Marko Rößler

Dynamische Anwendungspartitionierung für heterogene adaptive Computersysteme

Marko Rößler

Dynamische Anwendungspartitionierung für heterogene adaptive Computersysteme

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Universitätsverlag Chemnitz
2014

Impressum

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Angaben sind im Internet über http://dnb.d-nb.de abrufbar.

Diese Arbeit wurde von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität Chemnitz als Dissertation zur Erlangung des akademischen Grades Dr.-Ing. genehmigt.

Tag der Einreichung	11. Dezember 2013
Betreuer	Prof. Dr.-Ing. Ulrich Heinkel
1. Gutachter:	Prof. Dr.-Ing. Ulrich Heinkel
2. Gutachter:	Prof. Dr.-Ing. habil. Christian Haubelt
Tag der Verteidigung:	21. Mai 2014

Technische Universität Chemnitz/Universitätsbibliothek
Universitätsverlag Chemnitz
09107 Chemnitz
http://www.tu-chemnitz.de/ub/univerlag

Herstellung und Auslieferung
Verlagshaus Monsenstein und Vannerdat OHG
Am Hawerkamp 31
48155 Münster
http://www.mv-verlag.de

ISBN 978-3-944640-29-7

http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-151837

Bibliographische Angaben

Dynamische Anwendungspartitionierung für heterogene adaptive Computersysteme
Marko Rößler – 2014 – 210 Seiten
60 Abbildungen, 10 Tabellen, 5 Algorithmen, 211 Literaturquellen
Technische Universität Chemnitz, Fakultät für Elektrotechnik und Informationstechnik, Dissertationsschrift

Kurzreferat

Die Dissertationsschrift stellt eine Methodik und die Infrastruktur zur Entwicklung von dynamisch verteilbaren Anwendungen für heterogene Computersysteme vor. Diese Computersysteme besitzen vielfältige Rechenwerke, die Berechnungen in den Domänen Software und Hardware realisieren. Als erster Schritt wird ein übergreifendes und integriertes Vorgehen für den Anwendungsentwurf auf Basis eines abstrakten "Single-Source" Ansatzes entwickelt. Durch die Virtualisierung der Rechenwerke wird die preemptive Verteilung der Anwendungen auch über die Domänengrenzen möglich.

Die Anwendungsentwicklung für diese Computersysteme bedarf einer durchgehend automatisierten Entwurfsunterstützung. In der Arbeit wird dazu der vorgeschlagene Ansatz formalisiert und eine neuartige Unterbrechungspunktsynthese entwickelt, die ein hinsichtlich Zeit und Fläche optimiertes, preemptives Verhalten für beliebige Anwendungsbeschreibungen generiert. Das Verfahren wird beispielhaft implementiert und mittels einer FPGA-Prototypenplattform mit Linux-basierter Laufzeitumgebung anhand dreier Fallbeispiele unterschiedlicher Komplexität validiert und evaluiert.

Schlagwörter

Hardware-Software-Codesign, High-Level-Synthese, Dynamische Partitionierung, Unterbrechungspunktsynthese, Entwurf heterogener Systeme, Adaptive Computersysteme

Inhaltsverzeichnis

Abkürzungsverzeichnis

ACS	Adaptives Computersystem
AK	Ausführungskontainer
ALAP	As Late As Possible
ALU	Arithmetic Logic Unit
ASAP	As Soon As Possible
ASIC	Application Spezific Integrated Curcuit
BMBF	Bundesministerium für Bildung und Forschung
CDFG	Control-Dataflow Graph
CFG	Control Flow Graph
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DFG	Data Flow Graph
DMA	Direct Memory Access
DSP	Digitaler Signalprozessor
E/A	Ein-/Ausgabe
EDA	Electronic Design Automation
EDK	Embedded Development Kit
EPROM	Electrical Programmable Read-Only Memory

FCFS	First Come First Serve
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSMD	Finite State Machine with Datapath
GPS	Global Positioning System
GPSV	Generalisierte Plattform zur Sensordatenverarbeitung
GPU	Graphic Processing Unit
GPZV	Generische Plattform für Systemzuverlässigkeit und Verifikation
HDL	Hardware Description Language
HLS	High-Level-Synthese
HW	Hardware
IEEE	Institute of Electrical and Electronics Engineers
ILP	Integer Linear Program
LUT	Lookup-Table
MCS	Multicomputersystem
MHS	Microprocessor Hardware Specification
MLFS	Multilevel-Feedback Scheduling
MMAP	Memory Mapping
MPS	Multiprozessorsystem
MUL	Multiplikationseinheit
NoC	Network on Chip

PLB	Processor Local Bus
PPC	Power PC
RAM	Random Access Memory
RPU	Reconfigurable Processing Unit
RT	Register-Transfer
SJF	Shortes Job First
SMP	Symmetrisches Multiprozessor-System
SoC	System on Chip
SoPC	System on a Programmable Chip
SRAM	Static Random Access Memory
SSE	Schaltkreis- und Systementwurf
SW	Software
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLSI	Very Large Scal Integrated
VNM	Von-Neumann Maschine
XPS	Xilinx Platform Studio
XUP	Xilinx University Program

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit am Lehrstuhl für Schaltkreis- und Systementwurf (SSE) der TU Chemnitz. Ich möchte Prof. Ulrich Heinkel danken, der mir die Möglichkeit gab auf dem Gebiet des kombinierten Hardware/Software-Entwurfes zu forschen, diese Arbeit anzufertigen und auf internationalen Konferenzen vorzustellen. Prof. Christian Haubelt danke ich für die Übernahme des Zweitgutachtens.

Die Idee für die Arbeit entstand im Rahmen der Projekte GPSV und GPZV, die vom Bundesministerium für Bildung und Forschung im Programm *Unternehmen Regionen - InnoProfile* unter den Förderkennzeichen 03IP505 und 03IPT505X gefördert wurden. Projektinhalt war eine generalisierte Plattform zur Sensordatenverarbeitung mit vielfältigen heterogen gelagerten Systemteilen. In diesem Zusammenhang erhielt ich erstmals tiefer gehende Einblicke in die Co-Design-Methodik.

Wesentlichen Einfluss auf das Gelingen der Arbeit hatte das Team vom SSE. Ich danke vor allem Dr. Jan Langer für die fruchtbaren Diskussion und Anregungen. Seine Gabe zur Erfassung und Vereinfachung komplexester Sachverhalte haben insbesondere hinsichtlich der graphentheoretischen Inhalte sowie der linearen Optimierungen zum Erfolg der Arbeit beigetragen. Thomas Horn danke ich für die profunden grammatikalischen Kenntnisse und sein fachliches Verständnis. Dana Linnemann sei für das Korrekturlesen gedankt. Gleichfalls danke ich Enrico Billich und Mathias Harder für das Engagement in deren Diplomarbeiten, die zur Validierung der ersten Ansätze und Ideen führten. Daniel Fross gebührt darüber hinaus Dank für seine Arbeiten zu Partikelfiltern, einem zentralen Anwendungsbeispiel dieser Arbeit. Allen anderen aktuellen und ehemaligen Kollegen soll mein herzlicher Dank gelten. Besonders hervorheben möchte ich Prof. Dietmar Müller und Dr. Vasco Jerenić. Auf deren Anregung hin wuchs in mir das Bedürfnis eine solche Arbeit zu verfassen.

Größter Dank gilt meinen Eltern, die mich zu einem neugierigen und wissensdurstigen Menschen geformt haben. Meiner Frau Silke und meinen Kindern Jannes, Arend und Henning danke ich für den Freiraum und die Nachsicht in der einen oder anderen schweren Stunde. Euch widme ich dieses Werk!

1 Einleitung

1.1 Motivation

In der Halbleiterbranche findet seit mehreren Jahrzehnten eine stete Entwicklung der Technologie hin zu kleineren Strukturbreiten und höherer Integrationsdichte statt. Dies führt zu einer höheren Leistungsfähigkeit bei geringerem Energiebedarf, sodass immer komplexere Systeme aufgebaut werden können. Das letzte Jahrzehnt ist in Folge dessen geprägt von mobilen Geräten, die tragbar und batteriegetrieben sind. Die Verkaufszahlen in Abbildung 1.1 des Endkundenmarktes belegen diesen Trend. 2012 überstieg der Absatz von Tablets[1] erstmals den von Desktop PCs und erreichte das Niveau verkaufter Notebooks [Ein13]. Im Bereich tragbarer Telefone hat das multifunktionelle Smartphone das klassische Mobiltelefon mit Sprachfunktion abgelöst und ist heute zum entscheidenden Treiber der Elektronik und IT-Branche geworden. Die beeindruckende Zahl von einer halben Milliarde verkaufter Smartphones im Jahr 2013 unterstreicht diesen Aspekt. Prognosen gehen davon aus, dass der Trend zur weiteren Funktionsintegration bei gleichzeitiger Miniaturisierung in der Elektronik, getrieben durch weitere Skalierung der minimalen Strukturbreiten der Halbleiter, anhalten wird [Com12]. Anwendungsseitig erfasst diese Entwicklung Uhren und Brillen, die durch Integration von Elektronik schlau gemacht werden und dadurch Multifunktionalität erreichen.

Die Baugröße und der Energieverbrauch sind bei dieser Entwicklung sowohl treibende als auch limitierende Faktoren. Anhand der Hauptkomponenten des Smartphones (Display, Steuer-Elektronik und Akku) lässt sich eine wesentliche Auswirkung auf die Mikroelektronik ableiten. Displays werden zukünftig faltbar sein und damit weiter an Verbaugröße verlieren. Die Firma Samsung stellte kürzlich ein rollbares Display unter dem Namen *Youm* vor [Hof13]. Weil die Energiedichte von Batterien nur mäßige Steigerungen erwarten lässt, wird künftig bei abnehmender Baugröße nur ein gleiches oder sogar reduziertes Energiebudget zur Verfügung stehen. Für die Elektronik ergibt sich daher die Notwendigkeit weiterer Integration bei größerer Energieeffizienz. In diesem Zusammenhang verfolgt beispielsweise Samsung das

[1] *Tablet (engl.)*: tragbarer flacher Computer

Ziel, die Steuer-Elektronik eines Smartphones in ein bis zwei Schaltkreisen zu integrieren, um damit für die Herausforderungen der kommenden 5 Jahre gerüstet sein zu können [Goe13]. Ähnliche Ziele haben sich die Firmen Qualcomm und Intel gesteckt. Das bekannte Schlagwort des *System on Chip* (SoC) wird damit dann umfänglich verwirklicht und das Bild der klassischen Leiterplatte in den Schaltkreis verschoben.

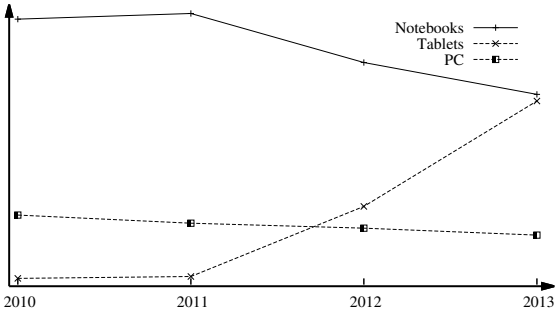
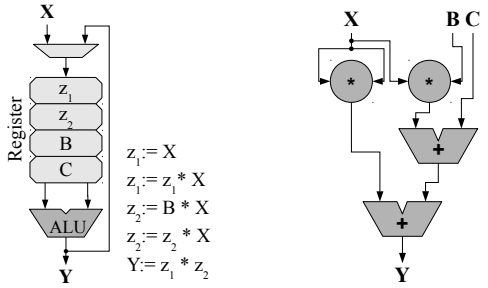


Abbildung 1.1: Verkaufszahlen von Computersystemen im Endkundenmarkt zwischen 2010 und 2013 [Ein13]

Für die Mikroelektronik führt das zwangsläufig zur Integration der analogen Komponenten sowie der Prozessoren (CPU, Grafik, Audio, Modem, u.a.) in einen Schaltkreis als heterogenes System. Die kommenden Technologieknoten unterhalb von 28 Nanometern, die Transistoren der FinFet-Technologie sowie die dreidimensionale Integration von Schaltkreisen bilden aus heutiger Sicht die Grundlagen für diese Entwicklung. Bezüglich Energieeffizienz wird dies wesentliche Verbesserungen herbeiführen, jedoch das Problem der Wärmekonzentration bzw. -abfuhr intensivieren. Einen zusätzlichen Fortschritt verspricht die tiefgreifende Integration von Hardware (HW) und Software (SW) innerhalb eines Schaltkreises, um dann durch einen intelligenten, flexiblen und anwendungsbezogenen Einsatz der heterogenen Rechenwerke die Effizienz zu steigern. Zhang, Peng, Fu und Hu [Zha+13] zeigen beispielhaft, dass durch eine optimale dynamische Verteilung von SW auf heterogenen Prozessoren rechenleistungsbezogene Energieeinsparungen von bis zu 20 Prozent erreicht werden. Weitere Spielräume ergeben sich, wenn Funktionalität, die originär in SW auf Prozessoren abläuft, durch HW-Beschleunigung unterstützt oder gänzlich auf HW abgebildet wird [CS12; Hun+09; Mit07] und

die Leistungsaufnahme bereits auf hohen Abstraktionsebenen des Entwurfes Berücksichtigung findet [Str+11].

Durch die Fortschritte im Bereich der reprogrammierbaren Logik bei *Field Programmable Gate Arrays* (FPGA)-Schaltkreisen, ist es möglich Funktionalitäten dynamisch, d.h. zur Laufzeit wechselnd, in HW auszuführen. Neben der herkömmlichen temporalen Verteilung von Operationen auf den generischen Rechenwerken eines Prozessors, wird so eine spatiale Verteilung (vgl. Abbildung 1.2) auf mehrere spezialisierte Rechenwerke innerhalb programmierbarer HW-Strukturen möglich. Ein System, das sowohl spatiale als auch eine temporale Berechnungsabbildung erlaubt und die Zuordnung zur jeweiligen Domäne erlaubt, wird als *adaptives Computersystem* (ACS) [Kas05] bezeichnet.



(a) temporal (b) spatial

Abbildung 1.2: Temporale und spatiale Verteilung von Berechnungsschritten

Einen weiteren motivierenden Aspekt bilden Computersysteme, die autonom an neuralgischen Punkten beispielsweise bei der Fahrerassistenz im Auto, der Steuerung im Kraftwerk oder der Patientenüberwachung, arbeiten und daher ausfallsicher und zuverlässig sein müssen. Ausfallsicherheit kann erzeugt werden, indem ein System tolerant gegenüber auftretenden Fehlern agiert und im Idealfall sogar selbstheilende Fähigkeiten besitzt. In einem ACS lassen sich Aufgaben zur Laufzeit dynamisch zwischen Systemteilen migrieren, sodass bei Ausfall eines Systemteils ein anderer übernehmen kann.

Beim Entwurf ausfallsicherer Systeme gilt es einen Kompromiss[2] zwischen Leistungsfähigkeit, Ressourceneinsatz und Fehlertoleranz zu finden, sodass

[2] *Kompromiss*: engl. tradeoff

sich ein dreidimensionaler Entwurfsraum aufspannt. Herkömmliche Computersysteme bilden in diesem Raum einen Punkt, weil deren Eigenschaften zum Entwurfszeitpunkt festgelegt werden. ACS ermöglichen ein, in Abbildung 1.3 exemplarisch als Kugelkörper gezeichneten, größeren Bereich, in dem dieser Tradeoff zur Laufzeit an unmittelbare Bedürfnisse der Anwendung angepasst werden kann.

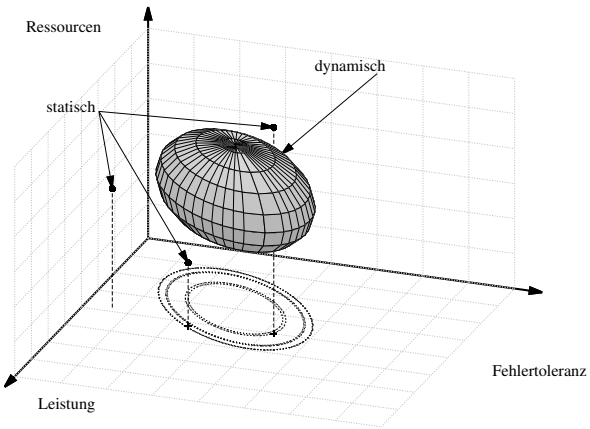


Abbildung 1.3: Entwurfsraum zuverlässiger und ausfallsicherer Computersysteme [DSN05]

Die Entwicklung von Anwendungen für ACS ist eine komplexe Herausforderung aus einer Kombination des SW- und des HW-Entwurfs. Der Entwurf auf hohen Abstraktionsebenen ist dabei die wesentliche Grundlage, wobei durch die Abstraktion eine Auftrennung von Technologie- und Prozessordetailwissen von der eigentlichen Anwendungsentwicklung erreicht wird. Dadurch wird eine gleichbleibend hohe Entwurfsproduktivität bei steigender Komplexität möglich. Bis heute erfolgte die Abstraktion jedoch separat innerhalb der HW- und SW-Bereiche. Eine durchdringende Integration beider Entwurfswelten ist heute noch nicht gegeben. Der Entwickler ist gezwungen den Datenaustausch zu organisieren, die korrekte Datenformatierung zu beachten, sowie die Synchronisation des Ablaufes und die Festlegung bzw. Suche nach geeig-

neten Migrationszeitpunkten zur Lastverteilung zu bewerkstelligen. Das ist fehleranfällig und zeitaufwändig. Gegenstand der Forschung ist es, für ACS die Abstraktion in beiden Entwurfsdomänen auf ein gemeinsames Niveau zu heben. Ziel muss es sein, den Anwendungsentwurf über die HW/SW-Grenze hinweg durch eine entsprechende Entwurfsumgebung weiter zu automatisieren und zu optimieren.

1.2 Ziele der Arbeit

Im Rahmen der Arbeit soll eine Entwurfsmethodik und eine Infrastruktur entwickelt werden, die es erlaubt Anwendungen für ein ACS zu entwickeln und bei der Ausführung dynamisch zu verteilen. In ACS sind dazu generische Multifunktionsprozessoren und programmierbare HW-Strukturen als heterogene Rechenwerke integriert. Das Konzept prozessorzentrischer Computer [HP07; CSG99; Sin07; Tan06] mit der Ausführung in der SW-Domäne wird durch die vorliegende Arbeit um die Domäne der HW erweitert, sodass die Anwendung auf einem System aus kooperierenden HW- und SW-Komponenten bearbeitet wird und dazu die Rechenwerke des ACS gegenüber der Anwendungsschicht virtualisiert werden.

Die parallelen Ausführungsstränge der Anwendungen sollen dynamisch über die Rechenwerke des ACS verteilt und bei Bedarf umverteilt bzw. migriert werden können. Dies soll eine prozessspezifische Optimierung zeitkritischer, sicherheitsrelevanter oder ressourcenintensiver Anwendungsprozesse bezüglich Performance, Ressourcenbedarf und Leistungsaufnahme zur Laufzeit ermöglichen. Die Entwurfsmethodik soll sich dabei weitgehend an der Programmierung herkömmlicher Rechnersysteme orientieren und Standardwerkzeuge sowie bekannte Programmiersprachen nutzen. In der Arbeit wird dazu von einer *C*-basierten Anwendungsbeschreibung ausgegangen. Diese Beschreibung soll in Form parallel ablaufender Teilaufgaben, als *Prozesse* oder *Threads* [3], gegliedert sein. Ein Übersetzungs- und Synthesewerkzeug transformiert diese Beschreibung in einen Code, der vom heterogenen ACS interpretiert werden kann. Die Transformation muss spezifisch für jeden Rechenwerkstyp geschehen. Darüber hinaus werden bei der Transformation spezielle Strukturen in den Code injiziert, die die Grundlage des domänenübergreifenden Managements der Teilaufgaben bilden. Dazu gehört unter anderem das Starten, Anhalten, Umlagern/Migrieren und Töten der Teilaufgaben.

[3] *Threads (engl.)*: Programmfäden, parallele Kontrollflüsse in einem Programm

Im Rahmen der Arbeit sind geeignete Methoden zu finden und gegebenenfalls zu entwickeln, die sowohl die Übersetzung als auch die Injektion der Management-Strukturen realisiert. Es ist eine heterogene Systemarchitektur für ein ACS zu entwerfen, die über Infrastruktur zur domänenübergreifenden Verarbeitung verfügt. Notwendig sind neben dedizierten Rechenwerken auch Komponenten zur Speicherung von Daten, Kommunikationskanäle zum Transport der Daten, sowie die Anbindung der gesamten Architektur an die Umwelt über Ein- und Ausgabeschnittstellen. Ein weiterer Teil der Infrastruktur muss die internen Rekonfigurationsprozesse der programmierbaren Rechenwerke steuern. Eine wesentliche Aufgabe besteht in der dynamischen Systemverwaltung der Teilaufgaben. Ein eingebettetes Betriebssystem ist für das Management der Rechenwerke und als Laufzeitumgebung der ACS zu realisieren. Der Entwurf der HW-Strukturen und die Implementierung einer minimalistischen SW-Steuerschicht für das ACS stellt einen wesentlichen Teil der Arbeit dar.

Zum Abschluss der Arbeit soll das Gesamtwerk anhand von Beispielen qualifiziert und eingeordnet werden. Dabei soll der Nachweis geführt werden, dass durch den Entwurf auf hoher Abstraktionsebene eine geeignete Methodik und ein effizienter Weg zur Programmierung einer adaptiven, heterogenen Rechnerarchitektur gegeben ist. Die Validierung der Ergebnisse soll auf einem heterogenen *System on a programmable Chip* (SoPC) als prototypisches ACS erfolgen. Die Ziele und der Ansatz der Arbeit wurden in [Röß09; RH08] veröffentlicht.

Zentrale Fragen:

1. Wie kann ein integrierter Entwurfsablauf für eine heterogene adaptive Systemarchitektur gestaltet werden und welche einheitliche Beschreibungssprache ermöglicht die zugehörige Anwendungsentwicklung auf hoher Abstraktionsebene?
2. Welche Architektur ist für diesen Zweck geeignet und welche Abhängigkeiten bestehen zum Entwurfsprozess?
3. Wie kann Unterbrechung und Migration von Anwendungen und Anwendungsteilen domänenübergreifend realisiert werden?
4. Welche Kosten im Hinblick auf Laufzeit, Latenz und Ressourceneinsatz entstehen durch die Unterbrechungs- und Migrationsfähigkeit einer Anwendung?
5. Wie ist eine Laufzeitumgebung auf der adaptiven Rechnerarchitektur zu gestalten und anhand welcher Algorithmen lassen sich eine beschränkte Anzahl heterogener Rechenwerke verwalten?

6. Welche Anwendungen eignen sich für die Evaluation der entwickelten Entwurfsmethodik und Infrastruktur?

1.3 Aufbau der Arbeit

In Kapitel 2 werden die theoretischen Grundlagen im Zusammenhang mit heterogenen ACS behandelt. Dazu wird zunächst auf die parallele Rechentechnik mit Vertiefung auf der Struktur der Rechenwerke in den Domänen HW und SW eingegangen. Anschließend wird der Entwurf auf hoher Abstraktionsebene mittels High-Level-Synthese (HLS) als Basis der in der Arbeit entwickelten Entwurfsmethodik vorgestellt. Schließlich werden parallele Programmierkonzepte eingeführt, sowie mögliche Strategien zur Ablaufplanung[4] für Laufzeitumgebungen diskutiert. Im darauf folgenden Kapitel 3 wird der Stand der Technik auf den Gebieten Anwendungsentwurf, Laufzeitumgebung und Anwendungsunterbrechung/-migration vorgestellt und mit den in dieser Arbeit vorgestellten Ansätzen verglichen.

Kapitel 4 stellt den vorgeschlagenen Entwurfsablauf für dynamisch verteilbare Anwendungen auf einem heterogenen Computersystem vor, sodass im anschließenden Kapitel 5 auf die automatisierte Umsetzung von dynamisch migrier- und unterbrechbaren Anwendungsteilen eingegangen werden kann. Dabei werden Problemdefinition und -lösung bei der Synthese eines optimalen Satzes von Unterbrechungspunkten präsentiert.

Für ein heterogenes ACS bedarf es einer speziellen Laufzeitumgebung, deren Anforderungen und mögliche Umsetzung in Kapitel 6 untersucht werden. Kapitel 7 umfasst die Implementierung der vorgestellten Methodik und beispielhafte Umsetzung auf einem ACS. Weiterhin werden die notwendigen HW-Erweiterungen zur Unterstützung der Unterbrechungsfähigkeit sowie eine speziell entwickelte Simulationsumgebung für heterogene ACS beschrieben. In Kapitel 8 werden die Ergebnisse der Arbeit in Bezug auf Anwendungsbeispiele unterschiedlicher Komplexität und Struktur diskutiert. Im letzten Kapitel 9 wird die Arbeit zusammengefasst und ein Ausblick auf weitere Entwicklungsmöglichkeiten gegeben.

[4] *Ablaufplanung*: engl. Scheduling

2 Grundlagen

In diesem Kapitel werden grundlegende theoretische Aspekte und Problemstellungen der parallelen Rechentechnik für die Domänen HW und SW aufgezeigt. Anschließend wird der Entwurf und die Synthese von Verhaltensbeschreibungen für beide Domänen zur automatisierten Abbildung von Anwendungen auf HW/SW-Systeme diskutiert. Schließlich werden relevante Programmiermodelle für parallele Anwendungen und Schedulingstrategien für HW/SW-Systeme vorgestellt.

2.1 Parallele Rechentechnik

Zeitgemäße Rechner besitzen mehrere gleichzeitig (parallel) arbeitende Rechenwerke mit dem Ziel, einen Algorithmus oder eine Anwendung schneller auszuführen, als es mit einer einzigen Einheit möglich wäre. Systeme, die mehrere Operationen parallel bearbeiten können, werden parallele Rechensysteme genannt. Parallelität ist auf den Ebenen der Befehle, der Rechenwerke und der Systeme erreichbar. Dabei gilt, das mit höherer Organisationsebene größerer Performanzgewinn zu erwarten ist. Durch Maßnahmen auf Befehlsebene, d.h. innerhalb der Rechenwerke, kann eine fünf- bis zehnfache Beschleunigung erreicht werden. Durch verschaltete Rechenwerke sind Faktoren von 50 bis 100 und mehr zu erreichen [Tan06]. Die verteilte Berechnung auf mehreren Computersystemen erzeugt theoretisch unbegrenzte Gewinne, denen allerdings Schranken durch die Parallelisierbarkeit der Anwendung und die Leistungsfähigkeit des Kommunikationsnetzes gesetzt sind [Amd67]. Im Allgemeinen fallen alle Computersysteme mit hoher Rechenleistung, wie zum Beispiel Server, Großrechner, Hochleistungsrechner und Massiv-parallele Computer (Supercomputer), in die Kategorie paralleler Rechner. Spätestens seit dem Einzug von *Multicore Prozessoren*[1] gehören Arbeitsplatzrechner, tragbare Rechner und teilweise sogar eingebettete Systeme[2] dazu. Außerdem

[1] *Multicore Prozessor (engl.):* Schaltkreis, der Varianten mehrerer Prozessorkerne integriert

[2] *eingebettetes System*: engl. embedded system

zählen Systeme mit mindestens einem unabhängig arbeitenden Spezialprozessor (z.B. für Grafikberechnung, Bildkodierung oder Kryptographie) zur Kategorie paralleler Rechensysteme.

In den folgenden Unterkapiteln wird auf die strukturellen Eigenschaften paralleler Rechensysteme und deren Rechenwerke eingegangen. Den Abschluss der Kapitels bildet die Diskussion und Einordnung der Varianten zum Aufbau eines heterogenen ACS.

2.1.1 Domänen der Berechnung

Für Computersysteme gibt es zwei grundsätzliche Wege eine Anwendung bzw. einen Algorithmus zu realisieren [CH02]. Diese leiten sich aus der Struktur des Rechenwerkes ab. Beim ersten Prinzip erfolgt die Berechnung *in Hardware* mit fest verdrahteter Struktur, in der die Abfolge der Operationen inhärent festgelegt ist. Die Synthese der Schaltungsstruktur zur Berechnung einer Anwendung, kennzeichnet die HW-Domäne. Verbreitet sind *maskenprogrammierbare anwendungsspezifische Schaltkreise*[3] [HM04] die speziell für die Bearbeitung eine Anwendung entworfen und gefertigt werden. Die Abfolge der Instruktionen des gefertigten Schaltkreises ist nicht veränderbar. ASIC sind gekennzeichnet durch hohe Berechnungsgeschwindigkeit mit darauf bezogen geringer Leistungsaufnahme sowie hohe Entwurfskosten und äußerst geringe Flexibilität bei der Modifikation des Algorithmus nach der Fertigung des Systems.

Das zweite Prinzip berechnet den Algorithmus *in Software* auf programmierbaren *Prozessoren*, die eine durch das Programm festgelegte Folge von Befehlen ausführen. Dazu zählen unter anderem Universalprozessoren[4], Grafikprozessoren[5] und digitale Signalprozessoren (DSP). Ein maschinen- bzw. prozessorspezifischer Übersetzer[6] erzeugt den Programmcode einer Anwendung als Ablaufplan wohldefinierter Befehle. Durch Änderungen des SW-Programms kann die Befehlsfolge und damit der berechnete Algorithmus modifiziert werden, ohne Veränderungen in der HW-Struktur vorzunehmen zu müssen. Diese Lösung besitzt eine hohe Flexibilität und geringere Entwurfskosten, da auf Standardprodukte im Massenmarkt zurückgegriffen werden kann. Allerdings

[3] *anwendungsspezifischer Schaltkreis*: engl. Application Specific Integrated Curcuit (ASIC)
[4] *Universal- oder Zentralprozessor*: engl. Central Prozessing Unit (CPU)
[5] *Grafikprozessor*: engl. Graphic Processing Unit (GPU)
[6] *Übersetzer*: engl. compiler

ist die Berechnungsgeschwindigkeit und die bezogene Leistungsaufnahme einer Berechnung mittels maskenprogrammierbarem ASIC unterlegen.

Mit rekonfigurierbaren HW-Komponenten, die nach Herrmann und Müller [HM04] als Klasse der anwenderprogrammierbaren ASIC bezeichnet werden, lassen sich Rechenwerke realisieren, die günstige Eigenschaften der zuvor genannten Prinzipien kombinieren. Aufgrund der Berechnungskapazität haben FPGA dabei besondere Relevanz. Verglichen mit einer reinen SW-Lösung ist mit rekonfigurierbarer HW eine höhere Berechnungsleistung möglich und gleichzeitig eine weitaus größere Flexibilität als bei einer fest verdrahteten HW-Lösung gegeben. Streng gesehen gehören diese Rechenwerke in die Domäne der HW und sind damit begrifflich zunächst nicht vom ersten Prinzip unterscheidbar.

	Masken-programmier-barer ASIC	**Rekonfigurierbare Hardware (anwenderpro-grammierbarer ASIC)**	**Prozessor**
Ausführungs-domäne	Hardware	Hardware	Software
Art der Berechnung	datenorientiert	daten- und befehlsorientiert	befehlsorientiert
Bindungszeit-punkt der Operationen	statisch zum Fertigungszeit-punkt	statisch zum Ladezeitpunkt oder dynamisch zur Laufzeit	dynamisch zu jedem Taktzyklus zur Laufzeit
Kosten für den Anwendungsent-wurf	hoch	mittel	gering
Flexibilität nach der Fertigung	gering	mittel	hoch
Berechnungs-leistung	hoch	hoch	mittel

Tabelle 2.1: Vergleich der Realisierungsmöglichkeiten von Rechenwerken

Verallgemeinert kann ein Rechenwerk als eine Menge von Operationen angesehen werden, die auf einen Satz von Operanten abgebildet werden [Ros+09]. Der Zeitpunkt dieser Abbildung bzw. der Bindung ist ein geeignetes Kriteri-

um, die drei vorgestellten Realisierungen von Rechenwerke zu unterscheiden. Tabelle 2.1 zeigt die diskutierten Eigenschaften der Rechenwerke und die Zuordnung zur Ausführungsdomäne.

2.1.2 Parallelität der Software-Domäne

Die Bandbreite verfügbarer Prozessoren der SW-Domäne ist riesig und erstreckt sich in vielfältigen Variationen und Spezialisierungen über sämtliche Anwendungsnischen vom eingebetteten System bis zu Höchstleistungs-Rechnerverbünden. Zur Unterscheidung der Prozessoren bestehen nach Teich [Tei97] drei wesentliche Kriterien:

- Vielzweckmäßigkeit versus Anwendungsspezifik,
- Schaltkreis versus Layoutzelle[7]
- sowie der Grad an Konfigurierbarkeit.

Ein weiteres Kriterium bildet der Grad der Parallelität bei Daten- und Befehlsverarbeitung [HM04]. Die grundlegenden Prinzipien paralleler Verarbeitung in Prozessoren sowie daraus ableitbare parallele Rechnerarchitekturen sind Gegenstand der folgenden Ausführungen.

2.1.2.1 Parallelisierung auf Befehls- und Operationenebene

Diese niedrigste Ebene der Parallelität wird innerhalb eines Prozessors umgesetzt und soll im Folgenden betrachtet werden. Die Struktur eines Prozessors gliedert sich in einen Daten- und einen Steuerpfad, die über einen Bus an wahlfreiem Speicher[8] (RAM) für Daten und Befehle angebunden sind. Der Datenpfad besteht aus Rechenwerken für Arithmetik und Logik sowie Registersätzen zur Speicherung von Zwischenergebnissen. Der Steuerpfad übernimmt das Laden und Dekodieren der Befehle und führt die dekodierten Operationen durch Steuerung des Datenpfades aus. Der einfachste Prozessor arbeitet die Befehle sequentiell ab und transformiert pro Rechenschritt genau ein Datenobjekt. Ein Beispiel hierfür ist die klassische Von-Neumann Maschine (VNM) *[NBG89]*, bei der ein Rechenschritt (Befehlsbearbeitungszyklus) aus den in Abbildung 2.1 dargestellten Phasen: *Befehl holen – Befehl dekodieren und Operanten holen – Befehl ausführen – Resultat abspeichern* besteht.

[7] *Layoutzelle*: engl. core
[8] *Speicher mit wahlfreiem Zugriff*: engl. Random Access Memory (RAM)

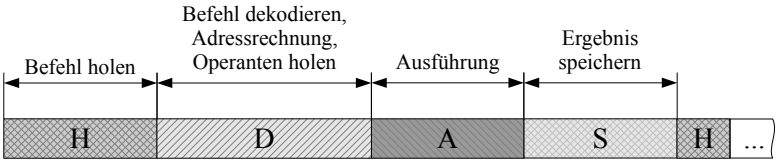


Abbildung 2.1: Befehlsbearbeitungszyklus einer Von-Neumann-Maschine

Fließbandbearbeitung[9], als grundlegende Art paralleler Datenverarbeitung, wird zur Parallelisierung auf Befehlsebene eingesetzt. Auf einem Fließband (engl. Pipeline) werden die Phasen eines Arbeitsganges überlappend ausgeführt. Im Idealfall (einer unendlich langen Warteschlange am Eingang der Pipeline) wird die Produktivität um die Anzahl der Pipeline-Stufen k gesteigert. Die Anzahl der Stufen ist begrenzt durch die maximale Granularität der Bearbeitung. Bezogen auf die Granularität des Befehlsbearbeitungszyklus einer VNM lässt sich eine vierstufige *Befehls-Pipeline* (vgl. Abbildung 2.2) aufbauen und dessen Leistung um diesen Faktor steigern. Der reale Produktivitätsgewinn wird jedoch durch Konflikte im Kontrollfluss, im Datenfluss oder in der Verfügbarkeit geteilter Ressourcen eingeschränkt. Ein Kontrollflusskonflikt besteht, wenn ein Sprungbefehl von der Beendigung bzw. Ergebnis einer vorherigen Berechnung abhängt. Das erzwingt das Leerlaufen der Pipeline und zieht im ungünstigsten Fall eine Verzögerung um $k - 1$ Takte nach sich. Die spekulative Vorhersage des Sprungziels[10] wirkt gegen diesen Effekt. Mit dynamischen Prädiktionsalgorithmen werden bis zu 83 % der Sprungziele erkannt [MH86]. Für arithmetische Operationen lassen sich mittels Pipelining gleichfalls Geschwindigkeitssteigerungen erzielen [Gil93], wenn die betreffende Operation in gleichartige Teiloperationen zerlegbar ist. Bei Multiplikationen ist dies zum Beispiel möglich [XW95]. *Operation-Pipelines* eignen sich besonders für Fließkomma-Recheneinheiten[11] [Gup+91].

[9] *Fließbandverarbeitung*: engl. pipelining
[10] *Sprungzielvorhersage*: engl. branch prediction
[11] *Fließkomma-Recheneinheit*: engl. Floating Point Unit (FPU)

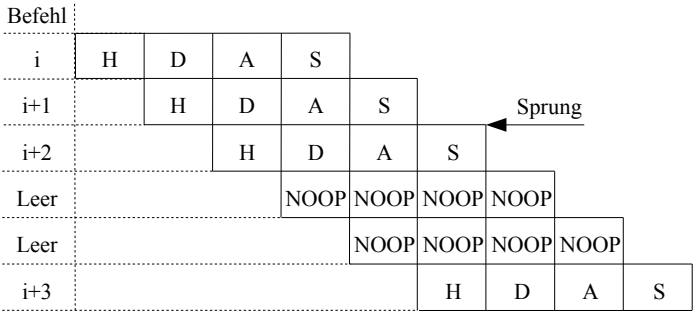


Abbildung 2.2: Leerlaufen einer Befehls-Pipeline ($k = 4$) nach einem Spungbefehl

Nebenläufigkeit ist eine weitere Facette von Parallelität. Sie wird auch als *echter Parallelismus* bezeichnet und unterscheidet sich vom Pipelining. Bezogen auf ein bestimmtes Abstraktionsniveau liegt Nebenläufigkeit genau dann vor, wenn ein Rechner gleichzeitig Aktionen ausführt, die vollständig auf diesem Niveau definiert wurden. Zum Beispiel erzeugen superskalare Prozessorarchitekturen [AC87] mit mehrfachem Befehls-Pipelining eine Nebenläufigkeit auf der Befehls- bzw. Operationenebene [Tan06]. Abbildung 2.3 zeigt diesen Ansatz. In der zeitlich längsten Stufe der Pipeline (Befehlsausführung) werden mehrere Befehle gleichzeitig an vielfache Funktionseinheiten übergeben. Funktionseinheiten sind beispielsweise Rechenwerke für Arithmetik[12] und Fließkommarechnung oder Komponenten zum Laden oder Speichern im Hauptspeicher.

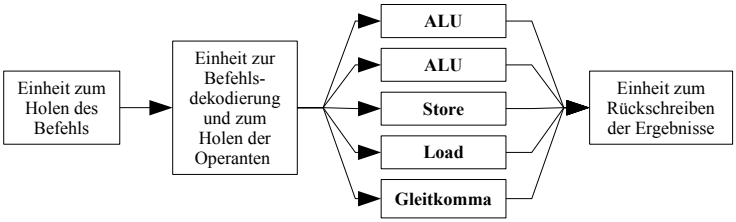


Abbildung 2.3: Superskalare Pipeline eines Mehrzweckprozessors

[12] *Arithmetik-Rechenwerk*: engl. Arithmetic Logic Unit (ALU)

Die Ressourcen eines superskalaren Prozessors werden von einem Programm allerdings nur teilweise ausgelastet, d.h. Funktionseinheiten bleiben ungenutzt, weil beispielsweise eine Häufung von kostspieligen Speicherzugriffen notwendig ist. Eine Leistungssteigerung kann über die Ausnutzung dieser Freiräume erzielt werden, indem eine zweite Anwendung gleichzeitig ausgeführt wird. Diese Technik wird als *Multithreading* oder *Hyperthreading* bezeichnet [KM03] und ist nochmals in fein- und grobgranulares Multithreading [LGH94; Squ+96] zu unterscheiden. Die Verwaltung eines zweiten Programmflusses erfordert die Duplikation von Befehlszähler, Registersatz und Unterbrechungsverwaltung[13] [Tan06] sowie einen geeigneten Zuteilungsalgorithmus der gemeinsam genutzten Funktionseinheiten auf die Programmflüsse. Aus Sicht von Betriebssystem und Anwender arbeitet in einem Multithread-Prozessor ein zweites unabhängiges Rechenwerk. Systolische Felder [KL79] realisieren eine weitergehende Parallelisierung des Pipeline-Prinzips nach dem *Anti-Maschinen Paradigma* [Har02b].

Für die effiziente Ausnutzung von Parallelisierung auf Anweisungsebene werden neben der HW-Unterstützung auch geeignete Compiler benötigt. Deren Aufgabe ist die automatische Erkennung und Ausbeutung möglicher Parallelität in den Anwendungsprogrammen. Die Datenabhängigkeit zwischen Operationen wird innerhalb eines Blockes über Verzweigungen hinweg und innerhalb aufeinanderfolgender Schleifen analysiert und daraus ein kontinuierlicher Operations-Fluss mit bestmöglicher Auslastung der Funktionseinheiten erzeugt. Ähnliche Optimierungsprobleme spielen auch bei der Verhaltenssynthese von HW-Beschreibungen eine Rolle und werden in Kapitel 2.2.4 näher erläutert.

2.1.2.2 Parallelisierung auf Prozessorebene - Multiprozessor- und Multicomputersysteme

Durch das Koppeln von Prozessoren lassen sich Rechensysteme mit erhöhter Berechnungsleistung konstruieren. Dabei wird in *lose* und *eng* gekoppelte Systeme unterschieden.

Multiprozessorsysteme (MPS) koppeln Prozessoren *eng* zu einer Gruppe um einen gemeinsamen physischen Hauptspeicher. *Homogene MPS* integrieren Mehrzweckprozessoren, die von einer einzigen Betriebssysteminstanz verwaltet werden und Berechnungsaufgaben höherer Granularität (auf Thread-, Prozess- oder Anwendungsebene) nebenläufig bearbeiten. Haben

[13] *Unterbrechungsverwaltung*: engl. interrupt controller

alle Prozessoren die gleiche Sicht auf das System und eine Äquidistanz zum gemeinsamen Speicher, wird von einem symmetrischen Multiprozessor-System[14] (SMP) [CSG99] gesprochen. Die Leistungssteigerung von Computern wurde in den letzten Jahren durch die Integration von mehreren (zwei bis zu zehn) Prozessorkernen und deren Zwischenspeicher[15] innerhalb eines Schaltkreises erreicht, die als *Multicore* bezeichnet werden. Aktuell findet der Übergang von *Multicore-* zu *Manycore-Schaltkreisen*[16] mit mehr als 40 Prozessoren statt [Kon+09].

Bei *heterogenen MPS* werden neben den Mehrzweckprozessoren zusätzlich Spezialprozessoren integriert, zum Beispiel zur Bearbeitung von Netzwerkprotokollen, kryptographischen Algorithmen oder Multimediainhalten. Leistungssteigerung entsteht durch die Entlastung des Mehrzweckprozessors durch die Spezialprozessoren, die nebenläufig Algorithmenteile berechnen. MPS werden beim aktuellen Stand der Halbleitertechnologie oft als SoC integriert. Prozessoren, Fließkommaeinheiten, Cache, Speichermanagement, Kommunikationsschnittstellen und rekonfigurierbare Logik werden beim SoC um eine zentrale Kommunikationsstruktur angeordnet. Die Kommunikation innerhalb des Schaltkreises erfolgt über hierarchische Bussysteme [Fly97] oder Netzwerke[17].

Lose gekoppelte Multicomputersysteme (MCS) haben keinen gemeinsamen Speicher auf der Architekturebene. MCS stellen einen Rechnerverbund[18] aus zusammenarbeitenden Einzelrechnern [Pfi98] dar, der über ein hochleistungsfähiges Kommunikationsmedium vernetzt wird. Jeder Verbundteilnehmer ist ein vollständiger Rechner mit eigenem Betriebssystem, ggf. auch ein MPS. Der Datenaustausch zwischen den Teilnehmern ist nachrichtenbasiert. MCS überwinden damit die Einschränkung der schlecht skalierenden MPS mit einer Größe von bis zu 256 dedizierten Prozessoren [Tan06]. Ab dieser Größe relativiert die Wahrscheinlichkeit von blockierenden Speicherzugriffen den Leistungsgewinn eines zusätzlichen Prozessors [HP07].

2.1.3 Speicherorganisation in parallelen Rechnersystemen

Anhand der Speicherorganisation werden parallele Rechner in zwei Hauptklassen unterschieden: *Verteilter Speicher* und *Gemeinsamer Speicher* [CSG99;

[14] *symmetrisches Multiprozessor-System*: engl. symmetric multiprocessor
[15] *Zwischenspeicher*: engl. Cache
[16] *Manycore (engl.)*: Schaltkreise mit mehr als 10 Prozessorkernen
[17] *Netzwerk innerhalb eines Schaltkreises*: engl. Network on Chip (NoC)
[18] *Rechnerverbund*: engl. cluster

HP07]. In Rechnern mit verteiltem Speicher hat jedes Rechenwerk eigenen lokalen Speicher, mit dem es unabhängig arbeitet. Ein dediziertes Kommunikationsnetzwerk realisiert den Datenaustausch nachrichten- oder paketbasiert zwischen den Speichern. Es besteht kein Kohärenzproblem zwischen den Speichern, weil sich jede Änderung ausschließlich lokal auswirkt. Bei *gemeinsamen Speicher*[19] haben alle Rechenwerke über einen globalen Adressraum ein identisches Bild vom einem Speicher. Die Rechenwerke arbeiten unabhängig voneinander und Änderungen im Speicher sind für alle unmittelbar sichtbar. Für den Systemanwender sind die vorgestellten Speichermodelle letztendlich nicht sichtbar. Die über der HW liegenden Schichten (Betriebssystem und Bibliotheken) schaffen aus der Perspektive des Anwenderprogramms eine einheitliche Sicht auf den parallelen Rechner.

Bereits in den 80er Jahren erfolgte die Einführung eines hierarchisch aufgebauten Speichersystems, das als ein einheitlicher virtueller Speicher dem Programm sichtbar ist. Zunächst begründet durch Speichertechnologien mit unterschiedlichen Zugriffsgeschwindigkeiten wurden später, mit der Möglichkeit Speicherzellen in größerem Umfang in den Prozessorschaltkreis zu integrieren, ein "nahe" dem Prozessor gelegener Cache als zweite Speicherschicht realisiert. Kleiner als der Hauptspeicher, hält ein Cache zum Zwecke des schnelleren Zugriffs für die aktuelle Berechnung Teile der Daten und Befehle bereit. Gefüllt wird der Cache, bei Zugriffsfehlern[20], direkt aus dem Hauptspeicher. Eine Ersetzungsstrategie verwirft längere Zeit ungenutzte Einträge nach dem Verdrängungsprinzip, wenn die Kapazitätsgrenze des Cache erreicht ist. Sind mehrere Cache-gestützte Rechenwerke mit einem Hauptspeicher verbunden besteht ein Kohärenzproblem. Speicheranforderungen können gleichzeitig erteilt werden und sich gegebenenfalls überlappen bzw. überholen. Es muss durch spezielle Maßnahmen eine definierte Reihenfolge des Sichtbarwerdens von Schreiboperationen, die *Konsistenz des Speichers,* sichergestellt werden [AG96].

2.1.4 Rechentechnik mit rekonfigurierbaren Hardwarestrukturen

2.1.4.1 Klassifikation rekonfigurierbarer Strukturen

Die Architektur rekonfigurierbarer Strukturen lässt sich anhand spezifischer Eigenschaften: Granularität, Konfigurationsarten, Art des Konfigurationsspeichers und die Form der Systemintegration klassifizieren [CH02;

[19] *gemeinsamen Speicher*: engl. shared memory

[20] *Zugriffsfehler auf den Zwischenspeicher*: ein angeforderter Wert nicht im Cache verfügbar (engl. cache miss)

Org05; Bob07] bzw. unterscheiden. Im Folgenden sollen diese Eigenschaften grundlegend eingeführt werden, um für die Arbeit relevante Realisierungsvarianten abzuleiten. Für einen detaillierten Überblick am Markt verfügbarer rekonfigurierbarer Systeme, Schaltkreise und Strukturen sei auf die Arbeiten [Org05; Bob07; Ros+09] verwiesen.

Granularität Rekonfigurierbare Logikstrukturen unterscheiden sich in der Größe ihrer Basiselemente, die in *feinkörnig* und *grobkörnig* unterteilt werden. Feinkörnige Strukturen konfigurieren die HW auf Bitniveau. Sowohl das Routing von Signalen als auch deren logische Verknüpfung und deren Speicherung kann bitweise beeinflusst werden kann. Damit lassen sich Systeme mit extrem hoher Flexibilität realisieren, die praktisch Funktionen jeglicher Komplexität abbilden können. Eine Begrenzung bildet hierbei lediglich die Menge an zur Verfügung stehenden Ressourcen. Der Nachteil besteht in der sehr großen Menge an Konfigurationsbits und notwendiger Siliziumfläche. Die Dauer für das Laden einer Konfiguration ist darüber hinaus direkt von der Anzahl der Konfigurationsbits abhängig und kann bis zu mehreren Millisekunden betragen. In diese Kategorie fallen praktisch alle verfügbaren FPGA-Schaltkreise der Firmen Actel [Act13], Atmel [Atm13], Lattice [Lat13] und Xilinx [Xil13a].

Grobkörnige Architekturen begegnen diesem Problem, indem die Konfiguration auf größere funktionelle Verarbeitungseinheiten[21] beschränkt wird. Dazu werden Makros von Funktionseinheiten, wie ALUs oder Multiplikatoren, Block-RAM und Verbindungseinheiten fest integriert und ausschließlich die Verbindungen zwischen diesen variabel gestaltet. Die Schaltkreisfläche wird damit effektiver genutzt und die Anzahl der Konfigurationsbits ist geringer als bei feinkörnigen Architekturen. Grobkörnige Architekturen werden meist für den Aufbau von Datenflussmaschinen genutzt. Kommerzielle Beispiele sind der Xputer [HHW90], die XPP-Plattform [Lod+06] der Firma PACT und das PicoArray [Pan+06] der Firma Picochip.

Konfigurationsarten Die Konfiguration einer Struktur, auch Kontext genannt, wird bei programmierbarer HW im Konfigurationsspeicher gehalten. Als *Single-Context*[22] wird eine Struktur bezeichnet, wenn ein einziger Konfigurationsspeicher vorgesehen ist. Bei Rekonfiguration wird die Funktion der HW für den gesamten Ladevorgang der neuen Konfiguration unterbrochen.

[21] *Verarbeitungseinheit*: engl. Processing Element
[22] *Single-Context (engl.)*: einfacher Konfiguraionsspeicher

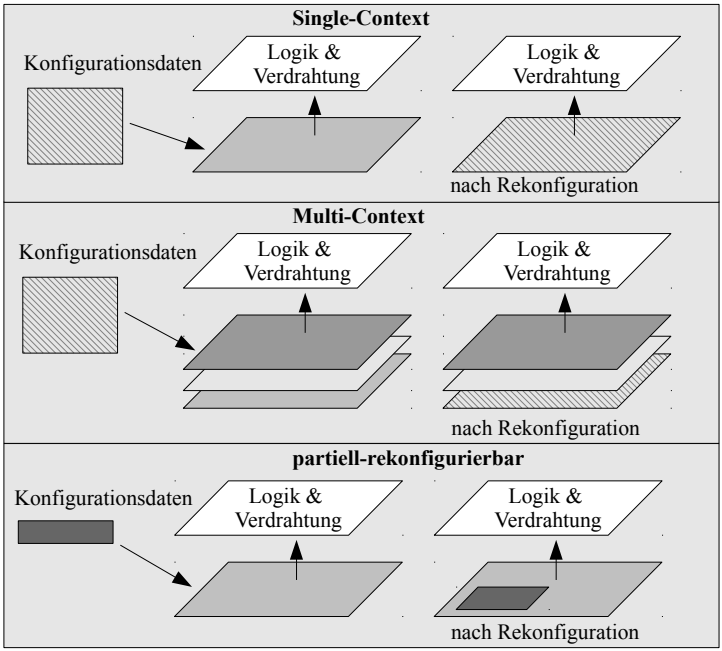


Abbildung 2.4: Konfigurationsarten bei konfigurierbarer HW-Strukturen nach Compton und Hauck [CH02]

Multi-Context[23] Strukturen umgehen dieses Problem, indem der Speicher mehrfach ausgeführt ist und während des Betriebs eines Kontextes ein zweiter vorgeladen werden kann. Das Umschalten zwischen Kontexten kann dadurch innerhalb weniger Takte erfolgen. Bei partiell rekonfigurierbaren Strukturen kann der Konfigurationsspeichers teilweise neugeladen werden, während die Struktur des unangetasteten Teils operationsfähig bleibt. Die Konfigurationsart stellt ein Unterscheidungsmerkmal konfigurierbarer Strukturen dar [CH02]. Abbildung 2.4 illustriert die möglichen Arten. Für den Aufbau einer heterogenen ACS sind kurze Rekonfigurationszeiten notwendig. Die kürzesten Zeiten erreichen Multi-Context Strukturen. Die partielle

[23] *Multi-Context (engl.)*: Konfigurationsspeicher für mehrere Kontexte die gleichzeitig gehalten werden

Rekonfigurierbarkeit ist für große Strukturen günstig, um eine feingliedrige Nutzung zu ermöglichen.

Systemintegration Das Klassifikationsmerkmal Systemintegration unterscheidet, auf welche Art die (re-)konfigurierbare Struktur in das System eingebunden ist und welche Teile des Systems ggf. innerhalb der konfigurierbaren Struktur realisiert sind. Dieses Merkmal kennzeichnet die Grenze zwischen System und konfigurierbarer Struktur, die innerhalb des Schaltkreises oder außerhalb auf der Leiterplatte verlaufen kann. Es lassen sich die in Tabelle 2.2 aufgeführten Klassen unterscheiden. Integriert ein System mehrere konfigurierbare Strukturen, können Mischformen dieser Klassen entstehen.

