungen der linearen Teilstücke der Transferfunktionen multipliziert, wobei eine shift-and-add-Technik zum Einsatz kommt, um hier mit 8-Bit Ganzzahlen rechnen zu können. Mittels konditionaler Ausführung (Abschnitt 3.1.1) wird dann der jeweilige Abschnitt der Transferfunktion ausgewählt. Die Multiplikation der transformierten Hue und Saturation liefert ein 16-Bit Ergebnis, das durch den Schwellenwert sofort wieder auf 8 Bit reduziert wird. Nach der Segmentierung liegen drei Bilder, jeweils eines für rote, blaue und gelbe Pixel vor, die in den folgenden Schritten sequenziell verarbeitet werden.

Der **Median-Filter** arbeitet ebenfalls auf 8-Bit Subworten, in denen jeweils ein Pixel gespeichert ist, da eine kompaktere Kodierung der Binärbilder die Ausnutzung der Datenparallelität mit SIMD-Operationen bei der Akkumulation der Nachbarpixel erschweren würde, und die Bilder im nächsten Schritt wieder auf 8 Bit pro Pixel aufgeweitet werden. Die Bilder werden hier in Spalten verarbeitet und die Filtermaske, die durch mehrere Register gebildet wird, wird in der Art eines Schieberegisters gefüllt, so dass in jedem Durchlauf der Zeilenschleife Ergebnisse für acht nebeneinander liegende Pixel produziert werden.

Das **Connected Component Labeling** führt, wie in Abschnitt 6.1.4 bereits beschrieben, zwei Durchläufe durch das Bild aus und ordnet jedem Pixel ein 8-Bit Label zu. Dazu werden Minima von bereits vergebenen Labels berechnet und durch eine einfache Fallunterscheidung (mittels konditionaler Ausführung) neue Label vergeben, falls in der Nachbarschaft des aktuellen Pixels noch keine Labels vergeben wurden. Das Setzen der Bedingung für diese Fallunterscheidung (also die CS-Operation) geschieht dabei in der ohnehin verwendeten Operation zur Ermittlung der Labels der Nachbarpixel und erfordert somit keine zusätzliche Operation. Das Setzen eines neuen Labels, falls in der Nachbarschaft kein Label gefunden wurde (also die CR-Operation), erfordert nur eine zusätzliche `MOVE`-Operation. Nach der Median-Filterung ist die Anzahl von 255 möglichen Labels pro Farbe ausreichend, um alle Blobs voneinander zu unterscheiden.

Bis zu dieser Stelle ist die Komplexität der algorithmischen Schritte unabhängig vom Bildinhalt, da sie die Pixel im jeweiligen Eingabebild unabhängig voneinander verarbeiten. Erst in den folgenden Schritten, in denen die Verarbeitung auf Ebene der Blobs erfolgt, hängt die Anzahl der Verarbeitungsschritte von der Anzahl der gefundenen Blobs ab.

Um die **umschließenden Rechtecke** für die Blobs zu ermitteln, wird eine Tabelle mit 255 Einträgen angelegt (ein Eintrag für jedes mögliche Label). Das Rechteck wird durch die linke untere und die rechte obere Ecke beschrieben, wobei jede Koordinate in 16 Bit abgelegt wird, so dass ein Rechteck vollständig in einem 64-Bit Datenwort bzw. Register gehalten werden kann. Da Register nicht in einer derart großen Zahl vorhanden sind, wird die Tabelle im internen Speicher

Tabelle 6.1: Varianten des TUKUTURI Prozessors

Name	Beschreibung
1xFU	Jede funktionale Einheit ist genau einmal vorhanden
2xFU	Funktionale Einheiten mit Ausnahme der MMU[1] können verdoppelt werden; X2-Modus nicht aktiv
2xFU.X2	Funktionale Einheiten mit Ausnahme der MMU[1] können verdoppelt werden; X2-Modus aktiv

[1] MMU: Multiply and Accumulate Unit

abgelegt. Für jedes Pixel wird der entsprechende Tabelleneintrag geladen (`LOAD`), mit den gefundenen Pixelkoordinaten aktualisiert und wieder in den lokalen Speicher zurückgeschrieben (`STORE`). In einem Durchlauf durch das Bild werden für jedes gelabelte Pixel die Koordinaten des Rechtecks in dem entsprechenden Tabelleneintrag aktualisiert. Anschließend wird der DtB-Vektor von der linken Kante des Rechtecks zum Blob bestimmt und durch die Breite des Rechtecks dividiert. Dann erfolgt die **Formklassifikation** durch Skalarmultiplikation mit dem Supportvektor der jeweiligen SVM und Addition des Threshold, wonach das Vorzeichen des Ergebnisses die Entscheidung der SVM angibt.

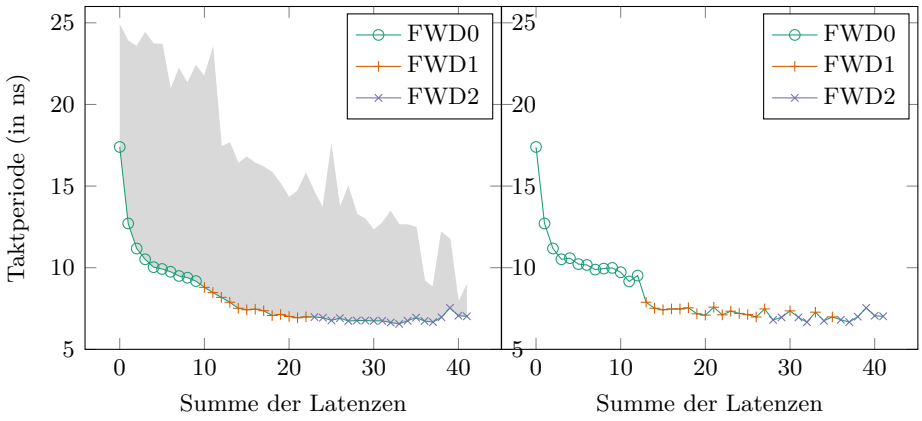
Für die Implementierung mit X2 Operation Merging wurden die Operationen zum Zugriff auf dem lokalen Speicher soweit wie möglich durch FIR-Register realisiert, für die der Kompiler eine automatische Fusionierung durchführen kann (vgl. Abschnitt 3.1.3 bzw. Abschnitt 4.2.2). Die Verwendung konditionaler Ausführung (Condition-Set bzw. Condition-Read Modi, vgl. Abschnitt 3.1.1) wird nicht automatisch vom Kompiler fusioniert und wurde daher manuell umgesetzt.

6.2.2 Vorgehen und Darstellung der Ergebnisse

In diesem Abschnitt wird das Vorgehen beim Profiling der Anwendung auf dem TUKUTURI-Softprozessor sowie die Darstellung der Ergebnisse erläutert. Die Erklärungen gelten dabei nicht nur für die Implementierung mit dem Basisinstruktionssatz, sondern ebenfalls für die Implementierungen mit erweitertem Befehlssatz und bei Verwendung von dynamischer Rekonfiguration.

Hardwaresynthesen

Für die im Folgenden dargestellte Evaluation wurden verschiedene Konfigurationen der TUKUTURI-Architektur mithilfe des in Abschnitt 5.8 beschriebenen evolutionären Algorithmus erstellt und für ein Virtex-6 FPGA synthetisiert. Grundsätzlich wurden drei verschiedene Varianten der Architektur untersucht, die in Tabelle 6.1 aufgeführt sind. Die Variante 1xFU enthält jede funktionale



(a) Konfigurationen mit minimaler Periode **(b)** Konfigurationen mit maximalen FPS

Abbildung 6.8: Minimal erreichte Taktperiode für die Variante 1xFU. Der linke Graph zeigt die minimal erreichte Taktperiode abhängig von der Anzahl der eingefügten Pipelinestufen. Der graue Bereich markiert den Wertebereich der Taktperiode für die einzelnen Konfigurationen mit entsprechender Summe der Latenzen (siehe Erklärung im Text). Der rechte Graph zeigt die Taktperiode der Konfigurationen, die maximale FPS erreichen.

Einheit einmal. Nur die `MV`-Befehle können dann parallel verwendet werden, da diese immer in jedem Issue-Slot zur Verfügung stehen. In der 2xFU Variante können funktionale Einheiten verdoppelt werden, so dass die entsprechenden Operationen parallel in beiden Issue-Slots des Prozessors verwendet werden können. Die MMU (MAC and Multiply Unit) kann nicht verdoppelt werden, da die MAC-Operation (Multiply and Accumulate) zwei Zielregister beschreibt und damit nicht parallel verwendet werden kann. In jeder Konfiguration einer 2xFU-Variante ist mindestens eine Einheit verdoppelt. In der 2xFU.X2 Variante können funktionale Einheiten ebenfalls verdoppelt werden, zusätzlich kann für verdoppelte Einheiten auch der X2-Modus aktiviert werden. In diesen Konfigurationen ist der X2-Modus für die `MV`-Befehle sowie für die ALU-Einheit immer aktiv, da diese Einheiten in den manuell fusionierten Operationen zum Speicherzugriff bzw. der konditionalen Ausführung verwendet werden.

Die Platzierung von LUT- und Register-Instanzen auf die FPGA-Ressourcen während der Hardwaresynthese ist mit einer Zufallskomponente versehen. Im Synthesetool kann dazu der Zufallsgenerator mit einem *Seed* initialisiert werden. Zu jeder Konfiguration wurden daher fünf Synthesen mit unterschiedlichen Seeds durchgeführt, um Ausreißer in der erzielbaren Taktperiode zu vermeiden.

In Abbildung 6.8 ist die Verteilung der minimalen Taktperiode für verschiedene Konfigurationen des TUKUTURI-Prozessors in der 1xFU-Variante über der Summe der Latenzen, die in den entsprechenden Konfigurationen eingefügt wurden, dargestellt.

Konfigurationen mit einer Summe der Latenzen von 0 entsprechen der Grundkonfiguration, in der keine zusätzlichen Pipelinestufen eingefügt wurden. Ein Inkrement dieser Summe der Latenzen kann folgende Gründe haben:

- In eine funktionale Einheit wurde eine zusätzliche Pipelinestufe eingefügt. Dies kann eine Einheit sein, die bisher noch nicht mit einer zusätzlichen Stufe konfiguriert war, oder eine Einheit, die bereits zusätzliche Pipelinestufen enthält. Für die meisten Einheiten können bis zu drei zusätzliche Stufen eingefügt werden. Zusätzliche Pipelinestufen werden ebenfalls für die Coprozessoren, z. B. den Divider, berücksichtigt.
- Es wurden eine zusätzliche Decoderstufe (`DE`) oder Extra-Register für Speicheroperationen oder Sprungbefehle eingefügt (`BUSW`, `FLOW`).
- Die Anzahl zusätzlicher Forwarding-Stufen (`FWD`) geht in die Summe der Latenzen mit einem Gewicht von 7 ein, da sich eine zusätzliche Forwarding-Stufe auf die Latenz jeder der 7 funktionalen Einheiten auswirkt.

Der linke Graph zeigt die Konfigurationen, die für die entsprechende Summe an Latenzen die minimale Taktperiode erreicht haben. Für eine gegebene Summe von Latenzen erhält man auch Konfigurationen mit höheren Taktperioden, z. B. wenn die zusätzlichen Pipelinestufen in Einheiten eingefügt werden, die nicht im kritischen Pfad liegen. Dies ist im linken Graphen durch den grau hinterlegten Bereich dargestellt, der den Wertebereich der Taktperioden für die jeweiligen Summen der Latenzen angibt.

Der NSGA-II Algorithmus aus Abschnitt 5.8 verwendet als Zielfunktion für die Optimierung die Taktperiode der Konfigurationen, da diese nach der Hardwaresynthese direkt ermittelt wird. Letztendliches Ziel bei der Optimierung ist jedoch die Maximierung der Anzahl der Bilder, die pro Sekunde verarbeitet werden können (Frames Per Second (FPS)). Da diese nicht ausschließlich von der Hardwaresynthese, sondern außerdem von der Kompilierung des Anwendungscodes abhängt, würde der Aufwand zur Ermittlung dieser Kennzahl, die die Kompilierung des Anwendungscodes und die Emulation des Systems mit Testdaten beinhaltet, den Fortschritt des NSGA drastisch verlangsamen. Zwar sind die Konfigurationen mit minimaler Taktperiode nicht zwangsläufig die Konfigurationen mit maximalen FPS, trotzdem ist die Verwendung der Taktperiode in der Optimierung ein ausreichendes Kriterium. Die Konfigurationen, die maximale FPS erreichen, sind mit den jeweils erreichten Taktperioden im rechten Graphen der Abbildung 6.8 dargestellt.

Auswahl der Konfigurationen: NSGA-II

Die Auswahl bzw. Optimierung der TUKUTURI-Konfigurationen erfolgt mit dem bereits in Abschnitt 5.8 erläuterten genetischen Algorithmus (NSGA-II).

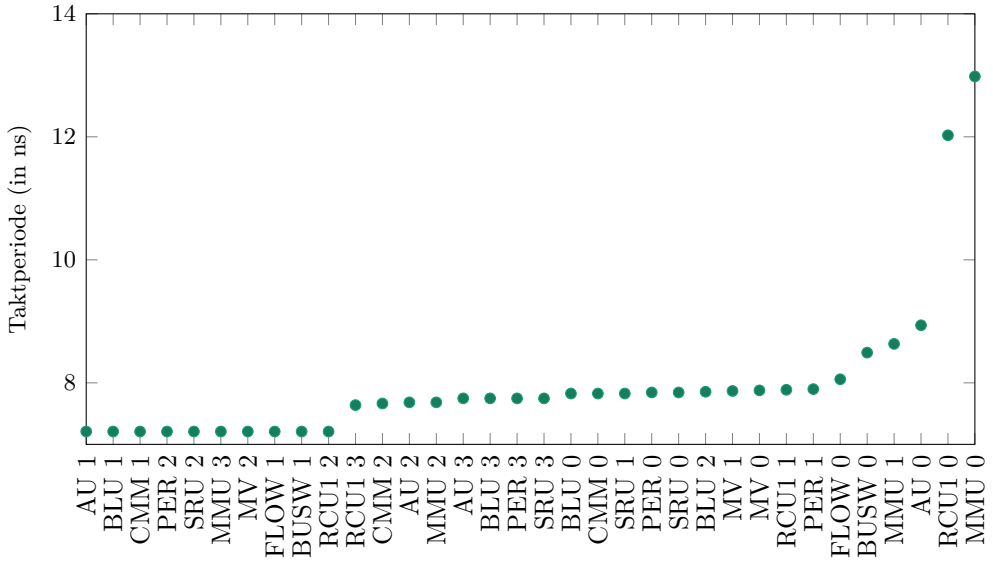


Abbildung 6.9: Analyse des kritischen Pfads für verschiedene TUKUTURI-Konfigurationen

Unter anderem wird dort beschrieben, wie Informationen zum kritischen Pfad in der Architektur untersucht werden, um einerseits zielgerichtet Nachkommen im genetischen Algorithmus zu erzeugen, und andererseits über den Abbruch des Verfahrens zu entscheiden. Die Abbildung 6.9 zeigt ein Beispiel für eine solche Auswertung für alle Konfigurationen aus einer Generation des Algorithmus. Auf der horizontalen Achse sind verschiedene Konfigurationsparameter aufgeführt. So steht beispielsweise die Bezeichnung `AU 2` für eine Arithmetic Unit mit zwei zusätzlichen Pipelineregistern (also einer Latenz von 3 Taktzyklen). Die Parameter `FLOW` und `BUSW` sind in Tabelle 3.1 erläutert. Da die Grafik nur Konfigurationen mit `FWD 1` und `DE 1` enthält, sind diese Parameter selbst nicht zu finden.

Für jeden Parameter wurde unter allen Konfigurationen der aktuellen Generation im GA die minimal erreichbare Taktperiode ermittelt. Diese wird als Abschätzung für den kritischen Pfad dieser funktionalen Einheit bzw. des parametrisierten Moduls verwendet. Die Einheiten bzw. Parameter sind dann aufsteigend nach Taktperiode sortiert. Tendenziell sollten also die Konfigurationen einer Einheit mit einer höheren Zahl von Pipelineregistern einen kürzeren kritischen Pfad besitzen und daher weiter links stehen als die Konfigurationen derselben Einheit mit einer geringeren Zahl von Pipelinestufen. Dies ist im Beispiel für die `MMU` zu sehen, für die die vier Konfigurationen `MMU 0`, `MMU 1`, `MMU 2` und `MMU 3` in der erwarteten Reihenfolge auftreten. Für andere Einheiten ist die Sortierung nicht korrekt, beispielsweise sind `AU 2` und `AU 3` vertauscht und beide langsamer als

`AU 1`. Dies kann zum einen daran liegen, dass die Parameter 2 und 3 nur in TUKUTURI-Konfigurationen auftraten, in denen andere Einheiten einen höheren kritischen Pfad aufwiesen und damit den kritischen Pfad der `AU` verdeckten. In diesem Fall sollten durch die in Abschnitt 5.8 beschriebenen Operatoren optimierte Konfigurationen für die nächste Generation abgeleitet werden. Zum anderen können auch Sättigungseffekte auftreten, die eine weitere Beschleunigung einer Einheit durch Einfügen weiterer Pipelinestufen verhindern. Dies führt dann zu sehr geringen Unterschieden in den ermittelten Taktperioden und den flachen Bereichen in der Grafik.

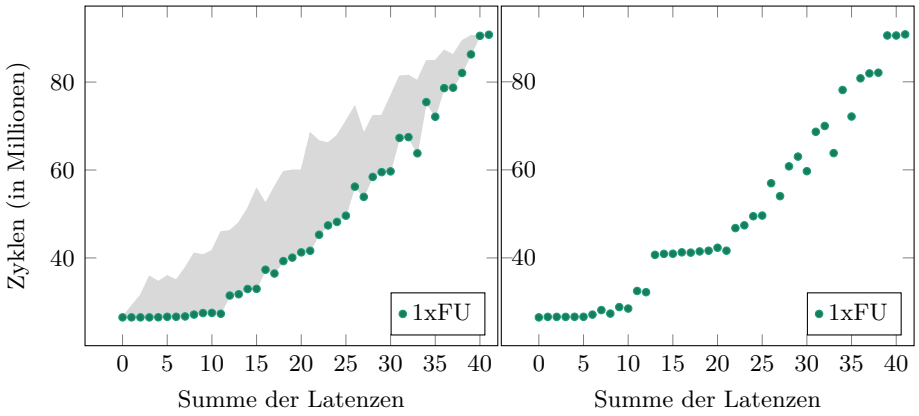
Um das Abbruchkriterium für den GA zu entscheiden, wird zunächst für jede Einheit bzw. jeden Parameter gezählt, wie oft ein höherer Parameter (also zusätzliche Pipelinestufen) zu einem langsameren Ergebnis führte als ein niedrigerer Wert. Dabei wird, um den oben genannten Sättigungseffekt zu berücksichtigen, eine Toleranz von 5 % von der jeweils schnellsten Konfiguration der Einheit verwendet. Die Anzahl dieser *Inversionen* über alle Parameter wird aufsummiert und der Algorithmus abgebrochen, wenn die Summe 0 erreicht oder einen kleinen Wert unterschreitet.

Begonnen wurde die Optimierung mittels des NSGA-II mit einer Variante 1xFU des TUKUTURI, bei der jede funktionale Einheit einmal im Prozessor vorhanden ist. Aus den Ergebnissen wurden dann entsprechende Konfigurationen der Varianten 2xFU bzw. 2xFU.X2 abgeleitet. Dazu sind aus allen Ergebnissen für die 1xFU-Konfigurationen zu jeder Summe der Latenzen die zwei Konfigurationen ausgewählt worden, die maximale FPS erreichen. Ohne Veränderung der Pipelineparameter wurden dann Konfigurationen für alle Kombinationen aus verdoppelten Einheiten bzw. Einheiten mit X2-Modus erstellt. Ein weitere Optimierung der Pipelineparameter mittels des GA wurde nicht vorgenommen.

Kompilierung

Neben der Synthese des TUKUTURI für unterschiedliche Konfigurationen ist die Verkehrszeichendetektion für die entsprechenden Prozessorkonfigurationen auch kompiliert worden. Bei der Kompilierung für Varianten, die nicht den X2-Modus verwenden (1xFU, 2xFU), wurde ein Schedulinglevel $o = 12$ gewählt. Die Kompilierung für die Variante mit X2-Modus (2xFU.X2) erfolgte mit Scheduling- bzw. Mergelevel $o = 6$, $x = 4$. Die EA-basierte Registerallokation kam nicht zum Einsatz, da hier keine Einschränkungen in den Registerbänken vorgenommen wurden und der Effekt des EA auf die Registerallokation in diesem Fall vernachlässigt werden kann. Diese gewählten Parameter sollten nach Abbildung 4.19 gute Ergebnisse bei moderater Laufzeit erzielen.

Die Kompilierung des Assemblercodes ist zufallsabhängig. Um Ausreißer zu vermeiden, wurde der Anwendungscode daher für jede TUKUTURI-Konfiguration fünfmal kompiliert. Die zur Kompilierung verwendeten EAs aus Abschnitt 4.3



(a) Konfigurationen mit minimalen Zyklen **(b)** Konfigurationen mit maximalen FPS

Abbildung 6.10: Taktzyklen für einen Frame für die Verkehrszeichendetektion, kompiliert für verschiedene TUKUTURI-Konfigurationen (Minimalkonfiguration). Der grau hinterlegte Bereich zeigt die Streuung der Zyklenzahl für die 1xFU-Variante. Die Punkte in beiden Grafiken gehören jeweils zu den Konfigurationen mit maximal erreichten FPS, für die in der linken Grafik die Anzahl der Taktzyklen gezeigt ist.

verwenden die Codegröße als Zielfunktion, da diese nach der Kompilierung direkt gemessen werden kann. Das eigentliche Ziel bei der Optimierung der TUKUTURI-Architektur ist aber die Anzahl der pro Sekunde verarbeiteten Bilder (FPS), wofür nicht die Codegröße, sondern die Anzahl der benötigten Taktzyklen (in Kombination mit der Taktfrequenz der entsprechenden Konfiguration) ausschlaggebend ist, die in einer Emulation des Systems ermittelt werden können. Um den hohen (zeitlichen) Aufwand der Emulation zu umgehen, wurde der in Abschnitt 5.6 vorgestellte *Profiling Estimator* verwendet, um Abschätzungen des Profilings für alle Kombinationen von synthetisiertem Prozessor und kompiliertem Code zu ermitteln. Dazu wurde zunächst für jede Variante des TUKUTURI eine Emulation des Prozessors mit entsprechend kompiliertem Anwendungscode durchgeführt, um Profilingergebnisse für die Verkehrszeichendetektion zu ermitteln. Anschließend wurde der in Abschnitt 5.6 vorgestellte *Profiling Estimator* verwendet, um Abschätzungen des Profilings für alle Konfigurationen einer Variante des Prozessors zu ermitteln. Da für jede Konfiguration jeweils fünf Synthesen und fünf Kompilierungen vorliegen und diese beliebig kombiniert werden können, werden mittels des Estimators zu jeder Konfiguration 25 Profilings ermittelt. Ergebnisse sind in Abbildung 6.10 dargestellt. Im linken Graphen sind die Konfigurationen, die minimale Taktzyklen erreichen, eingetragen. Der grau hinterlegte Bereich kennzeichnet den Wertebereich, der durch die verschiedenen Konfigurationen erzielt wird. Im rechten Graphen der Abbildung sind die Konfigurationen eingetragen, die in Emulation die höchsten FPS erreicht haben.

