Lehrstuhl für Rechnertechnik und Rechnerorganisation

Metamorphosys: Entwurf eines autonomen adaptiven Systems

Jürgen Jeitner

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. H. M. Gerndt
Prüfer der Dissertation:	
	1. Univ.-Prof. Dr. A. Bode
	2. Univ.-Prof. Dr. E. Jessen, em.

Die Dissertation wurde am 10.04.2007 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 13.06.2007 angenommen.

Abstract

Many high-performance, embedded applications work in rapidly changing environments. Both power and size constraints limit hardware, while performance requirements demand algorithm-specific architectures. Adaptive systems combine flexibility of software with the possibility of adapting hardware to varying requirements. These adaptive systems contain reconfigurable computing devices integrated in a fixed infrastructure and unchangeable components. Reconfigurable computing devices enable hardware architecture changes in response to the changing environment. Various static and dynamic adaptation methods already exist. However, existing dynamic methods are restricted to do hardware changes prior application execution. In this work we develop a new dynamic method, which allows application-specific hardware changes during application runtime. In order to achieve this goal a new method of hardware-based monitoring and controlling is utilized. Because the system hardware layer has a very restricted view in matters of application goals, it is necessary to involve all system layers into the adaptation process. Therefore we develop the METAMORPHOSYS concept. To proof the quality of the concept, the adaptive processor–as an example system component–was developed. This processor incorporates one single off-the-shelf processor-core with additional reconfigurable resources and pattern-based monitoring and controlling functions. Different benchmark programs have shown that the developed adaptive processor equals other approaches in terms of computing performance but show a reduced utilization of hardware-resources by at least 20%. The results of this work are also applicable in the domain of the design of high-performance computers, which are based on fast general-purpose processor-cores and additional configurable accelerator devices.

Kurzfassung

Viele Anwendungen im Bereich der eingebetteten Systeme erfordern hohe Rechenleistung und eine zeitnahe Anpassung an sich ändernde Umgebungsbedingungen. Hierfür eignen sich *adaptive Systeme*, die neben der Flexibilität durch Software zusätzlich eine Anpassung der System-Hardware erlauben. Adaptive Systeme verfügen über eine vorgegebene, unveränderbare Struktur mit zusätzlichen rekonfigurierbaren Ressourcen. Letztere erlauben eine Hardware-Anpassung an konkrete Anwendungen oder Anwendungsbereiche mittels anwendungsspezifischer logischer Schaltungen. Diesbezüglich existieren statische und grobkörnige dynamische Anpassungsmechanismen. Im Zuge dieser Arbeit wurde ein feingranularer Ansatz untersucht, der es ermöglicht eine zeitnahe dynamische Anpassung auf Grund des Anwendungsverhaltens durchzuführen. Die Grundlage der Adaption ist eine hardwarebasierte Überwachung und Regelung der, mit rekonfigurierbaren Kapazitäten ausgestatteten, Systemkomponenten. Die gesammelten Messdaten werden in Form von Mustern abgelegt und ausgewertet. Auf Grund der beschränkten Sichtweise der Hardware im Bezug auf die Zielsetzung einer Anwendung und des Gesamtsystems, bedarf es diverser Erweiterungen aller Systemschichten. In diesem Zusammenhang ist das METAMORPHOSYS -Konzept entstanden, das sämtliche Systemschichten mit in den Adaptionsprozess einbindet. Exemplarisch werden die Überwachungs- und Regelungsmechanismen am adaptiven Prozessor gezeigt. Die Untersuchungen mithilfe verschiedener, in ihren Adaptionsmöglichkeiten beschränkten, Bewertungsprogramme ergeben zu vergleichbaren adaptiven Ansätzen äquivalente Leistungsdaten für den adaptiven Prozessor und darüber hinaus einen ca. 20% geringeren Ressourcenbedarf. Des Weiteren schaffen diese Untersuchungen die Grundlagen für den Entwurf von Hochgeschwindigkeitsprozessoren, die aus einem leistungsfähigen Kern und zusätzlichen dynamisch umdefinierbaren *Acceleratoren* bestehen.

Danksagung

Im Laufe der Arbeit an dieser Dissertation unterstützten mich viele Menschen mit Worten und Taten, bei denen allen ich mich sehr herzlich bedanken möchte. Obgleich allen ein Platz auf dieser Seite zustünde, kann im Folgenden nicht jeder namentlich erwähnt werden:

Allen voran gebührt Prof. Dr. Arndt Bode mein größter Dank. Insbesondere dafür, dass er mir überhaupt erst die Arbeit an meiner Dissertation ermöglichte und in meinen Jahren am Lehrstuhl für Rechnertechnik und Rechnerorganisation der Technischen Universität München ein hervorragendes Arbeitsumfeld zur Verfügung gestellt hat.

Zudem bin ich Prof. Eike Jessen zu Dank verpflichtet, da er kurzfristig als Gutachter zu gewinnen war und in sehr kurzer Zeit noch viele wertvolle Anmerkungen und Änderungsvorschläge einbrachte.

Außerdem hatte ich während der gesamten Zeit das Glück und die Gelegenheit mit herausragenden Wissenschaftlern und Persönlichkeiten zusammenzuarbeiten. Besonders zu erwähnen sind in diesem Zusammenhang Wolfgang Karl, Martin Schulz und Carsten Trinitis, die mir als Betreuer zur Seite standen.

Die Liste derjeniger, die mich am Lehrstuhl mit zahlreichen fachlichen und manchmal auch nicht so fachlichen Diskussionen, aber stets mit wertvollen Ideen bereicherten und in meiner Arbeit unterstützten, ist lang. Unter anderem zählen Georg Acher, Rainer Buchty, Jan-Thomas Czornack, Michael Eberl, Amitava Gupta, Elfriede Kelp, Edmond Kereku, Markus Lindermeier, Robert Lindhof, Thomas Ludwig, Peter Luksch, Martin Mairandres, Harald Meier, Jürgen Obermeier, Bruno Piochacz, Günther Rackl, Sabine Rathmayer, Karl-Heinz Seubert, Andreas Schmidt, Alexandros Stamatakis, Daniel Stodden, Jie Tao, Klaus Tilk, Max Walter, Josef Weidendorfer, Christian Weiß und Roland Wismüller zu ihnen.

Darüber hinaus möchte ich mich bei Beate Hinterwimmer, der *guten Fee* des Lehrstuhls, bedanken, dass sie mir stets bei organisatorischen Problemen mit Rat und Tat behilflich war.

Einen weiteren entscheidenden Beitrag zur Durchführung dieser Arbeit hat mein privates Umfeld geleistet. Obwohl alle wert wären erwähnt zu werden, beschränke ich mich auf die Nennung Dreier, deren Unterstützung in unmittelbarem Zusammenhang mit meiner Arbeit stand, ohne den Beitrag anderer schmälern zu wollen: Alexander Wißpeintner, der mich wissenschaftlich beraten und stets als Freund hinter mir gestanden hat. Und Peter Obermayer und Lambert Weeber, die auf Grund ihrer langjährigen Arbeitserfahrung bei technischen Problemen immer einen Ausweg wussten.

Am Ende angekommen, bleiben nur noch wenige aber nicht minder wertvolle: Meinen Eltern ist für ihre unermüdliche jahrelange Unterstützung, in jeglicher Hinsicht, zu danken. Ebenso

meiner kleinen Schwester, die es versteht mich immer aufzuheitern. Und meiner Partnerin Elke Braumiller, die, wenn auch verständlicher Weise, mit manchmal gemischten Gefühlen das Fortschreiten meiner Arbeit beäugte, einen wichtigen unverzichtbaren Beitrag geleistet hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen und Symbole

Abkürzungen:

ALU	Arithmetic Logic Unit
$ALUT$	Adaptive Look-up Table
ARM	Advanced RISC Machines Ltd
$ASIC$	Application Specific Integrated Circuit
$CISC$	Complex Instruction Set Computer
CLB	Complex Logic Block
CPU	Central Processing Unit
$CPLD$	Complex Programmable Logic Device
DCG	Deterministisches Clock-gating
$DPGA$	Dynamically Programmable Gate Array
DPM	Dynamic Power Management
DSP	Digital Signal Processor
DVS	Dynamic Voltage Scaling
EPI	Energie pro Instruktion
$EPIC$	Explicit Parallel Instruction Computer
FE	Funktionseinheit
FFT	Fast Fourier Transformation
$FPGA$	Field Programmable Gate Array
GAL	Generic Array Logic
ILP	Instruction Level Parallelism
LAB	Logic Array Block
MAC	Multiply Accumulate
MUL	Multiply
NoC	Network on Chip
PAL	Programmable Array Logic
PLB	Pipeline Balancing
PLD	Programmable Logic Device
PPM	Power and Performance Management
RAW	Read After Write
$RISC$	Reduced Instruction Set Computer
RFE	Rekonfigurierbare Funktionseinheit

RFU	Reconfigurable Function Unit
SHL	Shift Left
SHR	Shift Right
SoC	System on Chip
$VLIW$	Very Long Instruction Word
$VLSI$	Very Large Scale Integration
VM	Virtuelle Maschine
WAR	Write After Read
WAW	Write After Write

Symbole:

A_i	Eine ausgeführte Anwendung i.
c_i	Nicht optimierbares Teilstück i einer Anwendung.
G	Gewinn.
G_M	Gewinn der sich durch den Einsatz des Moduls M im Gegensatz zur Ausführung ohne M ergibt.
H_M	Häufigkeit des möglichen und tatsächlichen Einsatzes eines Moduls M.
k	Anzahl verfügbarer Kontexte bzw. rekonfigurierbarer Bereiche.
KW_i	Kontextwechsel zur Anwendung A_i. Beinhaltet zusätzlich die Anforderung zur Wiederherstellung des zugehörigen Hardware-Zustands.
m	Anzahl verfügbarer bzw. einsetzbarer Module.
M	Modul M.
$\backslash M$	Ohne Modul M.
$M(Px)$	Modul das am Optimierungspunkt Px eingesetzt werden kann.
n_{Max}	Obere (ganzzahlige) Schranke des Wertebereichs der Häufigkeit.
$n_{Max.Profil}$	Maximale Anzahl Einträge in einer Nutzungshistorie (Nutzungsprofil).
n_R	Gemessene Häufigkeit im Zeitraum T_R.
$n_{R.Max}$	Obere Schranke der Häufigkeit während eines Zeitraums T_R.
P_M	Optimierungspunkt für den Einsatz des Moduls M.
Px	Optimierungspunkt Nummer x einer Sequenz von Optimierungspunkten.
r	Rekonfigurationszeiten die zur Laufzeit einer Anwendung (innerhalb einer Zeitscheibe) durchgeführt werden.
$R_{K(i)}$	Zeitraum für die vom Betriebssystem angeforderten Rekonfigurationsmaßnahmen für die Anwendung A_i.
S_i	Sichern des Systemzustands am Ende der Zeitscheibe einer Anwendung A_i.
T	Zeitintervall.
$\overline{T}$	Länge eines Zeitraums T.
$T_{Anwendung}$	Rechenzeit einer Anwendung (bei Zeitscheibenverfahren, jeweils maximal mit der Länge einer Zeitscheibe).
$T_{Effekt}(M)$	Der (positive oder negative) Effekt auf die Ausführungszeit einer Anwendung, falls das Modul M eingesetzt wird.
T_G	Granularitätsintervall der Überwachung.
T_M	Ausführungszeit mit Modul M.
$T_{\backslash M}$	Ausführungszeit ohne Modul M.
T_{MA}	Modulanwahlzeit.
T_{MR}	Modulrekonfigurationszeit.
$T_{Überwachung}$	Gegebenes Überwachungsintervall (bestehend aus einem oder mehreren T_G).
u	Messwertumgebung.
W_i	Wiederherstellen des Systemzustands für die weitere Ausführung der Anwendung A_i.

$W_{K(i)}$	Zeitraum, den das Betriebssystem benötigt, um die Hardware-Konfiguration für die Anwendung A_i anzufordern.

Definitionen

1.

Einleitung und Motivation

Viele Anwendungen im Bereich der eingebetteten Systeme erfordern hohe Rechenleistung und müssen zudem zeitnah an sich ändernde Umgebungsbedingungen angepasst werden. Vor allem Energie- und Formfaktor-Vorgaben beschränken üblicherweise die verwendete Hardware, wobei andererseits die geforderte Rechenleistung häufig den Einsatz von anwendungsspezifischen Architekturen notwendig macht.

Moderne Fertigungsmethoden erlauben zwar die Herstellung immer umfangreicherer und komplexerer Chips, jedoch stellt sich der Fertigungslauf solcher Prozesse als mehr und mehr zeitaufwendig und kostenintensiv dar. Ein gangbarer Ausweg ist in diesem Zusammenhang der Einsatz rekonfigurierbarer Schaltungen. Systeme mit rekonfigurierbarer Logik verfügen in der Regel über eine feste Chip-Plattform, die zudem durch nachträgliche Anpassung der rekonfigurierbaren Anteile auf ein Einsatzgebiet oder eine konkrete Anwendung adaptiert werden können. Neben dem Aspekt der allgemeinen Kostensenkung durch den Einsatz von Massenbausteinen ist durch rekonfigurierbare Schaltungen ein hohes Maß an Anpassungsfähigkeit während und nach dem Entwurfsprozess gegeben.

Die hohe Flexibilität bei der Verwendung von Standardprozessoren ist ausschließlich auf die Software zurückzuführen und üblicherweise durch die *Überdimensionierung* des Prozessors gewährleistet: Einzelne Funktionen, Algorithmen oder Teilalgorithmen lassen sich häufig effizient in Hardware realisieren. Weniger performant zeigen sich diesbezügliche Software-Lösungen, die zum Ausgleich auf einen insgesamt leistungsfähigeren Prozessor angewiesen sind. Eine universelle Lösung sollte deshalb zusätzlich zur Flexibilität auf Seiten der Software auch in Hardware-Aspekten anpassbar sein.

Derartige adaptive Rechensysteme basieren häufig auf FPGAs, die mittlerweile zum einen ausreichend rekonfigurierbare Ressourcen für Rechenanwendungen bereitstellen und zum anderen brauchbare Resultate in Bezug auf Energie und maximale Taktfrequenzen liefern. Dennoch muss, um entscheidende Leistungssteigerungen zu erzielen, die Adaptivität in allen Entwicklungsphasen und darüber hinaus in den einzelnen Schichten des Zielsystems berücksichtigt werden. Grundvoraussetzung sind zum Beispiel Werkzeuge zur Anwendungsübersetzung, welche die Möglichkeiten der Hardware-Anpassung weitgehend ohne Eingriffe des Anwendungsprogrammierers ausschöpfen. Zusätzlich sollten diesem Mechanismen verfügbar gemacht werden, die eine manuelle Beeinflussung der adaptiven Fähigkeiten zulassen. Für eine dynamische Hardware-Anpassung, die nicht nur auf einer einmaligen Konfiguration zur Ausrichtung auf eine Anwendung oder ein Anwendungsgebiet beruht, repräsentiert das Betriebssystem die Systemschicht, die im laufenden Betrieb durch geeignete Maßnahmen das Rechensystem auf ausgeführte und auszuführende Anwendungen

unter Berücksichtigung aktueller Rahmenbedingungen einstellt. Darunter befindet sich die Hardware-Schicht.

In bisherigen adaptiven Rechensystemen existieren Hardware-Komponenten, die eine bestimmte Menge an Ressourcen bereithalten. Die Nutzung der Ressourcen obliegt ausschließlich dem Betriebssystem. Anwendungsspezifische Schaltungen werden zum Beispiel während der Prozesswechsel vorbereitet und Konfigurationslatenzen durch entsprechendes Scheduling vermindert. Auf feinkörnigere Anwendungsanforderungen, die sich der Sichtweise eines Betriebssystems entziehen, kann nicht dynamisch geantwortet werden. Die Beobachtung, Verfolgung und Ausnutzung solcher Abläufe bilden den Schwerpunkt dieser Arbeit. Ziel ist es durch eine zuverlässige Prognose des feingranularen Anwendungsverhaltens eine zu den gängigen statischen Methoden vergleichbare Leistungssteigerung mit der Forderung einer weitgehend optimalen Nutzung und Minimierung der eingesetzten rekonfigurierbaren Ressourcen zu erreichen.

Obwohl der zentrale Punkt die Bereitstellung und Integration geeigneter Überwachungs- und Regelungsmaßnahmen innerhalb der Hardware-Schicht eines Systems ist, müssen hierfür zusätzliche Vorkehrungen in oberen Systemschichten getroffen werden und Mechanismen eingebettet sein, die eine feinkörnige Hardware-Anpassung überhaupt erst ermöglichen. Die Resultate der Untersuchungen sind in das so genannte METAMORPHOSYS Konzept für den Entwurf eines autonomen adaptiven Gesamtsystems eingeflossen.

Das autonome adaptive Gesamtsystem stellt sich in fünf Schichten dar. Die vier horizontalen Schichten, Anwendungs- und Übersetzer-, Betriebssystem-, Virtuelle-Maschinen- und Hardware-Schicht, werden durch die vertikale PPM-Ebene (Performance and Power Management), die sämtliche Überwachungs- und Regelungsmechanismen der einzelnen Schichten umfasst, geschnitten. In Bezug auf die Überwachung und Regelung werden die Systembibliothek und der Komponentenpool als zusätzliche Systemkomponenten, die für den Einsatz adaptiver Komponenten notwendig sind, eingeführt. Der Komponentenpool beinhaltet alle im System verfügbaren Hardware-Module, die in adaptive Komponenten geladen werden können. Die Systembibliothek übernimmt die systemweite Veröffentlichung der Module und Moduleigenschaften.

Die Nutzung der Anpassungsfähigkeit, wie sie für das autonome adaptive System vorgestellt wird, erfordert verschiedene Erweiterungen in den einzelnen Systemschichten. Besonders für die Anwendungsübersetzung stellt die Gruppe adaptiver Recheneinheiten, die eine Teilmenge adaptiver Komponenten repräsentieren, eigene Anforderungen, die zum einen notwendig sind und zum anderen unterstützend zur dynamischen Anpassung beitragen. Das Einfügen von Bibliotheksinstruktionen, die die Verwendung von Hardware-Modulen anzeigen, in den VM-Code einer Anwendung stellt in diesem Zusammenhang die Grundvoraussetzung der dynamischen Adaptivität dar. Darüber hinaus ist eine automatisierte Modul-Generierung vorgesehen, die sowohl eine hardware-beschleunigte als auch reine software-basierte Ausführung in Abhängigkeit vom Konfigurationszustand des Systems erlaubt. Des Weiteren können Konfigurationsinstruktionen zur Einhaltung von Laufzeitkriterien bzw. Echtzeitanforderungen genutzt werden.

Auf Grund des Einsatzes von rekonfigurierbarer Logik existieren diverse Probleme, die ein Resultat der zeitaufwendigen Konfigurationsvorgänge sind. Diesbezüglich bietet das Konzept Betriebssystemerweiterungen, z.B. in Form spezieller Scheduling-Techniken, und Vorschläge zur Gestaltung einer eigenen Konfigurationsinfrastruktur, die zu einer vollständigen Auflösung bzw. Minimierung der Latenzproblematik führen.

Alle Systemschichten übernehmen Aufgaben in Bezug auf die Überwachung und Regelung des adaptiven Gesamtsystems. Zu den vorrangigen Aufgaben des Betriebssystems zählen z.B. die Regulierung der Spannung und Frequenz einzelner Systemteile und Komponenten. Des Weiteren zeigen sich Vorgänge, in Verbindung mit dem Einsatz rekonfigurierbarer Hardware, innerhalb von adaptiven Komponenten, die sich der Überwachung und Regelung durch das Betriebssystem entziehen. Die Rolle des Betriebssystems beschränkt sich auf die Einstellung von Rahmenbedingungen, wie z.B. die rechtzeitige Pufferung benötigter Hardware-Module. Dies bildet die Grundlage der Überwachung und Regelung innerhalb aktiver adaptiver Komponenten, die eigenständig zur optimierten Ausführung einer Anwendung beitragen. In diesem Zusammenhang ist das oberste Ziel, die vorhandenen rekonfigurierbaren Ressourcen weitestgehend optimal zu nutzen. Hierzu wird eine Regelung auf Basis von Nutzungshäufigkeiten und eine bedarfsorientierte Lösung bei zyklischem Anwendungsverhalten eingesetzt.

Eine zweite, im Bereich der Hochleistungsprozessoren wesentliche Bedeutung liegt im Einsatz von *Acceleratoren*, die zusammen mit dem Prozessor auf einem Chip untergebracht sind. Die Anzahl der auf einem Chip integrierbaren Transistorfunktionen ist in den letzten 25 Jahren exponentiell angestiegen. Mit den derzeit bekannten Prozessorarchitekturen ist eine Nutzung dieses technologischen Potentials nicht mehr möglich. Als Alternative weichen die Hersteller zurzeit auf Multi-Core-Chips aus. Diese erlauben im Allgemeinen weniger als einen n-fachen Durchsatzzuwachs mit n-facher Transistorenzahl, da der Durchsatz stark von der Effektivität der Anwendungsparallelisierung abhängt. Wenn der Einsatz der rekonfigurierbaren *Acceleratoren* mehr als den Faktor n erzielt, ist die adaptive Architektur überlegen. Jüngste Forschungsarbeiten, die auf der Supercomputing 2006 präsentiert wurden, zeigen, dass bereits statisch eingesetzte *FPGA-Acceleratoren* eine Leistungsverbesserung von 60% bis 125% im Gegensatz zu Standard-Mikroprozessoren wie z.B. Intels Pentium 4 aufweisen.

1.1 Struktur der Arbeit

Der gesamte Text lässt sich in zwei Teile gliedern. Der erste Teil der Arbeit behandelt die Grundlagen sowie eine Übersicht über den aktuellen Stand der Forschungen. Der zweite Teil umfasst den Aufbau des entstandenen Rechensystems, die theoretischen Aspekte des Konzepts und diesbezüglich essentielle Erweiterungen innerhalb der einzelnen Systemschichten und untermauert die Ergebnisse anhand der Experimente im Zusammenhang mit dem so genannten *adaptiven Prozessor*.

Im Einzelnen ist die vorliegende Arbeit wie folgt aufgebaut:

- Die **Grundlagen** bieten eine kurze Darstellung der für diese Arbeit relevanten Methoden und Techniken der Prozessorarchitektur. Anschließend findet sich eine prinzipielle Beschreibung aktueller FPGAs, deren Logikelemente und Infrastruktur sich für den Einsatz in Systemkomponenten des erarbeiteten Konzepts eignen. Des Weiteren wird der Bereich der eingebetteten Systeme als vorgesehener Einsatzschwerpunkt vorgestellt. Der abschließende Abschnitt gibt einen Überblick über die heutzutage gängigen Grundlagen des Systementwurfs unter Berücksichtigung der Faktoren Energie und Rechenleistung.

- Im Kapitel **Regelung von Systemen** werden grundlegende Begriffe der Regelungstheorie eingeführt. Darüber hinaus liefert das Kapitel eine Übersicht über gebräuchliche Regler. Abschließend werden Regelungsverfahren mit dem Schwerpunkt statistischer Prozesse angesprochen und die als *Tracking* bezeichnete Vorgehensweise eingeführt.

- Das Kapitel **Adaptive Rechensysteme** stellt eine Klassifikation von Rechensystemen anhand ihrer Eigenschaften vor. Im Weiteren werden die Möglichkeiten der dynamischen Anpassung bestehender Systeme aufgezeigt. Den Abschluss bildet eine detaillierte Beschreibung zweier existierender Projekte, die einen unmittelbaren Einfluss auf diese Arbeit haben und zusätzliche Kurzbeschreibungen weiterer Projekte, die insgesamt den Stand der Forschung aufzeigen.

- Das aus den eigenen Untersuchungen resultierende Konzept ist in Kapitel **Metamorphosys** beschrieben. Vorgestellt wird der Entwurf eines gesamten Rechensystems. Zudem beschreibt das Kapitel die Auswirkungen und die Nutzungsmöglichkeiten, die sich aus der Fähigkeit der Adaption ergeben. Darauf folgt eine ausführliche Behandlung der Überwachungs- und Regelungsmechanismen des Systems. Diesbezügliche Implementierungsaspekte stehen im Anschluss daran.

- Der praktische Einsatz des Konzepts wird exemplarisch im Kapitel **der adaptive Prozessor** durchexerziert. Einführend wird das Prozessordesign erläutert. Entsprechende Implementierungsaspekte schließen sich an. Beendet wird das Kapitel mit einer Zusammenfassung der, anhand des adaptiven Prozessors gewonnen, Ergebnisse im Zusammenhang mit dem autonomen adaptiven Rechensystem.

- Das nächste Kapitel gibt eine gesamte **Zusammenfassung** der Arbeit und einen **Ausblick** auf die nächsten Schritte. Zudem werden weiterführende Arbeiten angesprochen.

2.

Grundlagen

Im Vordergrund dieses Kapitels steht die Einführung der in der vorliegenden Arbeit verwendeten Begriffe und eine Übersicht über die in diesem Zusammenhang tangierten Themenbereiche.

Zu Beginn steht die auszugsweise Behandlung der Prozessorarchitektur. Im Anschluss folgt ein Überblick über programmierbare Logikbausteine. Beide Abschnitte bilden die Grundlage für den in Kapitel 6 vorgestellten adaptiven Prozessor. Des Weiteren gibt Abschnitt 2.3 eine Einführung zum Thema eingebettete Systeme, die das Haupteinsatzgebiet des autonomen adaptiven Gesamtsystems darstellen. Der letzte Abschnitt widmet sich den gängigen Methoden und Techniken des Systementwurfs unter den Gesichtspunkten des Energiebedarfs. Diesen kommt insbesondere im Bereich der eingebetteten Systeme eine tragende Rolle zu und sie sind seit einigen Jahren über diesen Einsatzbereich hinaus von zunehmender Bedeutung.

2.1 Prozessorarchitektur

Der folgende Abschnitt bietet einen Überblick über die für diese Arbeit relevanten Grundlagen und Techniken der Prozessorarchitektur. Behandelt werden die gängigen Befehlssatzarchitekturen, Leistungsmessung und Leistungsbewertung sowie der logische Maschinenbefehlszyklus und Fließbandverarbeitung.

2.1.1 Befehlssatzarchitektur

In der Literatur der letzten Jahre wird meist darauf hingewiesen, dass die historische Einteilung in CISC (*Complex Instruction Set Computing*) und RISC (*Reduced Instruction Set Computing*) Architekturen nicht mehr adäquat und aus dieser Sicht das Post-RISC-Zeitalter angebrochen ist [65, 95].

Der Vollständigkeit halber muss erwähnt werden, dass es mehrere Möglichkeiten gibt eine Befehlssatzarchitektur (ISA, *Instruction Set Architecture*) umzusetzen. Nach Hennessey und Patterson [72] wurde jedoch seit 1980 prinzipiell nur noch die Umsetzung von so genannten *load-store* Architekturen auf der Grundlage von Allzweckregistern verfolgt. Diese Art der Implementierung wird auch in der vorliegenden Arbeit favorisiert.

2.1.1.1 CISC (Complex Instruction Set Computing)

Der ursprüngliche Gedanke der CISC-Architektur beruhte auf der Feststellung, dass bei der Ausführung einer Anwendung, die meiste Zeit für das Holen der Daten in Anspruch genommen wird. Die logische Konsequenz daraus ist, Maschinenbefehle bereit zu stellen, die möglichst aufwendige Operationen auf den einmal geholten Daten durchführen können [92]. Zudem sollte eine verhältnismäßig geringe Programmgröße erzielt werden, um wertvolle Speicherressourcen zu sparen und kompakten, effizienten Code zu erhalten. Diese und eine Reihe weiterer Überlegungen, sowie die Forderung von Aufwärtskompatibilität von Befehlssätzen, führten zu den heute als klassische Merkmale einer CISC-Architektur geltenden Eigenschaften [42, 121]:

- Umfangreiche Befehlsarten
- Viele Adressierungsmodi
- Variable Länge der Instruktionen (vertikale Codierung)
- Eher wenige Register

2.1.1.2 RISC (Reduced Instruction Set Computing)

Der verstärkte Einsatz von Compilern und eine Abkehr von der Programmierung auf Maschinenbefehlsebene zeigten, dass Compiler nur einen Teil (typischerweise 70-80%) eines CISC-Befehlssatzes nutzen. Demzufolge sollte bei einem neuen Architekturansatz die Einfachheit einer Architektur im Mittelpunkt stehen. Dies führte zu den so genannten RISC-Architekturen, die folgende Eigenschaften aufweisen [92, 72, 95]:

- Einfacher/reduzierter Befehlssatz
- Wenige Adressierungsarten (meist *load-store*)
- Viele Register
- Feste Befehlslängen
- Befehle mit gleicher, meist einfacher, Ablaufstruktur für den Einsatz von Pipelines

2.1.1.3 Post-RISC

Mit dem Fortschreiten der technischen Möglichkeiten bei der Fertigung von Prozessoren (z.B. VLSI, *Very Large Scale Integration*) wurden die vormals bestehenden Restriktionen aufgehoben. Dies führte dazu, dass die einzelnen Eigenschaften von klassischen RISC- bzw. CISC-Architekturen nicht mehr als Unterscheidungskriterien greifen. Eine Einteilung aktueller Prozessoren in RISC- oder CISC-Architektur beruht im Allgemeinen auf der Einteilung ihrer Vorgänger [95]. Beim Entwurf heutiger Prozessoren bedient man sich aller Techniken und Hilfsmittel die eine Leistungssteigerung in Aussicht stellen ohne darauf zu achten, die Merkmale der historischen Einteilung in CISC und RISC aufrechtzuerhalten. Als wohl bekannteste Beispiele können diesbezüglich die IA32-Architektur (80x86)

von Intel [84] als CISC sowie die MPCxxxx-Prozessoren (PowerPC) von Motorola [113, 114] als RISC-Vertreter angeführt werden. Die aktuellen Prozessoren beider Hersteller weisen weder eindeutige CISC- noch RISC-Merkmale auf und gelten ausschließlich auf Grund ihrer frühen Vorgänger als CISC- bzw. RISC-Prozessoren.

Ein weiterer Aspekt der Befehlssatzarchitekturen sind die als VLIW/ EPIC (*Very Large Instruction Word/ Explicit Parallel Instruction Computing*) bezeichneten Architekturen. Im Gegensatz zu CISC und RISC ist bei VLIW/EPIC-Architekturen ausschließlich der Compiler für eine optimierte Ausführung des Anwendungscodes verantwortlich. Hierbei muss der Compiler anhand der statischen Codeinformation Befehlsabhängigkeiten erkennen und wenn möglich so auflösen, dass mehrere Befehle zu einer einzelnen Gruppe parallel ausführbarer Befehle (*Bundle*) zusammengefasst werden. Der zugehörige Prozessor verfügt über keinerlei Hardware zur weiteren Optimierung der Programmausführung. Inwiefern die vom Prozessor bereitgestellten Rechenressourcen genutzt werden, hängt somit nur vom Compiler und dem Parallelismus auf Instruktionsebene (*Instruction Level Parallelism*, ILP) des Anwendungscodes ab [23, 59, 92]. Aktuelle Vertreter der VLIW/EPIC-Architekuren sind z.B. die Intel IA64-Prozessoren Itanium [76] bzw. Itanium II [78].

2.1.2 Leistungsmessung, Bewertung und Effizienz

Allgemein wird die (Rechen-)Leistung von Rechenmaschinen durch die Geschwindigkeit und Qualität, mit der ein oder mehrere Aufträge bearbeitet werden, definiert [42, 72]. Zur Leistungsbewertung werden dementsprechend Messgrößen herangezogen, die einen Vergleich von Rechenmaschinen erlauben. Diesbezüglich sind die wichtigsten Größen:

- **Durchsatz**: Zahl der pro Zeiteinheiten bearbeiteten Aufträge
- **Ausführungszeit**: Zeit die vom Stellen der Aufgabe bis zur kompletten Fertigstellung erforderlich ist.
- **Verfügbarkeit**: Wahrscheinlichkeit eine Anlage zu einem gegebenen Zeitpunkt in einem funktionsfähigen Zustand anzutreffen.

Zwei Maschinen können z.B. auf Basis der Ausführungszeit einer Anwendung bzw. der Leistung in Relation gesetzt werden [72]:

$$n = \frac{Ausführungszeit\ Maschine_X}{Ausführungszeit\ Maschine_Y} = \frac{Leistung\ Maschine_Y}{Leistung\ Maschine_X} \tag{2.1}$$

Dies führt zur Aussage, dass $Maschine_X$ n-mal mehr leistet als $Maschine_Y$ oder umgekehrt.

Für eine weitgehend objektive Betrachtung von Durchsatz und Antwortzeit können Bewertungsprogramme (*Benchmarks*) herangezogen werden, deren primäres Ziel es ist, einheitliche Bedingungen auf unterschiedlichen Rechenanlagen zu schaffen, um einen Leistungsvergleich anstellen zu können. Um einen möglichst hohen Grad an Objektivität zu erreichen, sind Bewertungsprogramme, die jeweils auf unterschiedliche Aspekte der Leistung einer Rechenanlage abzielen, zu so genannten *Benchmark Suites* zusammengefasst. Die wohl bekanntesten Sammlungen von Bewertungsprogrammen für PCs und PC-ähnliche Rechner sind SPEC, Dhrystone und Whetstone. Darüber hinaus stellt z.B. die EEMBC (gesprochen: Embassy) *Benchmark Suite* [36] Bewertungsprogramme für *eingebettete Systeme* (siehe Abschnitt 2.3) bereit.

2.1.2.1 Geschwindigkeitsvergleich

Der Leistungsgewinn bzw. der Faktor (*Speedup*), um den die optimierte Ausführungszeit einer Aufgabe im Verhältnis zur ursprünglichen Ausführungszeit verkürzt wurde, beschreibt Amdahls Gesetz mit:

$$Speedup = \frac{Ausführungszeit\ ohne\ Optimierung}{Ausführungzeit\ mit\ Optimierung} = \frac{optimierte\ Leistung}{ursprüngliche\ Leistung} \tag{2.2}$$

In diesem Zusammenhang gilt, dass eine durchgeführte Optimierung dann zur Anwendung kommen muss, wenn diese eingesetzt werden kann. Meist können Optimierungen oder Erweiterungen nur für bestimmte Teile einer Aufgabe verwendet werden. Diesbezüglich weiterführende Gedanken finden sich in [72].