Typ der Systemintegration	**Beschreibung und Beispiele**	**Verlauf der Grenze zum System**	**Lage und Typ des Prozessors**
Embedded	Eingebettet im SoC [Put09]	Schaltkreis	Feste Layoutzelle in einem heterogenen Schaltkreis
Extern	Externe Erweiterung eines Systems zur HW-Beschleunigung	Leiterplatte	Hardcore in einem dedizierten Schaltkreis
Integriert	Integriert den größten Teil des System als konfigurierbares SoC	Schaltkreis	Hard- oder Softcore in einem konfigurierbaren Schaltkreis
Systemlos	Anwendung als Brückenschaltkreis oder Emulation eines digitalen Entwurfes	Leiterplatte	nicht vorhanden

Tabelle 2.2: Systemintegration von konfigurierbaren Strukturen

Ein heterogenes ACS hat mindestens eine rekonfigurierbare Struktur, die sowohl vom Typ *embedded*[24], *extern* oder *integriert* sein kann.

[24] *embedded (engl.)*: eingebettet

Konfigurationsspeicher und Zeitpunkt der (Re-)Konfiguration Die Speicherung der Konfigurationsbits innerhalb einer konfigurierbaren Struktur kann *flüchtigen* oder *nichtflüchtigen* Charakter haben. Flüchtiger Konfigurationsspeicher ist nach stromlosem Zustand oder Zurücksetzen aus einer externen Quelle mit der Konfiguration zu laden. In diese Kategorie fallen Multi-Context Strukturen und alle Strukturen mit statischem RAM[25] (SRAM) Technologie [HM04]. Eine permanente Speicherung der Konfiguration erlauben *Antifuse*[26]- und EPROM-basierte Strukturen. Erstere sind allerdings nur ein einziges Mal konfigurierbar.

Die Rekonfigurationen können zur *Compile-Time*[27] oder zur *Runtime*[28] des Systems festgelegt werden [Bob07]. Bei Ersterem ändert sich die Konfiguration der Struktur während der Berechnung nicht, wohingegen bei Zweiterem die Konfigurationen bedarfsweise dynamisch zum Zeitpunkt der Berechnung umgeladen werden.

Für ein heterogenes ACS kommen nur SRAM-basierte Konfigurationsspeicher in Betracht. Diese werden zur Laufzeit in Abhängigkeit der zu ladenden Anwendung vielfach umgeschrieben. Das Laden der Anwendung erfolgt durch das Betriebssystem aus externem Festspeicher (Flash-Speicher, Festplatten, entfernte Netzwerkressourcen). Im Falle einer systemintegrierenden konfigurierbaren Struktur (vgl. Tabelle 2.2) ist ein externer Permanentspeicher für die Bereitstellung der initialen Konfiguration notwendig.

2.1.4.2 Virtuelle Hardware

Das Konzept virtueller HW [Bre96] basiert auf Laufzeit-Rekonfiguration zur Maximierung der Funktionsdichte [WH02] und ist vergleichbar mit virtuellem Speicher bei Betriebssystemen. Die virtuelle HW stellt dabei die Summe aller möglichen HW-Konfigurationen dar. Die realen physischen Ressourcen sind dabei geringer als die Summe der Ressourcen, die die Konfigurationen benötigen würden. Auf der Basis von Zeit-Multiplex-Betrieb wird nur ein Teil der Konfigurationen geladen und nach Bedarf zur Laufzeit ausgetauscht. Laufzeit-Rekonfiguration erlaubt es damit, eine größere Anwendung auf die HW abzubilden als eine nicht Laufzeit rekonfigurierbare HW fassen

[25] *statischer RAM*: Speicher bei dem der Dateninhalt bei Anliegen der Betriebsspannung beliebig lange gespeichert werden kann

[26] *Antifuse (engl.)*: Ein zwei Ebenen verbindendes Element mit hohem Widerstand, dessen Dielektrikum bei Anlegen einer hohen Spannung schmilzt und damit leitend wird.

[27] *Compile-Time (engl.)*: Zeitpunkt der Übersetzung einer Systembeschreibung oder eines Programms

[28] *Runtime (engl.)*: Zeitraum des Systembetriebes

könnte. Während der Rekonfiguration kann die betreffende HW nicht zur Bearbeitung der eigentlichen Aufgabe genutzt werden. Deshalb gehen mit der Laufzeit-Rekonfiguration Kosten einher, die sich in Zeitaufwand als Rekonfigurationslatenz und zusätzlichen Ressourcen als doppelt ausgelegtem Konfigurationsspeicher niederschlagen.

2.1.5 Ableitung und Modellierung heterogener adaptiver Systemarchitekturen

Aus der in den Kapiteln 2.1.2 und 2.1.4 vorgestellten Rechentechnik lassen sich heterogene Systemmodelle ableiten [RBH09] und anhand der Lage der *Reconfigurable Processing Units*[29] (RPU) unterteilen. Eine Einteilung lässt sich in Multi-Chip- und Single-Chip-Architekturen treffen. Erstere kommen hauptsächlich im Bereich des High-Performance Computing zum Einsatz. Abbildung 2.5a zeigt eine homogene Multi-CPU Architektur, die durch leistungsfähige RPU flankiert wird (*Typ A*). Die adaptive Logik wird für einen breitbandigen Datenaustausch mittels hochfrequenter Ein-/Ausgabe (E/A) Schnittstellen direkt an den Hypertransport-Bus der Prozessoren angeschlossen. Beispielhaft können die Untersuchungen von Heinig, Strunk, Rehm und Schick [Hei+09] zum Betrieb von Virtex-FPGA-Schaltkreisen als RPU an Prozessoren der Firma AMD genannt werden. Durch den Einsatz von dedizierten RPU-Schaltkreisen kann die jeweils implementierte Funktionalität zur Laufzeit ganz oder teilweise variiert werden, ohne das übrige System zu beeinflussen.

Single-Chip Lösungen bieten im Bereich eingebetteter Systeme deutliche Vorteile hinsichtlich Kosten, Leistungsaufnahme, Fehleranfälligkeit und Handhabbarkeit. Als heterogene Single-Chip-Plattformen kommen entweder eingebettete RPU (*Typ B*) oder alleinstehende RPU-Plattformen (*Typ C*) in Frage. Schaltkreise nach *Typ B* integrieren eingebettete RPU als abgeschlossene Inseln programmierbarer Logik neben einem eigenständigen SoC aus fest verdrahteten Prozessor-, Speicher- und Kommunikationsblöcken. Ein entsprechendes Beispiel ist in Abbildung 2.5b dargestellt. Die Funktion der rekonfigurierbaren Insel kann unabhängig vom Betrieb des übrigen Schaltkreises variiert werden. Im Morpheus-Projekt [Put09] wurden RPU verschiedener Granularität integriert.

[29] *Reconfigurable Processing Unit (engl.)*: Prozessierungseinheit auf Basis (re-)konfigurierbarer HW-Strukturen

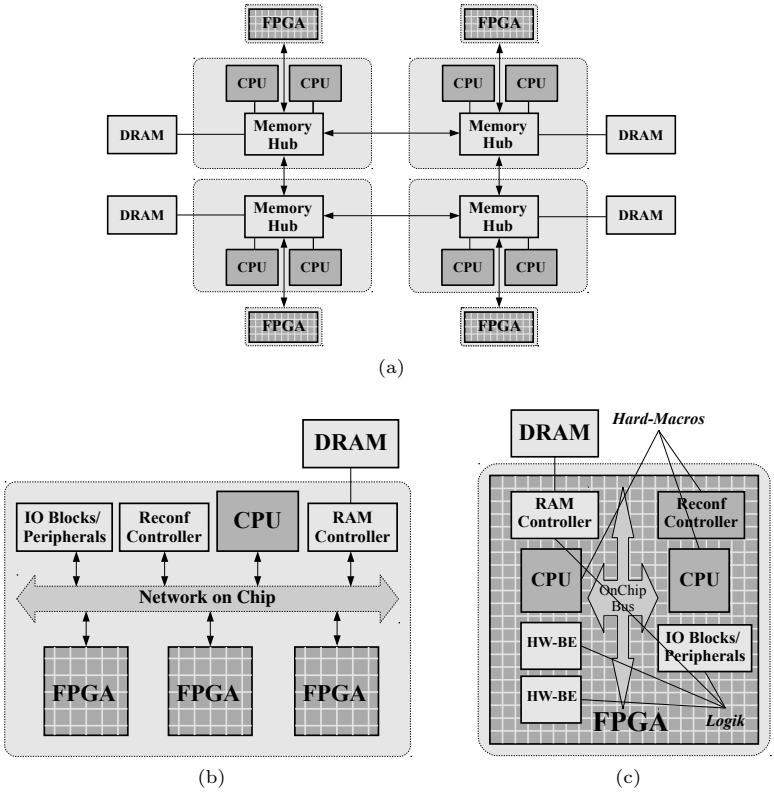


Abbildung 2.5: Grundlegende Typisierung heterogener ACS nach Lage der rekonfigurierbaren Logik als RPU

Schaltkreise nach *Typ C* bestehen hauptsächlich aus programmierbarer Logik wie in Abbildung 2.5c dargestellt. Dedizierte Hardmakro-Bereiche mit statischen Spezialfunktionen (z.B. Block-RAM, CPU oder Takt-Generatoren) werden zusätzlich in die RPU eingebracht. Auf diese Weise wird die Kapazität des Schaltkreises optimiert, weil die Realisierung dieser Funktionalitäten durch die RPU selbst ungleich mehr Siliziumfläche benötigt und darüber hinaus wesentlich performanter entworfen werden kann. Insbesondere die Integration von leistungsfähigen Prozessoren erscheint dabei zielführend. Werden

Infrastrukturteile des Prozessors (Busse, Peripherie und Speichercontroller) allerdings durch die RPU realisiert, ist das Gesamtsystem von der unterbrechungsfreien Verfügbarkeit dieser Logik abhängig. Soll das Konzept virtueller HW (vgl. Kapitel 2.1.4.2) genutzt werden, ergibt sich für die RPU die Notwendigkeit dynamisch partieller Rekonfiguration.

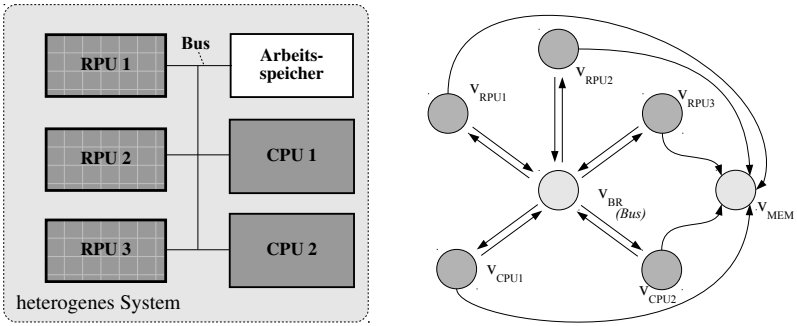
Für die Ausführung einer dynamisch verteilbaren parallelen Anwendung auf den heterogenen Architekturtypen *A*-*C* ergeben sich Vor- und Nachteile. Geht man bei *Typ A* von einem SMP-System aus, besteht zwar eine direkte Anbindung der RPU an Speicher und Prozessoren, diese ist jedoch außerhalb des Schaltkreises und damit beschränkt durch die Verfügbarkeit von speziellen E/A-Ports der RPU sowie Taktgeschwindigkeit und Bitbreite. Befinden sich mehrere RPU innerhalb eines Schaltkreises, müssen diese sich eine Schnittstelle zum Bus teilen. Der Einsatz eines RPU bezogenen Cache erscheint zwingend. Die Integration von RPU mit Mehrzweckprozessoren in einem Schaltkreis (*Typen B* und *C*) umgeht dieses Problem, es können jeweils Pfade zu Speicher und Prozessoren realisiert werden. Die Partitionierung der rekonfigurierbaren Fläche bei *Typ B* ist nicht veränderlich und kann damit nicht an den dynamisch wechselnden Bedarf angepasst werden. Die Typen A und C bieten durch partielle Rekonfiguration mehr Freiraum.

Der Weg und der Aufwand zu einer konkreten Realisierung eines heterogenen ACS unterscheiden sich erheblich zwischen den Typen. *Typ A* ist über handelsübliche Steckkarten realisierbar und erfordert im Maximalfall eine Entwicklung auf Leiterplattenniveau. *Typ B* bedingt hingegen einen umfangreichen *Full-Custom-Entwurf* des Schaltkreises. Für *Typ C* lassen sich die herstellerspezifischen Systementwurfswerkzeuge wie *Embedded Development Kit* (EDK)[Xil13b] von Xilinx oder *Quartus* [Alt13] von Altera für eine weitgehende automatisierte Entwicklung nutzen.

Entsprechend diversifiziert sind die Anwendungsfelder. *Typ A* ist für Großrechner geeignet, die hauptsächlich auf Datenverarbeitung ausgelegt sind, also in keinen expliziten technischen Kontext eingebunden sind. *Typen B* und *C* eignen sich für eingebettete Systeme. Teich [Tei97] definiert ein im technischen Kontext eingebettetes System als: “System, das dazu bestimmt ist, Funktionen als Antwort auf bestimmte Stimuli auszuführen und Daten informationstechnisch zu verarbeiten”. Neben dem Heterogenitätsaspekt zwischen HW und SW treten bei eingebetteten Systemen mechanisch-elektronische und digital-analoge Aspekte auf.

Zur Modellierung einer Architektur wird ein gerichteter Graph, der *Architekturgraph* G_A, nach Definition 2.1.1 genutzt, der eine Architekturspezifikation aus funktionalen Ressourcen in den Ausführungsdomänen HW und

SW, Kommunikationsressourcen als Bus oder Punkt-zu-Punkt Verbindung, sowie Speicherkomponenten erfasst. An den Knoten des Graphen können Kapazitätsschranken (z.B. verfügbare Logikblöcke des FPGA, Gatter in einem ASIC, Speichergröße oder Bandbreite) annotiert sein. Eine einfach strukturierte heterogene Architektur, auf der eine parallelisierte Anwendung domänenübergreifend ausgeführt werden kann, ist in Abbildung 2.6 als Blockbild und korrespondierender Architekturgraph dargestellt. In diesem Beispiel sind zwei Knoten der SW-Domäne über einen gemeinsamen Bus mit drei Knoten der HW-Domäne verbunden, wobei sämtliche Berechnungsknoten Zugriff auf einen gemeinsamen Speicher haben.



(a) Zielarchitektur (b) Architekturgraph G_A

Abbildung 2.6: Einfache heterogene Zielarchitektur und zugehöriger Architekturgraph

Definition 2.1.1. [Architekturgraph] Ein *Architekturgraph* $G_A(V_A, E_A)$ ist ein gerichteter, azyklischer Graph mit der Knotenmenge V_A und der Kantenmenge E_A, in dem jeder Knoten $v_i \in V_A$ eine Komponente der Typen (funktionale Ressource mit Ausführungsdomäne, Busressource, Speicher) und jede Kante $e = (v_i, v_j) \in E_A$ eine gerichtete Kommunikationsverbindung darstellt.

2.1.6 Modell eines heterogenen adaptiven Computersystems

Als Ergebnis der abgeleiteten Systemarchitekturen und Grundlage für alle folgenden Ausführungen, wird das in Abbildung 2.7 gezeigte Modell eines

heterogenen ACS mit virtualisierten Rechenwerken eingeführt, auf das sich alle *Typen A-C* abbilden lassen. Die Sichten auf das System werden in drei horizontalen Schichten dargestellt. Auf der obersten Schicht werden Anwendungen für den Nutzer ausgeführt, die in parallel ablaufende Teilanwendungen untergliedert sind. Die darunter liegende Virtualisierungsschicht stellt Ausführungskontainer (AK) für die Teilanwendungen bereit und bietet eine abstrakte Kommunikations- und Ausführungsschnittstelle, die den Datenaustausch und die Migration der Teilanwendungen zwischen den Kontainern im Sinne der Aufgabenstellung ermöglicht. Auf der untersten Ebene werden die Berechnungen bzw. Operationen in den Rechenwerken real ausgeführt. Diese Ebene gliedert sich in die Ausführungsdomänen HW und SW.

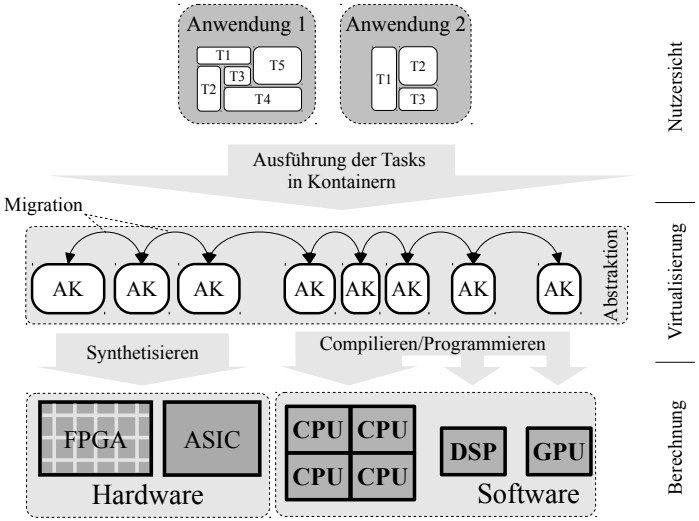


Abbildung 2.7: 3-Schichten Modell eines heterogenen ACS mit virtualisierten Rechenwerken für dynamische HW/SW-Partitionierung

2.2 Entwurf von Hardware- und Software-Systemen

Die Beherrschbarkeit des Entwurfs von komplexen digitalen HW/SW-Systemen und darauf ausführbaren Anwendungen bedingt eine systematische und gegliederte Vorgehensweise. Im Folgenden sollen Entwurfsstruk-

turierungen vorgestellt werden, die sowohl für die HW- als auch für die SW-Komponenten eines Systems anwendbar sind. Anschließend werden für die vorliegende Arbeit relevante Überführungs- bzw. Syntheseschritte näher ausgeführt.

2.2.1 Entwurfssichten, Abstraktionsebenen und Methodik des Systementwurfs

Zur Systematisierung werden Entwurfssichten und Abstraktionsebenen eingeführt. In [Har02a] wird eine hilfreiche Analogiezwischen den Entwurfssichten beim Systementwurf und dem Entwurf eines Gebäudes aufgestellt: Ein Bauingenieur entwickelt Pläne für Statik, Elektroinstallation, Wärme und Wasserversorgung, Feuermeldetechnik etc., die in Summe das Gebäude abbilden. Abstraktionsebenen reduzieren die Komplexität, indem Details ausgeblendet werden. Der Entwurfsprozess erfolgt dann schrittweise durch Übergänge zwischen den Ebenen unter Berücksichtigung sämtlicher Sichtweisen. Erfolgt der Übergang innerhalb einer Ebene, wird die Komplexität der Systemkomponenten reduziert. Ein Übergang zwischen den Ebenen verringert den Grad der Abstraktion. Jeder Entwurfsschritt konkretisiert und optimiert dabei den erreichten Entwurfszustand mit dem Endziel einer vollständigen Implementierung.

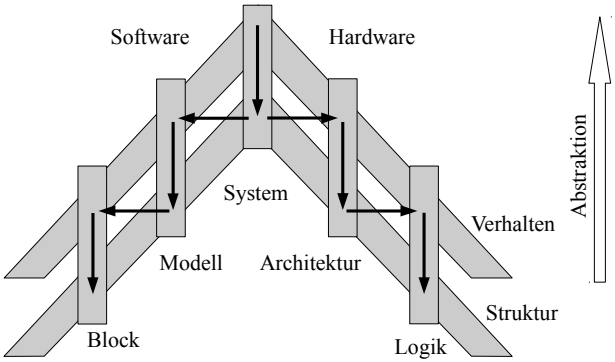


Abbildung 2.8: Abstraktionsebenen und Sichten beim Systementwurf nach Teich [Tei97]

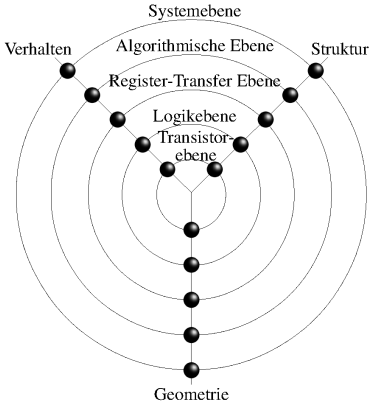
Eine allgemeine Strukturierung mit Abstraktionen für den Entwurf eines HW/SW-Systems nach Teich [Tei97] ist in Abbildung 2.8 dargestellt. Ausgehend von der *Systemebene*, in der das Gesamtsystem in Teilsysteme zerlegt wird und globale Eigenschaften definiert werden, gliedert sich der HW- und der SW-Bereich in jeweils zwei weitere Unterebenen. Im Bereich der HW beschreibt die *Architekturebene* das System als kommunizierende funktionale Blöcke, die arithmetische und logische Operationen ausführen. Die darunter liegende *Logikebene* bildet das Systemverhalten auf Basis von booleschen Gleichungen und die Systemstruktur durch Grundgatterelemente ab. Im SW-Bereich abstrahiert die *Modulebene* Funktion und Interaktion komplexer Codefragmente, wie zum Beispiel miteinander kommunizierende Prozesse oder Threads. Die folgende *Blockebene* modelliert SW-Programme auf Befehlsniveau, also durch elementare Operationen auf einer Rechnerarchitektur.

Nach Teich werden die Sichten *Verhalten* und *Struktur* unterschieden. In Ersterer werden Funktionen unabhängig von deren Implementierung beschrieben. Die strukturelle Sichtweise bezieht sich auf kommunizierende Komponenten, wobei deren Aufteilung und Kommunikation der letztendlichen Implementierung entspricht. Im Entwurfsprozess erfolgt die Überführung der Verhaltenssicht in eine Struktur innerhalb der Abstraktionsebene. Diese dient dann als Ausgangspunkt für die Verhaltensbeschreibung der nächst tiefer liegenden Ebene. Erfolgt dieser Übergang automatisiert unter Einbeziehung von ebenenspezifischen Optimierungskriterien, wird von *Synthese* gesprochen. Die *Systemsynthese* hat beispielsweise das Ziel aus einer Systembeschreibung eine Architektur zu erzeugen. Die Beweggründe für automatisierte Syntheseverfahren sind verkürzte Entwurfszyklen, Reduzierung von Entwurfsfehlern und eine optimale Exploration des Entwurfsraumes [Tei97].

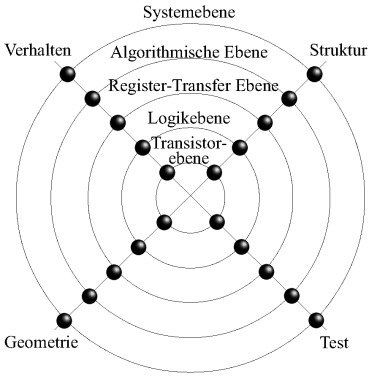
Die Trennung des Entwurfes in HW und SW erfolgt, nachdem die Struktur auf Systemebene festgelegt wurde. Dieser Entwurfsschritt wird als HW/SW-Partitionierung bezeichnet und kann statisch oder dynamisch erfolgen.

Für den Entwurf im HW-Bereich werden durch das beschriebene Modell wesentliche Entwurfsmerkmale ausgeblendet. Gajski und Kuhn [GK83] sowie Walker und Thomas [WT85] unterteilen den Entwurf im *Y-Diagramm* (vgl. Abbildung 2.9a) vielmehr in fünf Abstraktionsebenen. Anstatt der Architekturebene wird weiter in *algorithmische Ebene* und *Register-Transfer-Ebene* (RT-Ebene) unterteilt. Erstere dient der Auswahl der Verarbeitungsalgorithmen und damit auch der Daten- und Busbreiten sowie der Befehlssätze. Auf der RT-Ebene wird das System auf reale HW-Strukturen aus Verarbeitungseinheiten und Registern/Speichern abgebildet. Unterhalb der *Logikebene* wird noch eine Ebene für die Dimensionierung und Verschaltung von

Transistoren eingeführt. Außerdem lassen sich im HW-Bereich die zusätzlichen ebenenübergreifenden Sichten *Geometrie* und *Test* charakterisieren. Rammig [Ram89] erweiterte durch die Sicht des Tests das Y-Diagramm zum in Abbildung 2.9b gezeigten *X-Diagramm.*



(a) Y-Diagramm nach Gajski Gajski und Kuhn [GK83] und Walker Walker und Thomas [WT85]



(b) X-Diagramm nach Rammig Rammig [Ram89]

Abbildung 2.9: Entwurfsstrukturierung für digitale Systeme

Die HLS (vgl. Kapitel 2.2.4) als Übergang zwischen algorithmischer und RT-Ebene sowie die RT-Synthese als Brücke zwischen RT-Ebene und Logik stellen für die vorliegende Arbeit besonders relevante Synthesemechanismen dar.

Hardt [Har02a] leitet schließlich unter der Einbeziehung des SW-Entwurfs eine allgemeine Strukturierung ab, deren Ziel der Entwurf komplexer heterogener eingebetteter Systeme ist. Dazu wird der Begriff der Entwurfsdimension als klar abgrenzbares Teilziel beim Entwurf definiert. Das Teilziel stellt dabei ein vollständig entworfenes Teilsystem oder die vollständige Behandlung einer Anforderung aus der Entwurfsaufgabe dar. Beispiele für Entwurfsdimensionen sind Spezifikation, Modellierung, Verifikation, SW-Synthese, HW-Synthese, Betriebssystem oder Rekonfigurationssteuerung. Die mehrdimensionale Strukturierung nach Hardt lässt sich im *P-Diagramm,* wie in Abbildung 2.10 gezeigt, darstellen.

Das methodische Vorgehen beim Entwurf eines Systems lässt sich in drei Verfahren einteilen [HM04]. Der *Top-Down-Entwurf* schreitet vom Abstrak-

ten hin zum Detaillierten, durchläuft die Diagramme also von oben nach unten. Ausgehend von der Spezifikation wird die abschließende Implementierung des Systems durch stete Verfeinerung verfolgt. Dem gegenüber steht der *Bottom-Up-Entwurf,* der ausgehend von den technologischen Gegebenheiten die Lösung des Systemproblems sucht. Dieses Vorgehen wird beispielsweise angewandt, wenn der Entwickler die Frage stellt bzw. beantwortet: Mit welche Element einer verfügbaren Makrobibliothek kann die geforderte Funktion spezifikationsgerecht umgesetzt werden?

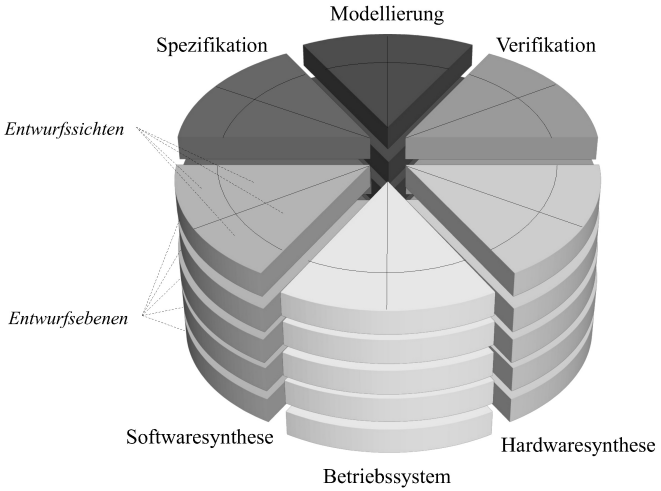


Abbildung 2.10: Entwurfsstrukturierung im P-Diagramm nach Hardt Hardt [Har02a]

Praktisch wird sich keiner der beiden Ansätze in Reinform einsetzen lassen. Gewöhnlich findet der *Meet-in-the-Middle* Ansatz Anwendung, bei dem die Entwicklung unter gleichzeitiger Betrachtung von Aspekten der oberen und unteren Abstraktionsebenen auf iterative Weise erfolgt [Mul01].

2.2.2 Entwurfsprozess anwendungsspezifischer Hardware

Für Rechenwerke in der HW-Domäne bilden rekonfigurierbare Strukturen als RPU die Basis. Im Speziellen sind FPGA für das Ziel dieser Arbeit ausschlaggebend (vgl. Kapitel 2.1.4). In Abbildung 2.11 sind die wichtigsten

Schritte im Entwicklungsprozess einer FPGA-Anwendung dargestellt. Weil die HW-Domäne als Teil des Gesamtsystems betrachtet wird, beziehen sich die folgenden Ausführungen auf eine alleinstehende Anwendung auf einem systemlosen FPGA (vgl. Tabelle 2.2), die als synchrone digitale Schaltung implementiert werden soll.

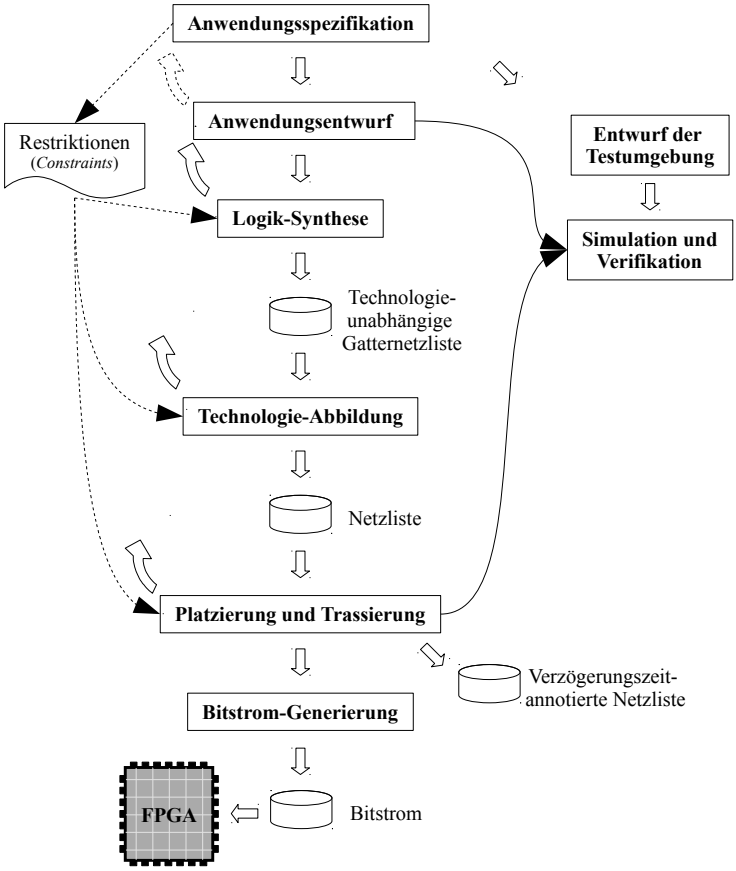


Abbildung 2.11: Anwendungsentwicklung für einen FPGA

Startpunkt ist die Spezifikation zur formalen oder nicht formalen Erfassung einer funktionalen Beschreibung, zeitlichem Verhalten (auf der Basis von Kausalität der Ereignisse) und Anforderungen/Restriktionen auf Systemebene [HM04]. Die Spezifikation sollte unbedingt vollständig und konsistent sein. Das ist praktisch erst nach mehreren Iterationen der Fall und es muss auch mit nicht eindeutigen Beschreibungen und Unvollständigkeit gerechnet werden. Anschließend wird aus der *komplexen* Verhaltensbeschreibung die Systemstruktur auf Basis von Komponenten mit *weniger komplexem* Verhalten abgeleitet, die Anwendung wird partitioniert.

Die nachfolgende Abbildung der partitionierten Anwendung auf einen FPGA ist eine semimanuelle Aufgabe und folgt dem strukturierten Entwurfsprozess nach Kapitel 2.2.1. Insbesondere der Schritt zur algorithmischen Repräsentation ist gegenwärtig weitgehend ohne Rechnerunterstützung zu bewältigen. Ausgangspunkt für den Einsatz der automatisierten Werkzeugkette ist eine formalisierte Beschreibung der Anwendung in algorithmischer oder RT-Ebene. Zur funktionalen Ausformulierung der Algorithmen stehen dem Entwicklungsingenieur Hochsprachen, wie zum Beispiel Matlab, C++, C oder Java, zur Verfügung. Mittels HLS (vgl. Kapitel 2.2.4) erfolgt die automatisierte Überführung in die RT-Ebene. Alternativ erlauben standardisierte HW-Beschreibungssprachen[30], wie zum Beispiel VHDL [Hei+00] oder Verilog [Hop06], dem Ingenieur die explizite Formulierung von Nebenläufigkeit in einer frei kombinierbaren Beschreibung von Struktur und Verhalten auf den unterschiedlichen Abstraktionsebenen. Die resultierende HW-Beschreibung ermöglicht die taktgenaue Modellierung, Simulation und Analyse von HW-Systemen [AR03]. Auch der grafisch basierte Entwurf mittels Editor zur Verschaltung von Basiselementen in einem Schaltplan soll zur Vollständigkeit erwähnt werden.

Parallel zur Codeumsetzung einer Anwendung wird die funktionale Korrektheit des Entwurfes sichergestellt und auftretende Fehler werden beseitigt. Traditionell wird der Entwurf dazu in einer Simulationsumgebung ausgeführt, durch Testvektorfolgen am Eingang stimuliert und die resultierende Folge der Ausgangsvektoren verifiziert[31]. Der zu testende Entwurf, an dieser Stelle *Model under Test*[32] genannt, wird dazu in eine *Testbench*[33] als Nachbildung der späteren Anwendungsumgebung eingebunden. Die Testumgebung wird in HDL oder artverwandten Systemsprachen (SystemC, SystemVHDL oder Sys-

[30] *Hardwarebeschreibungssprache*: engl. Hardware Description Language (HDL)
[31] *Verifizierung*: Die Prüfung oder ggf. der Beweis, dass das gewünschte Verhalten ohne Fehler realisiert wurde.
[32] *Model under Test (engl.)*: bezeichnet das zu testende Modell.
[33] *Testbench (engl.)*: Prüfstand

temVerilog) realisiert, die in einem geeigneten Mehrsprachensimulator ausgeführt werden. Im Zusammenhang mit dem Entwurfstest werden auch *Assertion*[34]-basierte und formale Verifikationsmethoden [Ren+06; Tis+08] eingesetzt, um die Einhaltung spezifizierter Eigenschaften sicherzustellen bzw. zu beweisen.

Eine synthetisierbare Beschreibung auf RT-Ebene ist Ausgangspunkt für die Erzeugung der Netzliste mittels *Logiksynthese*. Zunächst wird der Entwurf in einen Satz von Booleschen Gleichungen überführt und anschließend auf die FPGA-Technologie abgebildet. Die Abbildung erfolgt dabei auf die Logikzellen, im Regelfall auf *Lookup-Table*[35] und Register. Die *Netzliste* speichert das Ergebnis der Logiksynthese als Liste der genutzten Logikzellen, Speicherzellen und notwendigen Verbindungen zwischen diesen. Im folgenden Platzierungs- und Trassierungsschritt[36] werden zum einen die in der Netzliste aufgeführten Ressourcen auf den FPGA platziert, d.h. konkrete Instanzen ausgewählt. Zum anderen werden die Verbindungen über das Netzwerk des FPGA geleitet sowie Takttreiber und E/A-Ports angeschlossen. Nach diesem Schritt stehen die realen Verzögerungszeiten auf allen Pfaden der digitalen Schaltung fest und können gegen die spezifizierten Sollwerte geprüft werden. Auf dieser Ebene findet die Verifikation durch die Simulation der mit Verzögerungszeiten annotierten Netzliste[37] in der bestehenden Testumgebung statt. Werden die Sollwerte nicht erreicht, können die Synthese und Abbildungsschritte mit veränderten Optimierungsparametern erneut ausgeführt werden oder im ungünstigeren Fall muss weiter im Entwurfsablauf zurückgegangen werden.

Schließlich werden für die im FPGA genutzte Fläche die Werte der Konfigurationsbits festgelegt, die in Form eines Bitstromes zunächst gespeichert und zur Laufzeit in den Schaltkreis geladen werden. Der Test der Anwendung erfolgt nun in HW, sodass die Funktionalität der Anwendung schließlich validiert[38] werden kann.

[34] *Assertion (engl.)*: Behauptung
[35] *Lookup-Table (engl.)*: Speicher, in dem die Werte einer Funktion als Tabelle abgelegt werden. (LUT)
[36] *Platzierung und Trassierung*: engl. Place and Route
[37] *Simulation einer Verzögerungszeiten-annotierten Netzliste*: engl. backannotation
[38] *Validierung*: Überprüfung, eines gewünschten Verhaltes im Schaltkreis bzw. Endsystem.

2.2.3 Dynamisch partiell rekonfigurierbare Schaltungen

Der Entwurfsablauf einer Anwendung mit einem partiell rekonfigurierbaren FPGA unterscheidet sich wesentlich von der mit vollständiger Rekonfiguration. Zu Beginn der Codeumsetzung ist die Anwendung in statische und dynamische Teile zu zerlegen, die als Module bezeichnet werden. Das statische Modul enthält die unveränderlichen Komponenten wie z.B. die Rahmenbeschaltung, gewollt unveränderliche Schaltungsteile, Takt-Treiber, Bus-Strukturen, interne Rekonfigurationsschnittstelle oder E/A-Schnittstellen. Die dynamischen Module enthalten den variablen Systemteil, z.B. unterschiedliche Filter einer Audiostufe, die exklusiv Höhen, Tiefen oder Bänder filtern kann. Zum anderen muss ein Rekonfigurationsmanager entworfen oder generiert werden [Mei10], der die Laufzeitrekonfiguration ausführt.

Allgemein gliedert sich der Entwurf eines partiell konfigurierbaren Systems in vier Phasen:

1. Partitionierung und Modularisierung des Gesamtdesigns,
2. *Initial-Budgeting and Floorplanning (engl.)*: Planung der FPGA-Fläche und Festlegung der Lage von Modulen und deren äußerer Schnittstellen,
3. *Active-Implementation (engl.)*: RTL-Entwurf, Synthese und Platzierung/Trassierung zunächst des Top-Levels und des statischen Moduls und anschließend der dynamischen Module,
4. *Bitstream-Generation (engl.)*: Generierung der initialen und der partiellen Bitströme.

Bis zum gegenwärtigen Zeitpunkt beschränkt sich die Auswahl an Schaltkreisen, die technologisch partielle Rekonfiguration ermöglichen, auf Produkte der Firma Xilinx Xilinx Inc. [Xil13a], im Speziellen auf die Klasse der Virtex-FPGA. Die methodische Handhabung dynamisch partieller Rekonfiguration war seit über einem Jahrzehnt und damit auch über den Zeitraum der vorliegenden Arbeit Gegenstand der Forschung in wissenschaftlicher und Xilinx-interner Forschung. Unter anderem entstanden JBits [GLS99], PARBIT [HL01], der *Modular Design Flow* [Xil04], der *Early Access Design Flow* [Xil06] und der ReCoBus-Builder [KBT08]. Ein vorläufiger Abschluss der Entwicklung wurde mit der produktmäßigen Unterstützung partieller Rekonfiguration durch Xilinx in der EDK ab Version 12.1 gefunden. Für eine detaillierte Betrachtung der zuvor genannten Verfahren wird daher auf [RBH09] verwiesen.

2.2.4 High-Level-Synthese - Synthese algorithmischer Verhaltensbeschreibungen

Der Begriff HLS, auch Architektur-Synthese, Algorithmensynthese oder Verhaltenssynthese genannt, stellt einen möglichen Schritt im Entwurfsablauf von HW-Systemen dar, bei dem eine funktionale Spezifikation aus der algorithmischen Ebene in die strukturelle Beschreibung der RT-Ebene überführt wird (vgl. Abbildung 2.12). Es handelt sich um einen Verfeinerungsschritt, bei dem eine verhaltensbasierte Beschreibung von Aufgaben mit einer Granularität von logischen/arithmetischen Operationen (z.B. Schiebe-, Rotationsfunktionen, Addition und Multiplikation) auf eine strukturelle Beschreibung von Daten- und Kontrollpfad abgebildet wird [Tei97; De 94b; Gaj+92]. Im Datenpfad (Operationswerk) werden *funktionale Ressourcen* (Addierer, ALU, Multiplizierer, Komparatoren usw.), *Kommunikationsressourcen* (Multiplexer, Busse und Leitungsverbindungen) und *Speicherressourcen* (Register, Ringpuffer[39] und Speicherblöcke) auf der Ebene von RT-Blöcken verbunden. Der Kontrollpfad (Steuerwerk) koordiniert diese Blöcke und wird auf Logik-Ebene in Form eines Automaten realisiert.

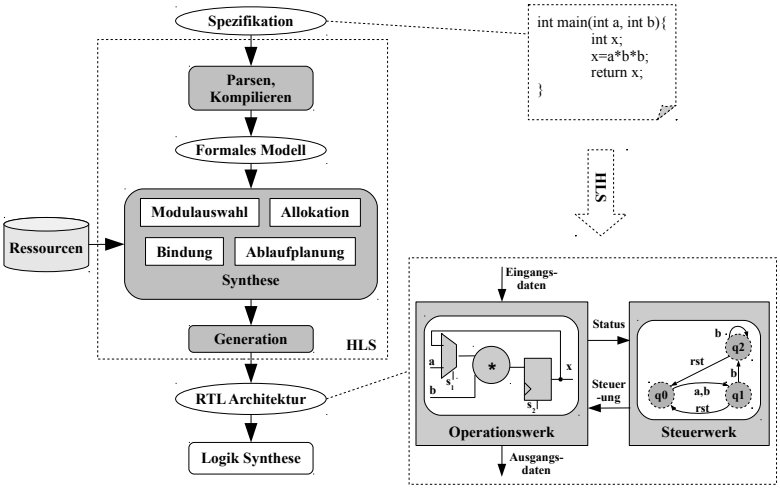


Abbildung 2.12: Entwurfsschritte bei der HLS

[39] Ringpuffer: First In First Out-Buffer (FIFO)

Das Wort *Synthese* impliziert hierbei eine automatische, d.h. rechnergestützte, Überführung zwischen den Entwurfsebenen. Alternativ besteht die Möglichkeit, den Transfer manuell umzusetzen. Profunde Kenntnisse des Algorithmus, der folgenden Logiksynthese und der Zieltechnologie vorausgesetzt können Entwürfe realisiert werden, die einer Synthese in puncto Performanz und Flächenbedarf deutlich überlegen sind. Händisch entwickelter RT-Code benötigt jedoch eine deutlich längere Entwicklungszeit [Röß+09] und ist in Bezug auf nachträgliche Anpassungen des Algorithmus und im Sinne einer Exploration des Entwurfsraumes weniger flexibel [Röß+11].

HLS bietet die folgenden Vorteile:

- Syntaktisch und Zeitlich fehlerfreier RT-Code durch generierungsbedingte Korrektheit[40],
- zügige Exploration des Entwurfsraumes auf hoher Abstraktionsebene,
- zügige Umsetzung komplexer Algorithmen durch Reduktion des Detailgrades,
- effektiver Einsatz von verlustleistungsreduzierenden Maßnahmen, wie z.B. zeitweise Abschalten von ungenutzten Schaltungsteilen [HVH99].

In den folgenden Abschnitten werden die grundlegenden Modelle und Definitionen der HLS nach Gajski, Dutt, Wu und Lin [Gaj+92], De Micheli [De 94b] und Teich [Tei97] eingeführt. Diese dienen als Grundlage für die formalisierte Analyse und Problemdarstellung im Zusammenhang mit dem Entwurf einer verteilten Anwendung auf einem heterogenen ACS im Sinne der Zielstellung der vorliegenden Arbeit (vgl. Kapitel 1.2).

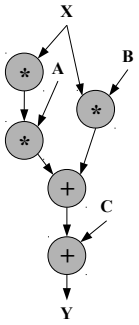
2.2.4.1 Modellierung von Verhaltensbeschreibungen

Als Modell zur Erfassung der Verhaltensbeschreibung dient zunächst ein gerichteter, azyklischer Graph[41]. Der sogenannte Problemgraph bildet die Reihenfolge der auszuführenden Operationen anhand von Datenabhängigkeiten ab. Ein Beispiel eines Problemgraphen zeigt Abbildung 2.13.

Definition 2.2.1. [Problemgraph] Ein *Problemgraph* $G(V, E)$ ist ein gerichteter, azyklischer Graph mit der Knotenmenge V und der Kantenmenge E, in dem jeder Knoten $v_i \in V$ eine Aufgabe (Task, Prozess, Anweisung, Elementaroperation) und jede Kante $e = (v_i, v_j) \in E$ eine Datenabhängigkeit darstellt.

[40] *konstruktions-/generierungsbedingte Korrektheit*: engl. correctness by construction
[41] *gerichteter azyklischer Graph*: engl. directed acyclic graph (DAG)

Ein Problemgraph nach Definition 2.2.1 erfasst als Datenflussgraph zunächst nur Verhalten ohne Kontrollstrukturen. Für die Modellierung von Verzweigungen und Iterationen (z.B. Schleifen) bedarf es der Erweiterung zu einem heterogenen Modell, dem *Kontroll-Datenflussgraphen*[42] (CDFG). In [De 94b] wird die Klasse der Sequenzgraphen definiert, bei der die Darstellung mittels hierarchisch verknüpfter azyklischer Graphenmodelle realisiert wird. Abbildung 2.14 zeigt die Modellierung mit Sequenzgraphen beispielhaft.



Beispiel 2.2.2. quadratische Gleichung:

```
y = A*x*x + B*x + C;
```

Abbildung 2.13: Datenflusse im Problemgraph von Beispiel 2.2.2

Definition 2.2.3. [Sequenzgraph] Ein *Sequenzgraph* bezeichnet eine Hierarchie von gerichteten Graphen. Ein generisches Element der Graphen heißt Einheit eines Sequenzgraphen. Eine Einheit ist ein erweiterter Problemgraph $G(V, E)$ mit Knotenmenge V und der Kantenmenge E sowie folgenden Eigenschaften:

- Eine Einheit besitzt zwei Arten von Knoten a) Aufgaben oder Operationen und b) Hierarchieknoten. Hierarchieknoten dienen der Verbindung von Einheiten der Hierarchie.
- Eine Einheit stellt einen azyklischen und polaren Graphen dar, d.h. es gibt zwei ausgezeichnete Knoten, den sog. Startknoten und den Endknoten. Beide Knoten sind Hierarchieknoten und stellen die Opertation NOP (= keine Operation) dar. Neben Start- und Endknoten gibt es drei weitere Hierarchieknoten, nämlich Modulaufruf (CALL), Verzweigung (BR) und Iteration (LOOP).

[42] *Kontroll-Datenflussgraph*: engl. Control Data Flow Graph

- Die Blätter der Hierarchie werden von Sequenzgraphen gebildet, die außer dem Start- und Endeknoten keine weiteren Hierachieknoten besitzen.

Beispiel 2.2.4. Berechnung mit verzweigendem Kontrollfluss:

```
if (cond)
    y = A*x*x + B*x + C;
else
    y = A*x + B*C;
```

Beispiel 2.2.5. Berechnung mit iterativem Kontrollfluss:

```
while (cond)
   y = A*x*x + B*x + C;
```

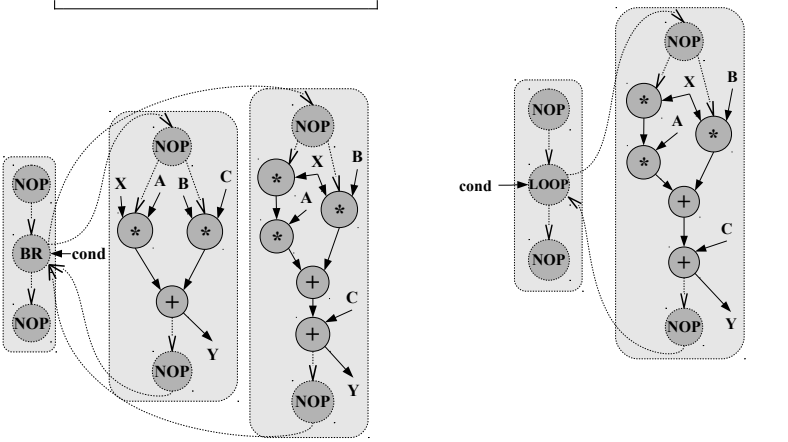


Abbildung 2.14: Modellierung von Kontroll- und Datenfluss mittels Sequenzgraphen

Mit der Größe der zu modellierenden Anwendung steigt der Umfang dieser umfassenden Graphen-Modelle extrem und verlangsamt Analysen und Transformationen. Zur Reduktion der Komplexität können CDFG wieder aufgebrochen, d.h. einzelne Details gekapselt werden. Bereiche bzw. Teilgraphen, die nur Hierarchieknoten vom Typ Start- und Endknoten besitzen, bilden einen *Basis-Block* ohne Kontrollfluss und damit einen eigenständigen Problemgraphen im Sinne der Definition 2.2.1 mit einheitlichem Eintritts- und Austrittspunkt des Datenflusses. Im CDFG reduzieren sich die Basis-Blöcke auf einen Knoten, dessen Kanten die "externen" Datenabhängigkeiten bilden. In Abbildung 2.15 ist für das Beispiel 2.2.6 ein CDFG dargestellt, der

das Sequenzgraphen-Modell um den Hierarchieknotentyp (JOIN) als explizite Zusammenführung zuvor aufgespaltenen Kontrollflusses erweitert. Für die getrennte Betrachtung von Kontroll- und Datenabläufe werden CDFG in separate Kontrollflussgraphen[43] und Datenflussgraphen[44] aufgespalten.

Beispiel 2.2.6. Berechnung mit komplexem Kontrollfluss:

```
1   i = 0;
2   j = 0;
3   while (i < MAX){
4     x = readmem();
5     if (cond){
6       y = A*x*x + B*x + C;
7       j++;
8     }
9     else
10      y = A*x + B*C;
11    writemem(y);
12    i++;
13  }
14  r = j;
```

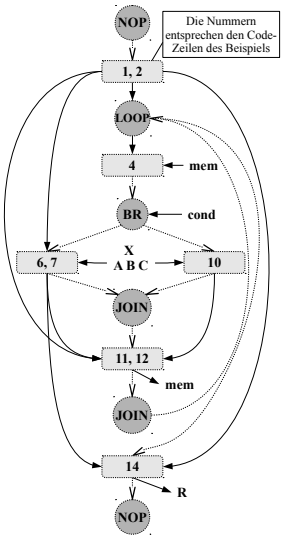


Abbildung 2.15: Vereinfachter Kontroll-Datenflussgraph mit Kapselung des unverzweigten Kontrollflusses in Basis-Blöcke

Neben der Erfassung des Verhaltens benötigt die HLS formalisierte Informationen über die Zielarchitektur, auf der die Eingangsbeschreibung ausgeführt werden soll. Die zur Verfügung stehenden Ressourcentypen werden mit den dazugehörigen Berechnungszeiten und Kosten in einem Ressourcengraph erfasst. Kanten im Ressourcengraph erfassen den Zusammenhang zwischen Aufgaben bzw. Operationen des Problemgraphen und Ressourcen.

Definition 2.2.7 (Ressourcengraph)**.** Ein *Ressourcengraph* $G_R(V_R, E_R)$ ist ein biparitер Graph. Die Knotenmenge $V_R = V \cup V_T$ enthält die Knotenmenge V des Problemgraphen $G(V, E)$. Jeder Knoten $r_k \in V_T$ stellt einen Ressourcentyp (z.B. Addierer, Multiplizierer etc.) dar. Eine Kante $(v_i, r_k) \in E_R$ mit

[43] *Kontrollflussgraph*: Control Flow Graph (engl.) (CFG)
[44] *Datenflussgraph*: Data Flow Graph (engl.) (DFG)

$v_i \in V$ und $r_k \in V_T$ modelliert die Realisierbarkeit von v_i auf einer Instanz des Ressourcentyps r_k. Ferner gibt es eine Kostenfunktion $c : V_T \rightarrow Z_O^+$, die jedem Ressourcentyp $r_k \epsilon V_T$ die Kosten $c(r_k)$ zuordnet, sowie eine Gewichtsfunktion $w : E_R \rightarrow Z_O^+$, die jeder Kante $(v_i, r_k) \in E_R$ die Berechnungszeit $w(v_i, r_k)$ zuordnet, die die Berechnungszeit von v_i auf r_k darstellt.

2.2.4.2 Formalisierung des Syntheseproblems

Das HLS-Problem gliedert sich in drei Grundaufgaben:

1. Allokation der Ressourcen: In welcher Anzahl stehen die einzelnen Ressourcen des Ressourcengraphen zur Verfügung?
2. Ablaufplanung: Zu welchem Zeitschritt wird eine Operation ausgeführt?
3. Bindung der Ressourcen: Abbildung von Operationen, Variablen und Datentransfer auf spezifische Ressourcen.

Eine zusätzliche "Modulauswahl" ist notwendig, wenn für eine bestimmte Operation mehrere Typen von Ressourcen verfügbar sind, also Operationsknoten mehrere ausgehende Kanten im Ressourcengraphen aufweisen. Beispielsweise kann eine Addition jeweils auf einem Addierer, einer ALU oder einer Addier-Multipliziereinheit ausgeführt werden, die unterschiedliche Eigenschaften in Bezug auf Fläche, Performanz und Leistungsaufnahme besitzen.

Verallgemeinert handelt es sich bei der HLS nach der Allokation α um ein mehrdimensionales Optimierungsproblem zur Festlegung eines Ablaufplanes ϕ und einer Ressourcenbindung β entsprechend der folgenden formalen Definitionen.

Definition 2.2.8 (Allokation)**.** Gegeben sei eine Spezifikation $(G(V, E), G_R(V_R, E_R))$. Eine Allokation ist eine Funktion $\alpha : V_T \rightarrow Z_O^+$, die jedem Ressourcentyp $r_k \in V_T$ die Anzahl $\alpha(r_k)$ verfügbarer Instanzen zuordnet.