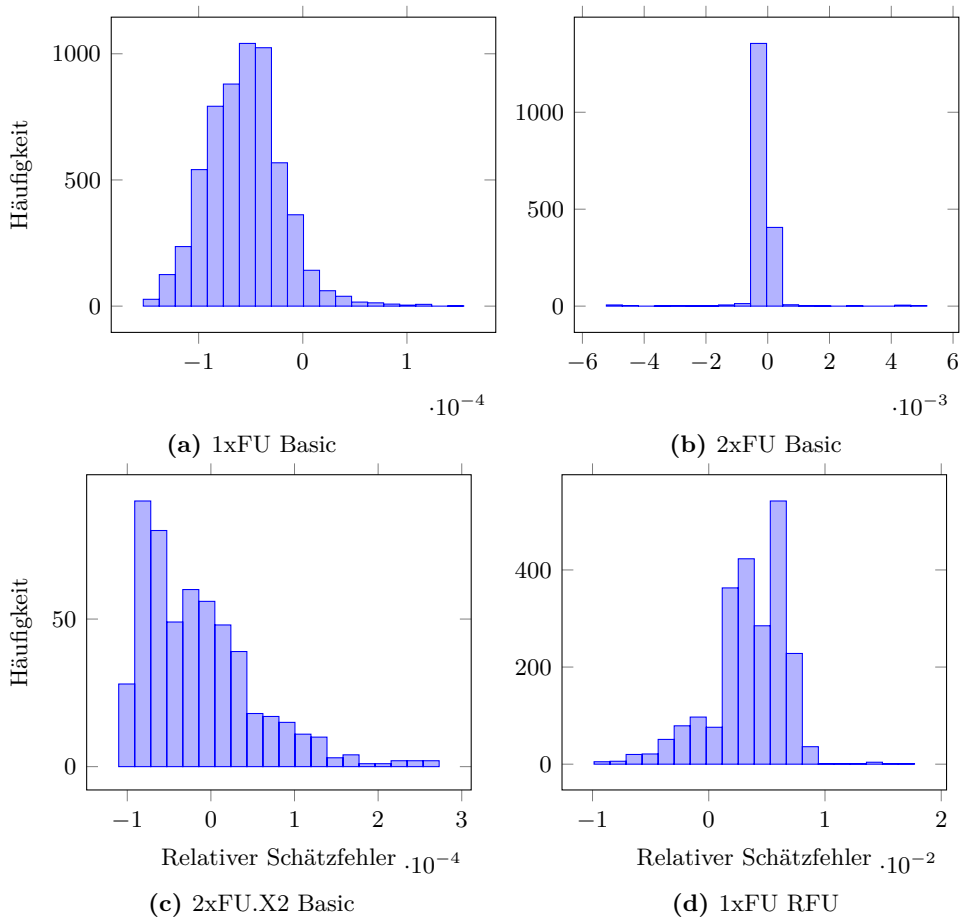


Abbildung 6.11: Fehler in der Abschätzung des Profiling mittels Profiling Estimator im Vergleich zum Profiling nach Emulation.

Der Fehler in den Profilingergebnissen, der durch die Verwendung des Profiling Estimators gegenüber den Ergebnissen aus einer Emulation entsteht, ist in Abbildung 6.11 dargestellt. Zur Evaluation dieser Schätzung wurden mehr als 1800 verschiedene Prozessorkonfigurationen mit entsprechend kompiliertem Code durch Emulation und anschließendes Profiling ausgewertet und mit den Ergebnissen des Profiling Estimators verglichen. Der Fehler in der geschätzten Anzahl der Taktzyklen betrug dabei durchschnittlich 0,0065 % und ist ausschließlich durch die DMA-Transfers verursacht. Dies liegt darin begründet, dass das DMA-Modul unabhängig von der Taktfrequenz des Prozessors immer mit 100 MHz arbeitet und sich damit die Dauer von Transfers gemessen in Taktzyklen des Prozessors verändert, wenn sich die Frequenz des Prozessors verändert. Wird der Unterschied der Taktfrequenz zwischen dem Referenz-Profiling und der aktuellen Prozessor-

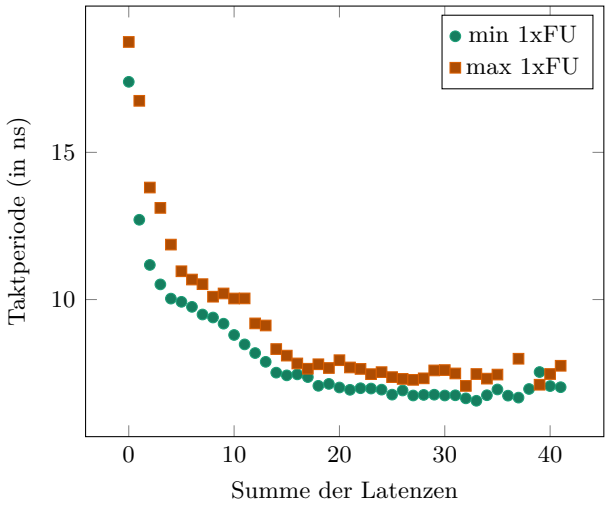


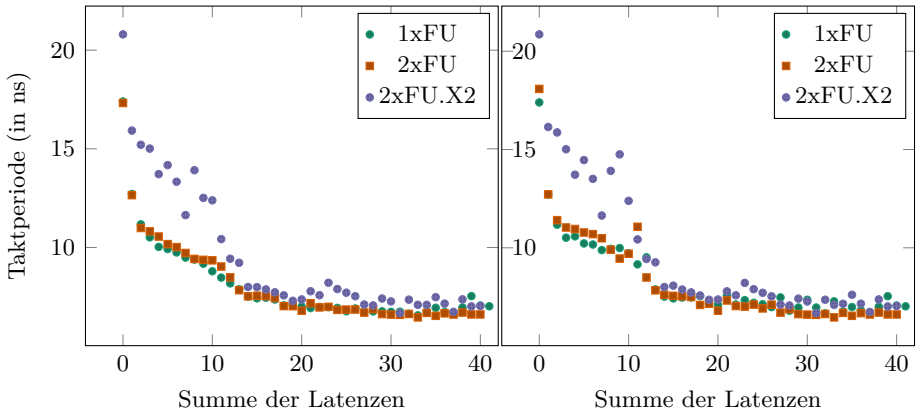
Abbildung 6.12: Minimal erreichte Taktperiode für die Minimal- und Maximalkonfiguration des TUKUTURI mit jeweils einer Instanz aller funktionalen Einheiten (1xFU) über der Anzahl der eingefügten Pipelinestufen (ausgedrückt als Summe der Latenzen).

konfiguration berücksichtigt, sinkt der durchschnittliche Fehler auf 0,0038 %. Eine weitere Verbesserung wird erreicht, wenn der Umrechnungsfaktor berücksichtigt, dass die Transfers bei einer Prozessorfrequenz von mehr als 100 MHz durch das DMA-Modul begrenzt sind. Dann sinkt der durchschnittliche Fehler auf 0,0036 %.

6.2.3 Profilingergebnisse

Wie in Abschnitt 3.2 beschrieben, kann die TUKUTURI-Architektur für minimierten Ressourcenaufwand an die verwendete Anwendung angepasst werden, indem funktionale Einheiten und Operationsmodi, die im Anwendungscode nicht zum Einsatz kommen, deaktiviert werden. Für die Evaluation wurde die Minimalkonfiguration für die 1xFU-Variante mit der entsprechenden Maximalkonfiguration des TUKUTURI, in der alle funktionalen Einheiten und Operationsmodi enthalten sind, verglichen.

Aus dem Vergleich von Minimal- und Maximalkonfiguration, der in Abbildung 6.12 dargestellt ist, kann entnommen werden, dass die Minimalkonfiguration des Prozessors durchgängig kürzere Taktperioden erreicht als korrespondierende Maximalkonfigurationen. Zur Begründung kann die Größe der Prozessorarchitektur herangezogen werden. Zum einen sind nicht benötigte Einheiten enthalten, zum anderen sind die vorhandenen Einheiten größer als in der Minimalkonfiguration, da sie mehr Optionen unterstützen. Dadurch kommt es zu einem höheren Routingaufwand bei der Hardwaresynthese für das FPGA, wodurch die Gesamtperformanz des Systems sinkt. Deshalb werden in der folgenden Evaluation



(a) Konfigurationen mit minimaler Periode **(b)** Konfigurationen mit maximalen FPS

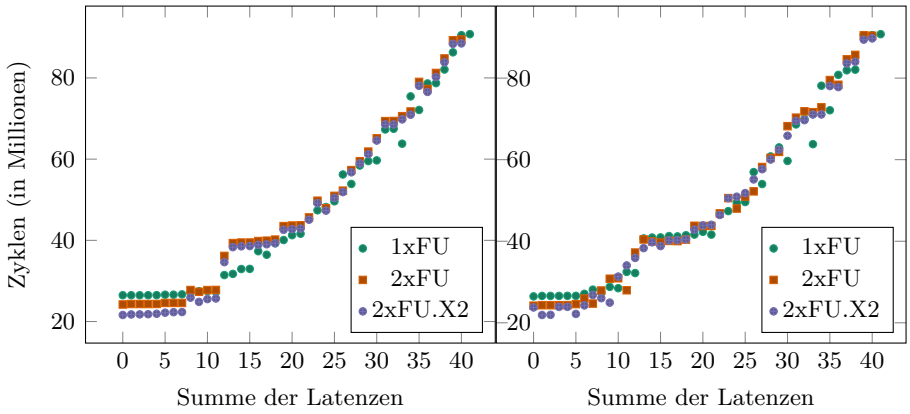
Abbildung 6.13: Vergleich der minimalen Taktperiode verschiedener Varianten.

nur noch die Minimalkonfigurationen der verschiedenen TUKUTURI Varianten untersucht.

Die Abbildung 6.13 zeigt die minimal erreichte Taktperiode für verschiedene Varianten des TUKUTURI-Prozessors für verschiedene Konfigurationen (also verschiedene Summen der Latenzen). Wie in Abbildung 6.8 sind dabei in der linken Grafik die Konfigurationen aufgeführt, die minimale Taktperiode erreicht haben, und in der rechten Grafik sind die Konfigurationen aufgeführt, die maximale FPS erreicht haben.

Der Vergleich der drei TUKUTURI-Varianten zeigt, dass die Taktperiode mit der Größe der Variante steigt: die 1xFU-Variante erreicht niedrigere Taktperioden als die 2xFU-Variante, und diese ist wiederum schneller als die 2xFU.X2-Variante. Dies ist auf erschwertes Routing durch die erhöhte Anzahl an Forwarding-Pfaden zurückzuführen. Für aktivierten X2-Modus (2xFU.X2) ist das Routing vor allem für die Konfigurationen mit FWD0 erschwert. Sobald eine FWD-Stufe eingefügt wird (Summe der Latenzen größer als 10), liegt die Taktperiode nur noch leicht über der für die 1xFU-Konfigurationen.

Mit steigender Zahl zusätzlicher Pipelinestufen sinkt die Taktperiode des Prozessors wie erwartet, womit die maximal erreichbare Taktfrequenz steigt. Wenn die minimale Taktperiode für Konfigurationen mit FWD0 eine Sättigung erreicht (ab einer Summe von etwa 10 zusätzlichen Pipelinestufen), kann die Taktperiode durch Einfügen einer zusätzlichen Pipelinestufe im Forwarding (FWD1) weiter verringert werden. Ab einer Zahl von etwa 25 zusätzlichen Pipelinestufen tritt eine Sättigung ein, ab der die Taktperiode nicht weiter verringert werden kann. Dann liegen die funktionalen Einheiten nicht mehr im kritischen Pfad bzw. der Verlust durch zusätzliches Routing bei weiterer Erhöhung der Anzahl an Pipelinestufen hebt den Gewinn durch weitere Pipelinestufen auf.



(a) Konfigurationen mit minimalen Zyklen **(b)** Konfigurationen mit maximalen FPS

Abbildung 6.14: Taktzyklen zur Verarbeitung eines Frames.

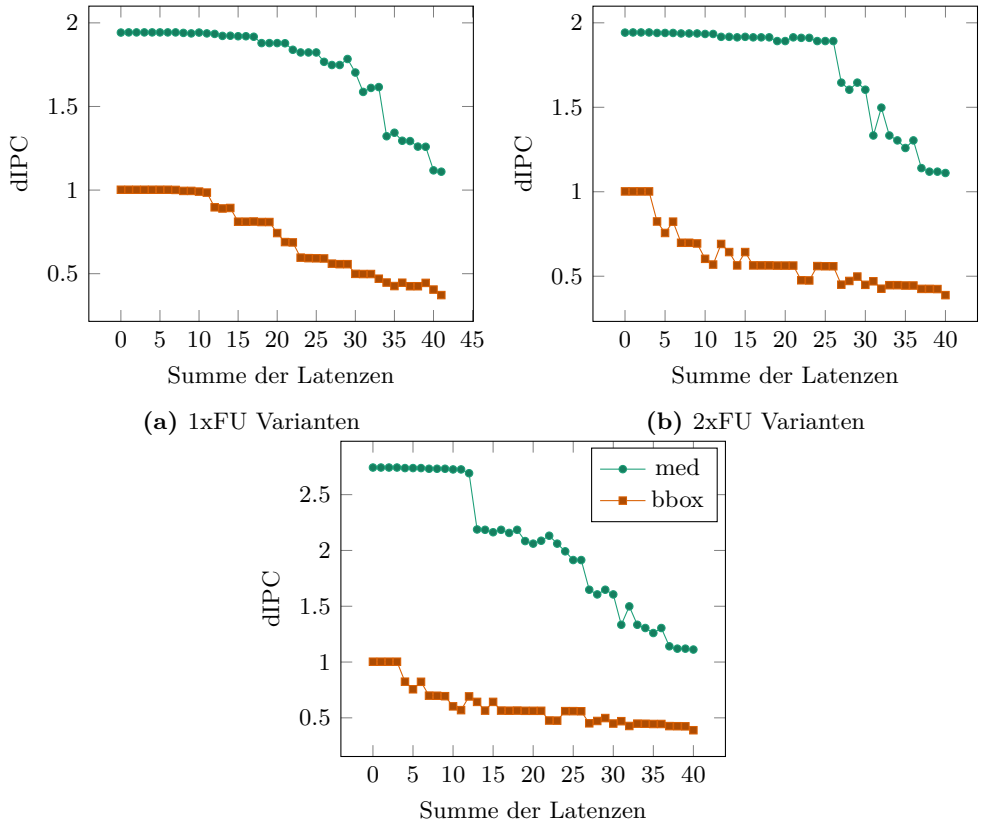
Eine steigende Taktfrequenz auf Kosten zusätzlicher Pipelinestufen führt nur dann zu einer Steigerung der Gesamtperformanz des Systems, wenn die Auswirkungen der erhöhten Latenzen der funktionalen Einheiten im Anwendungscode den Gewinn der höheren Taktfrequenz nicht überdecken.

Abbildung 6.14 zeigt die minimale Zahl von Taktzyklen der Verkehrszeichendetektion für einen Frame, kompiliert für die verschiedenen TUKUTURI-Konfigurationen abhängig von der Summe der Latenzen. Die folgenden Punkte beschreiben die Ergebnisse:

- Die zusätzlichen Latenzen der funktionalen Einheiten können anfangs (bis etwa 7) durch ein geeignetes Instruction Scheduling vom Kompiler ausgeglichen werden, ohne dass sich die Anzahl der Taktzyklen erhöht.

- Mit weiter steigender Summe der Latenzen steigt tendenziell auch die Anzahl der Taktzyklen, da mehr Wartezyklen in den Code eingefügt werden, die der Kompiler aufgrund von Abhängigkeiten zwischen den Operationen nicht mehr ausfüllen kann.

- Für die 1xFU-Variante benötigt der Code für die TUKUTURI-Konfiguration mit 32 zusätzlichen Pipelinestufen mehr Taktzyklen als für die Konfiguration mit 33 Stufen. Dies lässt sich damit begründen, dass der Code mit Latenz 32 zwei Forwarding-Stufen verwendet (FWD2), der mit Latenz 33 jedoch nur eine (FWD1). Die Anzahl der Forwarding-Stufen geht mit einem Gewicht von 7 in die Summe der Latenzen ein, daher muss im Vergleich der Konfigurationen mit Latenzen 32 und 33 nicht nur eine funktionale Einheit mit einer zusätzlichen Pipelinestufe versehen werden,



(a) 1xFU Varianten **(b)** 2xFU Varianten

(c) 2xFU.X2 Varianten

Abbildung 6.15: Dynamische Instructions Per Cycle (IPC) zweier exemplarischer Routinen der Verkehrszeichendetektion.

sondern es müssen 8 zusätzliche Pipelinestufen auf die Einheiten verteilt werden, wodurch die Zahl der benötigten Taktzyklen steigt.

Der Einfluss der zusätzlichen Pipelinestufen auf die Kompilierung kann auch aus der Abbildung 6.15 abgeleitet werden. Hier sind für zwei exemplarische Routinen der Verkehrszeichendetektion die maximal erreichbaren (dynamischen) Instructions Per Cycle (IPC) dargestellt. Dynamisch bedeutet in diesem Fall, dass die Anzahl der Instruktionen mit der Häufigkeit ihrer Ausführung gewichtet wurden. Zunächst ist zu erkennen, dass der Kompiler die Median-Routine, die viele Berechnungen durchführt, deutlich dichter anordnen kann als die Bounding-Box-Routine, in der viele Speicheroperationen ausgeführt werden. Durch die Unabhängigkeit vieler Operationen kann der Einfluss zusätzlicher Pipelinestufen im Median-Filter deutlich besser kompensiert werden als in der Bounding-Box-Routine. Dieser Effekt verstärkt sich, wenn die Anzahl der funktionalen Einheiten

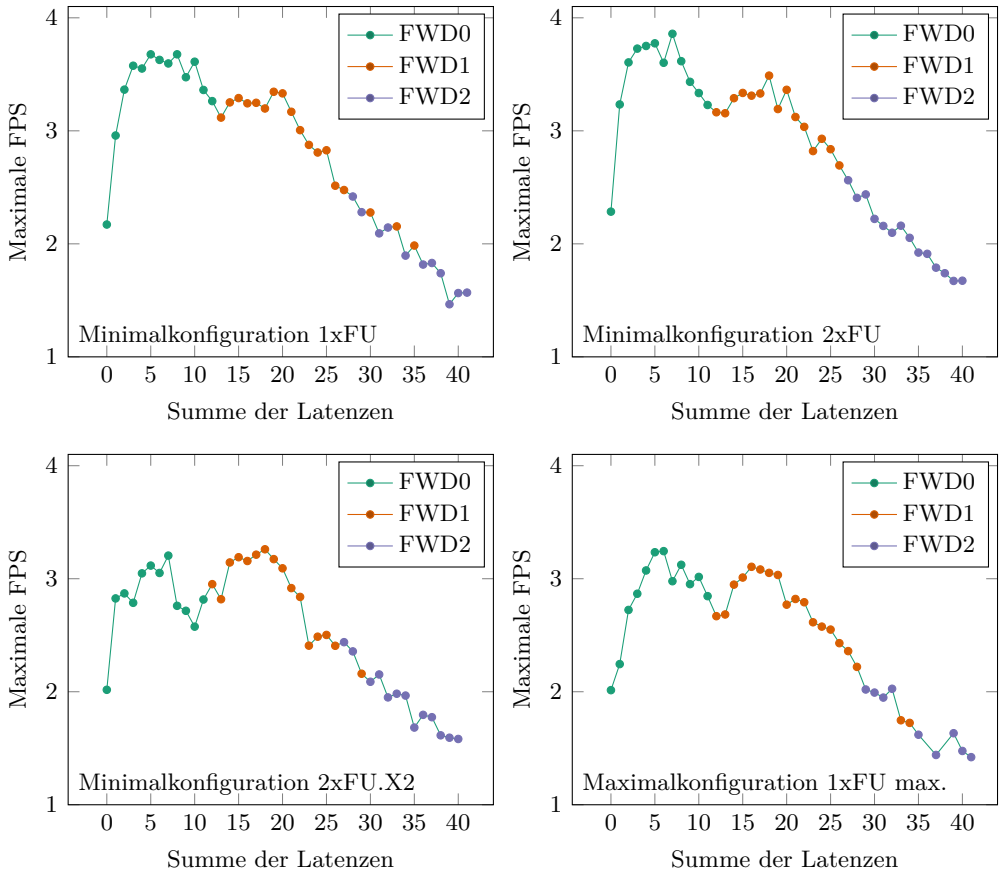


Abbildung 6.16: Maximal erreichbare Frames pro Sekunde (FPS) für verschiedene TUKUTURI-Konfigurationen in Abhängigkeit der jeweiligen Gesamtlatenz.

verdoppelt wird (2xFU-Variante), da hierdurch die Möglichkeiten für den Kompiler steigen. In der 2xFU.X2-Variante hingegen kann der Kompiler die zusätzlichen Latenzen weniger gut ausgleichen, da durch die Fusionierung der Operationen deren Unabhängigkeit genauso wie die Gesamtzahl der Instruktionen verringert wird.

Durch die Kombination aus steigender Taktfrequenz durch zusätzliche Pipelinestufen und nahezu konstanter Zahl von Taktzyklen durch optimales Instruction Scheduling wird eine gesteigerte Gesamtperformanz des Systems erreicht, wie in Abbildung 6.16 durch die erreichte Anzahl an FPS dargestellt. Daraus lassen sich folgende Beobachtungen ableiten:

- Die Gesamtperformanz des Systems steigt bis zu einer Anzahl von etwa 6 zusätzlichen Pipelinestufen an, da mit steigender Zahl der Pipelinestufen die Taktperiode des TUKUTURI-Prozessors sinkt und gleichzeitig die

Zyklenzahl für die Anwendung aufgrund des Instruction Scheduling nicht steigt.

- Mit dem Einfügen weiterer Pipelinestufen steigen die Latenzen der funktionalen Einheiten zu stark an, als dass der Kompiler dies durch das Instruction Scheduling ausgleichen könnte, wodurch der Gewinn der steigenden Taktfrequenz aufgehoben wird und die Gesamtperformanz sinkt.

- Mit dem Einfügen einer zusätzlichen Forwarding-Stufe (FWD1) wird zwar die Taktperiode des Prozessors nochmals gesenkt (vgl. Abbildung 6.13), doch dies führt zu einer höheren Zahl von Wartezyklen in der Anwendung, wodurch die FPS für Konfigurationen ohne zusätzliche Forwarding-Stufe (FWD0) nicht erreicht werden. Eine Ausnahme ist die X2-Variante (2xFU.X2), da hier die Taktperiode für FWD0 deutlich erhöht ist.

- Die Verdopplung der funktionalen Einheiten (2xFU) erhöht zwar die minimale Taktperiode für diese Konfigurationen gegenüber 1xFU, gleichzeitig sinkt aber auch die Codegröße, wodurch netto ein Gewinn in Gesamtperformanz erreicht werden kann.

- Für X2-Konfigurationen und die Maximalkonfiguration des Prozessors ist der Verlust in Taktfrequenz größer als der Gewinn in Taktzyklen, wodurch die Gesamtperformanz gegenüber den anderen Konfigurationen geringer ausfällt.

Abbildung 6.17 zeigt die erreichbaren FPS für verschiedene Konfigurationen des Prozessors abhängig von der Summe der Latenzen und der Anzahl der enthaltenen funktionalen Einheiten. Zu erkennen ist hier, dass es je nach Summe der Latenzen eine jeweils optimale Anzahl von funktionalen Einheiten gibt, insbesondere im Bereich einer Summe der Latenzen von etwa 4 bis 16. Mit mehr als 11 funktionalen Einheiten nimmt die Performanz des TUKUTURI wieder ab, da sich der zusätzliche Hardwareaufwand bzw. das Routing negativ auf die minimale Periode auswirkt.