2.1.2.2 Leistung und Effizienz

Bisher wurden die grundlegenden Möglichkeiten zur Leistungsbewertung von Rechenanlagen diskutiert. Für die Bewertung einer Prozessorarchitektur müssen auf obiger Basis Kriterien herangezogen werden, die Vergleiche verschiedener Paradigmen einer Architektur sowie unterschiedlicher Architekturen zulassen. Hierzu eignen sich in erster Linie Messungen unter Berücksichtigung des Taktzyklus (in Zeiteinheiten) oder der Frequenz (1/ Taktzyklus) eines Prozessors und die angefallene Ausführungszeit (Vgl. [72]):

$$CPU\ Zeit = CPU\ Taktzyklen\ für\ Programmausführung \times Dauer\ eines\ Taktzyklus \tag{2.3}$$

oder

$$CPU\ Zeit = \frac{CPU\ Taktzyklen\ für\ Programmausführung}{Taktfrequenz} \tag{2.4}$$

Eine weitere Betrachtung der Leistungsmessung und deren Auswertung führt zum so genannten CPI-Faktor (*Clock cycles per instruction*) bzw. IPC (*Instructions Per Clock cycle*), der die Länge des ausgeführten Instruktionspfades (Anzahl der ausgeführten Befehle eines Programms) in Relation zu den für die Ausführung benötigten Taktzyklen stellt:

$$CPI = \frac{Taktzyklen\ für\ die\ Programmausführung}{Anzahl\ ausgeführter\ Befehle} = \frac{1}{IPC} \tag{2.5}$$

Weiterführend wird in [72] die Formel der CPU Zeit in messbare Faktoren zerlegt, was zu drei Kriterien führt, welche die Leistungsbewertung eines Prozessors bestimmen:

- Länge der Taktzyklen
- CPI-Faktor respektive IPC-Faktor
- Anzahl der ausgeführten Instruktionen

Die Kriterien sind jeweils voneinander abhängig. Die Länge der Taktzyklen wird von der eingesetzten Hardware-Technologie und der Organisation des Prozessors bestimmt. Der CPI-Faktor hängt sowohl von der Organisation, wie auch der Befehlssatzarchitektur ab und die Anzahl der ausgeführten Instruktionen wiederum von der Befehlssatzarchitektur und dem verwendeten Übersetzer.

Effizienz

Möchte man eine Aussage über die Effizienz einer Architektur bzw. über deren Derivate bezüglich einer Anwendung treffen, erscheint die Zeit für die Ausführung von Programmen denkbar ungeeignet, da diese keine oder nur bedingt Rückschlüsse auf die Ausnutzung etwaiger eingesetzter Optimierungstechniken zulässt. Der Versuch die Effizienz einer Prozessorarchitektur anzugeben hängt maßgeblich von zwei Faktoren ab:

- Der theoretischen oberen Schranke für den maximalen Durchsatz von Maschinenbefehlen.
- Dem tatsächlichen Durchsatz von Befehlen.

Eine obere Schranke für den Durchsatz von Maschinenbefehlen wird hauptsächlich durch die Organisation des Prozessors bestimmt. Diesbezüglich ist von Interesse, wie viele Befehle je Takt abgeschlossen werden können und somit wird dieser Wert als theoretische obere Schranke für den Durchsatz des Prozessors angesehen. Der tatsächliche Befehlsdurchsatz (IPC) variiert meist stark, je nach ausgeführter Anwendung, und ist im Normalfall eine gemessene Größe. Die Effizienz einer Architektur in Bezug auf eine bestimmte Anwendung lässt sich folgendermaßen definieren [72]:

$$Effizienz_{Anwendung} = \frac{Theoretischer\ Maximaldurchsatz\ (je\ Taktzyklus)}{IPC} \tag{2.6}$$

Die Effizienz eines Prozessors für eine ausgewählte Menge von Anwendungen kann mit Hilfe des arithmetischen Mittels wie folgt beschrieben werden:

$$Effizienz_{gesamt} = \frac{1}{n} \sum \frac{t_i}{\sum t_i} \cdot Effizienz_{Anwendung_i} \tag{2.7}$$

2.1.3 Der Maschinenbefehlszyklus

Unabhängig vom Befehlssatz eines Prozessors besteht ein ausgeführter Prozess aus einer Folge von Maschinenbefehlen bzw. Instruktionen $i_0, i_1, \ldots i_n$. Zur Verarbeitung jeder einzelnen Instruktion sind im Allgemeinen die folgenden logischen Phasen notwendig [72, 121]:

- **Befehlsholphase** (BH): Der Maschinenbefehl wird aus dem Speicher in ein Befehlsregister geholt. Die Steuerung erfolgt mittels Befehlszähler.
- **Dekodierphase** (DK): Der Befehl wird aus dem Befehlsregister entnommen und entsprechend dem Operationscode werden die Steuersignale erzeugt sowie die Adressen aus dem Adressteil extrahiert und an die Operandenspeicher weitergeleitet.

- **Operandenholphase** (OH): Die Operanden werden gelesen.
- **Ausführungsphase** (AF): Die mittels den Steuersignalen gewählte Operation wird im Rechenwerk auf den Operanden ausgeführt.
- **Rückschreibphase** (ZS): Die Ergebnisse werden in den (Operanden-)Speicher zurück geschrieben.

2.1.4 Fließbandverarbeitung

Allgemein ist unter Fließbandverarbeitung (*Pipelining*) die Aufteilung des Befehlsablaufs in Einzelschritte (oder Stufen) zu verstehen. Die benötigte Zeit für die gesamte Verarbeitung eines Befehls bleibt hierbei in erster Näherung unverändert. Ein erhöhter Befehlsdurchsatz ist aber dann erkennbar, wenn das Fließband (*Pipeline*) gefüllt ist und dementsprechend die Verarbeitung der Einzelschritte überlappend ausgeführt wird.

Der Einsatz von Fließbandverarbeitung bedeutet für die Architektur eines Prozessors, dass die Einzelschritte des Maschinenbefehlszyklus durch physikalisch voneinander getrennte Teilwerke (Pipelinestufen), die jeweils durch Register (*Latches*) synchronisiert sind, realisiert werden. Die Abarbeitung jeder Instruktion erfolgt stufenweise. Im Idealfall ergibt sich bei vollständig gefüllter *Pipeline* ein Durchsatz, der dem Kehrwert der Verarbeitungszeit der längsten Pipelinestufe entspricht.

Der Ablauf des Maschinenbefehlszyklus unter Berücksichtigung einer Pipeline ist grundsätzlich mit der sequentiellen Befehlsverarbeitung identisch. Beim Design eines Prozessors spielen unterschiedliche Kriterien, wie z.B. die maximale resultierende Taktfrequenz, Energiebedarf (siehe Abschnitt 2.4) usw., bei der letztendlich gewählten Abbildung des Maschinenbefehlszyklus auf physikalische Teilwerke eine erhebliche Rolle. Vereinfachend wird im Folgenden nicht mehr zwischen logischen Phasen und physikalischen Teilwerken unterschieden, d.h. dass Phase, Teilwerk und Pipelinestufe als äquivalent angesehen werden.

2.1.4.1 Fließbandprobleme

Selbst unter der Annahme einer idealen Speicheranbindung sind drei Probleme bei der Verwendung von Pipelines (*Pipeline Hazards*) anzutreffen:

- **Ressourcenkonflikte** (*Structural Hazards*): Ressourcen des Prozessors sind nicht im erforderlichen Umfang vorhanden.
- **Datenabhängigkeiten** (*Data Hazards*): Daten werden benötigt, bevor diese verfügbar sind.
- **Kontrollflussabhängigkeiten** (*Control/ Branch Hazards*): Wenn die Sprungbedingung noch nicht ausgewertet ist, führen bedingte Sprünge zu einer schlechten Nutzung der Pipeline.

Jeder dieser Konflikte bzw. Abhängigkeiten führt zu einer Verschlechterung der Leistung eines Prozessors mit Fließbandverarbeitung, da das Auftreten eines Hazards zum Anhalten (*Stall*) der Verarbeitung bzw. zum Auffüllen von abhängigen Pipelinestufen mit Leeroperationen (NOOP- bzw. NOP-Operationen) führen kann. Die volle Leistung einer Pipeline wird jedoch nur erreicht, wenn die Pipeline mit produktiven Operationen gefüllt ist.

Ressourcenkonflikte

Allgemein tritt ein Ressourcenkonflikt dann auf, wenn ein Betriebsmittel, dass n-mal verfügbar ist, mehr als n-mal zu einem bestimmten Zeitpunkt benötigt wird. Ressourcenkonflikte können – ohne die Verwendung von rekonfigurierbarer Hardware – nicht zur Laufzeit eines Programms gelöst werden. Das heißt entweder es wurde bereits zum Entwurfszeitpunkt darauf geachtet, dass möglichst wenige oder keine Ressourcenkonflikte auftreten, oder es muss die Pipeline im Konfliktfall angehalten bzw. mit Leeroperationen, bis zur Lösung des Problems, gefüllt werden. Obwohl Ressourcenkonflikte in der Regel bereits zur Entwurfszeit einfach zu identifizieren sind, kann, z.B. aus Gründen der Platzersparnis oder wegen zu hoher Anforderungen an die Speicherbandbreite, eine meist kostengünstigere Variante mit bekannten Konflikten einer konfliktfreien Lösung vorgezogen werden (Vgl. [72]).

Datenabhängigkeiten

Datenabhängigkeiten entstehen immer dann, wenn eine Instruktion Daten einer Vorgängerinstruktion benötigt, die jedoch zum Zeitpunkt der Anfrage noch nicht lieferbar sind. Drei Arten von Datenabhängigkeiten führen zu einem Konflikt die mit zwei sequenziellen Instruktionen i und j wie folgt beschrieben werden können [72]:

- **RAW** (*Read After Write*): j versucht ein Ergebnis von i zu lesen, bevor dieses geschrieben wurde (echte Datenabhängigkeit).
- **WAW** (*Write After Write*): j schreibt sein Ergebnis, bevor das Ergebnis von i in den gleichen Speicher geschrieben wurde, was zu Dateninkonsistenz führt. (Output-Datenabhängigkeit)
- **WAR** (*Write After Read*): Befehl j hat sein Ergebnis bereits geschrieben, bevor i den ursprünglichen Wert lesen konnte. (Anti-Datenabhängigkeit)

Das Auftreten von RAW-Konflikte ist ausschließlich von der Reihenfolge der Maschinenbefehle bzw. den zugehörigen Registerzugriffen abhängig. Im Gegensatz dazu sind WAW- und WAR-Konflikte bedingt durch die Architektur des Prozessors. Diese treten nur dann auf, falls es mehrere Pipelinestufen gibt, die ein Schreiben in Register erlauben.

Kontrollflussprobleme

Die letzte Kategorie von Pipeline Hazards sind Kontrollflussprobleme. Jeder bedingte Sprung einer Anwendung kann ein Kontrollflussproblem hervorbringen, da bei einer gültigen Sprungbedingung alle nachfolgenden, bereits in der Pipeline befindlichen, Befehle ungültig werden bzw. nicht mehr zur Ausführung gebracht werden dürfen. Das heißt, dass die Pipeline erst nach der Neuberechnung des Befehlszählers, falls eine Verzweigung erfolgt ist, an einer anderen Programmadresse mit der Ausführung von Befehlen und somit einer Neufüllung der Pipeline beginnen kann.

2.1.4.2 Lösungen für Fließbandprobleme

Dieser Abschnitt gibt eine Übersicht über die Techniken zur Reduzierung und Behebung von Fließbandproblemen, die unmittelbar im Zusammenhang mit dem adaptiven Prozessor

stehen (siehe Kapitel 6). Eine detaillierte weiterführende Betrachtung findet sich z.B. in [72, 121, 92].

Bypassing

Für die Umgehung oder Minimierung von Wartezyklen in Bezug auf RAW-Datenabhängigkeiten sorgt das so genannte *Bypassing* oder *Forwarding*. RAW-Konflikte ergeben sich aus einer Programmsequenz. Falls eine Instruktion j das Ergebnis der vorangegangenen Instruktion i lesen möchte, bevor diese die Rückschreibphase passiert hat, muss das Ergebnis vorher schon bekannt gemacht werden. Die Lösung basiert darauf, dass das Ergebnis von i bereits nach der Ausführungsphase vorliegt. Dementsprechend werden Datenpfade eingeführt, die eine Ergebnisanlieferung an etwaige wartende Instruktionen ermöglichen ohne dass die Instruktion i vollständig abgearbeitet ist. Die Daten der Instruktion j werden nicht aus dem Registersatz geholt, sondern das Ergebnis wird direkt aus der Ausführungsphase von i geliefert. Diese Umgehung des regulären Pipelineablaufs bzw. des vorzeitigen Bereitstellens von Ergebnissen führte zu den Namen *Bypassing* oder *Forwarding*.

Register Renaming

Zu den bereits aufgeführten Datenabhängigkeiten existieren Abhängigkeiten die rein aus der Position oder Adresse (auch engl. *name*) von Daten resultieren und auf einen Mangel an Ressourcen zurückzuführen sind. Z.B. tritt dieser Fall ein, falls zwei Befehle i und j auf den gleichen Registern rechnen, ohne das i Daten von j benötigt und umgekehrt. Somit ergibt sich die Abhängigkeit nur aus der Position, nicht jedoch aus einer Werteabhängigkeit. Eine Lösung dieses Problems kann durch das *Umbenennen* von Registern erreicht werden. Für dieses Vorgehen benötigt ein Prozessor die Fähigkeit, eine eindeutige Abbildung von logischen auf physikalische Register vorzunehmen. Das *Register Renaming* bleibt für einen Übersetzer transparent und wird intern vom Prozessor vorgenommen.

Verzweigungsvorhersage

Zur Minderung der Kontrollflussabhängigkeiten wird üblicherweise eine Sprungvorhersage (*Branch Prediction*) eingesetzt. Ziel der Sprungvorhersage ist die Aufrechterhaltung einer gefüllten Pipeline. Wird das Sprungverhalten falsch vorhergesagt, muss dennoch die Pipeline geleert und neu gefüllt werden. Je tiefer eine Prozessorpipeline um so höher ist der Zeitverlust durch falsche Vorhersagen. Damit eine möglichst zuverlässige Vorhersage unter Berücksichtigung des Programmverlaufs erzielt wird, kommen vermehrt dynamische Sprungvorhersagen zum Einsatz.

2.2 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) zählen zu den *Programmable Logic Devices* (PLDs). Hierbei handelt es sich allgemein um Logikbausteine, deren interne Verschaltung respektive Funktionalität frei programmierbar ist. Die einmalige Programmierung solcher Bausteine heißt Konfiguration. Können Bausteine mehrfach oder beliebig oft konfiguriert werden, gelten diese als rekonfigurierbare Bausteine. In dieser Arbeit sind ausschließlich letztere von Bedeutung, weshalb die Begriffe Konfiguration und Rekonfiguration synonym Verwendung finden.

Neben der ursprünglichen *Programmable Array Logic* (PAL) bzw. *Generic Array Logic* (GAL) bilden heutzutage *Complex Programmable Logic Devices* (CPLDs) und FPGAs die am häufigsten eingesetzte Gruppe der PLDs. FPGAs stellen diesbezüglich die feinkörnigsten Bausteine dar und bieten umfangreiche Logik- und Speicherressourcen, die stets die Möglichkeit der Rekonfiguration eröffnen (Vgl. auch [2]).

2.2.1 Aufbau und Struktur von FPGAs

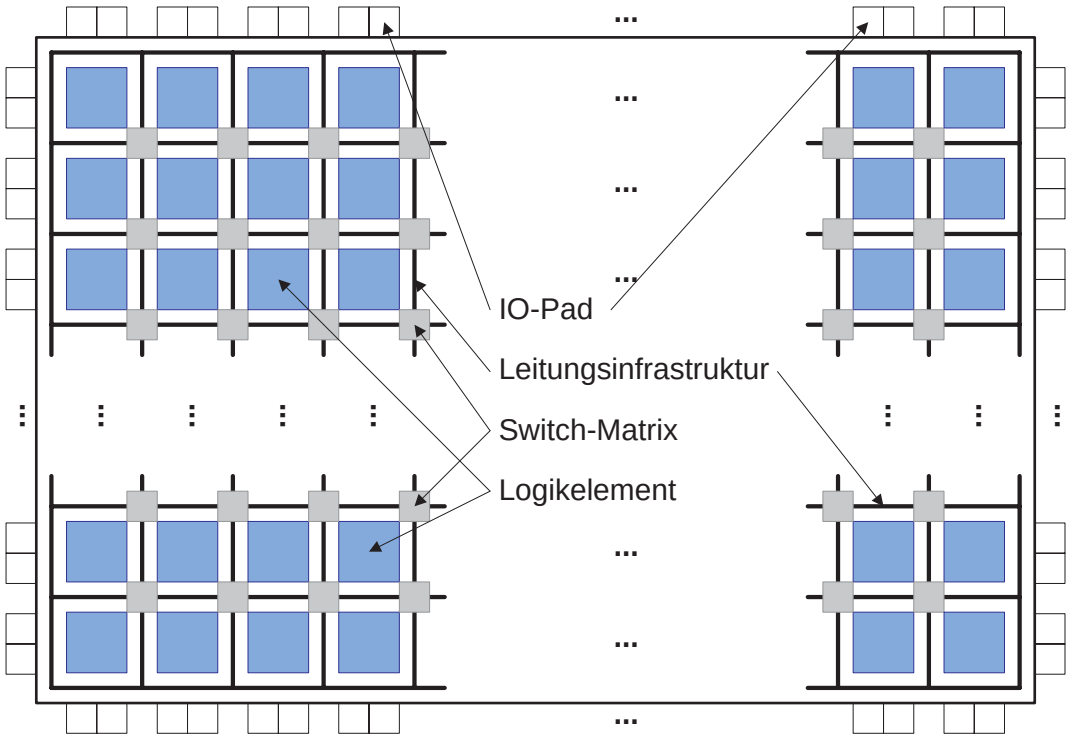

Abbildung 2.1: Prinzipieller Aufbau von FPGAs

FPGAs bestehen aus einzelnen Logikelementen (bzw. Logikzellen), die in Form einer Matrix angeordnet sind. Für die Kommunikation zwischen Logikelementen steht eine aufwendige Leitungsinfrastruktur mit unterschiedlichen Leitungslängen und Verbindungsmöglichkeiten zur Verfügung. Die Verbindung zwischen vertikalen und horizontalen Signalleitungen wird in Abhängigkeit von der programmierten Funktionalität mit Hilfe von Switch-Matrizen her-

gestellt. Darüber hinaus stellen FPGAs so genannte IO-Pads, entsprechend den zugehörigen Pins des FPGAs, bereit (Abbildung 2.1).

Der Aufbau und Funktionsumfang der einzelnen Bestandteile eines FPGAs variieren von Hersteller zu Hersteller und ändern sich zudem massiv mit jeder neuen Bausteingeneration. Zudem ist die Nomenklatur herstellerspezifisch. Z.B. bezeichnet Xilinx die Logikelemente der Virtex-FPGA-Familie als *Complex Logic Blocks* (CLBs). Für die vergleichbare Stratix-Serie von Altera werden die Logikelemente als *Adaptive Look-up Tables* (ALUTs) bezeichnet. Der Funktionsumfang beider Logikelemente ist jedoch weitgehend identisch [11, 155].

Des Weiteren sind in aktuellen FPGAs zusätzliche Ressourcen integriert. Hierzu zählen z.B. Speicherblöcke unterschiedlicher Größe, die über diverse, konfigurierbare Adressierungsmodi verfügen. Zusätzlich bieten *Digital Signal Processing*-Elemente (DSP-Elemente) die Möglichkeit eine gewisse Funktionalität in optimierter Form, ohne die Nutzung von Logikelementen, umzusetzen. Z.B. erlaubt ein DSP-Element die Realisierung einer kompletten 18×18 Bit Multiplikation ohne den zusätzlichen Einsatz von herkömmlichen Logikelementen [11, 155].

2.2.2 Rekonfigurierbarkeit von FPGAs

Von besonderer Wichtigkeit für das autonome adaptive Gesamtsystem ist die dynamische Rekonfigurationsfähigkeit von FPGAs. D.h. die Fähigkeit während des Betriebs Konfigurationsdaten an den FPGA zu übergeben und somit bei Bedarf unterschiedliche Funktionalität bereitzustellen[1]. Diesbezüglich unterscheidet man zwei Arten von Rekonfiguration:

1. **Partielle Rekonfiguration**, die nur einen ausgewählten Teil der FPGA-Ressourcen beeinflusst.

2. **Totale** (oder vollständige) **Rekonfiguration**, bei der stets alle Ressourcen neu konfiguriert werden.

Für die Verwendung im adaptiven Gesamtsystem ist die Art der Rekonfiguration von untergeordneter Bedeutung. Welche Rekonfigurationsart eingesetzt werden muss, hängt von der Integration der rekonfigurierbaren Elemente in einer Hardware-Komponente ab. Im Zusammenhang mit dem adaptiven Gesamtsystem bzw. dem exemplarisch vorgestellten adaptiven Prozessor eignet sich die totale Rekonfiguration (siehe Abschnitt 5.2.4 bzw. 6.2.3).

Die *Rekonfigurationszeit*, also der Zeitraum vom Beginn der Rekonfiguration bis zu deren Abschluss, ist maßgeblich für die resultierende Rechenleistung des adaptiven Prozessors respektive betroffener Systemteile und dementsprechend des Gesamtsystems. Während einer Rekonfigurationsphase ist der betroffene Teil (partielle Rekonfiguration) bzw. der gesamte rekonfigurierbare Bereich nicht funktionsfähig.

Unterschiedliche Arbeiten befassen sich mit dem Thema Rekonfigurationszeit und weisen dies als Nachteil von älteren sowie aktuellen FPGAs aus (Vgl. [2, 157, 30]). Obwohl inzwischen Übertragungsraten von bis zu 200 MBit/s [11, 155] unterstützt werden, dauern Rekonfigurationsvorgänge – in Abhängigkeit des Umfangs der zu ladenden Logikschaltung

[1] Sowohl Rekonfiguration als auch Konfiguration werden in der gesamten Arbeit synonym zu dynamischer Rekonfiguration verwendet.

– bis hin zu mehreren Millisekunden. Für die vorliegende Arbeit sind diese Zeiträume inakzeptabel. Dementsprechend enthält Abschnitt 6.2.3 einen Lösungsansatz, dessen Umsetzung im autonomen adaptiven Gesamtsystem vorstellbar ist und zu adäquaten Rekonfigurationszeiten führt.

2.2.3 FPGA Derivate

2.2.3.1 Multi-Kontext-FPGAs

Eine weitere Gruppe von FPGAs bilden die Multi-Kontext FPGAs oder *Dynamically Programmable Gate Array* (DPGAs). Solche FPGAs sind in der Lage mehrere unabhängige Logikschaltungen gleichzeitig in Form von Konfigurationsinformationen aufzunehmen. Jede unabhängige Schaltung wird hierbei als Kontext bezeichnet. Während des Betriebs kann zu einem bestimmten Zeitpunkt ausschließlich ein einziger Kontext aktiv sein. Die Einführung von Mutli-Kontext-FPGAs dient der Reduzierung von Latenzen durch Konfigurationsvorgänge (siehe vorherigen Abschnitt 2.2.2), benötigt jedoch zusätzliche Speicherressourcen innerhalb des FPGAs zur Ablage der Konfigurationsinformation [157, 30].

2.2.3.2 Mischform aus FPGA und unveränderbarem Prozessorkern

Sowohl von Xilinx als auch Altera wird zu den aktuellen FPGA-Familien jeweils eine FPGA-Baureihe mit fest integriertem Prozessorkern (*Hard-Core*) angeboten. Ein oder mehrere unveränderliche Prozessorkerne sind hierbei von rekonfigurierbaren Elementen umgeben. Der Platzbedarf des oder der Prozessorkerne verringert die Anzahl der rekonfigurierbaren Ressourcen im Vergleich zu den entsprechenden FPGA-Baureihen ohne integrierten Prozessor. Andererseits erlaubt dies z.B. eine schnelle Umsetzung von Systemen auf einem Chip (SoC). In diesem Zusammenhang setzt Xilinx auf einen PowerPC-Prozessorkern [156] und Altera auf die Prozessorkerne der Firma ARM (*Advanced RISC Machines*) unter der Bezeichnung Nios [12]. Diese Bausteine sind die Ursprünge des Accelerator-Ansatzes für Hochgeschwindigkeitsprozessoren (Vgl. [139, 94]).

2.3 Eingebettete Systeme

Unter *eingebetteten Systemen* versteht man im Allgemeinen Rechensysteme, die auf bestimmte Anwendungsbereiche spezialisiert und in einen technischen Kontext integriert sind. Dementsprechend erkennt man eingebettete Systeme häufig nicht als Computer. Im Gegensatz dazu stehen universal einsetzbare Systeme, wie z.B. Arbeitsplatzrechner usw., deren Entwurf keine Festlegung eines speziellen Verwendungszwecks vorsieht [2, 43, 148].

Eingebettete Systeme zeichnen sich vorrangig durch folgende Eigenschaften aus [137]:

- **Echtzeitfähigkeit**: Da eingebettete Systeme häufig in zeitkritischen Anwendungen zum Einsatz kommen (z.B. Prozess-Steuerungen), müssen bestimmte zeitliche Vorgaben bei der Verarbeitung eingehalten werden, um fehlerhafte Zustände zu vermeiden. In diesem Zusammenhang wird zwischen **harter** und **weicher Echtzeitfähigkeit** (siehe Abschnitt 2.3.4) unterschieden.

- **Heterogenität**: Eingebettete Systeme beinhalten oft unterschiedliche Hardware- und Software-Komponenten. Meist sind mehrere Mikroprozessoren, Mikrokontroller, DSPs sowie ASICs oder PLDs in ein Gesamtsystem integriert.

- **Verteilte Implementierung**: Die Kommunikation zwischen Komponenten eines eingebetteten Systems erfolgt häufig über ein oder mehrere gemeinsame, protokollbasierte Kommunikationssysteme. Hierbei sind zum Teil hohe Bandbreiten notwendig. Zudem liegen meist kritische zeitliche Anforderungen vor.

- **Begrenzte Ressourcen**: Auf Grund festgelegter Anwendungsbereiche sowie aus ökonomischen Erwägungen (z.B. Bauteilpreis) liegen in der Regel starke Ressourcenrestriktionen vor. Die Beschränkungen betreffen z.B. Stromverbrauch, Rechenleistung, Speichergröße, Formfaktor usw. Darüber hinaus sind eingebettete Systeme oftmals nur mit hohem Aufwand erweiterbar.

2.3.1 Leistungsaufnahme

Die elektrische Leistungsaufnahme ist in der Regel für eingebettete Systeme durch ihr Einsatzgebiet limitiert. In diesem Zusammenhang müssen zum einen direkte Einflüsse, wie z.B. Akku- bzw. Batteriebetrieb, Formfaktor, Kosten usw., berücksichtigt werden und zum anderen indirekte, wie z.B. geringe oder keine Kühlmöglichkeit des Systems: Ein Großteil der aufgenommenen elektrischen Energie wird als Abwärme abgegeben [2].

Die Leistungsaufnahme ist ein wichtiger Aspekt der beim Entwurf von eingebetteten Systemen und in zunehmendem Maße auch für Arbeitsplatzrechner zu beachten ist (siehe Abschnitt 2.4). Bei den üblicherweise eingesetzten CMOS-Bausteinen ist die Verlustleistung maßgeblich von den statischen Leckströmen der MOSFET-Transistoren sowie den Verlusten durch die dynamische Umladung bestimmt. Die Leckströme steigen mit der Anzahl der Transistoren und der Verkleinerung der Integrationsstrukturen. Im Gegensatz dazu steigt die dynamische Leistungsaufnahme nahezu linear mit der Taktfrequenz [70].

2.3.2 Beschränkungen der Speicherarchitektur

Durch die Anpassung eingebetteter Systeme an ein bestimmtes Einsatzgebiet ist üblicherweise die Größe und Anschlussart des Speichers definiert. Hierbei wird häufig eine direkte Kopplung des Speichers an einen Prozessor realisiert. Im Gegensatz dazu steht die Anbindung des Systemspeichers über einen eigenständigen Speicherkontroller, wie dies in der Regel bei Arbeitsplatzrechnern bzw. PC-ähnlichen Systemen der Fall ist.

Eine direkte Speicheranbindung führt meist zu einer Beschränkung des maximalen Adressraums sowie der Wahl des einsetzbaren Speichertyps (ROM, SRAM, SDRAM, etc.). Darüber hinaus kommen in eingebetteten Systemen vereinzelt Prozessoren mit integriertem Speicher zum Einsatz, um einen schnellen Datenaustausch zu erreichen.

Typischerweise werden Flash-EPROMs in Größenordnungen von 16 KByte bis 1 MByte als mehrfach programmierbare ROMs verwendet. Der Arbeitsspeicher ist meist in Form von SRAM integriert. Die Größe des Arbeitsspeichers liegt oftmals im Bereich von 4 KByte bis wenige 100 KByte (Vgl. auch [2]). Eine Ausnahme bilden die derzeitigen mobilen Endgeräte, wie Mobiltelefone, PDAs (Personal Desktop Assistent) usw., deren Arbeitsspeicher in der Regel mehrere MByte umfasst.

2.3.3 Mikroprozessoren für eingebettete Systeme

Dieser Abschnitt beschreibt in erster Linie den von der Firma ARM (*Advanced RISC Machines*) entwickelten ARM v4 Prozessorkern. Dessen Maschinenbefehlssatz und die Grundstruktur dienen als Vorlage für den in Kapitel 6 vorgestellten adaptiven Prozessor. Darüber hinaus stellen die ARM-Prozessoren die am weitesten verbreitete Prozessorfamilie im eingebetteten Bereich dar [2].

Des Weiteren findet man in eingebetteten System häufig Intel 80x86 (Intel IA32 [84, 80, 81] sowie mobile Derivate, z.B. Pentium M [79] etc.) bzw. kompatible Prozessoren. Letztere sind z.B. durch AMD Elan [4] und Alchemy [3] sowie durch National Geode [115] vertreten.

Zudem bieten Motorola (MPowerPC [38] ,Dragonball [111], ColdFire [112]), IBM (PowerPC [40, 91]), Hitachi (SuperH3, SuperH4, SuperH5 [126, 41], SH7708 [73]) usw. Prozessoren für den eingebetteten Bereich an.

Eine ausführliche Beschreibung der hier aufgeführten Prozessoren kann in [39] nachgelesen werden. Darüber hinaus werden dort weitere, im eingebetteten Bereich gängige Prozessoren sowie ältere 16 bzw. 8 Bit Prozessoren und Mikrocontroller aufgelistet und genauer behandelt.

2.3.3.1 ARM-Prozessoren und Prozessorkerne

ARM bietet aktuell folgende Produktfamilien an [17]:

- ARM7, ARM9 Thumb Family
- ARM9E, ARM10E Family
- ARM11 Family
- SecurCore Family

- OptimoDE Data Engine Family
- Cortex Family

Jede Produktfamilie enthält Prozessorserien in unterschiedlicher Ausführung zur Unterstützung spezieller Anforderungen in drei Systemkategorien [18]:

1. Eingebettete Echtzeit-Systeme: Einsatz in Storage-, Automotive-, industriellen und Netzwerk-Anwendungen.
2. Anwendungsplattformen: Geräte die Betriebssysteme wie Linux, Palm OS, Symbian OS und Windows CE in drahtlosen, Unterhaltungs- und bildverarbeitenden Anwendungen nutzen.
3. Sicherheitsanwendungen: Einsatz in *Smart Cards*, SIM-Karten und Terminals für den Zahlungsverkehr.

Befehlssatzarchitektur

Die Bereiche der eingebetteten Echtzeit-Systeme und Anwendungsplattformen werden von Prozessoren der ARM7 bis ARM11 Produktfamilien dominiert[2]. Die in diesen Familien eingesetzten Kerne werden als ARMv4 bis ARMv7 [17, 26] bezeichnet und geben die verwendete Befehlssatzarchitektur (ISA) wieder. Alle Kerne sind zum ältesten heute noch unterstützten ARMv4-Kern kompatibel. Dementsprechend repräsentieren die ARM7 bis ARM11 Prozessoren eine 32 Bit RISC-Architektur mit 32 Bit Adressraum, deren Befehlssatz 16 Allzweckregister bereithält.

Die ursprüngliche ARMv4 Befehlssatzarchitektur liegt in einigen ARM7 sowie den Intel StrongARM-Prozessoren (z.B. SA-1100 [89]) vor. Weit häufiger wurde jedoch der ARMv4T-Kern eingesetzt. Das T steht für *Thumb* und bezeichnet die Fähigkeit dieser Kerne in den so genannten *Thumb-Modus* zu wechseln. Dieser erlaubt die Ausführung von 16 Bit RISC-Instruktionen. Mit entsprechenden Übersetzern kann der Thumb-Modus zur Reduzierung der Codegröße (bis zu 35%) im Gegensatz zum äquivalenten 32 Bit Code einer Anwendungen beitragen. Die nachfolgenden Kernvarianten, mit Ausnahme des ARMv7M-Kerns, unterstützen den Thumb-Modus. Ab dem ARMv6-Kern wird optional bereits eine Weiterentwicklung namens *Thumb II* angeboten. Der ARMv7M verarbeitet, zu den regulären 32 Bit Instruktionen, ausschließlich die *Thumb II* Maschinenbefehle.

Darüber hinaus sind die einzelnen Kernversionen mit verschiedenen zusätzlichen Befehlserweiterungen erhältlich [20]:

- Enhanced DSP (E): Zusätzliche Befehle zur Verbesserung und Beschleunigung von DSP-Aufgaben.
- Jazelle (J): Befehlssatzerweiterung zur Unterstützung von Beschleunigungstechniken für Java.
- Single Instruction Multiple Data (SIMD): Einführung zusätzlicher Befehle zur Beschleunigung von Multimediaanwendungen.