Definition 2.2.9 (Ablaufplan)**.** Ein *Ablaufplan* eines Problemgraphen $G(V, E)$ ist eine Funktion $\phi : V \rightarrow Z_O^+$, die jedem Knoten $v_i \in V$ die Startzeit $t_i = \phi(v_i)$ zuordnet, sodass gilt:

$$\phi(v_i) \geq \phi(v_j) + d_j \quad \forall (v_i, v_j) \in E \ .$$

Definition 2.2.10 (Bindung)**.** Gegeben sei eine Spezifikation $(G(V,E), G_R(V_R,E_R))$. Die *Bindung ist ein Tupel von Funktionen*

- $\beta : V \rightarrow V_T$ mit $\beta(v_i) = r_k \in V_T$ und $(v_i, \beta(v_i)) \in E_R$;
- $\gamma : V \rightarrow Z^+$ mit $\gamma(v_i) \leq \alpha(\beta(v_i))$.

In Definition 2.2.10 gibt $r_k = \beta(v_i)$ den *Ressourcentyp* und $\gamma(v_i)$ die *Instanz des Ressourcentyps* r_k an, auf der v_i ausgeführt wird. Dabei muss $\gamma(v_i)$ kleiner gleich der Anzahl allokierter Instanzen des Ressourcentyps $\beta(v_i)$ sein, bzw. die Allokation des Typs $\beta(v_i)$ muss für alle Knoten $v_i \in V$ größer gleich $\gamma(v_i)$ sein.

Die Lösungsmöglichkeiten des Syntheseproblems ergeben sich entlang der Dimensionen: *Ausführungszeit* und *Ressourcenbedarf*, wobei sich eine (gültige) Implementierung durch die Einhaltung gegebener Restriktionen/Randbedingungen[45] in Bezug auf diese Dimensionen ergibt. Die Ausführungszeit lässt sich auch als Verzögerung bzw. *Latenz* zwischen dem Start der Berechnung des formulierten Problems bis zu deren Ende auffassen.

Definition 2.2.11 (Latenz)**.** Die Latenz L eines Ablaufplans ϕ eines Problemgraphen $G(V,E)$ ist definiert als

$$L = \underset{v_i \in V}{max} \{\phi(v_i) + d_i\} - \underset{v_i \in V}{min} \{\phi(v_i)\}$$

und bezeichnet damit die Anzahl der Zeitschritte des kleinsten Intervalls, das die Ausführungsintervalle aller Knoten $v_i \in V$ einschließt.

Definition 2.2.12. [Implementierung] Gegeben sei eine Spezifikation $(G(V,E), G_R(V_R,E_R))$ sowie evtl. eine Latenzschranke $\overline{L}$ und/oder eine Allokation α. Eine *gültige Implementierung* ist ein Quadrupel $(\phi, \beta, \gamma, \alpha)$, bestehend aus Ablaufplan ϕ nach Definition 2.2.9, Bindung β, γ nach Definition 2.2.10 und Allokation α nach Definition 2.2.8, das die folgenden Bedingungen erfüllt:

- $L \leq \overline{L}$ im Fall einer gegebenen Latenzbeschränkung und
- $| \{v_i : \beta(v_i) = r_k \wedge \phi(v_i) \leq t \leq \phi(v_i) + d_i\} | \leq \alpha(r_k)$
  $\forall r_k \in V_T, \forall t \in \left[\underset{v_i \in V}{min} \{\phi(v_i)\}, ..., \underset{v_i \in V}{max} \{\phi(v_i) + d_i\} \right]$ im Fall von Ressourcenbeschränkungen.

[45] *Restriktion/Randbedingung*: engl. Constraint

2.2.4.3 Lösungen der High-Level-Synthese

Die Synthesealgorithmen werden entsprechend der primären Optimierungsrichtung in die Klassen hierarchisch, zeitbeschränkt oder ressourcenbeschränkt oder unbeschränkt eingeteilt. Für die Ablaufplanung von datenflussorientierten Problemen mit flachem Kontrollfluss erzeugen die Verfahren *as soon as possible* (ASAP) [TS86], *as late as possible* (ALAP) und deren Modifizierungen gültige Implementierungen mit polinomialem Aufwand. Heuristisch lassen sich Ablaufpläne durch *list scheduling* [Gir84] und *forced directed scheduling* [PK89] bestimmen. Mit Hilfe eines *ganzzahlig-linearen Programms*[46] formulierten Hafer und Parker [HP83] eine formale Methode, um algebraische Funktionen auf die RT-Ebene zu synthetisieren und dabei sämtliche Grundaufgaben der HLS in einem Schritt zu betrachten.

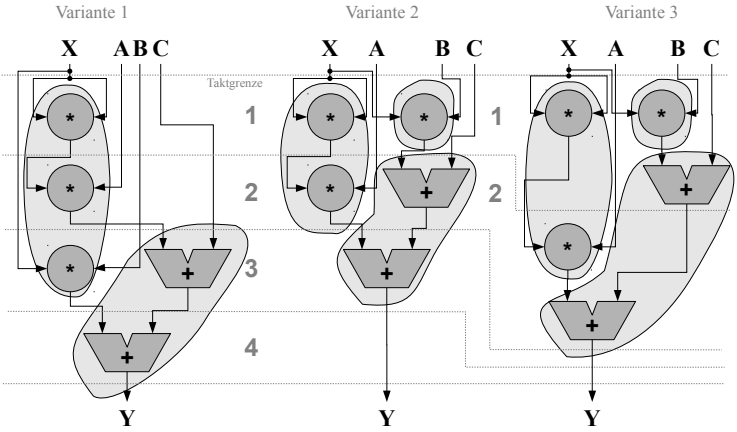


Abbildung 2.16: Optimierungsproblem bei der Synthese von Verhalten auf die RT-Ebene für das Beispiel 2.2.2

Für ein einfaches Problem mit linearem Kontrollfluss zeigt Abbildung 2.17 den sich ergebenden Entwurfsraum, der durch die Kriterien Taktperiode τ, Latenz L als Ausführungszeit $T_{EX} = L \cdot \tau$ sowie die Kosten der Ressourcen R (je nach Zieltechnologie Schaltkreisfläche, Gatter oder Anzahl der Logikblöcke eines FPGA) bestimmt ist. In Abbildung 2.16 sind Ablaufplanung und

[46] *ganzzahliges lineares Programm*: engl. Integer Linear Program (ILP)

Allokation von drei Synthesevarianten für das Beispiel 2.2.2 auf Seite 53 dargestellt. Die notwendigen Ressourcen $r \in R$ sind durch die grau hinterlegten Bereiche angedeutet. Die waagerechten Trennlinien zeigen die Taktgrenzen zwischen den einzelnen Operationen. *Variante 1* benötigt einen Multiplizierer und einen Addierer und stellt die Lösung mit geringstem HW-Aufwand ($|R| = 2$) dar. Das Ergebnis ist nach vier Taktperioden verfügbar. *Variante 2* nutzt einen zusätzlichen Multiplizierer ($|R| = 3$) und mindert damit die Berechnungsdauer um einen Takt. In *Variante 3* wurde zusätzlich die Taktperiode verlängert, sodass eine Addition und eine Multiplikation innerhalb eines Taktes berechnet werden können und das Ergebnis bereits nach zwei Takten bereitsteht.

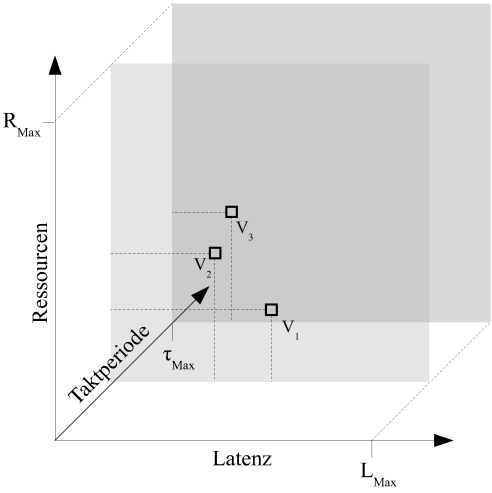


Abbildung 2.17: Entwurfsraum der HLS für Beispiel 2.2.2

Bei wiederholter Ausführung des Problemgraphen oder Algorithmen mit iterativen Anteilen ergibt sich weiteres Optimierungspotential für die Synthese, was im Allgemeinen als Schleifentransformation bezeichnet wird. Illustriert wird eine nützliche Schleifentransformation an Beispiel 2.2.13. Dort wird die Anzahl der Iterationen durch Ausführung zweier Additionen im Schleifenkörper halbiert, was unter Annahme eines geeigneten Speicherzugriffs auch für die benötigte Rechenzeit gilt. Verallgemeinert wird diese Schleifentransformation als “Aufrollen” einer Schleife bezeichnet. Im Maximalfall wird dabei

der Schleifenkörper um die Gesamtanzahl der Iterationen vervielfacht. Das Aufrollen einer Schleife bedingt neben dem zusätzlichen Ressourcenbedarf die wechselseitige Unabhängigkeit der einzelnen Operanten und damit der Operationen innerhalb des Schleifenkörpers.

Beispiel 2.2.13. Schleifenoptimierung mit fester Abbruchbedingung

Unoptimierte Form:

```
for (i=0; i<50; i++) {
    a[i] = a[i] + b[i];
}
```

Optimierte Form:

```
for (i=0; i<50; i+=2) {
    a[i] = a[i] + b[i];
    a[i+1] = a[i+1] + b[i+1];
}
```

Ist dies nicht gegeben, kann durch Pipelining, analog dem Prinzip des in Kapitel 2.1.2.1 beschriebenen Prozessor-Pipelinings, eine quasiparallele Berechnung des Schleifenkörpers erfolgen. Besitzt eine Schleife Datenabhängigkeiten ausschließlich zwischen Operanten innerhalb des Schleifenkörpers, kann *funktionales Pipelining* [PP88] zur Ablaufplanung genutzt werden. *Schleifenfaltung* [Gir87] wird bei Iterationsdatenabhängigkeiten eingesetzt. Weitere Transformationen, unter Einhaltung der algorithmischen Äquivalenz, zur Optimierung der Ressourcenauslastung und des Zeitverhaltes innerhalb des Datenpfades werden beispielsweise durch algebraische Umstellung von Ausdrücken oder die Verschiebung von Codefragmenten erreicht. Entsprechende Untersuchungen wurden bereits in [WT89] vorgestellt.

Die Abbildung der Datenspeicher innerhalb des Algorithmus auf die Speicherressourcen der Zieltechnologie (z.B. Register, Block-RAM, Speicher am Systembus, Eintor- oder Zweitor-Speicher) bildet einen weiteren Freiheitsgrad der HLS. Die Optimale Lösung ergibt sich aus einer Kostenfunktion mit den Parametern: Beschleunigung des Algorithmus, Flächenbedarf, Zugriffsbreite, Verfügbarkeit und Zugriffslatenz. Erste Arbeiten in diesem Bereich zeigten Balakrishnan, Majumdar, Banerji, Linders und Majithia [Bal+88]. Für datenflussorientierte Anwendungen stellen Zebelein, Falk, Haubelt und Teich [Zeb+12] ein Verfahren zur effizienten gemeinsamen Nutzung der Ressourcen vor.

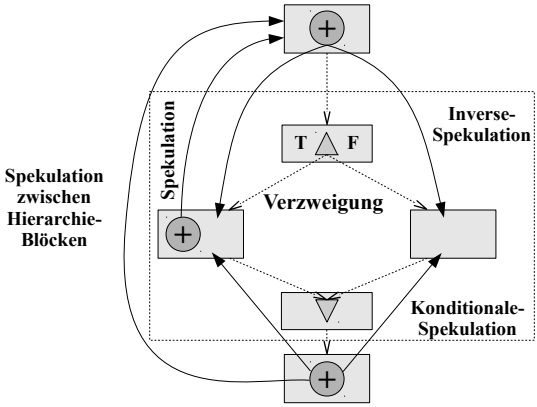


Abbildung 2.18: Spekulative Codeverschiebungen an einer bedingten Verzweigung nach Gupta, Singh, Gupta und Shukla [Gup+06]

Die bisher besprochenen Lösungsmöglichkeiten beschränkten sich auf die Umsetzung des Datenflusses ohne weitere Betrachtung des Kontrollflusses. Aktuelle Arbeiten zur Verbesserung der HLS analysieren den Kontrollfluss (Verzweigungen, Schleifen und Funktionsaufrufe) und bedienen sich teilweise bekannter Verfahren aus den Optimierungsstufen von SW-Compilern. Die Eliminierung nicht ausgeführter Codezweige, die vorgezogene Berechnung von gemeinsam genutzten Ausdrücken in Verzweigungen bzw. von iterationsunabhängigen Ausdrücken in Schleifen (engl. *common subexpression eleminati-on*) sind Beispiele für eine basisblockübergreifende Betrachtung. Die Analyse der Kombinationsmöglichkeiten von Schleifen und der spekulativen Ausführung von Codefragmenten an bedingten Verzweigungen erlauben Transformationsschritte, die den Kontrollfluss darüber hinaus verändern [Gup+06]. Durch spekulative Codeverschiebung an einer bedingten Verzweigung lassen sich Operationen *spekuliert*, *umgekehrt spekuliert*, *bedingt spekuliert* oder *zwischen den bedingten Zweigen verschieben* (vgl. Abbildung 2.18) und damit höhere Freiheitsgrade bei der Ablaufplanung erzeugen.

Werkzeuge, die HLS Methoden umsetzen, sind seit Jahren zahlreich verfügbar und unterscheiden im besonderen hinsichtlich der Eingabe- und Ausgabesprache. Besondere Bedeutung viel dem ersten kommerzielle Werkzeug *Design Compiler* [Kna96] von Synopsys zu, mit dem das Übersetzen von Verhaltensbeschreibungen in VHDL zu synthesefähigem VHDL auf RT-Ebene möglich war. Mittlerweile wurde der *Design Compiler* wieder vom

Markt genommen, weil sich VHDL zur Beschreibung von Verhalten auf Algorithmen-Ebene nicht durchsetzten konnte. Alle wichtigen Hersteller von Programmen zur Entwurfsautomatisierung[47] bieten heute C-basierte HLS-Werkzeuge an [Men10; Cad08; For08; Lim05]. Insbesondere die kommerziellen Werkzeuge integrieren die Kenndaten verschiedener Zieltechnologien oder realisieren Verlustleistungsoptimierung, beispielsweise durch Clock-Gating. Gleichfalls vielfältig sind die akademischen Ansätze zur HLS, von denen stellvertretend das *Olympus Synthese System* [De +90] und der *StreamsC*-Compiler [Gok+00] genannt werden sollen. Ein umfangreicher Überblick ohne Anspruch auf Vollständigkeit wird im Anhang A gegeben.

Aktuelle Arbeiten auf dem Gebiet der HLS befassen sich mit der Auflösung bestehender Einschränkungen, die beispielsweise bei Rekursion oder Zeigerarithmetik bestehen. Aussichtsreich erscheinen Ansätze aus der theoretischen Informatik auf dem Gebiet der Termersetzungssysteme [Ave95], die unter anderem in [MBH12] zur Darstellbarkeit von rekursiven C-Ausdrücken auf der RT-Ebene geführt haben.

Zusammenfassend soll noch einmal festgehalten werden, HLS-Werkzeuge erkennen automatisiert Instruktions- und Schleifenparallelität und sind damit in der Lage, seriellen C-Code in parallele HW anhand Zeit-, Flächen- und Performanz-Restriktionen zu überführen. Darüber hinaus ist stets eine explizite Annotation der Parallelität auf Prozessebene durch den Entwickler notwendig.

2.2.4.4 Ergebnis der High-Level-Synthese

Das Ergebnis der HLS, eine Implementierung (vgl. Definition 2.2.12), lässt sich durch das von Gajski, Dutt, Wu und Lin [Gaj+92] eingeführte Modell eines *Zustandsautomaten mit Datenpfad*[48] formal erfassen. FSMD-Modelle beschreiben ein digitales System auf RT-Ebene und bestehen aus Steuerwerk und Operationswerk.

Abbildung 2.19 zeigt ein Blockschaltbild als generische Umsetzung eines FSMD in digital-synchroner HW. Das Steuerwerk besteht aus einem Zustandsregister sowie zwei kombinatorischen Blöcken zur Berechnung des Folgezustands und der Ausgangssignale. Das Operationswerk wird über ausgehende Kontrollsignale des Steuerwerks gesteuert, sodass Operationen

[47] *Entwurfsautomatisierung*: engl. Electronic Design Automation (EDA)
[48] *Zustandsautomat mit Datenpfad:* (engl.) Finite State Machine with Datapath (FSMD)

und Speicherzugriffe in der notwendigen Sequenz aktiviert werden. Entsprechend meldet der Operationspfad über Status-Bit den Status, z.B. den Abschluss einer Operation, an das Steuerwerk zurück. Sowohl Steuer- als auch Operationswerk sind über Eingangs- und Ausgangssignale in den Kontroll- bzw. Datenpfad der übergeordneten Systemumgebung eingebunden. Für konkrete technologiebezogene Implementierungsmöglichkeiten eines FSMD, die Kombination und Kaskadierung mehrerer FSMD zu einem System sei auf [Gaj+92] verwiesen.

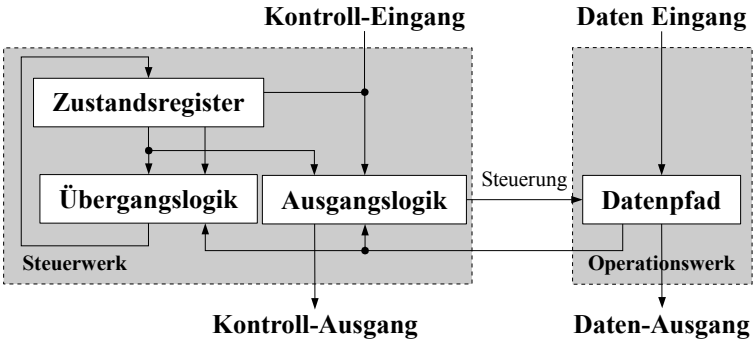


Abbildung 2.19: Generische Implementierung eines Zustandsautomaten mit Datenfluss nach Gajski et. al. Gajski, Dutt, Wu und Lin [Gaj+92]

In Abbildung 2.20 sind beispielhaft der Datenflussgraph zur iterativen Berechnung einer quadratischen Gleichung nach Beispiel 2.2.14 sowie der zugehörige Übergangsgraph dargestellt. Der Übergangsgraph zeigt neben dem Start- und Endzustand einen Zustand für die Initialisierung der Zählvariable und den Schleifenkörper, der aus vier einzelnen Takt-Schritten besteht. Der Datenflussgraph teilt sich in die Berechnung der Gleichung mit Speicherzugriffen sowie die Berechnung und Prüfung der Schleifenbedingung.

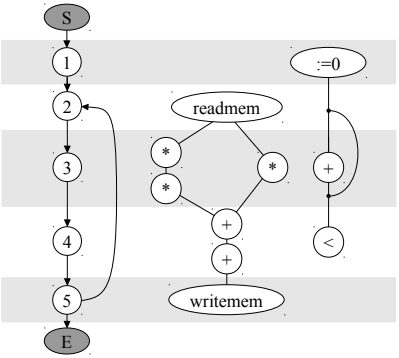


Abbildung 2.20: Übergangs- und Datengraph des FSMD einer möglichen Implementierung von Beispiel 2.2.14

Beispiel 2.2.14. Iterative Berechnung mit lesendem und schreibendem Speicherzugriff:

```
1  int x, y;
2  int i = 0;
3  do {
4      x = readmem();
5      y = A*x*x + B*x + C;
6      writemem(y);
7      i = i++;
8  } while (i < MAX);
```

2.3 Parallele Anwendungskonzepte und Programmiermodelle

Für eine effiziente Nutzung paralleler Rechenarchitekturen bedarf es einer Anwendung, die in parallelisierbare Fragmente gegliedert und auf die Rechenwerke verteilt werden kann. Die methodischen Ansätze der Parallelisierung und die entsprechenden Programmierkonzepte werden im Folgenden einführend erläutert.

Unter der *Parallelisierung* einer Anwendung wird der im Allgemeinen zusätzliche Aufwand bei der Überführung eines sequentiellen Programms in eine parallele Version bezeichnet [Sin07; CSG99]. Dabei erfolgt zunächst die *Dekomposition* der Gesamtaufgabe (Task) in voneinander abhängige Unteraufgaben (Subtasks), die zu einem gewissen Grad gleichzeitig abgearbeitet

werden können. Die Abhängigkeit der Subtasks erfordert eine bestimmte Ausführungsreihenfolge, die durch eine anschließende *Abhängigkeitsanalyse* ermittelt wird. Schließlich erfolgt die Bindung an die Rechenmaschine durch das Aufstellen eines Ablaufplanes (*Scheduling*). Der Ablaufplan legt durch die temporale (Ausführungszeitpunkt) und spatiale (Ausführungsort) Bindung das Laufzeitverhalten der Anwendung auf einem spezifischen parallelen System fest. Das Aufstellen des Ablaufplanes erfolgt statisch zum Entwurfs- bzw. Übersetzungszeitpunkt oder dynamisch zur Laufzeit durch das Betriebssystem (vgl. Kapitel 2.4). Die statische Ablaufplanung kann zusätzlich in das *Orchestrieren* und das Abbilden[49] gegliedert sein, wobei Ersteres die Reduzierung des Datentransfers (z.B. durch Lokalität der Daten) und Zweiteres die Bindung der Subtasks an eine Ausführungseinheit (durch Lokalität der Subtasks zueinander) zum Ziel hat. Abbildung 2.21 stellt die Schritte von der Anwendungsspezifikation bis zum ausführbaren Programm dar. Die Parallelität einer Anwendung unterscheidet sich von der Parallelität des Computersystems und lässt sich anhand des Grades und der Granularität bewerten. Der Grad beziffert die Anzahl gleichzeitig ausführbarer Subtasks bzw. Befehle. Die Granularität bezeichnet das Verhältnis der Größe der Subtasks zum Gesamtprogramm und wird in die Bereiche *grob*, *mittel* und *fein* eingeteilt.

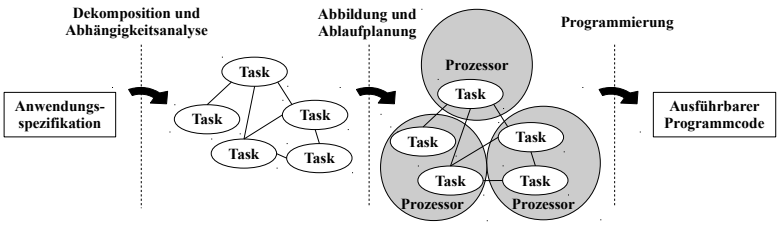


Abbildung 2.21: Parallelisierung einer Anwendung nach Sinnen [Sin07]

Für die Beschreibung einer parallelen Anwendung existieren zwei grundlegende Konzepte: *Prozesse* und *Multithreading*, mit unterschiedlichen Auswirkungen auf ein präemptives Laufzeitsystem[50]. Prozesse sind geschützte Programmeinheiten mit eigenem Code- und Datenbereich und einer eigenen Umgebung (Adressraum). Ein Prozesswechsel bedingt den Wechsel dieser Umgebung. Zur Ausführung eines Prozesses gehört ein Kontext, bestehend

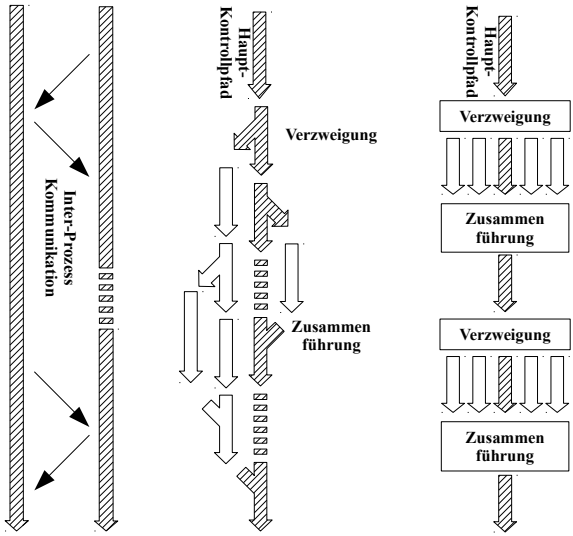
[49] *Abbildung*: engl. mapping

[50] *präemptiv*: vorsorglich; Im Zusammenhang mit dem Laufzeitsystem wird ein Programm vorsorglich unterbrochen, um einem anderen Programm die Ausführung zu ermöglichen.

aus den Registerwerten des Prozessors, der den Prozess aktiv bearbeitet. Dazu zählen gleichsam die Inhalte von Datenregistern, Befehlszähler, Zeiger in den Stapelspeicher[51] und alle weiteren Zustandsinformationen (z.B. Zugriffsrechte, Dateien, Priorität). Ein Prozesswechsel bedingt demnach auch einen Kontextwechsel. Es werden schwere und leichte Prozesse [Gil93] unterschieden. Schwere Prozesse besitzen Kontext und Umgebung, wohingegen sich eine Gruppe leichter Prozesse eine Umgebung teilt, sodass ein Umgebungswechsel innerhalb der Gruppe vermieden werden kann. Synchronisation und Interprozesskommunikation erfolgen über Ressourcen der Laufzeitumgebung (z.B Dateisystem, Kommunikationssockets oder speziell angeforderter gemeinsamer Speicher) wie in Abbildung 2.22a dargestellt und wurden im sogenannten *System V* für UNIX-Betriebssysteme [Ins; USA91] festgelegt. Eine standardisierte Programmierschnittstelle für eine medienunabhängige Nachrichtenübermittlung zwischen Prozessen wurde mit MPI [Inc08] geschaffen.

Beim Multithreading[52] besitzt eine Programmeinheit mehrere Kontrollflüsse, die innerhalb einer gemeinsamen Umgebung ablaufen. Das entsprechende Schema zeigt Abbildung 2.22b. Der Kontext eines Threads beschränkt sich dadurch auf Datenregister, Befehlszähler und den Zeiger auf den Stack, sodass beim Wechsel zwischen den Threads innerhalb einer Programmeinheit nur diese Inhalte eine Sicherung und Wiederherstellung benötigen. Die Kommunikation von Threads erfolgt über den gemeinsamen Adressraum und wird durch Mutex[53], Semaphoren, Bedingungsvariablen oder Barrieren synchronisiert. Die Laufzeitumgebung realisiert Threads entweder direkt durch das Betriebssystem oder im *Userspace*[54] durch eine Bibliothek oder einen eigenen gekapselten Prozess. Die Realisierung im *Userspace* ist ungünstig für die Lastverteilung in SMP-Systemen. Für Einzelprozessorsysteme besteht jedoch ein Performanzvorteile, da zur Verwaltung der Threads kein Umschalten in den Betriebssystemmodus notwendig ist.

[51] *Stapelspeicher*: engl. stack
[52] *Multithreading (engl.)*: Ausführung mehrerer paralleler Programmfäden
[53] *Mutex*: Abkürzung für *mutual exclusion (engl.)*
[54] *Userspace (engl.)*: Abgetrennter Bereich mit niederen Privilegien, in dem gewöhnliche Nutzeranwendungen ausgeführt werden.



(a) Prozesse

(b) Threads nach dem POSIX-Standard Institute of Electrical and Electronics Engineers [Ins94]

(c) OpenMP

Abbildung 2.22: Schematischer Vergleich paralleler Anwendungskonzepte

Aktuelle Betriebssysteme implementieren Threads direkt im System-Kern. Das gilt sowohl für klassische monolithische (Solaris, Linux) als auch moderne Micro-Kern-Systeme (z.B. Mach [Fre13]). Die meisten dieser Umsetzungen der Threadausführung sind sich konzeptuell ähnlich, unterscheiden sich jedoch wesentlich in deren direkter Programmierschnittstelle und diversen subtilen Einzelheiten. Anwendungen auf Basis von systemspezifischen Threads sind in keiner Weise portierbar. Eine standardisierte Programmierschnittstelle wurde durch das Institute of Electrical and Electronics Engineers (IEEE) mit dem POSIX-Standard [Ins94] geschaffen und unter dem Begriff *POSIX-Threads*, kurz *Pthreads* bekannt. Alternativ besteht mit *OpenMP* [CJP07] die Möglichkeit, auf höherer Abstraktionsebene threadbasierte Parallelisierung umzusetzen. *OpenMP* abstrahiert die Details zur die Behandlung von Daten-

komposition, Thread-Verwaltung, Nachrichtenvermittlung und Kontrollfluss aus parallelisierbaren Anteilen des Hauptthreads (vgl. Abbildung 2.22c).

Aus historischen Gründen beziehen sich die genannten Programmiermodelle in der Hauptsache auf die Sprachen C/C++. Die dahinter liegenden Konzepte sind in jüngeren Programmiersprachen, wie z.B. *Java*, *Python* oder *Visual Basic .NET*, gleichfalls umgesetzt. Weitere architekturspezifische Modelle sind *OpenCL* (Apple, Khronos Group), *CUDA* (NVidia), *Sequoia* (Stanford University), *Cilk* (MIT), *Ct* (Intel) und *CellSs* (Barcelona Supercomputing Center).

2.4 Scheduling-Strategien

System erlauben es, mehrere Aufgaben nebenläufig (Multitasking) auf mehreren Rechenressourcen (Multiprozessor) abzuarbeiten. Dazu bedarf es einer temporalen und spatialen Abbildung von Aufgaben $j \in J$ der Länge l_j auf eine Menge von Prozessoren $m \in M$. Die optimale Abbildung $J \rightarrow M$ mit der kürzesten Ausführungszeit für alle Aufgaben zu finden, wurde als NP-hartes Problem [GJ90] klassifiziert. Die Ermittlung einer Ablaufplanung erfolgt entweder statisch oder dynamisch. Statische Scheduling-Strategien weisen den Aufgaben den Ausführungszeitpunkt und die Rechenressourcen zum Übersetzungszeitpunkt auf der Basis geschätzter Ausführungszeiten zu. Setzt man perfekte Informationen über die Ausführungszeit voraus, lassen sich mit statischem Scheduling ideale Ablaufpläne erzeugen, mit denen kürzestmögliche Ausführungszeiten für alle Aufgaben realisierbar sind.

In der Realität sind die Ausführungszeiten häufig nichtdeterministisch, sodass von der maximal anzunehmenden Ausführungszeit[55] ausgegangen wird. Das führt zu einer suboptimalen Auslastung des Systems. Dynamische Ablaufplanung umgeht das Problem mangelnder Informationsgenauigkeit zur Übersetzungszeit durch Verschiebung der Scheduling-Entscheidung in die Laufzeit. Die einzelnen Planungsentscheidungen werden durch Ereignisse getriggert und erfolgen für die Menge ausführungsbereiter Aufgaben am Triggerzeitpunkt. Die Entscheidung kann damit in Abhängigkeit wechselnder Prozessorauslastung getroffen werden.

Die nachfolgenden Kriterien dienen der Charakterisierung von Scheduling-Strategien.

[55] *maximal anzunehmende Ausführungszeit*: engl. worst case execution time

Durchsatz Die abgearbeiteten Aufgaben je Zeiteinheit bestimmen den Durchsatz eines Systems. Die Scheduling-Strategie sollte den Durchsatz maximieren.

Prozessorauslastung Die Auslastung eines Prozessors ist ein Maß für die Zeit, in der der Prozessor Aufgaben ausführt. Erfolgreiches Scheduling maximiert die Prozessorauslastung.

Durchlaufzeit Die Durchlaufzeit ergibt sich aus der Dauer zwischen zwei aufeinanderfolgenden Zuteilungen eines Prozessors auf eine Aufgabe und ist durch das Scheduling minimal zu halten.

Wartezeit Die Zeit, die eine Aufgabe im wartenden Zustand, d.h. ohne zugeteiltem Prozessor, verbringt, sollte minimal sein.

Antwortzeit Die Dauer zwischen einer externen Anfrage an das System bis zu dessen Reaktion ist die zu minimierende Antwortzeit.

Echtzeit Man spricht von Echtzeit, wenn die Dauer eines Vorgangs bzw. die Wartezeit vorhersehbar ist, d.h. die Antwortzeit des Betriebssystems eine definierbare Zeitspanne nicht überschreitet.

Fairness Fairness ist ein Maß für die Gerechtigkeit und betrachtet unverhältnismäßig lange Wartezeiten einzelner Aufgaben gegenüber bevorzugten Aufgaben.

Dynamische Scheduling-Strategien lassen sich in *präemptive* und *nichtpräemptive* Verfahren unterscheiden. Erstere unterbrechen die Bearbeitung einer Aufgabe in Verantwortung des Betriebssystems. Zweitere warten entweder, bis die Aufgabe freiwillig, d.h. kooperativ, die Bearbeitung unterbricht oder die Bearbeitung vollständig abgeschlossen ist. Im Sinne der Aufgabenstellung werden in den folgenden Ausführungen ausschließlich präemptive Verfahren betrachtet. Die Optimierung der vorgestellten Kriterien hat teilweise gegensätzliche Auswirkungen und bedingt einen Kompromiss bei der Strategiewahl in Abhängigkeit von der Betriebsform des Zielsystems. Das Kriterium *Echtzeit* ist insbesondere für echtzeitfähige Betriebssysteme und Anwendungen von entscheidender Bedeutung. Diese werden aus den folgenden Betrachtungen ausgeschlossen und stellen einen Anknüpfungspunkt für weitere Forschungsarbeiten auf dem Gebiet heterogener ACS dar.

2.4.1 Shortest Job First (SJF)

Das Scheduling-Verfahren nach dem SJF-Prinzip startet Aufgaben mit der voraussichtlich kürzesten Ausführungszeit zuerst. Dazu werden die Aufgaben

in einer Liste nach deren potentieller Ausführungszeit sortiert. Aufgaben, die eine gegebene Ausführungszeitobergrenze überschreiten, werden am Ende der Liste entsprechend dem Eintreffen nach dem First-Come-First-Serve (FCFS) einsortiert.

Der Verwaltungsaufwand beim SJF ist mit der Abarbeitung einer Liste als gering einzuschätzen, wodurch eine hohe Auslastung der Prozessoren zu erwarten ist. Durch die bevorzugte Ausführung kleiner Aufgaben ist mit geringer Durchlaufzeit und minimalen Wartezeiten für kleine Aufgaben zu rechnen. Große Aufgaben erfahren allerdings eine geringe Fairness, die sich durch eine kleine Ausführungszeitobergrenze verbessern lässt. Sowohl Antwortzeit als auch Wartezeiten unterliegen in direkter Abhängigkeit mit der Prozessgröße hohen Schwankungen. Die Qualität des Verfahrens steht und fällt mit der Genauigkeit, in der die Ausführungslänge im Vorfeld bekannt sein kann. Die Priorisierung von Aufgaben ist bei SJF nicht vorgesehen.

2.4.2 Round-Robin

Das Round-Robin Verfahren ist das klassische Zeitscheibenverfahren nach dem FIFO-Prinzip. Alle Aufgaben stehen in einer Warteschlange und die vorderste Aufgabe kommt zur Ausführung. Die Ausführung erfolgt exklusiv für eine festgelegte Dauer einer Zeitscheibe. Endet die Bearbeitung nicht innerhalb dieser Zeitscheibe oder wird die Kontrolle freiwillig von der Aufgabe abgegeben, entzieht das Betriebssystem der Aufgabe die Berechnungsressource und reiht die Aufgabe hinten in die Warteschlange ein.

Dieses Verfahren behandelt alle Aufgaben gleich. Das erzeugt eine Gleichverteilung der Warte- und Durchlaufzeiten. Die Komplexität des Verfahrens ist durch die Warteschlange relativ gering, entsprechend ist eine gute Prozessorauslastung zu erwarten, die jedoch mit kleinen Zeitscheiben und damit einhergehenden häufigen Kontextwechseln abnimmt. Unfair werden Aufgaben mit intensiver E/A-Kommunikation behandelt, da mit jedem E/A-Vorgang die Rechenressource aufgegeben wird und damit selten die volle Zeitscheibe genutzt werden kann. In Szenarien mit unterschiedlich wichtigen Aufgaben ist dieses Verfahren ungeeignet, weil es keine Priorisierung vornimmt.

2.4.3 Prioritätsbasiertes Scheduling

Bei dieser Form des Schedulings besitzt jede Aufgabe eine Priorität und nur die Aufgabe mit der höchsten Priorität kommt zur Bearbeitung. Das kann

präemptiv oder nicht-präemptiv erfolgen, wenn eine neue Aufgabe mit höherer Priorität im System gestartet wird. Die Prioritätenvergabe erfolgt statisch oder dynamisch.

Die Durchlaufzeit hängt bei diesem Verfahren von der Größe der Aufgaben ab und lässt kaum allgemeine Aussagen zu. Eindeutig festzustellen sind hingegen Nachteile bezüglich der Kriterien Fairness, Wartezeit und Antwortzeit. Die Komplexität ist hingegen gering und erfordert bescheidenen Verwaltungsaufwand. Von besonderem Nachteil ist die unfaire Behandlung der niedrig priorisierten Prozesse, die bis zur Nichtbearbeitung reichen kann. Zudem kann es zur Prioritätsinversion kommen, wobei ein Prozess mit niedrigerer Priorität eine Systemressource belegt, die ein höher priorisierter Prozess gleichfalls anfordert. Die Ressource, deren Besitzer durch geringe Priorität niemals ausgeführt wird, kann ggf. nie frei werden. Um dies zu vermeiden, werden prioritätsbasierte Verfahren häufig mit anderen kombiniert, wie zum Beispiel beim Multi-Slot Scheduling aus [Sim01], das auch als priorisiertes Round-Robin Verfahren begriffen werden kann. Aufgaben mit höherer Priorität bekommen eine längere Zeitscheibe zugewiesen. Die vergebenen Prioritäten sind statisch und haben auch beim nächsten Aufruf den gleichen Wert (vgl. auch [Bil08]).

2.4.4 Multilevel-Feedback Scheduling (MLFS)

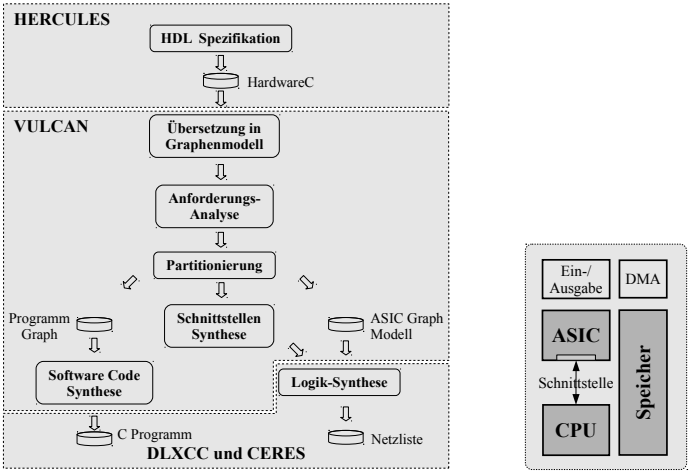
Diese Strategie basiert auf den Ideen von Corbató, Merwin-Daggett und Daley [CMD62] und kombiniert die zuvor genannten Verfahren, indem mehrere Warteschlangen unterschiedlicher Priorität eingeführt werden. Diese werden wiederum im Round-Robin Verfahren mit jeweils unterschiedlichen Zeitscheibenlängen bedient. Innerhalb der Warteschlange wird FCFS verfolgt. Die Aufgaben wechseln die Warteschlangen dynamisch auf Basis in der Vergangenheit verbrauchter Zeitscheiben. Auf diese Weise wird Fairness sowohl für E/A-intensive, als auch für rechenintensive Aufgaben erreicht. Mit MLFS wird, wie bei Round-Robin eine Gleichverteilung der Warte-, Antwort und Durchlaufzeiten erreicht. Nachteilig wirkt sich der hohe Verwaltungsaufwand durch die komplexe Verfahrenskombination aus, sodass mit einer verminderten Prozessorauslastung zu rechnen ist. Das MLFS wird in verschiedenen UNIX-Systemen eingesetzt [MM06].

3 Stand der Technik

3.1 Anwendungsentwurf für heterogene Computersysteme

Der Entwicklungsstand in der Computertechnik hat Ende der 80er Jahre zu Systemen geführt, die bis zum Endverbraucher im Massenmarkt abgesetzt werden können und eine Vielfalt anwendungsspezifischer SW und HW integrieren. Damit entstand der Bedarf nach geeigneten computergestützten Entwurfsverfahren und -werkzeugen, zunächst für deren Systementwurf und später für den Anwendungsentwurf auf diese Systeme. Der einheitliche Entwurf von gemischten HW/SW-Systemen ist Gegenstand vielfältiger Arbeiten, zum Beispiel von De Micheli [De 94a], und wird im Englischen als *HW/SW Co-Design* bezeichnet.

Mit dem *VULCAN*-Projekt stellt Gupta [Gup95] den ersten geschlossenen Entwurfsablauf für ein heterogenes System vor. Ausgangspunkt ist der Entwurf einer einheitlichen Spezifikation für beide Domänen in einer gemeinsamen Entwurfsumgebung. Für die HW/SW-Cosynthese wurde das HLS-Werkzeug *Hercules* [KD90] um die Synthese bzw. Generierung des entsprechenden SW-Codes aus der prozeduralen HW-Beschreibungssprache *HardwareC* [De +90] zu einem geschlossenen Entwurfssystem, durch das VULCAN-System [GM91] erweitert. Herzstück von VULCAN ist die automatisierte Systempartitionierung auf Basis eines graphentheoretischen Systemmodells auf Befehlsebene. Die Zielarchitektur von VULCAN, eine busgetriebene Prozessor-ASIC-Architektur mit gemeinsamen Speicher, ist in Abbildung 3.1b dargestellt. Bemerkenswert ist, dass der generische Prozessor als reprogrammierbare Ressource angesehen wird, der den Funktionsumfang eines oder mehrerer ASIC erweitert, also bereits in dieser frühen Phase der Forschung eine HW-zentrische Systemsicht vorlag.



(a) Entwurfsablauf und Syntheseschritte (b) Zielarchitektur

Abbildung 3.1: VULCAN-Projekt von R.K. Gupta Gupta [Gup95]

Abbildung 3.1a zeigt den Aufbau des VULCAN-Systems mit vorgeschalteter HLS und nachgelagertem SW-Compiler. Die abstrakte Sicht auf das VULCAN-System kann als Vorläufer der vorliegenden Arbeit gesehen werden und entspricht im grundlegenden Aufbau der Struktur aller folgenden Forschungsanstrengungen auf diesem Gebiet. Es liefert ein detailliertes und mathematisch fundiertes Systemmodell und garantiert Constraints bei der Ausführung des Graphen. Dies erlaubt den Schluss auf Verzögerungszeiten in der SW und HW sowie auf die auftretenden Buslasten und resultiert in einer optimalen aber statischen Systempartitionierung. Einschränkungen ergeben sich innerhalb dieser Grenzen durch den zwingenden Determinismus in der Eingangsbeschreibung und die statische Systempartitionierung. Durch den HW-orientierten Ansatz wird der Umfang der SW-Domäne auf in HW realisierbare Funktionen beschränkt.

Im Kontrast zur HW-zentrischen Sicht von VULCAN steht die Klasse der SW orientierten Cosyntheseverfahren. Einer deren frühester Vertreter ist mit *Cosyma* [EHB93] an der TU Braunschweig entwickelt worden. Ausgehend von einer Mikrokontroller-Progammbeschreibung erfolgt die Synthese eines spezifischen Zusatzprozessors auf RT-Ebene, der als Coprozessor neben dem

Mikrokontroller agiert und über einen gemeinsamen Speicher angebunden ist. Als Eingangsbeschreibungsprache dient C^x, eine ANSI-C Erweiterung für die Annotation von Zeit-Constraints sowie Task- und Intertask-Kommunikation. Die HW/SW-Partitionierung erfolgt automatisch und blockorientiert in einem zweifach geschachtelten Verfahren und auf der Basis geschätzter Ausführungs- und Kommunikationszeiten.

Ein wichtiger Vertreter dieser Klasse ist der von A. Koch et. al. an der TU Darmstadt vorgestellte COMRADE Compiler [KK05]. Ziel ist ein durchgehender Entwurfsablauf einer kombinierten HW/SW-Anwendung auf der Grundlage von uneingeschränktem C nach dem ANSI-Standard. Der zentrale Compiler ist eine Weiterentwicklung von GarpCC [CHW00] und NIMBLE MacMillen [Mac01]. An die Synthese angeschlossen folgt statisches und dynamisches Profiling, bei dem Ausführungskerne in der Eingangsbeschreibung identifiziert und anhand von Ausführungshäufigkeit und Komplexität bewertet werden. In die Bewertung wird auch die Menge der zwischen den Kernen zu transferierenden Daten einbezogen. Darauf aufbauend wird über die HW/SW-Partitionierung der Kerne entschieden, wobei zusätzlich eine optimale Auslastung der HW-Ressourcen angestrebt wird. Für die Optimierung werden die notwendigen Ressourcen eines HW-Kerns über die generische HW-Modulbibliothek GLACE [NK01] geschätzt. Die COMRADE-Methode besitzt kaum Beschränkungen hinsichtlich der Eingangsbeschreibung. C-Konstrukte, die nicht für HW synthetisierbar sind, werden in SW ausgeführt [LK07]. Sowohl HW/SW-Partitionierung als auch Ablaufplanung haben statischen Charakter. Bei COMRADE wandelt sich der Schwerpunkt des Entwurfes vom System hin zum Anwendungsentwurf auf einem HW/SW-System.

Das *MOLEN* Projekt von Prof. S. Vassiliadis der TU Delft befasst sich gleichfalls mit dem Anwendungsentwurf und legt ein gleichnamiges Programmierparadigma zu Grunde. Das sequentielle *Molen Paradigma* erlaubt die parallele Ausführung von Befehlen in rekonfigurierbarer HW. Der Befehlssatz eines generischen Prozessors wird erweitert, um Operationen auf rekonfigurierbaren Prozessoren beschleunigt auszuführen. Abbildung 3.2 zeigt die Organisationsform einer zugehörigen *Molen*-Rechenmaschine als heterogene Systemarchitektur. Zur Ausführung in der $\mu\rho$-Einheit des rekonfigurierbaren Prozessors kommen komplexe Operationen auf Basis von Mikrocodeprogrammierten HW-Kernen. Ein spezieller *Molen*-Compiler [Pan07] übersetzt ein durch Pragma annotiertes C und trifft dabei die Auswahl der in HW bzw. SW auszuführenden Operationen. Neuere Publikationen zeigen Arbeiten an einem linuxbasierten Betriebssystem sowie einer Unter-

stützung der *OpenMP [CJP07]* Thread-Bibliothek. Damit unterstützt das Molen-Projekt die Parallelisierung höherer Granularität. Das HLS-Werkzeug DRAWV [Yan+07] erzeugt die HW-Kerne auf Prozess- bzw. Thread-Ebene.

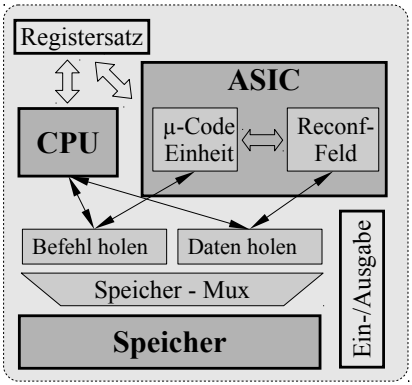


Abbildung 3.2: Organisation des polymorphen Prozessors im *MOLEN*-Projekt [VWC01]

Neben den besprochenen SW-orientierten Cosyntheseverfahren bestehen mit C2H [Alt09], Handle-C [Lim05], StreamC [Gok+00], SA-C [Naj+03], und SpecC [Gaj+00] weitere Projekte mit abgewandelten C-Dialekten, bei denen die HW/SW-Partitionierung jedoch manuell, d.h. durch explizite Auswahl des in HW abzubildenden Programmteils, erfolgen muss. Entwurfssysteme für Anwendungen auf rekonfigurierbaren Architekturen, die einen generischen Prozessor hauptsächlich für Steuerzwecke enthalten, sind unter anderem in [Ros+09; Sch+09] beschrieben.

Neben C/C++ eignen sich weitere Hochsprachen zum Einstieg in das *HW/SW Co-Design*. Mit JHDL [BH98] existiert eine Arbeit, die eine Java-basierte Anwendungsbeschreibung mit anschließender Abbildung und Ausführung auf einem HW/SW-System erläutert. Die Partitionierung erfolgt entlang einzelner Java-Klassen. Es wird eine heterogene Systemsimulationsumgebung mit Unterstützung dynamischer Rekonfiguration für die HW-Ressourcen beschrieben. Mit dem *Liquid Metal* [Hua+08] Projekt und der Java kompatiblen Sprache *LIME* [Aue+10] existiert eine aktuelle Arbeit, die ein breites Feld heterogener Architekturen adressiert und dynamische Repartitionierung zur Laufzeit unterstützt. Ausgehend von modellbasierten

Entwurfsverfahren ist auch ein *MATLAB/Simulink*-Modell als möglicher Entwurfseingang zum Co-Design zu sehen. In [Dil+07] wurde der Versuch unternommen, aus der Skriptsprache Python [Pyt13] heraus Anwendungen für ein heterogenes System zu entwickeln.

3.2 Laufzeitumgebungen und Architekturen

Laufzeitumgebung und Systemarchitektur hängen sehr eng miteinander zusammen. Eine Laufzeitumgebung ist auf einer Architektur nur lauffähig, wenn eine Portierung[1] auf einen spezifischen Prozessorentyp des Systems gegeben ist. Andererseits entfaltet eine Systemarchitektur die volle Leistungsfähigkeit nur mit einer Laufzeitumgebung, die die Spezifika der Architektur, z.B. durch geeignetes Scheduling und HW-Treiber, unterstützt. Zu der in Kapitel 2.1.5 eingeführten allgemeinen Typisierung heterogener Architekturen ist in [CH02] ein umfassender Überblick existierender Ansätze mit Kopplung von generischen Prozessoren[2] und rekonfigurierbarer Logik gegeben. Für die spezielle Betrachtung heterogener Architekturen in Verbindung mit Multithreading gibt [ZKG09] einen aktuellen Stand. Aus architektureller Sicht wird dort eine Untergliederung der Ansätze in keine, implizite oder explizite Multithreading-Unterstützung eingeführt. Eigene Untersuchungen und Recherchen wurden in [BRH10] veröffentlicht.

Nachdem rekonfigurierbare HW in den 90er Jahren den endgültigen Durchbruch in Breite und Anwendung gefunden hat, wurde die Notwendigkeit einer Laufzeitunterstützung für Anwendungen auf diesen Ressourcen erkannt. Brebner [Bre96] führt zunächst den Begriff der virtuellen HW ein (vgl. Kapitel 2.1.4.2). Darauf aufbauend definierte eine Forschergruppe der Universiy of South Australia um Grant Wigley in [DW99] die fundamentalen Funktionen eines Betriebssystems für Computersysteme mit rekonfigurierbaren Komponenten. Deren erste Referenzimplementierung [WK02] realisiert Interprozesskommunikation und eine statische Ablaufplanung, wobei die Anwendungen als DAG interpretiert, partitioniert und nach dem Gang-Prinzip [Fei97] auf die Ressourcen verteilt werden.

Basieren diese ersten Ansätze von Multitasking noch auf der kooperativen Freigabe der Rechenressourcen [SLM00], eröffnete sich mit der dynamischen Verteilung der HW-Ressourcen ein breites Forschungsfeld. Simmler [Sim01] entwickelte eine Client-Server basierte Laufzeitumgebung mit präemptiver

[1] Portierung: Anpassung einer SW an eine spezifische Plattform
[2] *generischer Prozessor*: engl. general purpose processor

Ablaufplanung [Lev+00]. Die Arbeiten von Danne [Dan04] erörtern das Speichermanagement in heterogenen Systemen und leiten daraus statische Ablaufverfahren bei konkurrierendem Buszugriff und die dynamische Allokation zentraler Speichersegmente ab. Schließlich wird in [DMP06] ein HW-basiertes Betriebssystem vorgestellt, welches alle Dienste in einem FPGA realisiert und das präemptive Multitasking auf FPGA basierten Rechenplattformen unter Berücksichtigung von Echtzeitanforderungen erlaubt. Ein alternativer Ansatz wird von Peck, Anderson, Agron, Stevens, Baijot und Andrews [Pec+06] im HThread-Projekt verfolgt. Deren Laufzeitumgebung unterstützt das threadbasierte Programmiermodell (vgl. Kapitel 2.3) auch für rekonfigurierbare HW und implementiert zusätzlich zentrale Funktionen des Betriebssystems in der HW-Domäne. Der ReConOS-Ansatz von Lübbers und Platzner [LP07] zielt ebenfalls in diese Richtung. Die Funktionalität der rekonfigurierbaren HW wird dort über eine Thread-Bibliothek sowie entsprechende Betriebssystemrufe angesprochen. Zusätzlich stehen der HW Systemrufe an die Laufzeitumgebung zur Verfügung [LP08], die über Stellvertreter-Threads in der SW-Domäne umgesetzt werden. Gemein ist diesen Ansätzen, dass deren Betrachtung auf die Verwaltung und die Schnittstellenanbindung der HW-Ressourcen eines heterogenen Systems beschränkt bleibt.

Eine dritte Kategorie der Laufzeitumgebungen umfasst Ansätze, die Ressourcen sowohl der SW- als auch der HW-Domäne verwalten und darüber hinaus eine Migration der Anwendung bzw. einzelner Teilaufgaben über die Domänen-Grenze hinweg realisieren und verwalten. Aufbauend auf die Xputer-Architektur von Hartenstein, Hirschbiel und Weber [HHW90] stellen Kress, Hartenstein und Nageldinger [KHN97] mit XOS eine Umgebung vor, die dynamische Ablaufplanung sowie Speicherallokation bereitstellt. Die Unterbrechung und Migration von Tasks auf Xputer-ALU-Array sind an den Kontenpunkten des DAG einer Anwendung möglich. Für FPGA-basierte heterogene Architekturen zeigen erstmals Noguera und Badia [NB02] entsprechende Untersuchungen und setzten dabei gleichfalls auf ein zweistufiges Analyseverfahren. Zunächst ermittelt eine statische Analyse einen Anwendungsgraphen, dessen Knoten zur Laufzeit auf HW- und SW-Einheiten verteilt werden. Den Funktionsnachweis des Schedulingverfahrens erbringen Badia et. al. auf Simulationsbasis. Die Gruppe um Mignolet et. al. realisierte mit OS4RS [Mar+04] einen Ansatz bis hin zur prototypischen Implementierung. Das in Schichten organisierte Betriebssystem verwaltet HW und SW dynamisch durch einen zweistufigen Scheduler [Mig+03], wobei das Auslösen der Kontextwechsel entweder umgebungsgetrieben, ablaufgetrieben oder datengetrieben erfolgen kann. Das zugrundeliegende Modell zur Interprozesskommunikation ist nachrichtenbasiert. Ein alternatives Verfahren

zur Kommunikation verfolgen Strunk, Heinig, Volkmer, Rehm und Schick [Str+09] auf Basis eines virtuellen Dateisystems.