Für die Datentransfers zwischen externem und internem Speicher mittels des DMA-Moduls wurde für fast alle Routinen ein double-buffering Schema verwendet, bei dem die Bilddaten in Blöcken geladen werden und der nächste Datenblock geladen wird, während der vorab transferierte Block bearbeitet wird. Die Routinen werden dabei voneinander isoliert, indem am Ende einer Routine auf den Abschluss aller Transfers gewartet wird, bevor die nächste Routine begonnen wird. Es finden also zwischen den Routinen keine Datentransfers statt. Dies dient der Modularität der Implementierung und erleichtert das Debugging. Diese Isolierung bedeutet, dass am Anfang einer Routine mindestens auf den ersten Datenblock gewartet werden muss, bis mit der Verarbeitung begonnen werden

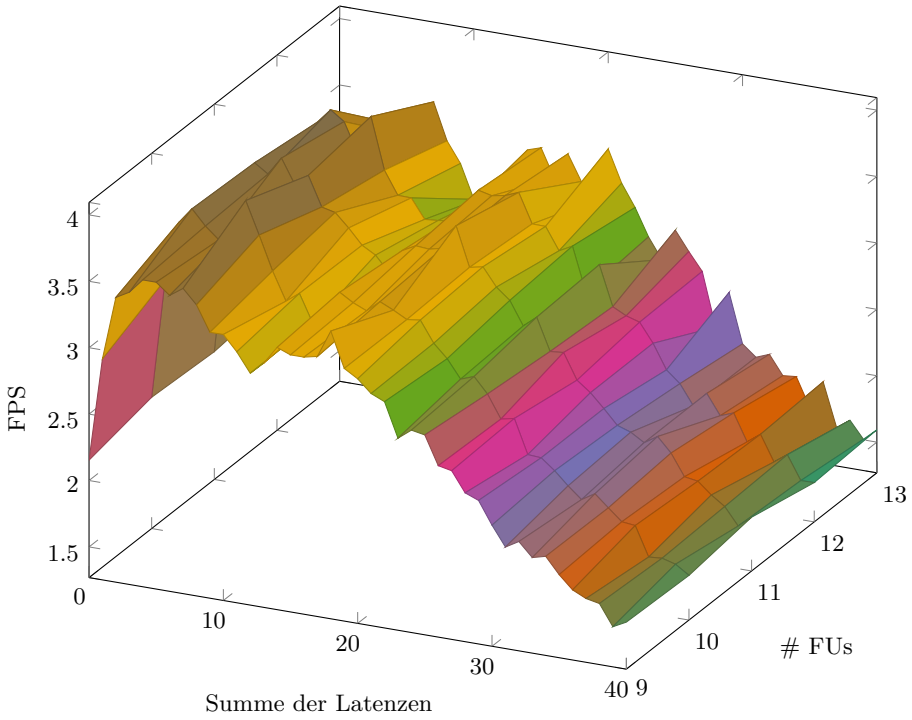


Abbildung 6.17: Erreichbare FPS in Abhängigkeit der Summe der Latenzen und der Anzahl der enthaltenen funktionalen Einheiten.

kann. Ebenso muss am Ende das Schreiben des letzten Ergebnisblocks abgewartet werden, bevor die Routine schließlich endet. Die Analyse der Speichertransfers ist in Tabelle 6.2 dargestellt.

Für diese Auswertung wurden Konfigurationen gewählt, die möglichst wenig zusätzliche Pipelinestufen oder Latenzen enthalten. In diesem Fall ist die Anzahl der Zyklen für die Datenverarbeitung am geringsten und damit auch die Anzahl der Zyklen, die für Hintergrundtransfers durch den DMA zur Verfügung stehen.

Die erste Zeile enthält absolute Zyklenzahlen für den gesamten Algorithmus zur Verarbeitung eines Frames. Die Spalte „Berechnung“ gibt dabei nur die Zyklen an, in denen der Prozessor nicht in einer Warteschleife auf Daten wartet. Die Zyklen zum Warten auf Daten sind aufgeteilt in die Initialisierung/Finalisierung am Anfang/Ende einer Routine und das Warten auf Transfers, die eigentlich im Hintergrund liegen sollten. Die letzte Spalte gibt schließlich die Anzahl an Zyklen an, in denen Daten im Hintergrund transferiert wurden. Die nachfolgenden Zeilen der Tabelle geben die Aufteilung der Taktzyklen für die einzelnen Routinen an. Dabei wird die Anzahl der Berechnungszyklen auf die Gesamtzahl für den gesamten Algorithmus bezogen. Die Anteile für die DMA Datentransfers sind immer auf die Gesamtzyklen der jeweiligen Routinen bezogen.

Tabelle 6.2: Analyse der Speichertransfers in der Implementierung mit Basisinstruktionssatz. Gezeigt sind die Zyklen zur Berechnung (relativ zur Gesamtzahl) sowie die Anzahl der DMA Zyklen (relativ zur Zyklenzahl der einzelnen Routinen).

(a) 1xFU-Variante

Routine	Berechnung	DMA Warten		DMA Transf.
		Init./Fin.	Bg.	Bg.
Gesamt	26590137	6924	0	2670258
Color	6,3545 %	0,0271 %	0 %	14,1856 %
Segment	13,0393 %	0,0242 %	0 %	12,5688 %
Median	13,9133 %	0,0280 %	0 %	33,6943 %
Labeling	62,5528 %	0,0104 %	0 %	3,5648 %
BBox	4,0699 %	0,0930 %	0 %	13,5411 %
Shape	0,0439 %	13,7308 %	0 %	0,3427 %

(b) 2xFU-Variante

Routine	Berechnung	DMA Warten		DMA Transf.
		Init./Fin.	Bg.	Bg.
Gesamt	26778978	6871	0	2756069
Color	6,4549 %	0,0267 %	0 %	14,0340 %
Segment	13,3829 %	0,0239 %	0 %	12,5461 %
Median	13,8333 %	0,0297 %	0 %	35,5079 %
Labeling	60,8887 %	0,0107 %	0 %	3,6640 %
BBox	5,3595 %	0,0580 %	0 %	10,4234 %
Shape	0,0548 %	11,3144 %	0 %	0,3361 %

(c) 2xFU.X2-Variante

Routine	Berechnung	DMA Warten		DMA Transf.
		Init./Fin.	Bg.	Bg.
Gesamt	21605708	6197	0	26785849
Color	5,0662 %	0,0387 %	0 %	20,0377 %
Segment	12,8488 %	0,0280 %	0 %	14,4012 %
Median	12,1254 %	0,0253 %	0 %	33,4048 %
Labeling	64,8704 %	0,0113 %	0 %	3,8172 %
BBox	5,0070 %	0,0931 %	0 %	12,0146 %
Shape	0,0532 %	13,1456 %	0 %	0,3502 %

Der Tabelle ist zu entnehmen, dass die Anzahl der Wartezyklen im Vergleich zur Gesamtzahl der Zyklen für einen Frame sehr gering ist. Der hohe relative Anteil in der Formklassifikation ist damit zu erklären, dass hier nur wenige Transfers im Hintergrund ausgeführt werden können, da ein wahlfreier Zugriff auf das Bild erfolgt und so der nächste Block nicht vorbestimmt ist.

Wie außerdem zu sehen ist, ist der Anteil der Hintergrundtransfers ebenfalls relativ gering. Die höchste Auslastung tritt mit etwa 33 % in der Median-Routine auf. Das bedeutet, dass die Hintergrundtransfers höchstens ein Drittel der zur Verfügung stehenden Taktzyklen verwenden. Daher muss in keinem Fall auf die Hintergrundtransfers gewartet werden und es werden durch die Datentransfers keine Verzögerungen erzeugt.

6.3 TUKUTURI-Implementierung mit Befehlssatzerweiterungen

6.3.1 Befehlssatzerweiterungen

Nach Analyse der innersten Schleifen, in denen der Großteil der Datenverarbeitung der jeweiligen Routinen des Algorithmus durchgeführt wird, wurden Teilaufgaben identifiziert, deren Umsetzung in Hardware zu einer Reduktion der Verarbeitungsschritte und damit zu einer schnelleren Verarbeitung führen. Dazu wurden kleinere Module als RFUs (Custom Instruction) und größere Module als RCUs (Coprozessoren) implementiert. In manchen Fällen ersetzen die neuen Instruktionen eine Folge von Grundoperationen aus der Basisimplementierung, in anderen Fällen ist die Struktur der Verarbeitung für den effizienten Einsatz der neuen Einheiten angepasst worden, beispielsweise zur besseren Ausnutzung von Datenparallelität in den RFUs. Die Implementierung mit speziellen Instruktionen liefert bitgenau identische Ergebnisse, wie die Implementierung im Basisinstruktionssatz. Die folgenden Absätze erläutern die Funktion der speziellen Instruktionen.

Für die **Farbraumtransformation** wird aufgrund der darin verwendeten Division der bereits erwähnte Coprozessor eingesetzt. Es sind keine zusätzlichen Hardwareerweiterungen für diese Teilaufgabe implementiert worden.

Zur Auswertung der Transferfunktionen in der **Farbsegmentierung** sind jeweils zwei RFUs für jede Farbe implementiert worden, von denen eine die Hue-Komponente und die zweite die Saturation-Komponente transformiert. Durch die Implementierung einer spezifischen Einheit für jede Komponente anstatt einer generischen Einheit zur Auswertung stückweiser linearer Funktionen sind die Einheiten sehr kompakt und effizient, da die konkreten Parameter bereits in der Hardwareanalyse bekannt sind. Die Einheiten verarbeiten acht 8-Bit Subworte parallel in einem Taktzyklus und verwenden dabei intern die gleiche shift-and-

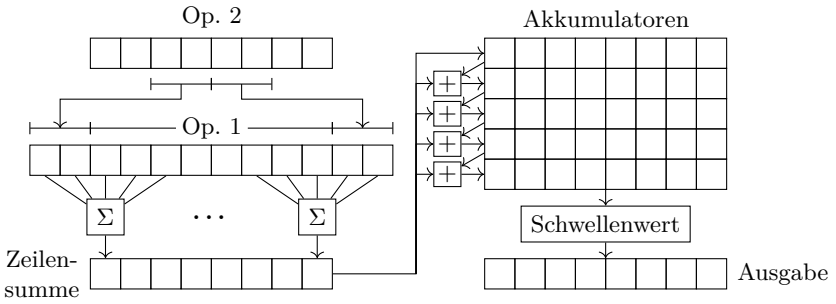


Abbildung 6.18: Funktionale Einheit für die Median-Berechnung. Der linke Teil der Darstellung zeigt kombinatorische Logik. Die Akkumulatorregister werden wie ein Schieberegister verwendet.

add-Technik wie die Implementierung im Basisinstruktionssatz. Sie ersetzen also direkt eine Sequenz von Operationen aus der Assemblerimplementierung mit dem Basisinstruktionssatz.

Das **Medianfilter** ist in einer eigenen RFU implementiert, die in Abbildung 6.18 dargestellt ist. Die Operanden der Einheit liefern acht 8-Bit Pixel (Operand 1) und vier zusätzliche Randpixel (Operand 2), um eine Zeile der 5×5-Pixel Filtermaske für acht Pixel gleichzeitig zu füllen. Die Pixel einer Zeile werden akkumuliert, um die Zeilensummen zu bilden, die dann auf die gespeicherten früheren Zeilen addiert wird. Dazu enthält die Einheit Register zur Speicherung von (akkumulierten) Pixelzeilen, die ähnlich wie ein Schieberegister verwendet werden. Auf die letzte Zeile des Akkumulatorarrays, die die Summen der letzten fünf Pixelzeilen enthält, kann dann der Schwellenwert angewendet werden, um den Median für acht Pixel zu liefern. Das Eingangsbild wird wie oben in Spalten verarbeitet. Nach einer Initialisierung mit zwei Nullzeilen und den ersten drei Bildzeilen am Anfang einer neuen Spalte liefert die Einheit ein Ergebnis nach jedem Aufruf. In der Assemblerimplementierung der Median-Routine werden zwei Instanzen der beschriebenen RFU eingesetzt. Diese verarbeiten parallel und unabhängig voneinander nebeneinander liegende Spalten von jeweils acht Pixeln.

Für das **Connected Component Labeling** sind zwei funktionale Einheiten für den ersten Durchlauf und ein Coprozessor für den zweiten Durchlauf implementiert worden. Um das initiale Label für ein Pixel im ersten Durchlauf zu berechnen, werden die Labels aus der Nachbarschaft des Pixels betrachtet. Dazu minimiert die erste funktionale Einheit, wie in Abbildung 6.19 dargestellt, die Labels von jeweils vier benachbarten Pixeln in einem durch die Bildzeile wandernden Fenster. Am Anfang einer neuen Bildzeile werden beim ersten Aufruf der Einheit in den zwei Operanden 16 aufeinander folgende Labels an die Einheit übergeben und die internen Register initialisiert. In den folgenden Aufrufen enthält der erste Operand die nächsten acht Label der Bildzeile, der zweite

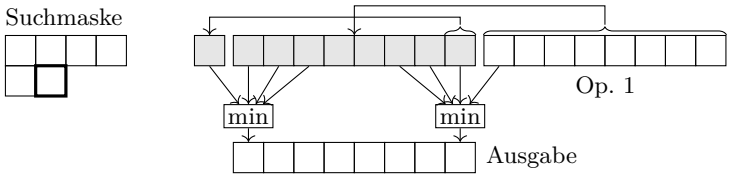


Abbildung 6.19: Erweiterte Suchmaske und Minimumsbestimmung für die erste Zeile der Maske im ersten Durchlauf des Connected Component Labeling. Die Einheit verwendet nur einen Operanden (Op. 1) für Pixelinformationen. Die Pixel des vorherigen Aufrufs werden intern in den grau hinterlegten Registern gespeichert.

Operand wird nicht verwendet. Bei jedem Aufruf der Einheit werden die Label der vorherigen Pixel, die im vorherigen Aufruf eingegeben wurden, wie in der Abbildung gezeigt in einem internen Register zwischengespeichert. Dann kann die Einheit die Minimierung für acht Pixel gleichzeitig durchführen.

Die zweite RFU für den ersten Durchlauf des Connected Component Labeling, die in Abbildung 6.20 gezeigt ist, berechnet dann die initialen Labels für die gesetzten Pixel. Das nächste noch nicht vergebene Label wird in einem internen Register (nxt_lbl) gespeichert. Der erste Operand enthält die minimierten Labels aus der Nachbarschaft des Pixels, wie von der ersten RFU berechnet. Der zweite Operand enthält die Pixel aus dem segmentierten Bild, da nur Pixel gelabelt werden dürfen, die auch gesetzt sind. Die Berechnung eines Labels findet in den mit „Minimum“ markierten Blöcken statt. Dazu wird zunächst geprüft, ob das entsprechende Pixel gesetzt ist. Ist dies der Fall, wird das Minimum der Labels aus der Nachbarschaft gewählt, sofern vorhanden. Sollte in der Nachbarschaft kein Pixel gelabelt sein, wird das intern gespeicherte Label verwendet und das Register inkrementiert. Sollte hingegen das Pixel nicht gesetzt sein, wird das gespeicherte Register unverändert in die nächste Stufe weitergegeben und kein Ausgabelabel für das Pixel erzeugt. Die acht Pixel der Ausgabe werden aus den jeweiligen Ausgaben der Berechnungsblöcke („out“-Register) zusammengesetzt. Die Einheit kann in drei verschiedenen Konfigurationen verwendet werden: In der ersten Konfiguration sind keine Pipelineregister eingefügt, so dass die gesamte Berechnung für acht Pixel in einem Taktzyklus durchgeführt wird. In der zweiten Konfiguration wird das blau eingezeichnete Pipelineregister aktiviert, so dass die Berechnung auf zwei Taktzyklen aufgeteilt wird. Die dritte Konfiguration enthält nur die grün dargestellten Pipelineregister, wodurch die Berechnung auf drei Taktzyklen aufgeteilt wird.

Zum Abschluss des Connected Component Labeling wird im zweiten Durchlauf das Bild zeilenweise von rechts unten nach links oben verarbeitet. Ein Coprozessor wählt dazu das kleinste Label aus einer (3×3)-Nachbarschaft um ein Pixel. Durch eine geschickte Reihenfolge, in der gelabelte Pixel an den Coprozessor übergeben werden, können Teilschritte der Minimierung nach jeder Übergabe berechnet

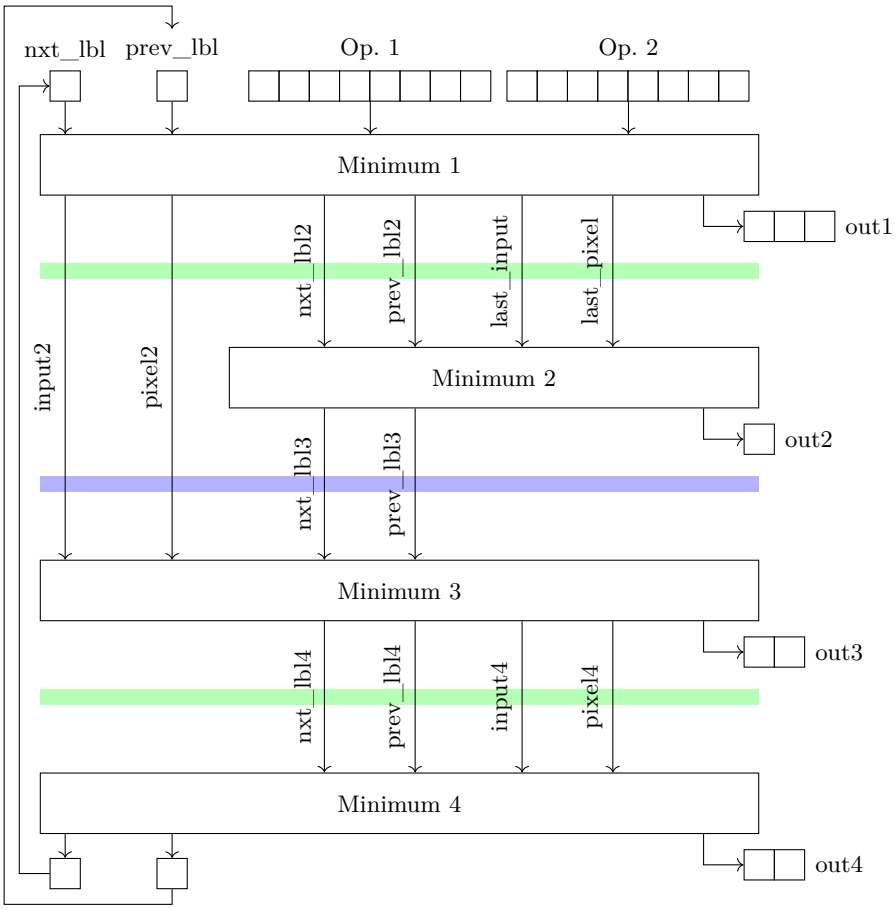


Abbildung 6.20: Berechnung der Labels für gesetzte Pixel im ersten Durchlauf des Connected Component Labeling. Die Pipelineregister (entweder blau order grün) sind optional: Die Einheit berechnet alle Labels in einem Taktzyklus, die Rechnung kann aber auch auf zwei oder drei Taktzyklen aufgeteilt werden.

werden. Diese Teilschritte können deutlich schneller als eine Minimierung in nur einem Schritt ausgeführt werden, wodurch der kritische Pfad im Coprozessor sinkt. Auch der Coprozessor verwendet Subwortparallelität. Es wird mit jedem Aufruf (`STORE`) ein Register mit acht Pixeln in einem von vier Modi übergeben. Der Coprozessor enthält einen internen Speicher, der in drei Spalten jeweils drei Zeilen speichert, wie in Abbildung 6.21 dargestellt. Die Berechnung des Minimums erfolgt für die Pixel in der rechten Spalte unter Einbeziehung jeweils eines Randpixels links und rechts. Je nach Modus beim Aufruf des Coprozessors werden die Daten in unterschiedliche Register abgelegt oder die Inhalte in den Registern verschoben. Modi 1 und 2 werden abwechselnd verwendet, um am Anfang einer neuen Bildzeile die internen Register zu füllen. Ein Aufruf im

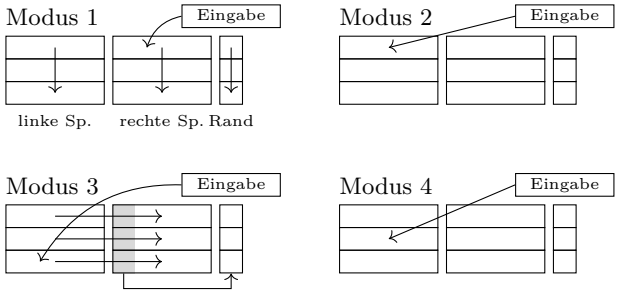


Abbildung 6.21: Datenübergabe an den Coprozessor für das Connected Component Labeling und Verschiebung der internen Register für vier verschiedene Modi

Modus 1 bewirkt eine Verschiebung der Registerinhalte um eine Zeile nach unten, schreibt Labeldaten in das frei gewordene Register in der rechten Spalte und setzt die Register für die Randspalte zurück. Modus 2 schreibt in das frei gewordene Register in der linken Spalte. Nach sechs Aufrufen sind die internen Register gefüllt und der Coprozessor liefert das erste Ergebnis. Mit dem Aufruf in Modus 3 werden die Inhalte der internen Register nach rechts verschoben, um in der Bildzeile weiter nach links vorzurücken und die nächsten Label in der dritten Zeile abzulegen. Modi 2 und 4 werden dann verwendet, um die linke Spalte wieder aufzufüllen. So werden diese drei Modi in der angegebene Reihenfolge verwendet, bis die Bildzeile vollständig abgearbeitet wurde. Jeder Aufruf in Modus 2 liefert ein Ergebnis. Die nächste Bildzeile beginnt dann wieder mit Aufrufen in den Modi 1 und 2.