[2] ARM8 existiert nicht.

- TrustZone: Bietet die Möglichkeit zwei Speicherbereiche getrennt voneinander zu verwalten.
- NEON: Stellt eine Verbesserung der SIMD-Erweiterung dar und erlaubt die Verarbeitung von 64 und 128 Bit SIMD-Befehlen.
- Vector Floating Point (VFP): Mit VFP wird eine zur IEEE 754 Norm konforme Vektorgleitkomma-Einheit bereitgestellt.

Zusätzlich existieren noch Befehlssatzerweiterungen für spezielle Debug-Anforderungen sowie die Möglichkeit Busschnittstellen (z.B. AMBA) direkt in die Prozessorkerne zu integrieren. Zudem wird eine Erweiterung zur dynamischen Regulierung der Prozessorleistung und des Energiebedarfs unter dem Namen *ARM Intelligent Energy Manager* angeboten (Details siehe Abschnitt 4.3.1).

Grundlegender 32 Bit Befehlssatz

Für den adaptiven Prozessor (siehe Kapitel 6) ist der reguläre 32 Bit RISC-Befehlssatz der ARM7- bis ARM11-Prozessoren ohne Befehlssatzerweiterungen von Bedeutung. Ohne die Anpassungsfähigkeit des adaptiven Prozessors zu berücksichtigen, ist dieser kompatibel zum ARMv4-Kern. Insgesamt kann die Kompatibilität bis hin zum ARMv7-Kern ausschließlich durch die Möglichkeit der dynamischen Anpassung erreicht werden. Das Ziel dieser Arbeit liegt jedoch nicht darin eine bestehende Prozessorarchitektur durch Anpassungsmechanismen nachzubilden, sondern eine bessere Adaption an ausgeführte Anwendungen zu erlangen. Dennoch bildet der 32 Bit Befehlssatz der ARM-Prozessoren die Basis für den Befehlssatz des adaptiven Prozessors. Dementsprechend hat der Befehlsumfang und die Codierung der Befehlsworte einen direkten Einfluss auf den Entwurf des adaptiven Prozessors und wird im Folgenden genauer betrachtet.

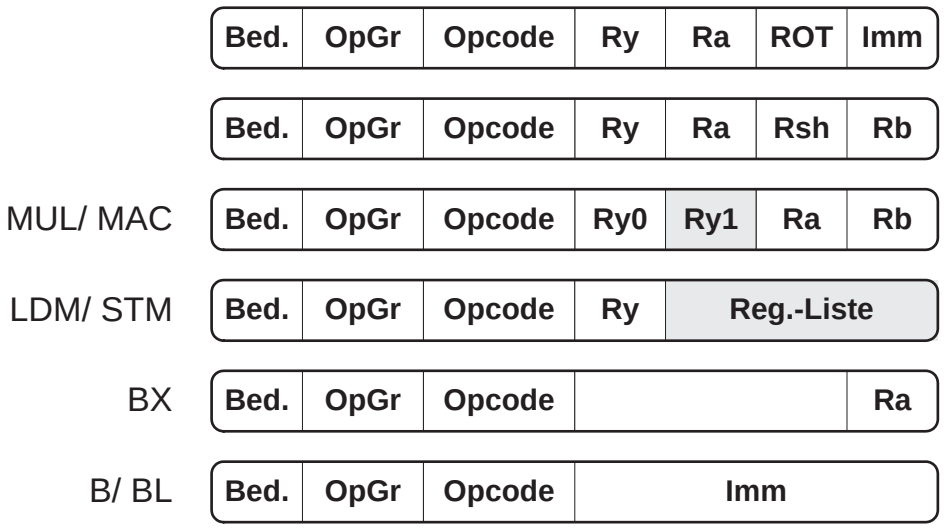

Abbildung 2.2: Befehlsgruppen des 32 Bit ARM Befehlssatzes

Grundsätzlich lässt sich der 32 Bit ARM-Befehlssatz in sechs Befehlsgruppen unterteilen (Abbildung 2.2). Die gezeigte Unterteilung basiert nicht auf der Funktionalität der Maschi-

nenbefehle sondern ergibt sich anhand der Anzahl und Interpretation der benötigten Register bzw. Operanden. Man erkennt in Abbildung 2.2 dass alle Befehlsgruppen über ein Bedingungsfeld, ein Opcode-Gruppenfeld und ein zugehöriges Opcode-Feld verfügen. Die meisten Maschinenbefehle nutzen ein Eingangsregister (*Ra*) sowie einen zweiten Operanden, der entweder ein Register (*Rb*) oder ein Konstantenwert (*Imm*) ist. Liegt ein Konstantenwert vor, wird dieser Wert immer um einen im Maschinenbefehl enthaltenen Wert rotiert (*ROT*). Für ein Register als zweiten Eingangswert muss zusätzlich ein Schieberegister (*Rsh*) für eine optionale Schiebeoperation angegeben werden[3]. Der eigentliche Operandenwert liegt erst nach Ausführung der Schiebeoperation vor. Die Ablage eines Ergebnisses erfolgt in einem einzelnen Ausgangsregister. Ob ein Ergebnis in das angegebene Register zurückgeschrieben wird, kann mit Hilfe des *Set*-Bits gesteuert werden (fehlt in Abbildung 2.2). Abweichend von diesem Grundschema der Maschinenbefehle existieren vier weitere Befehlsgruppen:

1. Für die Multiplikation und *Multiply Accumulate* (MAC) werden zwei bzw. drei Eingangsregister bereitgestellt. Im letzteren Fall fungiert das Ausgangsregister *Ry*1 zusätzlich als Eingangsregister. Hierbei sind keine zusätzlichen Schiebeoperationen vorgesehen. Sowohl Multiplikation als auch MAC können in zwei Arten durchgeführt werden: Als reine 32 Bit Operation liefern beide ein Ergebnis im Ausgangsregister *Ry*0. Darüber hinaus ist ein 64 Bit Ergebnis zulässig. Dies wird in Register *Ry*0 und *Ry*1 zurückgegeben.

2. Des Weiteren bilden zwei Speicherzugriffsbefehle eine eigene Befehlsgruppe. Mittels einer Registerliste (*Reg.* − *Liste*) können alle 16 Register des Prozessors mit einem Befehl gelesen oder geschrieben werden. Die Registerliste umfasst 16 Bit, die die einzelnen Prozessorregister repräsentieren. *Ry* gibt die Adresse im Speicherbereich an.

3. Um in den Thumb-Modus und wieder zurück zu wechseln verfügt der ARM über einen *Branch and Exchange*-Befehl (BX). Dieser erlaubt ein einziges Eingangsregister, das die Einsprungsadresse im Anwendungscode angibt. Ob in den Thumb-Modus geschaltet werden soll oder in den 32 Bit Modus, wird anhand des *least significant bit* des Eingangsregisters entschieden.

4. Zuletzt bilden die Sprungbefehle eine eigene Befehlsgruppe. Diese besitzen ausschließlich einen Konstantenwert als Eingabewert, der die Sprungadresse in Form eines vorzeichenbehafteten Offsets übergibt.

Die in Abbildung 2.2 gezeigte Unterteilung in Befehlsgruppen ist bereits in der ARMv4-Befehlssatzarchitektur [5] nicht mehr auf den ersten Blick ersichtlich, ist jedoch grundsätzlich entsprechend aufgebaut. Insgesamt lässt der ARM-Befehlssatz genügend Freiraum für die Integration zusätzlicher Instruktionen, wie diese für den adaptiven Prozessor unbedingt notwendig sind.

Der Platzbedarf für die Bedingung, die Opcode-Gruppe und den Opcode umfasst mindestens elf Bit eines 32 Bit Befehlswortes. Durch das evolutorische Wachsen des Befehlssatzes sind teilweise noch zusätzliche Bits für die eindeutige Unterscheidung von Operationen notwendig. In der Regel stehen jedoch 16 Bit für die Übergabe von Register bzw. Registerindices bereit. Zur Identifikation der 16 Register der ARM-Prozessoren sind entsprechend vier Bits pro

[3] Eigenständige Schiebbefehle fehlen im ARM-Befehlssatz. Dementsprechend muss, falls ausschließlich eine Schiebeoperation ausgeführt werden soll, dies in Kombination mit einer arithmetischen Operation dergestalt geschehen, dass das Ergebnis der Schiebeoperation unverändert bleibt.

Registerindex notwendig. Demzufolge sind für den adaptiven Prozessor Befehlserweiterungen vorstellbar, die über maximal vier Eingangs- und vier Ausgangsregister verfügen, falls jeder Eingangsregisterindex auch einen Ausgangsregisterindex repräsentiert. Diese Vorgehensweise ist empfehlenswert, da dieses Befehlsschema Ähnlichkeit mit den Multiplikations- bzw. MAC-Befehlen aufweist und deshalb nur geringfügige Anpassungen der Dekodiereinheit des adaptiven Prozessors im Vergleich zu der Einheit des ARM-Prozessors erfordert. Sind die vier Ein- oder Ausgangsregister nicht ausreichend, kann eine Lösung mittels Registerlisten, wie diese für *Multiple Load/Store*-Befehle eingesetzt werden, realisiert werden. Darüber hinaus existieren andere Möglichkeiten, die z.B. eine Werteübergabe mit Hilfe von Kellerspeichern vorsehen und häufig beim Einsatz von Coprozessoren Anwendung finden.

2.3.4 Betriebssysteme für eingebettete Systeme

Damit der Entwicklungsaufwand bei komplexeren eingebetteten Systemen reduziert werden kann, kommen verstärkt Standardbetriebssysteme zum Einsatz, die zum Teil für dieses Anwendungsgebiet entwickelt wurden. Hierdurch eröffnet sich die Möglichkeit modulare und abstrakte Softwareimplementierungen anzuwenden, die entsprechend portabler und leichter wartbar sind [2].

Speziell die Prozesssteuerung, als eines der wichtigsten Einsatzgebiete von eingebetteten Systemen, hängt maßgeblich vom Zeitverhalten des zu steuernden Systems ab. Die diesbezüglich notwendige zeitnahe Bearbeitung von Daten wird als Echtzeitbetrieb bezeichnet:

Definition 2.1 *Echtzeitbetrieb (DIN 44300)*

> Echtzeitbetrieb ist ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zufälligen zeitlichen Verteilung oder zu vorbestimmten Zeitpunkten auftreten. □

Dementsprechend werden Betriebssysteme, die für den Echtzeitbetrieb geeignet sind Echtzeitbetriebssysteme genannt. Die Umsetzung von Echtzeitbetriebssystemen garantiert verschiedene Zeitanforderungen. In diesem Zusammenhang unterscheidet man zwischen zwei Arten von Echtzeit [2, 146, 147]:

- **Harte Echtzeit** (*Hard Real Time*): Hat ein Prozess ein Ergebnis nicht bis zu einer vorher festgelegten Endzeit (*strict deadline*) geliefert, kann eine nicht akzeptable Verschlechterung der Systemleistung oder auch ein katastrophales Ereignis eintreten. Verletzungen der Deadline müssen deshalb in jedem Fall vermieden werden.
- **Weiche Echtzeit** (*Soft Real Time*): Berechnungsergebnisse werden möglichst bis zu festgelegten Endzeiten geliefert. Eine Verletzung festgelegter Endzeiten führt zu akzeptablen Verlusten.

Im folgenden werden drei Betriebssysteme beschrieben, die zur Zeit im eingebetteten Bereich weit verbreitet sind und sich als Betriebssysteme für das in Kapitel 5 vorgestellte autonome adaptive Gesamtsystem eignen.

2.3.4.1 PocketPC/ Windows CE

PocketPC/ WindowsCE von Microsoft [119] ist ein auf der Windows-API basierendes Betriebssystem für eingebettete Systeme. Es bietet eine Multi-Tasking-Umgebung und verschiedene Erweiterungen, die es für den Einsatz in mobilen Endgeräten prädestiniert.

Eine weitestgehende Architekturunabhängigkeit wird durch die so genannte *OEM Abstraction Layer* erreicht. Diese umfasst alle Betriebssystemteile die von Prozessor, Plattform oder allgemein von Hardware abhängen.

Neben dem ursprünglichen Haupteinsatzgebiet der mobilen Endgeräte nimmt die Bedeutung von WindowsCE als leichtgewichtiges Betriebssystem auch im industriellen Umfeld zu [88]. Durch das Systemdesign eignet sich WindowsCE jedoch nur bedingt für Anwendungen mit harter Echtzeit.

2.3.4.2 Linux

Aktuell unterstützt Linux die meisten Prozessoren. Zudem ist die individuelle Anpassungsfähigkeit von Linux ein Vorteil gegenüber den anderen Betriebssystemen. Hierbei stellt Linux die essentielle Funktionalität (virtueller Speicher, Prozess und Prozess(inter)kommunikation usw.) bereit und erlaubt darüber hinaus optionale Komponenten (z.B. Festplattenansteuerung mittels IDE etc.) je nach Verwendungszweck und Einsatzgebiet in den Kernel zu compilieren [2].

Linux ist ebenfalls nur beschränkt für den Einsatz unter harten Echtzeitbedingungen geeignet. Es existieren jedoch Erweiterungen, welche die Verwendung in Echtzeitsystemen erlauben (z.B. RTLinux [53]).

2.3.4.3 VxWorks

WindRiver stellt mit VxWorks [153] ein leichtgewichtiges und netzwerkfähiges Echtzeitbetriebssystem zur Verfügung. VxWorks arbeitet intern mit einem Microkernel, der nach Bedarf mit zusätzlichen Modulen erweitert werden kann. Der Kern verfügt über ein starres und entsprechend deterministisches *Task-Scheduling* und umfasst verschiedene Interprozesskommunikationsmechanismen.

2.4 Entwurf von Niedrigenergie- und Energiespar-Systemen

Dieser Abschnitt führt die grundlegenden Mechanismen der heutzutage üblicherweise eingesetzten Techniken zur Entwicklung von Niedrigenergie- und Energiespar-Systemen ein. Bis vor wenigen Jahren war die Betrachtung der Energie eines Systems ausschließlich für eingebettete Systeme von Bedeutung. In zunehmendem Maße trifft dies auf Grund der hohen Integrationsdichten und Taktraten von Prozessoren oder allgemein Komponenten für beinahe jedes Rechensystem zu. Höhere Integrationsdichten führen zu einem Anstieg der Leckströme. Mit zunehmender Taktrate erhöht sich die dynamische Leistungsaufnahme (siehe Abschnitt 2.4.1.1).

Als grundsätzliche Forderung beim Entwurf von energiesparenden Systemen oder Niedrigenergie-Systemen ist die Optimierung des Verhältnisses von Verlustleistung zu Rechenleistung zu betrachten. Hierfür eignet sich die Metrik Energie pro Instruktion (EPI) (Vgl. auch [63, 69, 70]):

$$\frac{Joule}{Instruktion} = \frac{\frac{Joule}{Sekunde}}{\frac{Instruktion}{Sekunde}} = \frac{Watt}{IPS} \tag{2.8}$$

Das heißt, nach Gleichung 2.8 kann Energie pro Instruktion (EPI) bzw. Watt pro Instruktion pro Sekunde gleichbedeutend verwendet werden. Beide Metriken eignen sich für Systeme, in denen Energieeffizienz in Verbindung mit Rechenleistung bzw. Instruktionsdurchsatz von Bedeutung sind. In der Literatur findet sich anstelle von $Watt/IPS$ oft der reziproke Ausdruck $IPS/Watt$ bzw. $MIPS/Watt$. $IPS/Watt$ ist als Metrik für die Energieeffizienz leichter nachvollziehbar, da hierbei ein höherer Wert eine höhere sowie ein niedrigerer Wert eine niedrigere Energieeffizienz darstellt.

Sowohl EPI als auch MIPS/Watt berücksichtigen nicht die Zeit, die für die Ausführung einer Instruktion benötigt wird. Für die Bewertung von Systemen, in denen die Ausführungszeit maßgeblich ist, wird die Metrik MIPS/Watt dahingehend erweitert, dass die Ausführungsverzögerung einfach oder quadratisch in den Ausdruck einfließt. Daraus ergibt sich die Metrik $MIPS^2/Watt$ respektive $MIPS^3/Watt$ [63, 62].

Laut Havinga und Smit [70] ist die Verwendung von EPI und davon abgeleiteten Metriken für den Vergleich von Rechensystemen nicht angemessen: Der Faktor Zeit und die Wirksamkeit von Instruktionen werden nicht entsprechend berücksichtigt. Angenommen ein Prozessor A hat einen doppelt so hohen Strombedarf wie ein Vergleichsprozessor B. Die Energieeffizienz von A kann dennoch besser als die von B sein, falls A eine Aufgabe in weniger als der Hälfte der Zeit bewältigt, die Prozessor B benötigt. Die Energieeffizienz sollte somit unabhängig von der Anzahl der ausgeführten Instruktionen sein. Demzufolge wird die *Energieeffizienz* wie folgt definiert:

Definition 2.2 *Energieeffizienz*

Die Energieeffizienz entspricht der theoretisch notwendigen Energie zur Bearbeitung einer bestimmten Aufgabe geteilt durch die tatsächlich benötigte Energie.

$$Energieeffizienz = \frac{Essentielle\ Energie\ für\ eine\ Aufgabe}{Tatsächlich\ benötigte\ Energie}$$

□

Die *essentielle Energie* ergibt sich laut Havinga und Smit aus der Leistungsaufnahme, die dadurch entsteht, dass die Signale, die ausschließlich zur Abarbeitung einer Aufgabe notwendig sind, übertragen werden. Im Gegensatz dazu fließen in die *tatsächlich benötigte Energie* alle zusätzlichen Steuerungssignale usw. mit ein.

Grundsätzlich scheint die Bestimmung der essentiellen Energie für die Abarbeitung einer Aufgabe als Basis eines system-übergreifenden Vergleichs geeignet. Wie die essentielle Energie im Detail ermittelt werden kann, bleibt jedoch offen.

2.4.1 Drei Ebenen des Systementwurfs

Für den energieeffizienten Entwurf eines Rechensystems stehen unterschiedliche Möglichkeiten zur Verfügung: Grundsätzlich sollten bereits zur Designzeit entsprechende Maßnahmen getätigt werden. Darunter fallen Entscheidungen zur Auswahl der Technologie, der Umsetzung von Hardware-Komponenten oder Vorkehrungen zur Unterstützung von dynamischen Maßnahmen der zugehörigen Software-Komponenten usw. Um ein energieeffizientes System zu erhalten, muss der Energiebedarf auf allen Ebenen Beachtung finden. Der Begriff System wird hierbei in Anlehnung an die DIN 19226 wie folgt definiert:

Definition 2.3 *System*

> Ein System ist eine in einem betrachteten Zusammenhang gegebene Anordnung von Elementen, die miteinander in Wechselwirkung stehen. Diese Anordnung wird auf Grund bestimmter Vorgaben gegenüber ihrer Umgebung abgegrenzt. □

Demzufolge werden unter den für diese Arbeit relevanten Systemen, den Rechensystemen (oder im Weiteren auch kurz Systeme), alle Hardware-Elemente bzw. Hardware-Komponenten, wie z.B. Peripheriegeräte, Prozessoren, Bussysteme usw., sowie zugehörige Software-Komponenten, wie z.B. Betriebssystem, Treiber etc., verstanden. Unter Berücksichtigung der Systemdefinition ist es zulässig, dass einzelne Komponenten wiederum als eigenständiges System betrachtet werden. Dies gilt für den in Kapitel 6 vorgestellten adaptiven Prozessor.

Entsprechend beschreiben Systemparameter die Eigenschaften eines Systems bzw. seiner Elemente. Die gemeinsame Ausprägung dieser Parameter zu einem definierten Zeitpunkt wird als *Zustand des Systems* bezeichnet. Der Zustand des Systems beinhaltet auch den Zustand seiner Elemente [51].

Havinga und Smit [70] teilen den Entwurfsablauf für Niedrigenergie- und Energiespar-Systeme in drei Abstraktionsebenen auf:

1. Die **System-Ebene** behandelt die Möglichkeiten im Hinblick auf die Energieeffizienz der einzelnen im Allgemeinen heterogenen Systemressourcen bzw. Systemkomponenten.

2. Die **Logik-Ebene** beinhaltet Methoden zur Energieoptimierung in Bezug auf Zustandsmaschinen, Dekodierung von Signalen usw.

3. Die **Technologie-Ebene** umfasst unter anderem Entscheidungen über den Formfaktor, Fertigungsprozesse sowie Layout und über die Verwendung asynchroner Logik im Gegensatz zu synchroner Logik.

Die System-Ebene repräsentiert die Ebene mit dem höchsten Abstraktionsniveau. Das niedrigste Abstraktionsniveau findet sich in der Technologie-Ebene. Dementsprechend sind die möglichen Ansätze für eine energieeffiziente Umsetzung innerhalb der Technologie-Ebene am konkretesten. Im Folgenden werden die für diese Arbeit interessanten Ansätze der einzelnen Ebenen herausgegriffen und in Form eines kurzen Überblicks dargestellt. Eine detaillierte Beschreibung der gängigsten Methoden kann z.B. in [70] nachgelesen werden.

2.4.1.1 Die Technologie-Ebene

Durch die weite Verbreitung CMOS-basierter Bausteine beschränkt sich der Abschnitt ausschließlich auf diese Technologie. Für CMOS-basierte Bausteine entfällt der überwiegende Teil auf die dynamische Leistungsaufnahme. Die dynamische Leistungsaufnahme P_d von CMOS-Schaltungen kann wie folgt angenähert werden:

$$P_d = C_{eff} V^2 f \tag{2.9}$$

P_d repräsentiert die Leistung in Watt. C_{eff} ist die effektive Umladungskapazität in Farad, V die Versorgungsspannung in Volt und f die Frequenz in Hertz. Hierbei ist die Leistungsaufnahme maßgeblich vom Laden und Entladen der Kapazität C_{eff} abhängig. C_{eff} wiederum wird von zwei Faktoren bestimmt: Der Kapazität C, die geladen oder entladen wird, und dem Aktivitätsgewicht α, das die Wahrscheinlichkeit der Lade- bzw. Entladevorgänge je Taktzyklus angibt:

$$C_{eff} = \alpha C \tag{2.10}$$

Entsprechend der beiden Gleichungen 2.9 und 2.10 ergeben sich vier Ansatzpunkte zur Verringerung der Leistungsaufnahme innerhalb der Technologieebene:

- Verminderung der Kapazität C.
- Verminderung der Versorgungsspannung V.
- Verminderung der Taktfrequenz f.
- Verminderung der Umladungsaktivität α.

Verringerung der Kapazitäten

Häufig entfällt ein Großteil des kapazitiven Energiebedarfs eines Bausteins auf die Notwendigkeit externe Systemkomponenten zu treiben. In der Regel liegen die externen Kapazitäten um Größenordnungen höher als die innerhalb eines Bausteins. Demzufolge kann der Energiebedarf durch die Minimierung der Pin-Anzahl oder zumindest der Kommunikation mit externen Komponenten verringert werden.

Eine weitere Möglichkeit der Kapazitätsreduzierung besteht darin, die Chip-Fläche zu verkleinern. Die Verkleinerung der Chip-Fläche beschränkt die verfügbaren Hardware-Ressourcen. Dies führt dazu, dass effektivere Methoden zur Energieoptimierung aus den darüber liegenden Abstraktionsebenen nicht oder nur zum Teil angewendet werden können.

Darüber hinaus kann durch ein entsprechendes Chip-Layout, das z.B. die Signalaktivität und Leitungslänge miteinbezieht, eine Verringerung des Energiebedarfs um bis zu 18% erreicht werden.

Regulierung der Versorgungsspannung und Taktfrequenz

Eine der effektivsten Methoden, den Energiebedarf eines integrierten Schaltkreises zu senken, ist die Verringerung der Versorgungsspannung. Reduziert man z.B. eine Versorgungsspannung von 5,0 Volt auf 3,3 Volt, verringert sich die Leistungsaufnahme um ca. 56%. Die Verringerung der Versorgungsspannung resultiert in einer Erhöhung interner Verzögerungen, die, um eine korrekte Ausführung zu gewährleisten, durch eine zusätzliche Verringerung der Taktfrequenz kompensiert werden müssen. Eine niedrigere Taktfrequenz führt jedoch zu einer Verschlechterung der Rechenleistung. Die Zeit für die Abarbeitung der gleichen Anzahl von Operationen nimmt entsprechend der Abnahme der Taktfrequenz zu.

Vermeidung unnötiger Signalaktivität

Der letzte Ansatzpunkt zur Reduzierung des Energiebedarfs innerhalb der Technologieebene ist eine Verminderung der Signalaktivität. Diesbezüglich gibt es drei Ursprünge unnötiger Signalaktivität:

- Transitionen die auf Grund ungleicher Signallaufzeiten (*Propagation Delays*) auftreten (*Glitches*).
- Signalaktivitäten in Einheiten, die nicht an der Berechnung teilhaben.
- Änderungen in Einheiten, deren Berechnungsergebnis redundant ist.

Unnötige Signalaktivität kann z.B. durch die Umordnung von Schaltungseingängen oder durch asynchrone Schaltungsdesigns verringert oder gänzlich vermieden werden.

2.4.1.2 Die Logik-Ebene

Clock-Gating

Da sich die Leistungsaufnahme von CMOS-Bauteilen proportional zur Taktfrequenz verhält, eignen sich zur Verringerung des Energiebedarfs Mechanismen die jeweils ungenutzte Einheiten temporär vom Systemtakt entkoppeln. Dies wird als *Clock-Gating* bezeichnet. Für den Einsatz von Clock-Gating in Prozessoren sind folgende Ansätze interessant:

Das von Li et al. [103] beschriebene *Deterministic Clock-Gating* (DCG) mit durchschnittlich 19% geringerem Energiebedarf koppelt Pipelinestufen erst dann an den Systemtakt, falls diese durch die aktuell verarbeitenden Befehle genutzt werden.

Ein anderer *Clock-Gating* Ansatz ist das *Pipeline Balancing* (PLB) mit einer Einsparung von durchschnittlich 9,9% und einer ca. 3% Verminderung der Rechenleistung [22, 103]. Die Balancierung der Pipeline wird mit einer Vorhersage des Parallelismus auf Instruktionsebene (*Instruction Level Parallelism*, ILP) auf der Grundlage eines 256 Zyklen langen Vorschaufensters zu erreichen versucht. Entsprechend der Vorhersage folgt ein An- bzw. Abschalten von voraussichtlich betroffenen und zusammenhängenden Pipeline-Komponenten.

Pipelining

Wilton et al. [152] zeigen, dass sich die Pipelinetiefe massiv auf den Energiebedarf von FPGA-Schaltungen auswirkt. Je mehr Stufen eine Hardware-Komponente besitzt, um so

geringer ist der dynamische Energieverbrauch. Der statische Anteil nimmt mit steigender Tiefe zu. Die Zunahme des statischen Energiebedarfs ist jedoch im Verhältnis zur Verringerung des dynamischen Anteils nicht maßgeblich. Dementsprechend kann durch eine Erhöhung der Anzahl der Pipeline-Stufen eine Reduktion des Energiebedarfs pro Operation zwischen 40 und 90% erreicht werden.

Modifikation von Zustandsmaschinen

Kommen Zustandsmaschinen während des Entwurfsprozesses von logischen Schaltungen zum Einsatz, können diese durch Umstrukturierung oder Dekomposition in ihrer Energieeffizienz verbessert werden. Beide Methoden zielen darauf ab, die Signalaktivität der resultierenden Verbindungsleitungen zu minimieren.

Zudem zeigt sich, dass die Bit-Kodierung der einzelnen Zustände enorme Auswirkungen auf die Qualität – in Bezug auf Größe, Geschwindigkeit, Energiebedarf usw. – der zu erzeugenden Schaltung hat (Vgl. z.B. [47]).

Logische Kodierung

Allgemein wirkt sich die gewählte Kodierung von Informationen auf die Leistungsaufnahme einer Schaltung aus. In diesem Zusammenhang ist die oberste Prämisse die Kodierung so zu wählen, dass während der Laufzeit möglichst wenige Transitionen ausgeführt werden müssen.

Sind keine Daten über das zeitliche Verhalten und den Inhalt der zu übertragenden Information vorhanden (z.B. zufällige Speicherzugriffe), kann beispielsweise dynamisch, auf der Grundlage des Hamming-Abstandes der Bit-Muster aufeinander folgender Daten, eine energieeffizientere Kodierung gewählt werden. Solange der Hamming-Abstand sequentieller Daten kleiner als $N/2$ ist, erfolgt eine unveränderte Übertragung. Für einen Abstand größer als $N/2$ werden die Daten invertiert gesendet und somit wiederum nur maximal $N/2$ Transitionen benötigt. Hierzu muss jedoch eine zusätzliche Signalleitung bereitgestellt werden, die die Art der gewählten Übertragung anzeigt.

Data Guarding

Data Guarding bezeichnet eine Methode zur Erkennung und Filterung redundanter Eingangsinformation. Hierfür wird einer Einheit zusätzliche Logik (*guard logic*) vorgelagert. Diese erkennt, ob die Eingangsdaten invariant zu den Ausgangsdaten sind und unterbindet gegebenenfalls die Weiterleitung der Eingangssignale.

Darüber hinaus kann die Position von Registern innerhalb einer logischen Schaltung deren Größe und Leistung stark beeinflussen. Die Umpositionierung von Registern einer Schaltung ohne deren Verhalten zu verändern wird als *Retiming* bezeichnet. Diese Vorgehensweise dient der Filterung und Reduzierung von Glitches.

2.4.1.3 Die System-Ebene

Programmierbarkeit und Rekonfigurierbarkeit der Recheneinheiten

In Bezug auf Prozessoren und Recheneinheiten eines Systems zeigt sich ein Dualismus zwischen Flexibilität und Effizienz im Hinblick auf Rechenleistung und Energieeffizienz. Havinga und Smit [70] unterteilen in diesem Zusammenhang die üblicherweise eingesetzten Recheneinheiten in folgende Klassen:

- **General-Purpose Prozessoren** bringen die größtmögliche Flexibilität, wobei die Energieeffizienz und Rechenleistung bis auf wenige Ausnahmen weit hinter den anderen Kategorien liegt.
- **Anwendungsbereich-spezifische Recheneinheiten** sind in ihrer Rechenleistung und Leistungsaufnahme besser als General-Purpose Prozessoren, können jedoch nur innerhalb bestimmter Anwendungsbereiche zum Einsatz gebracht werden.
- **Anwendungsspezifische Recheneinheiten** sind ausschließlich für eine Anwendung entwickelt und dementsprechend stark in ihrer Wiederverwendbarkeit eingeschränkt. Im Gegensatz dazu übertreffen diese Recheneinheiten im Allgemeinen die beiden obigen Kategorien sowohl in Bezug auf die erreichte Energieeffizienz als auch der resultierenden Rechenleistung.
- **Rekonfigurierbare Recheneinheiten** oder **Module** bieten die Möglichkeit sowohl anwendungsbereich-spezifische als auch anwendungsspezifische Aspekte gleichermaßen in einem System zu nutzen. Bis vor wenigen Jahren sorgte jedoch die Leistungsaufnahme der verfügbaren rekonfigurierbaren Elemente dafür, dass diese grundsätzlich für die Integration in kommerziellen Rechensystemen abgelehnt wurden [50, 107]. Zwischenzeitlich hat sich der Energieaspekt deutlich verbessert und es kristallisieren sich die rekonfigurierbaren Recheneinheiten als direkte Konkurrenten zu den anwendungs(bereich)-spezifischen Recheneinheiten heraus. Die Rechenleistung der rekonfigurierbaren Recheneinheiten ist meist geringfügig schlechter. Dementgegen steht eine wesentlich größerer Flexibilität durch die Möglichkeit der (dynamischen) Rekonfiguration (Vgl. auch [141, 110, 96]).

Betriebssysteme

Das Betriebssystem eines Rechensystems kann in vielfältiger Weise positiv auf die Energieeffizienz des Systems einwirken [70, 46, 34, 133]: Hierunter fallen z.B. diverse Speicheroptimierungstechniken, Kommunikationsoptimierung zwischen Systemkomponenten usw. Für den aktuellen Stand der vorliegenden Arbeit sind diese Methoden noch nicht von Bedeutung, müssen aber in weiterführenden Arbeiten Berücksichtigung finden (siehe Kapitel 7.1).

Im Folgenden wird ein Überblick über die für diese Arbeit bis dato relevanten Methoden gegeben:

Dynamische Spannungs- und Frequenzanpassung: Die dynamische Spannungs- und Frequenzanpassung (*Dynamic Voltage Scaling*, DVS) muss grundsätzlich in den einzelnen Systemkomponenten verankert (Vgl. [105, 132]) sein und wurde bereits in Abschnitt 2.4.1.1

über die Mechanismen der Technologieebene eingeführt. Das heißt, dass die physikalischen Komponenten eines Systems die notwendigen Fähigkeiten und entsprechende Hardware (z.B. Phase Locked Loop, PLL) zur Verfügung stellen müssen, um eine dynamische Spannungs- und Frequenzanpassung durchführen zu können. Häufig reicht jedoch die Systemsicht einer einzelnen Komponente für eine Regelung nicht aus, da z.B. dem Prozessor eines Systems keine Angaben über die Vorgaben und Zielsetzungen einer Anwendung bekannt sind.

Dynamisches Power-Management: Als dynamisches Power-Management (DPM) wird die Fähigkeit eines Systems bezeichnet, die ein gezieltes An- und Abschalten einzelner Systemteile zur Folge hat (Vgl. [105]). Hierbei überwacht das Betriebssystem die Ausnutzung der Systemkomponenten und nutzt deren Leerlaufzeiten (*idle time*) zur Abschaltung. Diesbezüglich wird in Kauf genommen, dass es bei einer Reaktivierung zu Zeitverzögerungen bei der Anwendungsausführung kommt. Systemkomponenten unterliegen einer gewissen Einschwingzeit, bis diese wieder einen stabilen elektrischen Zustand erreicht haben und somit rechenbereit sind. In diesem Zusammenhang kommen vermehrt Regelungen auf Basis statistischer Verfahren, die sich in die Kategorie der *Tracking*-Verfahren einreihen (siehe Abschnitt 3.5), zum Einsatz, um eine Minimierung etwaiger Zeitverzögerungen zu erhalten.