3.3 Unterbrechung und Migration

Für die einzelnen Domänen eines heterogenen Systems existieren Lösungen für die Unterbrechung und Migration. In der SW-Domäne ist es bereits seit der klassischen VNM (vgl. Kapitel 2.1.2) vorgesehen, nach der vollständigen Abarbeitung eines Befehls die Ausführung eines Programms zu unterbrechen und in eine Service-Routine zu springen. Aktuelle Prozessoren mit Befehlspipeline erlauben eine Unterbrechung des sequentiellen Kontrollflusses in nahezu jedem Takt. Den Ausführungszustand der Anwendung bilden die Funktionsregister des Prozessors (Befehlzähler, Zeiger auf Stapelspeicher, Registersatz usw.) ab. Diese Informationen können im Moment der Unterbrechung gesichert werden und bei Fortsetzung der Ausführung wieder hergestellt werden. Unter Beachtung der Gegebenheiten der Speicherarchitektur und Rettung des Stack-Inhaltes lässt sich durch Transfer der Registerinhalte die Anwendung auf einen beliebigen anderen Prozessor migrieren. Für weiterführende Details zur Unterbrechung in der SW-Domäne sei auf die Standardwerke in [HP07; Mar00] verwiesen.

Berechnungen in der HW-Domäne sind, unterstellt man eine synchrone digitale Schaltung, zu jedem Taktzyklus unterbrechbar. Durch Anhalten des Taktsignals werden die Register-Stufen zwischen den parallel arbeitenden kombinatorischen Schaltungsteilen an der Übernahme der folgenden Ausgabesignale gehindert. Der Ausführungstand der Berechnung wird durch den Zustand sämtlicher Register- und Signalzustände in der Schaltung repräsentiert. Erlaubt die Schaltungsstruktur einen lesenden und schreibenden Zugriff auf diese Informationen ist die Migration, zwischen ein und derselben oder zwei identischen Schaltungen möglich. Entsprechende Untersuchungen existieren bezüglich rekonfigurierbarer HW-Strukturen, vornehmlich im Umfeld von FPGA-Schaltkreisen.

Für FPGA lassen sich zwei grundlegende Verfahren unterscheiden. SRAM-basierte Schaltkreise erlauben das Rücklesen des Konfigurationsbitstromes. Ohne zusätzlichen Aufwand an HW-Strukturen kann der Zustand der Schaltung gesichert und durch erneutes Laden des Bitstromes wiederhergestellt werden. Der wesentliche Nachteil der sogenannten *Readback*-Verfahren[3] ist dessen Schaltkreisabhängigkeit. Da im Bitstrom sowohl Schaltungsstruktur

[3] *readback (engl.)*: zurücklesen

als auch Zustand kodiert sind, ist diese Technik zunächst nur für genau diesen einen Schaltkreis nutzbar. Weiterführende Arbeiten umgehend dieses Problem und extrahieren Zustandsinformationen mit erweiterten Kenntnissen über den Aufbau des Bitstromes. Das hat allerdings den Nachteil hochgradiger Technologieabhängigkeit. Arbeiten auf Basis der Readback-Methode sind [Tri+97; GLS99; Lev+00; SLM00; Sim01; WP02; Ull+04]. Koch, Ahmadinia, Bobda und Kalte [Koc+04] zeigen eine erweiterte Methode mit reduzierter Dispatchlatenz, indem der Konfigurationsbitstrom innerhalb des FPGA nach Zustandsinformationen gefiltert wird. Unterbrechung und Migration innerhalb der HW-Domäne wurde durch Kalte und Porrmann [KP05] auf Basis der Offline-Prozessierung von Bitströmen realisiert.

Das zweite Verfahren fügt zusätzliche HW auf RT-Ebene direkt in das Zieldesign ein. Die den Schaltungszustand haltenden Strukturen, die Register, werden um einen "Unterbrechungsmodus" erweitert, in dem das Auslesen oder Vorladen des Inhaltes möglich ist. Das Zieldesign erhält zusätzlich eine Schnittstelle über die diese Inhalte für die Systemverwaltung zugänglich sind. In [JTW07] werden die erweiterten Register beispielsweise in einer Scan-Kette zusammengeschalten und nach Außen über eine JTAG-Schnittstelle angebunden. Werden alle Register des Zieldesigns entsprechend angepasst, kann die Schaltung in jedem Takt und mit sehr geringer Dispatchlatenz unterbrochen werden. Entscheidender Nachteil ist der damit verbundene Aufwand an HW-Ressourcen. Alternative Ansätze [KKP97; KKP06; Hin+06] reduzieren diesen Aufwand indem die Schaltung nur zu ausgewählten Zeitpunkten unterbrochen werden kann. Dabei steigt die Verzögerung bei der Reaktion auf eine Unterbrechungsanfrage.

Die vorgestellten Ansätze erlauben jeweils die Migration innerhalb der gleichen Domäne, SW oder HW. Im Blickpunkt der vorliegenden Arbeit steht die domänenübergreifende Migration innerhalb des gesamten heterogenen Systems. Mignolet, Nollet, Coene, Verkest, Vernalde und Lauwereins [Mig+03] veröffentlichten den theoretischen Ansatz einer domänenübergreifenden Migration auf Basis von Unterbrechungspunkten in einer Verhaltensbeschreibung der OCAPI-xl Programmierumgebung [Van+01] im Zusammenhang mit dem Readback-Verfahren nachLevinson, Männer, Sessler und Simmler [Lev+00]. Diese Arbeit führt die Idee von Mignolet et. al. weiter und ergänzt sie um die automatische und optimierte Verteilung von Unterbrechungspunkten. Zusätzlich wird die Funktionalität anhand der praktischen Implementierung von Laufzeitumgebung und ACS nachgewiesen.

In der Arbeitsgruppe um J. Teich und C. Haubelt an der Universität Erlangen besteht mit der Sprache *SysteMoC* und dem Werkzeug *SystemCoDesi-*

gner ein Systemlevel-Entwurfsansatz für dynamisch rekonfigurierbare HW-Systeme, der auch HW/SW-Codesign Aspekte berücksichtigt [Str+07]. Koch, Haubelt, Streichert und Teich [Koc+07] stellen mit so genanntem *HW/SW-Morphing* einen drauf aufbauenden domänenübergreifenden Migrationsansatz vor, der ebenfalls auf Unterbrechungspunkten basiert. Als Morph- bzw. Checkpoints bezeichnetet, werden Punkte im Kontroll- und Datenfluss einer *SysteMoC*-Beschreibung definiert, an denen eine Sicherung des Ausführungsstandes und Migration erfolgen kann. In [KHT07] werden die Checkpoints als Sicherungspunkte genutzt, um fehlertolerante Systeme aufzubauen. Der zusätzliche Ressourcenbedarf und die einhergehenden Latenzprobleme wurden in diesen Arbeiten bereits erkannt und teilweise formal aufgefasst. Eine automatisierte und optimierte Verteilung erfolgt nicht. Die Anwendung ist gebunden an das inhärente Ausführungsmodell von *SysteMoC*, einem datenflussgetriebenen Schema voneinander abhängiger Tasks, die als Aktoren eigenständig ausgeführt werden sobald eingehende und ausgehende Ports Bereitschaft signalisieren. Die Implikationen auf eine entsprechende Laufzeitumgebung werden zunächst nicht betrachtet.

4 Entwurf dynamisch verteilter Anwendungen für heterogene Computer

Der als Ziel der Arbeit vorgestellte Entwurfsablauf für eine parallelisierbare Anwendung auf einem heterogenen System wurde in [RH06] erstmals veröffentlicht und soll in diesem Kapitel detailliert vorgestellt werden. In Kapitel 2 wurden die Grundlagen, Werkzeuge und Prinzipien des Entwurfes jeweils für die Domänen HW und SW erläutert und Laufzeitumgebungen vorgestellt. Dies bildet die Basis einer Entwurfsmethodik für dynamisch verteilte Anwendungen auf adaptiven HW/SW-Systemen. Der Gesamtablauf des Entwurfes fügt sich zu der in Abbildung 4.1 dargestellten Methodik. Das Vorgehen erfolgt iterativ wechselnd zwischen den auf der linken Abbildungsseite dargestellten Entwurfsschritten mit abnehmender Abstraktion und simulativen Test- bzw. Verifikationsschritten und Constraint-Prüfung auf der rechten Seite. In den folgenden Abschnitten werden der Ablauf und die Spezifika dieses Entwurfs erläutert, wobei für die grundlegende Ausführung einzelner Teil- und Syntheseschritte auf die Kapitel 2.3 und 2.2.2 verwiesen sein soll. Nicht in der Abbildung dargestellt ist die Entwicklung einer Test- und Stimulationsumgebung (Testbench) für Simulation und Verifikation. Die Testumgebung ist aus der Spezifikation abzuleiten und entsprechende Testfälle parallel zu den Entwurfsschritten zu verfeinern.

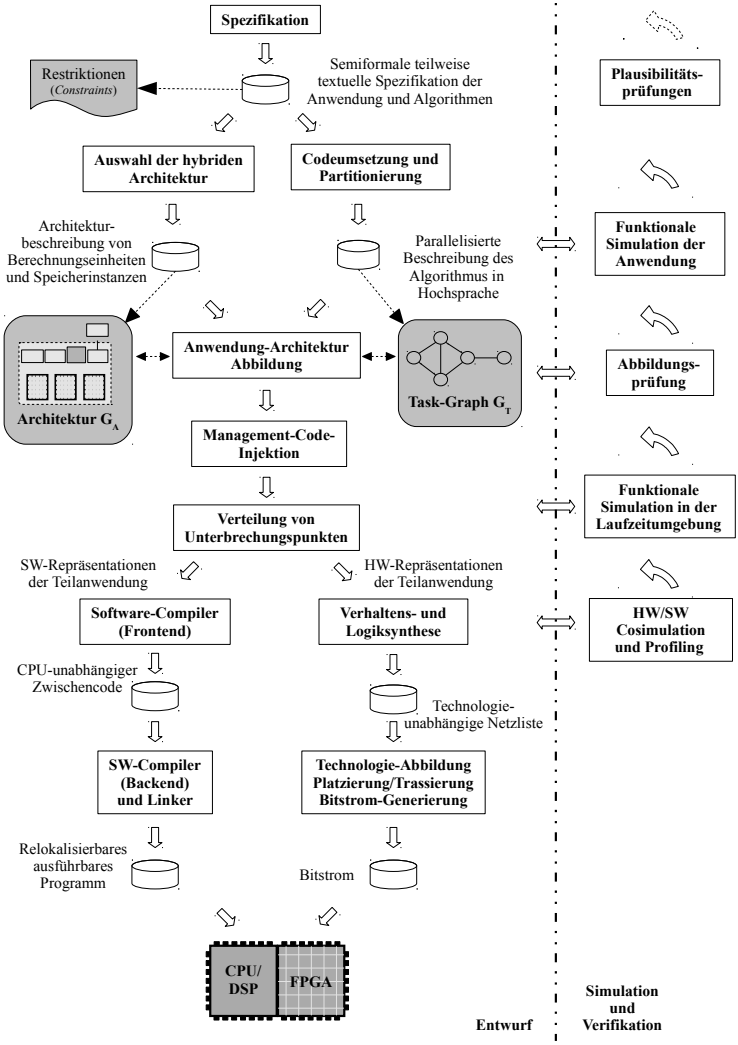


Abbildung 4.1: Entwurfsmethodik für eine dynamisch verteilbare Anwendung auf einem heterogenen System

4.1 Spezifikation

Den ersten Schritt im Entwurfsprozess bildet die Spezifikation, bei der durch den Einsatz von textueller und bildlicher Beschreibung, sowie Verweisen auf einzuhaltende Standards die Vorstellungen und Ansprüche an die Anwendung und das System dauerhaft und austauschbar festgehalten werden. Bei diesem Vorgang werden auch formalisierende Werkzeuge (z.B. *SpecScribe* [Pro+08], *Doors* [IBM12], *HP Quality Center* [Hew13], *MKS Integrity* [Ake09] oder *SpiritEd* auf Basis von *FrameMaker* [Lan+07]) eingesetzt, die Spezifikationsdaten auf Aspekte wie Plausibilität, Eindeutigkeit, Konsistenz oder die Erreichbarkeit von Zuständen prüfen können, sowie den Informationsaustausch zwischen Spezifikations- und Entwurfsingenieur verbessern. Darüber hinaus werden Referenzmodelle von Algorithmen oder Teilalgorithmen erstellt, die zur funktionalen Verifikation in folgenden Entwicklungsschritten dienen.

Das Ergebnis ist eine möglichst formale Beschreibung die neben der Funktionalität der Anwendung und deren Algorithmen auch Rahmenbedingungen an das System (z.B. maximale Leistungsaufnahme, Betriebstemperaturen oder Zuverlässigkeitsanforderungen), Schnittstellenparameter, Zeitverhalten (z.B. Reaktivität) oder Kostenparameter der Systemarchitektur [PKH10] festlegt.

4.2 Codeumsetzung und Partitionierung in eine parallelisierte Anwendung

Bei der anschließenden Partitionierung erfolgt die Dekomposition der Anwendung in parallelisierbare Teilaufgaben, entsprechend den in Kapitel 2.3 vorgestellten Verfahren als Threads oder Prozesse. Die Teilaufgaben werden dann als sequentielle Beschreibung in einer Hochsprache implementiert und in das Programmiermodell eingebunden. Dessen Kontroll-, Kommunikations- und Synchronisationskonzept wird bei diesem Entwurfsschritt umgesetzt und mit einer spezifischen Programmierschnittstelle (z.B. *pThreads* [Ins94] oder *System V* [USA91]) implementiert. Die Implementierung erfolgt dabei noch unabhängig von der späteren Ausführungsdomäne, allerdings bestehen Restriktionen, die eine Berechnungen in HW nicht erlauben bzw. erschweren. Diese sind

1. der Zugriff auf Betriebssystemaufrufe,
2. der Zugriff auf Dateisysteme,
3. Fließkommaberechnungen (durch enorm hohen Ressourcenbedarf),

4. rekursive Berechnungsvorschriften und
5. der Zugriff auf Funktionen einer Bibliothek, die ihrerseits diese Liste der Restriktionen nicht erfüllt.

Anwendungsteile, für die diese Restriktionen nicht einzuhalten sind, werden zwingend in SW ausgeführt. Diese Funktionalitäten sollten in eine eigene Teilanwendung gekapselt werden. Bei der Implementierung ist außerdem auf eine HLS optimierte Form zu achten, deren Grundlagen in [PT05; Fin10; Röß+09] eingeführt werden. Die hochsprachenbasierte Beschreibung erlaubt eine funktionale Simulation/Verifikation insofern, als dass eine Übersetzung mittels C-Compiler und Ausführung der Anwendung in einer Umgebung mit POSIX-Thread-Unterstützung auf dem Entwurfsrechner ohne Logik-Simulator möglich ist und dort gegen Referenzmodelle aus der Spezifikationsphase verifiziert werden kann.

In diesem Stadium kann die gesamte Anwendung auf einen Taskgraphen G_T abgebildet werden, wobei eine Teilaufgabe durch einen Knoten und die Abhängigkeiten im Kontrollfluss durch die Kanten repräsentiert werden. An den Knoten sind sowohl die Constraints bezüglich Ausführungsdomäne, als auch genutzte Ressourcen, wie zum Beispiel Speicher oder Mutex und Ausführungsprioritäten annotiert.

4.3 Architektur-Auswahl

Die Festlegung einer konkreten heterogenen Systemarchitektur erfolgt gleichzeitig zur Codeumsetzung und erlaubt anschließend die initiale Abbildung der Anwendung auf das System. Im ersten Schritt wird die grundlegende Entscheidung, anhand der in Kapitel 2.1.5 abgeleiteten Merkmale und Kriterien, für einen Architekturtypen (*Typen A-C*) getroffen und damit der schaltkreisbezogene Ort von Rechenwerken festgelegt. Dies bildet den Rahmen für die Gliederung des Systems auf Komponentenebene, was aus Sicht des heterogenen Systementwurfes die Festlegung von Anzahl und Umfang domänenbezogener Rechenwerke und zugehöriger Speicherressourcen darstellt. Ergebnis der Auswahl sind eine strukturelle Architekturbeschreibung aller:

- Prozessoren,
- rekonfigurierbarer HW-Bereiche,
- dem verbindenden ggf. hierarchischen Bussystem,
- Speicher sowie

- E/A-Komponenten.

Ein Beispiel für eine formal prüfbare Form einer solchen Beschreibung ist das "*Microprocessor Hardware Specification*"-Format (MHS) Xilinx Inc. [Xil08].

Aus einer solchen formalen Beschreibung lässt sich ein Architekturgraph G_A nach Definition 2.1.1 ableiten. Durch die Prüfung der Abbildbarkeit des Taskgraphen G_T der Anwendung auf G_A kann sicher gestellt werden, dass es eine gültige Implementierung (vgl. Definition 2.2.12) der Anwendung auf der gewählten Architektur geben kann. Das Ergebnis der Prüfung wirkt als Schranke für den weiteren Entwurfsprozess, d.h. weitere Entwurfsschritte erfolgen nur bei positiver Prüfung.

Mit der Architekturauswahl eng verknüpft ist die Auswahl einer geeigneten Laufzeitumgebung. Die Basis der Laufzeitumgebung bildet das Betriebssystem und eine zugehörige Thread-Bibliothek bzw. Bereitstellung geeigneter Interprozesskommunikation. Das zentrale Entscheidungsmerkmal ist die Unterstützung der spezifizierten Rechenwerke der SW-Domäne. Wird beispielsweise eine LEON3 [Dan+12] basierte symmetrische Multiprozessor-Architektur gewählt, muss eine Portierung des Betriebssystems für die SPARC-Architektur verfügbar sein, die zusätzlich die Verwaltung des SMP-Kerns erlaubt. Weiterhin muss ein Speichermanagement unterstützt werden, das den heterogen verteilten Anwendungsteilen eine einheitliche Sicht auf die Speicherarchitektur ermöglicht, die durch das Betriebssystem unterstützt wird.

4.4 Management-Code-Injektion

Das Verhalten einer Teilanwendung soll identisch sein, gleich in welcher Domäne die Ausführung erfolgt (Virtualisierung). Für den Kern der Berechnungen in der Teilanwendung erzeugen die HLS (vgl. Kapitel 2.2.4) und der SW-Compiler Implementierungen aus der algorithmischen Verhaltensbeschreibung. Abbildung 4.2 zeigt in den beiden Domänen SW bzw. HW drei Teilanwendungen und deren Verbindungen innerhalb des heterogenen Systems. Die Teilanwendungen interagieren mit globalen Ressourcen im Zuge der gegenseitigen Kommunikation (Datenaustausch oder Synchronisation) und mit der Laufzeitumgebung für Steuerungszwecke (Starten, Beenden oder Unterbrechen). Die Endpunkte der Verbindungen auf Seite der Teilanwendung unterscheiden sich zwischen den Ausführungsdomänen und sind zusätzlich abhängig vom Kontroll- und Datenfluss der Teilanwendung. Um eine einheitliche Sicht der Teilanwendung auf das System und des Systems

auf die Teilanwendung zu erhalten (Virtualisierung in AK), wird in diesem Entwurfsschritt ein domänenspezifischer Management-Code generiert und in die Implementierung der Teilanwendungen eingebracht.

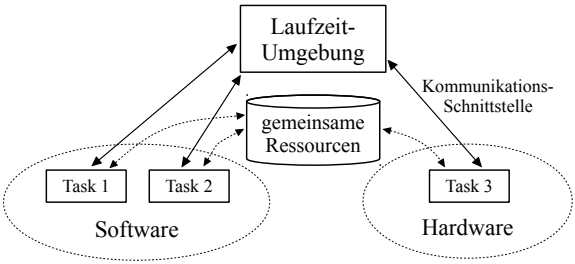


Abbildung 4.2: Tasks mit Kommunikations- und Managementverbindungen im heterogenen Gesamtsystem

Die Notwendigkeit des Management-Codes soll am Beispiel der Aquise einer Semaphore dargestellt werden. Betritt die Ausführung einer Teilanwendung einen kritischen Code-Bereich, wird ein zu Beginn des Bereiches platzierter Aufruf die Laufzeitumgebung um die Freigabe bzw. Zuweisung der Semaphore bitten. In der Verhaltensbeschreibung auf Hochsprachenebene wird dort ein Funktionsruf, z.B. *get_sema(),* stehen. Die Umsetzung des Aufrufes in der SW-Welt wird durch den Ruf einer Bibliotheks- oder Systemfunktion, z.B. *semget(),* nach dem *System V*-Standard [USA91] erfolgen, auf den die Funktion *get_sema()* durch den Linker abgebildet wird. Die erfolgreiche Referenzierung durch den Linker benötigt im Originalcode beispielsweise eine entsprechende Mantelfunktion, die eingefügt werden muss.

In der HW-Welt wird das Anfordern einer Semaphore durch eine zusätzliche kleine Komponente erfolgen, mit der die Kommunikation mit der Laufzeitumgebung über den Systembus durchgeführt wird. Trifft die HLS bei der Codeumsetzung auf die *get_sema()* Funktionen, muss eine solche Komponente in die Implementierung eingefügt werden und an den entsprechenden Stellen der Kontrollfluss in diese Komponente geleitet werden. Folglich ist diese Komponente durch Schnittstellensignale, z.B. über ein einfaches Handschlag-Verfahren, an den Hauptkontrollfluss anzubinden. Der Management-Code für die Umsetzung des *get_sema()* umfasst demnach das Handschlag-Verfahren für jeden einzelnen Aufruf und eine Instanz der Kommunikationskomponente.

Die komplexeste Aufgabe bei der Management-Code-Injektion besteht in der Realisierung von Unterbrechbarkeit und Migrationsfähigkeit für eine Teilanwendung. Die Synthese des entsprechenden Management-Codes sowie die damit verbundenen Analyse- und Optimierungsmöglichkeiten in Bezug auf eine Teilanwendung stellen einen wesentlichen der vorliegenden Arbeit dar und werden in Kapitel 5 eingehend erläutert.

4.5 Kompilierung und Synthese

Die Beschreibungen der Teilanwendungen liegen, angepasst um den spezifischen Management-Code, in einer Version für jede Ausführungsdomäne vor. Für jede Domäne folgt ein eigenständiger mehrstufiger Synthese- bzw. Codeumsetzungslauf, an dessen Ende eine ausführbare Repräsentation für die potentiellen Rechenwerke innerhalb der Domäne steht.

In der SW-Domäne wird schlussendlich Binärcode für einen Prozessor oder eine virtuelle Maschine erzeugt (vgl. Kapitel 2.1.2). Ein Compiler erzeugt in der Eingangsstufe nach lexikalischer und syntaktischer Prüfung sowie semantischer Analyse einen Zwischencode. Dieser Zwischencode ist unabhängig von der Architektur des Zielprozessors und dient als Grundlage für Programmoptimierungen [Mor97]. Die Ausgangsstufe des Compilers erzeugt daraus den ausführbaren prozessorspezifischen Objektcode. Der Linker löst extern referenzierte Aufrufe im Objektcode auf, indem der referenzierte Code entweder statisch eingebunden oder zur dynamischen Lauf- bzw. Ladezeitbindung vorgemerkt wird. Aus allen Teilanwendungen wird durch den Linker schließlich eine Bibliothek gebunden, die dann durch die Laufzeitumgebung des heterogenen Systems beim Start der Anwendung geladen wird. Sind in der SW-Domäne Prozessoren unterschiedlichen Typs vorhanden, wird für jeden Typ eine eigene Teilanwendungsbibliothek erzeugt.

Die Synthese für die HW-Domäne beginnt mit der HLS (vgl. Kapitel 2.2.4) und erzeugt aus der algorithmischen Verhaltensbeschreibung in Kombination mit dem HW-spezifischen Management-Code eine Teilanwendung auf RT-Ebene. Ausgehend von der RT-Beschreibung erfolgt die weitere Umsetzung entsprechend dem in Kapitel 2.2.2 eingeführten Entwurfsverfahren. Mittels der Logiksynthese wird die technologieunabhängige Gatternetzliste erzeugt, die anschließend auf die spezifische Logik eines Schaltkreistyps abgebildet und schließlich durch Platzierung und Trassierung auf konkrete Schaltkreisressourcen abgebildet wird. Pro Teilanwendung wird dann für jede konfigurierbare HW-Ressource ein eigener Konfigurationsbitstrom erzeugt. Für heterogene Systemarchitekturen nach Typ C (vgl. Kapitel 2.1.5), bei denen mehrere

AK innerhalb einer RPU vorgesehen sind, muss für jeden potentiellen AK ein eigener partieller Bitstrom pro Teilanwendung erzeugt werden.

Der Binärcode einer Anwendung für das heterogene System besteht folglich aus einer Kombination vielfältiger Teilanwendungsbibliotheken und Konfigurationsbitströme die als Resultat des Entwurfsablaufes erzeugt wurden. Sämtliche dieser Anwendungsteile werden zu einem eigenständigen Binärpaket zusammengefügt, einer *erweiterten Binärdatei* (engl. *FAT binary*). Die Laufzeitumgebung auf dem heterogenen System lädt diese Binärdatei bei Anwendungsstart und erhält Zugriff auf den Binärcode zur Ausführung einer Teilanwendung auf einem spezifischen Rechenwerk.

4.6 HW/SW-Co-Simulation

Die Simulation der dynamisch verteilbaren Anwendung ist die Basis für den Test und die Verifikation im gesamten Entwurfsprozess und läuft parallel zu den zuvor erläuterten Schritten ab. In Abhängigkeit der Entwurfsebene (vgl. Abbildung 2.8) erfolgt die Simulation mit zunehmendem Detailgrad in unterschiedlichen Simulatoren und Simulationsumgebungen. Weil sowohl die Anwendungsteile als auch die Laufzeitumgebung gleichzeitig ausgeführt bzw. simuliert werden, ist unterhalb der Systemebene auf struktureller Sicht eine HW/SW-Co-Simulation notwendig. Abbildung 4.3 zeigt die Simulationsumgebung auf den verschiedenen Abstraktionsebenen des Entwurfsprozesses.

Bei vorliegender Hochsprachenimplementierung, als Ergebnis der Codeumsetzung oder Implementierung der Spezifikation, erfolgt eine funktionale Überprüfung der Anwendung. Dazu wird die Anwendung in ihrer Gesamtheit durch den Hochsprachen-Compiler übersetzt und innerhalb des Betriebssystems auf dem Entwicklungsrechner (z.B. ein Linux/UNIX System) als SW-Prozess ausgeführt. Die Simulation erfolgt vollständig unabhängig vom heterogenen System und dessen Laufzeitumgebung. Das Einlesen von Stimuli und die Ausgabe der Anwendung kann beispielsweise direkt über das Dateisystem erfolgen und die parallel auszuführenden Anwendungsteile auf Prozesse oder Threads auf dem Entwicklungsrechner abgebildet werden. Auf dieser Entwurfsstufe lässt sich die Partitionierung sowie der Synchronisation und Kommunikation zwischen den Teilanwendungen verifizieren.

Im folgenden Schritt, nach Auswahl der Architektur/Laufzeitumgebung und dem Einfügen des Managements-Codes, erfolgt die Simulation innerhalb der Laufzeitumgebung des heterogenen Systems. Als Simulationsmaschine kann

ein physischer oder virtueller Rechner genutzt werden, der die Laufzeitumgebung ausführen kann. Die Rechenwerke der einzelnen Domänen sowie die gemeinsam genutzten Speicher werden virtuell dargestellt und aktiv durch die Laufzeitumgebung verwaltet. Entsprechend ist für ein Rechenwerk der HW-Domäne ein virtueller Prototyp des rekonfigurierbaren Schaltkreises [Lee+03; Sch+09] notwendig. Die Rechenwerke der SW-Domäne werden durch jeweils eigene Threads der Simulationsumgebung emuliert. Die Kommunikationskanäle bzw. Busse des Systems werden auf der Ebene einzelner Transaktionen modelliert. Das Zeitverhalten des heterogenen Systems ist durch unterschiedliche Ausführungsgeschwindigkeiten in den Domänen gekennzeichnet. Durch konfigurierbare Verzögerungen an den Unterbrechungspunkten wird dieses in grober Näherung abgebildet. Die Berechnungen erfolgen in den endgültigen Zahlenformaten, sodass die Auswirkungen (Rundung, Bereichsüberlauf etc.) der Festkommapropagierung analysiert und insbesondere über die Domänengrenze hinweg validiert werden können.

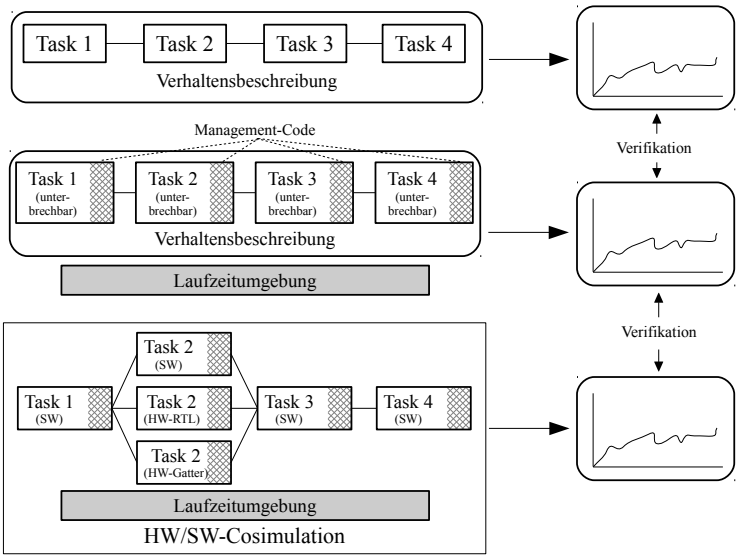


Abbildung 4.3: Simulationsumgebungen dynamisch verteilbarer Anwendungen auf einem heterogenen System

Die Simulation mit höchster Detailtreue erfolgt in einer Multidomain-Simulationsumgebung mittels Co-Simulation von HW und SW. Dazu wird an den Simulationsrechner, der die Laufzeitumgebung ausführt, ein HW-Simulator wie zum Beispiel Modelsim gekoppelt. Der virtuelle Prototyp des HW-Rechenwerks wird durch ein Modell auf RT- oder Logikebene ersetzt und es erfolgt die Simulation auf Basis der Ergebnisse von HLS bzw. Logiksynthese. Das zeitliche Verhalten der HW-Repräsentationen wird damit taktgenau nachgebildet, wodurch sich das Laufzeitverhalten der HW-Implementierung und des Management-Codes untersuchen lässt.

4.7 Eigenschafts- und Komplexitätsanalyse

Für eine Partitionierungsentscheidung des heterogenen Systems zur Laufzeit ist es notwendig, Eigenschaften der auszuführenden Teilanwendungen bezogen auf Komplexität und Ressourcenbedarf zu kennen. Soll beispielsweise ein geeignetes SW-Rechenwerk für die Ausführung einer gewissen Teilanwendung bestimmt werden, ist es nützlich den Speicherbedarf der SW-Implementierung zu kennen und ein Rechenwerk mit entsprechend freien Speicherkapazitäten zu wählen. Ein weiteres Beispiel ergibt sich bei dem Start einer Anwendung in der vielfältige neue Threads erzeugt werden. Es ist ersichtlich, dass mit Kenntnis über die Komplexität der einzelnen Threads eine optimierte Zuteilung der Ausführungsdomäne (HW oder SW) erreicht werden kann. In diesem Abschnitt werden entsprechende Metriken und Bestimmungsverfahren diskutiert.

Charakter / **Einfluss**	**Statisch**	**Dynamisch**
Software	Programmspeicher	Frei- und Stapel-Speicher
Hardware	Gatteräquivalente/ Fläche, maximale Frequenz	Verlustleistung
Hardware und Software	Komplexität	Verzögerung/ Durchsatz, erzeugte Buslast

Tabelle 4.1: Eigenschaften der Anwendungsteile und deren Einfluss auf die Systempartitionierung

Notwendig für die Partitionierungsentscheidung ist es, zusätzlich zur Laufzeit die primären Eigenschaften der einzelnen Anwendungsteile: Komplexität, Größe und Bearbeitungszeit zu kennen. Daraus können dann auch sekundäre Eigenschaften wie z.B. Leistungsaufnahme abgeschätzt werden. Grundsätzlich besitzen die Domänen jeweils eigene Metriken und entsprechende Verfahren zu deren Bestimmung. In Tabelle 4.1 werden relevante Eigenschaften gegliedert nach deren Wertstabilität und Einflussbereich dargestellt. Für die Wertstabilität wird unterschieden, wann die Eigenschaft quantifizierbar ist, statisch nach der Übersetzung oder dynamisch zur Laufzeit. Der Einflussbereich erstreckt sich über jeweils HW- bzw. SW-relevante Eigenschaften oder Metriken die sich für beide Domänen in gleicher Weise ergeben.

Die Größe des notwendigen Programmspeichers für eine Teilanwendung in SW entspricht, unter der Annahme statisch verlinkter Bibliotheken, der Größe der Programmdatei. Die genutzte Fläche bzw. Gatteräquivalente einer HW-Teilanwendung wird durch das Platzierungs- und Trassierungswerkzeug angegeben. Für die Komplexität unterscheiden sich die Metriken in SW- und HW-Domäne. Maßstäbe für die Komplexität von SW könnte beispielsweise die Schachteltiefe des Kontrollflusses sein, der durch eine statische Codeanalyse festzustellen ist. Für die HW lässt sich ein Komplexitätsmaß aus der Menge allokierter Arithmetik-Einheiten oder die Bitbreite der Operanten für Arithmetik-Berechnungen ableiten.

Die Bestimmung der dynamischen Eigenschaften erfolgt durch Online-Analyse- und Profiling-Verfahren, deren Aussagekraft von der Allgemeingültigkeit der angelegten Stimuli abhängen. Durch Analysewerkzeuge wie *gprof [GKM04]* oder *valgrind [NS07]* können die dynamischen Eigenschaften wie Durchsatz und Speicherbedarf von Teilanwendungen in der SW-Domäne auf Hochsprachenebene bestimmt werden. Vor dem Hintergrund, dass die Teilanwendungsbeschreibung den Restriktionen der HLS unterliegt und damit sowohl die dynamische Speicherallokation als auch Rekursion entfallen, ist die Komplexität der Speicherbedarfsanalyse reduziert.

Aussagen über die erzeugte Buslast lassen sich durch Simulation des heterogenen Systems auf Transaktionsebene ermitteln. Für die aussagekräftige Durchsatzmessung einer HW-Repräsentation ist eine taktgenaue Implementierung zu analysieren. Einschätzungen über die Reaktivität oder die Einhaltung geforderter Zeitschranken lassen sich vielmehr nicht durch Simulation, sondern durch Ausführung auf dem realen System treffen. Für Xilinx Schaltkreise erlaubt das Werkzeug *Chipscope* eine detaillierte Analyse des Systembuses.

5 Unterbrechung und Migration

Ein heterogenes ACS mit dynamischer Anwendungspartitionierung entscheidet zur Laufzeit darüber, auf welchem Rechenwerk ein spezifischer Anwendungsteil ausgeführt werden soll. Die dabei erzielbare Dynamik ist abhängig davon, zu welchem Zeitpunkt eine Partitionierungsentscheidung getroffen und umgesetzt werden kann. Bezogen auf eine Anwendung kann die Entscheidung jeweils zum Start einer Anwendung bzw. Teilanwendung getroffen werden und behält dann Gültigkeit bis zum Ende deren Ausführung. Alternativ dazu erzielt die Partitionierung *während der Ausführung* eine höhere Dynamik und verbessert die Flexibilität und Ausfallsicherheit des Systems (vgl. Kapitel 1.1). Dazu bedarf es der Möglichkeit eine Teilanwendung in deren Ausführung zu unterbrechen (Unterbrechbarkeit) und ggf. der Migrationsfähigkeit, um deren Ausführung zwischen den Rechenwerken verschieben zu können. Wesentliche Voraussetzungen sind es, die Zustandsinformationen (Kontext) der Teilanwendung unabhängig von der Ausführungsdomäne darzustellen, zu sichern und wieder her zu stellen. Darüber hinaus muss die Möglichkeit bestehen, an spezifischen Stellen aus dem Kontrollfluss heraus bzw. in ihn hinein zu springen.

Die existierenden Lösungen für Unterbrechung und Migration (vgl. Kapitel 3.3) innerhalb der einzelnen Domänen SW und HW sind für den domänenübergreifenden Ansatz nicht anwendbar (vgl. auch Kapitel 3.3). Zum einen lässt sich der sequentielle Kontrollfluss der SW-Domäne nicht auf den parallelen Kontrollfluss der HW-Domäne abbilden. Zum anderen ist es nicht möglich die Informationen des Ausführungsstandes (Kontext) ineinander zu wandeln. Sowohl Unterbrechungszeitpunkt als auch Zustandsdarstellung sind nicht direkt aufeinander abbildbar. Folglich bedarf es der Betrachtung auf einer höheren Abstraktionsebene, in der die Berechnung nicht einer bestimmten Ausführungsdomäne zugeschrieben ist. Zielführend erscheint die Untersuchung von Verhaltensbeschreibungen auf algorithmischer Ebene.

Die vorliegende Arbeit verfolgt dazu den Ansatz, algorithmusinvariante *Unterbrechungspunkte* (engl. Break- oder Switchpoint) einzuführen, die nach der Synthese als generische Kontrollzustände in allen Ausführungsdomänen vorliegen. Die zugehörigen Kontextinformationen werden ebenfalls auf algo-

rithmischer Ebene bestimmt und gleichfalls generisch, d.h. interpretierbar für sämtliche Ausführungsdomänen, zu den Unterbrechungspunkten annotiert. Diesen Ansatz verfolgten Kim et. al. in [KKP97; KKP06] bereits für die Umsetzung von Multitasking für VLSI[1]-Schaltkreise der HW-Domäne. Sowohl Mignolet, Nollet, Coene, Verkest, Vernalde und Lauwereins [Mig+03] als auch Koch, Haubelt, Streichert und Teich [Koc+07] beschreiben Ähnliches auch als Möglichkeit für heterogene Systeme. Im Folgenden soll dies für die algorithmische Beschreibungsebene weiterentwickelt und in Verbindung mit HLS für die dynamische Verteilung einer Anwendung auf einem heterogenen ACS unter Optimierungsgesichtspunkten formalisiert und erweitert werden.

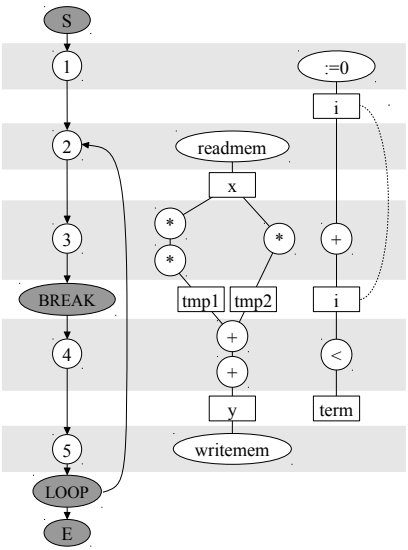
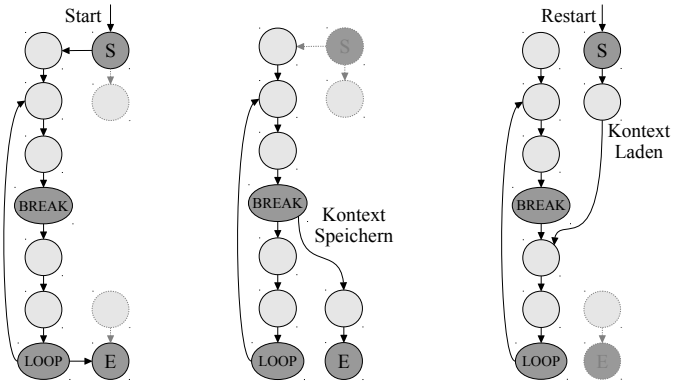


Abbildung 5.1: Kontroll- und Datenfluss mit Unterbrechungspunkt für das Beispiel 2.2.14

In Beispiel 2.2.14 auf Seite 64 wurde eine Verhaltensbeschreibung für die zyklische Berechnung von $y = Ax^2 + Bx + C$ gegeben. Die Abbildung 5.1 stellt den zugehörigen Kontroll- und Datenpfad dar. Im Kontrollgraph ist nach dem Knoten 3 beispielhaft eine Unterbrechungsmöglichkeit vorgesehen.

[1] *Very Large Scale Integration (engl.) (VLSI)*: Höchstintegrierter anwendungsspezifischer Schaltkreis



(a) Ausführungsstart (b) Unterbrechung (c) Fortsetzung

Abbildung 5.2: Modellierung von unterbrechbarem Verhalten

Bei einer unterbrochenen Ausführung der Berechnung aus Beispiel 2.2.14 auf Seite 64 ergibt sich der in Abbildung 5.2 dargestellte Kontrollfluss. Die Bearbeitung beginnt im Hierarchieknoten S. Dort wird geprüft, ob es sich um eine Erstausführung (vgl. Abbildung 5.2a) handelt und der originäre Kontrollfluss beibehalten wird oder die Fortsetzung einer zuvor unterbrochenen Ausführung (vgl. Abbildung 5.2c) vorliegt. In diesem Fall wird zunächst der alte Kontext geladen, um dann den Kontrollfluss nach dem Unterbrechungspunkt weiterzuführen. Erreicht die Ausführung einen Hierarchieknoten *BREAK* wird auf eine aktive Unterbrechungsanfrage geprüft. Liegt diese vor, wird der Berechnungspfad verlassen (vgl. Abbildung 5.2b), der Kontext gesichert und die Ausführung am Hierarchieknoten E beendet.

5.1 Modellbildung und Formalisierung

Als Basis für die weiteren Ausführungen soll dieser Ansatz formalisiert beschrieben werden. Ausgangspunkt dafür ist die in Kapitel 2.2.4.1 eingeführte Modellierung von Verhaltensbeschreibungen mittels CDFG als Sequenzgraph $G(V, E)$ nach Definition 2.2.3. Dabei bildet $v \in V$ eine Menge Knoten für

Operationen und Charakterisierungen der Hierarchie. Die Menge der Kanten $e \in E$ repräsentiert die Kontroll- und Datenabhängigkeiten zwischen den Knoten. Die Definition 2.2.3 wird um einen zusätzlichen Hierarchieknoten des Typs BREAK erweitert, an dem der Kontrollfluss gestoppt und wieder aufgenommen werden kann. Die Knoten $v_b \in BP$ bilden damit eine Teilmenge der Knoten $BP \subseteq V$ des Graphen $G(V, E)$, die einen Unterbrechungspunkt darstellen.

Ein *Pfad* $p = (v_1, v_2, v_3, ..., v_k)$ durch den Sequenzgraph bildet eine geordnete Folge von Hierarchieknoten, beginnend von $v_1$und endend in v_k. $(v_i, v_{i+1}) \in E$ sind dabei gültige Übergänge zwischen den Hierarchieknoten der Folge. Die Menge aller gültigen Pfade im Sequenzgraph ist mit P bezeichnet. Im Weiteren bildet $S = \{s_1, s_2, s_3, ..., s_n\}$ die Menge der sich aus den Operationsknoten ergebenden Speicher und die Menge $L \subseteq V \times S$ die Beziehung zwischen einem Hierarchieknoten und zwischenzeitlich dort gelesenen Speicherwerten. Entsprechend bildet $W \subseteq V \times S$ die Relation der Hierachieknoten mit Bezug zu schreibendem Speicherzugriff. Für Speicherwerte, die mit $(v_k, s) \in L \bigcap W$ in einem Zustand als gelesen und geschrieben zugeordnet werden, wird der lesende Zugriff zeitlich stets vor dem Schreibenden angenommen. Weiter wird die Annahme getroffen, das jedem Lesezugriff ein Schreibzugriff auf den Speicherwert in Zusammenhang mit einem vorgelagerten Kontrollzustand vorausgegangen ist. Schließlich ist jedem Speicher $s \in S$ die Eigenschaft $bitbreite : S \to \mathbb{N}$ zugeordnet, um den Aufwand für dessen Sicherung zu berechnen.

Formal stellt der Kontext nach Vorschrift 5.1 eine Abbildung der Hierarchieknoten von $G(V, E)$ auf den Raum aller Datenspeicher $s \in S$ dar.

$$context : V \to 2^S \tag{5.1}$$

Die Berechnungsvorschrift 5.2 ermittelt den Kontext für einen dedizierten Knoten und erfasst dazu die Datenspeicher, denen ein Wert zugewiesen wurde, der zu einem späteren Zeitpunkt der Ausführung benötigt werden wird. Dabei ist $p = (vpv')$ ein Pfad, der in v beginnt, in v' endet und dazwischen ein Teilpfad p mit keinem oder beliebig vielen Teilschritten liegt. Die Mengen L und W enthalten alle Tubel aus Datenspeichern S und Operationsknoten in V, bei denen Lese- beziehungsweise Schreibzugriff erfolgt.

$$context(v) = \left\{ s \in S | \exists (vpv') \in P : (v', s) \in L \wedge \bigwedge_{P}^{v_i} (v_i, s) \notin W \right\} \tag{5.2}$$

Demnach ist ein Datenspeicher s Teil des Kontextes an einem Knoten im Kontrollpfad, wenn ein Pfad vom Knoten v zu einem beliebigen Knoten v' existiert, in dem s gelesen wird. Auf diesem Pfad dürfen keine schreibenden Zugriffe auf den Datenspeicher erfolgen. Wenn im Knoten v selbst ein Schreibzugriff erfolgt, dann gehört der Datenspeicher s zum Kontext von v. Das ist der Fall, weil der Schreibzugriff in v bereits stattgefunden hat, wenn eine mögliche Unterbrechung in dem v folgenden *BREAK*-Knoten detektiert werden würde. Andererseits gehört s nicht zum Kontext eines Knotens v, wenn durch v der letzte lesende Zugriff auf den Datenspeicher im Kontrollfluss erfolgt. Der Wert von s wäre im Falle der Unterbrechung bereits konsumiert.

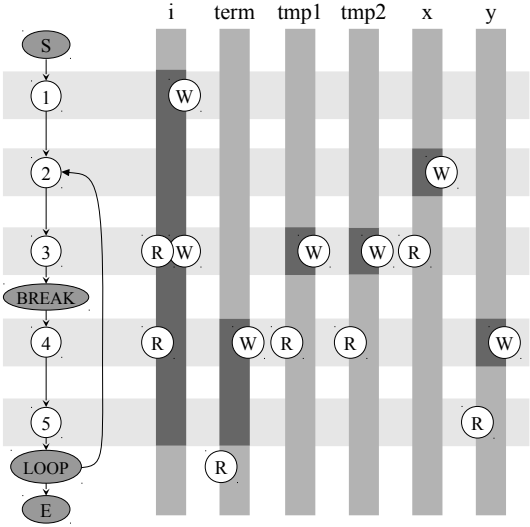


Abbildung 5.3: Lebenszeit der Datenspeicher im Datenpfad für das Beispiel 2.2.14

Die Abbildung 5.3 illustriert die *Lebenszeit* der Datenspeicher für das Beispiel 2.2.14 auf Seite 64. Die Spalten zeigen die Datenspeicher *i*, *term*, *tmp1*, *tmp2* sowie x und y des Datenpfades. Die waagerechten Balken zeigen den Bezug der Datenspeicher zum Kontrollpfad. Die Aktivität jedes Datenspeichers nach Definition 5.2 ist durch dunkelgrau hinterlegte senkrechte Markierungen in den Spalten dargestellt. Entsprechend des waagerechten Bezugs ergibt sich der Kontext für einen Knoten im Kontrollpfad auf den aktiv markier-

ten Datenspeicher. Für den Unterbrechungspunkt nach Knoten 3 sind das *i*, *tmp1* und *tmp2*.

Für das Beispiel lassen sich die Relationen L und W aller Lese- und Schreibzugriffe ableiten:

$$
\begin{aligned}
S &= \{i, term, tmp1, tmp2, x, y\} \\
W &= \{(s_1, i), (s_2, x), (s_3, i), (s_3, tmp1), (s_3, tmp2), (s_4, term), (s_4, y)\} \\
L &= \{(s_3, i), (s_3, x), (s_4, i), (s_4, tmp1), (s_4, tmp2), (s_5, y), (s_5, term)\} \quad .
\end{aligned}
$$

Es stellt sich die Frage nach einem Algorithmus zur generischen Ermittlung des Kontextes für die Knoten des Sequenzgraphen. Dieses Problem existiert in ähnlicher Form bei der Lebenszeit-Analyse von Variablen in SW-Compilern [Mor97], die zur Erkennung von ungenutzten Codefragmenten und bei der Registerallokation eingesetzt wird. Zur Problemlösung wird im Allgemeinen die rückwärts gerichtete Datenflussanalyse eingesetzt.

Algorithmus 5.1 Rückwärtige Erreichbarkeitsanalyse zur Bestimmung des Kontexts für Knoten in einem Sequenzgraph $G(V, E)$

```
 1: for all s ∈ S do
 2:   Q ← {v ∈ V | (v, s) ∈ R}
 3:   V ← Q
 4:   while Q ≠ ∅ do
 5:     x ← choosefromQ
 6:     Q ← Q \ {x}
 7:     for all y ∈ V | (y, x) ∈ E ∧ y ∉ S do
 8:       V ← V ∪ {y}
 9:       context(y) ← context(y) ∪ {s}
10:       if (y, v) ∉ W then
11:         Q ← Q ∪ {y}
```

Im vorliegenden Fall stellt sich die Kontextfrage im Bezug auf den Kontrollfluss. Deshalb wird die in Algorithmus 5.1 abgebildete Lösung als rückwärtige Erreichbarkeitsanalyse auf den Kontrollpfaden des Sequenzgraphes vorgeschlagen. Darin werden für jeden Datenspeicher s, ausgehend von allen Operationsknoten mit lesendem Zugriff, die rückwärtig erreichbaren Knoten bestimmt. Den dabei passierten Knoten wird der Datenspeicher als dem Kontext zugehörig markiert, bis entweder ein Operationsknoten mit Schreib-

zugriff oder der Startknoten passiert wird. Die Größe des Kontextes an einem Knoten v des Problemgraphen $G(V, E)$ ergibt sich dann nach Gleichung 5.3.

$$csize(v) = \sum_{context(v)}^{s} bitbreite(s) \tag{5.3}$$

5.2 Kosten für Unterbrechung und Migration

Die Unterbrechbarkeit und Migrationsfähigkeit von Teilanwendungen ist mit zusätzlichen Aufwänden, sprich Kosten, verbunden. Diese ergeben sich aus zusätzlich eingesetzten Ressourcen (zusätzliche HW-Strukturen) und Performanz-Verlusten (zusätzliche zeitliche Verzögerungen bei der Ausführung). Ermitteln lassen sich die Kosten durch den Vergleich zwischen dynamisch verteilbarer und statisch verteilter Anwendung, zuzüglich systemischer Aufwände. In diesem Abschnitt sollen die Kosten gegliedert und die einzelnen Abhängigkeitsfaktoren in einer Kostenfunktion gebündelt werden. Die Kostenfunktion dient später als Basis für die Optimierungsschritte der nachfolgenden Kapitel.

5.2.1 Kostenbetrachtungen zusätzlicher Hardwarestrukturen

Die Kostenbetrachtung muss aus drei verschiedenen Perspektiven erfolgen und die nachfolgenden Fragen stellen.

- Welche Kosten entstehen aus Systemsicht?
- Welche Aufwände ergeben sich für die unterbrechbare HW-Implementierung einer Teilanwendung?
- Welche Aufwände ergeben sich für die unterbrechbare SW-Implementierung einer Teilanwendung?

In Abbildung 5.4 sind die Erweiterungen der HW-Strukturen auf System und Modulebene angedeutet. Auf Systemebene wird ein globaler Speicher für die Aufbewahrung des Kontexts unterbrochener Teilanwendungen benötigt. Ein Transfer-Agent ist verantwortlich für den autonomen und direkten Transport der Kontext-Daten zwischen globalem und lokalem Kontext-Speicher. Für die HW-Implementierung einer Teilanwendung sind ein Zustandsautomat für das Kontext-Management, ein lokaler Kontext-Speicher sowie Datenspeicher für

kontextrelevante Werte im Operations- und Steuerwerk notwendig. Zur Sicherung und Rekonstruktion benötigt das Kontext-Management Zugriff auf diese Register. Für eine entsprechende SW-Implementierung werden keine Ressourcenaufwände hinsichtlich zusätzlicher HW-Strukturen veranschlagt. Es wird davon ausgegangen, dass die eingesetzten Prozessoren alle notwendigen Strukturen bereit stellen.