Zur Bestimmung der **umschließenden Rechtecke** für die gelabelten Blobs müssen für jedes gelabelte Pixel die Koordinaten des bisher gefundenen Rechtecks aktualisiert werden. Zu diesem Zweck ist eine RFU implementiert worden, die für ein Eingaberegister, das acht gelabelte Pixel aus dem Eingangsbild enthält, einen Deskriptor berechnet, der anschließend von einer zweiten RFU verwendet wird, um das Rechteck zu aktualisieren. Die Berechnung von Deskriptoren ist schematisch in Abbildung 6.22 dargestellt. Als Eingabe erhält die Einheit ein Register mit acht gelabelten Pixeln aus dem Eingabebild (Operand 1) sowie die Bildkoordinaten des ersten Pixels (Operand 2). Im Beispiel liegt das erste Pixel im Eingaberegister an Position $(48, 7)$ im Eingangsbild. Nach dem ersten Aufruf der RFU gibt diese den mit "Ausgabe$_1$" markierten Deskriptor aus. Die ersten drei Felder in diesem Deskriptor bezeichnen die y-Koordinate sowie die x-Koordinaten am Anfang (x_s) und am Ende (x_e) der gefundenen Sequenz von Pixeln mit gleichem Label. Die vierte Komponente im Deskriptor ist das Label selbst. Die Einheit speichert die Koordinaten der letzten gefundenen Sequenz, so dass beim nächsten Aufruf (mit gleichen Eingaberegistern) der Deskriptor für die nächste Sequenz von Pixeln ausgegeben werden kann. Für

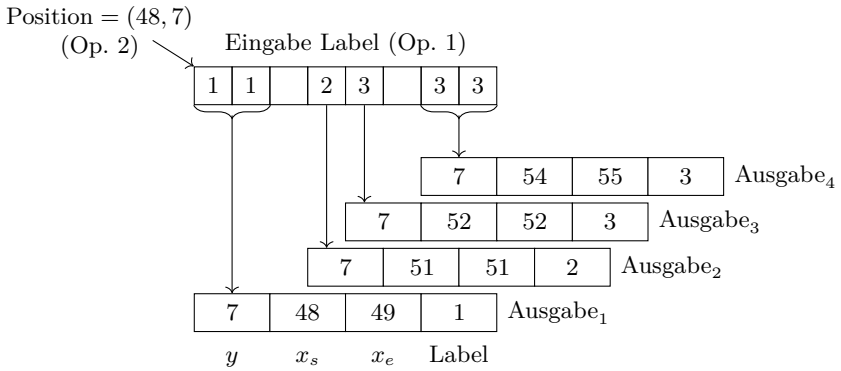


Abbildung 6.22: Detektion von Sequenzen von Pixeln mit gleichem Label

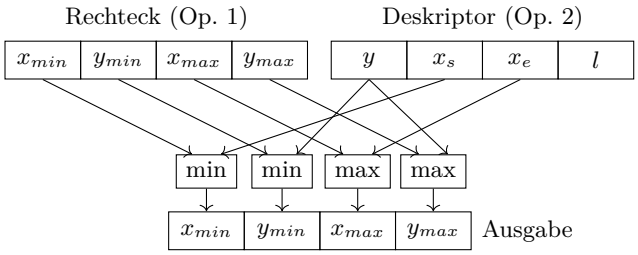


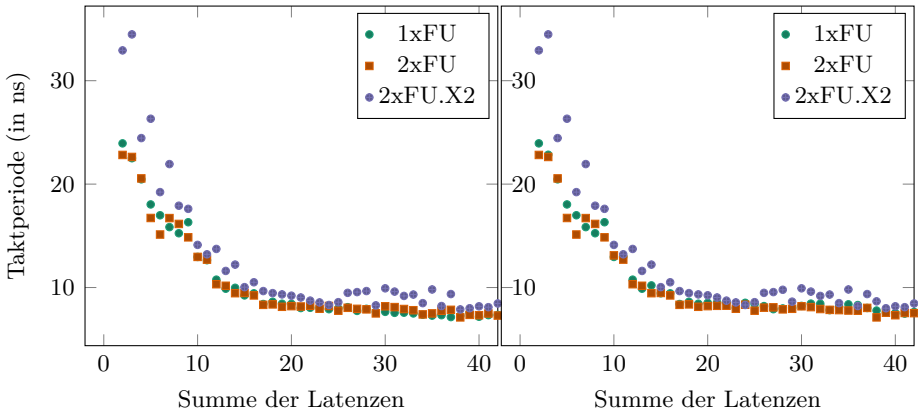
Abbildung 6.23: Aktualisierung der Koordinaten des umschließenden Rechtecks für einen Blob mit einem Deskriptor für eine Sequenz gelabelter Pixel

unterbrochene Sequenzen, wie in der Abbildung für Label 3 gezeigt, werden mehrere Deskriptoren nacheinander ausgegeben. Sobald die Einheit die letzte Sequenz von Pixeln in dem Eingaberegister ausgegeben hat, wird zusätzlich ein Flag im Flag-Register des TUKUTURI generiert, mit dem der aufrufende Code angewiesen wird, das nächste Eingaberegister zu übergeben. Die Aktualisierung eines umschließenden Rechtecks erfolgt dann in einer zweiten RFU, der im ersten Operanden das aktuelle Rechteck und im zweiten Operanden der von der ersten Einheit berechnete Deskriptor übergeben wird. Die Aktualisierung der Koordinaten des Rechtecks erfolgt dann, wie in Abbildung 6.23 gezeigt, durch Auswahl der kleinsten bzw. größten Werte für die Komponenten.

Die **Formklassifikation** verwendet neben dem bereits beschriebenen Coprozessor keine weiteren Erweiterungen.

6.3.2 Profilingergebnisse

Abbildung 6.24 zeigt die minimal benötigte Periode für Konfigurationen mit Befehlssatzerweiterungen. Wie zu erwarten fällt die minimal benötigte Taktperiode mit zunehmender Zahl der Pipelinestufen, bis ab etwa 25 zusätzlichen Stufen



(a) Konfigurationen mit minimaler Periode **(b)** Konfigurationen mit maximalen FPS

Abbildung 6.24: Vergleich der minimalen Taktperiode verschiedener Varianten.

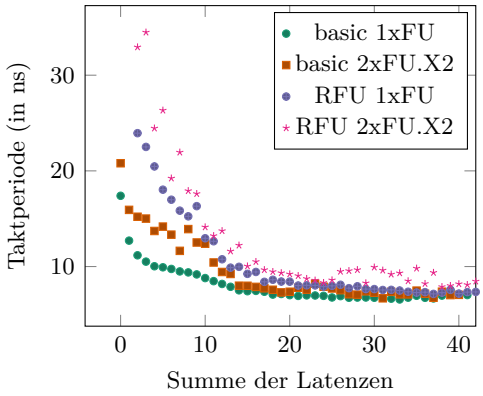
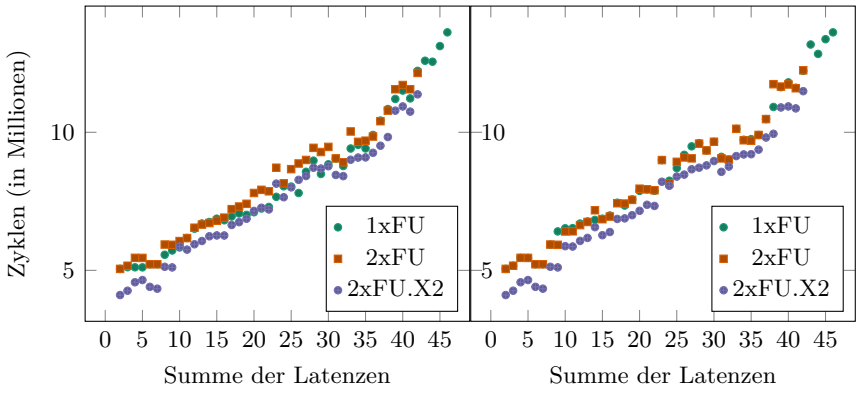


Abbildung 6.25: Vergleich der Implementierungen im Basisinstruktionssatz mit erweitertem Befehlssatz für 1xFU und 2xFU.X2 Varianten.

eine Sättigung eintritt, wenn die funktionalen Einheiten des Prozessors nicht mehr im kritischen Pfad liegen.

Im Vergleich zu den Konfigurationen mit Basisinstruktionssatz (siehe Abbildung 6.25) fällt auf, dass die größeren Varianten 2xFU und 2xFU.X2 nicht mehr so stark von der 1xFU-Variante abweichen. Dies ist damit zu begründen, dass mit den Befehlssatzerweiterungen bereits in der 1xFU-Variante eine hohe Anzahl an funktionalen Einheiten vorhanden ist, wodurch die Anzahl der Forwarding-Pfade stark steigt. Da in den 2xFU-Varianten nur die funktionalen Einheiten aus der Basiskonfiguration verdoppelt werden, sind die Auswirkungen, gemessen an der Gesamtzahl der Pfade gering.

Die hohe Anzahl an Forwarding-Pfaden bedingt einen großen Multiplexer in der Forwarding-Stufe (siehe Abbildung 3.3), wodurch die minimal nötige Taktperiode



(a) Konfigurationen mit minimalen Zyklen **(b)** Konfigurationen mit maximalen FPS

Abbildung 6.26: Taktzyklen zur Verarbeitung eines Frames.

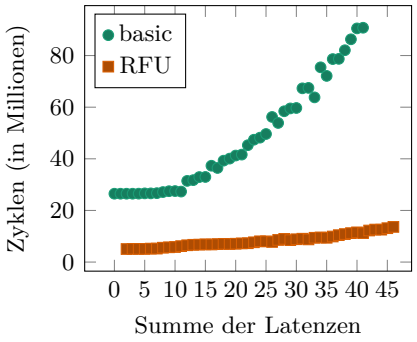
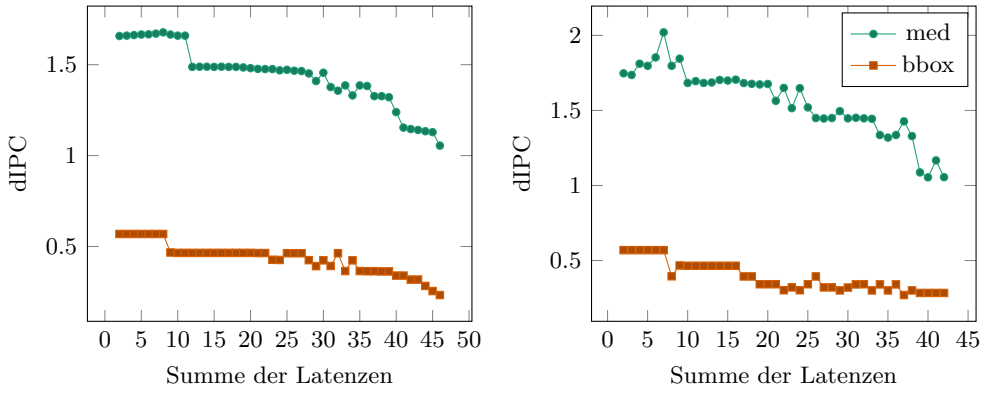


Abbildung 6.27: Vergleich der Taktzyklen für einen Frame im Basisinstruktionssatz und mit Befehlssatzerweiterungen (1xFU-Variante)

im Vergleich zu Konfigurationen mit Basisinstruktionssatz erhöht wird, wie ebenfalls in Abbildung 6.25 zu sehen ist. Nach einer zusätzlichen Pipelinestufe im Forwarding (FWD1) ab einer Summe der Latenzen von etwa 12 erreichen die Konfigurationen mit Befehlssatzerweiterungen nahezu die gleichen Taktperioden wie die Konfigurationen mit Basisinstruktionssatz.

Die Anzahl der benötigten Taktzyklen für die Verarbeitung eines Frames ist in Abbildung 6.26 dargestellt. Diese steigt mit der Anzahl zusätzlicher Pipelinestufen in der TUKUTURI-Architektur wie erwartet an. Die Varianten 1xFU mit einfacher Ausführung jeder funktionalen Einheit und 2xFU mit Verdoppelung der Einheiten aus dem Basisinstruktionssatz liegen dabei nahezu gleich auf. Erst wenn durch Aktivierung des X2-Modus die zusätzlichen Einheiten parallel verwendet werden können, kann der Kompiler einen kompakteren Code generieren, der weniger Taktzyklen benötigt.

Auffällig ist im Vergleich zur Implementierung im Basisinstruktionssatz (siehe



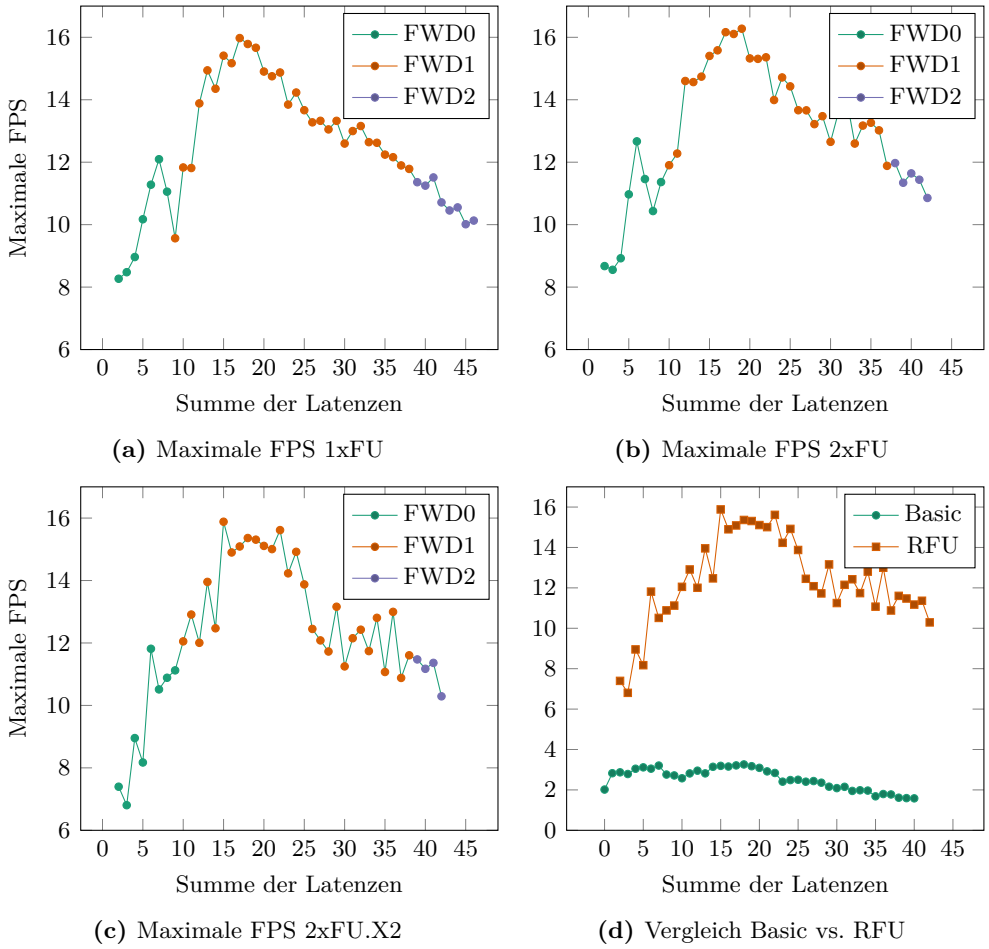
(a) 1xFU Varianten **(b)** 2xFU_X2 Varianten

Abbildung 6.28: Dynamische Instructions Per Cycle (IPC) zweier exemplarischer Routinen der Verkehrszeichendetektion.

Abbildung 6.27), dass die steigende Zahl von Pipelinestufen von Anfang an weniger vom Kompiler ausgeglichen werden kann. Dies lässt sich damit begründen, dass durch die Einführung der anwendungsspezifischen Operationen, die in den meisten Fällen eine ganze Sequenz von Basisinstruktionen ersetzen, der Code dichter gepackt ist und die verbliebenen Operationen stärker voneinander abhängen, so dass der Kompiler weniger Möglichkeiten zum Auffüllen von Wartezyklen aufgrund von gesteigerten Latenzen hat. Zu erkennen ist auch, dass der Anstieg der Taktzyklen mit zunehmender Zahl der Pipelinestufen deutlich flacher ausfällt als im Basisinstruktionssatz, da diese Stufen nur in den Operationen aus dem Basisinstruktionssatz eingefügt wurden, deren Anteil im Code durch die Verwendung der erweiterten Operationen verringert ist.

Die geringere Flexibilität bei der Kompilierung des Anwendungscodes in den Konfigurationen mit Befehlssatzerweiterungen ist auch in Abbildung 6.28 zu sehen, die den IPC für zwei exemplarische Routinen zeigt. Deutlich erkennbar ist die Auswirkung der zusätzlichen Forwarding-Stufe bei einer Summe der Latenzen von 12. Die damit eingeführten Latenzen können aufgrund der dichteren Packung des Codes nicht vollständig ausgeglichen werden.

Durch die stärkere Kompaktierung des Codes bei relativer Ähnlichkeit der minimalen Taktperiode der Prozessor-Konfigurationen kann die Anzahl verarbeiteter Bilder pro Sekunde (FPS) gesteigert werden, wie in Abbildung 6.29 zu sehen. Im Vergleich zu den Konfigurationen mit Basisinstruktionssatz (Abbildung 6.29d) wird die höchste Zahl an FPS hier für Konfigurationen mit einer zusätzlichen Stufe im Forwarding (FWD1) erreicht, da ohne diese Pipelinestufe die Taktperiode des Prozessors, wie oben erklärt (s. Abbildung 6.24), durch die hohe Anzahl funktionaler Einheiten und die damit einhergehende hohe Anzahl an Forwarding-Pfaden erhöht wird.



(a) Maximale FPS 1xFU

(b) Maximale FPS 2xFU

(c) Maximale FPS 2xFU.X2

(d) Vergleich Basic vs. RFU

Abbildung 6.29: Maximal erreichbare Frames pro Sekunde (FPS) für verschiedene TUKUTURI-Konfigurationen in Abhängigkeit der jeweiligen Gesamtlatenz.

Tabelle 6.3 zeigt die Auswertung der Datentransfers für die Implementierung mit Befehlssatzerweiterungen. Durch die Verwendung der Befehlssatzerweiterungen ist die Gesamtzahl der Taktzyklen zur Verarbeitung eines Frames gegenüber der Implementierung mit dem Basisinstruktionssatz gesunken. Da die Menge der zu transferierenden Daten aber gleich geblieben ist, nimmt der Anteil an Wartezyklen und Hintergrundzyklen für DMA Transfers zu. Am stärksten ist dieser Anstieg in der Median-Routine, in der die Kompaktierung des Codes durch die eingeführte Median-Einheit am größten ist. Dort erreicht die Auslastung der zur Verfügung stehenden Taktzyklen für die Datentransfer etwa 87 %. Trotz dieser Kompaktierung können die Transfers in der 1xFU- und der 2xFU-Variante vollständig im Hintergrund durchgeführt werden und führen keine Wartezyklen

Tabelle 6.3: Analyse der Speichertransfers in der Implementierung mit erweitertem Instruktionssatz. Gezeigt sind die Zyklen zur Berechnung (relativ zur Gesamtzahl) sowie die Anzahl der DMA Zyklen (relativ zur Zyklenzahl der einzelnen Routinen).

(a) 1xFU-Variante

Routine	Berechnung	DMA Warten		DMA Transf.
		Init./Fin.	Bg.	Bg.
Gesamt	5038308	13769	0	1744935
Color	33,4538 %	0,0253 %	0 %	13,1909 %
Segment	15,7354 %	0,0990 %	0 %	55,0100 %
Median	7,6155 %	1,8743 %	0 %	87,1823 %
Labeling	21,6727 %	0,2213 %	0 %	55,3885 %
BBox	21,0103 %	0,0949 %	0 %	12,6592 %
Shape	0,2385 %	12,7965 %	0 %	0,3362 %

(b) 2xFU-Variante

Routine	Berechnung	DMA Warten		DMA Transf.
		Init./Fin.	Bg.	Bg.
Gesamt	5037336	13442	0	1714315
Color	33,4624 %	0,0251 %	0 %	13,0173 %
Segment	15,7394 %	0,0976 %	0 %	54,0862 %
Median	7,5982 %	1,8093 %	0 %	84,4816 %
Labeling	21,6782 %	0,2200 %	0 %	54,8896 %
BBox	21,0157 %	0,0949 %	0 %	12,3579 %
Shape	0,2385 %	12,6659 %	0 %	0,3367 %

(c) 2xFU.X2-Variante

Routine	Berechnung	DMA Warten		DMA Transf.
		Init./Fin.	Bg.	Bg.
Gesamt	4289387	72799	15706252	4362186
Color	26,4206 %	0,0371 %	0 %	19,3621 %
Segment	15,5831 %	0,1159 %	0 %	61,4605 %
Median	6,5664 %	3,9696 %	14,8398 %	78,6452 %
Labeling	25,1632 %	0,2226 %	0 %	53,0436 %
BBox	24,3243 %	0,0949 %	0 %	12,5260 %
Shape	0,2721 %	12,9531 %	0 %	0,3406 %

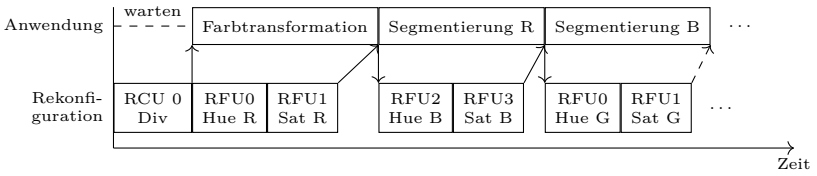


Abbildung 6.30: Schema der Rekonfiguration in der Verkehrszeichenerkennung. Die Rekonfiguration der Einheiten erfolgt im Hintergrund parallel zur Berechnung von Teilaufgaben.

ein. Nur in der 2xFU.X2-Variante ist die Kompaktierung so groß, dass in der Median-Routine nicht mehr genügend Berechnungszyklen vorhanden sind, um alle Datentransfers im Hintergrund auszuführen, wodurch hier auf die Datentransfers gewartet werden muss.

6.4 TUKUTURI-Implementierung mit partieller Rekonfiguration

Insgesamt werden für die Verkehrszeichenerkennung auf dem TUKUTURI 12 RFUs und 2 RCUs verwendet, wie in Abschnitt 6.3.1 beschrieben. Da diese Einheiten und die bereitgestellten Operationen spezifisch für die sequenziellen Teilaufgaben des Algorithmus sind, werden maximal zwei RFUs bzw. ein Coprozessor parallel verwendet. Die Komplexität der TUKUTURI-Architektur kann also verringert werden, indem die Einheiten dynamisch rekonfiguriert werden, anstatt alle Einheiten statisch einzubinden. Dazu werden vier Slots für RFUs und ein Slot für eine RCU bereitgestellt, so dass die Rekonfiguration von RFUs für die nächste Teilaufgabe durchgeführt werden kann, während andere Einheiten für die aktuelle Teilaufgabe verwendet werden.

6.4.1 Rekonfigurationsschema

Die Übertragung der partiellen Bitstreams an das ICAP-Modul wird vom DMA-ICAP-Controller durchgeführt. Dieser wird vom Programmierer durch Assemblerinstruktionen zum Beschreiben der Konfigurationsregister programmiert und verwaltet eine Warteschlange von Rekonfigurationen, wie in Abschnitt 3.4.1 beschrieben. Damit ist die Rekonfiguration parallel und unabhängig von den Berechnungen des TUKUTURI und auch von Datentransfers via Daten-DMA. Das Schema der Rekonfiguration ist in Abbildung 6.30 dargestellt.

Zunächst wird der Coprozessor für die Division rekonfiguriert, der in der ersten Routine verwendet wird. Ist die Rekonfiguration abgeschlossen, kann die Farbraumtransformation starten. Gleichzeitig wird die Rekonfiguration der

Einheiten für die Transferfunktionen in der Segmentierung nach roten Pixeln in den ersten beiden RFU-Slots gestartet. Wenn die Rekonfiguration dieser Einheiten abgeschlossen ist, bevor die Farbraumtransformation endet, kann die Segmentierung ohne Wartezeit starten. Parallel zur Segmentierung nach roten Pixeln werden die Einheiten zur Segmentierung nach blauen Pixeln in den letzten beiden Slots gestartet. Zu Beginn der Segmentierung nach blauen Pixeln werden die Einheiten in den ersten beiden Slots nicht mehr verwendet und können durch Einheiten zur Segmentierung nach gelben Pixeln rekonfiguriert werden. Auf diese Weise werden die vier RFU-Slots kontinuierlich rekonfiguriert, so dass während der Bearbeitung einer Teilaufgabe der Anwendung die Einheiten für die nächste Teilaufgabe im Hintergrund rekonfiguriert werden und nicht mehr benötigte Einheiten ersetzt werden. Analog wird während der Berechnung des Medianfilters zusätzlich zu den beiden RFUs der Coprozessor für das Connected Component Labeling rekonfiguriert. Nach dem Ende des Connected Component Labeling wird die RCU wieder durch den Coprozessor für die Division ersetzt, damit dieser bei der Berechnung der DtB-Vektoren für die Formklassifikation verwendet werden kann.