Einsatz von DVS und DPM bei der Anwendungszuteilung: Die Anwendungs- oder Prozess-Zuteilung (*Scheduling*) teilt den aktuell auf dem System ausgeführten Anwendungen (Prozessen) Rechenzeit – üblicherweise in gleichlangen Zeitscheiben – zu. Diesbezüglich stehen prinzipiell zwei Möglichkeiten zur Verfügung den Strombedarf des gesamten Systems zu reduzieren respektive die Energieeffizienz des Systems zu erhöhen (Vgl. auch [90]):

- Der *Scheduler* protokolliert die tatsächlichen Nutzungsphasen und die sich ergebenden Leerlaufzeiten der Systemkomponenten. Entsprechend werden Systemteile auf Basis dieser Information an- bzw. abgeschaltet.
- Für jeden Prozess wird die notwendige Rechenleistung errechnet, die dieser benötigt, um die bereitgestellte Zeitscheibe optimal auszunutzen [45]. Falls ein Prozess z.B. mit 100 % Rechenleistung 50 % der zugeteilten Zeitscheibe für die Berechnung benötigt, würde die Rechenleistung bzw. Frequenz halbiert werden. Demzufolge beansprucht der Prozess die gesamte Zeitscheibe. Die Berechnung erfolgt jedoch nur mit der halben Frequenz und trägt somit zur Energieeffizienz des Systems bei.

Sowohl DVS als auch DPM werden in der Regel nebeneinander in einem System eingesetzt. DPM unterliegt – im Gegensatz zu DVS – verhältnismäßig langen Reaktivierungszeiten abgeschalteter Systemteile. Das heißt, dass die Abschaltung von Systemteilen nur dann in Frage kommt, wenn *keine* kritischen Anforderungen in Bezug auf die Reaktionszeiten des Systems gelten.

3.

Regelung von Systemen

Dieses Kapitel klärt die Begrifflichkeiten der Regelungstechnik, die in der vorliegenden Arbeit Verwendung finden. Des Weiteren wird eine Einführung in die statistischen Verfahren der Regelungstechnik geliefert. Zum einen gewinnen diese Verfahren immer mehr an Bedeutung und zum anderen eignen sie sich, um die Abläufe im betrachteten autonomen adaptiven Gesamtsystem und dem adaptiven Prozessor zu beschreiben. Dennoch müssen die Verfahren für eine Hardware-Umsetzung in Komponenten des autonomen adaptiven Gesamtsystems vereinfacht und adaptiert werden (siehe Kapitel 5). Abschließend steht die Einordnung der implementierten Hardware-Regelung in die als Tracking (deutsch: Nachführung) bezeichnete Vorgehensweise.

In der Norm DIN 19226 ist der Begriff der Regelung wie folgt definiert:

Definition 3.1 *Regelung (DIN 19226)*

> Das Regeln, die Regelung, ist ein Vorgang, bei dem fortlaufend eine Größe, die Regelgröße (zu regelnde Größe), erfasst, mit einer anderen Größe, der Führungsgröße, verglichen und im Sinne einer Angleichung an die Führungsgröße beeinflusst wird. □

Merkmal des Regelns ist der geschlossene Wirkungsablauf (*closed-loop*), bei dem die Regelgröße im Wirkungsweg des Regelkreises fortlaufend sich selbst beeinflusst. Eine (physikalische) Größe, die so genannte *Regelgröße* eines Systems soll einem bestimmten zeitlichen Verlauf folgen. Dies kann auch bedeuten, dass die Größe konstant gehalten werden muss. Hierfür stehen im Allgemeinen Eingriffsmöglichkeiten in Form von verstellbaren Parametern (*Stellgrößen*) zur Verfügung. Die Stellgrößen werden auf Grund von fortlaufenden Messungen des realen Verhaltens der Regelgröße und deren Abweichung vom Sollverlauf (*Regelfehler*) angepasst. Dies ermöglicht eine Reaktion auf unbekannte externe Einflüsse, so genannte *Störgrößen*. Im Gegensatz dazu wird eine Manipulation der Stellgrößen ohne Kenntnis des realen Regelgrößenverlaufs als *Steuerung* bezeichnet [134].

3.1 Regelkreis und Wirkungsplan

Die einzelnen Glieder des Regelkreises sind nach DIN 19226 durch rechteckige Kästchen, *Block* genannt, symbolisiert (Abbildung 3.1). Ein- und Ausgangssignale werden durch Wirkungslinien dargestellt. Die Pfeilspitzen zeigen die *Wirkungsrichtung*. Treffen mehrere Signale an einer Stelle zusammen, wird diese durch eine kreisförmige *Additionsstelle* verdeutlicht.

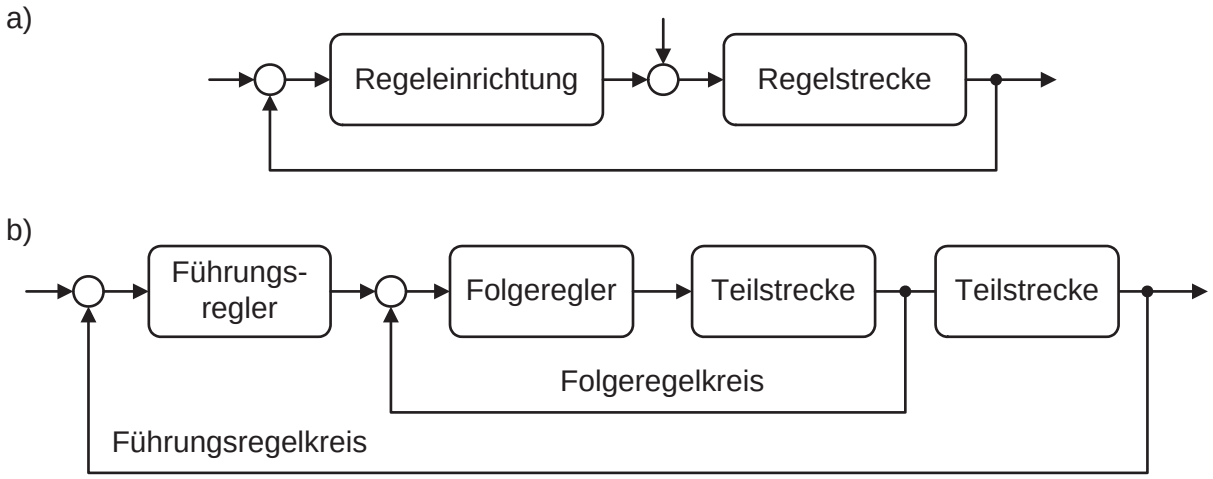

Abbildung 3.1: Vereinfachte Wirkungspläne

Signalverzweigungen sind durch Punkte auf den Wirkungslinien gekennzeichnet [122]. Die Darstellungen in Abbildung 3.1, welche die wirkungsmäßigen Zusammenhänge zwischen den Signalen wiedergibt, ohne gerätetechnische Einzelheiten zu berücksichtigen, wird nach DIN 19226 als Wirkungsplan bezeichnet.

Generell lässt sich ein Regelkreis in zwei Bereiche unterteilen [122]: Ein Bereich der durch die zu regelnde Anlage gegeben ist und als *Regelstrecke* bezeichnet wird. Der Bereich der *Regeleinrichtung* der die eigentliche Regelung vornimmt. Zur Regeleinrichtung zählen Messfühler, Messumformer, Regler usw., die für die Regelung notwendig sind. Teilbild 3.1 a zeigt einen vereinfachten Wirkungsplan. Ein kaskadierter Wirkungsplan ist in Teilbild 3.1 b skizziert.

3.2 Regler

Regler vergleichen kontinuierlich einen von Sensoren durch Messen ermittelten Istwert mit einem Sollwert (der Führungsgröße) und ermitteln aus dem Unterschied der beiden Größen (der so genannten *Regeldifferenz* bzw. *Regelabweichung*) eine Stellgröße, welche an die Steuerung weitergegeben wird. Ziel ist die Annäherung des Istwertes an den zu erreichenden Sollwert.

Wird der Sollwert erreicht bleibt das System nicht unbedingt in einem stabilen Zustand. Eine Überschreitung oder Unterschreitung des Sollwertes muss mit jeweils entgegengesetzten Regeleingriffen ausgeglichen werden. Nachdem der Sollzustand stabil erreicht ist, bleibt das System in seinem Zustand, bis durch Belastung oder veränderte Anforderungen eine Veränderung des Ist- oder Sollwerts auftritt [151].

Regler werden anhand des Wertebereichs der ausgegebenen Stellgrößen als stetige oder unstetige Regler charakterisiert [122]:

- **Stetige Regler** können jeden Wert des Stellbereichs annehmen. Die elementaren P- (Proportional), I- (Integral), D- (Differential), PI-, PID- usw. Regler sind stetige Regler (Vgl. auch [150, 21]).

- **Unstetige Regler** nehmen wenige diskrete Werte an. Der Zweipunktregler (z.B. Bimetallregler), der Dreipunktregler, Fuzzy-Regler usw. repräsentieren unstetige Regler (Vgl. auch [117, 21]).

Darüber hinaus unterscheidet man Regelsysteme auf Grund ihrer Systemantwort [100]:

- **Stabile Systeme**: Bei Anregung mit beschränkten Stellgrößen wird mit beschränkten Ausgangssignalen geantwortet. (BIBO-Stabilität; Bounded Input, Bounded Output)
- **Instabile Systeme**: Anregung mit beschränkten Signalen kann unbeschränkte Ausgangssignale erzeugen.

In diesem Zusammenhang stellt das Regelsystem des autonomen adaptiven Gesamtsystems als auch des adaptiven Prozessors jeweils ein *stabiles System* dar. Die Regelung basiert auf unstetigen Reglern mit beschränkten Stell- und Ausgangsgrößen. Gelten keine Beschränkungen der Stell- und Ausgangsgrößen, muss die Stabilität mittels mathematischen Modellen nachgewiesen werden [150].

3.3 Digitale Regler

Dieser Abschnitt gibt einen Überblick über die in der Praxis eingesetzten digitalen Regler. Ausgenommen sind Regler auf Basis statistischer Verfahren die gesondert in Abschnitt 3.4 behandelt werden.

3.3.1 Fuzzy-Regler

Fuzzy-Systeme sind Systeme, die Methoden der Fuzzy-Logik (deutsch: unscharfe Logik) bzw. Fuzzy-Arithmetik nutzen. Fuzzy-Logik ist eine Erweiterung der klassischen Logik auf Wahrheitswerte im gesamten Einheitsintervall $[0, 1]$. Auf diese Weise lassen sich Sachverhalte wesentlich differenzierter darstellen als in der klassischen Logik, die nur die Werte **wahr** und **falsch** kennt. Analog ist Fuzzy-Arithmetik eine Erweiterung der klassischen Arithmetik auf so genannte Fuzzy-Zahlen. Eine Anwendung der Fuzzy-Logik sind Fuzzy-Regler (*Fuzzy-Controller*). Sie kommen u.a. bei der Regelung von eingebetteten System, wie z.B. Brennöfen, Viedeokameras usw., zum Einsatz [117].

Fuzzy-Controller haben die Eigenschaft, dass sie sich einfach mit Expertenwissen und linguistischen Regeln entwickeln lassen und für einen Menschen (relativ) leicht zu verstehen sind, da sie aus WENN-DANN-Regeln aufgebaut sind. Die Nachteile sind, dass der so entstehende Fuzzy-Controller problemspezifisch ist und sich Umgebungsveränderungen nicht selbstständig anpassen kann. Die Umsetzung des Expertenwissens zur Designzeit legt den Fuzzy-Controller und seine Qualität fest. Ist das Expertenwissen unvollständig, ist es nicht möglich einen guten Controller zu erhalten. Alle Regeln des Controllers müssen vor dessen Einsatz bekannt sein [101, 117].

3.3.2 Neuronale Netze

Bei Künstlichen Neuronalen Netzen (KNN) handelt es sich um informationsverarbeitende Systeme, die sich an biologischen Nervensystemen von Lebewesen orientieren. Hierbei

steht nicht im Vordergrund, den Aufbau und die Funktionsweise eines Gehirns möglichst naturgetreu nachzubilden. Stattdessen wird das Wissen über biologische Neuronale Netze genutzt, um dort erfolgreiche Funktionalitäten künstlich nachzubilden. Analog zum biologischen Vorbild bestehen Künstliche Neuronale Netze aus sehr vielen einzelnen Einheiten, den so genannten Neuronen, die im Vergleich zum Gesamtsystem jeweils sehr einfach aufgebaut sind. Diese Neuronen sind untereinander hochgradig verbunden. Durch gerichtete Verbindungen wird die Kommunikation abgewickelt. Auf diese Weise ist es möglich, durch Künstliche Neuronale Netze mittels einfacher arithmetischer Funktionen sehr komplexe Eingabe-Ausgabe-Zusammenhänge abzubilden [37].

Künstliche Neuronale Netze stellen sich im Allgemeinen nicht problemspezifisch dar. Sie benötigen lediglich eine Struktur und ein entsprechendes Lernverfahren. Die Bearbeitung eines speziellen Problems wird mit Hilfe von Lernprozessen anhand ausgesuchter Trainingsbeispiele *anerzogen* (Vgl. [117, 134, 158]).

3.3.3 Erweiterungen Fuzzy-basierter Regler

Um den Nachteil der schlechten Anpassungsfähigkeit von Fuzzy-Reglern auszugleichen, werden grundsätzlich zwei Ansätze verfolgt, die beide zur Flexibilisierung von Fuzzy-Reglern beitragen:

- Neuro-Fuzzy-Regler: Mit Hilfe eines Neuronalen Netzes wird ein Fuzzy-Regler analysiert und optimiert. Das Neuronale Netz liefert nach vorhergehenden bereits bekannten Lernverfahren einen optimierten Fuzzy-Regler. Das Neuronale Netz ist kein Bestandteil des resultierenden Reglers. Treten Umgebungsänderungen auf, muss der Regler erneut analysiert und optimiert werden (Vgl. [117]).

- Genetische Fuzzy-Regler: Entsprechend den Neuronalen Netzen dienen hier genetische bzw. evolutionäre Algorithmen zur Optimierung eines Fuzzy-Reglers. Evolutionäre Algorithmen unterliegen einem gemeinsamen Konzept, das darauf beruht einen Evolutions-Zyklus so lange zu durchlaufen, bis eine optimale Lösung gefunden wurde. (Vgl. [32, 24, 33, 142]).

3.4 Regelungsverfahren auf Basis statistischer Zeitreihenanalyse

Bedingt durch die zunehmende Komplexität technischer Prozesse sowie die steigenden Anforderungen an die Qualität von Regelungsprozessen gewinnen statistische Methoden immer mehr an Bedeutung [74, 52]. Im Folgenden wird eine Einführung in die Grundlagen der statistischen Zeitreihenanalyse sowie ein Auszug der darauf aufbauenden Prognoseverfahren gegeben.

3.4.1 Statistische Zeitreihenanalyse

Eine Zeitreihe besteht aus einer geordneten Folge von Beobachtungen y_t eines Merkmals Y, die über einen Zeitraum hinweg erfolgen. Die Zeitpunkte $t = 1, ..., n$ können – müssen aber nicht – äquidistant sein.

Zur Identifikation von Abhängigkeiten, Entwicklungen usw. wird in der Statistik ein Modell konstruiert, mit dem die beobachtete Zeitreihe beschrieben werden kann. Hierfür gibt es zwei grundsätzlich verschiedene Modellformen: Zum einen das deterministische Modell, das in Form des Komponentenmodells in Abschnitt 3.4.1.1 beschrieben wird und zum anderen das in Abschnitt 3.4.1.2 erläuterte stochastische Modell. Die Eigenschaften des jeweiligen Modells sollten dabei mit denen der beobachteten Zeitreihe möglichst genau übereinstimmen, so dass die Werte der Reihe auch durch das Modell erzeugt worden sein könnten.

Der Modellbildungsprozess lässt sich in vier Phasen einteilen [37]:

- In der **Identifikationsphase** wird die Zeitreihe beispielsweise durch grafische Aufbereitung oder statistische Tests auf ihre Eigenschaften hin untersucht. Auf Grund dieser Eigenschaften und der weitergehenden Zielsetzung wird das grundsätzliche Modell zur Beschreibung der Zeitreihe gewählt.

- In der **Schätzphase** werden die Parameter des gewählten Modells beispielsweise durch Kalman-basierte Methoden oder ARMA-basierte (*Autoregressive Moving Average*) Methoden (z.B. die Box-Jenkins-Methode) geschätzt.

- In der **Diagnosephase** werden die geschätzten Parameter des Modells über Visualisierung oder statistische Tests überprüft. Liegen verschiedene Modellalternativen vor, so wird hier diejenige ausgewählt, die die Zeitreihe am besten erklärt.

- In der **Einsatzphase** wird das spezifizierte Modell verwendet, um das vorher festgelegte Ziel der Zeitreihenanalyse wie beispielsweise die Prognose zukünftiger Werte der Zeitreihe zu erreichen.

3.4.1.1 Komponentenmodelle

Liegen in den erhobenen Werten Regelmäßigkeiten vor, kann man die daraus resultierende Zeitreihe (evtl. als Zusammensetzung einzelner Bestandteile) beschreiben. Hierfür wird die Zeitreihe beispielsweise auf einen Trend oder auf periodisch wiederkehrende Schwankungen hin untersucht. Diese werden durch verschiedene Methoden quantifiziert, so dass sich die ursprüngliche Zeitreihe in ihre additiv bzw. multiplikativ verknüpften Komponenten zerlegen lässt. Gängige Komponenten einer solchen Zerlegung sind [125]:

- Ein Trend m_t , der die langfristige Veränderung des Niveaus der Zeitreihe ergibt.

- Ein Zyklus k_t (oft auch Konjunkturkomponente), der wiederkehrende, nicht notwendigerweise regelmäßige Schwankungen, wie beispielsweise die wirtschaftliche Konjunktur, beschreibt.

- Die Saison s_t, die Schwankungen mit regelmäßiger Periode repräsentiert.

- Der Rest u_t, der als übrig bleibende und nicht weiter erklärbare Komponente unregelmäßige Einflüsse oder Störungen enthält.

Mit ihren einzelnen Komponenten können Zeitreihen z.B. als additives oder multiplikatives Modell zusammengesetzt werden:

$$y_t = m_t + k_t + s_t + u_t \quad bzw. \quad y_t = m_t \cdot k_t \cdot s_t \cdot u_t$$

Bei der Aufstellung eines Komponenten-Modells muss beachtet werden, dass nicht in jeder Zeitreihe alle aufgeführten Komponenten vorhanden sein müssen und deshalb die Formeln leicht abweichen können. Beispielsweise werden insbesondere dann, wenn der Zeitraum der betrachteten Daten nicht über einen Zyklus hinausgeht, Trend und Zyklus in einer *glatten Komponente* g_t zusammengefasst [37].

3.4.1.2 Stochastische Prozesse

Eine chronologische Folge von Zufallsvariablen $\{Y\} = Y_1, Y_2, Y_3, \ldots, Y_t, \ldots$ wird als stochastischer Prozess in diskreter Zeit bezeichnet. Es handelt sich dabei also um einen Vorgang mit Zufallscharakter. Bei der Zeitreihenanalyse wird angenommen, dass es sich bei der beobachteten Zeitreihe $y_1, y_2, \ldots, y_n$ um eine mögliche (zufällige) Realisierung eines solchen Prozesses handelt. Da dies bedeutet, dass jeder einzelne Wert y_t durch eine eigene Zufallsvariable Y_t generiert wurde, ist es meist schwierig von den Werten ausgehend Rückschlüsse auf den stochastischen Prozess zu ziehen. Trotzdem muss versucht werden, aus den Informationen, die aus der beobachteten Zeitreihe gewonnen werden, das Modell eines stochastischen Prozesses derart zu schätzen, dass es sich bei der Zeitreihe um eine endliche Realisierung dieses Prozesses handeln könnte. Um die Bestimmung des stochastischen Prozesses zu vereinfachen, wird *a priori* eine Klasse von möglichen Prozessen vorgegeben. Für die Beschreibung des konkreten Prozesses $\{Y\}$ ist es in der Regel ausreichend, die ersten und zweiten Momente seiner Zufallsvariablen anzugeben [125]:

1. Mittelwertfunktion $\mu(t) \ := E(Y_t)$
2. Varianzfunktion $\sigma^2(t) \ := Var(Y_t)$
3. Autokovarianzfunktion $\gamma_j(t) \ := \ Cov(Y_t, Y_{t-j})$
4. Autokorrelationsfunktion $\rho_j(t) \ := \gamma_j(t) : (\sigma(t) \cdot \sigma(t-j))$

Bei j handelt es sich in den Autokovarianz- und Autokorrelationsfunktionen um den Abstand der jeweils betrachteten Zufallsvariablen Y_t und Y_{t-j}, der auch *Lag* (deutsch: Verzögerung) oder *Zeitlag* genannt wird.

Stationäre Prozesse

Oftmals wird eine gewisse zeitliche Stabilität des stochastischen Prozesses gefordert. Dies wird beispielsweise dadurch erreicht, dass jede endliche Folge von Zufallsvariablen $Y_1, \ldots, Y_m$ eine identische Wahrscheinlichkeitsverteilung besitzt, wie die um eine beliebige Anzahl von k Zeitpunkten verschobene Folge $Y_{1+k}, \ldots, Y_{m+k}$. Daraus folgt, dass die durch den Prozess gebildeten Zeitreihen einen beliebigen Startzeitpunkt haben können, da die Verteilungen vom

Zeitindex unabhängig sind. In einem solchen Fall spricht man von einem streng stationären Prozess [37].
In der Praxis ist eine solche Stationarität üblicherweise schwer nachweisbar. In der Regel ist allerdings auch schon eine schwache Stationarität ausreichend. Hierfür wird lediglich gefordert, dass die Zufallsvariablen des Prozesses in ihren ersten beiden Momenten übereinstimmen. Das heißt, es soll für alle t und j gelten [66]:

- $\mu(t) = \mu$
- $\sigma^2(t) = \sigma^2$
- $\gamma_j(t) = \gamma_j$

Weißes Rauschen: Ein Spezialfall der stochastischen Prozesse ist der so genannte *White-Noise-Prozess* bzw. das Weise Rauschen. Dieser Prozess ist deshalb von Interesse, da er anderen Prozessen wie beispielsweise den in Abschnitt 3.4.1.2 beschriebenen Moving-Average-Prozessen als Grundlage dient. Es handelt sich dabei um einen stationären Prozess ϵ, dessen Varianzfunktion $\sigma^2(t)$ zu allen Zeitpunkten t einen konstanten Wert annimmt und dessen Mittelwerte $\mu(t)$ und Autokovarianzen $\gamma_j(t)$ für alle t und j ($j \neq 0$) konstant 0 sind. Es ergibt sich aus der Stationarität, dass die Zufallsvariablen des Weißen Rauschens unabhängig und identisch verteilt sind (Vgl. [37, 125]).

Die Differenzenmethode

Zur Elimination einer Trend- oder Saisonkomponente eignet sich die Differenzenmethode. Hierbei handelt es sich um einen so genannten *linearen Filter*, der eine Zeitreihe durch Bildung von Differenzen in eine trend-freie Zeitreihe transformiert. Die gängigste Variante, die auch für die Eliminierung eines linearen Trends ausreicht, ist der Differenzenfilter 1. Ordnung. Die Werte y_t^* der neuen Zeitreihe werden durch die einfache Differenz von jeweils benachbarten Werten der ursprünglichen Zeitreihe gebildet (Vgl. [37, 98]):

$$y_t^* = \Delta y_t = y_t - y_{t-1} \ , \ f\ddot{u}r\ t = 2, \ldots, n$$

Für den Fall, dass ein polynomialer Trend vorliegt, ist es notwendig einen Differenzenfilter höherer Ordnung (entsprechend dem Grad des angenommenen Polynoms) anzuwenden.

Lineare Modelle

Das für die Zeitreihenanalyse gängigste lineare Modell ist das ARIMA-Modell (*Autoregressive Integral Moving Average*), das von Box und Jenkins in den 70er Jahren zu einem brauchbaren Modell für Prognosen weiterentwickelt wurde. Dieses wird durch eine Kombination eines Moving-Average-Prozesses und eines autoregressiven Prozesses einer durch Differenzenbildung stationärisierten Zeitreihe gebildet (Vgl. [145]).

Moving-Average-Prozesse: Ein stochastischer Prozess $\{Y\}$ heißt Moving-Average-Prozess (deutsch: gleitendes Mittel) der Ordnung q bzw. MA(q)-Prozess, falls gilt [127, 145]:

$$Y_t = \epsilon_t + \sum_{j=1}^{q} \alpha_j \epsilon_{t-j}$$

Bei gewichteten $\{\epsilon\}$ handelt es sich um Werte aus Weißem Rauschen und die Koeffizienten $\alpha_1, \ldots, \alpha_q$ sind reelle Faktoren. Demzufolge ist ein MA(q)-Prozess ein Prozess, der aus dem gleitenden Durchschnitt eines Weißen Rauschens mit der Fenstergröße q gebildet wird.

Autoregressive Prozesse: Ein stochastischer Prozess $\{Y\}$ heißt autoregressiver Prozess der Ordnung p bzw. AR(p)-Prozess, falls gilt [127, 145]:

$$Y_t = \epsilon_t + \sum_{j=1}^{p} \beta_j Y_{t-j}$$

$\{\epsilon\}$ repräsentiert wiederum Weißes Rauschen und die Koeffizienten $\beta_1, \ldots, \beta_q$ sind reelle Faktoren. Damit wird jedes Y_t des Prozesses als gewichtetes Mittel seiner p Vorgänger mit einem zufälligen Rest ϵ_t gebildet.

Autoregressive integrierte Moving-Average-Prozesse: Als Grundlage von autoregressiven integrierten Moving-Average-Prozessen (ARIMA-Prozessen) dienen ARMA-Prozesse, die eine Kombination von Moving-Average und autoregressiven Prozessen darstellen. Ein ARMA(p,q)-Prozess wird demnach wie folgt gebildet:

$$Y_t = \sum_{j=1}^{p} \beta_j Y_{t-j} + \epsilon_t + \sum_{j=1}^{q} \alpha_j \epsilon_{t-j}$$

Wobei $\{\epsilon\}$ Weißes Rauschen darstellt und die Koeffizienten $\beta_1, \ldots, \beta_p, \alpha_1, \ldots, \alpha_q$ reelle Faktoren sind. Voraussetzung für die Anwendung eines ARMA-Prozesses ist jedoch, dass es sich bei der beobachteten Zeitreihe um einen schwach stationären Prozess handelt. Da reale Zeitreihen oftmals Instationaritäten wie einen Trend oder saisonale Abhängigkeiten aufweisen, müssen diese durch Differenzenfilter in stationäre Prozesse transformiert werden. Dabei muss die Ordnung des Differenzenfilters hinreichend groß sein, damit die dadurch entstehende Zeitreihe stationär ist. Die Kombination einer Differenzenbildung mit einem ARMA(p,q)-Modell wird als ARIMA(p,d,q)-Modell bezeichnet, wobei mit dem Parameter d die Ordnung des Differenzenfilters angegeben wird [37] (Vgl. [98, 127, 145]).

3.4.2 Zeitreihenprognosen mit statistischen Modellen

Unter einer Prognose versteht man die *Vorhersage zukünftiger Ereignisse auf Grund von Vergangenheitsinformation* [127]. Die Anzahl der Werte, die zwischen dem letzten beobachteten und dem zu prognostizierenden Wert der jeweiligen Zeitreihe liegen, wird als Prognosehorizont bezeichnet. Liegt dieser bei eins, spricht man auch von einer Ein-Schritt-Prognose.

Eine Erweiterung auf Mehr-Schritt-Prognosen lässt sich einerseits durch eine entsprechende Anpassung der Modelle oder andererseits durch die iterierte Durchführung von Ein-Schritt-Prognosen erreichen.

Zur Durchführung einer Prognose eignen sich je nach Anwendung die Modelle, die im vorausgehenden Abschnitt beschrieben wurden. Um die Unterschiede deutlich zu machen werden im Folgenden exemplarisch drei Prognose-Verfahren dargestellt (Vgl. auch [37, 145]).

Komponentenmodell

Zur Prognose von zukünftigen Werten einer Zeitreihe mit Hilfe des Komponenten-Modells müssen zunächst die einzelnen additiv oder multiplikativ verknüpften Komponenten bestimmt werden. Dies sollte dergestalt geschehen, dass in der Größe u_t lediglich vernachlässigbare Störgrößen übrig bleiben. Anhand der gefundenen Funktionen für die glatte Komponente und Saisonschwankungen kann nun der Wert für den Zeitpunkt $t + 1$ berechnet werden, wobei $u_{t+1} = 0$ gesetzt wird. Diese Methode kann beispielsweise angewendet werden, wenn von einem grundsätzlich regelmäßigen Verlauf der Zeitreihe ausgegangen wird und die unerklärte Komponente z.B. durch Messfehler entstanden ist.

ARMA-Prognosen

Um zukünftige Werte einer Zeitreihe mit einem ARMA- bzw. ARIMA-Modell zu prognostizieren, müssen dessen Parameter zunächst z.B. anhand der Box-Jenkins-Methode bestimmt werden. Wurde ein adäquates Modell gefunden, kann unter der Hinzunahme eines Weißen Rauschens über die entsprechende Formel der Wert für y_{t+1} einfach berechnet werden. Es gilt zu berücksichtigen, dass die Prognose unter Verwendung eines ARIMA-Modells auf der durch Differenzenbildung stationarisierten Zeitreihe durchgeführt wird. Zur Gewinnung des gesuchten Prognosewertes muss der Einfluss des Differenzfilters beachtet werden.

3.5 Tracking

Als *Tracking* (Nachführung) bezeichnet man alle Bearbeitungsschritte, die der Verfolgung von (bewegten) Objekten dienen. Ziel dieser Verfolgung ist zum einen die Extraktion von Informationen über den Verlauf der Bewegung und die Lage eines Objektes und zum anderen die Verminderung von negativen Einflüssen, deren Ursache meist zufällige Messfehler (Messrauschen) sind. Die extrahierten Informationen können die Geschwindigkeit der Bewegung, die Beschleunigung sowie Informationen bezüglich der Lage zu einem bestimmten, oft in der Zukunft liegenden, Zeitpunkt sein. Die in diesem Zusammenhang verwendeten Begriffe *Objekt*, *Lage*, *Bewegung* und *Beschleunigung* müssen nicht zwingend geometrischer Natur bzw. Herkunft sein. Diese stehen oft auch als Synonym für andere Messwerte(-funktionen) und deren Ableitungen (z.B. finanzieller Wert, Zustand usw.) [144]. Insbesondere für die vorliegende Arbeit ist Tracking als die Nachführung von Zuständen zu verstehen.

Unabhängig von den gewählten Prognoseverfahren (siehe Abschnitt 3.4) lässt sich Tracking im Allgemeinen in folgende Verarbeitungsschritte unterteilen [144]:

- **Prädiktion**: In diesem Verarbeitungsschritt erfolgt die (rechnerische) Bestimmung der Lage- und Bewegungsinformationen anhand bekannter Geschichte und physikalischer oder mathematischer Gesetzmäßigkeiten.
- **Assoziation** (auch Gating): Insbesondere in Beobachtungsräumen, in denen sich in der Regel mehrere Objekte (*Multi-Target-Tracking*) befinden und diese nicht eindeutig über verschiedene Messzyklen identifizierbar sind, übernimmt diese Komponente die Zuordnung eines in früheren Messzyklen beobachteten Objektes zu einem aktuell gemessenen Objekt. Fehler in diesem Bearbeitungsschritt (*Missassignments*) wirken sich besonders schwer auf die Ergebnisse aus.
- **Innovation**: Die Bestimmung der aktuellen Lage und anderer bewegungsrelevanter Informationen erfolgt einerseits durch die Prädiktion und andererseits durch aktuelle Messungen (bzw. Berechnungen aus aktuellen Messungen). Der Innovationsschritt führt beide Ergebnisse gewichtet zusammen. Die Gewichtung kann sowohl dynamisch als auch statisch erfolgen. Eine Verschiebung der Anteile hin zur Prädiktion glättet die Ergebnisse stärker und eine größere Gewichtung der Messung führt zu Ergebnissen, die sich schneller auf Veränderungen der Messwerte einstellen.

Beispiele für üblicherweise eingesetzte Tracking-Algorithmen sind [144]:

- $\alpha/\beta/\gamma$ -Filter
- Kalman-Filter [149]
- ARMA- bzw. ARIMA-Filter [145] (siehe auch Abschnitt 3.4)
- Sequentielle Monte-Carlo-Methode [143]

4.

Adaptive Rechensysteme

Dieses Kapitel gibt einen Überblick über den aktuellen Stand der Forschung im Bereich der adaptiven Rechensysteme. In diesem Zusammenhang werden adaptive Rechensysteme definiert und klassifiziert. Die für die vorliegende Arbeit grundlegenden Projekte erfahren eine detaillierte Darstellung. Des Weiteren bildet eine Kurzbeschreibung weiterer Projekte den Abschluss dieses Kapitels.

A. Koch et al. definieren adaptive Rechensysteme folgendermaßen [96]:

Definition 4.1 *Adaptive Rechensysteme*

> Adaptive Rechensysteme haben die Fähigkeit, sich neben der konventionellen Programmierung durch Software auch in Hardware-Aspekten an die Erfordernisse der aktuellen Anwendung anpassen zu lassen. □

4.1 Klassifikation adaptiver Rechensysteme

Üblicherweise finden sich in der Literatur zwei spezielle Kategorien adaptiver Rechensysteme:

- **Autonome Rechensysteme** mit dem Anspruch, Anpassungen an sich zur Laufzeit ändernde (Umgebungs-)Anforderungen selbständig durchzuführen [54].
- **Organische Rechensysteme** deren Anpassungsmechanismen in Anlehnung an Lebewesen bzw. Organismen entworfen sind [128].