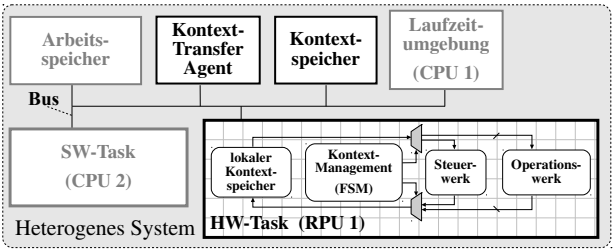


Abbildung 5.4: Zusätzliche HW-Strukturen zur Unterstützung von Unterbrechung und Migration bezogen auf System- und Modulebene

Es lassen sich vier Einflussfaktoren für die Kosten der zusätzlichen HW-Strukturen für eine gegebene Unterbrechungspunkt-Menge BP zu einem Problemgraphen $G(V, E)$ ableiten:

- Die Anzahl der Unterbrechungspunkte, denn mit jedem Unterbrechungspunkt ist ein statischer Mehraufwand verbunden. Beispielsweise werden im Steuerwerk zusätzliche Zustandsübergänge für jeden Unterbrechungspunkt benötigt.

$$r_{cost_1}(BP) = |BP| \tag{5.4}$$

- Die kumulative Größe, d.h. die Gesamtanzahl der Bits, der kontextrelevanten Datenspeicher, die im Falle irgendeines Unterbrechungspunktes gesichert bzw. wiederhergestellt werden müssen, zuzüglich der Statusinformationen des Kontrollpfads $state(BP)$.

$$r_{cost_2}(BP) = \sum_{S_{BP}}^{s} bitbreite(s) \tag{5.5}$$

mit

$$S_{BP} = \bigcup_{BP}^{v} context(v) + state(BP) \tag{5.6}$$

- Die Maximalgröße der Kontextinformationen über alle Unterbrechungspunkte einer Teilanwendung bestimmen beispielsweise die Größe des globalen Kontext-Speichers.

$$r_{cost_3}(BP) = \max_{S_{BP}}^{s}\{bitbreite(s)\} \tag{5.7}$$

- Die Anzahl der kontextrelevanten Datenspeicher, die einem Unterbrechungspunkt zugeordnet sind, summiert über alle Unterbrechungspunkte, bestimmen einen Teil der HW-Kosten, zum Beispiel die Größe des Kontext-ROM.

$$r_{cost_4}(BP) = \sum_{BP}^{v} csize(v) \tag{5.8}$$

$$R(BP) = w_1 \cdot r_{cost_1}(BP) + w_2 \cdot r_{cost_2}(BP) + w_3 \cdot r_{cost_3}(BP) + w_4 \cdot r_{cost_4}(BP) \tag{5.9}$$

Für die HW-Ressourcen R ergibt sich nach Gleichung 5.9 eine Kostenfunktion mit gewichteten Einzelfaktoren. Die Gewichte w sind abhängig von der Architektur des heterogenen Systems (vgl. Kapitel 2.1.5) und der Methode in der Unterbrechungspunkte in die Teilanwendung integriert werden. In Kapitel 8.4 werden die Faktoren beispielhaft für eine mögliche Implementierung eines HW-Rechenwerks bestimmt.

5.2.2 Zeitliche Aufwände einer Unterbrechung

Nach Teich [Tei97] kann das Maximum sämtlicher Verzögerungen, die sich durch eine Ausführungsunterbrechung ergeben, in der *Dispatchlatenz* L_D zusammengefasst werden. Es handelt sich dabei um einen dynamischen Kostenfaktor, der teilweise von Laufzeiteigenschaften des Systems bestimmt wird. L_D ergibt sich aus der längsten Zeitspanne, die sich zwischen der Unterbrechungsanforderung durch die Laufzeitumgebung bis zur Vollzugsmeldung durch die Teilanwendung ergeben kann. Bei diesem Vorgang lassen sich vier Phasen unterscheiden, die in Abbildung 5.5 dargestellt sind. Dazu zählt die

Propagierungszeit t_P als die Zeitdauer von Abgabe der Unterbrechungsanfrage durch die Laufzeitumgebung bis zum Anliegen des entsprechenden Signals an der Steuerlogik der Teilanwendung. Die anschließende Verzögerung t_U umfasst die Zeit bis das Unterbrechungssignal durch die Steuerlogik erkannt bzw. geprüft wird. Der Zeitraum, in dem die folgende Kontextsicherung stattfindet, wird durch t_S erfasst. t_{RP} gibt schließlich den Zeitbedarf für die Rückpropagierung der Bestätigung an. Die Dispatchlatenz tritt in dieser Form sowohl für die SW- als auch für die HW-Implementierung auf, wird sich jedoch in Abhängigkeit der konkreten heterogenen Computerarchitektur in der Größenordnung unterscheiden.

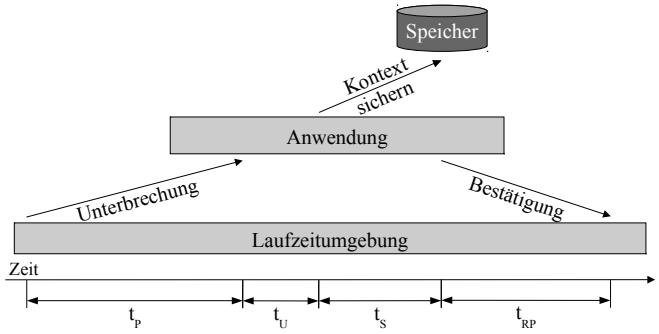


Abbildung 5.5: Zusammensetzung der Dispatchlatenz L_D

Die Anteile der Dispatchlatenz lassen sich in zwei Gruppen einteilen. t_P und t_{RP} sind ausschließlich abhängig von der Kommunikationsarchitektur des Systems. Eine direkte Verbindung zwischen Laufzeitumgebung und Teilanwendung verursacht Verzögerungen durch Signallaufzeiten im Bereich von Nanosekunden. Erfolgt die Kommunikation hingegen über mehrfach genutzte Medien (z.B. ein Bus oder ein gemeinsam genutzter Speicher) können Zugriffsverzögerungen in größeren Dimensionen auftreten, die abhängig von der Systemauslastung sind. Verwaltet die Laufzeitumgebung mehrere Teilanwendungen simultan oder teilt sich die Laufzeitumgebung die CPU mit anderen Anwendungen, entfällt auf t_{RP} ein zusätzlicher Anteil für die Dauer bis zur Reaktion auf die Unterbrechungsbestätigung. Diese ergibt sich beispielsweise durch Umschalten in die entsprechende Service-Routine.

t_U und t_S bilden die zweite Gruppe und sind ausschließlich von der Teilanwendung selbst abhängig. t_U ergibt sich aus der Anzahl der Kontrollkno-

ten, die zwischen dem Kontrollknoten, in dem eine Unterbrechungsanfrage eintrifft, und dem nächsten Unterbrechungsknoten liegen. Beispielhaft sollen zwei Extremfälle ausgeführt werden. Ist in einer unbestimmten Schleife des Kontrollflusses keine Unterbrechungsmöglichkeit vorgesehen, muss t_U mit unendlich angenommen werden. Befindet sich hingegen in jedem Kontrollzustand eine Unterbrechungsprüfung, ergibt sich t_U zur Taktfrequenz einer HW-Implementierung.

Die Dauer der Kontextsicherung t_S ergibt sich in Abhängigkeit der Lage eines Unterbrechungspunktes bezogen auf den Datenpfad nach Gleichung 5.10. Einfluss hat die Datengröße der Werte s im Kontext am Unterbrechungspunkt und die Bandbreite R_b mit den Kontextdaten, die in einen unabhängigen Kontext-Speicher übertragen werden können.

$$t_s(BP) = \frac{1}{R_b} \sum_{S_{BP}}^{s} bitbreite(s) \tag{5.10}$$

Die Größe des Kontextes ergibt sich über die Menge der Datenspeicher und Statusinformationen S_{BP} an einem Kontrollzustand entsprechend Gleichung 5.6. Formal ist die Dispatchlatenz einer Teilanwendung $G(V, E)$ mit den Unterbrechungspunkten $v_b \in BP$ damit wie folgt zu beschreiben:

$$L_D(BP) = max\{t_P() + t_{RP}()\} + \max_{BP}^{v_b}\{t_U(v_b) + t_S(v_b)\}. \tag{5.11}$$

Es zeigt sich damit, dass die Verteilung der Unterbrechungspunkte im Kontrollpfad einen entscheidenden Einfluss auf die Performanz und den Ressourcenbedarf hat und ein mehrdimensionales Optimierungsproblem aufspannt. Die folgenden Kapitel behandeln Methoden zur automatischen Verteilung der Unterbrechungsknoten und bieten Lösungen des Optimierungsproblems. Im Anschluss daran werden für Anwendungsbeispiele die zeitlichen Aufwände für das Einfügen von Unterbrechungspunkten untersucht und dargestellt.

5.3 Verteilung von Unterbrechungspunkten

Die Platzierung der Unterbrechungspunkte im Kontrollfluss der Anwendung stellt einen Schritt beim Entwurf einer dynamisch verteilbaren Anwendung dar und sollte möglichst automatisiert erfolgen. Abbildung 5.6 zeigt, ausgehend von einer Verhaltensbeschreibung, drei theoretisch mögliche Verfahren

für die synthetische Integration von Unterbrechungspunkten im Entwurfsablauf.

Das Einfügen von Unterbrechungspunkten direkt nach der Spezifikation (Variante 1) ist eine Möglichkeit. Die Unterbrechungslogik wird in der Hochsprache an ausgewählte Stellen des Kontrollflusses annotiert und danach gemeinsam mit der Teilanwendungsspezifikation mittels HLS bzw. SW-Compiler in die Repräsentation der jeweiligen Domäne überführt. Bei den Varianten 2 und 3 erfolgt die Integration auf RT-Ebene. Dazu werden die Unterbrechungspunkte in das Ergebnis der domänenspezifischen Synthese (HLS oder SW-Compiler) eingebracht. Anschließend erfolgt ein Re-Export in die Hochsprache als verhaltensbasierte Eingangsbeschreibung der jeweils anderen Synthese.

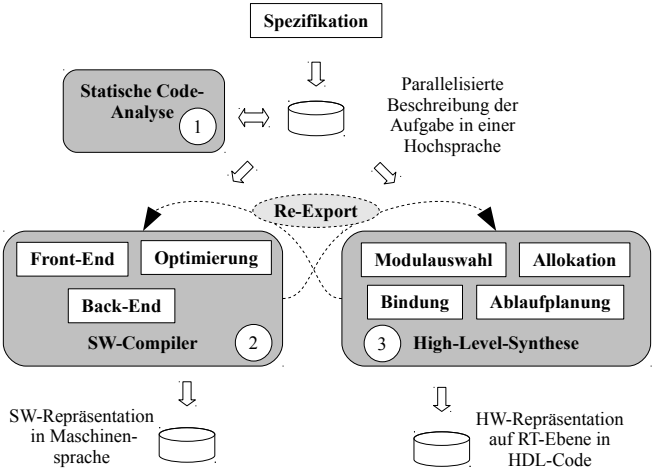


Abbildung 5.6: Möglichkeiten zur Integration generischer Unterbrechungspunkte beim Anwendungsentwurf für ein heterogenes HW/SW-System

Eine praktische Betrachtung der Varianten 2 und 3 kommt zu dem Schluss, dass Variante 2 nicht weiter verfolgt werden muss. Variante 3 bietet gegenüber Variante 2 den Vorteil einer HLS gerechten Prüfung der Eingangsspezifikation, die ausschließlich synthetisierbare Sprachkonstrukte der Hochsprache enthalten darf (vgl. Kapitel 2.2.4). Für Variante 2 müsste ein entsprechendes

Prüfwerkzeug zunächst die Einhaltung dieser Entwurfsregeln[2] sicher stellen. Mit Variante 3 zunächst die restriktivere Synthese durchzuführen erscheint darüber hinaus als sinnfällig.

In den folgenden Abschnitten werden die Verfahren nach Variante 1 und 3 dargestellt und Algorithmen für die Verteilung der Unterbrechungspunkte anhand der eingeführten Optimierungskriterien eingeführt. Für diese Erläuterungen wird von einer C-basierten Eingangsspezifikation ausgegangen, die alle in Kapitel 2.2.4 eingeführten Randbedingungen an eine HLS Umsetzung in HW erfüllt. Die grundlegenden Konzepte lassen sich in gleicher Weise auf verwandte Hochsprachen anwenden.

5.3.1 Unterbrechungspunkte auf Hochsprachenebene

Bei der Integration der Unterbrechungspunkte auf Hochsprachenebene ist es notwendig, die Unterbrechungslogik ebenfalls in der Hochsprache zu beschreiben und in geeigneter Weise mit der Eingangsspezifikation zu verbinden. Die domänenspezifischen Synthesewerkzeuge erzeugen daraus eine unterbrech- und migrierbare Teilanwendung für das heterogene System. Für die Hochsprache wird dazu ein Grundgerüst auf Basis einzelner Segmente benötigt. Jedes Segment muss einzeln angesprungen werden können und am Ende eines jeden Segmentes muss die Möglichkeit eines bedingten Sprungs zum Programmende bestehen. Der Daten- und Kontrollfluss der Eingangsspezifikation ist durch ein Analyseverfahren geeignet zu Segmentieren und auf das Grundgerüst abzubilden.

Auflistung 5.1 zeigt ein entsprechendes Gerüst auf Basis einer “durchfallenden” *switch-case*-Anweisung. Dem sequentiellen Kontrollfluss eines C-Programms folgend, würde die Ausführung in Abhängigkeit der Variablen *BrkPt* an einer *case*-Anweisung beginnen und nach Abarbeitung des *case*-Blocks in die folgende *case*-Anweisung “fallen”. Die Variable *BrkPt* dient als Eingangsparameter und trägt den Wert eines Unterbrechungspunktes an dem ggf. fortgesetzt werden soll oder den Wert *Null,* wenn keine Unterbrechung vorlag. Ein *case*-Block teilt sich in drei Abschnitte. Zu Beginn wird auf eine nötige Kontext-Wiederherstellung geprüft. Dies ist der Fall, wenn direkt von der *switch*-Anweisung her gesprungen wurde. Der Hauptteil führt das entsprechende Segment der Eingangsbeschreibung aus. Am Ende wird auf Unterbrechungsanforderung geprüft und ggf. der Kontext gespeichert und aus der gesamten *switch*-Anweisung gesprungen. Dieses Grundgerüst

[2] *Prüfung von Entwurfsregeln*: engl. Design Rule Check

lässt sich mit sämtlichen HLS-Werkzeugen (vgl. Anhang A) synthetisieren und außerdem leicht auf ein *if-then* Konstrukt abbilden.

```
preempted_BrkPt = 0;                                                    1
                                                                        2
switch(BrkPt){                                                          3
   case(0):                                                             4
      first_segment();        // of application code                   5
      if check_break(){       // are we preempted?                     6
         store_context(1);                                              7
         preempted_BrkPt = 1;                                           8
         break;                                                         9
        }                                                               10
   case(1):                                                             11
      if(BrkPt == 1)          // do we resume?                         12
         recover_context(1);                                            13
                                                                        14
      second_segment();       // of application code                   15
                                                                        16
      if check_break(){       // are we preempted?                     17
         store_context(2);                                              18
         preempted_BrkPt = 2;                                           19
         break;                                                         20
        }                                                               21
   ...                         // cont. for n-1 breakpoints             22
   case(n):                                                             23
      if(BrkPt == n)          // do we resume?                         24
         recover_context(n);                                            25
                                                                        26
      last_segment();         // of application code                   27
}                                                                       28
return(preempted_BrkPt);       // if < 1 task has completed            29
```

Auflistung 5.1 Grundgerüst einer unterbrechbaren Teilaufgabe in der Hochsprachenebene

Eine wesentliche Einschränkung ergibt sich, weil eine Segmentierung ausschließlich auf oberster Kontrollflussebene möglich ist. Komplexer Kontrollfluss (Schleifen oder Verzweigungen) ist so nicht auf das durchfallende *switch-case* Konzept abbildbar. Um den Sprung in die tieferen Kontrollflussebenen zu ermöglichen, muss das *switch-case* Konzept geschachtelt werden und auch für den Basis-Block im Schleifenkörper Anwendung finden. Darüber hinaus muss die Unterbrechungsprüfung konjunktiv in der Konditionsprüfung der Schleifen und Verzweigungen verknüpft werden.

Zur Automatisierung des Verfahrens eignen sich die Methoden der statischen Codeanalyse. Für die Analyse ist es notwendig, sämtliche Funktionsrufe der

C-Beschreibung *inline* einzubetten und keine Bibliotheksrufe zu verwenden. Umgekehrt betrachtet ist die Analyse nur für Code-Blöcke ohne externe Funktionsrufe einsetzbar. Die Eingangsbeschreibung wird zunächst auf einen hierarchischen Sequenzgraph $G(V, E)$ nach Definition 2.2.3 abgebildet. Für jeden Basis-Block der C-Beschreibung, der einen Basis-Block im Sinne der Modellierung nach Kapitel 2.2.4.1 darstellt, wird ein hierarchisch eingebundener Sequenzgraph als Teilgraph $G_{sub} \subseteq G$ mit $G_{sub}(V_{sub}, E_{sub})$ und $V_{sub} \subseteq V$ sowie $E_{sub} \subseteq E$ gebildet. Potentielle Unterbrechungspunkte sollen sich am Endknoten eines Teilgraphs befinden. Die Eingangsbeschreibung wird später in das zuvor beschriebene Grundgerüst eingefügt, indem ein Teilgraph einem *case*-Zweig zugeordnet wird. Das erfolgt in der Schachteltiefe, die der Hierarchieebene entspricht.

Die Freiheitsgrade bei der Verteilung der Unterbrechungspunkte verlangen zum einen die Entscheidung bis zu welcher Verschachtelungstiefe bzw. Hierarchietiefe unterbrochen werden soll. Zum anderen können größere Teilgraphen in kleinere zerlegt werden, um die Dichte der Unterbrechungspunkte zu erhöhen. Weil die Verteilungsentscheidung optimiert auf die Kostenparameter L_d und R aus Kapitel 5.2 getroffen werden muss, sind die verteilungsrelevanten Eigenschaften des Teilgraphen zu bestimmen.

Das ist einerseits die Kontextgröße am Endknoten, dem potentiellen Unterbrechungspunkt. Das kann nach Algorithmus 5.1 für jeden Teilgraphen durch rekursives Aufrollen der gesamten Graphenhierarchie erfolgen. Andererseits ist es die Bearbeitungsdauer $t_{ex}(G_{sub})$ des Teilgraphen, die direkten Einfluss auf den Anteil t_u der Dispatchlatenz hat. Diese kann durch Summation der Operationen im Teilgraph näherungsweise bestimmt werden, wobei der Operationstyp und die Operantenbreite die Wichtung einer Operation beeinflussen. Die Näherung bei diesem Vorgehen ergibt sich, weil die einmalige Traversierung jedes Untergraphen impliziert, dass jede Schleife, und an Verzweigungen jeder Pfad, genau einmal ausgeführt wird.

Der Algorithmus 5.2 zeigt eine Möglichkeit zur automatisierten Verteilung von Unterbrechungspunkten. Die Optimierungskriterien werden dabei insofern berücksichtigt, als dass eine obere Schranke für L_d eingehalten und dabei ein Minimum zusätzlicher HW-Ressourcen R eingesetzt wird. Der Algorithmus verfolgt das Schema, solange Teilgraphen als Unterbrechungspunkt zu markieren, wie die geforderte Obergrenze nicht eingehalten wird. Folglich werden ausgehend vom minimalen Ressourceneinsatz (kein Teilgraph ist als Unterbrechungspunkt markiert) mit jeder neuen Markierung Schritt für Schritt weitere Ressourcen gebunden. R bleibt damit während des gesamten Algorithmus auf einem Minimum.

Algorithmus 5.2 Verteilung von Unterbrechungspunkten auf der Hochsprachenebene

```
1:  for all g_sub ∈ G do
2:    Q ← get subgraphs(g)
3:    V ← ∅
4:    while Q ≠ ∅ do
5:      x ← choose fromQ
6:      Q ← Q \ {x}
7:      if L_D < latency(x) then
8:        if split(x) ≠ ∅ then
9:          Q ← Q ∪ split(x)
10:       else if get subgraphs(x) ≠ ∅ then
11:         Q ← Q ∪ get subgraphs(x)
12:       else
13:         V ← ∅
14:         Q ← ∅
15:     else
16:       V ← V ∪ {x}
```

Die Prüfung der Schranke L_d erfolgt über die Funktion *latenz*(), die für den Teilgraphen die Verzögerungen t_u und t_s ermittelt. Kann die Obergrenze nicht eingehalten werden, wird zunächst versucht den Teilgraphen in g_1 und g_2 aufzuspalten. Bezogen auf die C-Beschreibung wird ein Basis-Block damit in zwei Basis-Blöcke zerlegt. Idealerweise erfolgt die Trennung am Minimum von $context(g_1)$. Ist die Aufspaltung nicht möglich, werden die Teilgraphen der nächsttieferliegenden Hierarchieebene als Unterbrechungspunkte markiert. Existieren keine solche Teilgraphen dann kann die Schranke für L_d nicht eingehalten werden und der Algorithmus bricht ab. Die Menge $\mathcal{V}$ enthält als Ergebnis die Teilgraphen, die als Unterbrechungspunkt dienen.

Die hochsprachenbasierte Integration der Unterbrechungslogik vor der HLS ist nicht optimal. Durch die eingeschränkten Einsprungmöglichkeiten in eine Eingangsbeschreibung können Unterbrechungspunkte nur am Ende eines Basis-Blocks eingefügt werden. Erzwingt die Einhaltung der Latenzschranke eine Teilung des Teilgraphen, werden damit die Parallelisierungsoptionen auf Instruktionsebene für die folgende HLS eingeschränkt und eine suboptimale Implementierung erzeugt. Weil das Scheduling der Operationen vor der HLS nicht fest steht, ermitteln die Funktionen *latenz*() und *context*() Näherungswerte. Das Ergebnis ist daher ebenfalls suboptimal.

Weitere Nachteile ergeben sich durch die Umsetzung der Unterbrechungslogik mittels HLS. Für jeden Unterbrechungspunkt werden die Ressourcen zur Auswertung des Unterbrechungssignals und die Ressourcen zum Sichern und Wiederherstellen der Kontextinformationen (Steuerautomat und Speicherzugriffslogik) jeweils einmal erzeugt. Die Mehrfachnutzung dieser weitgehend identischen HW-Strukturen durch mehrere Unterbrechungspunkte wäre deutlich günstiger, ist jedoch durch eine Beschreibung auf Hochsprachenebene nicht realisierbar.

5.3.2 Unterbrechungspunkte auf RT-Ebene

Erfolgt die Integration von Unterbrechungspunkten auf RT-Ebene im Anschluss an die Optimierungsstufe der HLS, können die Einschränkungen des Verfahrens aus dem vorherigen Abschnitt relativiert und die einhergehenden Kosten weiter reduziert werden. Das Grundprinzip besteht darin, die von der HLS generierte Anwendungsbeschreibung auf RT-Ebene in geeignete Partitionen zu gliedern, zwischen denen unterbrochen werden kann. Aus der RT-Beschreibung werden dann die erweiterten HW- und SW-Repräsentationen mit identischen Unterbrechungspunkten erzeugt.

Ausgangspunkt für diesen Ansatz ist das in Abschnitt 2.2.4.1 eingeführte FSMD-Modell, weil sich damit nach Definition 2.2.12 Implementierungen, die durch eine HLS erzeugt wurden, formal abbilden lassen. Entsprechend kann das Modell die Ausgabe der Optimierungsstufe der HLS erfassen und zur Analyse auf geeignete Unterbrechungsstellen dienen. Bezogen auf das FSMD-Modell lässt sich ein Unterbrechungspunkt als Schnitt durch den Daten- und Steuerpfad entlang einer Taktgrenze begreifen, wobei sämtliche geschnittene Verbindungen im Datenpfad den unterbrechungspunktbezogenen Kontext bilden.

5.3.2.1 Integration und optimale Verteilung der Unterbrechungslogik mittels High-Level-Synthese

Es stellt sich die Frage, wo in einem FSMD geeignete Unterbrechungspunkte liegen, die hinsichtlich gegebener Schranken für *Dispatchlatenz* L_D und Ressourcenbedarf R ein Minimum an Kosten verursachen. Aus dieser Fragestellung ergibt sich ein mehrdimensionales Optimierungsproblem. Zur Problemformulierung werden aus der analytischen Betrachtung des FSMD Bedingungen formuliert, die sich aus den Kostenparametern nach Kapitel 5.2 ergeben.

Anschließend wird der Lösungsraum möglicher Verteilungen von Unterbrechungspunkten festgelegt und die zu optimierende Zielfunktion vorgestellt.

Für die graphentheoretische Analyse des FSMD wird das Steuerwerk auf einen Kontrollflussgraphen $G_K = (V_K, E_K)$ nach Definition 5.3.1 und das Operationswerk auf einen Datenflussgraphen $G_D = (V_D, E_D)$ nach Definition 5.3.2 abgebildet. Ausgehend von einer gültigen Implementierung der Verhaltensbeschreibung $G(V, E)$ existiert mit der Allokation α, dem Ablaufplan ϕ und der Bindung (β, γ) eine Abbildungsvorschrift $\delta : V_D \rightarrow V_K$, die für jede Operation in G_D die Zuordnung zu einem Kontrollzustand in G_K trifft. Beschreibt $S = \{s_1, s_2, s_3..., s_n\}$ die Menge aller Speicherwerte, die sich aus G_D mit δ ergeben und bilden V_K die Menge der Hierarchieknoten sowie V_D die Menge der Operationsknoten, dann kann mittels der Vorschriften 5.9 und 5.11 die Kontextermittlung für den FSMD erfolgen.

Definition 5.3.1. [Kontrollflussgraph] Ein *Kontrollflussgraph* $G_K = (V_K, E_K)$ ist ein gerichteter Graph, dessen Knotenmenge $V_K = \{v_0, v_1, v_2, ..., v_n\}$ die Zustände des Steuerwerks und die Kanten $e_i \in E_K$ mit $E_K \subseteq V_K \times V_K$ alle gültigen Übergänge zwischen den Zuständen abbildet. Mit dem Startzustand ist in V_K genau ein Knoten ohne eingehende Kante und mit dem Endzustand ein Knoten ohne ausgehende Kante enthalten.

Definition 5.3.2. [Datenflussgraph] Ein *Datenflussgraph* $G_D = (V_D, E_D)$ ist ein gerichteter Graph. Die Knotenmenge $V_D \subseteq V_K \times EXP$ umfasst alle Instruktionen einer gültigen Implementierung $(\phi, \beta, \gamma, \alpha)$ der Spezifikation $(G(V, E), G_R(V_R, E_R))$. Die Menge der Kanten $E_D \subseteq V_D \times S \times V_D$ umfasst die Datenabhängigkeiten zwischen allen Knotenpaaren $v_i, v_j \in V_D$ über den Speicherwert $s \in S$.

Die Speicherwerte $s \in S$ werden in einer HLS Implementierung in drei unterschiedlichen Varianten gehalten. Einfache Variablen und Zwischenwerte resultieren in Registern, Felder in strukturiertem Speicher und gepufferte E/A beispielsweise in einem Ringpuffer als FIFO-Speicher. Die Zusammenfassung in S ist eine vereinfachte Betrachtung, die diese Typen als identisch ansieht und letztendlich im Verhalten einer Variable gleichsetzt. Aus dieser Betrachtung resultiert beispielsweise, dass durch den lesenden Zugriff auf eine Stelle eines strukturierten Speichers der Speicher in seiner Gesamtheit zum Kontext gezählt wird und ggf. zu sichern ist.

5.3.2.2 Notwendige Unterbrechungspunkte für eine obere Schranke der Dispatchlatenz

Gesucht werden zunächst alle notwendigen Unterbrechungspunkte, die eine obere Schranke für die Dispatchlatenz garantieren können. Im Folgenden wird implizit davon ausgegangen, dass die Dispatchlatenz in Takten angegeben ist und ein Zustandsübergang genau eine Taktperiode τ benötigt. In Bezug auf den Kontrollfluss eines FSMD lassen sich folgende Bedingungen formulieren: Im Kontrollgraph darf, ausgehend von jedem Knoten, kein Pfad existieren,

1. der länger ist als die maximale Dispatchlatenz L_D, ohne das ein Unterbrechungspunkt erreicht wird, und

2. dessen Länge bis zum nächsten Unterbrechungspunkt und die taktbezogene Dauer $t_s \cdot \tau$ zum Speichern des Kontextes L_D überschreitet.

Diese Bedingungen berücksichtigen den Anteil t_u der Dispatchlatenz und implizieren, dass alle unbegrenzten Schleifen mindestens einen Unterbrechungspunkt beinhalten. Sonst könnte die Ausführung in der Schleife verbleiben, bis die Obergrenze überschritten wird. Für Schleifen mit statischer Anzahl von Durchläufen ist die Länge des Ausführungspfades bekannt und der Schleifenkörper kann als vollständig ausgerollt in den Kontrollgraph integriert werden.

In Algorithmus 5.3 wird ein Schema gezeigt, in dem die Nebenbedingungen aus dem Kontrollgraph des FSMD generiert werden können. Dazu wird über alle Kontrollzustände V_K iteriert. Für jedes $v \in V_K$ werden alle Pfade mit der maximalen Länge L_D extrahiert. Die Traversierung eines Pfades wird beendet, wenn ein Zustand wiederholt auf diesem Pfad angetroffen und damit eine Schleife detektiert wurde.

Algorithmus 5.3 Ermittlung notwendiger Zustandsfolgen, deren Unterbrechung sich als Nebenbedingungen für eine garantierte Dispatchlatenz L_D ergeben

```
1:  constraints ← ∅
2:  for all v ∈ V_K do
3:    Q ← {(v)}
4:    while Q ≠ ∅ do
5:      p ← choose from Q
6:      Q ← Q \ {p}
7:      if |p| = L_D then
8:        constraints ← constraints ∪ {purge(p)}
9:      else
10:       x ← last state in p
11:       succ ← {y ∈ V_K|(x, y) ∈ E_K}
12:       if (succ ∩ p) ≠ ∅ then
13:         constraints ← constraints ∪ {purge(p)}
14:       else
15:         for all y ∈ V_K|(x, y) ∈ E_K do
16:           Q ← Q ∪ {p + (y)}
```

Zusätzlich berücksichtigt Algorithmus 5.3 die Latenz t_s zur Speicherung des Kontextes an einem Zustand mit potentiellem Unterbrechungspunkt. Diese Latenz wird für jeden Kontrollzustand separat berechnet. Dazu wird die Bitbreite des dortigen Kontextes durch die Übertragungsrate R_b in Bit pro Taktschritt nach Gleichung 5.10 dividiert. Über die Hilfsfunktion *purge* nach Gleichung 5.12 werden die Zustände mit zu großem t_s aus dem Pfad entfernt und nicht in die Nebenbedingung übernommen. In *purge* beschreibt i die Position von Zustand v im Pfad p.

$$purge(p) = \{s_i \in p | i + latency(v) \leq L_D\} \tag{5.12}$$

Schließlich befinden sich in der Menge *constraints* die gesuchten Sätze möglicher Unterbrechungspunkte, die den Lösungsraum für das Optimierungsproblem ergeben. Durch Anwendung der Funktion *purge* stellen diese Sätze keine validen Pfade mehr dar. Die Obergrenze der maximalen Dispatchlatenz wird eingehalten, wenn in jedem Satz der Menge mindestens ein Zustand ein Unterbrechungspunkt ist.

Zur Illustration des Schemas wird das Ergebnis des Algorithmus in Abbildung 5.7 anhand einer Implementierung des Beispiels 2.2.14 auf Seite 64 mit

acht Zuständen im Kontrollpfad und einem Addierer, zwei Multiplizierern sowie einem Komparator im Datenpfad gezeigt. Der Kontrollflussgraph (mittig) zeigt den aktuell unter Analyse stehenden Zustand v_4 hervorgehoben. Ebenfalls mittig dargestellt ist die Größe des Kontexts am entsprechenden Kontrollzustand. Ziel der Analyse ist es herauszufinden, welche Folgezustände ein Unterbrechungspunkt sein müssen, wenn in Zustand v_4 eine Unterbrechungsanforderung eingeht und die Obergrenze der Dispatchlatenz eingehalten werden soll. Ausgehend von v_4 ergeben sich zwei mögliche Ausführungspfade. Einer verlässt die Schleife und kommt zum Ende der Ausführung in Zustand E, während der andere eine weitere Iteration ausführt. Im Zustand v_3 wird erkannt, dass es sich um eine Schleife handelt, weil mit v_4 der initiale Zustand ein zweites mal erreicht wird. Die beiden Pfade sind in der rechten Spalte von Abbildung 5.7 dargestellt.

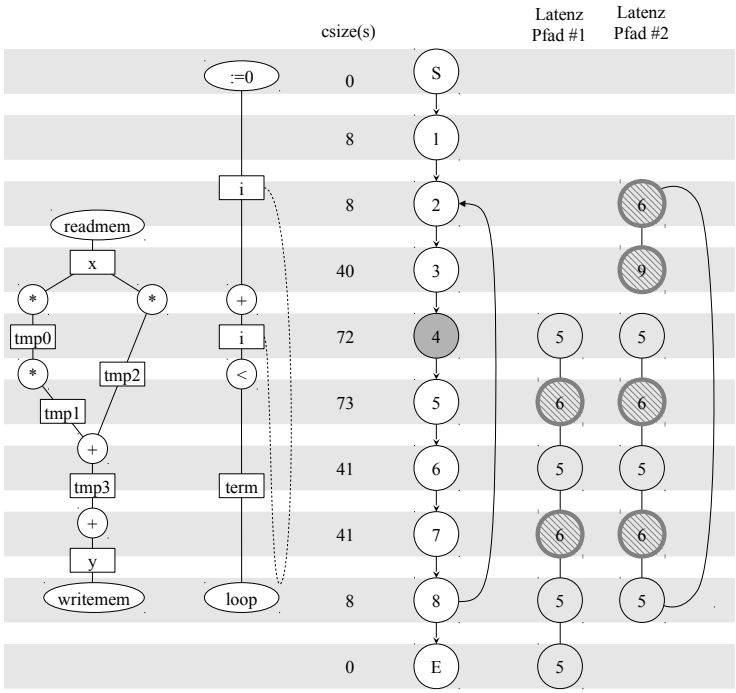


Abbildung 5.7: Zwei mögliche Ausführungspfade für Zustand v_4 in einer Implementierung von Beispiel 2.2.14 mit 8 Kontrollzuständen

Jedem Zustand sind in der linken Spalte die Größe der zugehörigen Kontextinformationen zugeordnet, wobei die Variablen eine Datenbreite von i 8 Bit, *term* ein Bit und alle anderen Variablen 32 Bit besitzen. Für die Übertragungsrate R_b wurde von 16 Bit pro Takt ausgegangen. Die maximale Latenz von 5 Takten sollte garantiert werden. Alle Zustände deren Latenz $max\{t_u(v) + t_s(v)\}$ die Anforderung von 5 Takten überschreiten, sind mit einem grau schraffierten Grund hinterlegt. Für die Analyse von Zustand v_4 ergibt sich die resultierende Menge Zustandssätze zu

$$constraints = \{..., \{v_4, v_6, v_8, v_E\}, \{v_4, v_6, v_8\}, ...\}.$$

Die erste Randbedingung ist ein Superset der Zweiten, so kann das Ergebnis als boolesche Nebenbedingung wie folgt erfasst werden

$$(v_4 \in BP) \bigvee (v_6 \in BP) \bigvee (v_8 \in BP).$$

5.3.2.3 Lösung des Optimierungsproblems

Die nachfolgend skizzierte Lösung ist das Ergebnis einer gemeinsamen Arbeit mit Jan Langer und wurde in [RLH12] veröffentlicht. Aus dem prinzipiellen Lösungsraum, alle Kontrollpunkte können ein möglicher Unterbrechungspunkte sein, ist eine Auswahl zu bestimmen, die minimale Kosten verursacht und den zuvor aufgestellten Constraints genügt. Einen Ansatz dieses Entscheidungsproblem zu lösen ist dessen Transformation in ein pseudo-boolesches 0/1-ILP-Optimierungsproblem. Für diese Klasse existieren dann 0/1-IPL- Problemlöser[3] [SS05; MR06] zur Bestimmung des Optimums. Der Auflöser erwartet eine Zielfunktion der Form $\overline{c}^T \overline{x}$ deren Ergebnis hinsichtlich des Maximums oder Minimums zu optimieren ist.

Für jeden Kontrollzustand $v \in V_k$ wird eine entsprechende boolesche Variable x_v eingeführt. Besitzt die Variable den Wert 0 ist der Kontrollzustand kein Unterbrechungspunkt. Der Wert 1 deutet entsprechend auf eine Unterbrechungsmöglichkeit in v hin.

$$x_v = \begin{cases} 1, & \text{wenn } v \in BP \\ 0, & \text{sonst} \end{cases} \tag{5.13}$$

[3] *Problemlöser*: solver (engl.)

Analog dazu wird für jeden Speicherwert $s \in S$ eine boolesche Variable x_s definiert, sodass deren Wert eine Aussage über die Verwendung der Variable in einem der Unterbrechungspunkte trifft.

$$x_s = \begin{cases} 1, & \exists\, v \in BP : s \in context(v) \\ 0, & \text{sonst} \end{cases} \tag{5.14}$$

Die booleschen Variablen nach Gleichungen 5.13 und 5.14 werden in einem Vektor $\overline{x} = \{x_{v_1}, ..., x_{v_n}, x_{s_1}, ..., x_{s_m}\}$ zusammengefasst. Die Gewichte $\overline{c} = \{c_{v_1}, ..., c_{v_n}, c_{s_1}, ..., c_{s_m}\}$ sind Konstanten, die sich direkt aus den Kostenfunktionen in Kapitel 5.2 ableiten und die linear mit den im Lösungsraum gekoppelten Kostenanteilen in die Optimierung einbezogen werden. Die Kostenfunktion 5.7 zur Bewertung des maximal notwendigen Kontextspeichers, kann folglich in diesem Ansatz nicht berücksichtigt werden. Aus den Gleichungen 5.4, 5.5 und 5.8 lassen sich die Gewichte ermitteln, diese betragen für alle Kontrollzustände

$$c_v = w_1 + w_4 \cdot csize(v)$$

und für die Speicherwerte

$$c_s = w_2 \cdot bitbreite(s).$$

Betrachtet man die so aufgestellte Zielfunktion, sucht der Optimierungsalgorithmus eine boolesche Belegung, d.h. eine Auswahl an Kontrollzuständen und Speicherwerten, deren akkumulierte HW-Kosten minimal sein sollen. Ohne weitere Bedingungen wäre das der Fall, wenn kein Zustand und kein Speicherwert, also alle Variablen x auf 0 gesetzt würden. Das ist noch nicht die gesuchte Lösung.

Vielmehr muss die Optimierung unter Berücksichtigung eines Satzes von Constraints erfolgen, die dem Auflösungsalgorithmus in einem Vektor von Ungleichungen der Form von Gleichungen 5.15 bereit gestellt werden.

$$\sum c_i x_i \geq k, c, k \in \mathbb{Z}^+, x_i \in \{0, 1\} \tag{5.15}$$

Ein Teil der Constraints ergibt sich aus der Verknüpfung zwischen den Kontrollzuständen und den Speicherwerten. Deren Zusammenhang ist eine Implikation, da ein bestimmter Kontrollzustand $v \in V_K$ die zum Kontext gehörigen Speicherwerte $context(v)$ impliziert.

Die logische Implikation ist in eine ILP Bedingung der Form 5.15 transformiert durch:

$$\bigwedge_{V}^{v}(x_v \rightarrow \bigwedge_{context(v)}^{s} x_s) = \bigwedge_{S}^{s}(x_s \vee \bigwedge_{V_s}^{v} \neg x_v) = \bigwedge_{S}^{s}(|V_s| \cdot x_s - \sum_{V_s}^{v} x_v \geq 0)$$

mit $V_s = \{v \in V | s \in context(v)\}$.

Die Constraints, die sich aus der Latenzschranke ergeben, sind der zweite Teil. Auf die gleiche Weise können diese nach Gleichung 5.16 in ILP Bedingungen gewandelt werden. Dabei werden sich gegenseitig einschließende Bedingungen entfernt.

$$\bigwedge_{constraints}^{c} \bigwedge_{c}^{v} x_v = \bigwedge_{constraints}^{c} (\sum_{c}^{v} x_v \geq 1) \tag{5.16}$$

Die Ausführung des 0/1-ILP-Auflösers ermittelt schließlich eine Belegung des Vektors $\overline{x}$, die zwar nicht notwendigerweise optimal ist, jedoch einen gültigen Satz aus Unterbrechungspunkten im Kontrollflussgraphen und aus zum Kontext zählenden Speicherwerten erzeugt.

5.3.2.4 Implementierung der Teilanwendungen in Hardware und Software

Der mit Unterbrechungspunkten annotierte FSMD ist auf die heterogenen Ressourcen des Systems abzubilden. Neben der logischen Funktion zur Berechnung der Kernanwendung sind dabei auch die Funktionalitäten für das Prüfen von Unterbrechungsanfragen, sowie das Speichern und Laden der Kontextvariablen zu realisieren. Für die jeweilige Ausführungsdomäne muss das auf unterschiedliche Weise erfolgen, da deren Prozessierung fundamentale Unterschiede aufweist (vgl. Kapitel 2.1.1).

Algorithmus 5.4 Reduzieren des Zustandsraumes im CFG für die SW-Implementierung

```
1:  combinable ← ∅
2:  for all v ∈ V_K do
3:    Q ← {(v)}
4:    while Q ≠ ∅ do
5:      p ← choose from Q
6:      Q ← Q \ {p}
7:      x ← last state in p
8:      succ ← {y ∈ V_K|(x, y) ∈ E_K}
9:      if (|succ| > 1) or (({x} ∩ BP) ≠ ∅) then
10:       combinable ← combinable ∪ {p}
11:     else
12:       z ← choose from succ
13:       pred ← {y ∈ V_K|(y, z) ∈ E_K}
14:       if (|pred| > 1) then
15:         combinable ← combinable ∪ {p}
16:       else
17:         Q ← Q ∪ {p ∪ succ}
```

Naheliegend ist die Umsetzung in HW durch eine synchron getaktete Schaltung, für die aus dem CFG der Kontrollpfad und aus dem DFG der Datenpfad erzeugt werden. Den Kontrollpfad realisiert ein Automat (Finite State Machine, FSM), dessen Automatenzustände den Knoten des CFG entsprechen. Die Zustandsübergänge werden durch die Kanten des CFG festgelegt. Zusätzlich werden bedingte Übergänge an den Zuständen eingeführt, die als Unterbrechungspunkt ausgewählt wurden. Im Falle einer Unterbrechungsanfrage erfolgt der Übergang in einen Teilautomaten, der die Kontextspeicherung und das Beenden der Ausführung realisiert. Der Datenpfad wird vom DFG ausgehend aus kombinatorischer Logik und Registerstufen für jeden Zustand im CFG erzeugt. Register, die kontextrelevante Variablen halten, werden mit der Fähigkeit ausgestattet, die Registerwerte im Falle einer Wiederaufnahme vorzuladen und bei einer Unterbrechung über einen zweiten separaten Ausgangsport auszulesen.

Für die SW-Implementierung werden CFG und DFG auf ein C-Template abgebildet. Dadurch wird eine sequentielle Folge der Operationen erzeugt und deren Ausführungsreihenfolge wieder festgeschrieben. Für die SW-Implementierung muss daher nur ein reduzierter Teil des Zustandsraumes im CFG abgebildet werden, um die korrekte Berechnungsvorschrift beizu-

behalten. Es sind ausschließlich die Zustände notwendig, die entweder den Kontrollfluss bestimmen (Verzweigungen oder Schleifen) oder die Unterbrechung und Wiederherstellung realisieren. Die restlichen Zustände können mit deren Vorgänger kombiniert werden. Die assoziierten Operationen im DFG werden mit dem entsprechenden neuen Zustand im CFG verknüpft.

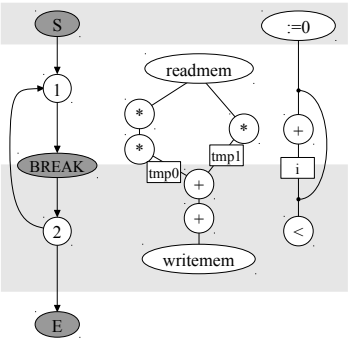


Abbildung 5.8: Unterbrechbare SW-Implementierung für das Beispiel 2.2.14

```
state = check_resume();                    1
while(state!=E){                           2
 switch(state){                            3
 case S:                                   4
   i = 0;                                  5
   break;                                  6
 case s1:                                  7
   x=read(input_x);                        8
   tmp0 = x * x * 150;                     9
   tmp1 = x * 130;                         10
   i++;                                    11
   check_irpt();                           12
   break;                                  13
 case s2:                                  14
   y=tmp0 + tmp1 + 125;                    15
   term= i < MAX;                          16
   write(y,output_y);                      17
   break;                                  18
 }                                         19
 state=cfg_trans(state);                   20
}                                          21
```

Auflistung 5.2: Reduzierter CFG, abgebildet auf das C–Template

In Algorithmus 5.4 wird ein mögliches Vorgehen gezeigt, um aus einem Graphen G_K und einer Menge Unterbrechungspunkte BP die kombinierbaren Zustände zu ermitteln. Für jedes $v \in V_K$ werden mögliche Pfade extrahiert. Die Traversierung eines Pfades wird beendet, wenn ein Zustand mehrere Nachfolger besitzt, ein Element von BP ist oder der Nachfolgezustand mehrere Vorgänger aufweist und damit der Beginn einer Schleife detektiert wurde. Die Menge Q hält den partiell traversierten Pfad. Pfade, deren Traversierung beendet wurde, werden in die Menge *combined* verschoben. Für das Beispiel 2.2.14 auf Seite 64 ergibt sich mit $BP = \{v_3\}$ das Resultat zu

$$combinable = \{..., \{v_S, v_1\}, \{v_2, v_3\}, \{v_4, v_5\}, \{v_E\}...\}. \tag{5.17}$$

Abbildung 5.8 zeigt den reduzierten Kontrollfluss und Datenfluss für das Beispiel 2.2.14 auf Seite 64. Die links dargestellte Auflistung 5.2 beinhaltet den wesentlichen Code-Ausschnitt des korrespondierend ausgefüllten C-Templates der SW-Implementierung. Das Template basiert auf einer Hauptschleife, die über eine *switch-case*-Anweisung iteriert. Eine *case*-Anweisung korrespondiert jeweils mit einem Knoten in G_K. In der letzten Anweisung der Schleife erfolgt das Rufen der Übergangsfunktion für die Ermittlung des Folgezustandes.

5.3.2.5 Unterbrechung von Schleifen mit Pipelining

Die Unterbrechung von Schleifen ist insbesondere bei gegebener oberer Schranke für die Dispatchlatenz von Relevanz. Schleifen müssen Unterbrechungspunkte enthalten, wenn die Anzahl der Schleifeniterationen unbestimmt ist oder die Anzahl der Ausführungsschritte im Schleifenkörper die maximal erlaubte Dispatchlatenz überschreiten. Dem Pipelining, das die HLS als wesentliches Mittel zur Optimierung von Schleifen nutzt (vgl. Kapitel 2.2.4.3), bedarf es hier gesonderter Betrachtung. Bei einer Pipeline wird der Datenfluss über mehrere Iterationen hinweg verschachtelt, indem nach einer Vorschrift $controlstep(s_i)$ für die Stufen $s \in S_{Pipe}$ der Pipeline mehrere Zustände $v \in V_{K_{Seq}}$ des Kontrollflussgraphen $G_{K_{Seq}}$ im Schleifenkörper gleichzeitig aktiviert werden. Für gleichzeitig aktivierte Zustände v sind die zuvor definierten Verfahren zur Kontextbestimmung (vgl. Algorithmus 5.1) und zur Abbildung des FSMD auf die Implementierungen in HW und SW jedoch nicht gültig. Ein ähnliches Problem wurde von Hao, Ray und Xie [HRX12] im Zusammenhang mit der formalen Äquivalenzprüfung von Ergebnissen der Verhaltenssynthese untersucht. Die nachfolgend gezeigte Lösung wurde in [RLH13a; RLH13b] veröffentlicht.

Zur Lösung des Problems bedarf es einer Transformation:

$$pipeloop : G_{K_{Seq}} \overset{controlstep(v_i)}{\rightarrow} G_{K_{Pipe}} \tag{5.18}$$

bei der ein Kontrollgraph $G_{K_{Pipe}}$ entsteht, dessen Knoten $V_{K_{Pipe}}$ den Stufen S_{Pipe} entsprechen und damit zu jedem Zeitschritt genau ein Zustand aktiv ist. In den nachfolgenden Ausführungen werden HLS-erzeugte Pipelines

zunächst formal beschrieben und anschließend eine entsprechende Transformation abgeleitet. Von Rau, Schlansker und Tirumalai [RST92] wurde diese Transformation unter der Bezeichnung *Modulo Scheduled Loop* in einem ähnlichen Kontext eingeführt.

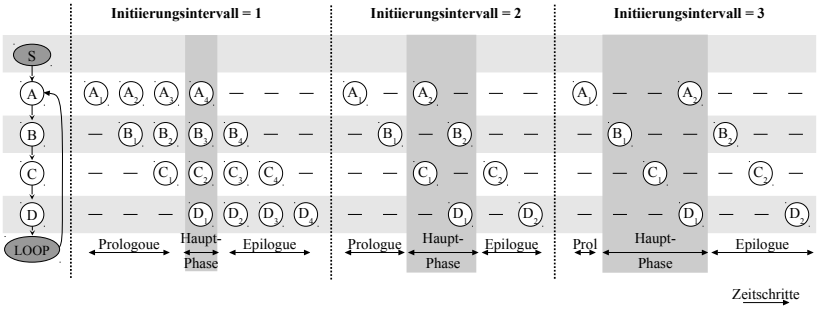


Abbildung 5.9: Ausführung eines Schleifenkörpers in einer Pipeline mit einer Tiefe von $p = 4$ bei unterschiedlichen Initiierungsintervallen

Pipelines lassen sich durch die Anzahl der Stufen bzw. die Pipelinetiefe p und die Länge des Initiierungsintervalls l zwischen der Aktivierung zwei aufeinanderfolgender Stufen charakterisieren. Für die Betrachtung HLS-erzeugter Pipelines, kann von einem festen Initiierungsintervall ausgegangen werden, da während der Ausführung keine Synchronisation mit externen Ereignissen erfolgt und damit alle Operatoren im gleichen relativen Zeitabstand innerhalb einer Stufe ausgeführt werden. Die Abbildung 5.9 zeigt eine Schleife mit 4 Zuständen im Schleifenkörper, der in einer Pipeline der Tiefe $p = 4$ ausgeführt werden kann. Die Indizes an den Zuständen deuten die k-te Ausführung der jeweiligen Stufe an. Mögliche Werte für das Initiierungsintervall liegen zwischen $1 \leq l \leq 4$. Der konkrete Wert ergibt sich aus den Datenabhängigkeiten zwischen den Stufen und sollte minimal sein um die höchstmögliche Performanz zu erreichen. Mit dem Maximalwert des Intervalls von $l = 4$ erfolgt die sequentielle Ausführung der Schleife. Verallgemeinert benötigt eine Implementierung ω Kontrollschritte, um eine Sequenz von n Iterationen auszuführen, wobei der folgende Zusammenhang gilt:

$$\omega = l(n - 1 + p). \tag{5.19}$$

Die Ausführung einer Schleife in Fließbandverarbeitung gliedert sich in eine *Prolog-Phase*, in der die Pipeline gefüllt wird, eine *Hauptphase*, in der die Pipelinestufen nach einem regulären Schema aktiviert werden, und eine *Epilog-Phase*, in der verbleibende Stufen bis zum Ende der Ausführung leer laufen. Die Anzahl der unterschiedlichen Stufen in den Phasen einer spezifischen Pipeline ergeben sich nach den Gleichungen 5.20-5.22. In jeder Stufe v_i wird ein Satz O_i von Operationen ausgeführt. In der regulären Hauptphase wird die Stufe i im Kontrollschritt k nach Gleichung 5.23 aktiviert.

$$size_{pro} = p - l \tag{5.20}$$

$$size_{haupt} = l \tag{5.21}$$

$$size_{epi} = p - \begin{cases} 0, & \text{wenn } p \mod l = 0 \\ l, & \text{sonst} \end{cases} \tag{5.22}$$

$$controlstep(v_i) : k = ((i-1) \mod l) + 1. \tag{5.23}$$

Der Transformationsalgorithmus 5.5 gliedert sich in drei Unterteile für den *Prolog*, den *Hauptteil* und den *Epilog* der Pipeline. Für jede Phase iteriert eine Hauptschleife über die Anzahl der Stufen und baut mit $v \in V_{K_{Pipe}}$ einen neuen linearen Kontrollflussgraphen $G_{K_{Pipe}}$ auf. Für jede Stufe v werden in einer inneren Schleife die aktiven Zustände im Kontrollgraph von $G_{K_{Seq}}$ nach der Vorschrift *constraint*() ermittelt und die durch $\sigma : V_{D_{Seq}} \rightarrow V_{K_{Seq}}$ assoziierten Operationen $o \in V_D$ dem untersuchten Zustand v des neuen Graphen $G_{K_{Pipe}}$ zugeordnet. Abschließend werden die drei erzeugten Graphen kombiniert und die iterative Hauptphase durch eine mit der Abbruchbedingung annotierte Kante vom letzten zum ersten Zustand ergänzt. Abbildung 5.10 veranschaulicht die Transformation für das Beispiel aus Abbildung 5.9 mit dem Initiierungsintervall $l = 1$ einer Unterbrechung in der Mitte des sequentiellen Schleifenkörpers.