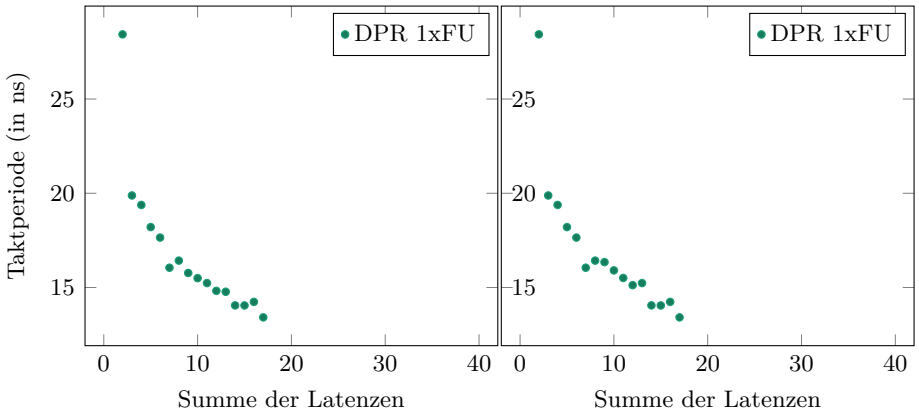
Die Dauer der Rekonfiguration ist abhängig von der Größe der zu rekonfigurierenden Bereiche (PRR, siehe Abschnitt 2.1.2). In Tabelle 6.4 sind die Größen der funktionalen Einheiten des Prozessors für die Minimalkonfiguration und ohne zusätzliche Pipelinestufen angegeben. Der erste Abschnitt der Tabelle enthält Angaben für die nicht rekonfigurierten Standard-Einheiten zum Vergleich. Die Angaben im zweiten Abschnitt der Tabelle geben Größe und Ressourcenbedarf für die rekonfigurierbaren Einheiten an. Die PRR, in die diese Einheiten platziert werden, umfassen für die Coprozessoren 1040 CLBs und für die funktionalen Einheiten 640 CLBs. In der Tabelle ist außerdem angegeben, welcher Anteil der aufgrund der Größe der Bereiche theoretisch zur Verfügung stehenden Ressourcen von den jeweiligen Einheiten verwendet wird. Der relativ große Overhead in den CLB-Größen für die PRR ist einerseits dadurch bedingt, dass die Größe der Bereiche immer in Vielfachen von 80 CLBs angegeben werden muss. Der weitaus wichtigere Grund ist jedoch, dass der Overhead für das Routing innerhalb der Bereiche benötigt wird und die Synthese mit kleineren Bereichen nicht immer gelingt. Außerdem werden für die Konfigurationen mit zusätzlichen Pipelinestufen zusätzliche Ressourcen verwendet. Die Größe der PRR ist in dieser Anwendung nicht kritisch, da die Anwendung nicht auf die Rekonfiguration warten muss (siehe Abschnitt 6.4.3). Daher wurde auf eine Optimierung der Größe der PRR verzichtet. Unter optimalen Bedingungen kann aus der Größe der rekonfigurierbaren Einheiten und den Spezifikationen des ICAP-Moduls (siehe Tabelle 2.1) die Anzahl der zur Rekonfiguration benötigten Zyklen berechnet werden. Für die rekonfigurierbaren funktionalen Einheiten sind dies 23 750 Zyklen und für die Coprozessoren 38 412 Zyklen.

Tabelle 6.4: Größen und Ressourcenbedarf der funktionalen Einheiten. Für rekonfigurierbare Einheiten ist zusätzlich der Anteil der verwendeten Ressourcen an der Größe der rekonfigurierbaren Bereiche angegeben.

Einheit	CLBs	Register	LUTs
AU	50	0	428
BLU	12	0	124
CMM	11	0	140
MMU	361	157	1578
PER	85	0	585
SRU	423	0	1457
Divider CP	508	594 (7,1 %)	1577 (37,9 %)
Labeling CP	308	706 (8,5 %)	1188 (28,6 %)
LUT H Rot	166	0 (0 %)	747 (29,2 %)
LUT S Rot	13	0 (0 %)	158 (6,2 %)
LUT H Blau	75	0 (0 %)	392 (15,3 %)
LUT S Blau	40	0 (0 %)	352 (13,8 %)
LUT H Gelb	8	0 (0 %)	119 (4,6 %)
LUT S Gelb	29	0 (0 %)	241 (9,4 %)
Median	121	320 (6,3 %)	449 (17,5 %)
Labeling 1	16	72 (1,4 %)	380 (14,8 %)
Labeling 2	61	121 (2,4 %)	265 (10,4 %)
BBox 1	25	6 (0,1 %)	155 (6,1 %)
BBox 2	10	0 (0 %)	95 (3,7 %)

6.4.2 Auswahl der Konfigurationen

Die Hardwaresynthese für Konfigurationen für dynamische Rekonfiguration ist deutlich aufwendiger als die Synthese statischer Konfigurationen. Daher wurde für die im Folgenden präsentierte Auswertung eine kleinere Menge von Konfigurationen ausgewählt. Diese leiten sich von den Konfigurationen mit erweitertem Befehlssatz ab, die ohne zusätzliche Forwarding-Stufe kleine Taktperioden bzw. hohe FPS erreichen. Zu jeder RFU-Konfiguration wurde eine DPR-Konfiguration abgeleitet, die identische Parameterwerte für alle Standardinstruktionen besitzt und die in den RFUs keine zusätzlichen Pipelinestufen verwendet. Zusätzlich wurden alle RFUs mit verschiedenen Anzahlen zusätzlicher Pipelinestufen synthetisiert. Aus der Grundkonfiguration ohne zusätzliche Stufen in den RFUs wurden dann verschiedene Kombinationen aus Basisprozessor und RFUs ausgewählt, die geringere Taktperioden erreichen.



(a) Konfigurationen mit minimaler Periode **(b)** Konfigurationen mit maximalen FPS

Abbildung 6.31: Vergleich der minimalen Taktperiode verschiedener Varianten.

6.4.3 Profilingergebnisse

Die minimal benötigte Taktperiode für Konfigurationen mit dynamischer Rekonfiguration ist in Abbildung 6.31 gezeigt. Auch in dieser Implementierung sinkt die minimal benötigte Taktperiode mit steigender Anzahl eingefügter Pipelinestufen.

Der Vergleich mit den statischen Konfigurationen in Abbildung 6.32 zeigt, dass die Konfigurationen mit dynamischer Rekonfiguration nur selten (Latenzen 3 und 4) geringere Taktperioden erreichen als die statischen Konfigurationen. Dies kann dadurch erklärt werden, dass die Synthese im Falle der DPR stärker eingeschränkt ist, da alle Einheiten in die definierten PRR synthetisiert werden müssen. Eine Optimierung der Platzierung und des Routing könnte manuell erfolgen, z. B. durch Area-constraints, dies wurde hier aber nicht durchgeführt.

Der Assemblercode für die Konfigurationen mit dynamischer Rekonfiguration ist – zumindest innerhalb der algorithmischen Routinen – der gleiche wie in den statischen Konfigurationen mit Befehlssatzerweiterungen, einzig die Programmierung des ICAP-DMA-Moduls zwischen den Aufrufen der Routinen ist hinzugekommen.

Der Vergleich der benötigten Taktzyklen von statischen und dynamischen Konfigurationen ist in Abbildung 6.33 dargestellt. Da in den Konfigurationen mit dynamischer Rekonfiguration Latenzen eher in den RFUs eingeführt wurden, die seltener eingesetzt werden als die Standardinstruktionen, steigt die Codegröße nicht so stark, wie in den statischen Konfigurationen.

Abbildung 6.34 zeigt die maximal erreichten FPS für Konfigurationen mit und ohne dynamische Rekonfiguration im Vergleich. Zu erkennen ist, dass die Konfigurationen mit DPR höhere FPS erreichen als die statischen Konfigurationen, solange diese keine zusätzliche Forwarding-Stufe verwenden. Die hohe

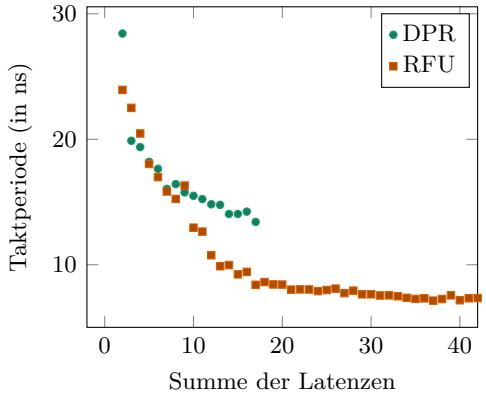


Abbildung 6.32: Vergleich der Implementierungen mit erweitertem Befehlssatz (RFU) und mit dynamischer Rekonfiguration (DPR).

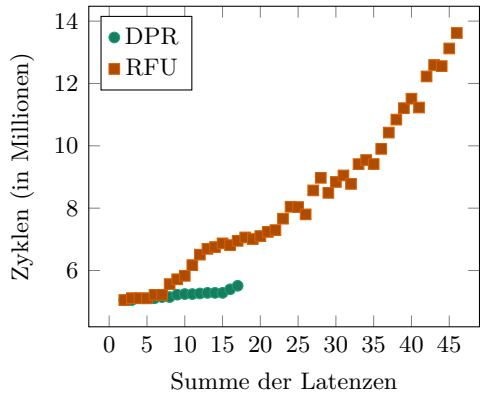


Abbildung 6.33: Vergleich der Taktzyklen für einen Frame mit Befehlssatzerweiterungen (RFU) und mit dynamischer Rekonfiguration (DPR).

Anzahl funktionaler Einheiten in den statischen Konfigurationen (RFU) führt zu einer hohen Anzahl an Forwardingpfaden, deren negativer Einfluss auf das Routing und die minimale Periode durch eine zusätzliche Forwardingstufe verringert wird. In den Konfigurationen mit dynamischer Rekonfiguration tritt das Problem der vielen Forwardingpfade nicht auf, da die Anzahl der funktionalen Einheiten geringer ist. Daher können diese Konfigurationen für FWD0 höhere FPS erreichen.

Die Auswertung der Speichertransfers für die Implementierung mit dynamischer partieller Rekonfiguration ist in Tabelle 6.5 gezeigt. Hier ist neben den bereits für die Implementierungen im Basisinstruktionssatz bzw. für die statischen Konfigurationen mit Befehlssatzerweiterungen beschriebenen Spalten für die Daten-DMA-Transfers auch eine Spalte für die Anzahl der Taktzyklen, in denen

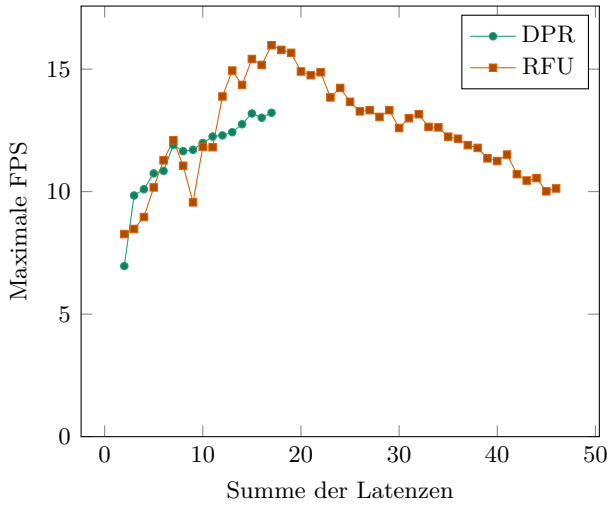


Abbildung 6.34: Vergleich der maximal erreichten FPS von Konfigurationen mit erweitertem Befehlssatz mit und ohne dynamische Rekonfiguration.

Tabelle 6.5: Analyse der Speichertransfers in der Implementierung mit dynamischer Rekonfiguration. Gezeigt sind die Zyklen zur Berechnung (relativ zur Gesamtzahl) sowie die Anzahl der DMA Zyklen (relativ zur Zyklenzahl der einzelnen Routinen).

Routine	Berechnung	DMA Warten		DMA Transf.	DMA ICAP
		Init./Fin.	Bg.	Bg.	Bg.
Gesamt	5038191	13074	0	1683461	361927
Color	33,4592 %	0,0249 %	0 %	12,8619 %	2,8100 %
Segment	15,7379 %	0,0966 %	0 %	53,3456 %	17,9097 %
Median	7,6167 %	1,7290 %	0 %	81,3859 %	21,9466 %
Labeling	21,6761 %	0,2169 %	0 %	54,0979 %	4,3292 %
BBox	21,0135 %	0,0949 %	0 %	12,1586 %	3,6160 %
Shape	0,2320 %	12,8460 %	0 %	0,3453 %	0,0000 %

das DMA-ICAP-Modul eine Rekonfiguration vornimmt, angegeben. Für die einzelnen Routinen ist diese Zahl wieder auf die Gesamtzahl der Taktzyklen in der jeweiligen Routine bezogen. Diese Angabe ist nicht ganz exakt, da die DMA-ICAP-Transfers nicht wie die Daten-DMA-Transfers in den Routinen isoliert sind. Stattdessen wird *vor* dem Beginn einer Routine der DMA-ICAP-Transfer für die darauf folgende Routine gestartet. Wenn im Idealfall die Rekonfiguration im Hintergrund, also parallel zur Verarbeitung der Daten erfolgt, muss am Beginn einer Routine nicht auf die Rekonfiguration der darin verwendeten funktionalen Einheiten gewartet werden (vgl. Abbildung 6.30).

Bei den Daten-DMA-Transfers gibt es, wie schon für die Implementierungen mit dem Basisinstruktionssatz beschrieben, Wartezyklen bei der Initialisierung/Finalisierung am Anfang/Ende der Routinen. Davon abgesehen können alle Datentransfers im Hintergrund ausgeführt werden, da ausreichend Berechnungszyklen zur Verfügung stehen, zu denen die Datentransfers parallel ablaufen können. Die maximale Auslastung der zur Verfügung stehenden Taktzyklen wird in der Median-Routine mit etwa 81 % erreicht.

Die letzte Spalte der Tabelle zeigt, dass die Datentransfers der partiellen Bitstreams zur Rekonfiguration der funktionalen Einheiten und des Coprozessors nur einen relativ geringen Anteil der zur Verfügung stehenden Taktzyklen ausnutzen. Wie schon in Abbildung 6.30 gezeigt, muss zu Beginn des Algorithmus auf die Rekonfiguration des Divisions-Coprozessors gewartet werden, der in der Farbraumtransformation verwendet wird. Dies geschieht allerdings nur vor dem ersten Frame[2]. Während der Verarbeitung wird der Divisions-Coprozessor durch einen Coprozessor für das Connected Component Labeling ersetzt. Dieser wird im weiteren Verlauf allerdings wieder durch den Coprozessor für Division ersetzt, da dieser in der Formklassifikation verwendet wird. Daraus ergibt sich, dass für den gesamten Algorithmus (Verarbeitung von 300 Frames) nur 38420 Taktzyklen auf die erste Konfiguration des Coprozessors für Division gewartet werden muss (im Vergleich zu den theoretisch benötigten 38 412 Zyklen im optimalen Fall, siehe Abschnitt 6.4.1). Alle anderen Rekonfigurationen laufen vollständig im Hintergrund ab.

6.5 Implementierung auf MIPS Softcore Prozessor

In [73] wurde eine Softcore-Implementierung eines selbst rekonfigurierbaren MIPS-Prozessors vorgestellt, dessen Architektur im folgenden Abschnitt näher erläutert wird. Die Verkehrszeichendetektion ist in [46] auf diesen Prozessor übertragen worden. Dazu wurde eine optimierte C++-Implementierung der Anwendung sowie eine Hardwareerweiterung zur Beschleunigung der Median-Filterung erstellt, die in Abschnitt 6.5.3 erklärt wird. Evaluationsergebnisse sind in Abschnitt 6.5.4 angegeben.

6.5.1 MIPS Softcore Prozessorarchitektur

Für die Implementierung des MIPS-Softcores wurde die MIPS32r2 Basisarchitektur als Grundlage verwendet. Diese wurde durch Daten- und Instruktionscaches sowie durch Anbindungen an einen lokalen IO-Bus erweitert. Der interne Multiplizierer wurde für die Umsetzung in einem DSP-Slice des FPGA optimiert [73].

[2] Die Benchmarks, die hier präsentiert werden, verarbeiten jeweils 300 Frames

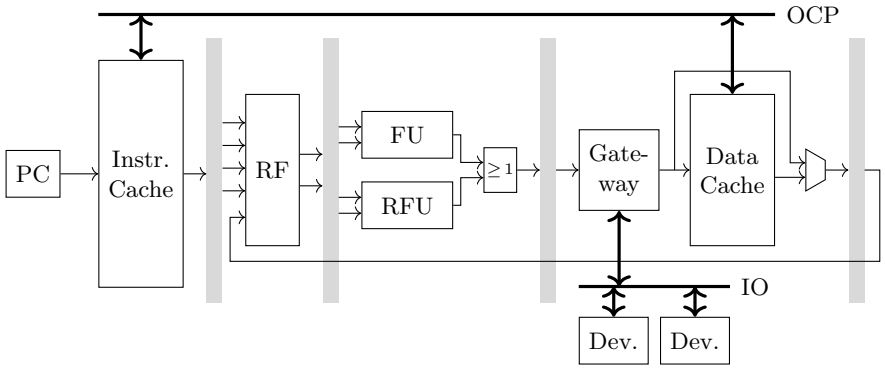


Abbildung 6.35: Vereinfachte Darstellung der modifizierten Pipelinestruktur des MIPS Softcore Prozessors

Der Prozessor verwendet eine klassische 5-stufige Pipeline wie in Abbildung 6.35 dargestellt, die keine Forwarding-Pfade oder Hazard-Detektion enthält. Die Verantwortlichkeit zum Verhindern von Konflikten in der Pipeline liegt beim Kompiler, der durch Einfügen von Wartezyklen (`NOP`) oder Delay-Slots nach Sprungbefehlen Konflikte auflösen muss. Der Prozessor enthält getrennte Daten- und Instruktionscaches, die über einen OCP-Bus auf einen einzigen Hauptspeicher zugreifen. Die Prozessorimplementierung verwendet, wie schon der TUKUTURI, das Emulationsframework UEMU (siehe Abschnitt 5.4), welches einem Hostcomputer Zugriff auf den Hauptspeicher über eine Ethernet Schnittstelle ermöglicht. Zusätzlich steht ein IO-Bus zur Verfügung, über den zusätzliche Module wie Coprozessoren (*RCUs*) in den lokalen Adressraum abgebildet werden und per `LOAD` und `STORE` Befehl angesprochen werden. Das Gateway ist für die Umsetzung der Adressen zuständig. Die Signale der CPU zum Zugriff auf den IO-Bus werden gepuffert, so dass für jede Anfrage an den Bus ein Delay von 2 Zyklen entsteht. Die Synchronisierung zwischen CPU und IO-Device erfolgt dadurch, dass ein Device die Annahme von Eingabedaten oder die Bereitstellung von Ausgabedaten mit einem ACK-Signal bestätigt. Bis zum Eintreffen dieses Signals wird die CPU angehalten.

Da der CPU-interne Zähler für CPU-Zyklen ebenfalls angehalten wird, wenn die CPU beispielsweise durch I/O-Operationen blockiert wird, wird ein zusätzliches IO-Device bereitgestellt, mit dem die Zahl der Gesamtzyklen (Sys-Zyklen) auch bei angehaltener CPU gezählt werden kann. Dieser Zähler kann zum Profiling von Anwendungen gelesen werden.

Die Architektur kann auch durch zusätzliche funktionale Einheiten in der Pipeline (*rekonfigurierbare funktionale Einheit (RFU)*) erweitert werden, über die anwendungsspezifische Instruktionen bereitgestellt werden können. Die in solchen Einheiten bereitgestellten Operationen können durch Erweiterung des LLVM-

Kompiler Backends oder durch intrinsische Funktionen im Anwendungscode verwendet werden.

Für RFUs und RCUs unterstützt der MIPS Softcore auch dynamische partielle Rekonfiguration, die in der hier verwendeten Implementierung der Verkehrszeichendetektion allerdings nicht genutzt wird, da nur eine Erweiterung vorgenommen wurde.

6.5.2 Implementierung der Verkehrszeichendetektion

Die Implementierung der Verkehrszeichendetektion in C++ erfolgte ausgehend von der Referenzimplementierung des Verfahrens und wurde für die Umsetzung auf dem MIPS optimiert. Gleitkommaberechnungen wie die Normierung der DtB-Vektoren (siehe Abschnitt 6.1.5) und die Auswertung der SVMs zur Formklassifikation wurden durch Festkomma-Berechnungen ersetzt.

Im Speicher, der hier 1 GB hält, wird das kompilierte Programm, das Eingangsbild, die Zwischenergebnisse und der Stack abgelegt. Weiterhin liegt dort ein Puffer zur Speicherung von Log-Einträgen, die für das Profiling genutzt werden. Bestimmte festgelegte Speicheradressen werden zur Kommunikation mit einem Hostprogramm über die UEMU-Schnittstelle verwendet.

6.5.3 Hardwarebeschleuniger für Median-Filterung

Erste Laufzeitanalysen ergaben, dass die Berechnung des Medianfilters mit Abstand die aufwendigste Routine der Verkehrszeichendetektion ist. Daher wurde in [46] ein Hardwarebeschleuniger als IO-Device implementiert, der für ein Eingangsbild von 640×480 Pixeln ein 5×5 Medianfilter berechnet. Dieses ist in Abbildung 6.36 schematisch dargestellt. Der Beschleuniger iteriert nicht über die Pixel im Filterfenster, sondern verarbeitet die 25 Eingabepixel parallel. Dazu wird ein Schieberegister verwendet, in das die Pixel des Bildes sequentiell eingegeben werden, wobei die Bildzeilen sequentiell verkettet sind. Das Schieberegister muss dazu vier ganze Bildzeilen und eine Breite des Filterfensters, also $l = 4 \cdot 640 + 5 = 2565$ Pixel enthalten.

Aus dem Schieberegister werden die 25 Pixel für die Median-Berechnung parallel abgegriffen und in einen Addiererbaum geleitet. Liegt die Summe der (binären) Pixel über 12, ist die Ausgabe 1, sonst 0. Nach Eingabe von l Pixeln liegt das erste Ergebnis vor, danach wird nach jedem weiteren Pixel das Filterfenster um eine Stelle nach rechts verschoben und ein Ergebnis generiert. Wenn $640 - 5$ Pixel eingegeben wurden, liegt die Filtermaske am rechten Bildrand und es müssen 5 weitere Pixel eingegeben werden, bis die Filtermaske am Anfang der nächsten Zeile liegt und das nächste Ergebnis generiert wird. Damit die Softwareroutine, die den Hardwarebeschleuniger ansteuert, diesen Sonderfall nicht behandeln muss und einen einfacheren Kontrollfluss verwenden kann, enthält

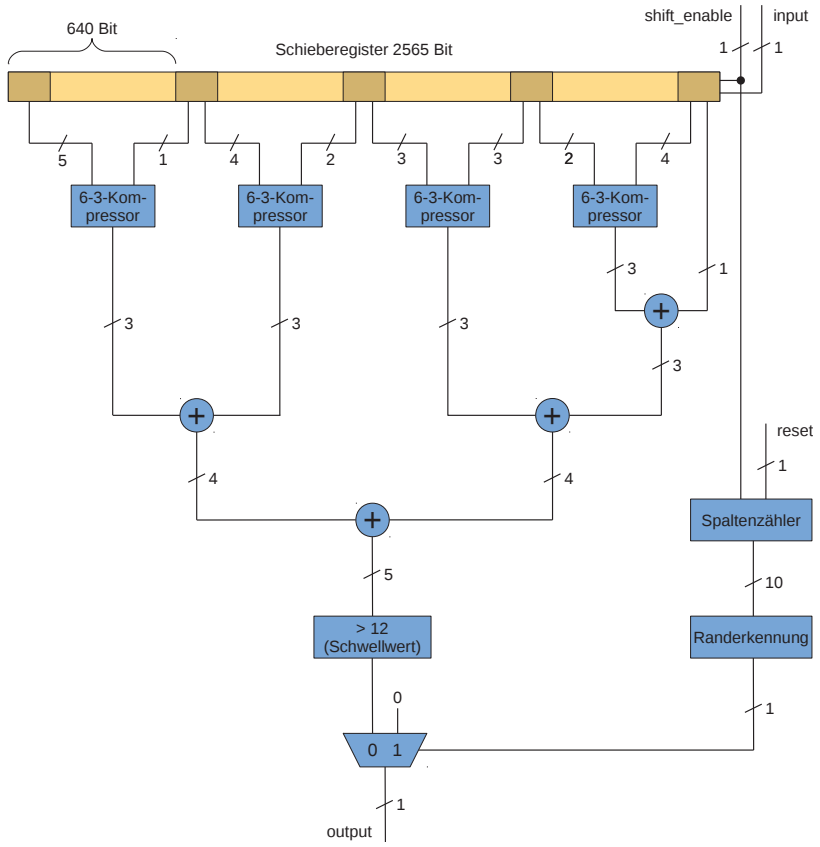


Abbildung 6.36: Schematische Darstellung des Hardwarebeschleunigers für die Median-Filterung [46]

der Hardwarebeschleuniger selbst einen Zähler und gibt am Ende einer Zeile 4 schwarze Pixel (Wert 0) aus und das nächste Median-Ergebnis mit dem 5. Pixel. Die Randzeilen über und unter dem Ergebnisbild werden vom Beschleuniger nicht generiert, sondern können durch Speicherung der Ergebnispixel in einen mit Nullen initialisierten Speicherbereich erreicht werden. Damit hat das Ergebnisbild der Median-Filterung wieder die identische Größe wie das Eingangsbild.