Eine Unterscheidung in autonome und organische Rechensysteme anhand festgelegter Systemeigenschaften ist bisher in der Literatur nicht erkennbar (Vgl. auch [51, 56]). Z.B. legt die Firma IBM mit dem so genannten *Autonomic Computing Manifest* eine Beschreibung eines autonomen Rechensystems vor, dessen Eigenschaften in Anlehnung an den menschlichen Organismus entwickelt werden sollen. Demzufolge handelt es sich hierbei eigentlich um einen Ansatz eines organischen Systems. Des Weiteren ist fragwürdig inwiefern Anpassungsfähigkeit und Autonomie bereits grundlegende natürliche bzw. organische Eigenschaften sind. Für die hier eingeführte Klassifizierung von Rechensystemen werden beide Eigenschaften als Voraussetzung eines organischen Rechensystems gesehen. Die Definition der Eigenschaft Autonomie basiert auf Systemeigenschaften aus dem Maschinenbau.

Die von Frank et al. [51] festgelegte Definition der *Selbstoptimierung* für Systeme des Maschinenbaus wird für die Charakterisierung autonomer adaptiver Rechensysteme zugrunde gelegt und *selbstoptimierend* auf *selbstständig* oder *autonom* verallgemeinert[1]. Dementsprechend liegt ein autonomes Rechensystem vor, falls folgende drei Aktionen wiederkehrend ausgeführt werden:

1. Analyse der Ist-Situation
2. Bestimmung der Systemziele
3. Anpassung des Systemverhaltens

Wird bei der Anpassung des Systemverhaltens zusätzlich die Hardware des Systems miteinbezogen, handelt es sich um ein autonomes adaptives Rechensystem. Somit lässt sich ein autonomes adaptives Rechensystem unter Einbeziehung der Definition adaptiver Systeme wie folgt definieren:

Definition 4.2 *Autonome adaptive Rechensysteme*

Autonome adaptive Rechensysteme verfügen, neben der dynamischen Anpassung von Hardware-Aspekten zudem über die Fähigkeit, die Ziele der Anpassung selbständig festzulegen und die Anpassungen selbst zu vollziehen. □

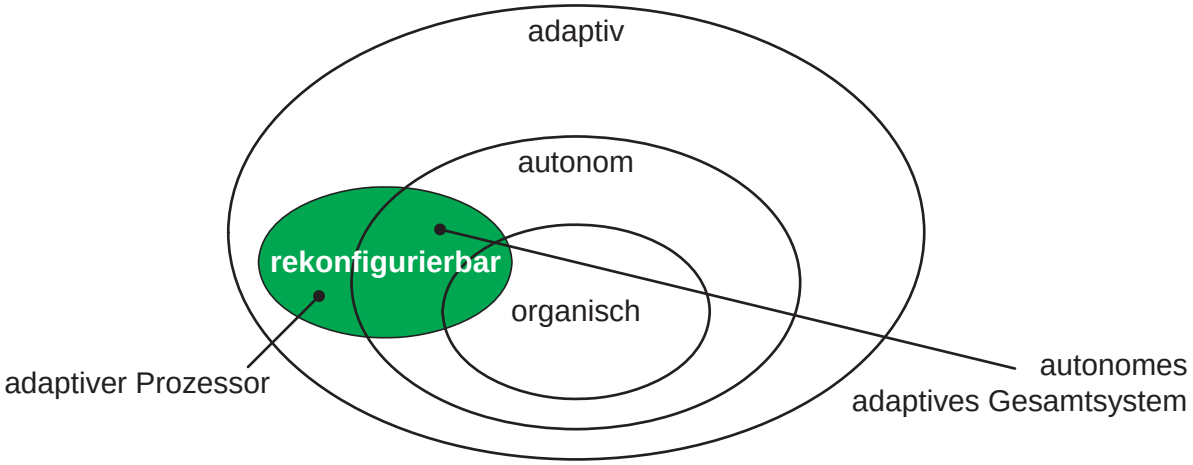

Abbildung 4.1: Einteilung von Rechensystemen

In diesem Zusammenhang stellt die Bestimmung der Systemziele das wichtigste Kriterium für die Selbstoptimierung dar (Vgl. [51]). Allgemein bedeutet eine eigenständige Bestimmung von Systemzielen die Selbständigkeit eines Systems unter Berücksichtigung von

[1] Selbstoptimierung impliziert, dass Anpassungen zielgerichtet mit dem Resultat einer Verbesserung vorgenommen werden. Entsprechend ist die Einschätzung einer Adaption als Optimierung von der Sichtweise des Betrachters abhängig. Ist dem Betrachter das Ziel nicht bekannt, kann der Eindruck entstehen, dass Anpassungen sinnlos oder unzweckmäßig sind. Legt das System eigenständig Anpassungsziele fest und versucht diese zu erreichen agiert es selbstständig. Sind die Ziele dem Betrachter bekannt können diese stets als Selbstoptimierung aufgefasst werden.

Umgebungsbedingungen. Dementgegen sind die Analyse des Ist-Zustandes und der Anpassung des Systemverhaltens per Definition der Regelung (Definition 3.1) in jedem System, das über Regelungsmechanismen verfügt, vorhanden. Für die Selbständigkeit ist wesentlich, dass Ziele durch das System situationsbedingt festgelegt oder angepasst werden.

Abbildung 4.1 illustriert die Klassifikation adaptiver Rechensysteme. Die autonomen Rechensysteme[2] stellen eine echte Teilmenge der adaptiven Systeme dar. Organische Systeme sind entsprechend obigen Festlegungen eine echte Teilmenge der autonomen Systeme. Rechensysteme mit rekonfigurierbarer Logik bilden ebenfalls eine Teilmenge der adaptiven Rechensysteme. Eine Einordnung muss anhand der Fähigkeiten bzw. Eigenschaften des jeweiligen Rechensystems vorgenommen werden. Dementsprechend ist das in dieser Arbeit behandelte autonome adaptive Gesamtsystem den autonomen Rechensystemen auf Basis rekonfigurierbarer Logik zuzuordnen. Betrachtet man den in Kapitel 6 vorgestellten adaptiven Prozessor als System, ist dieser als adaptives System zu klassifizieren, da Systemziele nicht eigenständig festgelegt werden.

4.2 Einteilung adaptiver Rechensysteme mit rekonfigurierbarer Logik

4.2.1 Klassifikation anhand der Rekonfigurationseigenschaften

Neema et al. [116] klassifizieren adaptive Systeme mit rekonfigurierbarer Logik in Abhängigkeit der Rekonfigurationseigenschaften folgendermaßen:

- **Algorithmische Rekonfiguration** (*Algorithmic Reconfiguration*) liegt dann vor, wenn die Schnittstelle und Funktionalität der rekonfigurierbaren Systemteile erhalten bleibt, jedoch der interne Algorithmus verändert wird.
- Von **Struktureller Rekonfiguration** (*Architectural Reconfiguration*) spricht man, wenn die rekonfigurierbare Logik zur Veränderung von Schnittstellen oder zur Anpassung der Infrastruktur (des Gesamtsystems oder von Teilsystemen) genutzt wird.
- **Funktionale Rekonfiguration** (*Functional Reconfiguration*) bezeichnet die Fähigkeit das System oder Teile davon mit Hilfe rekonfigurierbarer Logik mit neuen, zusätzlichen Fähigkeiten auszustatten.

Für das in dieser Arbeit behandelte adaptive Gesamtsystem (siehe Kapitel 5) und insbesondere für den adaptiven Prozessor (siehe Kapitel 6) sind sowohl die algorithmische als auch die funktionale Rekonfiguration von Bedeutung.

4.2.2 Klassifikation adaptiver Prozessoren

Des Weiteren liefert Esparza [44] eine Klassifikation von FPGA-basierten adaptiven Prozessoren auf Grund der Integration bzw. Kopplung der rekonfigurierbaren Teile mit dem übergeordneten System:

[2] Autonomen adaptiven Rechensysteme

- **Eigenständiger Prozessor** (*Stand-Alone Processor*): Der rekonfigurierbare Teil respektive FPGA agiert als eigenständiger Rechner, der mit der zentralen Recheneinheit ausschließlich über Ein-/Ausgabe-Schnittstellen kommuniziert. Durch die verhältnismäßig langsame Kommunikationsart eignet sich diese Klasse nur für Systeme, deren innere Kommunikation stark beschränkt ist.

- **Zusätzlicher Prozessor** (*Attached Processor*): Hierbei wird der FPGA als zusätzlicher Prozessor, vergleichbar mit Mehrprozessorsystemen, betrachtet. Die Kommunikation zwischen FPGA und weiterem Prozessor erfolgt über einen separaten Bus. Entsprechend dem Einsatz von Mehrprozessorsystemen gelten die gleichen Restriktionen in Bezug auf Speicherzugriffe usw.

- **Koprozessor**: In diesem Fall ist der FPGA als Koprozessor an einen Prozessor angebunden. Demzufolge werden Koprozessorbefehle zu dessen Ansteuerung genutzt.

- **Funktionseinheit**: Der rekonfigurierbarer Anteil ist in die Pipeline eines Prozessors integriert. Dies schafft neben dem parallelen Betrieb des FPGAs zu den vorhandenen Funktionseinheiten zudem die Möglichkeit, dass der FPGA auf die Registerdatei des Prozessors zugreift. Darüber hinaus ist es möglich den Maschinenbefehlssatz des Prozessors dynamisch zu erweitern.

Der in dieser Arbeit behandelte adaptive Prozessor bedient sich der Integration rekonfigurierbarer Logik auf der Grundlage von rekonfigurierbaren Funktionseinheiten (RFE bzw. englisch: RFU). Zusätzlich finden sich bei der Ansteuerung der RFEs auch Aspekte, die der Koprozessor-Klasse zuzuordnen sind (siehe Kapitel 6).

4.3 Dynamische Anpassung von Systemen

Im Bereich der eingebetteten Systeme und vor allem bei mobilen Endgeräten, werden seit Jahren unterschiedlichste Anstrengungen unternommen, um das Verhältnis zwischen Energie und Rechenleistung zur Laufzeit zu verbessern. Die ursprünglich eingesetzten, hauptsächlich auf Erfahrungswerten basierenden, Lösungen werden zwischenzeitlich weitgehend durch allgemeine, statistische Methoden der Regelungstheorie verdrängt [118].

4.3.1 Adaptivität von Systemen ohne rekonfigurierbare Logik

Die meisten Ansätze basieren auf der Nachführung relevanter Systemgrößen, wie z.B. Leistungsaufnahme, Rechenleistung etc. Hierbei kommen häufig verschiedene Filter (Kalman, ARMA und ARMA-Derivate usw.) zum Einsatz. Zudem existieren Ansätze auf Basis von Fuzzy-Controllern oder Neuronalen Netzen (Vgl. Kapitel 3). Die Regelung erfolgt stets über die Anpassung von Hardware-Parametern.

Unabhängig von den verwendeten Regelungsverfahren ist das Ziel der Regelungen stets die geforderten, meist anwendungsspezifischen, Kriterien mit minimalem Energiebedarf auszuführen. Die Einhaltung von Anwendungskriterien in einem bestimmten Maß, wird in der Regel als *Quality of Service* (QoS) bezeichnet. Welche Kriterien erfüllt sein müssen hängt massiv vom Einsatzgebiet bzw. der ausgeführten Anwendung ab. Im Telekommunikations-

und Netzwerkbereich sind z.B. Durchsatz oder Transportzeiten ausschlaggebend [118, 131]. Bei Multimediaanwendungen sind meist minimale Bild- bzw. Sample-Raten die maßgeblichen Qualitätskriterien [75, 106, 105].

In diesem Zusammenhang bieten ARM in Kooperation mit National Semiconductors sowie Intel kommerzielle Lösungen zur Regelung der Leistungsaufnahme und Rechenleistung an.

Intelligent Energy Management von ARM

Die Firmen ARM und National Semiconductors bieten unter dem Namen *Intelligent Energy Manager* (IEM) und *PowerWise* eine Kombination aus *Dynamic Power Management* (DPM) und *Dynamic Voltage Scaling* (DVS) an, die auf der Interaktion zwischen Prozessor und Spannungsversorgung basiert [16, 19]. Diesbezüglich reduziert ARMs Intelligent Energy Manager die Leistungsaufnahme des Prozessorkerns um bis zu 75%. Die PowerWise-Technologie bringt ihrerseits ca. 45% Energieersparnis in Bezug auf den Prozessor. Der gemeinsame Einsatz beider Techniken wird mit einer ca. 30% Energieersparnis für ein Gesamtsystem angegeben [14].

Intels Power Management

Obwohl über Jahre hinweg dem Energiebedarf von Prozessoren keine Aufmerksamkeit geschenkt wurde, hat sich in den letzten Jahren ein deutlicher Wandel vollzogen. Untersuchungen auf Basis der Metrik Energie pro Instruktion (EPI) zeigen hierbei eine deutliche Verbesserung bei den aktuellen Intel Prozessoren [63]. Zum Einsatz kommen die in Grundlagenabschnitt 2.4 angesprochenen Techniken zum Entwurf von Niedrigenergie- und Energiespar-Systemen.

In diesem Zusammenhang verfolgt Intel Optimierungsmöglichkeiten in den beiden unteren Systemebenen, der Technologie- und Logik-Ebene. Zum Beispiel wurde der Fertigungsprozess unter Berücksichtigung von Energiegesichtspunkten verbessert [85]. Darüber hinaus existieren Realisierungen des Dynamic Voltage Scaling (DVS) und Dynamic Power Managements (DPM): Ersteres wird bei Intel als *Speed-Stepping* (Vgl. [83, 79]) bezeichnet und erlaubt die dynamische Einstellung verschiedener Versorgungsspannungen und Frequenzen. Unter dem Namen *Intelligent Power Capability* fasst Intel diverse Mechanismen des Dynamic Power Managements zusammen: Hierbei werden verschiedene Prozessor-Modi (*Sleep-Mode*, *Deep-Sleep-Mode*, etc.) und das gezielte An- und Abschalten von Prozessorteilen angeboten [82, 86].

4.3.2 Adaptivität von FPGA-basierten Systemen

Neben einer Anpassung durch Hardware-Parameter eröffnen adaptive Systeme mit rekonfigurierbarer Logik zusätzlich die Möglichkeit die Hardware, funktional, strukturell und algorithmisch, anwendungsspezifisch zu ändern bzw. zu adaptieren. In diesem Zusammenhang werden FPGAs bzw. verfügbare FPGA-Ressourcen in der Regel als anwendungsspezifisch konfigurierbare Digitale Signalprozessoren (DSP) verstanden. Erwartungsgemäß führt diese Vorgehensweise zu sehr guten Speedups, die üblicherweise etwas geringer ausfallen, als dies für entsprechende ASICs oder DSPs der Fall wäre (Vgl. [25]). Im Gegensatz dazu erlaubt der Einsatz rekonfigurierbarer Ressourcen eine weitaus flexiblere Anpassung. Die Anpassung der

rekonfigurierbaren Hardware basiert üblicherweise auf der Grundlage statischer Information, z.B. der Anwendungsübersetzung, und erfolgt zu Beginn der Anwendungsausführung. Feingranulare Methoden zur dynamischen Hardware-Adaption während der Anwendungsausführung existieren nicht. Diesbezügliche Ansätze und Lösungen stehen im Mittelpunkt dieser Arbeit.

4.4 Bestehende FPGA-basierte adaptive Systeme

Im Laufe dieses Abschnitts werden Projekte FPGA-basierter adaptiver Systeme vorgestellt. Den beiden Projekten OneChip und Garp kommt in Bezug auf die vorliegende Arbeit eine besondere Bedeutung zu, da Ergebnisse beider in das autonome adaptive Gesamtsystem einfließen. Dementsprechend widmen sich die Abschnitte 4.4.1 und 4.4.2 ausführlich diesen Projekten. Weitere Projekte sind in Form einer Kurzzusammenfassung aufgeführt.

4.4.1 OneChip

OneChip [154, 87] ist ein rekonfigurierbarer Prozessor, der eine rekonfigurierbare Funktionseinheit (RFE) in die Pipeline eines RISC-Prozessors integriert. Der Prozessor verfügt über eine superskalare Architektur und unterstützt eine *out-of-order* Befehlsausführung sowie die dynamische Rekonfiguration der RFE. Darüber hinaus erlaubt die Konfigurationsverwaltung ein vorzeitiges Laden von Konfigurationsdaten in verfügbare Kontexte und arbeitet nach einem *Least Recently Used* (LRU) Algorithmus. Insgesamt werden für ausgesuchte Anwendungen im Vergleich zu Arbeitsplatzrechnern, wie z.B. SparcStation, UltraSparc usw., Beschleunigungsfaktoren von 30 und mehr angegeben.

Die aktuelle Implementierung des OneChip-Prozessors basiert auf der DLX RISC-Prozessorbeschreibung von Hennessy und Patterson [72]. Darauf aufbauend wurde die OneChip-Architektur um superskalare und dynamische Befehlsausführung, entsprechende RFE Befehlsspezifikationen, die Konfigurationsverwaltung der RFE und eine Speicherschnittstelle für den direkten Datenaustausch zwischen RFE und Hauptspeicher erweitert.

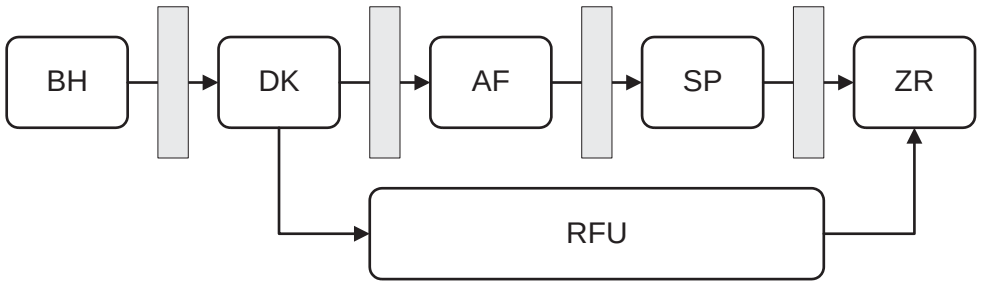

Abbildung 4.2: Schema der OneChip-Prozessorpipeline

Die Pipeline des OneChip-Prozessors besteht aus fünf Stufen: Eine Befehlsholphase (BH), Befehlsdekodierungsphase (DK), Ausführungsphase (AF), Speicherzugriffsphase (SP) und Rückschreibphase (ZR). Die RFE (bzw. RFU) ist parallel zur Ausführungsphase und Speicherzugriffsphase in die Pipeline integriert (Abbildung 4.2).

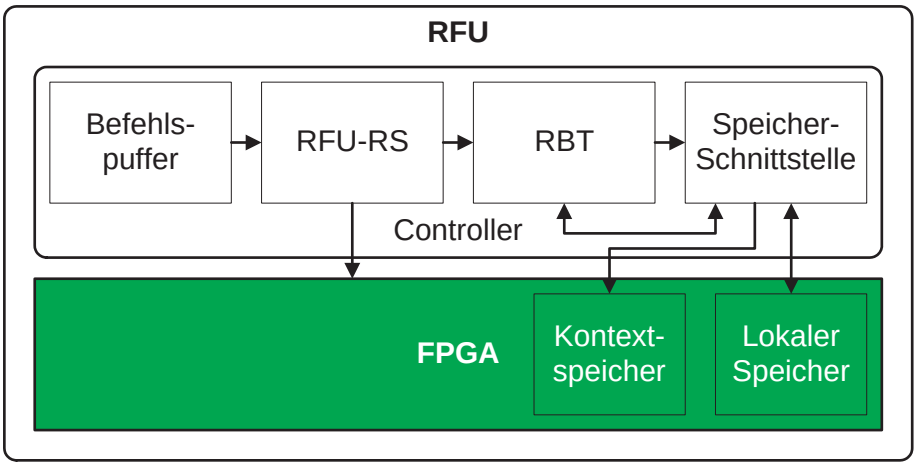

Abbildung 4.3: Aufbau der rekonfigurierbaren Funktionseinheit

Die RFE (bzw. RFU) besteht aus den in Abbildung 4.3 dargestellten zwei logischen Einheiten: Dem *Controller* und einem FPGA[3].

Der Controller enthält einen eigenen Befehlspuffer für die RFE-Befehle, eine Reservierungstabelle (*RSU-RS*), eine *Reservation Bits Table* (*RBT*) und eine eigenständige Speicherschnittstelle. Die RBT wird zur Verwaltung unterschiedlicher Kontexte im FPGA genutzt. Diesbezüglich stehen bis zu sechs Kontexte zur Verfügung. Hierbei enthält die RBT Informationen über die Speicheradresse der zugehörigen Konfigurationsdaten sowie über die Verfügbarkeit einzelner Konfigurationen. Wird eine Konfiguration angefordert die sich nicht im FPGA befindet, regt die RBT das Nachladen an. Sowohl diese Konfigurationsdaten als auch andere Daten werden über eine gemeinsame Speicherschnittstelle jeweils aus dem Hauptspeicher bezogen [44].

4.4.2 Garp und seine Nachfolgeprojekte

Der Garp [30] rekonfigurierbare, FPGA-basierte Koprozessor ist speziell für die Beschleunigung von Anwendungsschleifen konzipiert. Hierbei werden innere Schleifen aus C-Programmen extrahiert und auf ihre Umsetzung in Hardware getestet. Zeigt sich der Anwendungscode als geeignet, erfolgt eine automatische Hardware-Erzeugung. Die resultierenden Hardware-Schaltungen, die als *Module* bezeichnet werden, eignen sich zur Konfiguration des Garp Koprozessors. Dementsprechend erfolgt die Programmausführung zum einen Teil in Software und zum anderen in Hardware.

Garp wurde so entworfen, dass es nicht auf zusätzliche Übersetzerdirektiven oder Hinweise vom Anwendungsprogrammierer angewiesen ist. Bei Geschwindigkeitsvergleichen mit einer UltraSparc zeigt der Garp-Prozessor je nach Anwendung Beschleunigungsfaktoren von ca. 2 bis 24. Der Garp-Compiler nutzt das Front-End des SUIF C-Compilers für das Parsen und die Durchführung von Standardoptimierungen [30, 68].

[3]Der im OneChip-Prozessor eingesetzte FPGA stellt mehrere Kontexte zur Verfügung und ist demzufolge eigentlich ein Multi-Kontext-FPGA bzw. DPGA

Seit 1997 kooperiert die Abteilung Entwurf integrierter Schaltungen (E.I.S.) der TU-Braunschweig im Rahmen des Projekts *Nimble Compiler for Agile Hardware* unter anderem mit der Firma Synopsys (Advanced Technology Group) und der Universität Berkeley (BRASS). Als Ziel steht die Entwicklung eines durchgehenden Entwurfsflusses, der ohne Einschränkungen C in kombinierte Hardware/Software-Programme übersetzen kann, die dann auf einem geeigneten adaptiven Rechensystem effizient ausgeführt werden können [97].

Das Nimble-Nachfolgesystem *COMRADE* wurde bis 2004 deutlich weiterentwickelt und kann nun die ersten Beispielanwendungen korrekt in simulierte Hardware übersetzen. Darüber hinaus befasst sich ein laufendes DFG-Projekt (mit Schwerpunktprogramm 1148 assoziiert), auch mit der automatischen Integration von bestehenden Hardware-Blöcken mit compiler-generierten Datenpfaden.

Im einzelnen wird dabei zur Realisierung der Datenpfade ein Baustein Xilinx Virtex XCV1000 verwendet. Ein RISC microSPARC-IIep dient als CPU. Als Speicher stehen dem rekonfigurierbaren Prozessor und der CPU gemeinsam 64MB DRAM zur Verfügung. Die Kommunikation zwischen rekonfigurierbarem Prozessor und CPU erfolgt über einen PCI-Bus. Um die Kommunikationslatenzen zu verkleinern, verwendet auch der rekonfigurierbare Prozessor einen eigenen Cache. Die komplette Testplattform steckt als PCI-Karte in einem Wirtsrechner, läuft aber von diesem unabhängig unter einem eigenen Betriebssystem. Nur für Ein- und Ausgabeoperationen wird auf die Peripherie (Dateien, Netzwerk) des Wirtsrechners zugegriffen [96].

Ziel ist es den rekonfigurierbaren Prozessor zusammen mit einer CPU auf einen Chip zu integrieren. Obwohl die bisherigen Forschungsergebnisse auf einer PCI-Bus-gekoppelten Testplattform entstanden sind, zeigt sich, dass sich im Vergleich zu einem Pentium M Speedups von 2 bis 3 mit vergleichbarer Leistungsaufnahme erreichen lassen (Vgl. [97, 96]).

4.4.3 Überblick über weitere Universitätsprojekte

Dieser und der folgende Abschnitt liefern einen Überblick über weitere Projekte im Bereich der rekonfigurierbaren adaptiven Systeme. Eine umfangreiche Sammlung älterer FPGA-basierter adaptiver Systeme steht in [130] zur Verfügung.

ACS

Das so genannte Adaptive Computing System (ACS) [129] verfolgt einen Ansatz mit automatischer Generierung von Hardware-Schaltungen, wie diese auch im Garp-Projekt zu finden ist. Der Schwerpunkt von ACS liegt jedoch im Einsatzgebiet digitaler Signalverarbeitung.

Darüber hinaus bildet das Teilprojekt ROAR (Reliability Obtained By Adaptive Reconfiguration) der Stanford University [124] einen Bestandteil des ACS, dessen Forschungsziel in der Identifikation und Umsetzung von Verfügbarkeitskriterien für Systeme mit Hilfe von rekonfigurierbarer Hardware liegt.

PipeRench

PipeRench [102, 60, 61] wurde an der Carnegie Mellon University entwickelt. Hierbei handelt es sich um einen zusätzlichen rekonfigurierbaren Prozessor, der den PCI-Bus als Schnittstelle zum Prozessor des Systems nutzt. PipeRench besteht aus einem Verbindungsnetzwerk von Verarbeitungselementen (*Processing Element*), die in Form von Pipeline-Stufen organisiert sind. Jedes *Processing Element* ist aus Registern und ALUs aufgebaut. Zur Generierung der zugehörigen Konfigurationsinformation wird eine eigenständige Metasprache eingesetzt.

Chimaera

Chimaera ist eine Entwicklung der Northwestern University. Das rekonfigurierbare Feld ist als rekonfigurierbare Funktionseinheit in einen Prozessor integriert. Als rekonfigurierbare Ressourcen stehen ausschließlich konfigurierbare kombinatorische Logikgatter zur Verfügung. Die Kommunikation mit dem integrierten konfigurierbaren Feld wird mittels Schattenregister abgewickelt [67].

MorphoSys

Die Universität Irvine stellte 1999 das MorphoSys-Projekt [104] vor. Hierbei wird ein Feld rekonfigurierbarer Zellen mit zusätzlichem Kontextspeicher in Form eines rekonfigurierbaren Koprozessors an ein System angebunden. Das rekonfigurierbare Feld arbeitet auf den Daten eines Puffers, der mit Hilfe des DMA-Controllers bedient wird.

Remarc

Von der Universität Stanford ist das Remarc-Projekt. Remarc stellt einen rekonfigurierbaren Koprozessor zur Verfügung, der 64 programmierbare Einheiten bereithält. Jede Einheit ist auf 16 Bit Operationen beschränkt. Darüber hinaus implementiert jede Einheit einen Instruktions- und Daten-Speicher, mindestens eine ALU, sowie ein Instruktions- und mehrere andere Register. Ein integriertes rekonfigurierbares Feld greift auf die Datenregister des Koprozessors zurück. Eine zusätzliche Kontrolleinheit transferiert die Daten zwischen den Koprozessorregistern, den programmierbaren Einheiten und dem Systemprozessor. Adressiert werden vorrangig Multimedia-Anwendungen [109].

Prisc

Prisc [120] wurde an der Harvard Universität entwickelt. Kombinatorische rekonfigurierbare Elemente werden als rekonfigurierbare Funktionseinheit integriert. Eine Funktionseinheit hat jeweils zwei Eingänge und einen Ausgang. Der zugehörige Compiler analysiert den vorher generierten Anwendungscode und identifiziert sequentielle Instruktionen, die sich für die Ausführung in einer rekonfigurierbaren Einheit eignen.

Sonic

Von der Universität London stammt das Sonic-Projekt. Hierbei steht die Untersuchung des Parallelismus in bildverarbeitenden Algorithmen im Vordergrund. Zum Einsatz kamen

mehrere Verarbeitungselemente, die als PIPEs bezeichnet werden und mit einem Bus untereinander Verbunden sind. Die Kommunikation mit dem Prozessor des Systems erfolgt über den PCI-Bus. Die PIPEs können dementsprechend als zusätzlicher rekonfigurierbarer Prozessor verstanden werden [71].

4.4.4 Überblick über kommerzielle Projekte

Chameleon Systems

Die Chameleon CS2000 Familie kombiniert einen eingebetteten 32 Bit Prozessor mit einer rekonfigurierbaren 32 Bit Einheit. Beide sind über einen 128 Bit breiten *split-transaction* Bus miteinander verbunden. Der zugehörige Compiler erzeugt lauffähigen Anwendungscode indem der Objektcode der Anwendung mit vorab generierten Konfigurationsdaten der rekonfigurierbaren Einheit, in Form von Bitstreams, gekoppelt wird [31].

Triscend

Die E5 Produktreihe von Triscend [138] besteht aus konfigurierbaren Bausteinen nach dem *system-on-chip* Prinzip. Jeder Baustein umfasst einen eingebetteten Mikrokontroller, einen SRAM-Block, einen Systembus sowie konfigurierbare Logik auf einem einzelnen Chip. Die konfigurierbare Logik kann mit Hilfe von so genannten *Circuit Generator Modules* aus einer Bibliothek genutzt werden.

National Semiconductor

National kombiniert in der *National Adaptive Processing Architecture* (NAPA) einen konfigurierbaren Prozessor mit einem RISC-Prozessor, Speicher und einer Schnittstelle für ein Verbindungsnetzwerk [123]. Der zugehörige NAPA C Compiler [58] benötigt Benutzerunterstützung zur Anpassung an die tatsächlich vorhandene Zielarchitektur.

Annapolis

Die Annapolis Wildfire Serie ist eine rekonfigurierbare Multiprozessorsteckkarte die über einen VME- oder PCI-Bus an einen Prozessor angeschlossen wird. Eine Steckkarte besteht aus mehreren Xilinx FPGAs. Hierbei wird eine PCMCIA-Ausführung unter der Bezeichnung Wildcard angeboten. Beide Varianten sind mit Standard C und VHDL-Werkzeugen zu programmieren [15].

5.

Metamorphosys: Konzept eines autonomen adaptiven Systems

Metamorphose, die; -, -n, <griech.>: Umgestaltung, Verwandlung;

Hinter dem Kunstwort METAMORPHOSYS, einer Zusammensetzung aus Metamorphose und System, verbirgt sich die Fähigkeit der dynamischen Adaption eines Rechensystems.

Existierende Ansätze adaptiver Systeme erlauben im Allgemeinen eine Ausrichtung der rekonfigurierbaren Hardware-Ressourcen des Systems auf ein Anwendungsgebiet oder ein einzelnes Programm. Die Anpassung muss vor dem Ausführungsbeginn erfolgt sein. Umgebungs- oder Anforderungsänderungen zur Laufzeit einzelner Anwendungsprogramme werden weder erkannt noch behandelt. Grundsätzlich fehlen diesen Ansätzen Mechanismen der Überwachung und Regelung, die eine Anpassung rekonfigurierbarer Hardware-Ressourcen während der Ausführung eines Anwendungsprogrammes ermöglichen.

Ziel von METAMORPHOSYS ist es, durch die Einbeziehung aller Systemschichten eine dynamische Anpassung der System-Hardware an ausgeführte Programme mit einem bisher unbekannten Grad an Feinkörnigkeit zu erlangen. Um dieses Ziel zu erreichen, ist es notwendig ein Konzept für ein gesamtes Rechensystem, das sich selbständig reguliert und auf allen Systemebenen – einschließlich rekonfigurierbarer Ressourcen – anpassen kann, zu entwerfen.

Dieses Kapitel beschreibt schrittweise das Konzept des im Zuge dieser Arbeit entstandenen autonomen adaptiven Gesamtsystems. Die Einführung in das System wird mit einem Schichtenmodell und einem Überblick über die Aufgaben der einzelnen Schichten begonnen. Im Weiteren folgt die Identifikation logischer Systemkomponenten. Die Beschreibung des Aufbaus und der Funktionsweise der Systemkomponenten führt zu den Auswirkungen auf die Systemteile und diesbezügliche Erweiterungen. Anschließend werden das Fundament, die hardware-basierte Überwachung und Regelung des Gesamtsystems sowie diesbezügliche Implementierungsaspekte vorgestellt.

In diesem Zusammenhang ist unter Gesamtsystem, konform zur Systemdefinition 2.3, ein Rechensystem mit allen verfügbaren Hardware-Komponenten und essentiellen Software-Komponenten zu verstehen.

5.1 Konzept des Gesamtsystems

Dieser Abschnitt beschreibt das grundlegende Konzept des autonomen adaptiven Systems. Zu Beginn steht die Einführung des Systems als Schichtenmodell. Im Anschluss daran werden die logischen Systemteile vorgestellt und deren Aufbau und Funktionalität erläutert.