Algorithmus 5.5 Transformation vom sequentiellen Kontrollfluss einer Schleife zum Kontrollfluss einer Pipeline

Generieren der Prolog-Phase

1: **for all** $\{v_i \in V_{K_{Seq}} \mid i \in \{1, 2, \ldots, size_{pro}\}\}$ **do**
2: $\quad V_{K_{Pipe}} \leftarrow V_{K_{Pipe}} \cup \{v_i\}$
3: $\quad E_{K_{Pipe}} \leftarrow E_{K_{Pipe}} \cup \{(v_{i-1}, v_i)\}$
4: $\quad$ **for all** $\{v_j \in V_{K_{Seq}} \mid j \in \{1, 2, .., i\}\}$ **do**
5: $\quad\quad$ **if** $\{(controlstep(v_j))\} \cap \{(j \mod (l))\} \neq \emptyset$ **then**
6: $\quad\quad\quad O_i \leftarrow O_i \cup O_j$

Generieren der Hauptphase

1: **for all** $\{v_i \in V_{K_{Seq}} \mid i \in \{size_{pro} + 1, \ldots, p\}\}$ **do**
2: $\quad V_{K_{Pipe}} \leftarrow V_{K_{Pipe}} \cup \{v_i\}$
3: $\quad E_{K_{Pipe}} \leftarrow E_{K_{Pipe}} \cup \{(v_{i-1}, v_i)\}$
4: $\quad$ **for all** $\{v_j \in V_{K_{Seq}} \mid j \in \{1, 2, .., p\}\}$ **do**
5: $\quad\quad$ **if** $\{(controlstep(v_j))\} \cap \{(j \mod (l))\} \neq \emptyset$ **then**
6: $\quad\quad\quad O_i \leftarrow O_i \cup O_j$
7: $E_{Pipe} \leftarrow E_{Pipe} \cup \{(v_{size_{pro}+1}, v_p)\}$

Generieren der Epilog-Phase

1: **for all** $\{v_i \in V_{K_{Seq}} \mid i \in \{p + 1, \ldots, p + size_{epi}\}\}$ **do**
2: $\quad V_{K_{Pipe}} \leftarrow V_{K_{Pipe}} \cup \{v_i\}$
3: $\quad E_{K_{Pipe}} \leftarrow E_{K_{Pipe}} \cup \{(v_{i-1}, v_i)\}$
4: $\quad$ **for all** $\{v_j \in V_{K_{Seq}} \mid j \in \{i, i + 1, .., p\}\}$ **do**
5: $\quad\quad$ **if** $\{(controlstep(v_j))\} \cap \{(j \mod (l))\} \neq \emptyset$ **then**
6: $\quad\quad\quad O_i \leftarrow O_i \cup O_j$

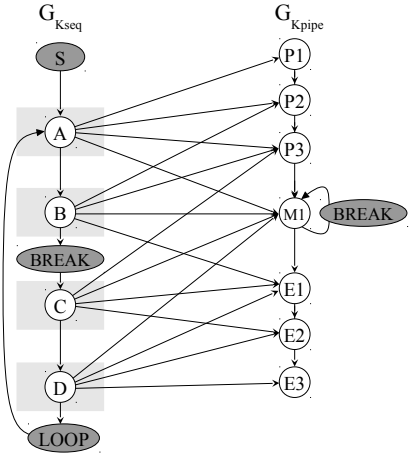


Abbildung 5.10: Transformation Kontrollflusses einer sequentiellen Schleife in eine Pipeline mit vier Ausführungsschritten bei einem Initiierungsintervall $l = 1$

6 Laufzeitumgebung und Systemverwaltung

6.1 Ausführungsmodell

Im Rahmen der vorliegenden Arbeit sollen die parallelen Anwendungskonzepte nach Kapitel 2.3 für ein heterogenes ACS als zentraler Bestandteil einer Entwurfsmethodik umgesetzt werden. Eine Anwendung ist dabei komponiert aus mehreren parallel ablaufenden Teilanwendungen. Für ein heterogenes System stellen dabei Prozesse und Threads ein einheitliches Konstrukt der parallelen Teilanwendung dar, das im Folgenden kurz als *Task* bezeichnet werden soll. Ein Task wird als eine abgeschlossene Einheit von Befehlsfolgen betrachtet (vgl. Kapitel 2.3), die kontroll- und datenbedingte Abhängigkeiten zur Systemumgebung oder anderen Tasks besitzen kann. Ausgangspunkt bildet eine Teilanwendungsbeschreibung in einer Hochsprache, die entlang der Grenzen einer Funktion bzw. Prozedur verfasst ist.

Die Ausführung einer Anwendung beginnt mit dem Start des Haupt-Tasks, der die Kontrolle über das Ausführen bzw. Verzweigen[1] in weitere Tasks besitzt. Mit der sogenannten *Fork-Join-Parallelisierung* ergibt sich für die Anwendung die in Abbildung 6.1 dargestellte flache, abgeschlossene Struktur. Der Datenaustausch zwischen den Tasks erfolgt über *gemeinsamen Speicher*. Konkurrierende Zugriffe auf Werte im Speicher können mit *Mutex* gesichert werden. Zur Synchronisation können Tasks mittels *Zusammenführungen*[2] auf Beendigung eines anderen Task warten oder die Ausführung von einer *Bedingungen*[3] abhängig machen. Dieser Satz an Primitiven stellt das Minimum zur Umsetzung einer parallelisierten Anwendung dar und bildet die Basis der prototypischen Umsetzung als Nachweis des Konzepts der Arbeit. In Tabelle 6.1 sind diese Primitiven anhand der korrespondierenden Funktionen aus dem POSIX-Standard [Ins94] beispielhaft gezeigt.

[1] *verzweigen*: to fork (engl.)
[2] *zusammenführen:* to join (engl.)
[3] *Bedingung*: condition (engl.)

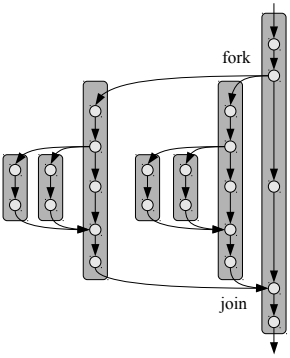


Abbildung 6.1: Typische Struktur einer Anwendung mit Fork-Join-Parallelisierung

Das Ausführungsmodell unterbrechbarer nebenläufiger Tasks ermöglicht Wettlaufsituationen[4] und Blockaden im Programmablauf. Deren Ausschluss liegt in der Verantwortung des Anwendungsentwurfes und bedingt den richtigen Einsatz von Mutex zum Schutz kritischer Bereiche und der Synchronisation mittels Bedingungen.

Funktionsklasse	Beschreibung
Kontrollfluss	pthread_create() pthread_join() pthread_exit()
Wechselseitiger Ausschluss (Mutex)	pthread_mutex_init() pthread_mutex_lock() pthread_mutex_unlock() pthread_mutex_destroy()
Bedingung	pthread_cond_init() pthread_cond_wait() pthread_cond_signal() pthread_cond_broadcast() pthread_cond_destroy()

Tabelle 6.1: Für das Anwendungsmodell notwendige Funktionen am Beispiel des POSIX-Standards [Ins94]

[4] *Wettlaufsituation*: race condition (engl.)

6.2 Anforderungen

Aus dem Ausführungsmodell ergeben sich die Anforderungen an Laufzeitumgebung und Systemverwaltung, die das Management der Rechenwerke, der Anwendungen und der in Bearbeitung befindlichen Tasks umfassen.

Abbildung 6.2 zeigt Aufgaben, die sich bei der Verwaltung eines heterogenen Systems mit dem Ausführungsmodell [ZKG09] ergeben. Ausgangspunkt bilden die simultan ausgeführten Nutzeranwendungen und die Laufzeitumgebung selbst (Sektion A). Abhängig vom Systemzustand, externer Anforderungen und der Struktur der auszuführenden Anwendung entscheidet ein zentraler Scheduler (Sektion B) über die Verteilung der Tasks in die Ausführungsdomänen HW und SW. Aufgrund von Unterschieden in Bezug auf Ausführungszeit, Rekonfigurationsdauer und Ressourcenmanagement sind in einer zweiten Stufe die Domänen SW (Sektion C) und HW (Sektion D) einzeln zu verwalten. Für die RPU ist in Sektion C zusätzlich eine Platzierungsentscheidung zu treffen und die Rekonfiguration durchzuführen.

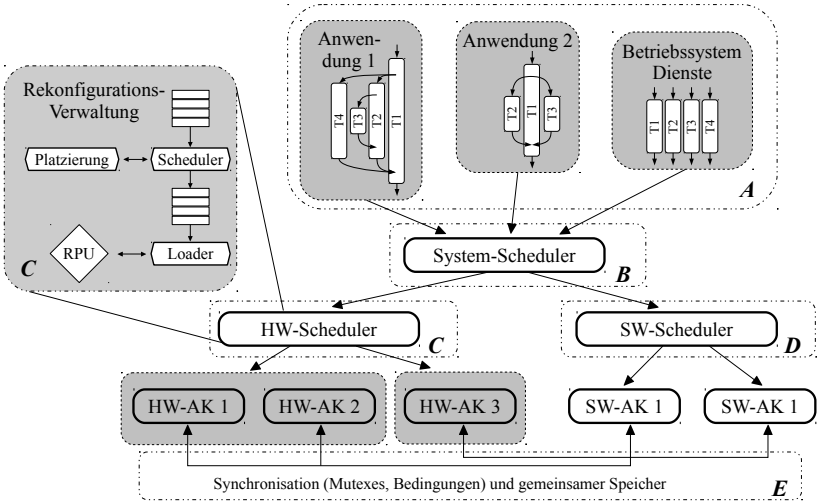


Abbildung 6.2: Aufgaben zur Verwaltung eines heterogenen Systems für die Unterstützung eines parallelen Ausführungsmodells [ZKG09]

Für die Tasks muss die Kommunikation mit der Laufzeitumgebung und anderen Tasks unabhängig von Ausführungsort und -domäne möglich sein. Zur Entkopplung der unterschiedlichen Ausführungsgeschwindigkeiten bedarf es einer geeigneten Schnittstelle und einem definierten Kommunikationsprotokoll (Sektion E). Der Kommunikationskanal darf auch bei Unterbrechung und Migration des Kommunikationspartners (Laufzeitumgebung oder Task) nicht blockieren oder abreißen. An dieser Stelle muss die Virtualisierung mittels AK stattfinden, sodass sowohl der Task als auch die Systemverwaltung eine einheitliche Sicht auf das heterogene ACS erhält.

6.3 Konzeption und Aufbau

Die im Rahmen der Arbeit entworfene Laufzeitumgebung gliedert sich, entsprechend der in Abbildung 6.3 skizzierten Hauptteile, in: Systemverwaltung, HW-Manager, SW-Manager und Rekonfigurations-Manager. Schnittstellen besitzt die Laufzeitumgebung zu Festspeicher, Hauptspeicher, Rechenwerken und Nutzer.

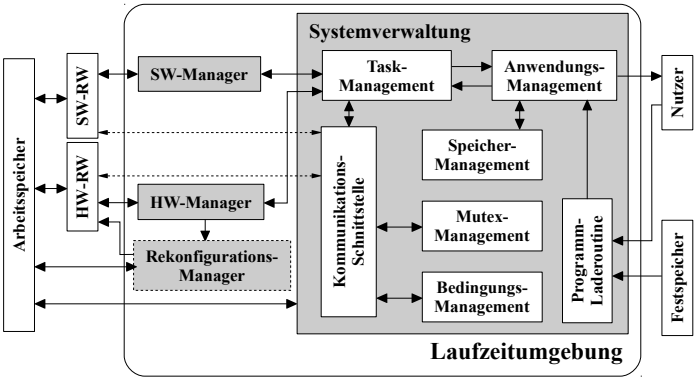


Abbildung 6.3: Aufbau der Laufzeitumgebung für ein heterogenes System

6.3.1 Programm-Laderoutine, Anwendungsmanager und Speichermanagement

Die Programm-Laderoutine lädt auf Anforderung die erweiterte Binärdatei einer zu startenden Anwendung aus dem Festspeicher und entpackt deren

Inhalt. Während dieser Initialisierungsphase werden neben den Binärcodes der Tasks für die verschiedenen Rechenwerke auch die Programmbeschreibungsdatei extrahiert und im Arbeitsspeicher abgelegt. Die Programmbescheibungsdatei enthält den Taskgraphen G_T, der die Abhängigkeiten der Anwendungsteile untereinander abbildet, den Haupttask der Anwendung identifiziert und die Größe des notwendigen globalen Speichersegments spezifiziert. Darüber hinaus sind am Taskgraph Constraints hinsichtlich der Ausführungsdomäne für die Tasks annotiert. Die Daten werden dem Anwendungsmanager übergeben, der seinerseits die Ablauffähigkeit der Anwendung auf dem heterogenen System prüft. Die Ablauffähigkeit ist gegeben, wenn die notwendigen Systemressourcen zur Ausführung der Anwendung vorhanden sind. Schließlich wird ein globales Speichersegment vom Speichermanagement angefordert und der Haupttask gestartet.

Das Speichermanagement verwaltet den globalen Speicher, der für alle Rechenwerke und AK des Systems als kontinuierlicher Adressraum sichtbar ist. Bei Anwendungsstart werden zusammenhängende Segmente dynamisch zugewiesen, die der Anwendung als globaler Speicher und dem Task-Management als Kontext-Speicher zur Verfügung stehen. In diesem Zusammenhang tritt eine externe Fragmentierung des Speichers auf, die durch einen geeigneten Zuweisungsalgorithmus zu minimieren ist. Interne Fragmentierung kann hingegen, bei der Zuweisung auf Wörter (32 Bit Datenbreite) gerundeter Bereiche, vernachlässigt werden. Zum einen entwickelt sich die Anzahl zuzuweisender Speicherblöcke linear mit der Anzahl gestarteter Anwendungen, die als beschränkt zu betrachten ist. Zum anderen sind die Speicherblöcke groß im Vergleich zur Rundungsgrenze.

6.3.2 Kommunikationsschnittstelle

Die Laufzeitumgebung kommuniziert zur Koordination der Task-Ausführung und für die Zuteilung zentraler Ressourcen über ein Kommando-Protokoll mit den ablaufenden Tasks. Dazu besitzt jede Task-Instanz eine entsprechende Kommando-Schnittstelle, die gekapselt über den AK sowohl in der SW- als auch in der HW-Repräsentation implementiert ist. Die Kommunikation erfolgt nach dem Frage-/Antwort-Prinzip auf dedizierten Kanälen. Abbildung 6.4 zeigt die Punkt-zu-Punkt-Verbindungen zwischen Laufzeitumgebung und Task. In Richtung des Tasks erfolgt die Übertragung der Nachrichten in jeweils einem separaten Kanal für Anfragen und Antworten, um deren Behandlung innerhalb der HW in getrennten Automaten zu realisieren.

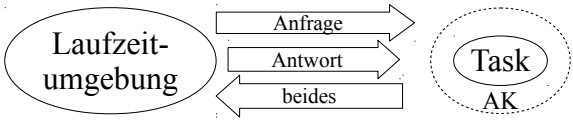


Abbildung 6.4: Kommunikationsschnittstelle zwischen Laufzeitumgebung und Task

Für die Kanäle sind Nachrichten-Puffer vorgesehen, um die Entkopplung der Kommunikationspartner untereinander und die Trennung der Kommunikation von der Berechnung innerhalb eines Tasks zu erreichen. Die Puffergröße kann durch das Frage-/Antwort-Prinzip auf eine Nachricht beschränkt werden. Für den Betrieb der Schnittstelle ist zu bedenken, dass die Kommunikation aus Sicht der Laufzeitumgebung simultan mit mehreren Tasks erfolgt. Außerdem können sich die Kanalenden auf Seiten der Tasks während dessen Lebenszeit zwischen den Ausführungsdomänen verschieben. Eine effiziente Lösung ergibt sich, indem die Nachrichtenpuffer auf der Kanalseite des Tasks integriert werden und die Laufzeitumgebung die Puffer nichtblockierend abruft. Das ist mit einer Polling-Strategie[5] ohne zusätzlichen HW-Aufwand realisierbar. Die Signalisierung über ein Interrupt-Signal[6] sollte vorgesehen werden, wenn sich Laufzeitumgebung und SW-Task ein gemeinsames Rechenwerk teilen oder die Laufzeitumgebung im Userspace ausgeführt wird, um unnötige Kontextwechsel zur Laufzeitumgebung für den Statusabruf des eingehenden Kanals zu vermeiden.

Die Kommunikation erfolgt nach einem nachrichtenbasierten Protokoll. Der Kopfteil der Nachricht kodiert das auszuführende Kommando und der Lastteil trägt zugehörige Parameter. Tabelle 6.2 zeigt die gerichteten Nachrichten für den Informationsaustausch zwischen Laufzeitumgebung und Task. In [Bil08] ist die Realisierbarkeit dieser Kommunikationsschnittstelle mittels HLS-Werkzeug CoDeveloper [PT05] nachgewiesen und eine detaillierte Definition der Nachrichten auf Bitebene beschrieben.

[5] *Polling (engl.)*: Strategie für den Datenaustausch nach dem Abrufverfahren.
[6] *Interrupt (engl.)*: Unterbrechung

Nachrichten-klasse	Laufzeitumgebung			Task	
	Kommando	Last		Kommando	Last
Task-Management	T_Run	Task-ID, Parame-ter	$\Longrightarrow$		
	ACK/NACK	Task-ID	$\Longleftarrow$	T_Create	
	ACK/NACK		$\Longleftarrow$	T_Join	Task-ID
	T_Pause		$\Longrightarrow$	ACK	Task-ID, Break-Point
	T_Resume	Task-ID, Break-Point	$\Longrightarrow$		
			$\Longleftarrow$	Exit	Task-ID, Rückga-bewert
Mutex-Management	ACK/NACK	Mutex-ID	$\Longleftarrow$	M_Create	
	ACK/NACK		$\Longleftarrow$	M_Lock	Mutex-ID
			$\Longleftarrow$	M_Unlock	Mutex-ID
			$\Longleftarrow$	M_Destroy	Mutex-ID
Bedingungs-Management	ACK/NACK	Cond-ID	$\Longleftarrow$	C_Create	Create
	ACK/NACK		$\Longleftarrow$	C_Wait	Cond-ID
			$\Longleftarrow$	C_Signal	Cond-ID
			$\Longleftarrow$	C_Bcast	Cond-ID
			$\Longleftarrow$	C_Destroy	Cond-ID

Tabelle 6.2: Kommunikationsprotokoll zwischen den Tasks und der Laufzeitumgebung

6.3.3 Ressourcenverwaltung und temporale Ablaufplanung der Tasks

Aus den Anforderungen der Sektionen B-E (vgl. Abbildung 6.2) ergeben sich die Aufgaben zur Verwaltung des Systems, der Tasks, der Ausführungsdomänen HW und SW sowie der Synchronisationsmechanismen (vgl. Abbil-

dung 6.3). Der Funktionsumfang der sich ergebenden Komponenten der Laufzeitverwaltung und konzeptionelle Lösungsansätze für diese Aufgaben werden in den folgenden Abschnitten erläutert.

Systemverwaltung Auf dieser Verwaltungsebene ist die gesamte Architektur aus Berechnungselementen, Speichern und Kommunikationsmitteln sicht- und steuerbar. Zentrale Aufgabe ist die Steuerung dieser Ressourcen durch Aktivieren, Deaktivieren oder Modifizieren, um externen Anforderungen an Systemparameter (z.B. Performanz, Leistungsverbrauch) zu genügen und dabei von den Möglichkeiten des domänenübergreifenden präemptiven Konzeptes zu profitieren. Beispielsweise durch das Verlagern der Tasks eines gering ausgelasteten Rechenwerks auf andere aktive Rechenwerke, um den Leistungsverbrauch des ACS zu reduzieren. Die Basis für die Verwaltung des Systems bildet der Architekturgraph G_A nach 2.1.1 als generische Beschreibung der zugrunde liegenden Systemarchitektur.

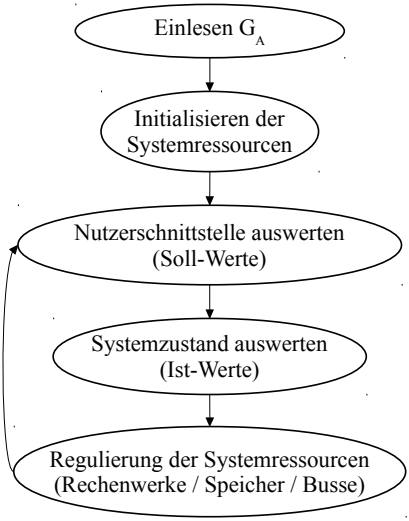


Abbildung 6.5: Ablaufdiagramm der Systemverwaltungsroutine

Abbildung 6.5 zeigt das Schema der Verwaltungsroutine. Nach dem Einlesen von G_A erfolgt die Initialisierung der Systemressourcen, z.B. das Ausführen eines Urladers (engl. Bootloader) für ein Rechenwerk der SW oder das Laden

des Bitstromes für den statischen Teil eines AK der HW (vgl. Kapitel 2.2.3). Die Hauptschleife der Systemverwaltung kann mit einer Regelschleife verglichen werden. Zunächst werden die Nutzereingaben ausgewertet und damit Soll-Werte für die regelbaren Systemparameter ermittelt. Anschließend werden die Ist-Werte dieser Parameter bestimmt und dem Steueralgorithmus übergeben. Dieser trifft schließlich die Verwaltungsentscheidungen zur Minimierung der Soll-Ist-Differenz. Die Regelalgorithmen für einzelne oder mehrere voneinander abhängige Systemparameter obliegen dem Nutzer der Laufzeitumgebung und des dynamisch partitionierbaren Systems.

Task-Management Die Verwaltung zur Ausführung angewiesener Tasks im System erfolgt in einer Liste mit Informationen über deren Ausführungszustand und Ausführungsdomäne. Aus Sicht des Task-Managers durchläuft ein Task das in Abbildung 6.6 dargestellte Zustandsdiagramm. Ausgehend von der Ausführungsanweisung beginnt das Leben eines Tasks im Zustand *Neu*. Dabei werden die notwendigen Ressourcen allokiert und die Übergabeparameter bei Start des Tasks am Kommunikationskanal bereitgestellt. Kommt der Task auf einem Rechenwerk zur *Ausführung*, werden die Parameter gelesen und der Zustand kann zwischen den Zuständen *Ausführung*, *Unterbrochen* und *Beendet* wechseln. Nach der Beendigung folgt der Zustand *Abgeschlossen*. In diesem Zustand wurde der Rückgabewert von der Laufzeitumgebung empfangen, die auf eine Zusammenführung wartenden Tasks wurden aktiviert und die allokierten Ressourcen können schließlich wieder freigegeben werden.

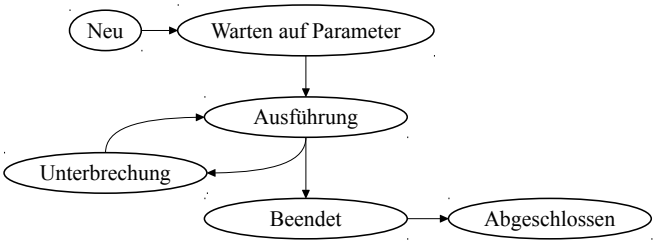


Abbildung 6.6: Zustände eines Tasks aus Sicht der Laufzeitumgebung

Vor dem Eintritt in den Zustand der *Ausführung* erfolgt die Zuordnung der Ausführungsdomäne, d.h. die Partitionierungsentscheidung für diesen An-

wendungsteil bezüglich HW und SW ist zu treffen. Dazu werden zunächst die im Entwurfszeitpunkt festgelegten Constraints beachtet. Die Constraints sind im Taskgraphen G_T der Anwendung annotiert und ergeben sich zum einen aus zwangsläufigen Partitionierungsvorgaben, die bei der statischen Codeanalyse im Entwurfsschritt *Codeumsetzung und Partitionierung* (vgl. Kapitel 4.2) festgelegt wurden. Zum anderen werden Ablaufpläne berücksichtigt, die bei einem möglichen statischem Scheduling der auszuführenden Anwendung im Entwurfsschritt *Eigenschafts- und Komplexitätsanalyse* festgelegt wurden. Liegen keine statisch ermittelten Bedingungen vor, erfolgt die Evaluation dynamischer Kriterien für die Zuweisung der Ausführungsdomäne. Hierbei werden die Task-bezogenen Eigenschaften aus G_T herangezogen sowie aktuelle Systemparameter evaluiert. Diese können sein:

- Auslastung der Domäne,
- Priorität von Task oder Anwendung,
- Nähe zum Ausführungsort weiterer Tasks dieser Anwendung und
- Kosten im heterogenen ACS
  (Ist die Rekonfiguration eines AK erforderlich?).

Im Rahmen der Arbeit wurde ein Round-Robin basiertes präemptives Verfahren implementiert. Tasks im Zustand *Neu* und *Unterbrochen* werden dazu in eine Warteschlange sortiert. Existieren freie Rechenwerke wird der erste geeignete Eintrag mit einer domänenspezifischen Zeitscheibe zur Ausführung gebracht und ggf. nach Ablauf der Zeit unterbrochen. Als geeignet gelten Einträge die in der Domäne des verfügbaren Rechenwerks ausführbar sind. Auf Basis der Erkenntnisse aus [Bil08] wurde das klassische Round-Robin Verfahren um die Dynamisierung der Zeitscheibenlänge ergänzt, um den Verlust im Zusammenhang mit einem Kontextwechsel zu reduzieren. Die Dynamisierung erfolgt indem die Zeitscheibenlänge mit wachsender Warteschlange vergrößert wird.

Mutex- und Bedingungs-Management In diesem Teil der Laufzeitumgebung werden Mutex und bedingte Variablen verwaltet, die zur Synchronisation und Ablaufsteuerung zwischen Tasks genutzt werden. Die Mutex können als Marke interpretiert werden, wobei eine Marke zu jedem Zeitpunkt genau einem oder keinem Task zugewiesen sein kann. Konkurrieren mehrere Tasks um die Zuweisung einer Marke, ist durch die Laufzeitumgebung eine Auswahl des begünstigten Tasks zu treffen. Das Auswahlverfahren lässt sich als Ablauf- bzw. Schedulingproblem auffassen, sodass die in Abschnitt 2.4 eingeführten Entscheidungsgrundsätze gelten.

Mit Bedingungsvariablen können Tasks auf das Eintreten einer Bedingung warten oder den Eintritt einer Bedingung an darauf wartende Tasks signalisieren. Die Signalisierung kann dabei entweder an einen (Single-Cast) oder an alle (Broadcast) auf die Bedingung wartenden Tasks erfolgen. Warten mehrere Tasks auf eine Bedingung, trifft bei der Single-Cast Signalisierung die Laufzeitumgebung die Auswahl des begünstigten Tasks und damit eine Schedulingentscheidung.

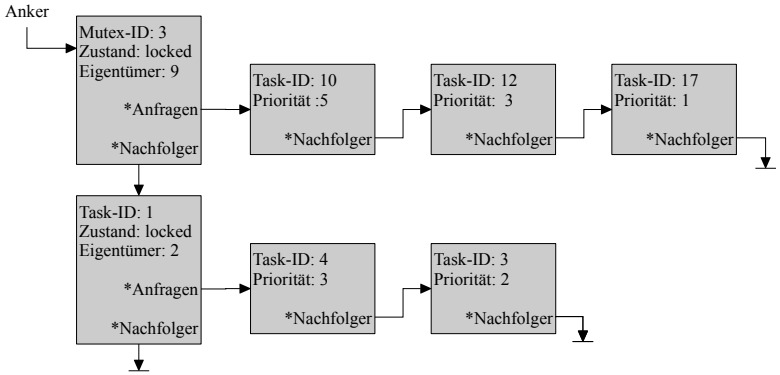


Abbildung 6.7: Prioritätsbasiertes Scheduling mit Alterung zur Mutexvergabe

Aus Sicht der Laufzeitumgebung lassen sich sowohl Mutex als auch Bedingungsvariablen in einer hierarchischen Liste zentral verwalten (vgl. Abbildung 6.7). Die Hauptliste enthält die gültigen Instanzen mit Informationen über Zustand, Eigentümer der Ressource und einer verzweigenden Auflistung der auf die Ressource wartenden Tasks. Instanzen der Hauptliste werden über Initialisierungs- bzw. Destruktionsaufrufe erstellt oder entfernt. Instanzen einer Warteschlange werden jeweils bei Zuweisungsanfrage einer geblockten Mutex bzw. beim Warten eines Tasks auf eine Bedingung erzeugt.

Für die Schedulingentscheidung der Verteilung von Mutex und Bedingungsvariablen wurde im Rahmen der Arbeit beispielhaft ein prioritätsbasiertes Zuteilungsverfahren mit Alterung [Har08] umgesetzt. Die Warteschlange wird dazu nach Priorität sortiert. Die Zuteilung der Marke erfolgt stets an den ersten Task in der Warteschlange, der sich im Ausführungszustand befindet. Bei einer Zuteilungsentscheidung werden die Prioritäten der verbleibenden Ein-

träge in der Liste inkrementiert, um Fairness gegenüber niederpriorisierten Tasks zu gewährleisten.

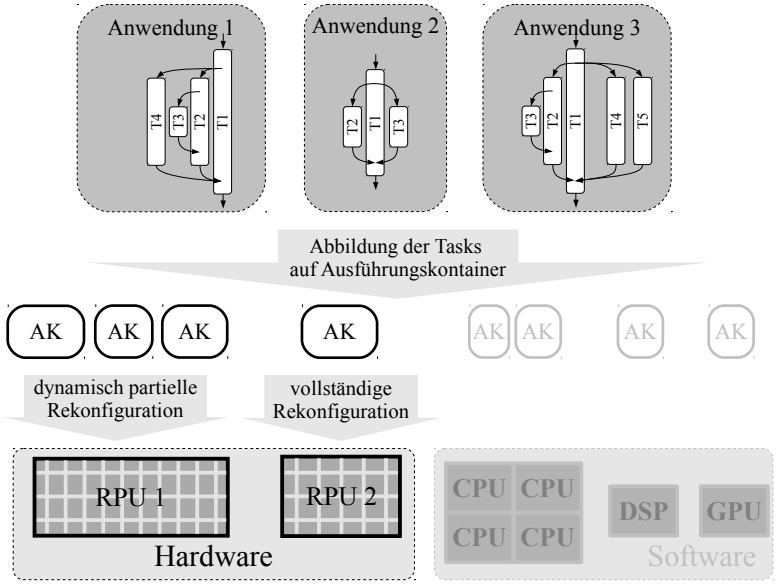


Abbildung 6.8: Abbildung zur Ausführung angewiesener Tasks auf RPU der HW-Domäne

HW-Manager (Rekonfigurationsmanager) Der HW-Manager ordnet den zur Ausführung in der HW-Domäne angewiesenen Tasks den AK in den RPU des heterogenen ACS zu und führt darüber hinaus die notwendigen Schritte zur Rekonfiguration der RPU über den Rekonfigurationsmanager aus. Abbildung 6.8 illustriert diese Schritte. Die Verwaltung der RPU stellt dabei die zentrale Aufgabe dar. Der HW-Manager erhält initial über den Architekturgraphen G_A Kenntnis über die verfügbaren AK der HW-Domäne, deren Kapazität sowie die Zuordnung zu entsprechenden RPU (z.B. ein FPGA) und dessen Rekonfigurationsschnittstelle. AK können im Zustand *belegt* oder *frei* sein, wobei im freien Zustand der vorherige ggf. noch bestehende Konfigurationszustand zu halten ist. Die Rekonfiguration eines AK erfolgt durch die Bereitstellung des Task- und AK-spezifischen Bitstromes an der Rekonfigurationsschnittstelle der zugehörigen RPU. Liegen mehrere AK in einer RPU

erfolgt die Rekonfiguration durch einen partiellen Bitstrom ohne Beeinträchtigung der anderen AK der RPU (vgl. Kapitel 2.2.3). Andernfalls wird die RPU vollständig rekonfiguriert. Schließlich wird durch Aktivieren des entsprechenden Reset-Signals des AK die Abarbeitung des Tasks gestartet.

Der HW-Manager kommuniziert dem Task-Management den Belegungszustand der AK. Im Gegenzug weist das Task-Management die Tasks zur Ausführung in der HW-Domäne an. Diese Tasks befinden sich dabei im Zustand *Neu* oder *Unterbrochen* und werden in eine Warteschlange aufgenommen. Nach dem FIFO-Prinzip werden die Einträge konkreten AK zugewiesen, wobei für die Auswahl zunächst geprüft wird, ob ein freier AK existiert, die bereits für diesen Task konfiguriert wurde. Ist dies nicht der Fall wird ein AK mit ausreichender Kapazität gewählt. Damit erfolgt für den Task der Übergang in den Zustand *Ausführung* und die Kommunikationsschnittstelle wird über die aktuelle Lage der Kommunikationskanäle in Bezug auf den AK informiert. Erreicht ein Task den Zustand *Beendet* bzw. *Unterbrochen* wird der AK freigegeben.

Stehen für eine Zuweisung eines Tasks mehrere AK mit unterschiedlicher Kapazität zur Auswahl, muss eine Platzierungsentscheidung getroffen werden. Das stellt ein zweidimensionales Platzierungsproblem dar. Im Rahmen der Arbeit wurde dazu *Best-Fit* als minimales Schema [Jr+80] realisiert, das den kleinsten passenden AK auswählt und damit externe Fragmentierung minimiert. Steht kein geeigneter AK zur Verfügung, wird der Task zurück an das Task-Management gegeben. Es ist denkbar, bei der Auswahl des AK die Größe der in der Warteschlange stehenden Tasks und Umlagerungsmöglichkeiten bereits laufender Tasks innerhalb der HW-Domäne zu berücksichtigen und einen prioritätsbasierten Tradeoff innerhalb der HW-Domäne zu erreichen. Diese Entscheidung führt zu einem Partitionierungs- und Schedulingproblem mit NP-vollständiger Komplexität, dessen Lösung vielfältig untersucht wurde [LC91; Die+00]. Die relativ hohe Rekonfigurationszeit von RPU gilt bei der Umlagerung innerhalb der HW-Domäne als wesentliches Hemmnis [JTW07]. Im Kontext dieser Laufzeitumgebung wird deshalb darauf gesetzt, die Zeitscheiben für die Ausführung in HW geeignet zu wählen und kein weiteres Scheduling in der HW-Domäne durchzuführen.

Betrachtet man das Rechenprofil des HW-Managers in der Laufzeitumgebung, zeigt sich, dass der Transfer der Bitstromdaten zur Rekonfigurationsschnittstelle der RPU einen überproportionalen Zeitanteil besitzt. Das wird insbesondere durch die großen Bitströme heutiger RPU kritisch, bei denen der Durchsatz der Rekonfigurationsschnittstellen nicht in gleichem Maße gesteigert werden kann wie die Anzahl der Konfigurationsbits zunimmt. Eine

geeignete Lösung, die insbesondere auch für heterogene Systemarchitekturen nach Typ C (vgl. Kapitel 2.1.5) anwendbar ist, wird in [Mei10] vorgestellt.

SW-Manager Die Rechenwerke der SW-Domäne (Prozessoren) und die zugehörigen AK werden vom SW-Manager verwaltet. Über den Architekturgraphen G_A erhält der SW-Manager Kenntnis von den Rechenwerken der SW-Domäne, deren Typ, Fähigkeiten und Lage im globalen Adressbereich. Für jeden Prozessor wird ein AK angelegt. Für Multitasking-fähige Prozessoren können mehrere AK erzeugt werden. Dies kann auch dynamisch erfolgen, sodass die Anzahl der AK in der SW-Domäne durch den SW-Manager bedarfsabhängig variiert werden kann.

Aus Sicht des Task-Managements haben HW- und SW-Manager die gleiche Aufgabe, einen zur Ausführung angewiesenen Task im Zustand "Neu" oder "Unterbrochen" durch Zuweisung eines AK in den Zustand "Ausführung" zu überführen. Stehen Rechenwerke mit unterschiedlichen Eigenschaften zur Verfügung (z.B. CPU und GPU), existieren Freiheitsgrade bei der Zuweisung, sodass die Entscheidung hinsichtlich eines spezifischen Optimums getroffen werden kann. Optimierungskriterien sind beispielsweise Leistungsaufnahme oder Performanz des ACS. Als unmittelbare Entscheidungskriterien bei der AK-Zuweisung dienen der noch verfügbare Programmspeicher und die maximale Anzahl paralleler AK im Rechenwerk. Darüber hinaus kann die Wahl des Rechenwerks optimal auf die im G_T annotierten Eigenschaften des auszuführenden Tasks abgestimmt werden. Beispielsweise sollte ein Task mit intensiver Matritzenrechnung einem Vektorprozessor zugeordnet werden.

Der SW-Manager führt dann die Schritte zum Laden des Tasks in den AK und zur Aktivierung des AK durch. Für alleinstehende Rechenwerke wird der spezifische Binärcode in den Programmspeicher geladen und der Ausführungszeiger auf den ersten Befehl des Tasks gesetzt. Für Rechenwerke, die durch ein Betriebssystem verwaltet werden, wird ein gekapselter nebenläufiger Prozess als AK gestartet. Abschließend wird die Kommunikationsschnittstelle über die Lage der Kommunikationskanäle des Tasks bezüglich des AK informiert.

6.3.4 Übersicht des Ausführungsablaufs einer Anwendung

Mit den beschriebenen Komponenten und Funktionen der Laufzeitumgebung gliedert sich der Ablauf einer Anwendung in Teilschritte, die den in Abbildung 6.9 nummerierten Kanten entsprechen und nachfolgend näher erläutert werden.

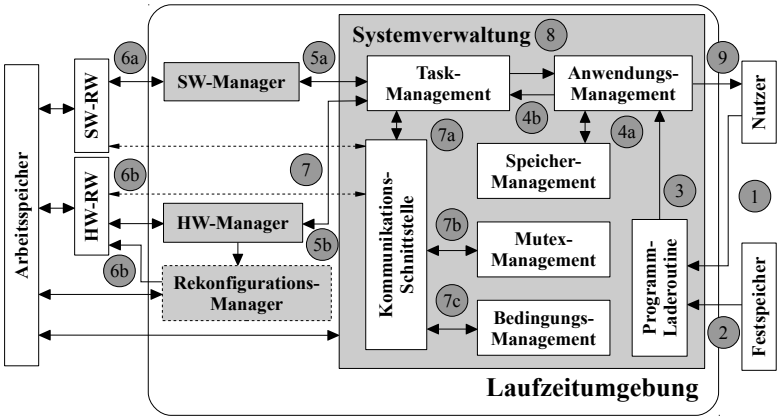


Abbildung 6.9: Ablauf in der Laufzeitumgebung beim Start einer Anwendung

1. Anfrage aus der Systemumgebung bzw. des Nutzers zum Ausführen einer Anwendung.
2. Laden des Binärcodes der Anwendung durch die Laderoutine.
3. Übergabe des Anwendungsgraphen an das Anwendungs-Management. Hier wird die Lauffähigkeit der Anwendung auf dem System geprüft.
4. Start der Anwendung durch:
   a) Allokation eines globalen Speicherblocks im Arbeitsspeicher,
   b) Start des Haupt-Tasks der Anwendung durch den Task-Manager.
5. Scheduling der zur Ausführung angewiesenen Tasks durch Zuweisung einer Ausführungsdomäne,
   a) in SW oder
   b) in HW.
6. Zuweisung der im Wartezustand befindlichen Tasks an AK innerhalb der zugewiesenen Ausführungsdomäne.
   a) Der SW-Manager verwaltet die AK der SW-Domäne.
   b) Der HW-Manager verwaltet die AK der HW-Domäne, wobei der Rekonfigurationsmanager ggf. das Umkonfigurieren der RPU ausführt.

7. Tasks kommunizieren während der Ausführung mit der Laufzeitumgebung,
   a) zum Starten oder Zusammenführen mit anderen Tasks,
   b) zum Anfordern oder Freigeben einer Mutex oder
   c) zum Signalisieren oder Warten auf eine Bedingung.
8. Die Beendigung des Haupt-Tasks und der zugehörigen Kinder-Tasks wird an das Anwendungs-Management gemeldet.
9. Nach Freigabe des globalen Anwendungsspeichers wird dem Initiator der Anwendung deren Rückgabewert geliefert.

7 Implementierung der Entwurfsmethodik

Die Entwurfsmethodik für dynamisch verteilbare Anwendungen (vgl. Kapitel 4) auf heterogenen Systemen (vgl. Kapitel 2.1.5) wurde im Rahmen der vorliegenden Arbeit beispielhaft implementiert und auf einer heterogenen Zielplattform anhand von Beispielanwendungen validiert. Im folgenden Kapitel wird die eingesetzte Zielplattform zur Validierung der implementierten Analyse- und Synthesewerkzeuge sowie der Beispielanwendungen vorgestellt. Anschließend werden die realisierten HW-Erweiterungen zur Migrationsfähigkeit erläutert und die dafür gültigen Parameterwerte zur optimalen Unterbrechungspunktverteilung bestimmt.

7.1 Zielplattform, Betriebssystem und Laufzeitumgebung

Als Zielplattform für das heterogene ACS wurde das XUP-Board der Firma XILINX als Prototypen-Entwicklungsumgebung genutzt, das einen VirtexII-Pro-FPGA als zentralen Schaltkreis integriert. Dieser SRAM-basierte FPGA-Schaltkreis eignet sich für den Aufbau eines konfigurierbaren SoC, da neben den frei konfigurierbaren Schaltkreisressourcen zwei PowerPC-Prozessoren (PPC) fest, als Hardcore bzw. Hardmakro, integriert sind. Mit einem Taktteiler, einer internen Rekonfigurationsschnittstelle und Block-RAM sind weitere wichtige Peripherieeinheiten als fest verdrahtete Ressourcen vorhanden. Weitere systemintegrierte Peripherieblöcke wie Speicher-Controller, der Ethernet-Controller und ein RS232-Core sind als Softmakro ausgeführt. Sämtliche Komponenten sind über den Processor Local Bus (PLB) angebunden. Als Schaltkreisversion ist ein XC2VP30 [Xil11] mit ca. 31.000 Logikzellen und 428 Kb verteilter RAM-Zellen auf der Prototypenplattform verbaut. Die beiden PPC-Prozessoren lassen sich zu einem homogenen Multiprozessorsystem zusammenschalten. Für die vorliegenden Untersuchungen wurde jedoch zur Vereinfachung der Fehlersuche und für die Reproduzierbarkeit der Ergebnisse ein Einzelprozessorsystem konfiguriert. In Abbildung 7.1 ist das konfigurierbare System innerhalb des FPGA-Schaltkreises und die weitere FPGA-externe Systemperipherie dargestellt. Die Prototypen-Entwicklungsumgebung enthält dazu 256 MB DDR-Ram, einen 100-MBit-

Ethernet-Phy-Schaltkreis sowie RS232-Treiberschaltkreise. Dargestellt sind ausserdem die Zwischenspeicher des PPC, der Daten- und der Befehlscache. Diese sind für das Betriebssystem und die Anwendungen aktiviert. Die Untersuchungen der heterogenen Anwendungen und der Zugriff auf den gemeinsamen Speicher erfolgt ohne Cache, d.h. die AK der SW-Domäne greifen direkt auf den Hauptspeicher zu und die AK der HW-Domäne instanziieren keinen eigenen lokalen Cache. Die als HLS-Modul gekennzeichneten Blöcke bilden die AK der HW-Domäne.

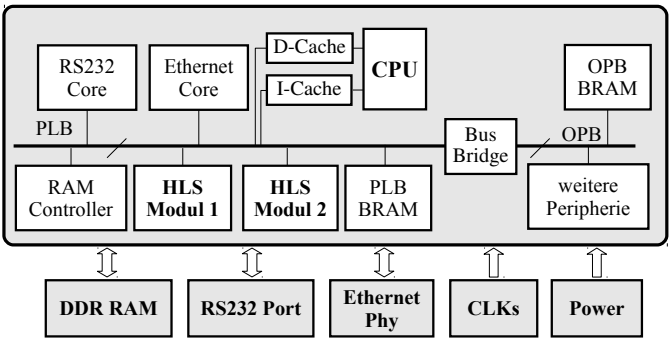


Abbildung 7.1: Realisiertes System im Virtex-II-Pro-Schaltkreis als heterogene Zielplattform

Als Betriebssystem arbeitet ein Linux mit einem Kern der Version 2.6.24, wobei die Portierung auf den *PowerPC 405* genutzt wird. Die originalen Quellen des Kernel [Lin13] wurden um Treiber für die peripheren Komponenten der EDK und eigenen Code zur Behebung gefundener Implementierungsfehler ergänzt. Als Linux-Distribution kommt *OpenWRT* [Ope13] zum Einsatz, das neben einer vollständigen Übersetzungsumgebung eine Paketverwaltung integriert, auf den verbreiteten Boot-Lader *Universal Boot Loader* (U-Boot) [DEN13] setzt und als Grundlage die *uClibc* [And11] als reduzierte Variante einer C–Bibliothek verwendet. Als C-Compilerumgebung dient der GNU-Compiler *GCC* in Version 4.4 als Cross-Compiler[1]. Die in Kapitel 6 beschriebene Laufzeit- und Systemverwaltungsumgebung wurde für den Userspace implementiert. Die Komponenten der Systemverwaltung arbeiten

[1] Cross-Compiler: Ein Compiler der auf dem Entwicklungsrechner läuft, aber Kompilate für das Zielsystem (z.B. eines PPC-Prozessors) erzeugt.

nebenläufig als Threads. Der Zugriff auf die HW-Ressourcen erfolgt mittels Speicher-Abbildung[2] (MMAP).

7.2 Automatisierung des Entwurfsablaufes

7.2.1 Synthesewerkzeuge

Als Systementwurfsumgebung für den FPGA-Schaltkreis wurde das EDK in der Version 10.1 mit dem integrierten RT-Synthesewerkzeug *Xilinx Platform Studio* (XPS) genutzt. Die partiell dynamische Rekonfiguration wurde für die prototypische Umsetzung des vorliegenden Ansatzes ausgeschlossen. Hintergrund sind die damit einhergehende Fehleranfälligkeit und die eingeschränkte Beobachtbarkeit der Implementierung zur Laufzeit. Außerdem ist der zu erwartende Erkenntnisgewinn im Hinblick auf die vielfältigen, bestehenden Forschungsarbeiten auf diesem Gebiet als gering einzuschätzen (vgl. Kapitel 2.2.3).

Bei der HLS wird der *StreamsC*-Compiler [Gok+00] bzw. die daraus entstandene kommerzielle Lösung *CoDeveloper* [PT05] eingesetzt. Beide Werkzeuge stellen den Basisumfang von HLS-Funktionen bereit, die in Abschnitt 2.2.4 eingeführt wurden. Die kommerzielle Lösung bietet neben tiefgreifenderen Optimierungen hinsichtlich Array-zu-Speicher-Abbildung und Schleifen eine erweiterte Unterstützung von Fließkommaberechnung und die automatisierte Erzeugung der HW/SW-Schnittstelle zur Anbindung an den Systembus. Es werden insbesondere die FPGA-Entwicklungswerkzeuge und die Bus-Systeme der FPGA-Hersteller (Xilinx mit PLB/AXI und Altera mit AMBA) direkt unterstützt. Darüber hinaus besitzt der CoDeveloper eine verlässliche VHDL-Exportfunktion und instanziiert dedizierte Berechnungsressourcen in den FPGAs. Auf Interna des Compilers wie beispielsweise Zwischenformate besteht jedoch kein Zugriff.

Als akademisch getriebene Prototypenentwicklung besitzt der *StreamsC*-Compiler diese Vorteile nicht. Allerdings bietet die Verfügbarkeit des Quellcodes die Möglichkeit, Anpassungen an Optimierungs- und Generierungsstufen durchzuführen. Aktuelle HLS-Werkzeuge (vgl. Anhang A) sind S*treamsC* hinsichtlich der erreichbaren Performanz (Fläche, Zeit und Quellcode-Größe) überlegen, was jedoch für den Nachweis der diskutierten Methodik keine weiteren Nachteile birgt [BRH09].

[2] *Speicher-Abbildung*: Einblendung eines physikalischen Adressbereiches in den Arbeitsspeicherbereich eines Programms (engl. Memory Mapping).

7.2.2 Anwendungsanalyse und Codeinjektion

7.2.2.1 Codeinjektion auf Hochsprachenebene

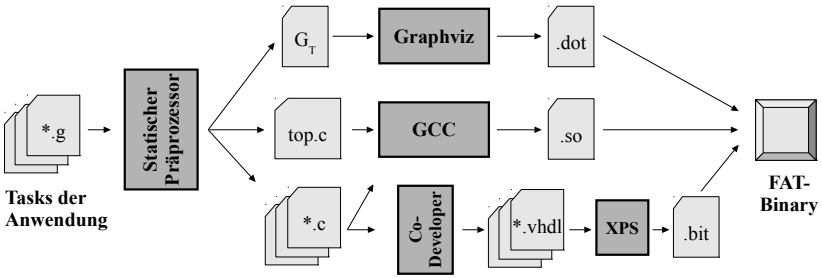


Abbildung 7.2: Werkzeugkette der Unterbrechungsintegration auf Hochsprachenebene

In Abbildung 7.2 wird der Ablauf für die Unterbrechungsintegration in der Hochsprache (vgl. Kapitel 5.3.1) dargestellt. Zentraler Bestandteil ist die Präprozessierung der C-Eingangsbeschreibung mittels statischer Codeanalyse zur Aufteilung in unterbrechbare Segmente nach Algorithmus 5.2. Zusätzlich werden Funktionsrufe an die POSIX-Thread-Bibliothek durch Systemrufe an die Laufzeitumgebung ersetzt und die Einhaltung der in Kapitel 4.2 eingeführten Syntax-Einschränkungen geprüft. Die Gesamtstruktur der Anwendung wird extrahiert und daraus der Taskgraph G_T sowie die Hauptfunktion der Anwendung erzeugt. Der Taskgraph wird im DOT-Format erzeugt, das durch die *Graphviz*-Biobliothek weiter verarbeitet werden kann. Analyse und Codeumwandlung für die Threads erfolgen auf einem abstrakten Syntaxbaum aus dem dann synthesefähiger C-Code auf Basis des Grundgerüstes aus Auflistung 5.1 exportiert wird. Die Implementierung des Präprozessors erfolgte in Python durch Erweiterung des Paketes *pycparser [Ben08]* in Version 0.7. Nach der abschließenden Codeumsetzung mittels EDK und GCC erfolgt das Archivieren des FAT-Binary mittels dem ZIP-Dateiformat.

7.2.2.2 Codeinjektion auf RT-Ebene

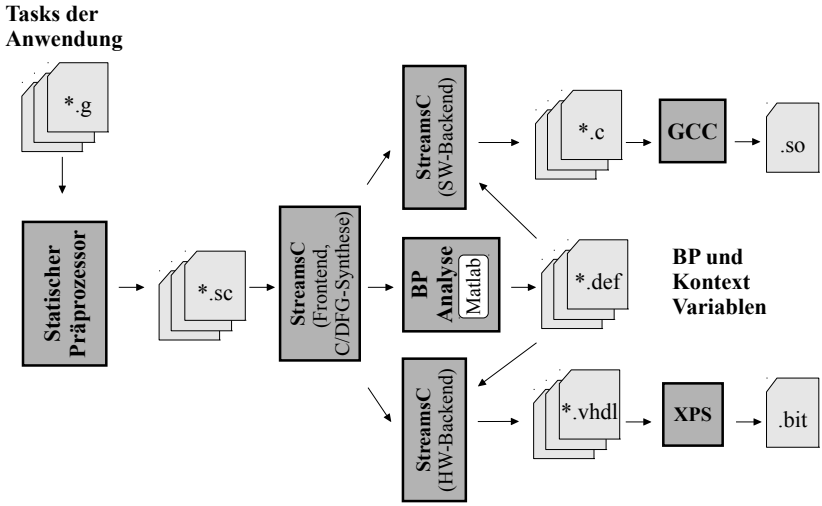


Abbildung 7.3: Werkzeugkette für die Unterbrechungsintegration auf RT-Ebene

Die Werkzeugkette für die Integration der Unterbrechungspunkte auf RT-Ebene nach Kapitel 5.3.2 unterscheidet sich im unteren Zweig und ist in Abbildung 7.3 dargestellt. Zentraler Kern der Kette ist der *StreamsC*-Compiler. Dessen Frontend und Synthesestufen erzeugen den Zwischencode, aus dem später der synthetisierte Kontroll- und Datenfluss als Graph extrahiert wird. Dabei erfolgt auch die Schleifenumwandlung nach Algorithmus 5.5. Die Unterbrechungspunkt-Analyse der Graphen nach Algorithmen 5.1 und 5.3 realisiert eine Python-basiertes Skript, welches die Nebenbedingungen nach Gleichungen 5.15 und 5.16 dem externen ILP-Solver von MATLAB [The13] übergibt. Aus dem Ergebnis erzeugt ein jeweils eigenständiges Backend den Quellcode für die HW und die SW aus dem Zwischencode des Frontends. Im HW-Backend werden die im nachfolgenden Kapitel erläuterten HW-Erweiterung generiert und instanziiert. Die abschließende Codeumsetzung und das Archivieren des FAT-Binary erfolgt analog zu Kapitel 7.2.2.1.

7.3 Hardware-Erweiterungen

7.3.1 Unterbrechungslogik und Kontextspeicher

Um eine Unterbrechungsfähigkeit der HW-Domäne zu ermöglichen sind Erweiterungen in den Rechenwerken der HW notwendig. Die grundlegende Funktion dieser Erweiterungen wurde in Kapitel 5.2.1 dargelegt und gliedert sich in einen Anteil für Unterbrechungslogik sowie operations- und steuerpfad-spezifische Anteile. Abbildung 7.4 zeigt die Erweiterungen für das Beispiel 2.2.14 auf Seite 64. Im Folgenden wird die konkrete Umsetzung dieser Erweiterungen als Basis für die Analysen der vorliegenden Arbeit vorgestellt.