Zur Ansteuerung des Hardwarebeschleunigers werden Schreib- und Lesebefehle verwendet. Jedem IO-Device stehen dafür eine bestimmte Anzahl eigener Adressen zur Verfügung. Eine Adresse wird für das Zurücksetzen der Einheit, zum Beispiel am Anfang eines neuen Bildes verwendet. Ein Schreibvorgang auf die zweite Adresse übergibt ein Eingangspixel an den Beschleuniger, das Lesen von dieser Adresse liefert ein Ergebnispixel.

Tabelle 6.6: Profilingergebnisse für die MIPS Softcore Prozessorimplementierung der Verkehrszeichendetektion für einen Frame [46]. CPU-Zyklen werden nicht gezählt, wenn die CPU angehalten wird, Sys-Zyklen geben die Gesamtzahl an.

Routine	**Sys-Zyklen**	**CPU-Zyklen**
Farbraumtransformation	37 635 292	36 396 691
Segmentierung R	29 566 042	28 824 624
Segmentierung B	29 227 272	28 526 913
Segmentierung Y	33 607 130	32 897 713
Summe Segmentierung	92 400 444	90 249 250
Medianfilter (SW)	315 676 438	314 278 755
Segmentierung + Median (HW)	103 558 642	97 598 947
Medianfilter (HW)	11 158 198	7 349 697
Labeling R	10 759 490	9 656 191
Labeling B	15 752 188	14 124 366
Labeling Y	14 051 206	12 520 261
Summe Labeling	40 562 884	36 300 818
Bounding Box R	7 187 428	4 997 916
Bounding Box B	7 784 434	5 546 687
Bounding Box Y	7 614 494	5 368 660
Summe Bounding Box	22 586 356	15 913 263
Formklassifikation	329 286	53 013
DtB Vektoren	284 174	45 983
Klassifikation	45 112	7030
Gesamt (SW-Median)	509 190 700	493 191 790
Gesamt (HW-Median)	204 672 460	186 262 732

6.5.4 Evaluation

Für die Auswertung der Verkehrszeichendetektion auf dem MIPS Softcore Prozessor verwendet die Implementierung den oben erwähnten Puffer für Log-Einträge, mit dem von der Anwendung ein Debug-String (z. B. der Methodenname) und die aktuellen Zählerstände der CPU- bzw. Sys-Zyklen gespeichert werden können. Die Anzahl der Sys-Zyklen ist mindestens so hoch wie die Zahl der CPU-Zyklen. Im Idealfall, d. h., wenn die CPU nie angehalten würde, wären die Anzahlen gleich. Dieser Fall tritt jedoch nicht ein, so dass die CPU-Zyklen immer kleiner sind

als die Sys-Zyklen. Die Differenz kann als die Anzahl der CPU-Stalls verwendet werden. Die Zählerstände können vom Anwendungsprogramm auch zurückgesetzt werden, beispielsweise am Anfang einer Methode. Über die Ethernet-Anbindung des UEMU-Frameworks kann das Hostprogramm die Informationen nach Abschluss der Berechnung abrufen. Das System wird hier auf einem Xilinx Virtex-6 FPGA emuliert und mit einer Frequenz von 100 MHz getaktet.

Bei Verwendung der Softwareimplementierung des Median-Filters wird das segmentierte Eingangsbild nicht linear gelesen, da über die Pixel für alle Positionen des Filterfensters iteriert wird. Daher schreibt die Segmentierungsroutine das Ergebnis in den RAM und die Medianroutine liest es von dort. Wird hingegen der Hardwarebeschleuniger verwendet, nimmt dieser die Pixel des segmentierten Bildes sequentiell entgegen, weshalb eine Zwischenspeicherung nach der Segmentierung nicht nötig ist. Daher wurden in diesem Fall die Segmentierung und die Median-Filterung zusammengefasst, um den Overhead durch das Speichern und das erneute Einlesen des Bildes zu vermeiden.

Die Profilingergebnisse sind in Tabelle 6.6 angegeben und werden in den folgenden Punkten erläutert.

- Es gibt einen deutlichen Unterschied in der Anzahl der CPU-Stalls zwischen Routinen, die Pixel sequentiell verarbeiten (Farbraumtransformation, Segmentierung, Medianfilter, Labeling) und den Routinen, die Pixel nichtsequentiell verarbeiten (Bounding Box, Formklassifikation). Wie bei einem Cache-basierten System zu erwarten ist, liegt die Zahl der CPU-Stalls für sequentiellen Speicherzugriff deutlich unter der Zahl für nicht sequentiellen Zugriff.

- Die hohe Anzahl von CPU-Stalls bei Verwendung des Hardwarebeschleunigers für die Median-Filterung ist damit begründet, dass die Schreib- und Lesezugriffe auf IO-Devices aufgrund der im Bussystem verbauten Register jeweils zwei Taktzyklen warten müssen. Trotz dieser Wartezyklen ist die Summe der Taktzyklen für Segmentierung und softwarebasierten Median höher als die Zahl der Zyklen für die Segmentierung mit hardwarebasiertem Median.

- Durch Verwendung des Hardwarebeschleunigers für die Median-Filterung wird in dieser Routine eine Beschleunigung in Berechnungszyklen (CPU) um den Faktor 42,8 und in Ausführungszyklen (Sys-Zyklen) um den Faktor 28,3 erreicht. Die Gesamtanwendung wird dadurch um den Faktor 2,65 (CPU) bzw 2,49 (Sys) beschleunigt. Dies entspricht einer Steigerung der erreichten Frames pro Sekunde von 0,2 auf 0,489.

Der Bedarf des Systems an FPGA-Ressourcen ist in Tabelle 6.7 angegeben. Für jede Ressource ist die zur Verfügung stehende Anzahl angegeben. Das Gesamtsystem besteht aus dem UEMU-Framework, dem MIPS Softcore Prozessor und dem

Tabelle 6.7: Ressourcenbedarf des MIPS Softcore Prozessors und des Hardwarebeschleunigers [46]

Ressource		Gesamtsystem		HW-Beschl.		Zuwachs
LUT	(150 720)	22 442	(14,9 %)	133	(0,0882 %)	0,596 %
Distr. RAM	(58 400)	2472	(4,23 %)	80	(0,137 %)	3,34 %
Flipflop	(301 440)	15 703	(5,21 %)	49	(0,0163 %)	313 %
Block-RAM	(416)	46	(11,1 %)	0	(0 %)	0 %
DSP	(768)	8	(1,04 %)	0	(0 %)	0 %

Hardwarebeschleuniger für die Median-Filterung. Der Anteil der Ressourcen, die auf dem Hardwarebeschleuniger entfallen und den Zuwachs, den dieser Anteil am Gesamtsystem ausmacht, ist ebenfalls angegeben.

Insgesamt ist der Anteil der Ressourcen, die für den Hardwarebeschleuniger eingesetzt werden, sehr klein. Der größte Anteil entfällt auf die distributed RAMs, da diese zur Umsetzung des Schieberegisters für die Pixel verwendet werden. Ein distributed RAM kann als 32-Bit-Schieberegister verwendet werden, das einen 1-Bit-Ausgang am Ende und einen weiteren an einer wählbaren Position bereitstellt. In dem Teil des Schieberegisters, der unter dem Filterfenster liegt, müssen konsekutive Positionen gelesen werden, weshalb hier Flipflops verwendet werden.

6.6 Implementierung auf Texas Instruments C6748-DSP

Der TM320C6748 DSP von Texas Instruments [99] ist ein low-power, fixed- und floating-point Prozessor basierend auf dem C674x DSP-Kern mit 456 MHz. Die Prozessorarchitektur wird im nächsten Abschnitt genauer erläutert. Für die Implementierung der Verkehrszeichendetektion wurde die C++-Referenzimplementierung herangezogen und für den TI-Prozessor optimiert. Die Evaluation dieser Implementierung ist in Abschnitt 6.6.2 angegeben.

6.6.1 C6748-Architektur

Die C674x DSP Architektur ist in Abbildung 6.37 gezeigt. Der Prozessor enthält zwei separate Datenpfade, die jeweils vier funktionale Einheiten und 32 32-bit-Register enthalten. Die folgende Auflistung beschreibt die relevanten Aspekte dieser Architektur [100].

- Funktionale Einheiten aus Datenpfad A schreiben in Registerbank A, die Einheiten aus Datenpfad B schreiben in Registerbank B. Da jede Einheit

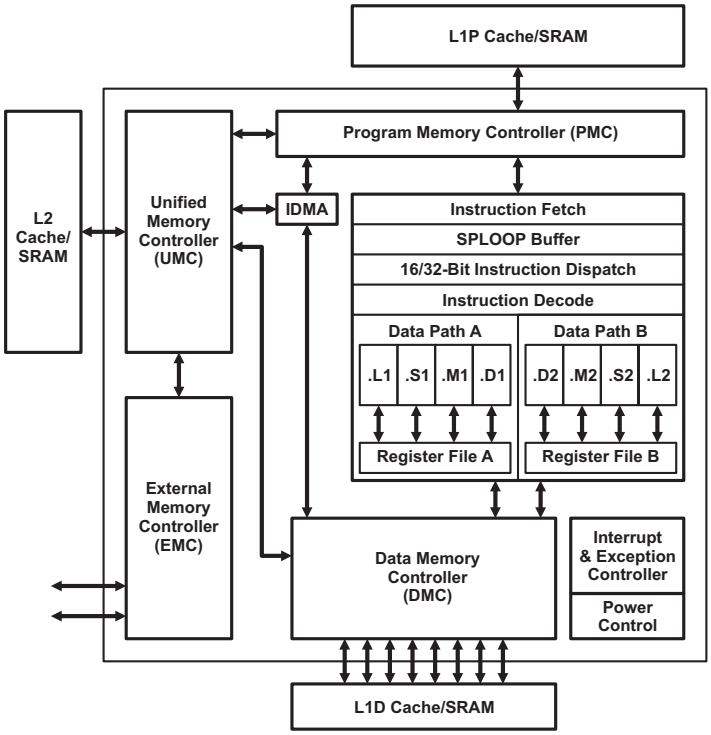


Abbildung 6.37: C674x DSP Blockdiagramm [100]

einen eigenen Schreibport auf die Register besitzt, können bis zu acht Operationen gleichzeitig ausgeführt werden (VLIW).

- Der Prozessor enthält zwei *cross-paths*, die eine Registerbank mit dem jeweils gegenüberliegenden Datenpfad verbinden. So kann ein Operand einer funktionalen Einheit aus Datenpfad A aus Registerbank B gelesen werden und umgekehrt. Soll ein Register, das im vorherigen Zyklus geschrieben wurde, über einen solchen cross-path gelesen werden, wird durch die Hardware automatisch ein Wartezyklus eingefügt.

- Für den Speicherzugriff stehen pro Registerbank jeweils ein Lese- und ein Schreibpfad zur Verfügung, über die 64-bit Daten gelesen bzw. geschrieben werden können. Gleichzeitiges Schreiben in eine Registerbank von zwei Ladeoperationen ist nicht möglich.

- Der Prozessor unterstützt konditionale Ausführung für die meisten Operationen. Dazu kann eines von sechs Registern (A0-A2, B0-B2) und die Bedingung *null* oder *nicht null* ausgewählt werden. Zu Beginn der Ausführungsphase (E1) wird die Bedingung geprüft, und die Operation abhängig vom Ergebnis ausgeführt oder abgebrochen.

- Die Pipeline des DSP ist in drei Stufen *Fetch*, *Decode* und *Execute* unterteilt. Die Fetch-Stufe enthält vier Phasen, die Decode-Stufe enthält zwei Phasen. Diese werden von jeder Operation durchlaufen. Die Ausführungsstufe ist in bis zu 10 Phasen unterteilt, von denen verschiedene Operationen unterschiedlich viele durchlaufen.

- Der Prozessor enthält Hardwareunterstützung für die Ausführung von Schleifen mit Software-Pipelining. Dabei handelt es sich um eine Technik zum Scheduling von Schleifen, bei der mehrere aufeinander folgende Iterationen einer Schleife überlappt werden, um die Parallelität auf Instruktionsebene (ILP) zu erhöhen.

  Im Prozessor steht dazu ein *loop buffer* zur Verfügung, der bis zu 14 VLIW-Instruktionen (mit jeweils bis zu acht Operationen) speichern kann. Eine Operation `SPLOOP` sorgt dafür, dass nachfolgende Operationen nicht nur ausgeführt werden, sondern gleichzeitig in den loop buffer kopiert werden, bis eine Operation `SPKERNEL` das Ende der Schleife markiert. Mit der `SPLOOP`-Operation wird außerdem das *iteration interval (ii)* angegeben, mit dem festgelegt wird, dass alle *ii* Zyklen eine neue Iteration der gespeicherten Schleife ausgeführt wird. Zusätzliche Register (*loop buffer count, inner loop count*) steuern den Ablauf und die Gesamtzahl der Iterationen in der Schleife. Einzelne Operationen in der Schleife können maskiert werden, wodurch sie nur im ersten Durchlauf und nicht bei weiteren Iterationen aus dem loop buffer ausgeführt werden.

6.6.2 Evaluation der Verkehrszeichendetektion

Für die Implementierung der Verkehrszeichendetektion auf dem C6748-DSP wurde die Referenzimplementierung des Algorithmus vereinfacht und optimiert. So wurden die Parameter des Algorithmus für eine feste Bildgröße von 640×480 Pixeln fixiert und Speicher für Bilder und Zwischenergebnisse statisch anstatt dynamisch allokiert. Zusätzliche Pragmas im Code geben dem Kompiler zusätzliche Informationen, z. B. über die Anzahl von Schleifendurchläufen und ermöglichen ein besseres Scheduling von Schleifen, entweder durch Loop Unrolling oder durch die Verwendung von Software-Pipelining. Die Evaluationsergebnisse für die Implementierung auf dem C6748-DSP für ein Frame sind in Tabelle 6.8 dargestellt und werden in den folgenden Punkten erläutert. Alle angegeben Zyklen sind reine CPU-Zyklen ohne Berücksichtigung von Stalls durch Cache-Misses, um den Vergleich mit dem TUKUTURI-Prozessor auf Architekturebene führen zu können.

- Die Farbraumtransformation nach Listing 6.1 enthält Kontrollflussoperationen, die davon abhängen, welche Farbkomponente (rot, grün oder blau) für

Tabelle 6.8: Prozessorzyklen für die Verkehrszeichendetektion auf dem TI C6748-DSP für einen Frame

Routine	**Zyklen**	**dyn. IPC**
Farbraumtransformation	83 809 810	0,970 67
Segmentierung R	35 516 211	1,239 45
Segmentierung B	32 410 877	
Segmentierung Y	39 106 260	
Transferfunktion R	35 055 389	1,163 79
Transferfunktion B	31 950 055	
Transferfunktion Y	38 645 438	
Multiplikation HS	1 382 466	6,995 26
Summe Segmentierung	107 033 348	
Medianfilter	40 947 156	4,479 14
Labeling R	7 072 822	
Labeling B	6 783 858	
Labeling Y	10 351 651	
Erster Durchlauf R	3 517 574	1,500 26
Erster Durchlauf B	3 392 648	
Erster Durchlauf Y	4 937 861	
Zweiter Durchlauf R	3 555 248	1,665 67
Zweiter Durchlauf B	3 391 210	
Zweiter Durchlauf Y	5 413 790	
Summe Labeling	24 208 331	
Bounding Box R	4 113 842	0,714 78
Bounding Box B	3 998 419	
Bounding Box Y	7 486 027	
Summe Bounding Box	15 598 288	
Formklassifikation	17 313	0,864 55
DtB Vektoren	17 259	0,815 47
Klassifikation	1080	1,648 15
Gesamt	271 614 246	1,645 42

ein Pixel am stärksten ist. Dies verhindert ein Software-Pipelining durch den Kompiler. Die verwendeten Divisionen werden durch die Laufzeitbibliothek des TI-Prozessors bereitgestellt und führen zu einer Verringerung des IPC. Ohne Beachtung der Divisionsroutine läge der IPC bei 1,193.

- Die Auswertung der stückweise linearen Transferfunktionen für die Segmentierung beinhaltet ebenfalls Kontrollstrukturen, die ein Software-Pipelining der Schleife verhindern. Hier kommt allerdings eine effizientere Divisions-

routine zum Einsatz als in der Farbraumtransformation, wodurch der IPC positiv beeinflusst wird. Ohne Berücksichtigung der Divisionsroutine läge der IPC bei 0,835.

- Die Multiplikation von Hue- und Saturation-Komponente in der Segmentierung ist vom Kompiler fast vollständig in einer kompakten Schleife mit Software-Pipelining umgesetzt worden, wodurch der hohe IPC erklärt wird.

- Beim Medianfilter ist die innerste Schleife, die das Aufsummieren der Pixel in der Filtermaske durchführt, vom Kompiler abgerollt und mittels Software-Pipelining deutlich kompakter umgesetzt worden als die äußeren Schleifen, in denen der IPC bei etwa 2,7 liegt, so dass der IPC der gesamten Routine auf etwa 4,5 steigt.

- Im Connected Component Labeling sind Schleifen zwar ebenfalls vom Kompiler abgerollt worden, da jedoch das Label, das für ein Pixel vergeben wird, von dem Label der vorherigen Pixel abhängt, ist hier der Parallelisierungsgrad relativ gering.

- Die Bestimmung der umschließenden Rechtecke für die Blobs enthält viele unvorhersehbare Zugriffe auf den Speicher (*random access*), um die Koordinaten der Bounding Boxes für die unterschiedlichen Labels nach jedem gefundenen Pixel zu aktualisieren. Diese können vom Kompiler nur schwer parallelisiert werden, wodurch sich der niedrige IPC erklärt.

- Die Formklassifikation wird klar von der Bestimmung der DtB-Vektoren dominiert. Da hierbei Schleifen zum Einsatz kommen, deren Iterationszahl vorher nicht bekannt ist, konnte der Kompiler kein Software-Pipelining verwenden. Außerdem kommt hier wieder eine Divisionsroutine zum Einsatz. Die abschließende Formklassifikation mittels SVM, also die Berechnung des Skalarprodukts zweier Vektoren, ist wieder mittels Software-Pipelining umgesetzt. Aufgrund der Einfachheit der Schleife sind Prolog und Epilog gemeinsam etwa so groß wie die Schleife selbst, weshalb der IPC trotzdem unter 2 liegt.

Insgesamt erreicht die Implementierung der Verkehrszeichendetektion auf dem TI-Prozessor damit eine Rate von 1,68 Frames pro Sekunde.

6.7 Vergleich und Diskussion der Ergebnisse

Der Vergleich der drei Implementierungen der Verkehrszeichendetektion auf dem TUKUTURI, dem MIPS-Softcore und dem TI-Prozessor zeigt, dass verschiedene Aspekte für die Gesamteffektivität eines Prozessorsystems verantwortlich sind.

So zeigt sich, dass bei Verwendung eines Cachesystems zum Speicherzugriff Wartezyklen entstehen, die in einer Implementierung unter Verwendung von direkten Speichertransfers mittels DMA zum größten Teil vermieden werden können, insbesondere, wenn die Speicherzugriffe gemäß einem vorhersagbaren Muster erfolgen, das dem Programmierer eine genaue Planung der Speichertransfers erlaubt. In den Routinen, in denen der Speicherzugriff nicht einem solchen Muster folgt, wie in den Routinen zur Formklassifikation, treten sowohl bei Verwendung eines Caches als auch bei direkten Speichertransfers Wartezyklen auf.

Auch die Spezialisierung des Prozessorsystems spielt eine gewichtige Rolle. Die Einführung anwendungsspezifischer Operationen in den Prozessor erlaubt eine deutliche Beschleunigung der Datenverarbeitung. In der Implementierung auf dem MIPS-Prozessor mit der Hardwareeinheit konnte für die Medianfilterung eine hohe Beschleunigung erzielt werden. Diese wird aber insbesondere dadurch erreicht, dass der MIPS-Prozessor und das Speichersystem beim Übergang von der farbbasierten Segmentierung zum Medianfilter umgangen werden. Diesem Muster folgend wäre eine dedizierte Hardwareimplementierung ohne Verwendung des Prozessors für die meisten Routinen am günstigsten. Dem steht allerdings eine deutlich reduzierte Flexibilität der Implementierung gegenüber. Die TUKUTURI-Implementierung bietet feingranulare Erweiterungen des Befehlssatzes, die anstelle eines großen funktionalen Blocks (z. B. Medianfilterung) kleinere Aufgaben umsetzen (z. B. Akkumulieren von Pixelwerten), die zur Umsetzung verschiedener komplexerer Operationen durch das Anwendungsprogramm zusammengesetzt werden können. Dadurch ist eine Anpassung an geänderte Anforderungen oder algorithmische Parameter einfacher umzusetzen. Komplexere Operationen wie die Fixpunktdivision können durch Coprozessoren bereitgestellt werden, die unabhängiger vom TUKUTURI arbeiten können. Die Möglichkeit zur dynamischen partiellen Rekonfiguration erlaubt die Verwendung einer großen Zahl von Hardwareerweiterungen, ohne die statische Größe des Prozessorsystems zu erhöhen. Die Verwendung feingranularer Einheiten erlaubt es, die Rekonfiguration im Hintergrund ohne Wartezyklen in die Datenverarbeitung einzuführen. Der MIPS-Prozessor kann diese Möglichkeiten ebenfalls verwenden, wie in Abschnitt 3.4.2 erläutert. Hier wurde diese Möglichkeit nicht eingesetzt, da der kritische Pfad der Anwendung im MIPS-Prozessor selbst und nicht in dem Coprozessor liegt.