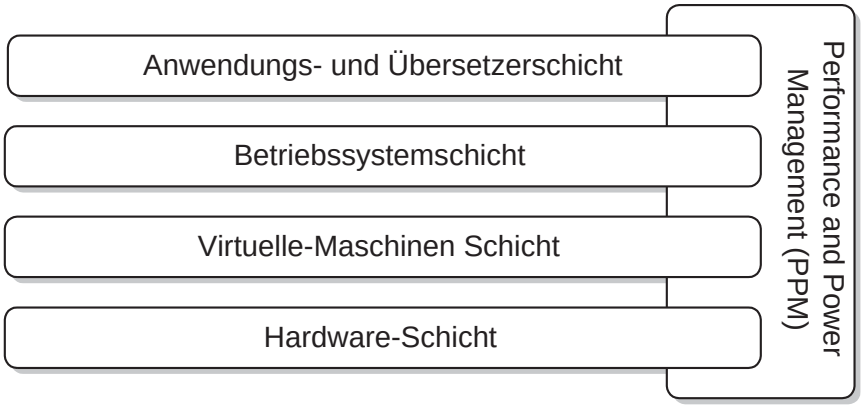

Abbildung 5.1: Die vier Schichten des adaptiven Systems

Als Schichtenmodell präsentiert sich das System in den in Abbildung 5.1 dargestellten vier Schichten:

- Die **Anwendungs- und Übersetzerschicht** umfasst sowohl alle auf dem System ausgeführten und ausführbaren Anwendungen als auch einen speziellen Übersetzer, der benötigt wird, um ein Programm für die Abarbeitung auf dem adaptiven System entsprechend vorzubereiten.

- Die **Betriebssystemschicht** ermöglicht die Regelung der darunter liegenden Schichten und bietet Anwendungen die Möglichkeit, auf vorgegebene Regelungs- und Steuermechanismen einzuwirken und diese zu manipulieren. Das Vorhandensein der Betriebssystemschicht ist – im Gegensatz zu den anderen Schichten – vom Einsatzgebiet des Systems abhängig. Speziell im Bereich der eingebetteten Systeme wird häufig auf ein Betriebssystem wegen Echtzeitkriterien oder Ressourcenbeschränkungen verzichtet. Wird vom Einsatz eines Betriebssystems abgesehen, müssen gegebenenfalls Regelungen von ein oder mehreren Anwendungen übernommen werden.

- Die **Virtuelle-Maschinen-Schicht** ist als leichtgewichtige Transformationsschicht mit minimaler Funktionalität für eine einheitliche Sicht der darunter liegenden Systemteile verantwortlich.

- Die **Hardware-Schicht** beinhaltet alle physischen Bestandteile des Systems, unabhängig davon, ob ein einzelner Bestandteil konfigurierbar, programmierbar oder unveränderbar vorliegt. In diesem Kontext wird darauf hingewiesen, dass konfigurierbar, in Bezug auf das adaptive System, stets die Fähigkeit der gesamten oder partiellen dynamischen Rekonfiguration einer physischen Komponente beschreibt.

Alle vier Schichten aus Abbildung 5.1 werden von einer vertikalen Ebene, die mit *Performance and Power Management (PPM)* beschriftet ist, geschnitten. Das bedeutet, dass jede horizontale Schicht des adaptiven Systems mindestens einen Bestandteil umfasst, der entweder die Regelungs- und Steuermechanismen des Systems unterstützt oder aktiv daran teil hat.

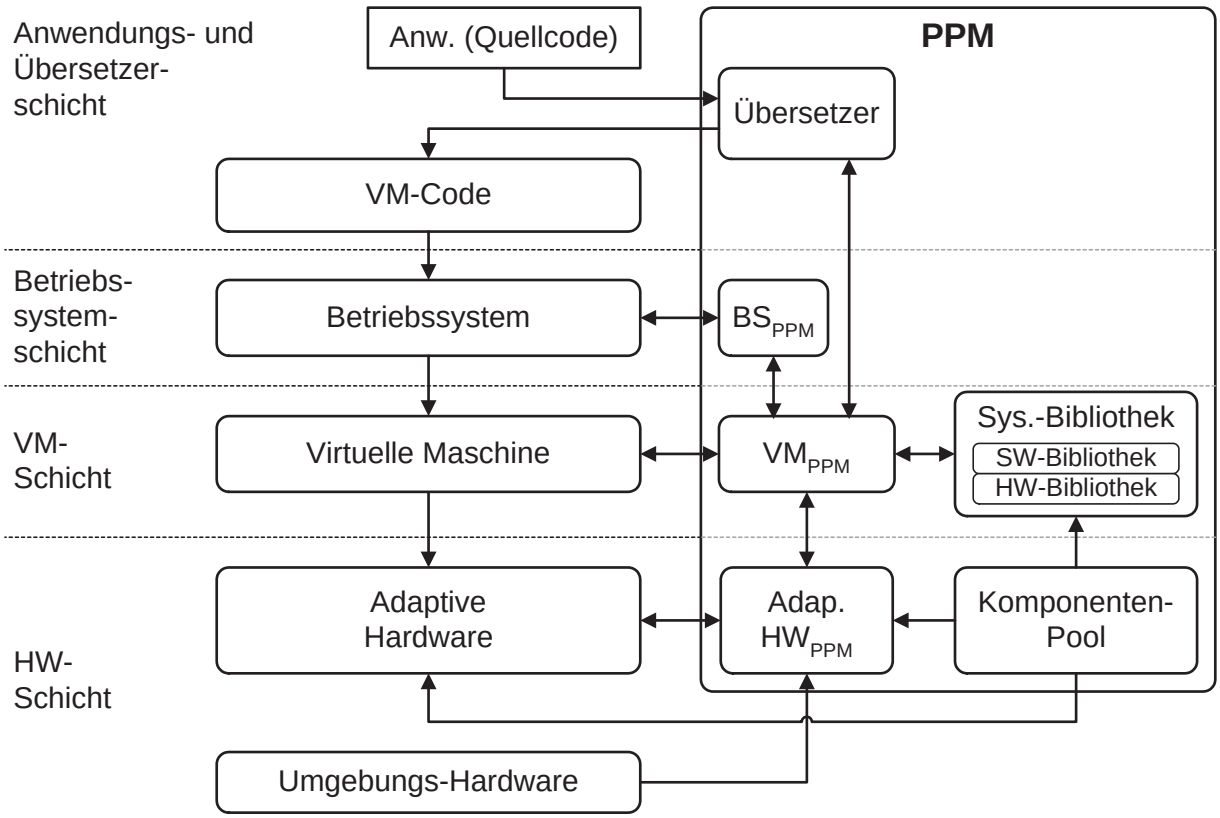

Abbildung 5.2: Die logischen Bestandteile des adaptiven Systems

Betrachtet man das adaptive System anhand seiner logischen Bestandteile, ergibt sich das in Abbildung 5.2 erkennbare Systemmodell. Die Unterteilung in die zuvor eingeführten Schichten ist durch gepunktete Linien angedeutet.

Das Kernstück des adaptiven Systems bildet die *Performance and Power Management (PPM)* Komponente. Diese umfasst insgesamt sechs Teilkomponenten. Hierbei wird der *Übersetzer* logisch der PPM zugeordnet, obwohl dieser nicht aktiv an der dynamischen Steuerung des Systems beteiligt ist. Zudem steht Übersetzer stellvertretend für den Compiler und zusätzliche Werkzeuge, wie Linker, Assembler usw., die insgesamt entweder der Übersetzung dienen oder als unterstützende Werkzeuge eingesetzt werden. Des Weiteren bilden die Teile des Betriebssystems (BS_{PPM}) sowie der virtuellen Maschine (VM_{PPM}) und der Hardware ($Adap.HW_{PPM}$) jeweils einen Bestandteil der PPM, die mit der dynamischen Systemregelung und der Bereitstellung von Regelungsmechanismen betraut sind.

Darüber hinaus befinden sich die Komponenten *Systembibliothek* und *Komponentenpool* innerhalb der PPM. Die Systembibliothek verwaltet Software- und Hardware-Lösungen für unterschiedliche Aufgaben. Zum einen bietet die Systembibliothek dem Anwendungspro-

grammierer an, bestimmte Algorithmen oder Teile von Algorithmen in Form von Bibliotheksaufrufen in eine Anwendung zu integrieren, und zum anderen macht sie alle systeminternen Hardware-Schaltungen, die für die Anpassung von konfigurierbaren Systemteilen notwendig sind, verfügbar. Alle Hardware-Schaltungen, im Weiteren auch als Module bezeichnet, liegen in Form von *Bitstreams* vor und sind im *Komponentenpool* gespeichert.

Die in Abbildung 5.2 eingezeichneten Verbindungslinien stellen die primären Kommunikationskanäle der logischen Komponenten untereinander dar. Die Pfeilrichtung steht stellvertretend für den Informations- und Datenfluss. Somit ergibt sich der prinzipielle Ablauf von der Anwendungsentwicklung bis hin zur Ausführung der Anwendung folgendermaßen: Ein Anwendungsprogramm liegt in Hochsprache vor und wird mit Hilfe der Übersetzer-Komponente in VM-Code übersetzt. Der Übersetzer kann mittels der VM_{PPM} auf die vorhandenen Software- und Hardware-Ressourcen der Systembibliothek zugreifen.

Die weitere Vorbereitung zur Ausführung des VM-Codes wird vom Betriebssystem übernommen. Hierunter fallen gängige Maßnahmen, wie z.B. Prozessverwaltung, Prozesskommunikation oder die Ablaufplanung (*Scheduling*) für mehrere Anwendungsprozesse [121]. Darüber hinaus müssen Mechanismen bereitgestellt werden, die die Adaptivität von physischen Komponenten mit berücksichtigen, was zwangsweise zu einer Erweiterung der Zustandsinformation (Kontext[1]) einer Anwendung führt (Details siehe Abschnitt 5.2). Des Weiteren fällt dem Betriebssystem die Aufgabe zu, die Rahmenbedingungen für die Regelung und Steuerung der tiefer liegenden Komponenten zu setzen. Zu den Rahmenbedingungen zählen z.B. *Dynamic Power-Management*, verschiedene Scheduling-Techniken, die die Leistungsaufnahme des Systems beeinflussen, oder *Dynamic Voltage Scaling* (Vgl. Grundlagenabschnitt 2.4). Alle Teile des Betriebssystems, die mit dem erweiterten Anwendungskontext und der Systemregelung in Verbindung stehen, bilden in Abbildung 5.2 die logische Komponente BS_{PPM}. Damit die Software-Teile des gesamten adaptiven Systems weitgehend unabhängig von der unterhalb befindlichen Hardware bleiben, greift die BS_{PPM}, ebenso wie der Übersetzer, nicht direkt auf die Systembibliothek oder Hardware-Komponenten zu, sondern nutzt die Methoden der VM_{PPM}.

Nachdem vom Betriebssystem das adaptive System auf die Ausführung einer Anwendung eingestellt wurde, gelangt diese zur virtuellen Maschine (VM). Bereits bei der Einführung des Schichtenmodells wurde darauf hingewiesen, dass die VM als leichtgewichtige Schnittstelle zwischen Software und Hardware fungiert. Deshalb wird die Einstellung der Systemrahmenbedingungen durch das Betriebssystem oder in Ausnahmefällen durch Benutzeranwendungen vollzogen[2].

Die Regelung des gesamten Systems ist mit den bekannten Ansätzen, die in Kapitel 3 dargestellt wurden, im Allgemeinen sehr rechenintensiv und meist anwendungsspezifisch. Letzteres bedeutet, dass bei einer Änderung des Einsatzgebietes des adaptiven Systems, stets die VM_{PPM} weitreichenden Änderungen unterzogen werden müsste. Um dies zu vermeiden, ist es sinnvoll, die VM als Schnittstelle zu konzipieren, die nicht aktiv an der Regelung partizipiert, aber eine einheitliche Sicht der an der Regelung beteiligten oder betroffenen Systemteile liefert und die hierzu benötigten Kommunikationsmechanismen bereitstellt. Als

[1] Im Zusammenhang mit Betriebssystemen wird der Begriff Kontext für die Zustandsinformation eines Prozesses benutzt. Bei rekonfigurierbaren Bausteinen bezeichnet Kontext einen Hardware- oder internen Speicher-Bereich, der eine Logikschaltung und somit Schaltungs-Information aufnehmen kann.

[2] Der direkte Zugriff von Anwendungen auf die Mechanismen der VM_{PPM} fehlt in Abbildung 5.2, aus Gründen der Übersichtlichkeit.

zusätzliche Aufgabe fällt der VM die Umwandlung des auszuführenden VM-Codes in Maschinencode zu. Für die Umwandlung werden zwei Vorgehensweisen für zwei Arten von VM-Instruktionen unterschieden:

1. Herkömmliche VM-Instruktionen, die mit einer bestehenden, für das adaptive System unveränderbaren Vorschrift auf Maschineninstruktionen abgebildet werden.

2. Adaptive VM-Instruktionen, die mit Hilfe der Systembibliothek aufgelöst werden, und zudem für unterschiedliche Systeme sowie für einzelne Anwendungen variieren können.

Die zweite Instruktionsart ist eine Besonderheit des adaptiven Systems und ermöglicht die Verwendung spezieller Hardware, die anwendungsspezifisch eine effiziente Nutzung der vorhandenen Systemressourcen eröffnet. Zur Veranschaulichung der Arbeitsweise der VM dient folgendes Beispiel:

Eine angenommene Anwendung benutzt eine Routine zur FFT-Berechnung (*Fast Fourier Transformation*). Dies wurde zur Übersetzungszeit erkannt und auf einen einzigen Befehl abgebildet, den die VM als FFT identifiziert. Stößt nun die VM bei der Ausführung auf diesen Befehl, sind zwei Situationen möglich:

1. Eine FFT-Implementierung ist aktuell als physische Funktionseinheit in der Hardware-Schicht des Systems verfügbar.

2. Es befindet sich keine solche Einheit in der aktuellen Hardwarekonfiguration.

Im ersten Fall wird die VM die Instruktion als eigenständigen Maschinenbefehl direkt an das System weiterleiten. Für den zweiten Fall muss dieser Befehl entweder durch eine Reihe von Maschinenbefehlen ersetzt oder die Hardwareschicht angehalten werden, ein entsprechendes Modul nachzuladen. Erfolgt aus diversen Gründen keine Rekonfiguration, wird eine Befehlsersetzung mit Hilfe der an die VM_{PPM} gebundenen Software-Bibliothek vollzogen.

An dieser Stelle bleibt die Frage offen: Wie kann während der Übersetzung ein Algorithmus oder Teilalgorithmus im Programmfluss erkannt werden? Ob und wie eine Erkennung in Zukunft möglich ist, kann nicht beantwortet werden. Der an dieser Stelle propagierte Lösungsweg liegt deshalb nicht in der Erkennung von (Teil-)Algorithmen, sondern beschränkt sich auf den Einsatz von Bibliotheksaufrufen und einer eingeschränkten, automatisierten Hardware-Erzeugung, wie sie im Garp-Projekt bzw. dessen Nachfolgeprojekten untersucht wurde (Vgl. Abschnitt 4.4.2). Das heißt, um Programme zu schreiben, die zugesichert eine hohe Rechenleistung und effiziente Ressourcennutzung des Systems anstreben, muss auf vorgefertigte Bibliotheken zurückgegriffen werden. Für das System andererseits bedeutet dies, dass eine Funktion im Software-Teil der Systembibliothek ihr Pendant in Form einer Funktionseinheit im Hardware-Teil haben sollte. Es ist nicht zwingend erforderlich, jede Bibliotheksfunktion auf ein Modul in Hardware abzubilden. Die Ausführung einer Anwendung ist in keinem Fall gefährdet, jedoch deren Ausführungs*zeit*.

Die Hardware- und Software-Bibliothek sind folglich für die Übersetzung und die Ausführung einer Anwendung von Bedeutung. Die Softwarebibliothek muss demnach über ein umfangreiches Funktions- bzw. Algorithmenrepertoire verfügen. Im Gegenzug dazu ist es erforderlich, dass alle – oder zumindest eine Vielzahl – dieser Bibliothekseinträge in Form von Hardware-Implementierungen zugänglich gemacht werden. Abgesehen von der hier vorgestellten Koexistenz von Software- und Hardware-Implementierung entspricht dies im Wesentlichen dem

in verschiedenen, voneinander unabhängigen, Arbeiten evaluiertem Einsatz von zusätzlicher Hardware in Form von Funktionseinheiten. Da die Rentabilität dieses Ansatzes in Projekten wie OneChip oder Garp usw. (siehe Abschnitt 4.4) bereits gezeigt werden konnte, ist es sinnvoll, diese Methode in ein adaptives Gesamtsystem unter oben angedachten Erweiterungen einzugliedern.

Außerdem erkennt man in Abbildung 5.2, dass die Systembibliothek an den *Komponenten-Pool* angeschlossen ist. Der Komponenten-Pool beinhaltet alle für das System benötigten und ladbaren Module bzw. logischen Schaltungen in Form von Bitstreams. Hier finden sich unter anderem die über die Hardwarebibliothek öffentlich gemachten Funktionseinheiten. Zudem müssen noch zusätzliche Schaltungen bereitgestellt werden, die für den *internen* Gebrauch allein dem System vorbehalten sind. Schnittstellen zu verschiedenen Bussystemen, Ein-Ausgabegeräten und Ähnliches, fallen in diesen Bereich. Eine weitere, spezielle Gruppe von Modulen ist ebenfalls dort angesiedelt: Die Gruppe *kombinierter Funktionseinheiten*. Hierbei handelt es sich um Funktionseinheiten, die sich an dem Prinzip der vertikalen Codierung orientieren und nicht auf die Abarbeitung von gesamten Algorithmen abzielen. Eine in der Zwischenzeit – besonders bei eingebetteten Systemen – weit verbreitete Einheit, die in diesem Kontext exemplarisch angeführt werden kann, ist die *Multiply Accumulate* (MAC) Einheit. Solche Funktionseinheiten sowie deren Aufbau und Anwendung im adaptiven System werden in Kapitel 6 detailliert behandelt.

Neben dem Komponenten-Pool existieren drei weitere Komponenten in der Hardware-Schicht: Die *Umgebungs-Hardware*, die stellvertretend für die Teile des Systems steht, die nicht veränderbar sind, über die dennoch Informationen vorhanden sein sollten. Als Paradebeispiel gilt in diesem Zusammenhang der Akku von mobilen Endgeräten. Es sollte zu jedem Zeitpunkt bekannt sein, über welche Restladung der Akku verfügt. Entsprechende Änderungen im System können anhand dieser Daten vorgenommen werden, um z.B. eine Verringerung des Strombedarfs herbeizuführen. Letzten Endes kann der Akku jedoch vom System selbst nicht verändert werden. Alle vom System in ihrer Struktur oder Funktionalität änderbaren Hardwareteile sind unter dem Begriff *adaptive Hardware* zusammengefasst. Jede physische Systemkomponente, die unter den Begriff adaptive Hardware fällt, wird als *adaptive Komponente* bezeichnet. Die dritte Komponente (Abbildung 5.2) auf dieser Ebene ist logisch der PPM zugeordnet ($Adap.HW_{PPM}$). Die Datensammlung und Auswertung, sowie die Auslösung von Rekonfigurationsaufgaben wird mit Hilfe dieser Komponente durchgeführt. Eine Rekonfiguration kann somit unmittelbar durch Abläufe innerhalb der Hardware-Schicht oder durch Anforderungen aus höheren Schichten initiiert werden. Für die Regelungsabläufe der Hardware-Schicht gilt, dass die Rahmenbedingungen bereits vom Betriebssystem vorgegeben sind. Weiterführende Zusammenhänge in Bezug auf die Überwachung und Regelung des adaptiven Systems werden in Abschnitt 5.3 thematisiert.

5.1.1 Aufbau der Systembibliothek und des Komponentenpools

Abbildung 5.3 zeigt die Bestandteile der Systembibliothek und des Komponentenpools sowie deren Zusammenhänge. Der linke Bildteil enthält die Bestandteile der Systembibliothek und des Komponentenpools, die zur Aufrechterhaltung der Systemfunktionalität notwendig und demzufolge für alle adaptiven Komponenten von Bedeutung sind. Im Gegensatz dazu werden die Bestandteile des rechten Bildteils ausschließlich für die Erweiterung des Funkti-

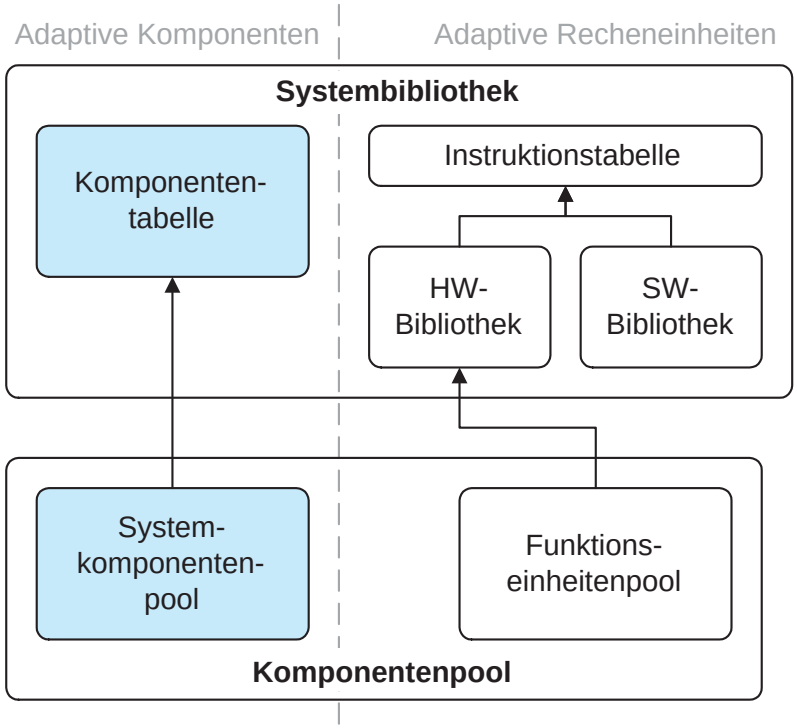

Abbildung 5.3: Komponenten der Systembibliothek und des Komponentenpools

onsumfangs adaptiver Recheneinheiten, wie z.B. des adaptiven Prozessors (Vgl. Kapitel 6) oder adaptiver Koprozessoren, eingesetzt.

Die Systembibliothek beinhaltet vier Komponenten: Die *Instruktionstabelle*, die eigentliche *Software-* sowie *Hardware-Bibliothek* und eine *Komponententabelle*. Die ersten drei Komponenten bilden eine logische Einheit und repräsentieren den Bestandteil der Systembibliothek, der für die Übersetzung und Ausführung von Anwendungen von Bedeutung ist. Die vierte Komponente, die Komponententabelle, ist von den anderen entkoppelt und realisiert die Referenzierung von systeminternen Bestandteilen, die der Kontrolle des Betriebssystems unterliegen.

Der Komponentenpool setzt sich aus zwei Teilen zusammen: Dem *Funktionseinheitenpool* und *Systemkomponentenpool*. Beide Pools stehen stellvertretend für eine Menge von Modulen, die in Form von Bitstreams vorliegen, und für den Einsatz in der adaptiven Hardware verfügbar sind.

Der Systemkomponentenpool ist an die Komponententabelle gebunden und mit Hilfe der VM für das Betriebssystem erreichbar. Im Systemkomponentenpool befinden sich diejenigen Schaltungen, die für die Aufrechterhaltung des Systems notwendig sind. Z.B. können hier verschiedene Konfigurationen für Ein-Ausgabegeräte, Bussysteme oder Netzwerkgeräte abgelegt werden. Hier befinden sich Module für Systemteile, die als Betriebsmittel vom Betriebssystem verwaltet und gesteuert werden, und somit von keiner Benutzeranwendung direkt genutzt werden können.

Dementgegen enthält der Funktionseinheitenpool die Module, die über die Software- und Hardware-Bibliothek direkt zur Ausführung von Anwendungen dienen. Die Verbindung

zwischen Modulen der Hardware-Bibliothek und dem korrespondierenden Eintrag in der Software-Bibliothek stellt die Instruktionstabelle her.

5.1.1.1 Schematischer Aufbau und Verwaltung der Software- und Hardware-Bibliothek

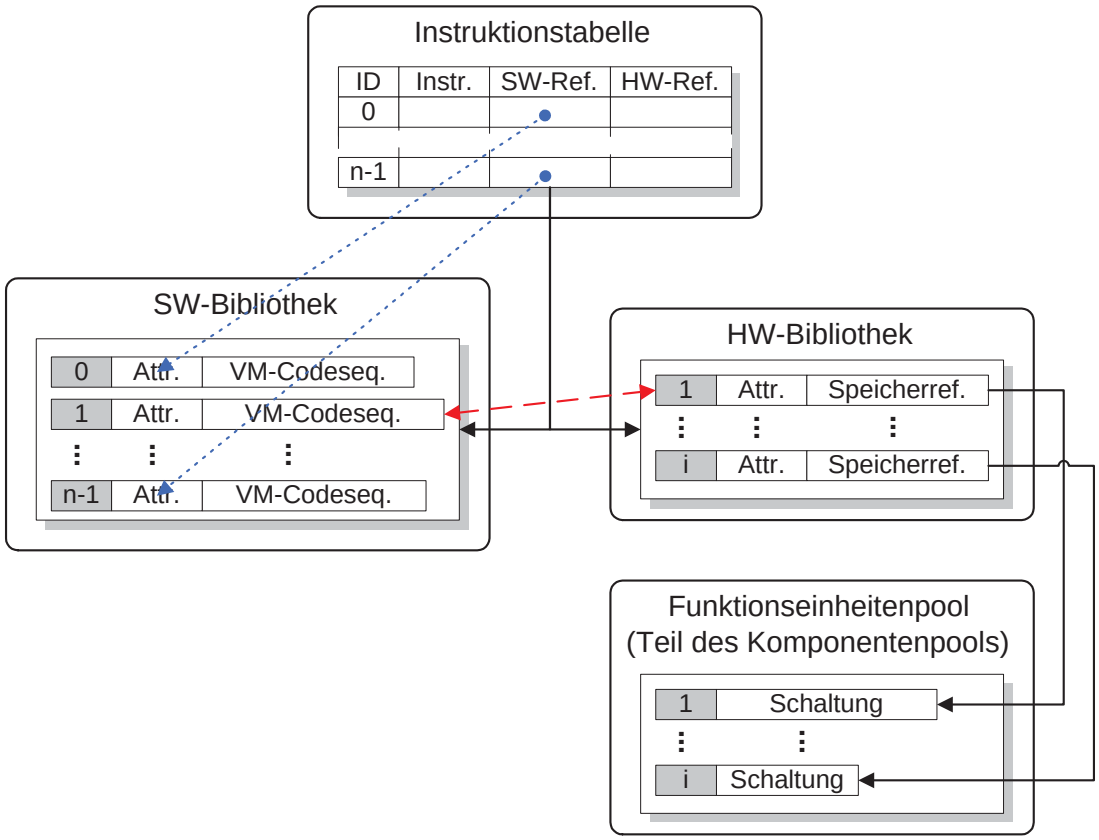

Abbildung 5.4: Struktur der Software- und Hardware-Bibliothek

Nachdem die Zusammenhänge der einzelnen Bestandteile vorgestellt wurden, greift dieser Abschnitt die gekoppelten Teile, die insgesamt die Software- und Hardware-Bibliothek bilden, heraus und beschreibt deren Aufbau. In Abbildung 5.4 erkennt man die Instruktionstabelle, die Software- und Hardware-Bibliothek und den daran angeschlossenen Funktionseinheitenpool.

Die Instruktionstabelle enthält vier Spalten für jeden Eintrag: Die *ID* ist ein eindeutiger Index, der die Identifizierung eines Eintrags in der Tabelle ermöglicht. *Instr.* repräsentiert den Maschinencode des korrespondierenden Bibliotheksaufrufs. Wird während der Übersetzung einer Anwendung ein Bibliotheksaufruf erkannt, wird dieser durch den in der Tabelle festgelegten Maschinencode ausgetauscht. Bei der Ausführung der Anwendung entscheidet die VM anhand der aktuellen Systemkonfiguration, ob der Befehl durch eine Befehlssequenz aus der Software-Bibliothek ersetzt, oder unverändert weitergeleitet wird. Dies bedeutet, dass der für die Ausführung verantwortliche Prozessor in der Lage sein muss, diesen Maschinencode einer Funktionseinheit zuzuordnen. Die Spalte *SW-Ref.* enthält die Verknüpfung in

die eigentliche Software-Bibliothek. Dementsprechend referenziert *HW-Ref.* das zugehörige Modul, respektive Schaltung, in der Hardware-Bibliothek.

Für die Software-Bibliothek gilt, dass jeder Eintrag genau einem Eintrag der Instruktionstabelle zugewiesen ist. In der Hardware-Bibliothek können Einträge fehlen. Dennoch muss jeder gültige Eintrag der Hardware-Bibliothek einer Schaltung, in Form eines Bitstreams, im Funktionseinheitenpool eindeutig identifizierbar sein. Zur Verdeutlichung wurde in Abbildung 5.4 die jeweilige ID grau hinterlegt eingezeichnet und nummeriert. Das heißt, in der exemplarischen Darstellung sind für i Werte von 2 bis $n-1$ zulässig. Die gestrichelte Linie markiert die sich in dieser Darstellung eindeutig zuordenbaren Gegenspieler. Die gepunkteten Linien heben die sich in der Abbildung ergebenden Referenzen hervor. Die Spalte *Attr.* (Attribut), die sowohl in der Software- als auch in der Hardware-Bibliothek zu finden ist, beschreibt die Eigenschaften des zugehörigen Eintrags. Für die VM-Codesequenzen (*VM-Codeseq.*) beinhaltet diese Spalte jeweils die Länge der folgenden Codesequenz in Maschinenworten. Die Attribute auf der Hardware-Seite umfassen darüber hinaus weitere Eigenschaften, wie z.B. den durchschnittlichen Energiebedarf, die minimale Auswirkung auf die Rechenleistung, falls die Schaltung eingesetzt wird, usw.

5.1.1.2 Verwaltung der Systemkomponenten-Module

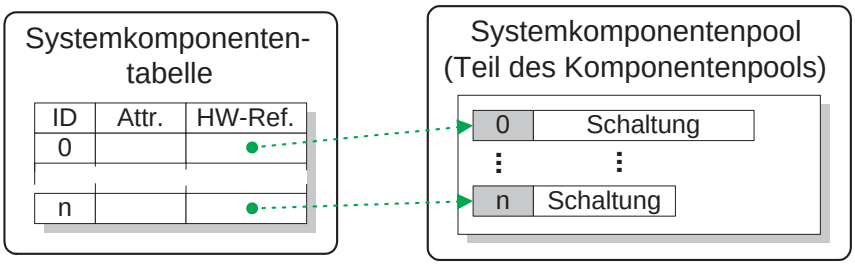

Abbildung 5.5: Verwaltung der Systemkomponenten

Abbildung 5.5 skizziert den Aufbau der Systemkomponententabelle und des Systemkomponentenpools. Die Systemkomponententabelle enthält einen eindeutigen Index (*ID*) für jeden Eintrag. Die Eigenschaften der Module sind in der Spalte *Attr.* abgelegt. Eine Eigenschaft beschreibt die Größe der referenzierten Schaltung als Bitstream. Weitere Eigenschaften sind z.B. die maximale Übertragungsrate für Bus- oder allgemein Kommunikations- und Speicherkomponenten, usw. Die letzte Spalte verweist auf die Schaltung innerhalb des Systemkomponentenpools.

Der Systemkomponentenpool beinhaltet alle Module, bzw. Schaltungen, in Form von Bitstreams. Jedes Modul muss einem Eintrag in der Systemkomponententabelle eindeutig zugewiesen sein und umgekehrt.

5.2 Nutzung und Auswirkung der Adaptivität

Um die Möglichkeiten des vorgestellten adaptiven Konzepts nutzen zu können, müssen verschiedene funktionale Erweiterungen der Systemteile, im Gegensatz zu herkömmlichen, nicht veränderbaren Systemen, vorgenommen werden. Diese Erweiterungen sind unabhängig von den Regelungs- und Steuermechanismen, die in Abschnitt 5.3 diskutiert werden. Im Folgenden handelt es sich ausschließlich um grundlegende Probleme und deren Lösungen, die sich allgemein aus dem Einsatz adaptiver Systemkomponenten ergeben.

5.2.1 Einsatz adaptiver Recheneinheiten

Die adaptiven Recheneinheiten repräsentieren eine spezielle Kategorie adaptiver Komponenten. Hierzu zählen beispielsweise der adaptive Prozessor (siehe Kapitel 6) oder adaptive Koprozessoren, DSPs (Digitale Signal Prozessoren) usw. Allgemein werden alle adaptiven Komponenten, die eine eigenständige Abarbeitung von Maschinenbefehlen aufweisen, als adaptive Recheneinheiten bezeichnet. Eine gesonderte Behandlung ist deshalb erforderlich, da die Anpassungsfähigkeit solcher Komponenten bereits zur Übersetzungszeit von Anwendungen, bzw. bei der Codegenerierung, mit berücksichtigt werden muss. In diesem Zusammenhang ergeben sich verschiedene Erweiterungen im Hinblick auf die Anwendungsübersetzung, sowie diesbezüglich notwendige Anpassungen der Systembibliothek und des Komponentenpools.

5.2.1.1 Anwendungsübersetzung für das adaptive System

Für die Erzeugung einer Anwendung, die die Möglichkeiten des adaptiven Systems nutzt, stehen grundlegend zwei Varianten, wie sie in Abbildung 5.6 dargestellt sind, zur Verfügung:

1. Der Benutzer greift auf bereits vorgefertigte Bibliothekseinträge des Systems zurück.

2. Der Compiler des Systems soll eigenständig versuchen, durch eine automatisierte Software- und Hardware-Generierung performanten Code für die virtuelle Maschine zu erzeugen.

Teilbild 5.6 a) zeigt den Übersetzungsmechanismus, falls der Benutzer einen *Bibliotheksaufruf* manuell in eine Anwendung integriert. Der Compiler liest den korrespondierenden Aufruf im Quellcode: Wird der entsprechende Eintrag in der Softwarebibliothek erkannt, trägt der Compiler an der korrespondierenden Position im VM-Code (siehe Abbildung 5.6 a) die zugehörige Bibliotheksinstruktion (*Bib.-Instr.*) ein.