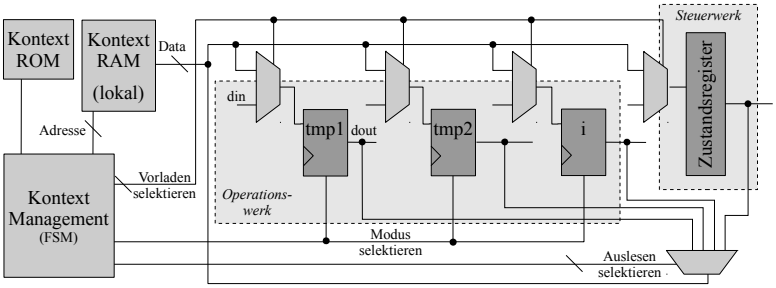


Abbildung 7.4: Unterbrechungslogik der HW-Implementierung für das Beispiel 2.2.14

Die zentrale und generische Unterbrechungslogik besteht aus dem Kontext-Management, dem lokalen Kontext-Speicher und einem Kontext-ROM. Auflistung 7.1 zeigt deren RT-Implementierung. Der ROM-Inhalt wird während der Unterbrechungsanalyse erzeugt und enthält die Zuordnung, welche Datenspeicher (Variablen) für einen spezifischen Unterbrechungspunkt kontextrelevant sind, also gesichert bzw. wiederhergestellt werden müssen. Für das Beispiels 7.4 enthält der Kontext-ROM einen Eintrag, der die Datenspeicher *i*, *tmp1* und *tmp2* dem Unterbrechungspunkt 1 zuordnet. Das Sichern und Wiederherstellen realisiert das Kontext-Management in Form eines Automaten (*LoadStoreFSM*), der auf Anfrage die relevanten Variablen aus dem Datenpfad in einen lokalen RAM, der als FIFO ausgeführt ist, schreibt bzw. von diesem liest. Über einen Multiplexer werden die entsprechenden Eingänge der Signale aus dem Datenpfad selektiert.

```
architecture rtl of preemption_hardware is
   signal bprom_addr_s     : std_logic_vector(MAX_BP_WIDTH_G +
                             MAX_PREG_PER_BP_WIDTH_G-1
                             downto 0);
   signal bprom_regaddr_s  : std_logic_vector(
                             MAX_PREG_WIDTH_G-1 downto 0);
   signal kfifo_data_i_s   : std_logic_vector(31 downto 0);
   signal stmux_addr_s     : std_logic_vector(
                             MAX_PREG_WIDTH_G-1 downto 0);

begin  -- rtl
   -- input output assignment
   preempt_regs_load_i_out <= kfifo_dout_d;
   kfifo_din_d             <= kfifo_data_i_s;

   bp_rom_1 : bp_rom
      generic map (
         MAX_PREG_WIDTH_G         => MAX_PREG_WIDTH_G,
         MAX_PREG_PER_BP_WIDTH_G => MAX_PREG_PER_BP_WIDTH_G,
         MAX_BP_WIDTH_G           => MAX_BP_WIDTH_G)
      port map (
         clk  => clk,
         rst  => reset,
         addr => bprom_addr_s,
         q    => bprom_regaddr_s);

   StoreMux_1 : StoreMux
      generic map (
         N => MAX_PREG_WIDTH_G)
      port map (
         addr     => stmux_addr_s,
         in_data  => preempt_regs_data_o_in,
         out_data => kfifo_data_i_s);

   LoadStoreFSM_1 : LoadStoreFSM
      generic map (
         MAX_PREG_WIDTH_G         => MAX_PREG_WIDTH_G,
         MAX_PREG_PER_BP_WIDTH_G => MAX_PREG_PER_BP_WIDTH_G,
         MAX_BP_WIDTH_G           => MAX_BP_WIDTH_G)
      port map (
         clk                => clk,
         reset              => reset,
         go_in              => go_in,
         done_out           => done_out,
         cnxt_size_out      => cnxt_size_out,
         load_store_sel_in  => load_store_sel_in,
         breakpoint_in      => breakpoint_in,
         bprom_addr_out     => bprom_addr_s,
         bprom_regaddr_in   => bprom_regaddr_s,
         kfifo_we_out       => kfifo_we,
         kfifo_rd_out       => kfifo_rd,
         restore_reg_select => preempt_regs_load_select_out,
         store_addr_select  => stmux_addr_s,
         status_out         => OPEN);
end rtl;
```

Auflistung 7.1 RT-Implementierung der Unterbrechungslogik

Im Operationswerk bzw. Datenpfad werden kontextrelevante Datenspeicher mit vorladbaren Registern, sogenannten *PreemptRegister* (vgl. Auflistung 7.2), ausgestattet. Das Signal *preempt_mode_in* kontrolliert den Unterbrechungsmodus des Registers, bei dem der Registerinhalt unabhängig vom Takt- bzw. Dateneingang gehalten wird. Über das Signal *load_select_in* ist das Vorladen des Inhaltes möglich. Die gegebene RT-Beschreibung wird bei der RT-Synthese zu einem Register mit vorgeschaltetem Multiplexer synthetisiert.

```
entity PreemptRegister8x1 is
   port (
      clk              : in  std_logic;
      reset            : in  std_logic;
      load_select_in   : in  std_logic; -- select "load" input
      preempt_mode_in  : in  std_logic; -- signal restore mode

      d_in             : in  std_logic_vector(7 downto 0);
      load_in          : in  std_logic_vector(7 downto 0);
      d_out            : out std_logic_vector(7 downto 0));

end PreemptRegister8x1;


-- purpose: Synthesizable RTL of the preemption register
architecture rtl of PreemptRegister8x1 is
begin  -- rtl
   register_proc : process(clk, reset)
   begin
      if (reset = '1') then
         d_out <= (others => '0');
      elsif (clk'event and clk = '1') then
         if preempt_mode_in = '1' then
            if load_select_in = '1' then
               d_out <= load_in;
            end if;
         else
            d_out <= d_in;
         end if;
      end if;
   end process;
end rtl;
```

Auflistung 7.2 Beispiel eines vorladbaren Registers für Kontextvariablen der HW-Implementierung

Erweiterungen erfolgen auch im Steuerwerk des Tasks. In die Zustandsübergänge werden zusätzliche Unterbrechungssignalprüfungen eingebaut, sodass

ggf. ein Übergang in einen Sicherungszustand (vgl. Abbildung 5.2b) erfolgen kann, der zunächst das Kontextmanagement mit dem Signal *go_in* triggert und schließlich auf die Erfolgsmeldung durch das Signal *done* wartet.

7.3.2 Strombasierte Hardware-Software Schnittstelle

Über die HW/SW-Schnittstelle wird die in Kapitel 6.3.2 beschriebene befehlsorientierte Kommunikation zwischen der Laufzeitumgebung und den AK der HW-Domäne realisiert. Die Domänen SW und HW arbeiten bei unterschiedlichen Befehlsausführungs- bzw. Berechnungsraten und Taktfrequenzen, sodass es an deren Schnittstelle einer wechselseitigen Entkopplung bedarf [Gup95].

```
entity StrmFifoRead is
 generic (
   ElemWidth : integer := 32;        -- Width in bits
   Depth     : integer := 16);       -- Depth of fifo

 port (
   -- Fifo/PE interface
   Clk    : in std_ulogic;           -- System clock in
   Reset  : in std_ulogic;           -- System reset

   -- Bus interface
   BusData     : in  std_logic_vector(ElemWidth-1 downto 0);
   FifoFull    : out std_logic;      -- fifo ready to receive
   FifoIncrEn  : in  std_logic;      -- fifo pointer increment

   -- Reader process interface
   Data      : out std_logic_vector(      -- Stream input data
                 ElemWidth-1 downto 0);
   RdEn      : in  unsigned(0 downto 0);  -- Data write enable
   RdRdy     : out unsigned(0 downto 0);  -- Ready for data
   OpenEn    : in  unsigned(0 downto 0);  -- Open stream
   CloseEn   : in  unsigned(0 downto 0);  -- Close stream
   EOStream  : out unsigned(0 downto 0);  -- End of stream
   Err       : out unsigned(0 downto 0)); -- Error flag
end StrmFifoRead;
```

Auflistung 7.3 Schnittstelle eines gepufferten und datenflussgesteuerten Stromendes in der HW-Implementierung

Dafür werden gerichtete stromorientierte Kanäle[3] eingesetzt, die sowohl eine Daten-Flusssteuerung zur Synchronisation als auch eine Datenpufferung zur Entkopplung implementieren. Auflistung 7.3 zeigt dazu beispielhaft die Entität einer Kanalinstanz in einem AK der HW-Domäne zur Kommunikation von SW in Richtung HW. SW-seitig erfolgt der Zugriff auf den Kanal über den Systembus. Lesender Zugriff erfolgt durch die *Reader-Schnittstelle*, die neben dem Datenkanal auch Kontroll-, Status- und Fehlerinformationen des Kanals bereitstellt. Lesender und schreibender Zugriff auf die Kanäle erfolgt für die HW blockierend. Die SW greift nicht blockierend auf den Kanal zu und kann dazu über den Kanalstatus den möglichen Erfolg einer anstehenden Lese- oder Schreiboperation abfragen.

7.3.3 Basisstruktur eines Ausführungskontainers in Hardware

Innerhalb der RPU wird für jeden AK eine eigene Peripheriekomponente erzeugt, die im Sprachgebrauch der Firma Xilinx als *Pcore*[4] bezeichnet wird. Ein AK der HW-Domäne besitzt eine Basisstruktur nach Abbildung 7.5, die grundlegende Steuer- und Kommunikationsfunktionalität unabhängig vom geladenen Task bereit stellt. Als Schnittstelle zum Systembus dient die PLB_IPIF-Komponente [Xil05] aus der Komponentenbibliothek der Entwurfsumgebung EDK. Dahinterliegend ermöglicht eine Registerbank der Laufzeitumgebung den Zugriff auf den Steuerblock des AK. Dieser kontrolliert die AK hinsichtlich Unterbrechungsanfragen, deren Bestätigung sowie den Reset der Task-Logik. Der Steuerblock integriert zusätzlich die in Abschnitt 7.3.1 erläuterte und in den Abbildungen 5.4 und 7.4 dargestellte Unterbrechungslogik. Für den Zugriff auf Daten und Kontextinformationen im zentralen Speicher ist die Systembusschnittstelle des AK mit Masterzugriff und direktem Speicherzugang[5] (DMA) ausgestattet. Damit kann der AK eigenständig die Arbitrierung des Busses anfordern und den Datentransfer mit Speicherkomponenten im System durchführen. Der Ressourcenverbrauch der Basisstruktur einer AK und verschiedener in der AK realisierter Tasks auf den Beispielanwendungen ist in Tabelle 7.1 gegeben.

Für dynamisch rekonfigurierbare Rechenwerke der HW-Domäne (vgl. Kapitel 2.2.3) realisiert der Steuerblock zusätzlich den Zugang zum Konfigurationsspeicher und bildet mit der PLB_IPIF-Komponente das statische Modul.

[3] *Datenstrom*: engl. data stream

[4] *Pcore (engl. Peripheral Core):* aus dem Sprachgebraucht der Xilinx-Entwurfswerkzeuge und steht für eine Peripheriekomponente zum Anschluss an einen Systembus

[5] *Direkter Speicherzugang*: (engl.) Direct Memory Access (DMA)

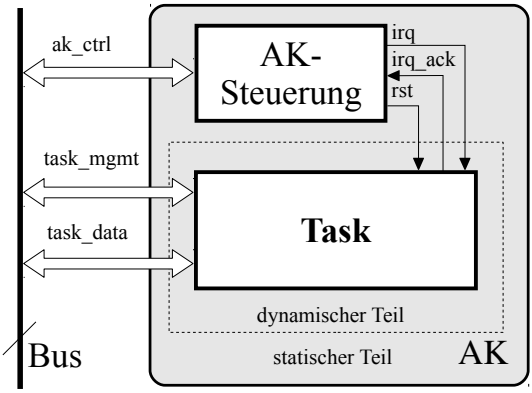


Abbildung 7.5: Basisstruktur eines AK der HW-Domäne im FPGA

Das dynamische Modul enthält ausschließlich den auszuführenden Task. Die Signalentkopplung zwischen statischem und dynamischem Modul realisieren Busmakros die ebenfalls zur Basisstruktur des AK zählen. Die vorliegende prototypische Implementierung macht aus Gründen der komplizierten Fehlerverfolgung und Beobachtbarkeit keinen Gebrauch von partiell dynamisch rekonfigurierbaren Strukturen der RPU.

	Kontroll-zustände	#BP	Kontext Variablen	Slices	Flip Flops	LUT
Basisstruktur	-	-	-	467	576	629
while_loop	10	-	-	894	1087	1387
-sequentiell	10	6	1	959	1105	1555
-pipeline	13	6	5	1071	1112	1709
partikel_filter	130	-	-	1134	1156	2365
-sequentiell	130	11	10	1764	1714	3007
-pipeline	170	7	19	2537	2218	4316

Tabelle 7.1: Ressourcenverbrauch der Basisstruktur eines AK der HW-Domäne mit verschiedenen geladenen Teilanwendungen

8 Ergebnisse

8.1 Anwendungsbeispiele

Die Untersuchungen im Rahmen der vorliegenden Arbeit erfolgten anhand von Beispielen, die in diesem Kapitel kurz erläutert und deren Eigenschaften in Bezug auf die Entwurfsmethodik charakterisiert werden.

8.1.1 Auf-/Abwärtszähler

Ein Zähler bildet das einfachste Anwendungsbeispiel, dass zur Validierung der grundlegenden Funktion von Kommunikations-, Migrations- und Task-Management genutzt wurde. Die Struktur der Anwendung besteht aus drei Teilen *SMCounter*, *SMWorker* und *SMWatcher* sowie einem gemeinsamen Speicher und einer Mutex. Der Zähler erhält seinen Anfangswert aus dem gemeinsamen Speicher. Abgesehen vom Offset zu seinem Speicherbereich im gemeinsamen Speicher bekommt der *SMCounter* seine individuelle Schrittweite und den Endwert mitgeteilt. Die *SMCounter* müssen von anderen Tasks erstellt und konfiguriert werden. Bei jedem Durchlauf der Hauptschleife versucht der Zähler den Mutex mit der ID seines Offsets per *M_Lock* Anfrage zu sperren, holt sich danach den aktuellen Wert der Zählvariable aus dem gemeinsamen Speicher, in- bzw. dekrementiert diese, schreibt sie zurück und gibt abschließend den Mutex wieder frei. In der gleichen Schleife überprüft er nach Freigabe des Mutex, ob der Scheduler eine *T_Pause* Anfrage zur Unterbrechung gesendet hat. Sollte dies so sein, pausiert er wie gefordert. Wenn der Zähler hingegen seinen Endwert erreicht hat, beendet er sich und schickt eine *T_Exit* Meldung.

SMWorker hat die Aufgabe, eine gegebene Anzahl *N SMCounter* mit dem gleichen Offset zu erstellen und zu starten. Außerdem obliegt ihm die Pflicht, den gemeinsamen Mutex der *SMCounter* zum Schutz des gemeinsamen Speichers zu initialisieren und am Ende zu zerstören. Zur Überprüfung des Fortschrittes der Zähler wird ein weiterer Task gestartet, der die Speicheradresse der Zählvariablen überwacht und in regelmäßigen Abständen über dessen

Betrag informiert, bis der Endwert erreicht wurde. Dieser *SMWatcher* kann auf Grund des *printf*() Aufrufs nur in der SW-Domäne laufen. Das Zusammenspiel aller Anwendungen zeigt Abbildung 8.1.

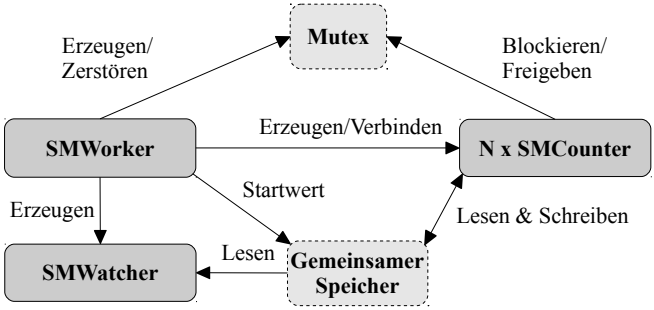


Abbildung 8.1: Tasks am Beispiel eines Auf-/Abwärtszählers

8.1.2 MJPEG Kodierung

Anwendung ist die Kodierung und Dekodierung eines Bildes zum Zweck der verlustbehafteten Reduktion der digitalen Bilddatenmenge für Speicherung und Übertragung. Der Kern Beispiels ist eine JPEG-Kodierung [ISO94]. Implementiert wurde die Erweiterung zur Bewegtbildkomprimierung als Motion-JPEG (MJPEG) in C für die HLS als Basis der Untersuchungen von Billich [Bil07]. Das Verfahren ist in Abbildung 8.2 dargestellt, wobei sich der Algorithmus in die vier Bearbeitungsschritte Farbraumkonvertierung, Kosinustransformation, Quantisierung und Huffman-Kodierung für die Komprimierung sowie deren Inverse für die Dekomprimierung gliedert. Nach dem Einlesen des Bildes werden die Farbkomponenten der Chrominanz *Cb* und *Cr* durch Unterabtastung (4:2:0) auf ein Viertel ihrer ursprünglichen Datenmenge reduziert. Im Anschluss wird das Bild durch Kosinustransformation in den Frequenzbereich gebracht. Hierbei kommt eine HW geeignete Ganzzahl-Kosinustransformation zum Einsatz [AK00]. Dabei findet eine blockweise (8x8) Dekorrelation der voneinander abhängigen Pixel statt. Die anschließende Quantisierung fasst hochfrequente Anteile, die vom menschlichen Auge in natürlichen Bildern kaum unterschieden werden, zusammen. Die letzte Stufe ist die Entropiekodierung nach *Huffman* durch Zick-Zack-Abtastung des 8x8-Datenblocks. In der Dekodierungsstrecke werden diese

Verfahren umgekehrt. Das Verfahren ist insbesondere durch Unterabtastung der Farben und Quantisierung verlustbehaftet.

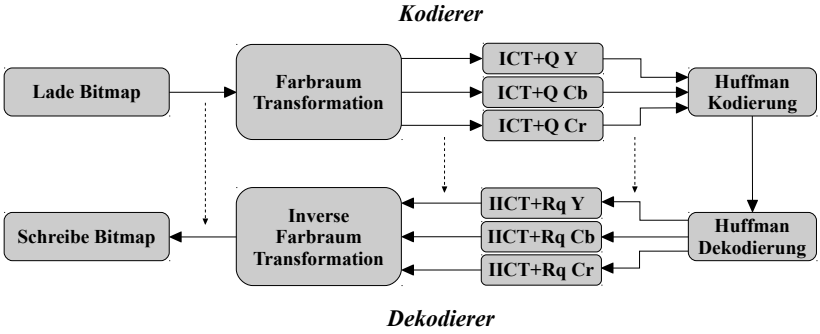


Abbildung 8.2: Bearbeitungsschritte des JPEG-Komprimierungsverfahren

Eine Stufe arbeitet unabhängig mit den Ergebnissen der vorherigen Stufe, sodass eine parallele Bearbeitung durch eine Pipeline möglich ist. Entsprechend kapselt die Implementierung jeweils eine Stufe als einzelnen Task, der flexibel zwischen Rechenwerken des heterogenen Systems verschoben werden kann. Untersuchungen zur günstigen HW/SW-Partitionierung der Bearbeitungskette sowie deren dynamischen Partitionierung auf Basis hochsprachenintegrierter Unterbrechungspunkte wurden in [BRH08] beschrieben.

8.1.3 Partikel-Filter

Im Folgenden wird das Anwendungsbeispiel eines Partikel-Filters als Technik zur Lokalisierung eines beweglichen Objektes vorgestellt, das im Rahmen des Projektes Generalisierte Plattform zur Sensordatenverarbeitung (GPSV) der BMBF-Initiative InnoProfile an der Professur Schaltkreis- und Systementwurf untersucht wurde. Im Projekt wird der Filter zur Schätzung der Position eines mobilen Funkknoten im Raum genutzt, wobei Distanzinformationen zu stationären Referenzpunkten, sogenannten Ankern, mit bekannter Position verarbeitet werden. Die entsprechenden Arbeiten wurden unter anderem in [Fro+08] veröffentlicht. Dieses Verfahren ähnelt dem *Global Positioning System* (GPS), das Pseudo-Entfernungen zu Satelliten als Basis nutzt.

Der Partikelfilter ist eine Implementierung des Bayes-Filter-Algorithmus, bei dem die neue Verteilung der Schätzerzustände aus einer zufälligen Auswahl

von Elementen der aktuellen Verteilung approximiert wird. Im vorliegenden Fall besteht der zu schätzende Zustand aus der unbekannten Position (x,y,z) des Mobilfunkknotens, sodass sich der Filter mit M Partikeln nach Gleichung 8.1 ergibt. Jedes Partikel stellt dabei eine mögliche Position des Mobilknotens dar. Mit jeder eintreffenden Distanzmessung wird der Filterzustand erneuert und die geschätzte Position verfeinert. Die Partikel konzentrieren sich schließlich nach einer hinreichenden Anzahl Messungen in einem Bereich, in dem sich die tatsächliche Position des mobilen Knotens befindet.

$$\mathbf{p}^{[m]} = (x^{[m]}, y^{[m]}, z^{[m]})^T, m = 1..M \tag{8.1}$$

Eine Erneuerung des Filterzustandes zum Zeitpunkt t besteht aus den folgenden Schritten:

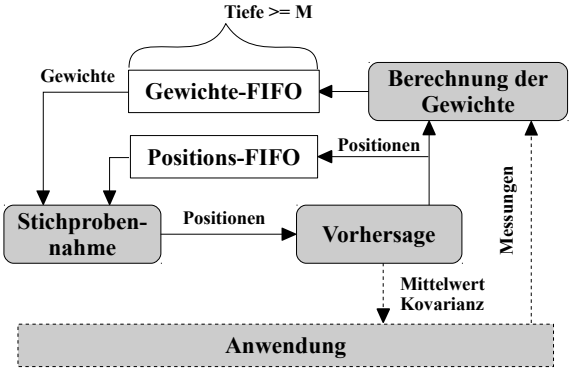
1. *Vorhersage.* Für jedes Partikel wird eine hypothetische Position $\mathbf{p}_t^{[m]}$ zum aktuellen Zeitpunkt t auf Basis der vorherigen Position $\mathbf{p}_{t-1}^{[m]}$ ermittelt. Jedes neue Partikel wird aus einer angenommenen Verteilung entsprechend eines Bewegungsmodells ermittelt.

2. *Gewichtsberechnung.* Der nächste Schritt besteht aus der Berechnung eines Gewichtes $\mathbf{w}_t^{[m]}$ für jedes Partikel $\mathbf{p}_t^{[m]}$ unter Einbeziehung der Distanz d_t zwischen dem Objekt und einer Ankerposition p_a. Das Gewicht ergibt sich nach den Gleichungen 8.2 und 8.3, wobei Δd die Differenz zwischen gemessener und erwarteter Distanz darstellt. Die Skalierungskonstante k charakterisiert die Qualität der Distanzinformation im Wertebereich zwischen 1 und 0.

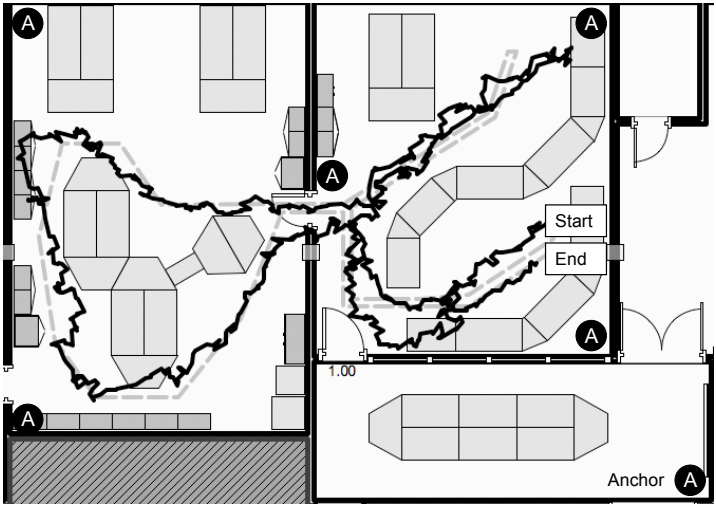
$$w_t^{[m]} = \frac{k}{k + |\triangle d|}, k > 0 \tag{8.2}$$

$$\Delta d = d_t - |\mathbf{p}_t^{[m]} - \mathbf{p}_a| \tag{8.3}$$

3. *Stichprobennahme.* Das abschließende Set von Partikeln wird durch eine Auswahl aus dem hypothetischen Set von Schritt 1 gebildet. Die Wahrscheinlichkeit der Auswahl eines Partikels in das finale Set ergibt sich aus dessen Gewicht, sodass die weniger wichtigen Partikel aussortiert werden. Entsprechend fokussiert das resultierende Set auf Regionen mit hoher Aufenthaltswahrscheinlichkeit. In dieser Implementierung wurde ein *Low Variance Sampler* nach [TBF05] realisiert.



(a) Blockschema der Partikelfilterimplementierung



(b) Ergebnisse in einer Indoor-Testumgebung mit 6 Ankern

Abbildung 8.3: Partikelfilter in Lokalisierungsanwendung mit einer mittleren Schätzgenauigkeit von 65 cm [Fro+10; Lan+11]

Das Blockschema der High-Level Implementierung zeigt Abbildung 8.3a. In Abbildung 8.3b ist das Ergebnis eines Probelaufes für ein Objekt in einer Testumgebung bestehend aus 6 Ankern dargestellt. Die schwarze ZickZack-Linie stellt den geschätzten Verlauf der realen Bewegung (graue Linie) dar. Die Implementierung greift für die neuen Partikel im Vorhersage-Schritt auf das Kartenmaterial inklusive Möblierung insofern zurück, als dass neue Partikel nicht in möblierte Bereiche oder Wände gesetzt werden. Zu weiterführenden Details der Indoor-Lokalisierung sei aus Platzgründen auf [Fro+10; Lan+11] verwiesen. [TBF05] bietet eine umfassende Diskussion der Grundlagen von Partikelfiltern.

8.2 Funktionsnachweis und Schedulinganalyse

Für die Funktionsanalyse, die Analyse des System-Schedulers und die Abschätzung der Umschaltdauer zwischen den Rechenwerken wurden Versuche mit dem Anwendungsbeispiel des Auf-/Abwärtszählers durchgeführt. Dieses kleine Beispiel ist besonders für die Analyse geeignet, weil das Verhalten einfach und klar strukturiert ist. Eine mögliche Unterbrechung des Zählers erfolgt an genau einem Unterbrechungspunkt mit einer Latenz von genau einem Takt und einem Kontext, der mit nur 96 Bit aus dem Zählerwert, der Schrittweite und des Endwertes besteht. Die Ergebnisse der Versuche werden entsprechend maßgeblich vom System (HW-Struktur, Betriebssystem, Laufzeitumgebung) und Entscheidungen der Ablaufplanung beeinflusst und lassen sich daher in gleicher Form auf die komplexen Anwendungsbeispiele übertragen.

Im Versuch wurden für zwei Varianten V1 und V2 die Laufzeiten auf der Zielplattform ermittelt. In beiden Varianten werden vier Aufwärtszähler mit jeweils um ein erhöhter Schrittweite gestartet, die bis zum Wert von einer Milliarde zählen. Entsprechend wird der vierte Zähler die wenigsten Durchläufe benötigen. Der Unterschied zwischen V1 und V2 liegt im Verzicht auf das Scheduling. V1 wird nicht unterbrochen, sodass das Rechenwerk erst bei Beendigung eines Zählers freigegeben wird. Für den Versuch wurden zwei Rechenwerke in der HW-Domäne der Zielplattform aktiviert. Die resultierenden Laufzeiten sind in Tabelle 8.1 gegeben. Für V1 ergeben sich für die Zähler unterschiedliche Startzeiten, weil die Rechenwerke nacheinander belegt werden. Bei V2 werden hingegen alle Zähler zum Zeitpunkt t=0 gestartet und im Round-Robin Verfahren zwischen den Rechenwerken verteilt.

		Cntr 1	Cntr 2	Cntr 3	Cntr 4	ges.	# Wechsel
V1	Start	0	0	15	25		0
	Ende	30	15	25	32,5	32,5	
	Laufzeit	30	15	10	7,5	32,5	
V2	Start	0	0	0	0		37
	Laufzeit	38,5	24	19	15,5	38,5	

Tabelle 8.1: Laufzeit des Auf-/Abwärtszähler auf der Zielplattform in Sekunden

Der Laufzeitvergleich zeigt mit 6 Sekunden eine um 18 % längere Laufzeit von V2 gegenüber V1. Der naheliegenden Vermutung einer durch die Unterbrechungen und den notwendigen Kontextwechsel verursachten Verluste bei V2 kann bei Betrachtung der effektiven Belegung der Rechenwerke entgegnet werden. Mit 62,5 Sekunden ist die effektive Berechnungszeit für V1 und V2 identisch. In Abbildung 8.4 ist die Auslastung der beiden Rechenwerke für den Versuch dargestellt und es zeigt sich, dass der ungenutzte Zeitraum von Modul 2 für die längere Rechenzeit verantwortlich ist und die Umschaltdauer zwischen Rechenwerken unterhalb der zeitlichen Auflösung des Versuchsaufbaus von 100 Millisekunden liegen muss. Adäquate Versuche wurden mit weiteren Kombinationen von Rechenwerken aus beiden Domänen durchgeführt, womit der Nachweis über die Funktion der dynamischen Verteilung von Berechnungen auf dem heterogenen ACS geführt werden konnte.

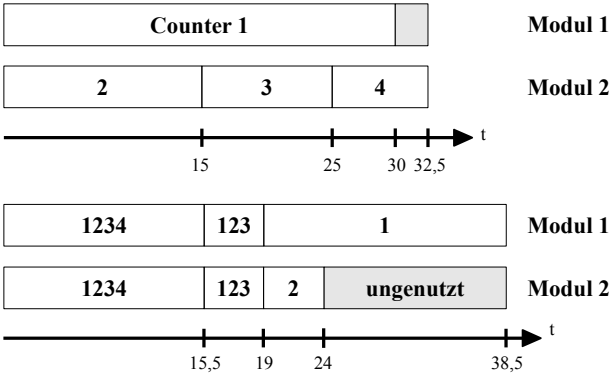


Abbildung 8.4: Auslastung der Rechenwerke für den Auf-/Abwärtszähler

Für eine genauere Bestimmung der Umschaltdauer und insbesondere auch der zeitlichen Kosten bei Domänenwechsel wurden die Versuche abgewandelt und jeweils drei Zähler in einer identischen Schrittweite sowie einer kürzeren Zeitscheibe zur Provokation häufiger Wechsel genutzt. Dabei wurden bei einem Laufzeitunterschied von 5,8 Millisekunden 89 Wechsel durchgeführt, sodass der zeitliche Verzug im Rahmen weniger Mikrosekunden (hier 64 μs) angenommen werden kann. Der Domänenwechsel ergab sich mit 68 Millisekunden eine Größenordnung über dem innerhalb der HW-Domäne. Als Ursache für diese Verzögerung konnte das Scheduling innerhalb der pThread-Bibliothek ausgemacht werden, die sowohl die AK der SW-Domäne als auch die Laufzeitumgebung als einzelne pThreads erzeugt und ausführt. Der Beschleunigungsfaktor, der für den Zähler durch die Rechenwerke der HW-Domäne im Vergleich zu denen der SW-Domäne erreicht wurde, beträgt 9,86. Weitere Details zu den Versuchen können in [Bil08] nachgelesen werden.

8.3 Analyse der Unterbrechungspunktverteilung

Die im Kapitel 5 beschriebenen Verfahren zur Unterbrechung und Migration von Teilanwendungen wurden anhand der Anwendungsbeispiele aus Kapitel 8.1 analysiert und evaluiert. Für die Verteilung von Unterbrechungspunkten auf RT-Ebene nach 5.3.2 wurden Beispiele mit unterschiedlichem Kontrollfluss beleuchtet. Zwei davon besitzen überschaubare Komplexität: das Beispiel 2.2.14 *while_ loop* auf Seite 64 und das Beispiel *cascaded_for_if* einer Schleife mit verzweigtem Kontrollfluss im Schleifenkörper. Mit einem Partikelfilter (vgl. Kapitel 8.1.3) und verschiedenen Stufen einer MJPEG-Kodierung (vgl. Kapitel 8.1.2) wurden komplexe Anwendungen untersucht. Der Partikelfilter ist von zyklischem Ablauf gekennzeichnet, wohingegen bei der Kosinustransformation oder dem Huffman-Kodierer der MJPEG-Kette mehrschichtige Verzweigungen im Kontrollfluss enthalten sind.

Tabelle 8.2 zeigt für diese Beispiele die Ergebnisse der Unterbrechungspunktverteilung auf RT-Ebene (vgl. Kapitel 5.3.2) unter Einhaltung der in Spalte 3 gegebenen maximalen Dispatchlatenz. Dabei wurde die Ressourcengewichtung zu $w_1 = 10$, $w_2 = 3$ und $w_3 = 1$ angenommen und von einer Bitrate $R_b = 16$ zur Speicherung des Kontextes ausgegangen. Die zweite Spalte gibt für die Kernberechnung der Beispiele die Anzahl der Quellcode-Zeilen an. Die Spalten 4-6 zeigen die Anzahl Unterbrechungspunkte, die Anzahl der Variablen, die in irgendeinem Unterbrechungspunkt als Kontext identifiziert wurden und die damit zusammenhängende Anzahl Bit. Die letzte Spalte enthält die Laufzeit des ILP-Solver in Sekunden. Die Erzeugung des Problems

Beispiel	**LOC**	L_D	**# BP**	**# Var**	**# Bit**	**Laufzeit** Solver
while_loop	9	5	4/10 (40%)	2/8 (25%)	40/201 (20%)	< 1s
cascaded_ _for_if	16	5	3/11 (27%)	4/10 (40%)	66/165 (40%)	< 1s
partikel_ filter	68	40	11/130 (8%)	10/120 (8%)	243/2846 (8%)	19s
MJPEG						
- *ColTrans*	98	100	5/41 (12%)	5/49 (10%)	74/1008 (7%)	3s
- *InvColTrans*	100	100	4/50 (8%)	3/47 (6%)	18/938 (2%)	2s
- *CosTrans*	315	100	10/432 (2%)	9/335 (3%)	58/6869 (1%)	104s
- *InvCosTrans*	211	100	11/387 (3%)	11/75 (15%)	98/1040 (9%)	50s
- *Huffman*	515	100	17/648 (3%)	11/60 (18%)	97/669 (14%)	57s
- *InvHuffman*	533	100	10/567 (2%)	0/19 (0%)	0/332 (0%)	3s

Tabelle 8.2: Ergebnis von vier Anwendungsbeispielen für die Verteilung von Unterbrechungspunkten nach Abschnitt 5.3.2 mit den Parametern: $w_1 = 10, w_2 = 3$ und $w_3 = 1$, einer Bitrate $R_b = 16$ und der maximalen Dispatchlatenz L_D

überstieg bei keinem der Beispiele eine Sekunde. Für den ILP-Solver wurde die Funktion *bintprog* aus der MATLAB Optimierungs-Toolbox The MathWorks Inc. [The13] genutzt. Alle Berechnungen wurden auf einem 2.6 GHz Dual-Core Prozessor mit 16 GB Hauptspeicher durchgeführt.

Für eine weiterführende Einschätzung der Methodik wurden Verläufe für das Partikelfilter-Design aufgenommen. Für unterschiedliche Werte der maximalen Dispatchlatenz L_D wurden die optimalen Unterbrechungspunktmengen bestimmt. Abbildung 8.5 zeigt die entsprechenden Ergebnisse. Für Latenzen die kleiner als 17 Takte liegen, konnte keine Lösung gefunden werden. Für Latenzen die größer/gleich 32 liegen, werden keine weiteren Verbesserungen hinsichtlich der Anzahl notwendiger Unterbrechungspunkte oder hinsichtlich der notwendigen HW-Ressourcen gefunden. Die Verarbeitungszeit des Sol-

ver für den Partikelfilter ist in Abbildung 8.6 gegeben. Sie schwankt in einer Breite von 10 Sekunden bis zu 17 Minuten. Es zeigt sich, dass die Laufzeit des Solver für die Werte von L_D am größten ist, die etwas höher sind als die kleinste Latenz, die gefunden werden kann. Die Zeit, die für die Generierung des Problems benötigt wird steigt zunächst mit L_D, übersteigt jedoch den zweistelligen Millisekundenbereich selbst für sehr große Latenzen nicht.

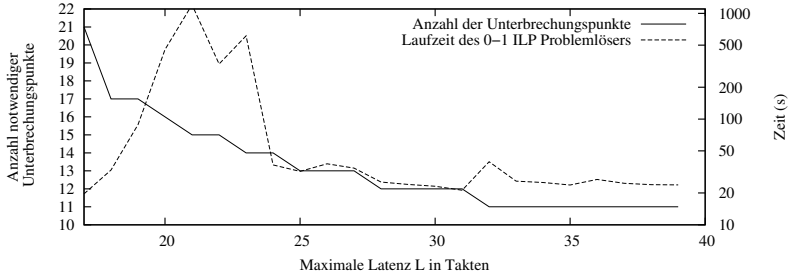


Abbildung 8.5: Unterbrechungspunktmengen und Laufzeit des ILP-Solver für unterschiedliche Dispatchlatenzen anhand der Beispielanwendung *Partikelfilter*

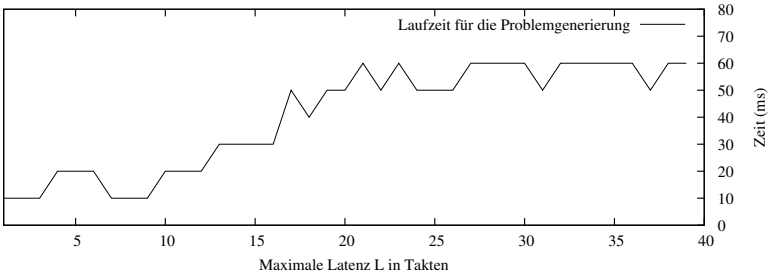


Abbildung 8.6: Laufzeit der Problemgenerierung für die Beispielanwendung *Partikelfilter*

Die Ergebnisse der Untersuchungen hinsichtlich der Verteilung von Unterbrechungspunkten in schleifenzentrischen Anwendungen zeigt Tabelle 8.3. Dargestellt sind die Ausführungszeiten für sequentielle Schleifenimplementierungen und die durch Pipelining optimierte Form. Die Spalten 3 und 4 zeigen die Anzahl eingefügter Unterbrechungspunkte und zugehöriger Kon-

textvariablen unter Einhaltung der Obergrenze für die L_D, die in Spalte 2 gegeben ist. In den Spalten 5 und 6 sind die Ausführungszeiten der nicht aktiv unterbrochenen SW-Implementierung gezeigt, wobei die Spalte 6 die Umsetzung des reduzierten Kontrollflussgraphen nach Algorithmus 5.4 repräsentiert. Die Ausführung der SW-Implementierung erfolgte ebenfalls auf einem 2.6 GHz Dual-Core Prozessor mit 16 GB Hauptspeicher. Die Hauptschleife der *while_loop*-Implementierung wurde 10^7–mal ausgeführt. Der Partikelfilter iteriert über 8192 Partikel. Die Spalte 7 zeigt die Taktzyklen die bei der Ausführung der HW-Implementierung notwendig sind.

	L_D	#BP	#Var	t_{ex} der SW-Implementierung (in Sekunden)		#Takte der HW-Implementierung
				CFG abgebildet auf FSM	Reduzierter CFG abgebildet auf FSM	
while_loop (10^7 Iterationen)						
-sequentiell	6	3/10 (30%)	1/8 (20%)	0.12	0.06	$70 * 10^6$
-pipeline	6	4/13 (30%)	5/7 (17%)	0.05	0.03	$25 * 10^6$
partikel_filter (8192 Partikel)						
-sequentiell	40	11/130 (8%)	10/120 (8%)	0.205	0.180	$32 * 10^4$
-pipeline	40	7/170 (4%)	19/117 (16%)	0.165	0.123	$25 * 10^4$

Tabelle 8.3: Ausführungszeiten und Unterbrechungspunkte verschiedener Schleifenimplementierungen für die Beispiele *while_loop* und *partikel_filter*

Die Ergebnisse zeigen für alle Designs die erwarteten Geschwindigkeitsverbesserungen durch die Optimierung des HLS-Werkzeuges mittels Fließbandverarbeitung. Sowohl die HW-Implementierung als auch die SW-

Implementierung erreicht eine beschleunigte Ausführung. Für den Partikel-Filter ist diese weniger stark, weil in dessen Hauptschleife nur 3 unabhängige Kontrollschritte durch die HLS identifiziert werden und die Pipeline entsprechend kurz ausfällt. Bemerkenswert ist, dass der relative Geschwindigkeitsgewinn in der SW höher ist, als der Gewinn bei der HW-Implementierung.

Bezüglich Unterbrechungspunkten und Kontextvariablen wird durch die Fließbandoptimierung letztendlich eine Mehrfachnutzung der Unterbrechungslogik erzielt. Bei gleicher Obergrenze führt dies bei gleichbleibender Dispatchlatenz zu weniger Unterbrechungspunkten. Diese Auswirkungen sind besonders für die Pipeline mit kleinen Initiierungsintervallen zu beobachten. Die Anzahl der kontextrelevanten Variablen steigt dabei allerdings, da mit einem Zustand, der einen potentiellen Unterbrechungspunkt darstellt, eine größere Anzahl Operationen assoziiert wird.

8.4 Gewichtsfaktoren zur Unterbrechungspunktverteilung

Der Ressourcenbedarf R ist in Kapitel 5.2.1 als wesentlicher Optimierungsfaktor bei der Verteilung von Unterbrechungspunkten theoretisch eingeführt und klassifiziert worden. Die Gewichtsfaktoren $w_{1..4}$ wurden eingeführt, um deren Einfluss bei der Optimierung der Unterbrechungspunktverteilung zu berücksichtigen. Diese Faktoren sind in Abhängigkeit der konkreten Implementierung eines HW-Rechenwerks zu bestimmen.

In Kapitel 7.3 ist die Umsetzung der HW-Erweiterungen auf der prototypischen Zielplattform beschrieben. Mit Kenntnis des konkreten Ressourcenbedarfs ist es möglich, diese Konstanten zu bestimmen und durch deren Angabe Rückschlüsse auf die tatsächliche Verteilung der HW-Kosten zu ziehen. Als Lösungsansatz ergibt sich, aus Gleichung 5.9 für vier unterschiedliche Unterbrechungspunktverteilungen BP_m mit $m \in \mathbb{N} \mid 1 \leq m \leq 4$, das in Gleichung 8.4 gezeigte lineare Gleichungssystem.

$$\begin{pmatrix} R_{idx}(BP_0) \\ \vdots \\ R_{idx}(BP_m) \end{pmatrix} = f(W) = \begin{pmatrix} r_{cost_1}(BP_0) & \dots & r_{cost_4}(BP_0) \\ \vdots & & \vdots \\ r_{cost_1}(BP_m) & \dots & r_{cost_4}(BP_m) \end{pmatrix} \begin{pmatrix} w_1 \\ \vdots \\ w_4 \end{pmatrix} \tag{8.4}$$

Der konkrete Ressourcenverbrauch eines AK der HW-Domäne lässt sich nach der Synthese für den Virtex-II-Pro-FPGA auf der Basis notwendiger LUT, FlipFlops, dedizierter Multipilikationseinheiten (MUL) und Block-RAM-Zellen angeben. Für den nominalen Vergleich des Bedarfes und damit der

quantitativen Bestimmung der Matrize auf der linken Seite des Gleichungssystems 8.4 ist eine Wichtung der Ressourcenanteile im FPGA notwendig. Als Vergleichsmaß wird die benötigte Fläche als Flächenindex R_{idx} herangezogen, der auf einem Verhältnis von *1 | 1 | 32 |* 96 für *LUT | FlipFlop | MUL | Block-RAM* basiert. Diese Werte können für den Virtex-II-Pro als realistisch angenommen werden [Xil11]. In Tabelle 8.4 sind der Ressourcenbedarf und der Flächenindex für vier unterschiedliche Unterbrechungspunktverteilungen des Partikelfilters mit jeweils 130 Kontrollzuständen angegeben. Die Einzelanteile der HW-Kosten r_{cost1} bis r_{cost4} der Implementierungen sind ebenfalls in Tabelle 8.4 angegeben. Nach den Gleichungen 5.4, 5.5, 5.7 und 5.8 aus Kapitel 5.2 lassen sich die charakteristischen Aufwände für eine konkrete Verteilung von Unterbrechungspunkten so beschreiben.

Eigenschaften der Implementierungen		Variante 1	Variante 2	Variante 3	Variante 4
Eigenschaften	Dispatch-latenz	17	21	22	28
	Anzahl Unterbrechungs-punkte	21	15	15	12
	Anzahl Kontext-variablen	26	19	16	11
FPGA-Ressourcen	FlipFlops	2149	2051	1955	1744
	LUT	4550	3967	3586	2954
	MUL	6	3	3	3
HW-Kosten	R_{idx}	6796	6114	5637	4794
	r_{cost1}	21	15	15	15
	r_{cost2}	832	608	512	384
	r_{cost3}	256	352	320	63
	r_{cost4}	126	65	63	45

Tabelle 8.4: Ressourcenverbrauch unterschiedlicher Implementierung des Partikelfilters unter Variation der Latenz

Mit den gegebenen Werten in Tabelle 8.4 können schließlich die Gewichtsfaktoren der Einzelaufwände für die vorgestellten HW-Erweiterungen auf der Zielplattform des Virtex-II-Pro-FPGA ermittelt werden. Der Ergebnisvektor als Lösung des linearen Gleichungssystems ergibt sich dann wie folgt:

$$w_1 = 119,95 \quad w_2 = 2,35 \quad w_3 = 7,67 \quad w_4 = 2,82 \ .$$

Der Vergleich der Faktoren zeigt, dass die Anzahl der Unterbrechungspunkte die größte Auswirkung auf die benötigten HW-Ressourcen besitzt und damit den Erweiterungen des Kontrollpfades der Großteil der Kosten zuzuordnen ist. Die Gesamtgröße kontextrelevanter Werte hat einen deutlich kleineren Einfluss auf die Aufwände der HW. Die unterbrechungspunktbezogenen Kontextgrößen sind kaum relevant.

8.5 Fazit

Es kann konstatiert werden, dass mit dem vorgeschlagenen Entwurfsverfahren auf der algorithmischen Abstraktionsebene Anwendungen für ein heterogenes ACS entwickelt werden können und dabei eine dynamische Partitionierung über die HW/SW-Grenze hinweg realisierbar ist. Es bestehen in diesem Zusammenhang keine Einschränkungen sowohl für einfach strukturierte Anwendungen, als auch für Anwendungen mit komplexem und verschachteltem Kontrollfluss. Mit vertretbarem Aufwand im Entwurfsprozess lässt sich auch für mehrfach verschachtelte Kontrollstrukturen ein geeigneter Satz von Unterbrechungspunkten automatisiert bestimmen.

Die zusätzlichen Strukturen für die Unterbrechungsfähigkeit der generierten Implementierungen erzeugen für beide Domänen zwar vernachlässigbare Performanzeinbußen, jedoch fallen die damit verbundenen Ressourcenaufwände deutlich ins Gewicht. Einerseits ermöglicht die Mehrfachnutzung der Unterbrechungslogik eine Entkopplung der zusätzlichen Ressourcen von der Größe und Komplexität der Anwendung, sodass die HW-Kosten anteilig sinken. Andererseits zeigt die Wichtung der Einflussfaktoren auf die HW-Kosten am realisierten Prototyp, dass die Anzahl der Unterbrechungspunkte überproportionalen Einfluss hat. Eine adäquate Wahl der unteren Schranke für die Dispatchlatenz ist deshalb notwendig um eine effiziente Implementierung großer Berechnungskerne zu realisieren. Diese Kerne lassen in Anbetracht des zeitlichen Verzugs durch die Umlagerung zwischen Rechenwerken erwarten, die Flexibilität eines heterogenen ACS zur Laufzeitadaption bestmöglich auszunutzen.

9 Schluss

9.1 Zusammenfassung

In dieser Arbeit wurde eine Entwurfsmethodik vorgestellt, die es erlaubt, dynamisch partitionierbare Anwendungen für ein heterogenes adaptives Computersystem zu entwickeln. Diese Computersysteme besitzen unterschiedliche Rechenwerke in den Domänen Hardware und Software, deren Aufgaben dynamisch verändert werden können. Das System ist damit in der Lage, sich an Anforderungen zur Laufzeit anzupassen und damit stets an einem optimierten Arbeitspunkt hinsichtlich Parametern, wie zum Beispiel Performanz, Energieaufnahme oder Fehlertoleranz, zu wirken.

Die Anwendungsentwicklung für diese heterogenen adaptiven Systeme ist eine komplexe Aufgabe mit eng vermaschten Entwurfsabläufen aus den Gebieten der Hardware- und Software-Entwicklung. Durch die Anhebung der Entwurfsabstraktion auf ein Niveau, das unabhängig von der Ausführungsdomäne ist, eröffnet sich das Potential eines übergreifenden Anwendungsentwurfes mit durchdringender Integration beider Entwurfswelten. Vorliegend wurde die Abstraktion auf der verhaltensbasierten Systemebene vorgeschlagen und darauf aufbauend ein Virtualisierungsansatz der Rechenwerke mittels Ausführungskontainern verfolgt. Ausgehend von einer einzigen in parallele Ausführungsstränge gegliederten Beschreibung wurden Anwendungen realisiert, deren Teile auf sämtlichen Rechenwerken des heterogenen Systems ablauffähig sind und zwischen diesen migrieren können.

Die vorliegende Dissertation beschäftigt sich mit der Fragestellung, inwieweit dieses Entwurfsverfahren automatisiert auf Basis eines High-Level-Synthese-Ansatzes realisierbar ist und dabei die Unterbrech- und Migrierbarkeit der nebenläufigen Anwendungsteile umsetzbar erscheint. In diesem Zusammenhang wurden Algorithmen vorgestellt und untersucht, die eine automatisierte Verteilung von Unterbrechungspunkten, unter Berücksichtigung minimaler Zusatzaufwände hinsichtlich Ressourcen und Ausführungszeit, erlauben.

Die im Rahmen der Arbeit entstandene Erweiterung des High-Level-Synthese-Werkzeuges *StreamsC* zur Unterbrechungspunktverteilung und die prototypische Laufzeitumgebung für das heterogene adaptive Computersystem

wurden mittels Anwendungsbeispielen unterschiedlicher Komplexität untersucht und validiert. Anhand einer FPGA-Prototypenplattform konnte die Realisierbarkeit der vorgeschlagenen Methodik und die Funktion der Implementierung nachgewiesen werden. In diesem Zusammenhang wurden auch die Laufzeit der avisierten Unterbrechungspunktsynthese und der zusätzliche Ressourcenverbrauch quantifiziert.

9.2 Ausblick

Weiterführende Arbeiten sollten die Unterbrechungspunktverteilung verbessern. Insbesondere im Hinblick auf die Ergebnisse der Implementierung und das nun bestehende Wissen zu Typen und Gewichtung der zusätzlichen Ressourcenaufwände, ließe sich der Algorithmus optimieren. Zusätzlich könnte die Abwägung weiterer Parameter erfolgen. Zum Beispiel durch eine Bewertung der Performanzunterschiede von Unterbrechungspunktsätzen. Diese werden insbesondere in der SW-Domäne auftreten, da dort die Unterbrechungsprüfung sequentiell erfolgt und entsprechende Auswirkungen auf die Ablaufzeit bestehen.

Bisher wurden im Rahmen der Arbeit zunächst nur einfache Verfahren für die Ablaufplanung der Anwendungsteile auf den heterogenen Rechenwerken eingesetzt. Die Fragestellung der Ablaufplanung stellt einen eigenen Forschungsschwerpunkt dar und sollte insbesondere im Hinblick auf quantifizierbare maximale Dispatchlatenz, als auch die systemspezifische Migrationszeit untersucht werden. Dabei sollten auch die Laufzeiteigenschaften von Anwendungsteilen auf den unterschiedlichen Rechenwerken Berücksichtigung finden. Ansätze zur Evaluation dieser Eigenschaften bietet das statische und das dynamisches Profiling der Anwendungsteile.

In Bezug auf die Anwendungsbeschreibung mit threadbasierten parallelem Ausführungsmodell wurden sich in dieser Arbeit auf die pThread-Bibliothek beschränkt. Einen weiteren Ansatz zur Abbildung von Threads in der HW-Domäne zeigen Cabrera, Martorell, Gaydadjiev, Ayguade und Jiménez-González [Cab+09] durch den Einsatz der OpenMP-Bibliothek. Die Untersuchung der Nutzbarkeit dieses Ausführungsmodells für den vorliegenden Ansatz erscheint vielversprechend und wird Gegenstand zukünftiger Arbeiten sein.

Verbesserungspotential besteht bei der Umsetzung der Laufzeitumgebung. Es existieren vielfältige Freiheitsgrade, die von der Realisierung im Kernel-

oder Userspace bis hin zu einer verteilten Implementierung über mehrere Rechenwerke reichen. In der vorliegenden Arbeit wurden diese nicht tiefgründig betrachtet und ausgewertet. Weiterführende Arbeiten sollten die Frage beantworten, wo die Optima im beschriebenen Entwurfsraum liegen. Darüber hinaus bestehen Optimierungsspielräume beim eingesetzten Betriebssystem, der Basis der Laufzeitumgebung. Eine vielversprechende Möglichkeit stellt die direkte Einbeziehung der Multiprozessorverwaltung, in das Management der SW Domäne dar. Alternativ könnte mit einem minimalistischen Betriebssystemansatz der systemverwaltungsbedingte Zusatzaufwand reduziert werden.

Die prototypische Implementierung des heterogenen Computersystems integriert ein einschichtiges Speichersystem und sämtliche Speicherhierarchien hinsichtlich Cache oder Auslagerungsspeicher sind für die Anwendungen nicht verfügbar. Ein mehrschichtiges Speichersystem lässt jedoch deutliche Performanzgewinne erwarten. Insbesondere der Zugriff auf den gemeinsamen Speicherbereich würde von lokalem Cache am Rechenwerk profitieren. Das bedingt jedoch dem Einsatz eines entsprechenden Kohärenzprotokolls und dessen Realisierung für die virtualisierten Ausführungskontainer. Sollte die vorliegende Umsetzung über den Funktionsnachweis hinaus Verwendung finden, erscheint das Schließen dieser Lücke als zwingend.

Intensive Untersuchungen sind für echtzeitkritische Anwendungen im Zusammenhang mit dynamischer HW/SW-Partitionierung notwendig. Für weiche Echtzeitanforderung erfolgt zwar im Zusammenhang mit dem Verteilungsalgorithmus bereits insofern eine Beachtung, als dass mit der oberen Schranke für die Dispatchlatenz ein deterministischer Zeitansatz für die Belegung der Rechenwerke gegeben ist. Eine Berücksichtigung weiterer notwendiger Aspekte für ein echtzeitfähiges System erfolgten zur Begrenzung der Untersuchungskomplexität zunächst nicht. Insbesondere sind Fragestellungen in Bezug auf Unterbrechungssperre, Ablaufplanung und das bereits genannte rechenwerkspezifische Profiling der Anwendungsteile, zu beleuchten.