Bei der Verarbeitung großer Datenmengen, wie sie in bildverarbeitenden Verfahren typisch sind, spielt auch die Möglichkeit zur Parallelisierung eine große Rolle. In der MIPS-Implementierung ist diese Möglichkeit nicht gegeben, da Operationen sequenziell ausgeführt werden und der Datenpfad auch nicht in unabhängige Subworte unterteilt werden kann. In Hardwareerweiterungen können Daten parallel verarbeitet werden, wie in der verwendeten Hardwareerweiterung für die Medianfilterung geschehen. Durch die mangelnde Unterstützung seitens des MIPS-Prozessors müsste dann allerdings wieder der gesamte Algorithmus in spezialisierte Hardware ausgelagert werden. Der TUKUTURI-Prozessor

bietet Parallelität auf zwei Ebenen, einerseits parallele Issue-Slots, mit denen unterschiedliche Operationen parallel ausgeführt werden können (VLIW), sowie andererseits eine Aufspaltung des 64-Bit-Datenpfades in Subworte, auf denen die gleiche Operation parallel ausgeführt werden kann (micro SIMD). Damit erreicht die Implementierung schon ohne Befehlssatzerweiterungen und ohne Verdoppelung der funktionalen Einheiten höhere Frameraten. Die in der Bildverarbeitung häufig vorkommenden Datenformate mit 8 Bit pro Pixel (Graustufen) oder Binärbilder können im TUKUTURI mit acht Subworten in einem 64-Bit-Register sehr kompakt umgesetzt werden. Dies zeigt sich auch im Vergleich mit der TI-Implementierung. Obwohl der Prozessor mit bis zu 8 Issue-Slots deutlich mehr Operationen parallel ausführen könnte als der TUKUTURI, werden deutlich geringere Frameraten erreicht, da die Parallelisierung auf Instruktionsebene schwieriger ist. Trotz der Möglichkeiten für effizientes Scheduling von Operationen in Schleifen (Software-Pipelining), die der Kompiler mit Unterstützung durch den Programmierer einsetzt, kann der volle Parallelisierungsgrad nur selten erreicht werden. Im TUKUTURI hingegen werden die zwei zur Verfügung stehenden Issue-Slots für die arbeitsintensiven Routinen stärker ausgelastet.

Durch das Operation Merging im TUKUTURI kann die Zahl parallel ausgeführter Operationen nochmals erhöht werden, was insbesondere bei sehr regulären Algorithmen, die viele gleiche Operationen ausführen, zu einer weiteren Kompaktierung und Erhöhung der Parallelität führen kann.

Insgesamt zeigt sich, dass der Einsatz feingranularer Befehlssatzerweiterungen und die Kombination verschiedener Möglichkeiten zur Parallelisierung der Datenverarbeitung eine effektive Implementierung von bildverarbeitenden Systemen ermöglicht. Deren Flexibilität wird durch die Programmierbarkeit eines Prozessorsystems erhalten.

7 Ausblick

Die Implementierung eines Kompiler-Backends mittels evolutionärer Algorithmen für Instruction Scheduling, Registerallokation und Operation Merging zeigt eine stärkere Kompaktierung des Codes als die Verwendung statischer Heuristiken. Außerdem bietet die iterative und adaptive Natur dieser Verfahren eine erhöhte Flexibilität, die es erlaubt, neben hoher Codekompaktierung auch andere Optimierungsziele zu spezifizieren. So könnte beispielsweise durch geeignetes Instruction Scheduling in Kombination mit der Registerallokation die Switching-Aktivität auf den Daten- und Adressleitungen verringert werden, wodurch die Leistungsaufnahme des Prozessors bei Abarbeitung eines Programms verringert werden kann.

Die Verwendung evolutionärer Algorithmen führt allerdings auch zu einem extremen Anstieg der Laufzeit verglichen mit der Verwendung statischer Heuristiken. In der hier vorgestellten Methode multiplizieren sich die Laufzeiten mit jeder Stufe (Instruction Scheduling, Registerallokation, Operation Merging), in der evolutionäre Algorithmen verwendet werden, da für jedes Individuum einer Stufe die Optimierung in der nachfolgenden Stufe durchgeführt wird. Durch eine engere Verknüpfung bzw. Integration der Stufen, beispielsweise durch Kombination des Operation Merging und des Instruction Scheduling, könnte die Multiplikation der Laufzeiten verringert werden. Eine weitere Schwierigkeit ist die randomisierte Natur der evolutionären Algorithmen, wodurch das Ergebnis der Kompilierung nicht deterministisch ist. Insbesondere in Kombination mit der Parallelisierung ist eine deterministische Version der Algorithmen schwer umzusetzen. Dies stellt insbesondere für das Debugging eines Anwendungscodes oder des Kompiler-Backends selbst eine Hürde dar, da ein Kompilierungsergebnis nicht zuverlässig reproduziert werden kann.

Am Institut für Mikroelektronische Systeme ist auf Basis der hier vorgestellten Arbeiten zum EA-basierten Kompiler-Backend ein DFG-Projekt (Gegenüberstellung von evolutionären Algorithmen und maschinellen Lernverfahren für energiebewusstes Instruction Scheduling, PA 2762/2-1) eingeworben worden, in dem die Vorteile der EA-basierten Heuristiken mit der Geschwindigkeit und dem Determinismus von statischen Heuristiken kombiniert werden sollen. Dazu wird für eine Prozessorarchitektur, die flexibel beschrieben werden kann, sowie eine Menge von Anwendungsprogrammen mittels des EA-basierten Kompiler-Backends eine optimale Kompilierung vorgenommen, um Trainingsdaten zu generieren. Anschließend wird mit Methoden des maschinellen Lernens anhand

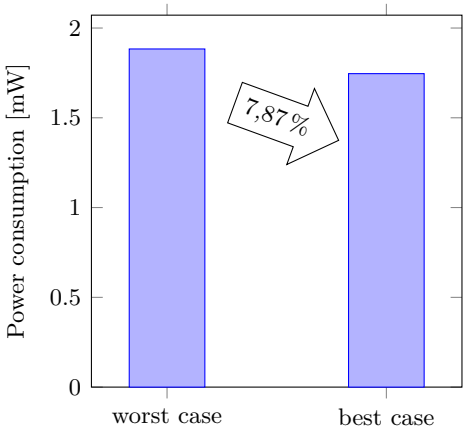


Abbildung 7.1: Gesunkene Leistungsaufnahme eines Prozessors durch Register Renaming für einen synthetischen Benchmark [113]

dieser Trainingsdaten eine statische Heuristik für die Prozessorarchitektur abgeleitet. Mittels dieser Heuristik wird die Kompilierung gegenüber der Verwendung evolutionärer Algorithmen deterministisch und deutlich beschleunigt, erreicht aber hoffentlich trotzdem eine vergleichbare Güte wie die EA-basierte Kompilierung. Zusätzlich wird in dem Projekt ein Modell für die Leistungsaufnahme des Prozessors bei Ausführung verschiedener Operationen ermittelt, so dass nicht nur die Kompaktierung und damit verbunden die Ausführungsgeschwindigkeit eines Programms, sondern auch der Energiebedarf bei der Kompilierung berücksichtigt und optimiert werden kann. In einer Vorarbeit konnte gezeigt werden, dass durch eine geschickte Wahl von Registeradressen, eine Art von Register Renaming, die Leistungsaufnahme eines Prozessors um etwa 7,9 % gesenkt werden konnte, wie in Abbildung 7.1 dargestellt [113].

In Bezug auf die Hardwareimplementierung zeigt sich, dass die Kombination von hohem Parallelisierungsgrad (SIMD, VLIW) mit feingranularen Hardwareerweiterungen (funktionalen Einheiten) eine gute Adaption an die Anwendung und damit eine effiziente Umsetzung ermöglicht. Die Implementierung auf dem FPGA bietet den Vorteil der Flexibilität, so dass Erweiterungen und Hardwareparameter in mehreren Zyklen adaptiert und evaluiert werden können. Ein Nachteil der FPGA-Implementierung ergibt sich aus den hohen Kosten für das Routing. Dies zeigt sich insbesondere in den Versionen der funktionalen Einheiten, die mit zusätzlichen Pipelinestufen ausgestattet werden. Ohne den Einfluss des Routings, beispielsweise in einer ASIC-Implementierung, könnte der Gewinn durch zusätzliche Pipelinestufen höher ausfallen, und die Unterschiede (in der minimaler Periode) könnten deutlicher sein, wodurch der NSGA-II zur Optimierung der Architektur schneller zu besseren Konfigurationen konvergieren könnte.

8 Zusammenfassung und Fazit

Für die effektive Umsetzung moderner Anwendungen der Signalverarbeitung, deren Anforderungen aufgrund steigender Komplexität von Algorithmen und einer Zunahme der zu verarbeitenden Datenmenge stetig steigen, können anwendungsspezifische Prozessoren eingesetzt werden. Dabei kann eine spezifische Prozessorarchitektur für jede Anwendung von Grund auf entworfen werden, oder eine bestehende Architektur durch Konfiguration an die Anwendung angepasst werden. Die Konfiguration bestehender Architekturen oder Architekturvorlagen stellt eine einfachere Methode dar, da eine Grundfunktionalität schon vorhanden ist und auf bereits verifizierten Komponenten der Architektur aufgebaut werden kann. Allerdings kann diese Methode den Entwurf der anwendungsspezifischen Architektur auch einschränken, wenn die benötigten Konfigurationen nicht bereitgestellt werden.

In dieser Arbeit wurde ein Entwurfsprozess vorgestellt, der auf der flexiblen TUKUTURI-Architektur basiert, die umfangreich konfiguriert werden kann. Da die Architekturbeschreibung in VHDL vorliegt, kann die Anpassung an eine Anwendung sehr detailliert erfolgen. Die Architekturvorlage bietet die Möglichkeit zur Parallelisierung auf Daten- und Instruktionsebene (SIMD, VLIW) sowie das Operation Merging (X2-Modus) zur effektiven Umsetzung rechenintensiver Algorithmen mit regulärer Struktur. Weitere Merkmale, wie etwa die konditionale Ausführung von Operationen, erlauben eine kompakte und damit schnelle Umsetzung verschiedener Anwendungen, nicht nur im Bereich der Bildverarbeitung, sondern beispielsweise auch in der Audiosignalverarbeitung, wie in verschiedenen Arbeiten im Rahmen des Exzellenzclusters „Hearing4all“ gezeigt wurde [27]. Die bereitgestellten Funktionen können feingranular konfiguriert werden, um damit die Architektur für minimalen Ressourcenaufwand und maximale Taktfrequenz zu optimieren. Die Architekturvorlage bietet verschiedene Möglichkeiten zur Erweiterung des Funktionsumfangs, beispielsweise durch eine enge Integration funktionaler Einheiten, die anwendungsspezifische Operationen in der Pipeline des Prozessors bereitstellen, oder durch Anbindung komplexerer Coprozessoren, die unabhängig von der Prozessorpipeline arbeiten. Zusätzlich zu dieser statischen Konfiguration des Prozessors wird bei Umsetzung als Softcore-Prozessor auf einem FPGA die Möglichkeit zur dynamischen Rekonfiguration bereitgestellt, durch die der Prozessor mit einer großen Menge von Erweiterungen ausgestattet werden kann und gleichzeitig die vorhandene Fläche optimal ausgenutzt werden kann, wenn nicht alle Erweiterungen gleichzeitig verwendet werden.

Für den Entwurf der anwendungsspezifischen Ausprägung der Architekturvorlage wurde ein Toolflow vorgestellt, der die verschiedenen Phasen des Entwurfsprozesses unterstützt. Dieser Workflow basiert auf einer generischen Beschreibung der Funktionalität der TUKUTURI-Architektur, aus der zum einen die Konfiguration des generischen VHDL-Codes für die Synthese oder Simulation des Prozessors abgeleitet werden kann und zum anderen der Assembler Kompiler für die konkrete Ausprägung der Architektur konfiguriert wird. Zur Anwendungsentwicklung wird eine funktionale Simulation des Prozessors in C bereitgestellt, die eine graduelle Übertragung in Assemblercode gestattet. Zur Evaluation der Umsetzung einer Anwendung auf einer konkreten Ausprägung der Architektur werden verschiedene Profilingmethoden bereitgestellt, die Laufzeitinformationen aus einer (zyklengenauen) Pipelinesimulation oder aus der FPGA-Emulation des Prozessors sammeln. Aus einem bereits ermittelten Profiling kann mittels des Profiling Estimators das Profiling für eine modifizierte Architektur des Prozessors geschätzt werden, um den Emulationsschritt zu umgehen. Schließlich kann der Assembler Kompiler den benötigten Funktionsumfang der Architektur ermitteln und in Form einer Minimalkonfiguration ausgeben. Eine automatische Optimierung der Architekturparameter durch einen evolutionären Algorithmus erlaubt eine zielgerichtete Exploration des Entwurfsraums.

Ein wichtiges Element des Workflows ist der Assembler Kompiler, der zur Übersetzung von Anwendungscode für die Prozessorarchitektur genutzt wird. Die Wahl einer VLIW-Architektur für den Prozessor führt zu einer gesteigerten Komplexität des Kompilers, der das Instruction Scheduling durchführen muss, bei dem die einzelnen Operationen des Programms möglichst kompakt in Instruktionswörtern platziert werden. Dabei handelt es sich um ein kombinatorisches Problem mit einer Vielzahl möglicher Lösungen, die bei den in der Praxis vorkommenden Programmen nicht alle untersucht werden können, um die optimale Lösung zu finden. Stattdessen werden Heuristiken eingesetzt, um Lösungen zu bewerten. Durch die Flexibilität der Prozessorarchitektur ist die Entwicklung einer statischen Heuristik, die in allen Fällen gute Ergebnisse liefert, sehr schwer bzw. unmöglich. Zusätzlich sind vom Kompiler noch weitere Aufgaben zu lösen, wie die Registerallokation oder das automatische Operation Merging. Daher wurden in dieser Arbeit dynamische Heuristiken auf Basis von evolutionären Algorithmen entwickelt, die sich adaptiv sowohl an die Prozessorkonfiguration als auch an die Charakteristik der Anwendung anpassen.

Der Workflow ist anschließend an einer exemplarischen Anwendung, die eine Verkehrszeichendetektion in Farbbildern umsetzt, evaluiert worden. Die Anwendung besteht aus mehreren Routinen, die unterschiedliche Charakteristika in Bezug auf Rechenkomplexität und Speicherkomplexität aufweisen. Es zeigt sich, dass die kombinierte Optimierung von Hardwareparametern und Kompilierung der Anwendung einen hohen Grad der Anpassung an die Anwendung und damit eine Beschleunigung der Datenverarbeitung ermöglicht. Verschiedene Kompiler-

techniken für Operation Merging, Instruction Scheduling und Registerallokation wurden ausführlich untersucht. Auch bei Adaption der statischen Prozessorkonfiguration kann sich die Kompilierung an die geänderte Konfiguration anpassen und eine gute Kompaktierung des Codes erreichen. Durch eine Spezialisierung der Prozessorarchitektur mittels Instruktionssatzerweiterungen kann die Umsetzung der Anwendung nochmals verbessert werden. Die negativen Auswirkungen einer hohen Anzahl funktionaler Einheiten auf die Taktfrequenz der Prozessorarchitektur können durch dynamische partielle Rekonfiguration der Einheiten verringert werden.

Die verschiedenen Konfigurationen des TUKUTURI-Prozessors mit Basisinstruktionssatz, mit Befehlssatzerweiterungen und mit dynamischer partieller Rekonfiguration sind mit Implementierungen auf einem MIPS Softcore Prozessor und einem TI C6748-DSP verglichen worden. Der MIPS-Prozessor ist dazu um einen anwendungsspezifischen Coprozessor erweitert worden. Für den TUKUTURI-Prozessor im Basisinstruktionssatz werden zur Berechnung eines Frames zwischen 20 und 40 Millionen Taktzyklen benötigt. Dagegen benötigt der MIPS-Prozessor ohne Hardwareerweiterungen etwa 500 Millionen Zyklen. Der TI-DSP benötigt etwa 270 Millionen Zyklen. Für einen TUKUTURI mit Hardwareerweiterungen sinkt die Zahl der Taktzyklen auf etwa 5 bis 7 Millionen Zyklen, wohingegen der MIPS-Prozessor mit Hardwareerweiterung etwa 200 Millionen Zyklen benötigt.

Literaturverzeichnis

Literatur des Autors (als Erstautor)

[28] F. Giesemann, L. Gerlach und G. Payá-Vayá. „Evolutionary Algorithms for Instruction Scheduling, Operation Merging, and Register Allocation in VLIW Compilers“. In: *Journal of Signal Processing Systems* 92 (2020), S. 655–678. DOI: 10.1007/s11265-019-01493-2.

[29] F. Giesemann, G. Payá-Vayá und H. Blume. „A Hardware/Software Environment for Specializing Dynamic Reconfigurable Generic VLIW-SIMD ASIP Architecture“. In: *ICT.OPEN 2012 Conference, Proceedings of the*. 2012.

[30] F. Giesemann, G. Payá-Vayá, H. Blume, M. Limmer und W. Ritter. „A comprehensive ASIC/FPGA prototyping environment for exploring embedded processing systems for advanced driver assistance applications“. In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014, Proceedings of the*. Juli 2014, S. 314–321. DOI: 10.1109/SAMOS.2014.6893227.

[31] F. Giesemann, G. Payá-Vayá, H. Blume, M. Limmer und W. R. Ritter. „Deep Learning for Advanced Driver Assistance Systems“. In: *Towards a Common Software/Hardware Methodology for Future Advanced Driver Assistance Systems – The DESERVE Approach*. Hrsg. von G. Payá Vayá und H. Blume. River Publishers, 2017. Kap. 6. ISBN: 9788793519145. DOI: 10.13052/rp-9788793519138.

Literatur des Autors (als Koautor)

[5] O. J. Arndt, D. Becker, F. Giesemann, G. Payá-Vayá, C. Bartels und H. Blume. „Performance evaluation of the Intel Xeon Phi manycore architecture using parallel video-based driver assistance algorithms“. In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014, Proceedings of the*. Juli 2014, S. 125–132. DOI: 10.1109/SAMOS.2014.6893203.

[6] F. Badstübner, R. Ködel, W. Maurer, M. Kunert, A. Rolfsmeier, J. Pérez, F. Giesemann, G. Payá-Vayá, H. Blume und G. Reade. „The DESERVE Platform: A Flexible Development Framework to Seemlessly Support the ADAS Development Levels“. In: *Towards a Common Software/Hardware Methodology for Future Advanced Driver Assistance Systems – The DESERVE Approach*. Hrsg. von G. Payá-Vayá und H. Blume. River Publishers, 2017. Kap. 2. ISBN: 9788793519145. DOI: 10.13052/rp-9788793519138.

[71] S. Nolting, G. Payá-Vayá, F. Giesemann und H. Blume. „Exploring Dynamic Reconfigurable CORDIC Co-Processors Tightly Coupled with a VLIW-SIMD Soft-Processor Architecture“. In: *Applied Reconfigurable Computing*. Hrsg. von K. Sano, D. Soudris, M. Hübner und P. C. Diniz. Bd. 9040. Lecture Notes in Computer Science. Springer International Publishing, 2015, S. 401–410. ISBN: 978-3-319-16213-3. DOI: 10.1007/978-3-319-16214-0_36.

[72] S. Nolting, G. Payá-Vayá, F. Giesemann, H. Blume, S. Niemann und C. Müller-Schloer. „Dynamic Self-Reconfiguration of a MIPS-Based Soft-Processor Architecture“. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Mai 2016, S. 172–180. DOI: 10.1109/IPDPSW.2016.158.

[73] S. Nolting, G. Payá-Vayá, F. Giesemann, H. Blume, S. Niemann und C. Müller-Schloer. „Dynamic self-reconfiguration of a MIPS-based soft-core processor architecture“. In: *Journal of Parallel and Distributed Computing* (Okt. 2017). DOI: 10.1016/j.jpdc.2017.09.013.

[78] G. Payá-Vayá und H. Blume, Hrsg. *Towards a Common Software/Hardware Methodology for Future Advanced Driver Assistance Systems: The DESERVE Approach*. River Publishers, 2017. ISBN: 9788793519145. DOI: 10.13052/rp-9788793519138.

[80] G. Payá-Vayá, J. Martín Langerwerf, F. Giesemann, H. Blume und P. Pirsch. „Instruction merging to increase parallelism in VLIW architectures“. In: *System-on-Chip (SOC), International Symposium on*. Okt. 2009, S. 143–146. DOI: 10.1109/SOCC.2009.5335660.

Literatur

[1] A. V. Aho, M. S. Lam, R. Sethi und J. D. Ullman. *Compiler: Prinzipien, Techniken und Werkzeuge*. 2. Aufl. Pearson Studium, 2008. ISBN: 9783827370976.

[2] T. Äijö, P. Jääskeläinen, T. Elomaa, H. Kultala und J. Takala. „Integer Linear Programming-Based Scheduling for Transport Triggered Architectures“. In: *ACM Transactions on Architecture and Code Optimization* 12.4 (Dez. 2015), S. 1–22. DOI: 10.1145/2845082.

[3] F. Anjam, L. Carro, S. Wong, G. L. Nazar und M. B. Rutzig. „Simultaneous Reconfiguration of Issue-width and Instruction Cache for a VLIW Processor“. In: *Proc. International Conference on Embedded Computer Systems: Architecture Modeling and Simulation*. Samos, Greece, Juli 2012.

[4] F. Anjam, M. Nadeem und S. Wong. „A VLIW Softcore Processor with Dynamically Adjustable Issue-slots“. In: *Proc. International Conference on Field-Programmable Technology*. Beijing, China, Dez. 2010.

[7] J. Bahuleyan, R. Nagpal und Y. N. Srikant. „Integrated energy-aware cyclic and acyclic scheduling for clustered VLIW processors“. In: *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, Apr. 2010. DOI: 10.1109/ipdpsw.2010.5470906.

[8] J. E. Baker. „Reducing Bias and Inefficiency in the Selection Algorithm“. In: *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. Cambridge, Massachusetts, USA: L. Erlbaum Associates Inc., 1987, S. 14–21. ISBN: 0805801588.

[9] L. Barthe, L. V. Cargnini, P. Benoit und L. Torres. „The SecretBlaze: A Configurable and Cost-Effective Open-Source Soft-Core Processor“. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, Mai 2011. DOI: 10.1109/ipdps.2011.154.

[10] L. Bauer und J. Henkel. *Run-time Adaptation for Reconfigurable Embedded Processors*. Springer New York, 2011. DOI: 10.1007/978-1-4419-7412-9.

[11] J. Becker, T. Pionteck und M. Glesner. „DReAM: A Dynamically Reconfigurable Architecture for Future Mobile Communication Applications“. In: *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*. Hrsg. von R. W. Hartenstein und H. Grünbacher. Springer Berlin Heidelberg, 2000, S. 312–321. DOI: 10.1007/3-540-44614-1_34.

[12] P. Briggs. „Register Allocation via Graph Coloring“. Diss. Rice University, 1992.

[13] Cadence Inc. *Tensilica Customizable Processor and DSP IP*. 14. Feb. 2019. URL: https://ip.cadence.com/ipportfolio/tensilica-ip.

[14] F. Campi, A. Cappelli, R. Guerrieri, A. Lodi, M. Toma, A. L. Rosa, L. Lavagno, C. Passerone und R. Canegallo. „A reconfigurable processor architecture and software development environment for embedded systems“. In: *Proceedings International Parallel and Distributed Processing Symposium.* IEEE Comput. Soc, 2003. DOI: 10.1109/ipdps.2003.1213314.

[15] D. Capalija und T. S. Abdelrahman. „A high-performance overlay architecture for pipelined execution of data flow graphs“. In: *2013 23rd International Conference on Field programmable Logic and Applications.* IEEE, Sep. 2013. DOI: 10.1109/fpl.2013.6645515.

[16] K. Chapman. *PicoBlaze for Spartan-6, Virtex-6, 7-Series, Zynq and UltraScale Devices (KCPSM6).* Techn. Ber. Xilinx Inc., 2014.

[17] G. Charitopoulos, I. Koidis, K. Papadimitriou und D. Pnevmatikatos. „Hardware Task Scheduling for Partially Reconfigurable FPGAs“. In: *Applied Reconfigurable Computing.* Hrsg. von K. Sano, D. Soudris, M. Hübner und P. C. Diniz. Springer International Publishing, 2015, S. 487–498. DOI: 10.1007/978-3-319-16214-0_45.

[18] Cobham Gaisler Inc. *LEON/GRLIB Configuration and Development Guide.* 2010.

[19] *Codasip.* 8. März 2019. URL: https://www.codasip.com/.

[20] J. M. Codina, J. Sanchez und A. Gonzalez. „A unified modulo scheduling and register allocation technique for clustered processors“. In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques.* IEEE Comput. Soc, 2001. DOI: 10.1109/pact.2001.953298.