Als zweite Übersetzungsvariante illustriert Teilbild 5.6 b) die automatische Generierung von Schaltungs- und VM-Codeteilen für die Ausnutzung des adaptiven Systems. Bereits vor einigen Jahren wurde diese Methode im Zuge des Garp-Projekts untersucht. Hierbei erzeugt der Compiler selbstständig Hardware-Schaltungen aus inneren Schleifen von ANSI C-Programmen (siehe Abschnitt 4.4.2). Der Nachteil dieser Methode ist, dass ein einmal erzeugter Anwendungscode nur in Zusammenhang mit den generierten Hardware-Modulen

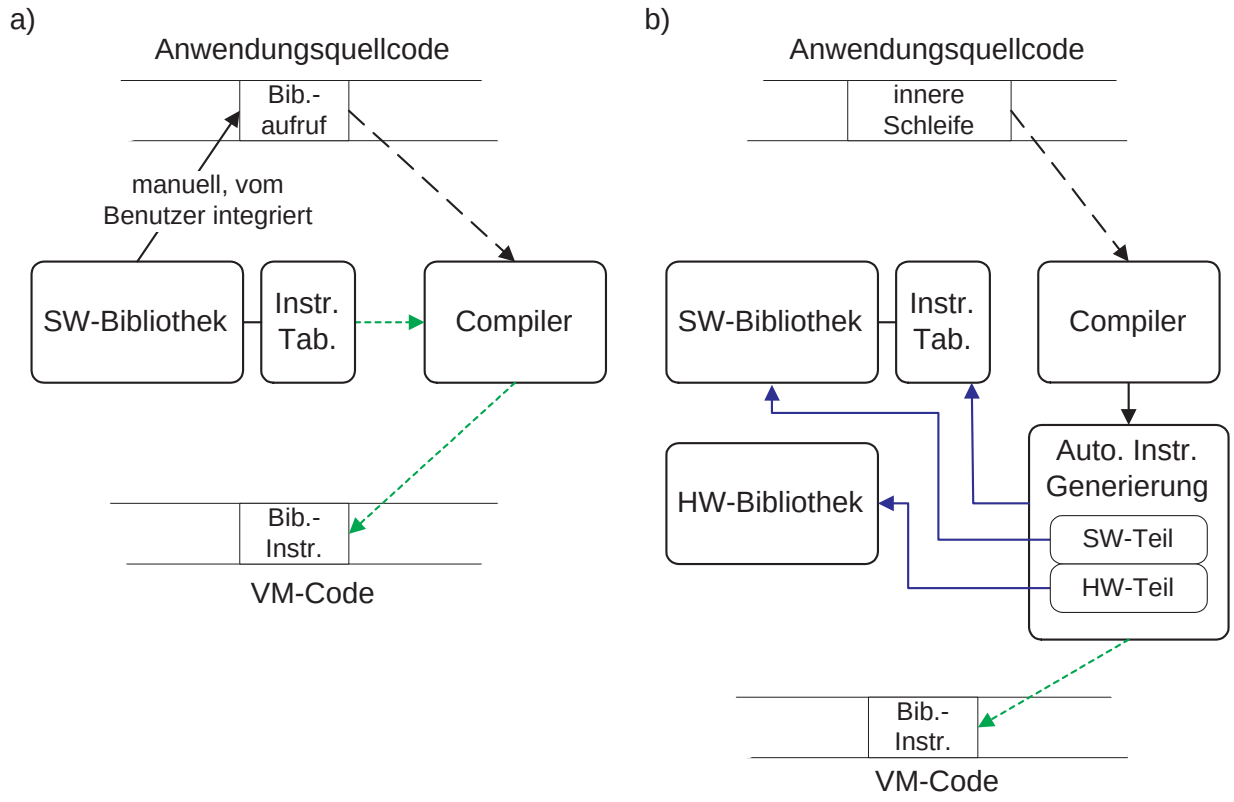

Abbildung 5.6: Codegenerierung aus einer Hochsprache

läuft. Für das in dieser Arbeit behandelte adaptive System muss die automatisierte Codeerzeugung deshalb um die zusätzliche Erzeugung des VM-Codes – als Pendant zur Hardware-Variante – erweitert werden.

Eine diesbezüglich geeignete Vorgehensweise liegt z.B. als Erweiterung des Garp-Compilers vor [29]. Darüber hinaus zeigen A. Koch *et al.* [93] eine verbesserte Variante, die im Zusammenhang mit dem adaptiven System, bei der Generierung von VM-Code und Hardware-Schaltungen, favorisiert wird.

In Abbildung 5.6 ist die Erzeugung der VM- und Hardware-Lösung entsprechend durch die Aufteilung in Software- und Hardware-Teil angedeutet. Die Ausführung der Anwendung kann mit und ohne Hardware-Unterstützung erfolgen. Darüber hinaus ist es notwendig, einen eindeutigen Operationscode für die Bibliotheksinstruktion, die im VM-Code eingefügt wird, zu erzeugen, und diesen in der Instruktionstabelle (*Instr.Tab.*) einzutragen.

Zu den beiden beschriebenen Methoden der Anwendungserstellung und Übersetzung müssen Mechanismen bereitgestellt werden, die es dem versierten Benutzer erlauben eigene Bibliothekseinträge zu erzeugen und diese, je nach Bedarf, in der Systembibliothek zu integrieren. Speziell in Hinblick auf die Energieeffizienz von Schaltungen (Vgl. Grundlagenabschnitt 2.4) oder anwendungsspezifischen Anforderungen muss dies gewährleistet sein.

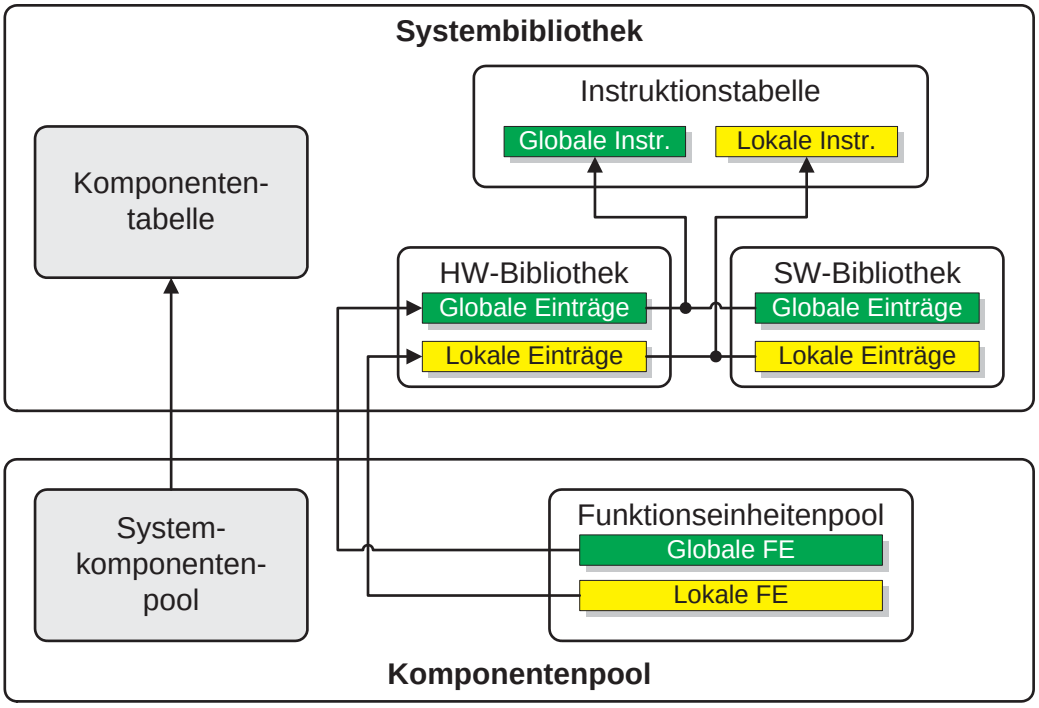

Abbildung 5.7: Erweiterung der Systembibliothek und des Komponentenpools

5.2.1.2 Erweiterung der Systembibliothek und des Komponentenpools

Ein Nachteil der automatischen Schaltungserzeugung ist die schlechte Wiederverwendbarkeit der generierten Schaltungen. Dies gilt auch innerhalb einer einzigen Anwendung, da bereits geringe Implementierungsunterschiede in den betrachteten Schleifen zu neuen Hardware-Ergebnissen führen. Bei verschiedenen Anwendungen sind die jeweils auf diese Weise erstellten Schaltungen in der Regel nicht mehrfach verwendbar. Um dieses Problem zu lösen, wird für das adaptive System eine Unterteilung der relevanten Systembibliothekskomponenten in einen globalen und lokalen Bereich vorgeschlagen (Abbildung 5.7). Die Komponententabelle und der Systemkomponentenpool sind von dieser Unterteilung ausgenommen. Betroffen sind die Komponenten, die in direktem Zusammenhang mit der Ausführung einzelner Anwendungen stehen. Man erkennt in Abbildung 5.7, dass die Instruktionstabelle, die Software- und Hardware-Bibliothek und der zugehörige Funktionseinheitenpool jeweils in einen globalen und einen lokalen Bereich eingeteilt wurden. Der globale Bereich enthält Einträge, die allen Anwendungen bereitstehen. Im lokalen, anwendungsspezifischen Bereich finden sich umgekehrt jeweils die für die aktuell ausgeführte Anwendung benötigten Einträge.

Der Aufbau beider Bereiche ist innerhalb einer Komponente identisch und entspricht im Wesentlichen dem in Abschnitt 5.1.1.1 vorgestelltem Schema. Zusätzlich muss beachtet werden, dass die Eindeutigkeit der Bibliotheksinstruktionen gewährleistet ist. Jeder lokale Bibliotheksteil ist nur im Zusammenhang mit der entsprechenden Anwendung gültig. Somit referenziert ein bestimmter lokaler Index für unterschiedliche Anwendungen im Normalfall unterschiedliche Code- bzw. Schaltungsteile. Andererseits darf kein lokaler Index mit einem globalen Index übereinstimmen.

Die Anzahl der Einträge innerhalb beider Bereiche ist aus konzeptioneller Sicht nicht begrenzt. Für eine Implementierung sind die verfügbaren Ressourcen im gewählten Einsatzgebiet ausschlaggebend. In diesem Zusammenhang ist zu bedenken, dass eine Limitierung der Anzahl der Tabelleneinträge nicht zwangsläufig einen geringen Ressourcenbedarf zur Folge hat. Der benötigte Speicherplatz ist zusätzlich von der Länge der VM-Codesequenzen der Software-Bibliothek und der Größe der vorhandenen Module bzw. des zugehörigen Bitstreams des Funktionseinheitenpools abhängig.

Darüber hinaus müssen weitere Überlegungen, die vom Einsatzgebiet des Systems und den ausgeführten Anwendungen abhängen, bei einer Umsetzung des adaptiven Systems berücksichtigt werden. Gelten starke Ressourcenrestriktionen, muss gegebenenfalls auf einen lokalen Bibliotheksteil verzichtet werden. Eine Implementierung ohne lokalen Teil kann des Weiteren durch vorgegebene Laufzeitkriterien erforderlich sein, da die Verwaltung lokaler Einträge zwangsläufig eine Erweiterung des Anwendungskontexts nach sich zieht und evtl. zeitliche Konflikte beim Anwendungswechsel hervorbringt. Im Gegensatz zu den Beschränkungen durch äußere Einflüsse, sind zusätzliche Erweiterungen vorstellbar. Lassen sich z.B. die auf dem System zur Ausführung gebrachten Anwendungen dergestalt kategorisieren, dass für jeweils mehrere Anwendungen gleiche oder ähnliche Algorithmen eingesetzt werden können, kann es sinnvoll sein, zu den oben eingeführten lokalen und globalen Bereichen einen dritten einzuführen, der jeweils Hardware-Lösungen für eine ganze Gruppe von Anwendungen bereithält.

Alle diese Überlegungen greifen erst dann, wenn eine genaue Spezifikation des Einsatzgebietes, der Umgebungsbedingungen und Anwendungsanforderungen bekannt ist, und sind nur der Vollständigkeit halber angeführt. Zudem ergeben sich hieraus keine grundsätzlichen Änderungen für das vorgestellte Konzept.

5.2.1.3 Aufbau des VM-Codes für das adaptive System

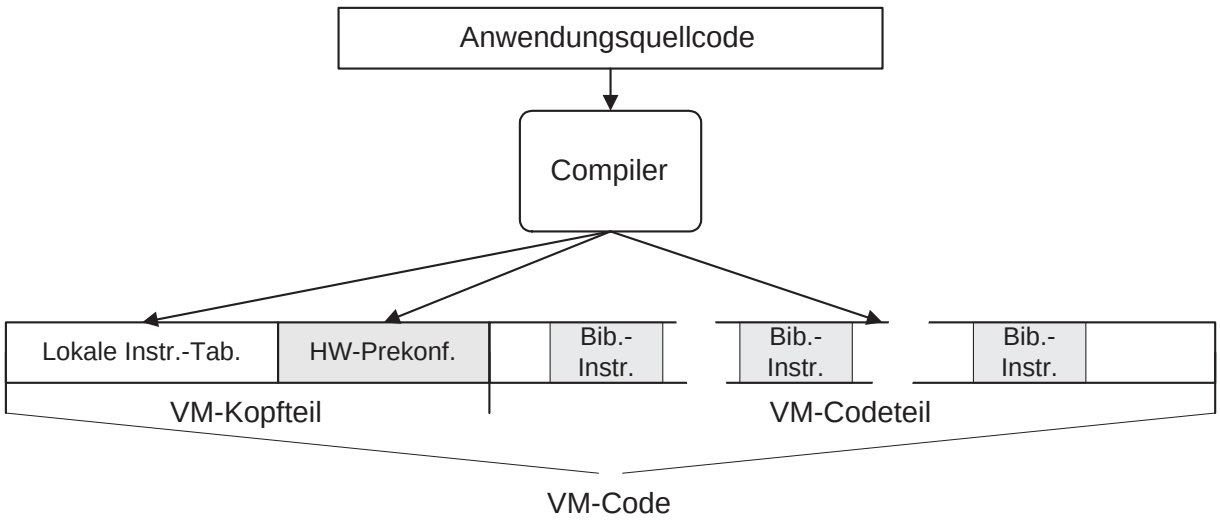

Abbildung 5.8: Aufbau eines generierten VM-Programms

Nach den obigen Ausführungen wird ein Anwendungsquellcode allgemein in einen VM-Code, wie in Abbildung 5.8 skizziert, übersetzt. Der resultierende Code kann in einen *Kopfteil* und einen reinen *Codeteil* zerlegt werden.

Der Kopfteil beinhaltet die Daten für die lokalen Bibliothekseinträge (*lokale Instr.-Tab.*), sowie Anweisungen für eine Vorkonfiguration (*HW-Prekonf.*) der adaptiven Hardware. Ob der lokale Bibliotheksteil vorhanden ist, hängt von den verwendeten Bibliotheksinstruktionen ab. Werden nur globale Aufrufe durchgeführt, entfällt dieser Teil. Die Vorkonfiguration ist eine optionale Erweiterung, die es ermöglicht statische Information, die bereits zur Übersetzungszeit bekannt ist, zu Beginn der Ausführung einer Anwendung einfließen zu lassen. Der Vorkonfigurationsteil besteht aus Konfigurationsinstruktionen, die im Folgenden behandelt werden, und wird als Teil der Anwendung interpretiert und vor dem eigentlichen Programmcode zur Ausführung gebracht. Eine solche Vorgehensweise erlaubt vorab eine Anpassung der adaptiven Hardware. Die eigentliche Adaptivität des Systems greift erst während der Laufzeit. Somit ist die Vorkonfiguration nur eine unterstützende Maßnahme, um evtl. entstehende Latenzen durch das Laden und Konfigurieren der Hardware zu verringern. Zusätzlich sollte beachtet werden, dass, falls keine statische Vorkonfiguration erfolgt, das System sozusagen mit einer *Standardkonfiguration* starten muss, und erst zur Ausführungszeit durch das Anwendungsverhalten Veränderungen erwogen werden können, die gegebenenfalls zur Übersetzungszeit bekannt sind.

Der Codeteil besteht aus VM-Instruktionen mit zwischengelagerten Bibliotheksinstruktionen, die zur Ausführungszeit von der VM-Schicht des Systems entsprechend mit den Instruktionen der Software-Lösung ersetzt, oder, bei verfügbarer Hardware, direkt weitergeleitet werden.

Als Erweiterung der VM-Instruktionen sind die Konfigurationsinstruktionen zu sehen. Diese ermöglichen ein software-gesteuertes Konfigurieren der darunter liegenden adaptiven Hardware. Das Einfügen von Konfigurationsinstruktionen muss sowohl eigenständig vom Übersetzer als auch unter Anweisung vom Programmierer, mit Hilfe von Übersetzerdirektiven, durchführbar sein.

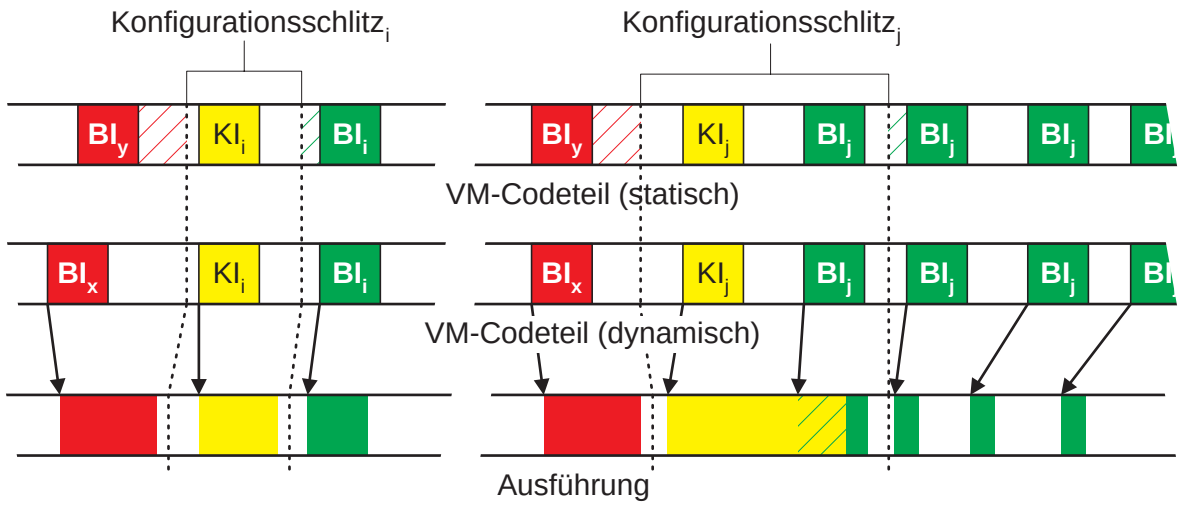

Abbildung 5.9: Konfigurationserweiterungen für den VM-Code

Abbildung 5.9 zeigt den Einsatz der Konfigurationsinstruktionen innerhalb zweier unabhängiger Codefragmente (linker bzw. rechter Bildteil) einer fiktiven, übersetzten Anwendung. Die oberste Zeile repräsentiert die statische Codesicht des Übersetzers. Darunter, als zweite Zeile, ist die korrespondierende Codesequenz zur Laufzeit abgebildet. Beide Zeilen repräsentieren jeweils nur eine Reihenfolge. Die dritte und unterste Zeile zeigt das exemplarische Ausführungsverhalten beider Codeteile. Diese Zeile stellt einen möglichen zeitlichen Verlauf dar.

Hierbei ist, für die statische und dynamische Codesicht, die Reihenfolge der Instruktionen sowie die Position der hervorgehobenen Instruktionen von Bedeutung. Die unterste Zeile betrachtet dementgegen das zeitliche Verhalten des Codes. Dies ist eine rein qualitative Betrachtung, um die Problematik aus Sicht des Übersetzers zu verdeutlichen. Das reale Laufzeitverhalten der gleichen Codesequenz variiert im Normalfall von Ausführung zu Ausführung, kann aber durch *worst-case* und *best-case* Abschätzungen für ein real existierendes System eingegrenzt werden. Für die Betrachtungen im Zusammenhang mit Abbildung 5.9 ist eine exakte Bestimmung des Laufzeitverhaltens im Verhältnis zur zugehörigen Instruktion nicht notwendig. Es soll die allgemeine Problematik der Rekonfiguration veranschaulicht werden. Umgekehrt können Sonderfälle auftreten oder vom Übersetzer erzwungen werden, die eine stark vereinfachte Behandlung erlauben. Eine Darstellung der Sonderfälle erfolgt im Anschluss an die allgemeine Betrachtung.

Abbildung 5.9 stellt die beiden grundsätzlichen Konstellationen bei der Verwendung von Konfigurationsinstruktionen dar:

1. Die Konfigurationsinstruktion betrifft ein einziges Auftreten einer Bibliotheksinstruktion (linker Bildteil). D.h. die angeforderte Konfiguration muss spätestens bis zur Ausführung der Bibliotheksinstruktion abgeschlossen und die entsprechende Hardware verfügbar sein.

2. Dem Übersetzer ist bereits zur Übersetzungszeit bekannt, dass eine bestimmte Bibliotheksinstruktion mehrfach ausgeführt wird. In diesem Fall kann die Konfiguration mit einer (wie im rechten Bildteil exemplarisch gezeigt) oder mehreren Bibliotheksinstruktionen überlappend stattfinden. Bis die Konfiguration abgeschlossen ist, werden die zwischenzeitlichen Bibliotheksaufrufe mit den entsprechenden VM-Codesequenzen der Software-Bibliothek abgearbeitet.

Die *Konfigurationsschlitze* in beiden Bildteilen repräsentieren den Codebereich, der für eine Konfiguration in Frage kommt. Somit entspricht ein Konfigurationsschlitz einer Menge von aufeinander folgenden Codepositionen.

Die Anfangsposition eines Konfigurationsschlitzes ergibt sich aus der vorhergehenden Bibliotheksinstruktion BI_y und der zugehörigen *worst-case* Abschätzung der Ausführungszeit dieser Instruktion in Hardware (schraffierter Bereich im Anschluss an BI_y). Die Ausführungszeit, die aus einer Codeersetzung resultiert, ist nicht von Bedeutung, da in diesem Fall keine adaptiven Ressourcen von BI_y belegt sind, und deshalb die Konfiguration ohne Konflikte durchgeführt werden kann. Das vom Übersetzer als vorhergehende Bibliotheksinstruktion ermittelte BI_y ist nicht zwangsläufig identisch mit der Bibliotheksinstruktion BI_x, die während der tatsächlichen Ausführung vor der Konfigurationsinstruktion KI_i bzw. KI_j auftritt. Der Grund dafür liegt in der statischen Anwendungssicht des Übersetzers. Demzufolge muss BI_y stets so gewählt sein, dass, unabhängig vom dynamischen Codever-

lauf, keine Konflikte mit einer nachfolgenden Konfiguration auftreten. Exemplarisch zeigt der linke Teil der Abbildung 5.9 den Fall, in dem $BI_y \neq BI_x$ gilt. Im rechten Teilbild wurde $BI_y = BI_x$ gewählt.

Um die Endposition eines Konfigurationsschlitzes zu erhalten, müssen die Dauer der angeforderten Konfiguration sowie die *best-case* Abschätzung der Ausführungszeit bis zum Auftreten der betreffenden Bibliotheksinstruktion betrachtet werden. Ein Konfigurationsschlitz ist demzufolge eine Abbildung des Ausführungsverhaltens auf statische Codepositionen unter Berücksichtigung einer konfliktfreien Konfiguration (gepunktete Linien). Da dem Übersetzer keine – oder nur in beschränktem Maße – Laufzeitinformation zur Verfügung steht, müssen Anfangs- und Endposition eines Konfigurationsschlitzes abgeschätzt werden. Die Einschränkungen, die sich daraus für die Codepositionen ergeben, sind exemplarisch in Abbildung 5.9 schraffiert angedeutet.

Neben der allgemeinen Betrachtung der Konfigurationsproblematik aus Sicht des Übersetzers existieren diesbezüglich folgende Sonderfälle:

- Falls nur eine einzige Bibliotheksinstruktion innerhalb der gesamten Anwendung oder in Anwendungsphasen, die sich eindeutig unterscheiden lassen, auftritt, ist es ausreichend, die Konfiguration zu Beginn der Anwendung bzw. einer Phase durchzuführen. Die adaptive Hardware, respektive der adaptive Prozessor, muss angewiesen werden die entsprechende Hardware dauerhaft, bzw. bis zur Freigabe durch die Anwendung, unverändert beizubehalten.

- Treten nur gleiche Bibliotheksinstruktionen in einer Anwendung bzw. in Anwendungsphasen auf, ist wiederum eine Konfiguration zu Beginn der Anwendung bzw. jeder Anwendungsphase möglich.

- Es sind ausreichend adaptive Ressourcen vorhanden, um alle angeforderten Konfigurationen ohne Konflikte durchführen zu können.

Der letzte Sonderfall ist, in Bezug auf den Einsatz des adaptiven Systems im Bereich der eingebetteten Systeme, äußerst unwahrscheinlich. Zudem müssen Vorkehrungen getroffen werden, um etwaige Echtzeitanforderungen erfüllen zu können. Aus diesem Grund sind zwei unterschiedliche Arten von Konfigurationsinstruktionen vorgesehen:

1. Eine **weiche Konfigurationsinstruktion**, die als Hinweis der Anwendung für die virtuelle Maschine und die adaptive Hardware zu interpretieren ist. Das bedeutet, dass die Konfiguration wünschenswert – aus Sicht der Anwendung – ist, aber keine Laufzeitanforderungen verletzt werden, falls die geforderte Hardware nicht zur Verfügung steht.

2. Eine **harte Konfigurationsinstruktion**, die die Hardware anweist, in jedem Fall eine Konfiguration durchzuführen, um vorgegebene Echtzeitkriterien erfüllen zu können.

In diesem Zusammenhang kann man eine Erweiterung der beiden Konfigurationsarten um eine mehrstufige Priorität einzelner Konfigurationsinstruktionen in Erwägung ziehen. Ob dies Vorteile gegenüber der Einordnung in weiche und harte Anweisungen bringt, muss in weiterführenden Arbeiten untersucht werden.

5.2.2 Erweiterung des Betriebssystems

Der Zustand adaptiver Komponenten wird aus Sicht des Betriebssystems wie der aller anderen Betriebsmittel behandelt. Zudem muss für adaptive Komponenten die aktuelle Konfigurationsinformation gesichert werden. Diese Erweiterung führt im Vergleich zu nicht adaptiven Systemen zwangsläufig zu einer Vergrößerung der Zustandsinformation [121, 135]. Die bereits in Abschnitt 5.1 eingeführte Systembibliothek und die diesbezügliche Referenzierung von Hardware-Schaltungen mit Hilfe eindeutiger Indices erlaubt es, die Konfigurationsinformation einer Komponente ausschließlich durch die entsprechenden Indices aus der Systembibliothek abzulegen. Die Sicherung der Hardware-Konfiguration in Form eines Bitstreams ist nicht erforderlich. Darüber hinaus existieren physikalische Grenzen, die für jede adaptive Komponente die Anzahl der zur gleichen Zeit geladenen Module und demzufolge die zusätzlich nötige Zustandsinformation einschränken.

Des Weiteren können adaptive Komponenten ausgemacht werden, die zu jedem Zeitpunkt nur jeweils eine einzige Schaltung als Konfiguration zulassen. Unter anderem gilt dies für adaptive Ein-Ausgabekomponenten, die einerseits im Verlauf einer Anwendung umkonfiguriert werden dürfen, andererseits zu jedem Zeitpunkt, auf Grund ihrer physikalischen Schnittstellen, nur die Funktionalität einer einzelnen Schaltung ausführen können. Dies bedeutet für die Zustandsinformation, dass für solche Komponenten nur ein einzelner zusätzlicher Wert, nämlich der Index der aktuellen Schaltung, gerettet werden muss.

Für adaptive Recheneinheiten, wie z.B. den adaptiven Prozessor, ist prinzipiell eine beliebige und variable Anzahl der geladenen Module denkbar. Module für adaptive Recheneinheiten sind in erster Linie Funktionseinheiten. Die Restriktionen für solche adaptiven Komponenten ergeben sich aus zwei beschränkenden Faktoren: Zum einen ist es auf Grund des limitierten Parallelismus auf der Instruktionsebene (ILP) von Anwendungen allgemein nicht sinnvoll, beliebig viele Funktionseinheiten in einen Prozessor zu integrieren, und zum zweiten ergeben sich technische und physikalische Probleme, die durch die nötigen Verbindungsstrukturen der Prozessorteile untereinander entstehen (Vgl. Kapitel 6). Für das adaptive System OneChip [154], das über eine rekonfigurierbare Einheit verfügt, die zu einem Zeitpunkt nur eine Schaltung zulässt, wäre dementsprechend ein Index pro Anwendungswechsel (Taskwechsel) zusätzlich zu den Allzweckregistern und Statusregistern des Prozessors notwendig. Das in dieser Arbeit vorgestellte adaptive System erlaubt mehrere Module gleichzeitig im adaptiven Prozessor. Eigene Simulationsergebnisse belegen, dass die Nutzung der ladbaren Schaltungen je Anwendung und Funktionsumfang der einzelnen Schaltungen auf sechs bis acht zu jedem Zeitpunkt beschränkt ist. Die diesbezüglichen Untersuchungen sind detailliert in Kapitel 6 beschrieben.

Insgesamt steigt der Aufwand für einen Wechsel der Anwendungen (Prozesswechsel) im adaptiven System im Gegensatz zu nicht adaptiven Systemen. Der Mehraufwand durch die Sicherung der Konfigurationsinformation ist dennoch verhältnismäßig gering. Problematisch ist in diesem Zusammenhang die Zeit, die für die Wiederherstellung einer Konfiguration der adaptiven Teile des Systems benötigt wird. Die Konfigurationszeit ist zudem ein Nachteil, der das gesamte adaptive System betrifft und bestimmte Maßnahmen technischer Natur erfordert, die nachfolgend in einem gesonderten Abschnitt behandelt werden. Des Weiteren entstehen durch Prozesswechsel verschiedene Konfliktsituationen in Verbindung mit der Rekonfigurationsfähigkeit des Systems.

Darüber hinaus muss zwischen Systemzustand, dem funktionalen Zustand zu einem bestimmten Zeitpunkt, der sich durch Sichern von Registerinhalten festhalten lässt, und Hardware-Zustand, der den logischen und funktionalen Zustand einer adaptiven Komponente beschreibt, unterschieden werden. Die herkömmliche Zustandsinformation, die ein Betriebssystem für Prozesswechsel benötigt, enthält den funktionalen Systemzustand zu einem festgelegten Zeitpunkt. Als Zusatz enthält die Zustandsinformation des adaptiven Systems die Daten der aktuellen Hardware-Konfiguration in Form von eindeutigen Indices (Konfigurationsinformation), die jeweils eine einzige Schaltung eindeutig referenzieren. Auch die erweiterte Zustandsinformation des adaptiven Systems beschreibt ausschließlich den funktionalen Zustand des Systems. Im Gegensatz dazu muss für die Ausführung von Konfigurationen, unabhängig vom Urheber des Vorgangs, der logische und funktionale Zustand der Hardware, der insgesamt als Hardware-Zustand bezeichnet wird, mit berücksichtigt werden. Während einer Konfiguration ist die Funktionalität der gesamten bzw. der von der Konfiguration betroffenen Teile einer Komponente nicht definiert. Ist der Hardware-Zustand einer Komponente gültig, kann dieser mit Hilfe der Konfigurationsinformation bei einem Prozesswechsel gesichert werden. Aus Sicht des Betriebssystems ist interessant, ob eine Konfiguration abgeschlossen und somit die betroffene Komponente einsatzbereit ist oder nicht. Problematisch sind demzufolge Situationen, in denen sich Prozesswechsel mit Konfigurationsabläufen überlappen.

5.2.2.1 Konflikte durch Prozesswechsel

Der Prozesswechsel wird von Betriebssystemen in zwei Phasen durchgeführt. Die erste Phase dient der Konservierung des aktuellen Zustands des Rechensystems. Während der zweiten Phase erfolgt die Wiederherstellung des Zustandes für den Folgeprozess [121]. Grundsätzlich gilt diese Betrachtung ebenso für das Betriebssystem des adaptiven Systems: Das Sichern des aktuellen Hardware-Zustands kann, ebenso wie die Wiederherstellung des Hardware-Zustands, logisch den beiden Phasen herkömmlicher Systeme zugeordnet werden. Die Wiederherstellung des Hardware-Zustands unterscheidet sich dennoch wesentlich von der des restlichen Systemzustands. Aus Sicht des Betriebssystems ist ein Prozesswechsel dann abgeschlossen, wenn die gesamte Zustandsinformation des zur Ausführung bereitstehenden Prozesses in die zugehörigen Speicherbereiche und Register kopiert wurde. Nach der Abarbeitung des letzten Maschinenbefehls, der im Zusammenhang mit der Zustandswiederherstellung steht, ist der Vorgang beendet. Bei der Restauration des Hardware-Zustands ist dies nicht zwingend der Fall. Da die Rekonfiguration vom Betriebssystem angestoßen, jedoch in der Hardware-Schicht des System gesteuert und durchgeführt wird, unterscheidet sich der Endzeitpunkt der Rekonfiguration zur Wiederherstellung des Hardware-Zustands im Allgemeinen von dem der Wiederherstellung durch das Betriebssystem.