A Werkzeuge zur High-Level-Synthese

A.1 Kommerzielle Werkzeuge

Werkzeugname	Institution	Eingabe	Ausgabe	Bemerkungen
CatapultC [Men10]	Mentor Graphics/-Calypto	C++	VHDL, Verilog	Eines der ersten großen kommerziellen Werkzeuge
C2H [Alt09]	Altera	C	VHDL	Integriert in die SOPC-Builder Entwicklungsumgebung
C2R [Ahu+09]	CebaTech	C	Verilog	
Cynthesiszer [For08]	Forte Design Systems	SystemC	Verilog	Umsetzung von SystemC Transaktions-Level Modellen
C-To-Silicon [Cad08]	Cadence	SystemC	VHDL, Verilog	
Vivado/-AutoPilot [Con08]	Xilinx	C	VHDL, Verilog	Kommerzieller Nachfolger von xPilot heute integrierter Teil der Xilinx Werkzeuge
CoDeveloper [PT05]	Impulse	C	VHDL, Verilog	Kommerzieller Nachfolger von StreamsC
Handle-C [Lim05]	Celoxica/-Agility	C	VHDL	Wurzeln liegen im Oxford University Computing Laboratory
Bach C [Kam+01]	Sharp	C/C++	VHDL	

Werkzeugname	Institution	Eingabe	Ausgabe	Bemerkungen
Bluespec [HA99]	Bluespec	System Verilog	Verilog	Umsetzung von Eigenschaften mittels "Term-Rewriting"
CyberWork-Bench [Wak99]	NEC	C	Verilog	Entwicklung von NEC
Behavorial Compiler [Kna96]	Synopsys	VHDL	VHDL	Erstes kommerzielles EDA-Werkzeug zur HLS

A.2 Akademische Ansätze und Werkzeuge

Werkzeugname	Institution	Eingabe	Ausgabe	Bemerkungen
CASH [Ven+04]	Carnegie Mellon University	C	asyn-chrones Verilog	Synthese von asynchronen Schaltungen
DWARV [Yan+07]	Delft University of Technology	C	VHDL	Compiler für HW-Kerne der MOLEN-Rechenarchitektur
GAUT [Mar+93]	Université de Bretagne-Sud	C	VHDL, Sys-temC	
HardwareC [De +90]	Stanford University	C		
Liquid Metal [Hua+08]	IBM Research	LIME (Java)	VHDL	
Ocapi-xl [Van+01]	IMEC Institut Belgien	C++	Verilog, VHDL	
ROCCC [Vil+10]	University of California Riverside	C		

Werkzeugname	Institution	Eingabe	Ausgabe	Bemerkungen
Single Assignment-C [Naj+03]	California University	C		
SPARK	University of California	C	VHDL	
SpecC [Gaj+00]	SpecC Technology Open Consortium	C based		
StreamsC [Gok+00; FGL01]	Los Alamos National Labratories	C	VHDL	
Trident Compiler [TGP07]	University of California	C	VHDL	Java basiertes Compiler Framework spezialisiert für Fließpunkt Algorithmen
Transmogrifier C	University of Toronto	C		

Literatur

[AC87] T. Agerwala und J. Cocke. *High performance reduced instruction set processors*. Techn. Ber. IBM Computer Science, 1987.

[Act13] Actel Semiconductor Corporation. 2013. URL: `http://www.actel.com` (besucht am 24. 06. 2013).

[AG96] Sarita V. Adve und Kourosh Gharachorloo. "Shared Memory Consistency Models: A Tutorial". In: *IEEE Computer Magazine* 29.12 (1996), S. 66–76. ISSN: 0018-9162.

[Ahu+09] S. Ahuja, S.T. Gurumani, C. Spackman und S.K. Shukla. "Hardware Coprocessor Synthesis from an ANSI C Specification". In: *IEEE Design & Test of Computers* 26.4 (Juli 2009), S. 58–67. ISSN: 0740-7475.

[AK00] M.D. Adams und F. Kossentni. "Reversible integer-to-integer wavelet transforms for image compression: performance evaluation and analysis". In: *IEEE Transactions on Image Processing* 9.6 (2000), S. 1010–1024. ISSN: 1057-7149.

[Ake09] Doug Akers. *An Innovative Approach to Managing Software Requirements*. Techn. Ber. MKS Inc., 2009.

[Alt09] Altera Inc. *Nios II C2H Compiler User Guide*. 2009.

[Alt13] Altera Inc. *Quartus Design Suite*. 2013. URL: `http://www.altera.com/products/software` (besucht am 01. 10. 2013).

[Amd67] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *AFIPS '67: Proceedings of the spring joint computer conference*. Atlantic City, New Jersey: ACM, Apr. 1967, S. 483–485.

[And11] Erik Andersen. *uClibc - Projekt*. 2011. URL: `http://www.uclibc.org` (besucht am 15. 04. 2011).

[AR03] S.M. Aziz und Marko Rößler. "A parameterisable VHDL model of an avionics data bus". In: *TENCON '03: Proceedings of the IEEE Conference on Convergent Technologies for Asia-Pacific Region*. Bd. 4. Okt. 2003, S. 1604–1608.

[Atm13] Atmel Corporation. 2013. URL: `http://www.atmel.com` (besucht am 09. 03. 2013).

[Aue+10] Joshua Auerbach, David F. Bacon, Perry Cheng und Rodric Rabbah. "Lime: a Java-compatible and synthesizable language for heterogeneous architectures". In: *ACM SIGPLAN Notices* 45.10 (Okt. 2010), S. 89–108. ISSN: 0362-1340.

[Ave95] Jürgen Avenhaus. *Reduktionssysteme*. Springer, 1995.

[Bal+88] M. Balakrishnan, A.K. Majumdar, D.K. Banerji, J.G. Linders und J.C. Majithia. "Allocation of multiport memories in data path synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 7.4 (Apr. 1988), S. 536–540. ISSN: 0278-0070.

[Ben08] Eli Bendersky. 2008. URL: `http://pypi.python.org/pypi/pycparser` (besucht am 01. 10. 2013).

[BH98] P. Bellows und B. Hutchings. "JHDL - an HDL for reconfigurable systems". In: *FCCM '98: Proceedings of the 6th IEEE Symposium on Field-Programmable Custom Computing Machines*. Apr. 1998, S. 175–184.

[Bil07] Enrico Billich. "Implementierung eines Motion-JPEG En- und Decoders mit Hilfe des High-Level C-Synthese Werkzeug CoDeveloper". Studienarbeit. Technische Universität Chemnitz, Sep. 2007.

[Bil08] Enrico Billich. "HW/SW-übergreifendes Preemptive Multitasking auf einem Configurable System on Chip mit dem High-Level C-Synthese Werkzeug CoDeveloper". Diplomarbeit. Technische Universität Chemnitz, Sep. 2008.

[Bob07] Christophe Bobda. *Introduction to Reconfigurable Computing*. Springer, 2007.

[Bre96] Gordon J. Brebner. "A Virtual Hardware Operating System for the Xilinx XC6200". In: *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*. London, UK: Springer, 1996, S. 327–336. ISBN: 3-540-61730-2.

[BRH08] Enrico Billich, Marko Rößler und Ulrich Heinkel. “Optimale HW/SW-Partitionierung einer MPEG-Codierung mit dem High-Level ESL-Werkzeug CoDeveloper”. In: *DASS ’08: Tagungsband der Dresdner Arbeitstagung Schaltungs- und Systementwurf*. Dresden, Mai 2008, S. 79–84. ISBN: 3-9810287-2-4.

[BRH09] Enrico Billich, Marko Rößler und Ulrich Heinkel. “HW/SW-übergreifendes Multithreading auf einem Configurable System on Chip mit High-Level C-Synthese Werkzeug "CoDeveloper"”. In: *MST ’09: Tagungsband der 9. Chemnitzer Fachtagung Mikrosystemtechnik - Mikromechanik & Mikroelektronik*. Nov. 2009, S. 39–44. ISBN: 978-3-00-029135-7.

[BRH10] Enrico Billich, Marko Rößler und Ulrich Heinkel. “Effiziente Auslastung der heterogenen Ressourcen eines Systems durch domain-übergreifendes Multithreading”. In: *MBMV ’10: Tagungsband des 13. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen*. 2010, S. 127–136.

[Cab+09] Daniel Cabrera, Xavier Martorell, Georgi Gaydadjiev, Eduard Ayguade und Daniel Jiménez-González. “OpenMP Extensions for FPGA Accelerators”. In: *SAMOS ’09: Proceedings of the 9th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. Samos, Greece: IEEE Press, 2009, S. 17–24. ISBN: 978-1-4244-4502-8.

[Cad08] Cadence Design Systems Inc. *Cadence C-to-Silicon Compiler Delivers On The Promise Of High-level Synthesis*. online. 2008.

[CH02] Katherine Compton und Scott Hauck. “Reconfigurable computing: a survey of systems and software”. In: *ACM Computing Surveys* 34.2 (2002), S. 171–210. ISSN: 0360-0300.

[CHW00] Timothy J. Callahan, John R. Hauser und John Wawrzynek. “The Garp Architecture and C Compiler”. In: *IEEE Computer* 33 (2000), S. 62–69. ISSN: 0018-9162.

[CJP07] Barbara Chapman, Gabriele Jost und Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. Cambridge: The MIT Press, 2007. ISBN: 0262533022, 9780262533027.

[CMD62] Fernando J. Corbató, Marjorie Merwin-Daggett und Robert C. Daley. “An experimental time-sharing system”. In: *AIEE-IRE ’62 (Spring): Proceedings of the spring joint computer conference*. San Francisco, California: ACM, 1962, S. 335–344.

[Com12] International Roadmap Commitee. *International Technology Roadmap for Semiconductors*. Techn. Ber. ITRS, 2012.

[Con08] J. Cong. "A new generation of C-base synthesis tool and domain-specific computing". In: *SOCC '08: Proceedings of the IEEE International SOC Conference*. 2008, S. 386–386.

[CS12] D. Chen und D. Singh. "Invited paper: Using OpenCL to evaluate the efficiency of CPUS, GPUS and FPGAS for information filtering". In: *FPL '12: Proceedings of the 22nd International Conference on Field Programmable Logic and Applications*. 2012, S. 5–12.

[CSG99] David E. Culler, Jaswinder Pal Singh und Anoop Gupta. *Parallel Computer Architecture*. Morgan Kaufmann Publishers, 1999.

[Dan+12] Martin Danek, Leos Kafka, Luks Kohout, Jaroslav Skora und Roman Bartosinski. *UTLEON3: Exploring Fine-Grain Multi-Threading in FPGAs*. Springer, 2012. ISBN: 1461424097.

[Dan04] Klaus Danne. "Operating Systems for FPGA Based Computers and Their Memory Management". In: *ARCS '04: Proceedings of the Workshop on Organic and Pervasive Computing*. Bd. P-41. GI-Edition Lecture Notes in Informatics (LNI). GI LNI. Bonn: Köllen Verlag, März 2004, S. 195–204.

[De +90] G. De Micheli, D. Ku, F. Mailhot und T. Truong. "The Olympus synthesis system". In: *IEEE Design & Test of Computers* 7.5 (Okt. 1990), S. 37–53. ISSN: 0740-7475.

[De 94a] G. De Micheli. "Computer-aided hardware-software codesign". In: *IEEE Micro* 14.4 (Aug. 1994), S. 10–16. ISSN: 0272-1732.

[De 94b] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. 1st. McGraw-Hill Higher Education, 1994. ISBN: 0070163332.

[DEN13] DENX Software Engineering. *Universal Boot Loader (U-Boot) - Projekt*. 2013. URL: `http://www.denx.de/wiki/U-Boot` (besucht am 15. 04. 2013).

[Die+00] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck und B. Schmidt. "Dynamic scheduling of tasks on partially reconfigurable FPGAs". In: *Computers and Digital Techniques, IEE Proceedings* - 147.3 (Mai 2000), S. 181–188. ISSN: 1350-2387.

[Dil+07] Tom Dillon, Jeremy Paatela, Guenter Dannoritzer und Scott Hussong. "Accelerating Algorithm Implementation in FPGA/ASIC Using Python". In: *HPEC '07: Proccedings of the 11th Workshop on High Performance Embedded Computing*. 2007.

[DMP06] K. Danne, R. Mühlenbernd und M. Platzner. "Executing Hardware Tasks on Dynamically Reconfigurable Devices Under Real-Time Conditions". In: *FPL '06: Proceedings of the 16th International Conference on Field Programmable Logic and Applications*. Aug. 2006, S. 1–6.

[DSN05] Tudor Dumitraş, Deepti Srivastava und Priya Narasimhan. "Architecting Dependable Systems III". In: Hrsg. von Rogério Lemos, Cristina Gacek und Alexander Romanovsky. Berlin, Heidelberg: Springer, 2005. Kap. Architecting and implementing versatile dependability, S. 212–231. ISBN: 3-540-28968-2, 978-3-540-28968-5.

[DW99] Oliver Diessel und Grant Wigley. *Opportunities for Operating Systems Research in Reconfigurable Computing*. Techn. Ber. University of South Australia, 1999.

[EHB93] R. Ernst, J. Henkel und T. Benner. "Hardware-software cosynthesis for microcontrollers". In: *IEEE Design & Test of Computers* 10.4 (Dez. 1993), S. 64–75. ISSN: 0740-7475.

[Ein13] Bundesverband Technik des Einzelhandels e.V. *Consumer Electronics Markt Index*. Techn. Ber. 2013.

[Fei97] Dror G. Feitelson. *Job Scheduling in Multiprogrammed Parallel Systems*. Techn. Ber. 1997.

[FGL01] Jan Frigo, Maya Gokhale und Dominique Lavenier. "Evaluation of the Streams-C C-to-FPGA compiler: an applications perspective". In: *FPGA '01: Proceedings of the ACM/SIGDA 9th international symposium on Field Programmable Gate Arrays*. Monterey, California, United States: ACM, 2001, S. 134–140. ISBN: 1-58113-341-3.

[Fin10] Michael Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corp., Mai 2010.

[Fly97] David Flynn. "AMBA: Enabling Reusable On-Chip Designs". In: *IEEE Micro Magazine* 17.4 (1997), S. 20–27. ISSN: 0272-1732.

[For08] Forte Design Systems Inc. *Cynthesizer Datasheet*. online. 2008.

[Fre13] Free Software Foundation, Inc. *Advantages of GNU Mach*. 2013. URL: `http://www.gnu.org/software/hurd/microkernel/mach/gnumach.html` (besucht am 06. 10. 2013).

[Fro+08] Daniel Froß, Jan Langer, Marko Rößler und Ulrich Heinkel. "Tracking of Mobile Nodes in Sensor Networks". In: *Novel Algorithms and Techniques In Telecommunications, Automation and Industrial Electronics*. Hrsg. von Tarek Sobh, Khaled Elleithy, Ausif Mahmood und MohammadA. Karim. Springer Netherlands, 2008, S. 438–443. ISBN: 978-1-4020-8736-3.

[Fro+10] Daniel Froß, Jan Langer, André Froß, Marko Rößler und Ulrich Heinkel. "Hardware Implementation of a Particle Filter for Location Estimation". In: *IPIN '10: Proceedings of the Indoor Positioning and Indoor Navigation Conference*. Zurich, Switzerland: IEEE Computer Society, 2010, S. 1–6.

[Gaj+00] D. D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer und S. Zhao. *SpecC: Specification Language and Methodology*. Boston: Kluwer Academic Publishers, März 2000, S. 328. ISBN: 0-7923-7822-9.

[Gaj+92] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu und Steve Y.-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Hrsg. von Daniel D. Gajski. Norwell, MA, USA: Kluwer Academic Publishers, 1992. ISBN: 0-7923-9194-2.

[Gil93] Wolfgang K. Giloi. *Rechnerarchitektur*. Springer, 1993.

[Gir84] E. Girczyc. "Automatic Generation of Microsequenced. Data Paths to Realize ADA Circuit Descriptions". Magisterarb. Carleton University, 1984.

[Gir87] E. Girczyc. "Loop Winding – A Data Flow Approach to Functional Pipelining". In: *ISCAS '87: Proceedings of the IEEE International Symposium on Circuits and Systems*. Mai 1987, S. 382–385.

[GJ90] Michael R. Garey und David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990. ISBN: 0716710455.

[GK83] D.D. Gajski und R.H. Kuhn. "New VLSI Tools". In: *IEEE Computer* 16.12 (Dez. 1983), S. 11–14. ISSN: 0018-9162.

[GKM04] Susan L. Graham, Peter B. Kessler und Marshall K. McKusick. "gprof: a call graph execution profiler". In: *ACM SIGPLAN Notices* 39 (Apr. 2004), S. 49–57. ISSN: 0362-1340.

[GLS99] S. Guccione, D. Levi und P. Sundararajan. "Jbits: A java-based interface for reconfigurable computing". In: *MAPLD '99: Proceedings of the Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference*. 1999.

[GM91] Rajesh K. Gupta und Giovanni De Micheli. *VULCAN - A System for High-Level Partitioning of Synchronous Digital Circuits*. Techn. Ber. CSL-TR-471. Stanford University, 1991.

[Goe13] Richard Goering. *Samsung DAC 2013 Keynote: EDA, Semis 'Not Well Prepared' for Next Mobile Revolution*. online. Juni 2013. URL: `http://www.cadence.com/Community/blogs/ii/archive/2013/06/06/samsung-dac-2013-keynote-eda-semis-not-well-prepared-for-next-mobile-revolution.aspx` (besucht am 03. 09. 2013).

[Gok+00] Maya B. Gokhale, Janice M. Stone, Jeff Arnold und Mirek Kalinowski. "Stream-Oriented FPGA Computing in the Streams-C High Level Language". In: *FCCM '00: Proceedings of the 8th IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2000, S. 49. ISBN: 0-7695-0871-5.

[Gup+06] Sumit Gupta, Gaurav Singh, Rajesh Gupta und Sandeep Shukla. "EDA for IC System Design, Verification, and Testing". In: Hrsg. von Luciano Lavagno Grant Martin und Louis Scheffer. CRC Prentice Hall, 2006. Kap. Parallelizing High-Level Synthesis: A Code Transformational Approach to High-Level Synthesis, pages.

[Gup+91] Smeeta Gupta, Robert M. Perlman, Thomas W. Lynch und Brian D. Mcminn. "Normalizing pipelined floating point processing unit". 5058048. Okt. 1991.

[Gup95] Rajesh Kumar Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. Kluwer Academic Publishers, 1995.

[HA99] James C Hoe und Arvind. "Hardware Synthesis from Term Rewriting Systems". In: *IFIP Conference Proceedings*. Bd. 162. 1999, S. 595–619.

[Har02a] Wolfram Hardt. "Integration von Verzögerungszeit-Invarianz in den Entwurf von eingebetteter Systeme". Habilitationsschrift. Universität Paderborn, Fachbereich Mathematik/Informatik, 2002.

[Har02b] Reiner Hartenstein. “Reconfigurable Computing: urging a revision of basic CS curricula (keynote address)”. In: *ICSENG ’02: Proceedings of the 15th International Conference on Systems Engineering*. Las Vegas, USA, Aug. 2002.

[Har08] Matthias Harder. “Abbildung des Posix-Thread-Modells auf ein Configurable System on Chip für das High-Level C-Synthese Werkzeug CoDeveloper”. Diplomarbeit. Technische Universität Chemnitz, Apr. 2008.

[Hei+00] Ulrich Heinkel, Martin Padeffke, Werner Haas, Thomas Buerner, Herbert Braisz, Thomas Gentner und Alexander Grassmann. *The VHDL Reference*. John Wiley & Sons, 2000.

[Hei+09] Andreas Heinig, Jochen Strunk, Wolfgang Rehm und Heiko Schick. “ACCFS – Operating System Integration of Computational Accelerators Using a VFS Approach”. In: *Reconfigurable Computing: Architectures, Tools and Applications*. Hrsg. von Jürgen Becker, Roger Woods, Peter Athanas und Fearghal Morgan. Bd. 5453. Lecture Notes in Computer Science. 10.1007/978-3-642-00641-8_44. Springer, 2009, S. 374–379.

[Hew13] Hewlett Packard. *HP Quality Center Website*. 2013. URL: `http://www.hp.com/go/qualitycenter` (besucht am 08. 10. 2013).

[HHW90] Reiner W. Hartenstein, Alexander G. Hirschbiel und M. Weber. “Xputers: An Open Family of Non-Von Neumann Architectures”. In: *Architektur von Rechensystemen, Tagungsband, 11. ITG/GI-Fachtagung*. Berlin, Germany: VDE-Verlag GmbH, 1990, S. 45–58. ISBN: 3-8007-1688-7.

[Hin+06] Heiko Hinkelmann, Andreas Gunberg, Peter Zipf, Leandro Soares Indrusiak und Manfred Glesner. “Multitasking Support for Dynamically Reconfigurable Systems”. In: *FPL ’06: Proceedings of the International Conference on Field Programmable Logic and Applications*. 2006.

[HL01] E. Horta und J. W. Lockwood. *PARBIT: a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (FPGAs)*. Techn. Ber. WUCS-01-13. Washington University in Saint Louis, Department of Computer Science, Juli 2001.

[HM04] Göran Herrmann und Dietmar Müller. *ASIC - Entwurf und Test*. Carl Hanser Verlag, 2004.

[Hof13] Felix Hoffmann. *Youm: Samsung stellt flexibles Display vor*. online. Jan. 2013. URL: `http://www.computerbild.de/artikel/cb-News-Handy-Youm-Samsung-zeigt-flexibles-Display-8047539.html` (besucht am 03. 09. 2013).

[Hop06] Bernhard Hoppe. *Modellbildung für Synthese und Verifikation*. München: Oldenbourg, 2006.

[HP07] John L. Hennessy und David A. Patterson. *Computer Architecture - A Quantitative Approach*. Elsevier Science Publishers, 2007.

[HP83] L.J. Hafer und A.C. Parker. "A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 2.1 (Jan. 1983), S. 4–18. ISSN: 0278-0070.

[HRX12] Kecheng Hao, Sandip Ray und Fei Xie. "Equivalence checking for behaviorally synthesized pipelines". In: *DAC '12: Proceedings of the 49th Annual Design Automation Conference*. DAC '12. San Francisco, California: ACM, 2012, S. 344–349. ISBN: 978-1-4503-1199-1.

[Hua+08] Shan Shan Huang, Amir Hormati, David F Bacon und Rodric Rabbah. "Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary". In: *Object-Oriented Programming (ECOOP)*. Springer, 2008, S. 76–103.

[Hun+09] Hillery C. Hunter, Erik M. Nystrom, Daniel A. Connors und Wen-mei W. Hwu. "Hardware-Compiler Co-Design for Adjustable Data Power Savings". In: *Microprocessors and Microsystems* 33.4 (2009), S. 244–253. ISSN: 0141-9331.

[HVH99] E. Hwang, F. Vahid und Yu-Chin Hsu. "FSMD functional partitioning for low power". In: *DATE '99: Proceedings of the Conference on Design, Automation and Test in Europe*. 1999, S. 22–28.

[IBM12] IBM Corporation. *Getting Started with Rational DOORS*. Techn. Ber. 2012.

[Inc08] IBM Inc. *MPI Programming Guide*. Bd. Version 4, Release 3.2. 2008.

[Ins] Institute of Electrical and Electronics Engineers. *UNIX Specification, Version 4 (IEEE Std 1003.1)*. Institute of Electrical and Electronics Engineers.

[Ins94] Institute of Electrical and Electronics Engineers. *P1003.1: Standard for Information Technology – Portable Operating System Interface (POSIX)*. New York, Dez. 1994.

[ISO94] ISO. *ISO/IEC 10918-1:1994: Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines*. Geneva, Switzerland: International Organization for Standardization, 1994, S. 182.

[Jr+80] Edward G. Coffman Jr., M. R. Garey, David S. Johnson und Robert Endre Tarjan. "Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms". In: *SIAM Journal on Computing* 9.4 (1980), S. 808–826.

[JTW07] S. Jovanovic, C. Tanougast und S. Weber. "A Hardware Preemptive Multitasking Mechanism Based on Scan-path Register Structure for FPGA-based Reconfigurable Systems". In: *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*. Aug. 2007, S. 358–364.

[Kam+01] T. Kambe, A. Yamada, K. Nishida, K. Okada, M. Ohnishi, A. Kay, P. Boca, V. Zammit und T. Nomura. "A C-based synthesis system, Bach, and its application". In: *ASP-DAC '01: Proceedings of the Asia and South Pacific Design Automation Conference*. 2001, S. 151–155.

[Kas05] Nico Kasprzyk. "COMRADE - Ein Hochsprachen-Compiler für Adaptive Computersysteme". Diss. Technischen Universität Braunschweig, März 2005.

[KBT08] D. Koch, C. Beckhoff und J. Teich. "ReCoBus-Builder – A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAS". In: *FPL '08: Proceedings of the 18th International Conference on Field Programmable Logic and Applications*. Sep. 2008, S. 119–124.

[KD90] D. Ku und G. De Micheli. "High-level synthesis and optimization strategies in Hercules and Hebe". In: *Euro ASIC '90*. Mai 1990, S. 124–129.

[KHN97] Rainer Kress, Reiner W. Hartenstein und Ulrich Nageldinger. "An operating system for custom computing machines based on the Xputer paradigm". In: *FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*. London, UK: Springer, 1997, S. 304–313. ISBN: 3-540-63465-7.

[KHT07] Dirk Koch, Christian Haubelt und Jürgen Teich. "Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation". In: *FPGA '07: Proceedings of the ACM/SIGDA 15th international symposium on Field Programmable Gate Arrays*. Monterey, California, USA: ACM, 2007, S. 188–196. ISBN: 978-1-59593-600-4.

[KK05] Andreas Koch und Nico Kasprzyk. "High-Level-Language Compilation for Reconfigurable Computers". In: *ReCoSoC '05: Proceedings of the International Conference on Reconfigurable Communication-centric SoCs*. Montpellier, France, Juni 2005.

[KKP06] Kyosun Kim, Ramesh Karri und Miodrag Potkonjak. "Micropreemption synthesis: an enabling mechanism for multitask VLSI systems". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.1 (Jan. 2006), S. 19–30. ISSN: 0278-0070.

[KKP97] Kyosun Kim, Ramesh Karri und Miodrag Potkonjak. "Micropreemption synthesis: an enabling mechanism for multi-task VLSI systems". In: *ICCAD '97: Proceedings of the IEEE/ACM international conference on Computer-aided design*. San Jose, California, United States: IEEE Computer Society, 1997, S. 33–38. ISBN: 0-8186-8200-0.

[KL79] H.T. Kung und C.E Leiserson. "Sparse Matrix Proceedings 1978". In: Hrsg. von I. S. Duff und G. W. Stewart. Society for Industrial und Applied Mathematics, 1979. Kap. Systolic Arrays (for VLSI), S. 256–282.

[KM03] David Koufaty und Deborah T. Marr. "Hyperthreading Technology in the Netburst Microarchitecture". In: *IEEE Micro* 23.2 (2003), S. 56–65. ISSN: 0272-1732.

[Kna96] David W. Knapp. *Behavioral synthesis: digital system design using the synopsys behavioral compiler*. Upper Saddle River, NJ, USA: Prentice Hall, 1996. ISBN: 0-13-569252-0.

[Koc+04] D. Koch, A. Ahmadinia, C. Bobda und H. Kalte. “FPGA architecture extensions for preemptive multitasking and hardware defragmentation”. In: *FPT ’04: Proceedings of the IEEE International Conference on Field-Programmable Technology*. Dez. 2004, S. 433–436.

[Koc+07] D. Koch, C. Haubelt, T. Streichert und J. Teich. “Modeling and Synthesis of Hardware-Software Morphing”. In: *ISCAS ’07: Proceedings of the IEEE International Symposium on Circuits and Systems*. 2007, S. 2746–2749.

[Kon+09] G.K. Konstadinidis, M. Tremblay, S. Chaudhry, M. Rashid, P.F. Lai, Y. Otaguro, Y. Orginos, S. Parampalli, M. Steigerwald, S. Gundala, R. Pyapali, L.D. Rarick, I. Elkin, Y. Ge und I. Parulkar. “Architecture and Physical Implementation of a Third Generation 65 nm, 16 Core, 32 Thread Chip-Multithreading SPARC Processor”. In: *IEEE Journal of Solid-State Circuits* 44.1 (Jan. 2009), S. 7–17. ISSN: 0018-9200.

[KP05] H. Kalte und M. Porrmann. “Context saving and restoring for multitasking in reconfigurable systems”. In: *FPL ’05: Proceedings of the 15th International Conference on Field Programmable Logic and Applications*. Aug. 2005, S. 223–228.

[Lan+07] Jan Langer, Vasco Jerinić, Ulrich Heinkel und Frank Dresig. “SpiritEd: A Register Specification System integrating IP-XACT and Adobe FrameMaker”. In: *IP Based Electronic System Conference (IP)*. Grenoble, France, 2007.

[Lan+11] Jan Langer, Daniel Froß, Enrico Billich, Marko Rößler und Ulrich Heinkel. “Multi-Level Synthesis on the Example of a Particle Filter”. In: *SPL ’11: Proceedings of the Southern Conference on Programmable Logic - Designer Forum*. 2011.

[Lat13] Lattice Semiconductor Corporation. 2013. URL: `http://www.latticesemi.com/` (besucht am 09. 03. 2010).

[LC91] K. Li und K.-H. Cheng. “Job scheduling in a partitionable mesh using a two-dimensional buddy system partitioning scheme”. In: *Parallel and Distributed Systems, IEEE Transactions on* 2.4 (Okt. 1991), S. 413–422. ISSN: 1045-9219.

[Lee+03] T.K. Lee, A. Derbyshire, W. Luk und P.Y.K. Cheung. “High-level language extensions for run-time reconfigurable systems”. In: *FPT ’03: Proceedings of the IEEE International Conference on Field-Programmable Technology*. Dez. 2003, S. 144–151.

[Lev+00] L. Levinson, R. Männer, M. Sessler und H. Simmler. "Preemptive Multitasking on FPGAs". In: *FCCM '00: Proceedings of the 8th IEEE Symposium on Field-Programmable Custom Computing Machines*. FCCM '00. Washington, DC, USA: IEEE Computer Society, 2000, S. 301–302. ISBN: 0-7695-0871-5.

[LGH94] James Laudon, Anoop Gupta und Mark Horowitz. "Interleaving: a multithreading technique targeting multiprocessors and workstations". In: *ACM SIGPLAN Notices* 29 (Nov. 1994), S. 308–318. ISSN: 0362-1340.

[Lim05] Celoxica Limited. *Handle-C Language Reference Manual*. online. 2005.

[Lin13] Linux Kernel - Projekt. 2013. URL: `http://www.kernel.org` (besucht am 15. 04. 2013).

[LK07] H. Lange und A. Koch. "An Execution Model for Hardware/Software Compilation and its System-Level Realization". In: *FPL '07: Proceedings of the 17th International Conference on Field Programmable Logic and Applications*. Aug. 2007, S. 285–292.

[Lod+06] Andrea Lodi, Claudio Mucci, Massimo Bocchi, Andrea Cappelli, Mario De Dominicis und Luca Ciccarelli. "A Multi-Context Pipelined Array for Embedded Systems". In: *FPL '06: Proceedings of the 16th International Conference on Field Programmable Logic and Applications*. IEEE Computer Society, 2006, S. 1–8.

[LP07] E. Lübbers und M. Platzner. "ReconOS: An RTOS Supporting Hard- and Software Threads". In: *FPL '07: Proceedings of the 17th International Conference on Field Programmable Logic and Applications*. Aug. 2007, S. 441–446.

[LP08] E. Lübbers und M. Platzner. "A portable abstraction layer for hardware threads". In: *FPL '08: Proceedings of the 18th International Conference on Field Programmable Logic and Applications*. Sep. 2008, S. 17–22.

[Mac01] Don MacMillen. *Nimble Compiler Environment for Agile Hardware*. USA: Storming Media LLC, 2001.

[Mar+04] T. Marescaux, V. Nollet, J.-Y. Mignolet, A. Bartic, W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde und R. Lauwereins. "Run-time support for heterogeneous multitasking on reconfigurable SoCs". In: *Integration, the VLSI Journal* 38 (Okt. 2004), S. 107–130. ISSN: 0167-9260.

[Mar+93] E. Martin, O. Sentieys, H. Dubois und J-L Philippe. “GAUT: An architectural synthesis tool for dedicated signal processors”. In: *EURO-DAC ’93: Proceedings of European Design Automation Conference*. 1993, S. 14–19.

[Mar00] Claus Märtin. *Rechnerarchitekturen - CPUs, Systeme, Software-Schnittstellen*. Hrsg. von Claus Märtin. Bd. 2. Auflage. München: Carl Hanser Verlag, 2000.

[MBH12] Lars Middendorf, Christophe Bobda und Christian Haubelt. “Hardware synthesis of recursive functions through partial stream rewriting”. In: *DAC ’12: Proceedings of the 49th Annual Design Automation Conference*. DAC ’12. New York, NY, USA: ACM, 2012, S. 1207–1215. ISBN: 978-1-4503-1199-1.

[Mei10] André Meisel. “Design Flow für IP basierte, dynamisch rekonfigurierbare, eingebettete Systeme”. Diss. Technische Universität Chemnitz, Feb. 2010.

[Men10] Mentor Graphics Cooperation. *Catapult C Synthesis*. 2010. URL: `www.menthor.com/catapult` (besucht am 22. 10. 2010).

[MH86] S. McFarling und J. Hennesey. “Reducing the cost of branches”. In: *ISCA ’86: Proceedings of the 13th annual International Symposium on Computer Architecture*. Bd. 14. 2. Tokyo, Japan: ACM, 1986, S. 396–403.

[Mig+03] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde und R. Lauwereins. “Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip”. In: *DATE ’03: Proceedings of the Conference on Design, Automation and Test in Europe*. 2003, S. 986–991.

[Mit07] Arvind Mithal. “Bluespec at MIT”. In: *The First Bluespec Workshop*. Computer Science & Artificial Intelligence Lab, Massachusetts Institute of Technology. Cambridge, Aug. 2007.

[MM06] Jim Mauro und Richard McDougall. *Solaris Internals (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006. ISBN: 0131482092.

[Mor97] Bob Morgan. *Building an Optimizing Compiler*. Digital Press, 1997.

[MR06] Vasco M. Manquinho und Olivier Roussel. “The first evaluation of pseudo-boolean solvers (PB 05)”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 2 (2006), S. 103–143.

[Mul01] Dietmar Müller. *Script zur Vorlesung Systementwurf*. Fakultät für Elektrotechnik und Informationstechnik, Technische Universität Chemnitz. 2001.

[Naj+03] Walid A. Najjar, Wim Böhm, Bruce A. Draper, Jeff Hammes, Robert Rinker, J. Ross Beveridge, Monica Chawathe und Challes Ross. "High-Level Language Abstraction for Reconfigurable Computing". In: *IEEE Computer* 36 (2003), S. 63–69. ISSN: 0018-9162.

[NB02] Juanjo Noguera und Rosa M. Badia. "Dynamic run-time HW/SW scheduling techniques for reconfigurable architectures". In: *CODES '02: Proceedings of the Tenth International Workshop on Hardware/Software Codesign*. CODES '02. Estes Park, Colorado: ACM, 2002, S. 205–210. ISBN: 1-58113-542-4.

[NBG89] John von Neumann, Arthur W. Burks und Herman H. Goldstine. "Preliminary discussion of the logical design of an electronic computing instrument (1946)". In: *Perspectives on the computer revolution* (1989), S. 39–48.

[NK01] Tilman Neumann und Andreas Koch. "A Generic Library for Adaptive Computing Environments". In: *FPL '01: Proceedings of the 11th International Conference on Field Programmable Logic and Applications*. FPL '01. London, UK, UK: Springer, 2001, S. 503–512. ISBN: 3-540-42499-7.

[NS07] Nicholas Nethercote und Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM SIGPLAN Notices* 42 (Juni 2007), S. 89–100. ISSN: 0362-1340.

[Ope13] OpenWRT Linux Distribution. 2013. URL: `https://openwrt.org` (besucht am 15. 04. 2013).

[Org05] Mario Orgis. "Untersuchung und Implementierung dynamisch partieller Rekonfiguration auf Xilinx FPGA unter Verwendung von Standardentwurfswerkzeugen". Diplomarbeit. Technische Universität Chemnitz, 2005.

[Pan+06] Gajinder Panesar, Daniel Towner, Andrew Duller, Alan Gray und Will Robbins. "Deterministic parallel processing". In: *International Journal of Parallel Programming* 34.4 (2006), S. 323–341. ISSN: 0885-7458.

[Pan07] Elena Moscu Panainte. "The Molen Compiler for Reconfigurable Architectures". Diss. Technische Universiteit Delft, 2007.

[Pec+06] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot und D. Andrews. "Hthreads: A Computational Model for Reconfigurable Devices". In: *FPL '06: Proceedings of the 16th International Conference on Field Programmable Logic and Applications*. Aug. 2006, S. 1–4.

[Pfi98] Georg F. Pfister. *In Search of Clusters*. Bd. 2. Auflage. Upper Saddle River, NJ: Prentice Hall, 1998.

[PK89] P.G. Paulin und J.P. Knight. "Force-directed scheduling for the behavioral synthesis of ASICs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 8.6 (Juni 1989), S. 661–679. ISSN: 0278-0070.

[PKH10] Uwe Pross, Karl Kröber und Ulrich Heinkel. "Abhängigkeitsanalyse und Parameterberechnung auf Spezifikationsebene". In: *13. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*. Dresden: Fraunhofer Verlag, Feb. 2010, S. 197.

[PP88] N. Park und A.C. Parker. "Sehwa: a software package for synthesis of pipelines from behavioral specifications". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 7.3 (März 1988), S. 356–370. ISSN: 0278-0070.

[Pro+08] U. Pross, E. Markert, J. Langer, A. Richter, C. Drechsler und U. Heinkel. "A Platform for Requirement Based Formal Specification". In: *FDL '08: Proceedings of the 11th Forum on Specification, Verification and Design Languages*. Sep. 2008, S. 237–238.

[PT05] David Pellerin und Scott Thibault. *Practical FPGA programming in C*. Prentice Hall modern semiconductor design series. Prentice Hall, 2005, S. 428.

[Put09] Wolfram Putzke-Röming. "Dynamic System Reconfiguration in Heterogeneous Platforms". In: Hrsg. von Michael Hübner Nikolaos Voros Alberto Rosti. Lecture Notes in Electrical Engineering 40. Heidelberg: Springer, 2009. Kap. MORPHEUS Architecture Overview, S. 31–37.

[Pyt13] Python Software Foundation. *Python Programming Language - Official Website*. 2013. URL: `http://www.python.org` (besucht am 17. 06. 2013).

[Ram89] Franz Josef Rammig. *Systematischer Entwurf digitaler Systeme: von der System- bis zur Gatter-Ebene*. Stuttgart: Teubner Verlag, 1989.

[RBH09] Marko Rößler, Enrico Billich und Ulrich Heinkel. "Hybride Architekturen und Rekonfigurationsmechanismen für paralleles Hard- und Software-Threading". In: *MST '09: Tagungsband der 9. Chemnitzer Fachtagung Mikrosystemtechnik - Mikromechanik & Mikroelektronik*. Nov. 2009, S. 124–127. ISBN: 978-3-00-029135-7.

[Ren+06] T. Renner, T. Bluhm, A. Schneider, J. Knaeblein, R. Zavala und U. Heinkel. "Formale Spezifikation und Verifikation abstrakter Beschreibungen von Telekommunikationsprotokollen". In: *MBMV '06: Tagungsband des 9. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen*. 2006.

[RH06] Marko Rößler und Ulrich Heinkel. "Konzept für die dynamische Verteilung von Prozessen auf Hardware/Software-Systemen". In: *DASS '06: Tagungsband der Dresdner Arbeitstagung Schaltungs- und Systementwurf*. Dresden, Mai 2006, S. 97–101.

[RH08] Marko Rößler und Ulrich Heinkel. "Preemptive HW/SW Threading by combining ESL methodology and coarse grained reconfiguration". In: *RoCoSoC '08: International Workshop Reconfigurable Communication-centric Systems-on-Chip*. Barcelona, Spanien, Juli 2008.

[RLH12] Marko Rößler, Jan Langer und Ulrich Heinkel. "Finding an Optimal Set of Breakpoint Locations in a Control Flow Graph". In: *SSD'12: Proceedings of the IEEE Multi-Conference on Systems, Signals and Devices*. Chemnitz, März 2012, S. 120–122. ISBN: 978-1-4673-1591-3.

[RLH13a] Marko Rößler, Jan Langer und Ulrich Heinkel. "Efficient preemption of loops for dynamic HW/SW partitioning on configurable systems on chip". In: *ESLsyn '13: Proceedings of the Electronic System Level Synthesis Conference*. Austin, Texas, 2013.

[RLH13b] Marko Rößler, Jan Langer und Ulrich Heinkel. "Synchronisation von Schleifenkörpern zur dynamischen Ablaufplanung über die HW/SW-Grenze eines Configurable System on Chip (CSoC)". In: *MBMV '13: Tagungsband des 15. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen*. 2013, S. 219–228.

[Ros+09] Alberto Rosti, Fabio Campi, Philippe Bonnot und Paul Brelet. "Dynamic System Reconfiguration in Heterogeneous Platforms". In: Hrsg. von Michael Hübner Nikolaos Voros Alberto Rosti. Lecture Notes in Electrical Engineering 40. Heidelberg: Springer,

2009. Kap. SoA of Reconfigurable Computing Architectures and Tools, S. 13–27.

[Röß+09] Marko Rößler, Hailu Wang, Ulrich Heinkel, Nur Engin und Wolfram Drescher. “Rapid prototyping of a DVB-SH turbo decoder using high-level-synthesis”. In: *FDL ’09: Proceedings of the 12th Forum on Specification, Verification and Design Languages*. Sep. 2009, S. 1–6.

[Röß+11] Marko Rößler, Daniel Froß, Jan Langer und Ulrich Heinkel. “FPGA-Accelerated Exploration of Monte Carlo Simulations Using High-Level Design Methodology”. In: *Workshop on Design Methods and Tools for FPGA-Based Acceleration of Scientific Computing at IEEE DATE conference*. Grenoble, März 2011.

[Röß09] Marko Rößler. “Parallel Hardware- and Software Threads in a Dynamically Reconfigurable System on a Programmable Chip”. In: *DATE ’09: Proceedings of the Conference on Design, Automation and Test in Europe - PhD Forum*. 2009. ISBN: 978-3-9810801-5-5.

[RST92] B. Ramskrishna Rau, Michael S. Schlansker und P. P. Tirumalai. “Code Generation Schema for Modulo Scheduled Loops”. In: *MICRO ’92: Proceedings of the 25th Annual International Symposium on Microarchitecture*. 1992, S. 158–169.

[Sch+09] A. Schallenberg, W. Nebel, A. Herrholz, P.A. Hartmann und F. Oppenheimer. “OSSS+R: A framework for application level modeling and synthesis of reconfigurable systems”. In: *DATE ’09: Proceedings of the Conference on Design, Automation and Test in Europe*. Apr. 2009, S. 970–975.

[Sim01] Harald Simmler. “Preemptive Multitasking auf FPGA Prozessoren”. Diss. Universität Mannheim, 2001.

[Sin07] Oliver Sinnen. *Task Scheduling for Parallel Systems*. Hrsg. von Albert Y. Zomaya. Hoboken, New Jersey: John Wiley & Sons, 2007.

[SLM00] H. Simmler, L. Levinson und Reinhard Männer. “Multitasking on FPGA Coprocessors”. In: *FPL ’00: Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications*. London, UK: Springer, 2000, S. 121–130. ISBN: 3-540-67899-9.

[Squ+96] M.S. Squillante, R.E. Johnson, Shiafun Liu, S.R. Kunkel und R.J. Eickemeyer. "Evaluation of Multithreaded Uniprocessors for Commercial Application Environments". In: *ISCA '96: Proceedings of the 23rd Annual International Symposium on Computer Architecture*. Mai 1996, S. 203–203.

[SS05] H.M. Sheini und K.A. Sakallah. "Pueblo: a modern pseudo-Boolean SAT solver". In: *DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe*. Bd. 2. März 2005, S. 684–685.

[Str+07] Martin Streubühr, Carsten Riedel, Christian Haubelt und Jürgen Teich. "System Level Modeling and Performance Simulation for Dynamic Reconfigurable Computing Systems in SystemC". In: *MBMV '07: Tagungsband des 10. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen*. Erlangen, Germany, März 2007, S. 59–68.

[Str+09] Jochen Strunk, Andreas Heinig, Toni Volkmer, Wolfgang Rehm und Heiko Schick. "RunTime Reconfiguration for HyperTransport coupled FPGAs using ACCFS". In: *WHTRA '09: Proceedings of the First International Workshop on HyperTransportResearch and Applications*. Mannheim, Feb. 2009, S. 54–63.

[Str+11] M. Streubühr, R. Rosales, R. Hasholzner, C. Haubelt und J. Teich. "ESL power and performance estimation for heterogeneous MPSOCS using SystemC". In: *FDL '11: Proceedings of the 14th Forum on Specification, Verification and Design Languages*. 2011, S. 1–8.

[Tan06] Andrew S. Tannanbaum. *Computerarchitektur*. Pearson Sudium, 2006.

[TBF05] Sebastian Thrun, Wolfram Burgard und Dieter Fox. "The Particle Filter". In: *Probabilistic Robotics*. MIT Press, 2005. Kap. 4.3, S. 96–113.

[Tei97] Juergen Teich. *Digitale Hardware/Software Systeme*. Berlin Heidelberg: Springer, 1997.

[TGP07] J.L. Tripp, M.B. Gokhale und K.D. Peterson. "Trident: From High-Level Language to Hardware Circuitry". In: *IEEE Computer* 40.3 (März 2007), S. 28–37. ISSN: 0018-9162.

[The13] The MathWorks Inc. *Matlab Optimization Toolbox*. 2013. URL: `http://www.mathworks.de/de/products/optimization/` (besucht am 11. 02. 2013).

[Tis+08] C. Tischendorf, J. Langer, U. Prößs und U. Heinkel. "Generierung von VHDL-Modellen aus PSL-Eigenschaften". In: *DASS '08: Tagungsband der Dresdner Arbeitstagung Schaltungs- und Systementwurf*. 2008.

[Tri+97] S. Trimberger, D. Carberry, A. Johnson und J. Wong. "A time-multiplexed FPGA". In: *FCCM '97: Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*. Apr. 1997, S. 22–28.

[TS86] Chia-Jeng Tseng und D.P. Siewiorek. "Automated Synthesis of Data Paths in Digital Systems". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 5.3 (Juli 1986), S. 379–395. ISSN: 0278-0070.

[Ull+04] M. Ullmann, M. Huebner, B. Grimm und J. Becker. "An FPGA run-time system for dynamical on-demand reconfiguration". In: *IPDPS '04: Proceedings of the 18th International Parallel and Distributed Processing Symposium*. Apr. 2004, S. 135.

[USA91] USA UNIX System Laboratories Inc. *System V Interface Definition*. Bd. 5. Addison Wesley, Sep. 1991.

[Van+01] G. Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels und I. Bolsens. "Hardware/Software partitioning of embedded system in OCAPI-xl". In: *CODES '01: Proceedings of the Ninth International Workshop on Hardware/Software Codesign*. Copenhagen, Denmark: ACM, 2001, S. 30–35. ISBN: 1-58113-364-2.

[Ven+04] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea und Seth Copen Goldstein. *C to Asynchronous Dataflow Circuits: An End-to-End Toolflow*. Techn. Ber. Computer Science Department, Carnegie Mellon University, 2004.

[Vil+10] Jason R. Villarreal, Adrian Park, Walid A. Najjar und Robert Halstead. "Designing Modular Hardware Accelerators in C with ROCCC 2.0." In: *FCCM '10: Proceedings of the 18th IEEE Symposium on Field-Programmable Custom Computing Machines*. Hrsg. von Ron Sass und Russell Tessier. IEEE Computer Society, 2010, S. 127–134. ISBN: 978-0-7695-4056-6.

[VWC01] Stamatis Vassiliadis, Stephan Wong und Sorin Cotofana. "The MOLEN $\rho\mu$-Coded Processor". In: *FPL '01: Proceedings of the 11th International Conference on Field Programmable Logic and Applications*. London, UK: Springer, 2001, S. 275–285. ISBN: 3-540-42499-7.

[Wak99] Kazutoshi Wakabayashi. “C-based synthesis experiences with a behavior synthesizer, "Cyber"”. In: *DATE ’99: Proceedings of the Conference on Design, Automation and Test in Europe*. Munich: IEEE Computer Society, 1999, S. 390–393. ISBN: 0-7695-0078-1.

[WH02] M. J. Wirthlin und B. L. Hutchings. “Improving functional density using run-time circuit reconfiguration”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 6.2 (Feb. 2002), S. 247–256.

[WK02] Grant B. Wigley und David A. Kearney. “Research Issues in Operating Systems for Reconfigurable Computing”. In: *ERSA ’02: Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures*. CSREA Press, 2002, S. 10–16.

[WP02] Herbert Walder und Marco Platzner. “Non-preemptive Multitasking on FPGAs: Task Placement and Footprint Transform”. In: *ERSA ’02: Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures*. CSREA Press, 2002, S. 24–30.

[WT85] R.A. Walker und D.E. Thomas. “A Model of Design Representation and Synthesis”. In: *Proceedings of the 22nd Conference on Design Automation*. Juni 1985, S. 453–459.

[WT89] R.A. Walker und D.E. Thomas. “Behavioral transformation for algorithmic level IC design”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 8.10 (Okt. 1989), S. 1115–1128. ISSN: 0278-0070.

[Xil04] Xilinx Inc. *Two Flows for Partial Reconfiguration: Module Based and Difference Based*. Sep. 2004.

[Xil05] Xilinx Inc. *Product Specification PLB IPIF*. v2.02a. Apr. 2005.

[Xil06] Xilinx Inc. *Early Access Partial Reconfiguration User Guide*. März 2006.

[Xil08] Xilinx Inc. *Platform Specification Format Reference Manual*. EDK 10.1, Service Pack 3. Sep. 2008.

[Xil11] Xilinx Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*. v5.0. Juni 2011.

[Xil13a] Xilinx Inc. 2013. URL: `http://www.xilinx.com` (besucht am 01. 10. 2013).

[Xil13b] Xilinx Inc. *Embedded Platform Studio*. 2013. URL: `http://www.xilinx.com/tools/platform.htm` (besucht am 15. 04. 2013).

[XW95] Jiasheng Xu und Yueming Wang. "Pipelined multiplier for signed multiplication". 5404323. Apr. 1995.

[Yan+07] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Yi Lu und S. Vassiliadis. "DWARV: Delftworkbench Automated Reconfigurable VHDL Generator". In: *FPL '07: Proceedings of the 17th International Conference on Field Programmable Logic and Applications*. Aug. 2007, S. 697–701.

[Zeb+12] C. Zebelein, J. Falk, C. Haubelt und J. Teich. "A model-based inter-process resource sharing approach for high-level synthesis of dataflow graphs". In: *ESLsyn '12: Proceedings of the Electronic System Level Synthesis Conference*. 2012, S. 17–22.

[Zha+13] Ying Zhang, Lu Peng, Xin Fu und Yue Hu. "Lighting the dark silicon by exploiting heterogeneity on future processors". In: *DAC '13: Proceedings of the 50th Annual Design Automation Conference*. Austin, Texas: ACM, 2013, S. 821–827. ISBN: 978-1-4503-2071-9.

[ZKG09] Pavel G. Zaykov, Georgi K. Kuzmanov und Georgi N. Gaydadjiev. "Reconfigurable Multithreading Architectures: A Survey". In: *SAMOS '09: Proceedings of the 9th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. Samos, Greece: Springer, 2009, S. 263–274. ISBN: 978-3-642-03137-3.

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Thesen

1. Die steigende Komplexität heterogener Systeme erfordert neue Entwurfsansätze auf höheren Abstraktionsebenen.
2. Heterogene adaptive Computersysteme besitzen Freiheitsgrade, die erst durch eine dynamisch verteilte parallele Anwendung genutzt werden können.
3. Unterbrechung und Migration von Anwendungsteilen zwischen heterogenen Rechenwerken ist möglich und durch Virtualisierung der Rechenwerke erreichbar.
4. High-Level-Synthese erlaubt eine Single-Source-basierte Anwendungsentwicklung mit genau einem Design-Entry-Point für eine verteilte parallele Anwendung auf einem heterogenen Computersystem – oder anders, die algorithmische Ebene ist ein geeigneter Design-Entry zur Single-Source-Anwendungsentwicklung für heterogene Computersysteme.
5. Bestehende parallele Programmierkonzepte sind für heterogene Computersysteme einsetzbar.
6. Unter der Annahme, dass eine Anwendung auf einen gerichteten Daten-und Kontrollflussgraph abbildbar ist, kann ein Satz von Unterbrechungspunkten für präemptives Verhalten automatisiert erzeugt werden.
7. Es kann ein optimaler Satz von Unterbrechungspunkten im Kontroll- und Datenfluss einer Anwendung gefunden werden, für den eine obere Latenzschranke bei minimalem Ressourceneinsatz garantiert werden kann.
8. Für die hardwareseitigen Kosten der domänenübergreifenden Unterbrech- und Migrierbarkeit von Anwendungsteilen lassen sich Einflussfaktoren mit unterschiedlichen Gewichten definieren.
9. Der Entwurf dynamisch verteilbarer Anwendungen lässt sich durch bestehende Werkzeuge der Entwurfsautomatisierung realisieren.

10. Eine Laufzeitumgebung für die dynamische Verteilung von Anwendungsteilen unter präemptiven Gesichtspunkten existierte bisher nicht für heterogene adaptive Computersysteme.

11. Die Prinzipien präemptiver Schedulingverfahren gelten auch für heterogene adaptive Computersysteme, bedürfen jedoch spezieller Anpassungen, die sowohl die verschiedenen Umschaltzeiten der Rechenwerke, als auch die Affinität eines Task zu bestimmten Typen von Rechenwerken berücksichtigen.

12. Der hochsprachenbasierte Entwurf erlaubt die Simulation und Verifikation einer Anwendung auf allen Abstraktionsebenen.