[21] J. Coole und G. Stitt. „Adjustable-Cost Overlays for Runtime Compilation“. In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines.* IEEE, Mai 2015. DOI: 10.1109/fccm.2015.49.

[22] *CorExtend Instruction Integrator's Guide for M4K/4KE/4KS and M14K Family Cores.* MIPS Technologies. 2011.

[23] K. Deb, A. Pratap, S. Agarwal und T. Meyarivan. „A fast and elitist multiobjective genetic algorithm: NSGA-II“. In: *IEEE Transactions on Evolutionary Computation* 6.2 (Apr. 2002), S. 182–197. ISSN: 1089-778X. DOI: 10.1109/4235.996017.

[24] M. V. Eriksson, O. Skoog und C. W. Kessler. „Optimal vs. Heuristic Integrated Code Generation for Clustered VLIW Architectures“. In: *Proceedings of the 11th International Workshop on Software & Compilers for Embedded Systems (SCOPES '08).* SCOPES '08. Munich, Germany: ACM, 2008, S. 11–20.

[25] D. Estes. *SchedMachineModel: Adding and Optimizing a Subtarget*. 2014. URL: https://llvm.org/devmtg/2014-10/Slides/Estes-MISchedulerTutorial.pdf.

[26] J. A. Fisher. *Optimization of horizontal microcode within and beyond basic blocks: an application of processor scheduling with resources*. Techn. Ber. New York Univ., NY (USA). Courant Mathematics und Computing Lab., 1979.

[27] L. Gerlach, G. Payá-Vayá und H. Blume. „KAVUAKA: A Low Power Application Specific Hearing Aid Processor". In: *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. 2019, S. 99–104.

[32] D. E. Goldberg und K. Deb. „A Comparative Analysis of Selection Schemes Used in Genetic Algorithms". In: *Foundations of Genetic Algorithms*. Elsevier, 1991, S. 69–93. DOI: 10.1016/b978-0-08-050684-5.50008-2.

[33] R. C. Gonzales und R. E. Woods. *Digital Image Processing*. 2. Aufl. Prentice Hall, 1992.

[34] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt und A. Nicolau. *EXPRESSION: An ADL for System Level Design Exploration*. Techn. Ber. Department of Information und Computer Science, University of California, 1998.

[35] S. G. Hansen, D. Koch und J. Torresen. „High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro". In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, Mai 2011. DOI: 10.1109/ipdps.2011.139.

[36] T. Harbaum, C. Schade, M. Damschen, C. Tradowsky, L. Bauer, J. Henkel und J. Becker. „Auto-SI: An adaptive reconfigurable processor with run-time loop detection and acceleration". In: *2017 30th IEEE International System-on-Chip Conference (SOCC)*. IEEE, Sep. 2017. DOI: 10.1109/socc.2017.8226027.

[37] S. Hauck, T. W. Fry, M. M. Hosler und J. P. Kao. „The Chimaera reconfigurable functional unit". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.2 (Feb. 2004), S. 206–217. DOI: 10.1109/tvlsi.2003.821545.

[38] J. Heistermann. *Genetische Algorithmen – Theorie und Praxis evolutionärer Optimierung*. Bd. 9. TEUBNER-TEXTE zur Informatik. Vieweg+Teubner Verlag, 1994. DOI: 10.1007/978-3-322-99633-6.

[39] J. Henkel, L. Bauer, M. Hübner und A. Grudnitsky. „i-Core: A run-time adaptive processor for embedded multi-core systems“. In: *Proceedings of The International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. Juli 2011.

[40] J. Hennessy und D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2017. 936 S.

[41] J. L. Hennessy und T. Gross. „Postpass Code Optimization of Pipeline Constraints“. In: *ACM Transactions on Programming Languages and Systems* 5.3 (Juli 1983), S. 422–448. ISSN: 0164-0925. DOI: 10.1145/2166.357217.

[42] J. Holland. *Adaptation in Natural Artificial Systems*. University of Michigan Press, 1975.

[43] D. L. How und S. Atsatt. „Sectors: Divide & Conquer and Softwarization in the Design and Validation of the Stratix® 10 FPGA“. In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Mai 2016. DOI: 10.1109/fccm.2016.37.

[44] Intel Inc. *Nios II Processor Reference Guide*. Techn. Ber. Intel Inc., 2018.

[45] A. K. Jain, D. L. Maskell und S. A. Fahmy. „Throughput oriented FPGA overlays using DSP blocks“. In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. März 2016, S. 1628–1633. DOI: 10.3850/9783981537079_0685.

[46] E. Janßen. „Implementierung und Optimierung eines MIPS-basierten Soft-Core-Prozessorsystems zur Verkehrszeichenerkennung“. Masterarbeit. Institut für Mikroelektronische Systeme, Leibniz Universität Hannover, 2018.

[47] M. A. Kadi. „FPGU: A Flexible Soft GPU Architecture for General Purpose Computing on FPGAs“. Diss. Ruhr University of Bochum, Chair for Embedded Systems for Information Technology, 2017.

[48] K. Karuri und R. Leupers. *Application Analysis Tools for ASIP Design*. Springer New York, 2011. DOI: 10.1007/978-1-4419-8255-1.

[49] M. Kock, S. Hesselbarth, M. Pfitzner und H. Blume. „Hardware-Accelerated Design Space Exploration Framework for Communication Systems“. In: *Analog Integrated Circuits and Signal Processing* (2013). DOI: 10.1007/s10470-013-0127-6.

[50] C. Lee, J. K. Lee, T. Hwang und S.-C. Tsai. „Compiler Optimization on VLIW Instruction Scheduling for Low Power“. In: *ACM Transaction on Design Automation of Electronic Systems (TODAES)* 8.2 (Apr. 2003), S. 252–268. ISSN: 1084-4309. DOI: 10.1145/762488.762494.

[51] R. Leupers und S. Bashford. „Graph-based code selection techniques for embedded processors“. In: *ACM Transactions on Design Automation of Electronic Systems* 5.4 (Okt. 2000), S. 794–814. DOI: 10.1145/362652.362661.

[52] Y.-C. Lin, Y.-P. You und J. K. Lee. „Register Allocation for VLIW DSP Processors with Irregular Register Files“. In: *12th Workshop on Compilers for Parallel Computers (CPC 2006), Coruña ,Spain*. 9. Jan. 2006.

[53] M. Liu, W. Kuehn, Z. Lu und A. Jantsch. „Run-time Partial Reconfiguration speed investigation and architectural design space exploration“. In: *2009 International Conference on Field Programmable Logic and Applications*. IEEE, Aug. 2009. DOI: 10.1109/fpl.2009.5272463.

[54] A. Lodi, A. Cappelli, M. Bocchi, C. Mucci, M. Innocenti, C. DeBartolomeis, L. Ciccarelli, R. Giansante, A. Deledda, F. Campi, M. Toma und R. Guerrieri. „XiSystem: A XiRisc-Based SoC With Reconfigurable IO Module“. In: *IEEE Journal of Solid-State Circuits* 41.1 (Jan. 2006), S. 85–96. DOI: 10.1109/jssc.2005.859319.

[55] *LogiCORE IP XPS HWICAP*. Datenblatt. Xilinx Inc. 2010.

[56] M. Lorenz, R. Leupers, P. Marwedel, T. Drager und G. Fettweis. „Low-energy DSP code generation using a genetic algorithm“. In: *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*. 2001, S. 431–437. DOI: 10.1109/ICCD.2001.955062.

[57] M. Lorenz, L. Wehmeyer und T. Dräger. „Energy aware compilation for DSPs with SIMD instructions“. In: *Proceedings of the joint conference on Languages, compilers and tools for embedded systems software and compilers for embedded systems - LCTES/SCOPES '02*. ACM Press, 2002, S. 94–101. DOI: 10.1145/513829.513847.

[58] R. C. Lozano, M. Carlsson, G. H. Blindell und C. Schulte. „Combinatorial spill code optimization and ultimate coalescing“. In: *Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems - LCTES '14*. ACM Press, 2014, S. 23–32. DOI: 10.1145/2597809.2597815.

[59] R. C. Lozano, M. Carlsson, F. Drejhammar und C. Schulte. „Constraint-Based Register Allocation and Instruction Scheduling“. In: *Principles and Practice of Constraint Programming*. Hrsg. von M. Milano. Springer Berlin Heidelberg, 2012, S. 750–766. DOI: 10.1007/978-3-642-33558-7_54.

[60] P. R. Mahalingam. „Knowledge-Augmented Genetic Algorithms for Effective Instruction Template Selection in Compilers“. In: *Third International Conference on Advances in Computing and Communications*. IEEE, Aug. 2013, S. 21–24. DOI: 10.1109/icacc.2013.96.

[61] S. Maldonado-Bascón, S. Lafuente-Arroyo, P. Gil-Jiménez, H. Gomez-Moreno und F. Lopez-Ferreras. „Road-Sign Detection and Recognition Based on Support Vector Machines“. In: *Intelligent Transportation Systems, IEEE Transactions on* 8.2 (Juni 2007), S. 264–278. ISSN: 1524-9050. DOI: 10.1109/TITS.2007.895311.

[62] A. McGovern und E. Moss. „Scheduling Straight-line Code Using Reinforcement Learning and Rollouts“. In: *Proceedings of the 11th International Conference on Neural Information Processing Systems*. NIPS'98. Denver, CO: MIT Press, 1998, S. 903–909.

[63] B. Mei, S. Vernalde, D. Verkest, H. D. Man und R. Lauwereins. „ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix“. In: *Field Programmable Logic and Application*. Hrsg. von P. Y. K. Cheung und G. A. Constantinides. Springer Berlin Heidelberg, 2003, S. 61–70. DOI: 10.1007/978-3-540-45234-8_7.

[64] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[65] B. Neumann, T. von Sydow, H. Blume und T. G. Noll. „Application Domain Specific Embedded FPGAs for Flexible ISA-Extension of ASIPs“. In: *Journal of Signal Processing Systems* 53.1-2 (Mai 2008), S. 129–143. DOI: 10.1007/s11265-008-0211-9.

[66] M. Ng und M. Peattie. *Using a Microprocessor to Configure Xilinx FPGAs via Slave Serial or SelectMAP Mode*. Techn. Ber. Xilinx Inc., 2002.

[67] C. Nian, H. Yanxiang, C. Yong, L. Ximi und L. Qian. „PSO Based Instruction Scheduling for Low Power“. In: *International Conference on Computer Distributed Control and Intelligent Environmental Monitoring (CDCIEM)*. IEEE, März 2012. DOI: 10.1109/cdciem.2012.129.

[68] A. Nicolau, R. Potasman und H. Wang. „Register allocation, renaming and their impact on fine-grain parallelism“. In: *Languages and Compilers for Parallel Computing*. Hrsg. von U. Banerjee, D. Gelernter, A. Nicolau und D. Padua. Springer Berlin Heidelberg, 1992, S. 218–235. DOI: 10.1007/bfb0038667.

[69] T. G. Noll. „Application domain specific embedded FPGAs for SoC platforms“. In: *Irish Signals and Systems Conference 2004*. IEE, 2004. DOI: 10.1049/cp:20040506.

[70] S. Nolting, G. Payá-Vayá und H. Blume. „Optimizing VLIW-SIMD Processor Architectures for FPGA Implementation“. In: *ICT.OPEN 2011 Conference (Veldhoven, Netherlands)*. Bd. USB-Proceedings. 2011.

[74] K. Ovtcharov, I. Tili und J. G. Steffan. „TILT: A multithreaded VLIW soft processor family“. In: *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE, Sep. 2013. DOI: 10.1109/fpl.2013.6645553.

[75] K. Papadimitriou, A. Dollas und S. Hauck. „Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model“. In: *ACM Trans. Reconfigurable Technol. Syst.* 4.4 (Dez. 2011), 36:1–36:24. ISSN: 1936-7406. DOI: 10.1145/2068716.2068722.

[76] A. Parikh, S. Kim, M. Kandemir, N. Vijaykrishnan und M. J. Irwin. „Instruction Scheduling for Low Power“. In: *Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology* 37.1 (Mai 2004), S. 129–149. DOI: 10.1023/b:vlsi.0000017007.28247.f6.

[77] G. Payá-Vayá. „Design and Analysis of a Generic VLIW Processor for Multimedia Applications“. Diss. Institute of Microelectronic Systems, Leibniz Universität Hannover, 2011.

[79] G. Payá-Vayá, R. Burg und H. Blume. „Dynamic Data-Path Self-Reconfiguration of a VLIW-SIMD Soft-Processor Architecture“. In: *Workshop on Self-Awareness in Reconfigurable Computing Systems (SRCS) in conjunction with the 2012 International Conference on Field Programmable Logic and Applications (FPL 2012)*. 2012, S. 26–29.

[81] G. Payá-Vayá, J. Martín-Langerwerf, P. Taptimthong und P. Pirsch. „Design Space Exploration of Media Processors: A Parameterized Scheduler“. In: *2007 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. Juli 2007, S. 41–49. DOI: 10.1109/ICSAMOS.2007.4285732.

[82] G. Payá-Vayá, J. Martín-Langerwerf, P. Taptimthong und P. Pirsch. „Design Space Exploration of Media Processors: A Generic VLIW Architecture and a Parameterized Scheduler“. English. In: *Architecture of Computing Systems - ARCS 2007*. Hrsg. von P. Lukowicz, L. Thiele und G. Tröster. Bd. 4415. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, S. 254–267. ISBN: 978-3-540-71267-1. DOI: 10.1007/978-3-540-71270-1_19.

[83] F. Pflug. „Analysis of Instruction Scheduling Strategies for VLIW Processors in order to Reduce Register Spilling“. Masterarbeit. Institute of Microelectronic Systems, Leibniz Universität Hannover, 2015.

[84] H. Pohlheim. *Evolutionäre Algorithmen – Verfahren, Operatoren und Hinweise für die Praxis*. 2000. ISBN: 978-3-642-57137-4. DOI: 10.1007/978-3-642-57137-4.

[85] I. Rechenberg. „Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution“. In: (1973).

[86] G. Schewior, C. Zahl, H. Blume, S. Wonneberger und J. Effertz. „HLS-based FPGA implementation of a predictive block-based motion estimation algorithm – A field report“. In: *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*. IEEE, Okt. 2014. DOI: 10.1109/dasip.2014.7115633.

[87] A. Severance und G. G. F. Lemieux. „Embedded Supercomputing in FPGAs with the VectorBlox MXP Matrix Processor“. In: *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*. CODES+ISSS '13. Montreal, Quebec, Canada: IEEE Press, 2013, 6:1–6:10. ISBN: 978-1-4799-1417-3.

[88] L. G. Shapiro und G. Stockman. *Computer Vision*. Prentice Hall, 2002.

[89] D. She, Y. He, B. Mesman und H. Corporaal. „Scheduling for register file energy minimization in explicit datapath architectures“. In: *Proceedings of the IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*. März 2012, S. 388–393. DOI: 10.1109/date.2012.6176502.

[90] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh und E. C. Filho. „MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications“. In: *IEEE Transactions on Computers* 49.5 (Mai 2000), S. 465–481. DOI: 10.1109/12.859540.

[91] J. D. Souza, A. L. Sartor, L. Carro, M. B. Rutzig, S. Wong und A. C. S. Beck. „DIM-VEX: Exploiting Design Time Configurability and Runtime Reconfigurability“. In: *Proceedings of The 14th International Symposium on Applied Reconfigurable Computing*. Mai 2018, S. 367–378.

[92] N. Srinivas und K. Deb. „Muiltiobjective Optimization Using Nondominated Sorting in Genetic Algorithms“. In: *Evolutionary Computation* 2.3 (Sep. 1994), S. 221–248. ISSN: 1063-6560. DOI: 10.1162/evco.1994.2.3.221.

[93] J. Stallkamp, M. Schlipsing, J. Salmen und C. Igel. „The German Traffic Sign Recognition Benchmark: A multi-class classification competition“. In: *IEEE International Joint Conference on Neural Networks*. 2011, S. 1453–1460.

[94] G. Stitt und J. Coole. „Intermediate Fabrics: Virtual Architectures for Near-Instant FPGA Compilation“. In: *IEEE Embedded Systems Letters* 3.3 (Sep. 2011), S. 81–84. DOI: 10.1109/les.2011.2167713.

[95] *Stratix V Device Overview*. Datenblatt. Altera Corp. 2013.

[96] Stretch Inc. *The S6000 Family of Processors*. Techn. Ber. Stretch Inc., 2009.

[97] X. Su, H. Wu und J. Xue. „An Efficient WCET-Aware Instruction Scheduling and Register Allocation Approach for Clustered VLIW Processors“. In: *ACM Transactions on Embedded Computing Systems* 16.5s (Sep. 2017), S. 1–21. DOI: 10.1145/3126524.

[98] Synposys Inc. *Synopsys ASIP Designer*. 14. Feb. 2019. URL: https://www.synopsys.com/dw/ipdir.php?ds=asip-designer.

[99] Texas Instruments. *TMS320C6748 Fixed- and Floating-Point DSP Datasheet (SPRS590G)*. 2017.

[100] Texas Instruments. *TMS320C674x DSP CPU and Instruction Set (SPRUFE8B)*. 2010.

[101] S. Vassiliadis und D. Soudris, Hrsg. *Fine- and Coarse-Grain Reconfigurable Computing*. Springer Netherlands, 2007. DOI: 10.1007/978-1-4020-6505-7.

[102] L. Verdoscia und R. Giorgi. „A Data-Flow Soft-Core Processor for Accelerating Scientific Calculation on FPGAs“. In: *Mathematical Problems in Engineering* 2016 (2016), S. 1–21. DOI: 10.1155/2016/3190234.

[103] K. Vipin und S. A. Fahmy. „A high speed open source controller for FPGA Partial Reconfiguration“. In: *2012 International Conference on Field-Programmable Technology*. IEEE, Dez. 2012. DOI: 10.1109/fpt.2012.6412113.

[104] K. Vipin und S. A. Fahmy. „FPGA Dynamic and Partial Reconfiguration“. In: *ACM Computing Surveys* 51.4 (Juli 2018), S. 1–39. DOI: 10.1145/3193827.

[105] *Virtex-4 Family Overview*. Datenblatt. Xilinx Inc. 2004.

[106] *Virtex-5 Family Overview*. Datenblatt. Xilinx Inc. 2006.

[107] *Virtex-6 Family Overview*. Datenblatt. Xilinx Inc. 2009.

[108] *Virtex-6 FPGA Configuration User Guide*. Handbuch. Xilinx Inc. 2015.

[109] *Virtex-II Platform FPGA User Guide*. Xilinx Inc. 2007.

[110] *Virtex-II Platform FPGAs: Complete Data Sheet*. Datenblatt. Rev. 2014. Xilinx Inc. 2003.

[111] M. Wang, Y. Wang, D. Liu, Z. Qin und Z. Shao. „Compiler-assisted leakage-aware loop scheduling for embedded VLIW DSP processors“. In: *Journal of Systems and Software* 83.5 (Mai 2010), S. 772–785. DOI: 10.1016/j.jss.2009.11.727.

[112] Wave Computing. *MIPS Processors*. 14. Feb. 2019. URL: https://www.mips.com.

[113] R. Weinmann. „Verlustleistungsoptimierung von Registerzugriffen in einem Hörgeräteprozessor durch den Einsatz von genetischen Optimierungsalgorithmen“. Masterarb. Institute of Microelectronic Systems, Leibniz Universität Hannover, 2017.

[114] R. Wilhelm und D. Maurer. *Übersetzerbau*. Springer Berlin Heidelberg, 1997. DOI: 10.1007/978-3-642-59081-8.

[115] S. Wong, F. Anjam und M. F. Nadeem. „Dynamically Reconfigurable Register File for a Softcore VLIW Processor“. In: *Proc. Design, Automation and Test in Europe*. Dresden, Germany, März 2010.

[116] S. Wong, T. van As und G. Brown. „ρ-VEX: A reconfigurable and extensible softcore VLIW processor“. In: *2008 International Conference on Field-Programmable Technology*. IEEE, Dez. 2008. DOI: 10.1109/fpt.2008.4762420.

[117] S. Xiao und E. M.-K. Lai. „A Rough Programming Approach to Power-Balanced Instruction Scheduling for VLIW Digital Signal Processors“. In: *IEEE Transactions on Signal Processing* 56.4 (Apr. 2008), S. 1698–1709. DOI: 10.1109/tsp.2007.909003.

[118] S. Xiao und E. M.-K. Lai. „Instruction scheduling of VLIW architectures for balanced power consumption“. In: *Proceedings of the conference on Asia South Pacific design automation (ASP-DAC '05)*. 2005, S. 824–829. DOI: 10.1145/1120725.1121027.

[119] S. Xiao und E. M.-K. Lai. „VLIW instruction scheduling for minimal power variation“. In: *ACM Transactions on Architecture and Code Optimization* 4.3 (Sep. 2007), 18–es. DOI: 10.1145/1275937.1275942.

[120] Xilinx Inc. *LogiCORE IP XPS HWICAP (v5.00a)*. 2010.

[121] Xilinx Inc. *MicroBlaze Processor Reference Guide*. Techn. Ber. Xilinx Inc., 21. Juni 2018.

[122] Xilinx Inc. *ML605 Hardware User Guide*. 2010.

[123] Xilinx Inc. *Partial Reconfiguration User Guide*. 2011.

[124] L. Yan, B. Wu, Y. Wen, S. Zhang und T. Chen. „A Reconfigurable Processor Architecture Combining Multi-core and Reconfigurable Processing Unit“. In: *2010 10th IEEE International Conference on Computer and Information Technology*. IEEE, Juni 2010. DOI: 10.1109/cit.2010.484.

[125] X.-S. Yang. *Nature-Inspired Optimization Algorithms*. Elsevier Inc., 2014.

[126] P. Yiannacouras, J. G. Steffan und J. Rose. „Portable, Flexible, and Scalable Soft Vector Processors“. In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 20.8 (Aug. 2012), S. 1429–1442. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2011.2160463.

[127] H.-S. Yun und J. Kim. „Power-aware modulo scheduling for high-performance VLIW processors“. In: *ISLPED '01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design (IEEE Cat. No.01TH8581)*. ACM, 2001, S. 40–45. DOI: 10.1109/lpe.2001.945369.

[128] B. Zhang, K. Mei und N. Zheng. „Coarse-grained Dynamically Reconfigurable Processor for Vision Pre-Processing“. In: *Journal of Signal Processing Systems* 79.1 (Juli 2013), S. 45–61. DOI: 10.1007/s11265-013-0828-1.

[129] S. Zhong, J. Wei, W. Guo und Z. Wang. „Instruction scheduling using genetic algorithm with taboo search for TTA-like processors“. In: *International Conference on Computer Design and Applications*. IEEE, Juni 2010. DOI: 10.1109/iccda.2010.5541389.

[130] *Zynq-7000 SoC Data Sheet: Overview*. Datenblatt. Xilinx Inc. 2011.

Wissenschaftlicher Werdegang

Florian Giesemann, geboren am 27.03.1985 in Gehrden

Studium

2005–2009	**Bachelorstudium der Informatik** Leibniz Universität Hannover
2009	**Bachelorarbeit** „Untersuchung neuartiger Parallelitätsverfahren in einem generischen VLIW-Prozessor“
2009–2012	**Masterstudium der Informatik** Leibniz Universität Hannover
2012	**Masterarbeit** „Implementierung einer vollständigen Toolchain für einen dynamisch rekonfigurierbaren Softprozessor und Evaluierung mit einer exemplarischen Fahrassistenzanwendung“

Berufliche Laufbahn

2006–2012	**Hilfswissenschaftliche Tätigkeit** Institut für Mikroelektronische Systeme Leibniz Universität Hannover
2012–2018	**Wissenschaftlicher Mitarbeiter** Institut für Mikroelektronische Systeme Leibniz Universität Hannover