Dies führt zu einer dreiteiligen Betrachtung des Prozesswechsels für das adaptive System:

1. Retten des alten Systemzustands, inklusive der aktuellen Hardware-Konfigurationsinformation.

2. Initiieren der Wiederherstellung der Hardware-Konfiguration des Folgeprozesses.

3. Wiederherstellen des Systemzustands.

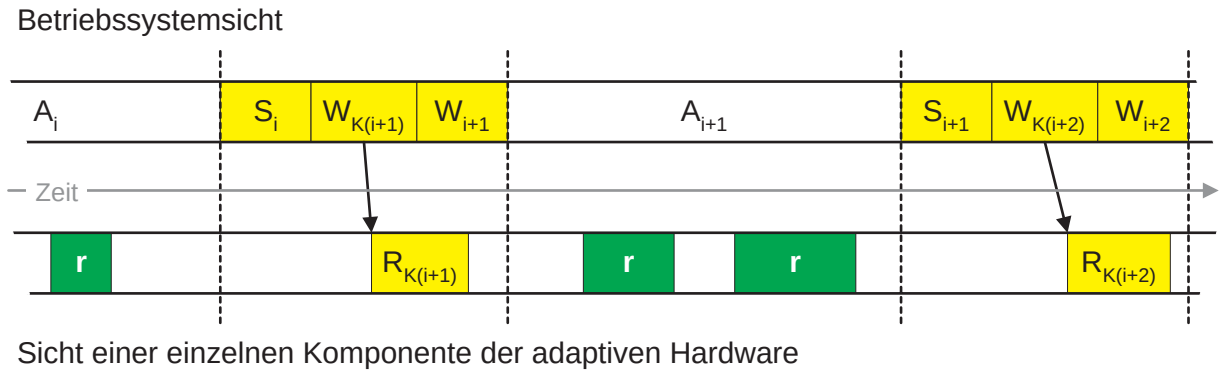

Abbildung 5.10: Idealisierte Betrachtung eines Kontextwechsels

Abbildung 5.10 stellt der Sicht des Betriebssystems (obere Abbildungshälfte) die Sichtweise einer einzelnen adaptiven Systemkomponente gegenüber. Die Darstellung von Rechenzeit und Zeit für einen Prozesswechsel entspricht jedoch nicht der Realität, sondern gibt den qualitativen Verlauf wieder. Man erkennt die Ausführung einer Anwendung A_i, den anschließenden Prozesswechsel zur Anwendung A_{i+1} und deren Ausführung bis zu einem erneuten Prozesswechsel aus Sicht des Betriebssystems. Unterhalb sind die Abläufe des korrespondierenden Zeitraums aus Sicht einer adaptiven Komponente skizziert. Konfigurationsabläufe, die von der jeweils aktuell ausgeführten Anwendung oder der Komponente selbst angestoßen werden, sind mit r markiert. R_K kennzeichnet die vom Betriebssystem zur Systemwiederherstellung angeforderten Rekonfigurationsmaßnahmen.

Für das Betriebssystem läuft jeder Prozesswechsel in den drei Teilen, Sichern des aktuellen Zustands (S), Hardware-Konfigurationen anfordern (W_K) und Wiederherstellen des Systemzustands (W), ab. Im Idealfall, der in Abbildung 5.10 dargestellt ist, sind die erforderlichen Konfigurationsmaßnahmen vor bzw. spätestens am Ende des Prozesswechsels aus Sicht des Betriebssystems abgeschlossen.

Diesbezüglich ergeben sich prinzipiell die beiden in Abbildung 5.11 gezeigten Konfliktsituationen. Im ersten Fall (Teilbild a) nimmt die Wiederherstellung der Konfiguration mehr Zeit in Anspruch als die Ausführung des Prozesswechsels (KW) durch das Betriebssystem. Dem gegenüber steht der zweite Fall (Teilbild b), der auftritt, falls eine Rekonfiguration – durch die laufende Anwendung oder hardware-seitig initiiert – bis zum Beginn des Prozesswechsels nicht abgeschlossen ist.

Für die Konstellation in Teilbild b) müssen zwei Fälle berücksichtigt werden:

1. Die laufende Konfiguration überschneidet sich mit der Zustandssicherung des Systems. Theoretisch ist dieser Vorgang nicht problematisch, wenn die adaptive Komponente auch während eines Konfigurationsvorgangs gültige Werte bei der Abfrage des Zustands liefert. Hierbei ist es irrelevant, ob die letzte abgeschlossene Konfiguration, oder die laufende Konfiguration als Status übermittelt wird. Dies rührt daher, dass eine Anwendung durch die Mechanismen der virtuellen Maschine unabhängig von der aktuell verfügbaren Hardware ausgeführt werden kann. Problematisch wird der Konfigurationszustand erst dann, wenn die Hardware der laufenden Konfiguration erforderlich ist, um vorgegebene Laufzeitkriterien zu erfüllen. Aus diesem Grund bietet sich an,

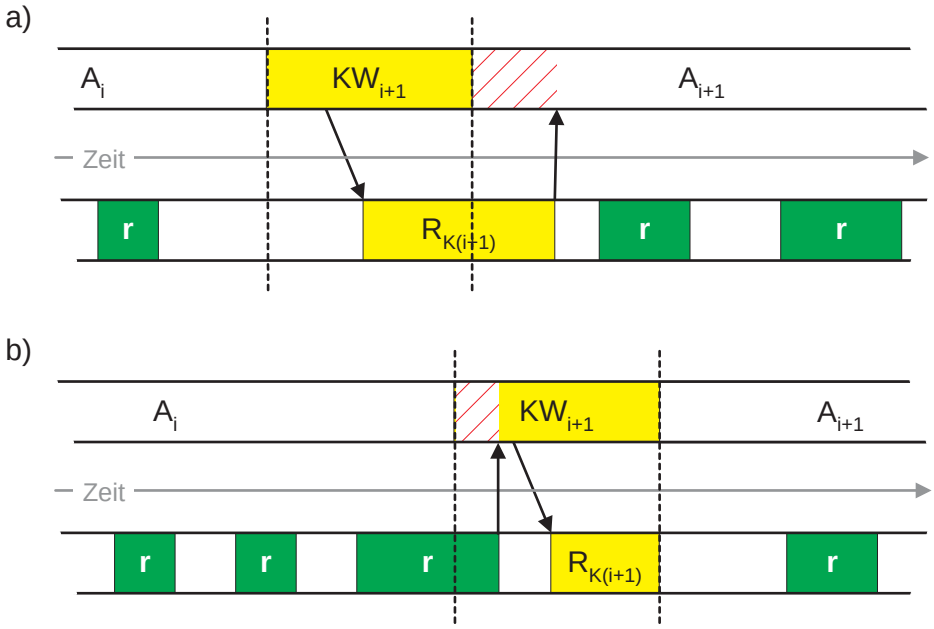

Abbildung 5.11: Probleme im Zusammenhang mit Anwendungswechsel

dass adaptive Komponenten stets den Zustand liefern, den sie nach abgeschlossener Konfiguration hätten. Dies ist möglich, da der Konfigurationszustand nur durch einen Index repräsentiert wird. Die zugehörige Detailinformation kann mittels dieses Index aus der Systembibliothek bezogen werden.

2. Die Wiederherstellungsanforderung des Betriebssystems wird durch die laufende Konfiguration blockiert. Um diese Situation aufzulösen, muss dem Betriebssystem die Möglichkeit gegeben sein, eine laufende Konfiguration zu jedem Zeitpunkt abzubrechen.

Die Behandlung des in Teilbild 5.11 a) aufgezeigten Konfliktfalls kann ohne Hardware-Unterstützung nur auf eine einzige Art stattfinden. Das Betriebssystem wartet auf den Abschluss der Hardware-Konfiguration und stellt anschließend die zugehörige Zustandsinformation der Komponente wieder her. Der Prozesswechsel ist somit immer nach den angeforderten Konfigurationsmaßnahmen beendet. Diese Vorgehensweise kann problemlos in speziellen Fällen eingesetzt werden, in denen alle Konfigurationszeiten im Verhältnis zur Rechenzeit von Anwendungen sehr kurz sind. Allgemein ist dies jedoch nicht gültig und kann zu der in Abbildung 5.12 dargestellten Situation führen. Das heißt, dass entweder die Rechenzeit einer Anwendung durch die Konfigurationszeit massiv eingeschränkt, oder umgekehrt die Zeit für einen Prozesswechsel unangemessen verlängert wird.

Demzufolge eignet sich das Warten bis zum Abschluss der erforderlichen Hardware-Konfiguration nur bedingt als Lösung der Prozesswechselproblematik. Aus Sicht des Betriebssystems kann eine Anwendung, unabhängig vom Status der darunter liegenden adaptiven Komponenten, zur Ausführung freigegeben werden. Somit scheint es möglich, eine Anwendung bis zum ersten Zugriff auf eine nicht fertig konfigurierte Komponente ohne Einschränkung auszuführen. Kann ein solcher Zugriff mit Hilfe der virtuellen Maschine

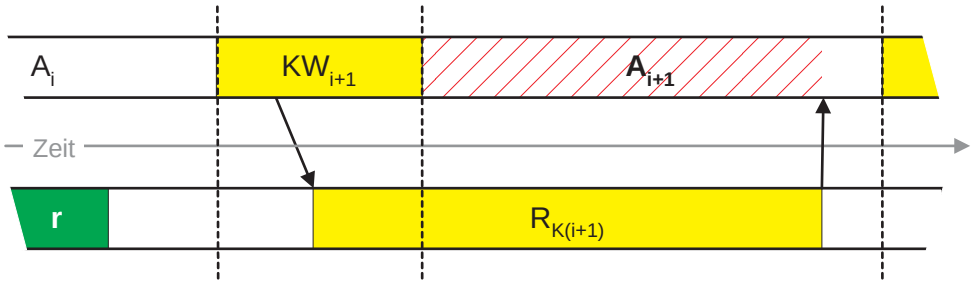

Abbildung 5.12: Unzureichende Rechenzeit einer Anwendung

durch eine Software-Lösung ersetzt werden, würde dies auch die weitere Ausführung erlauben. Ist dies nicht möglich, wird die Anwendung angehalten. Diese Vorgehensweise eignet sich zur (teilweisen) Überdeckung von etwaigen Konfigurationslatenzen, ist aber allgemein nicht zulässig. Das Problem entsteht durch die Zustandsinformation einer Komponente. Während der Konfiguration befindet sich eine Hardware-Komponente in einem nicht definierten Hardware-Zustand. Hat das Betriebssystem zu dieser Zeit versucht, die zugehörige Zustandsinformation an die Komponente weiterzuleiten, sind die Daten verloren, oder nur teilweise richtig abgelegt – in Abhängigkeit vom Fortschritt der Konfiguration.

Grundsätzlich bedeutet das, dass das Betriebssystem die Zustandsinformation von Komponenten mit laufender Konfiguration erst dann an diese weiterleiten darf, wenn die Konfiguration abgeschlossen und ein stabiler Hardware-Zustand erreicht ist. Versucht man dieses Problem mit Hilfe des Betriebssystems zu lösen, hat dies eine ständige Ausführungsunterbrechung der Anwendungen zur Folge. Zudem muss bei jeder Unterbrechung der Systemzustand abgelegt – ohne einen Prozesswechsel durchzuführen – und mit zusätzlichem Aufwand angepasst werden, da sich die gültige Zustandsinformation nicht mehr aus dem aktuellen Zustand ergibt, sondern aus der Kombination mindestens eines älteren Zustands und des aktuellen Zustands.

Eine wesentlich einfacher zu handhabende Möglichkeit bietet sich im Zusammenhang mit einer Hardware-Erweiterung an: Jede adaptive Komponente, deren Zustandsinformation sich nicht ausschließlich auf die aktuelle Konfiguration beschränkt, muss über die Fähigkeit verfügen, die Zustandsinformation während einer laufenden Konfiguration aufzunehmen und diese nach Erreichen eines stabilen Hardware-Zustands selbstständig als aktuellen Zustand zu übernehmen. Dass bedeutet, jede Komponente unterhält einen Puffer, der für diese Aufgabe zu jedem Zeitpunkt bereitsteht. Obwohl diese Erweiterung die Zeitproblematik in Verbindung mit Konfigurationsaufgaben nicht zufriedenstellend löst, bieten sich aus Sicht des Betriebssystems keine weiterführenden Maßnahmen zur Überdeckung von Latenzen, die sich aus Konfigurationsabläufen ergeben, an.

Aus diesem Grund muss für das Betriebssystem des autonomen adaptiven Gesamtsystems ein Scheduling-Verfahren, wie dies z.B. von S. Ghiasi und M. Sarrafzadeh vorgeschlagen wird (Vgl. Grundlagenabschnitt 2.4), angewendet werden, das auf einer Umordnung der Anwendungsreihenfolge basiert. Die Umordnung erfolgt so, dass die Anzahl der notwendigen Konfigurationsmaßnahmen beim Taskwechsel minimiert werden.

5.2.3 Implementierungsaspekte adaptiver Komponenten

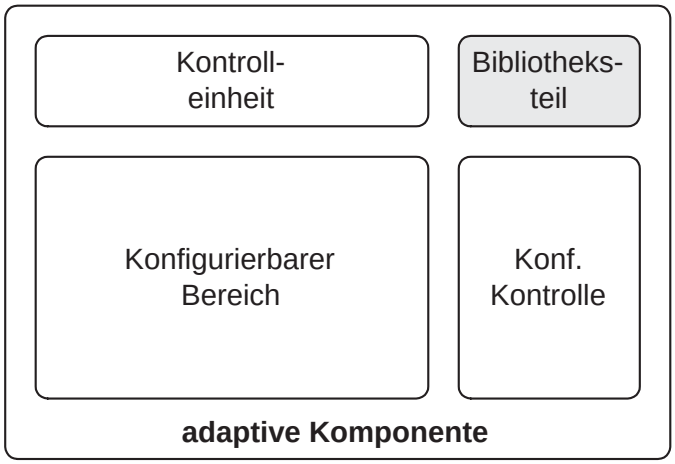

Abbildung 5.13: Logische Bestandteile adaptiver Komponenten

Die (Re-)Konfiguration von Systemkomponenten erfordert verschiedene Maßnahmen, die bei der Implementierung der adaptiven Komponenten beachtet werden müssen. Die notwendigen Mechanismen lassen sich zu den, in Abbildung 5.13 gezeigten, logischen Bestandteilen adaptiver Komponenten zuweisen.

Die *Kontrolleinheit* fungiert als Schnittstelle zu den oberen Systemschichten. Das heißt, dass die Kontrolleinheit zum einen den Hardware-Zustand für die VM, respektive für das Betriebssystem, zugänglich macht und zudem den funktionalen Zustand der Komponente verwaltet. Der *Bibliotheksteil* bildet die Schnittstelle zur Systembibliothek. Speziell die Daten der Systembibliothek, die für interne Abläufe einer Komponente von Bedeutung sind, werden hier zwischengespeichert. Ohne eine Betrachtung der Regelungs- und Überwachungsmechanismen, wie sie in Abschnitt 5.3 diskutiert werden, gilt, dass nur adaptive Recheneinheiten auf die Systembibliothek zugreifen und der Bibliotheksteil demzufolge für die übrigen adaptiven Komponenten entfällt. Die Hardware-Teile, die für eine Konfiguration bereitstehen, sowie die damit verbundenen Verwaltungsaufgaben übernehmen, sind als *konfigurierbarer Bereich* gekennzeichnet. Mechanismen, die das Initiieren und Abarbeiten von Konfigurationsaufgaben betreffen, sind in dem Bestandteil *Konfigurationskontrolle* zusammengefasst. Darüber hinaus stellt die Konfigurationskontrolle die Schnittstelle zum Komponentenpool des Systems bereit.

Eine detailliertere Darstellung der Komponentenbestandteile ist ohne Bezug zu einem vorgegebenen Einsatzgebiet des Systems und den sich daraus ergebenden Zusammenhängen nicht empfehlenswert. Der Aufbau jeder adaptiven Komponente muss zweckdienlich und mit möglichst geringem Ressourcenaufwand realisiert werden. Nur die Kenntnis der realen Rahmenbedingungen erlaubt eine derartige Umsetzung. In diesem Zusammenhang ist z.B. die Größe der Hardware-Module, die für eine einzelne adaptive Komponente zur Verfügung stehen, sowie die Anzahl der gleichzeitig innerhalb einer Komponente präsenten Schaltungen von Bedeutung. Ersteres erlaubt eine adäquate Anpassung der Größe des physisch notwendigen, konfigurierbaren Feldes. Letzteres ist entscheidend für den Aufwand zur Verwaltung von freien und belegten Konfigurationsbereichen. Kann z.B. nur eine einzige Schaltung in

eine Komponente geladen werden, entfallen sämtliche Verwaltungsaufgaben, die den konfigurierbaren Bereich betreffen, da es zu keinerlei Positionierungs- oder Überlappungsproblemen kommen kann.

Unabhängig davon bedarf es einer grundsätzlichen Klärung, wie der Zugriff auf adaptive Komponenten durchgeführt wird und allgemein etwaige Latenzen durch Rekonfigurationsabläufe minimiert werden können.

Das Vorhandensein eines Bibliotheksteils gilt vorerst als Spezialfall für adaptive Recheneinheiten. Deshalb folgt im nächsten Unterabschnitt eine Betrachtung der allgemein gültigen Zugriffsmöglichkeiten auf adaptive Komponenten, die daraufhin um die zusätzlichen, der adaptiven Recheneinheiten, ausgedehnt wird.

5.2.3.1 Zugriff auf adaptive Komponenten

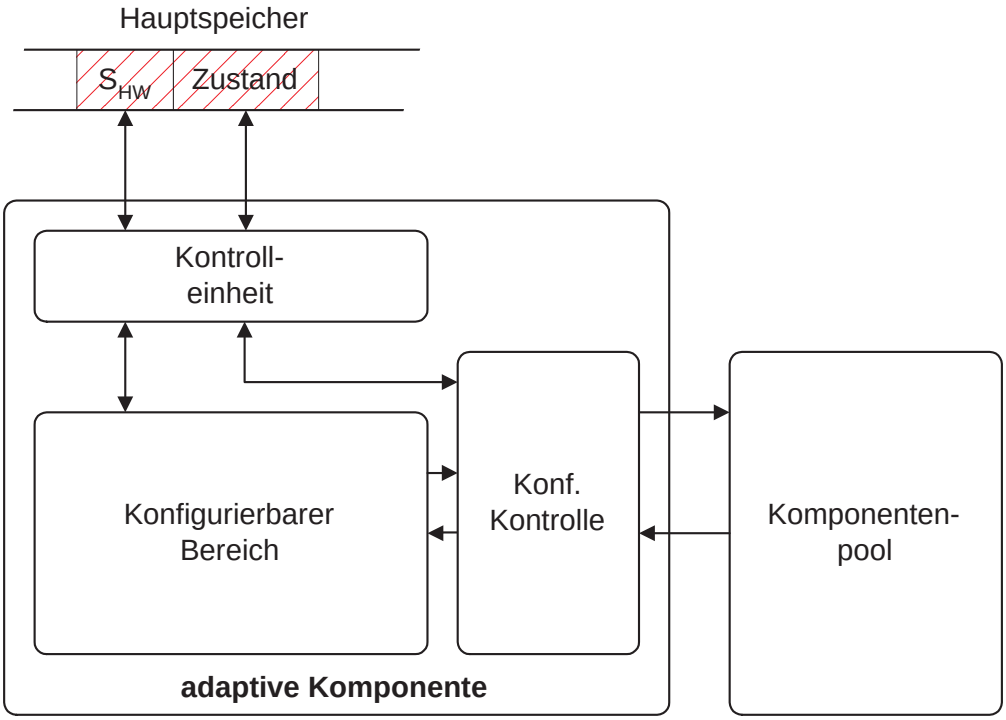

Abbildung 5.14: Realisierung des Zugriffs auf einzelne adaptive Komponenten

Bereits in Abschnitt 5.2.2 wurde im Zusammenhang mit Prozesswechseln auf die Problematik des Hardware-Zustands einer adaptiven Komponente und der benötigten Zustandsinformation hingewiesen. Diesbezüglich erkennt man in Abbildung 5.14, dass beide, sowohl der Hardware-Zustand bzw. aktuelle Hardware-Status (S_{HW}) als auch die übrige Zustandsinformation, mittels der Kontrolleinheit direkt in den Hauptspeicher abgebildet werden (*memory mapped*). Das Sichern und Laden der Zustandsinformation beschränkt sich demzufolge auf das Kopieren von Speicherbereichen und kann durch *Burst*-Zugriffe entsprechend beschleunigt werden. Die Kontrolleinheit muss hierbei stets gewährleisten, dass die Zustandsinformation gültig ist, unabhängig vom aktuellen Hardware-Zustand der Komponente. Wird, z.B. vom Betriebssystem, die Zustandsinformation überschrieben, obwohl der konfigurierbare

Bereich nicht vollständig konfiguriert ist, und sich somit in einem undefiniertem Zustand befindet, puffert die Kontrolleinheit die neue Zustandsinformation und reicht diese nach der abgeschlossenen Konfiguration weiter. Diese Vorgehensweise verbirgt die Konfigurationsvorgänge der adaptiven Komponenten vor oberen Systemschichten. Die Abarbeitung jeder Konfiguration erfolgt in der *Konfigurationskontrolle*. Diese übernimmt z.B. die Verteilung und Positionierung von Schaltungen und überwacht die Kommunikation zwischen Komponentenpool und konfigurierbarem Bereich. Überdies informiert sie die Kontrolleinheit über den Verlauf und Abschluss der Konfiguration(en).

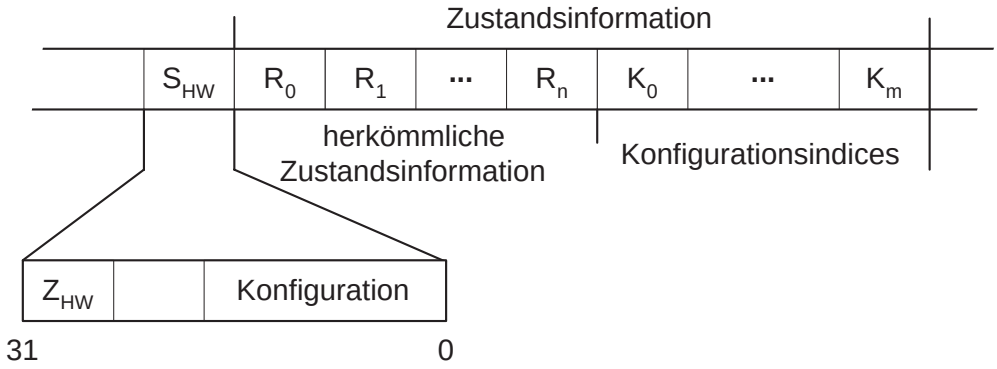

Abbildung 5.15: Prinzipieller Aufbau der Zustandsinformation

Allgemein ist ein Aufbau der Zustandsinformation und des Hardware-Status, wie in Abbildung 5.15 gezeigt, vorgesehen. Der Hardware-Status S_{HW} besteht aus dem aktuellen Hardware-Zustand Z_{HW}, der nur gelesen werden kann, und einem *Konfigurationsteil*. Wird in den Konfigurationsteil ein Konfigurationsindex geschrieben, weist dies die adaptive Komponente an, das entsprechende Modul aus dem Komponentenpool zu laden. Dieser Teil der S_{HW} sollte nicht ausgelesen werden. Im Gegensatz dazu repräsentieren die oberen drei Bits den aktuellen Hardware-Zustand Z_{HW}, der folgende Zustände beschreibt:

- Die Komponente ist **nicht konfiguriert** und alle adaptiven Ressourcen stehen zur Verfügung. Außerdem muss bei einem Prozesswechsel die Konfigurationsinformation nicht gesichert werden.
- Die Komponente ist **funktionsbereit** und enthält gültige Konfigurationsinformation.
- Die Komponente führt eine Konfiguration aus und kann darüber hinaus weitere Konfigurationsanfragen bearbeiten. Dieser Zustand wird als **offen konfigurierend** gekennzeichnet. Wird die Konfigurationsinformation ausgelesen, enthält diese den Zustand, der sich nach vollendeter Konfiguration einstellt.
- Können während der Konfigurationsabläufe keine neuen Anfragen erledigt werden, ist die Komponente als **blockiert konfigurierend** markiert. Das Auslesen der Zustandsinformation wird entsprechend dem Zustand offen konfigurierend behandelt.
- Die Komponente ist **autonom funktionsbereit** das heißt, dass keine Konfigurationsanfragen von Benutzeranwendungen angenommen werden.

Die Zustandsinformation aus Abbildung 5.15 ordnet die herkömmliche Zustandsinformation in Form von Registerinhalten $(R_0, \ldots, R_n)$ vor den zusätzlichen Konfigurationsindices $(K_0, \ldots, K_m)$ an. Die Gültigkeit und Aktualität der enthaltenen Daten unterliegt der Kontrolleinheit. Wird im Zuge eines Prozesswechsels die Zustandsinformation überschrieben, wird zuerst auf notwendige Konfigurationsmaßnahmen geprüft. Sind keine Änderungen erforderlich, leitet die Kontrolleinheit die Registerinhalte an die entsprechenden Teile des konfigurierbaren Bereichs weiter. Ansonsten erfolgt eine Konfiguration in Abhängigkeit von den Konfigurationsindices. Abschließend werden die Registerinhalte zugewiesen.

Demzufolge stehen zwei unabhängige Mechanismen zur Rekonfiguration einer adaptiven Komponente bereit. Zum einen kann über das Hardware-Statusregister und zum anderen über die Konfigurationsindices der Zustandsinformation eine Konfiguration initiiert werden. Letztere Methode ist jedoch dem Betriebssystem vorbehalten und sollte nur in Ausnahmefällen direkt von Benutzeranwendungen durchgeführt werden.

5.2.3.2 Erweiterter Zugriff auf adaptive Recheneinheiten

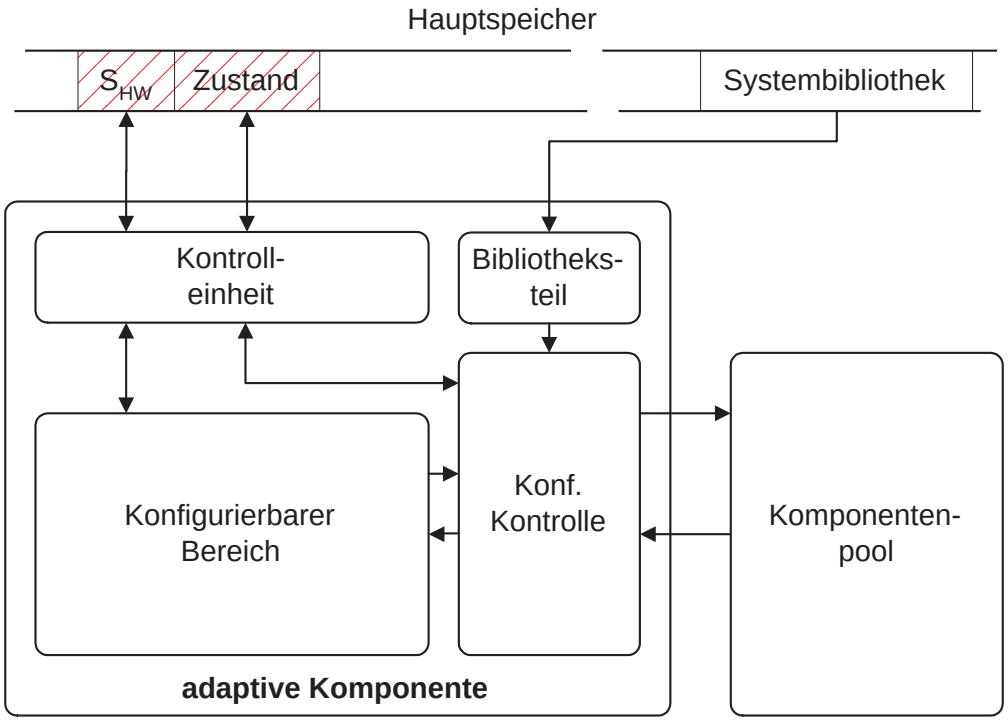

Abbildung 5.16: Der Zugriff auf adaptive Recheneinheiten

Betrachtet man die Zugriffsmöglichkeiten der adaptiven Recheneinheiten, ergibt sich das in Abbildung 5.16 gezeigte Schema. Zu den bereits behandelten Möglichkeiten muss bei den adaptiven Recheneinheiten zusätzlich die Systembibliothek in die internen Konfigurationsabläufe mit einbezogen werden. Prinzipiell führt dies dazu, dass sowohl der *Bibliotheksteil* als auch die *Kontrolleinheit* auf die *Konfigurationskontrolle* zugreifen. Wobei die Kontrolleinheit Konfigurationsanforderungen der oberen Systemschichten bedient und der Bibliotheksteil eigenständig Konfigurationen anfordern kann. Die Beschreibung der Abläufe innerhalb

des Bibliotheksteils werden in Kapitel 6, im Zusammenhang mit dem adaptiven Prozessor, eingehend erläutert. Der Zugriff auf die Systembibliothek enthält jedoch eine grundsätzliche Problematik, die in Verbindung mit Prozesswechseln steht.

Für jeden Prozess ist eine eigenständige lokale Bibliotheksinformation zulässig. Demzufolge erfordert jeder Prozesswechsel, dass alle adaptiven Recheneinheiten mit der aktuellen lokalen Bibliotheksinformation versorgt werden. Um dies in adäquater Zeit zu bewältigen, bietet sich eine direkte Abbildung des Speicherbereichs der lokalen Bibliothek in entsprechende Hardware-Register des Bibliotheksteils an. Im Gegensatz zum Hardware-Status und der Zustandsinformation ist die lokale Bibliotheksinformation unidirektional, d.h. die Bibliotheksdaten werden von den adaptiven Komponenten nur gelesen und nicht geändert. Deshalb ist eine $1 : n$ Verteilung von einem einzigen lokalen Bibliotheksspeicherbereich auf n unabhängige adaptive Recheneinheiten zulässig. Diese Umsetzung erlaubt einen Prozesswechsel mit minimalem Aufwand in Bezug auf die Datenaktualisierung des lokalen Bibliotheksteils, unabhängig von der Anzahl der physisch verfügbaren Einheiten.

Im Gegensatz dazu sind die Daten des globalen Bibliotheksteils im Allgemeinen nicht zeitkritisch. Diese können während der Initialisierung des Systems in die Komponenten transferiert werden und bleiben zur gesamten Laufzeit unverändert. Als Ausnahme ist anzumerken, dass für gewisse Einsatzgebiete die Initialisierung innerhalb eines engen Zeitrahmens erfolgen muss. In diesem Zusammenhang ist eine *memory-mapped* Umsetzung, entsprechend dem lokalen Teil, in Erwägung zu ziehen.

5.2.4 Infrastruktur für die Konfiguration

Das Hauptproblem beim Einsatz rekonfigurierbarer Logik ist die lange Dauer dynamischer Rekonfigurationsvorgänge [2, 157, 30, 48, 49]. Yamashina et al. [157] lösten diese Problematik mit der Einführung eines eigens entworfenen Multikontext-FPGAs, der in der Lage ist, bis zu acht voneinander unabhängige Schaltungen gleichzeitig in unterschiedliche Kontexte des FPGAs zu laden. Das Umschalten von einer Ebene (FPGA-Kontext) zu einer anderen wird ungeachtet der Größe der Schaltung mit 4,6 ns angegeben. Hierbei müssen die verfügbaren Schaltungen bereits vollständig in den FPGA geladen sein, bevor ein Umschalten zwischen den Ebenen möglich ist. Um die Ladezeiten zu verbessern wurden im Garp-Projekt ein hierarchischer Aufbau sowie eine Trennung vom Speicherbus des Systems vorgeschlagen (siehe auch Abschnitt 4.4.2). Die sich daraus ergebende Struktur ist vergleichbar mit einer einstufigen Speicherhierarchie für Hauptspeicherzugriffe und eignet sich grundsätzlich für den Einsatz im adaptiven System.

Dieser Ansatz ist schematisch in Abbildung 5.17 dargestellt. Der Konfigurationsspeicher des Komponentenpools bietet im Gegensatz zum Hauptspeicher eines Systems eine vereinfachte Zugriffsstruktur. Da Schaltungen in Form von Bitstreams sequentiell angeordnet sind und zudem nur in dieser Reihenfolge ausgelesen werden, ist der Zugriffsverlauf immer vorhersehbar und mit einfachen *Prefetching*-Mechanismen effizient realisierbar. Zudem wird, aus Sicht der adaptiven Komponenten, nur lesend auf die Daten des Komponentenpools zugegriffen, wodurch prinzipiell keine Dateninkonsistenz in eingesetzten Puffern auftreten kann. Dennoch gilt zu beachten, dass für Prozesswechsel die lokalen Bibliothekseinträge geändert werden und somit ein Invalidierungsmechanismus für etwaige korrespondierende Pufferdaten existieren muss. Die temporär gültigen lokalen Bibliotheksdaten reduzieren implizit die Größe der Module, die lokal abgelegt werden können, da diese stets in relativ kurzer Zeit,

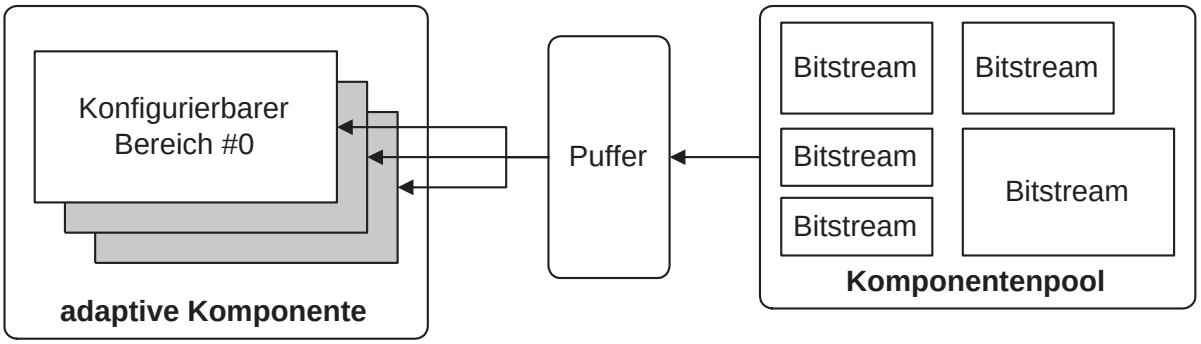

Abbildung 5.17: Speicherhierarchie für Konfigurationsvorgänge

im Verhältnis zur Rechenzeit der zugehörigen Anwendung, konfigurierbar sein müssen. Um diese Beschränkung zu umgehen, können z.B. mehrere Ebenen für einen Puffer eingeführt oder spezielle Bereiche ausgewiesen werden, die eine dauerhafte Zwischenspeicherung von größeren Schaltungen erlauben.

Man erkennt in Abbildung 5.17 eine adaptive Komponente, deren *konfigurierbarer Bereich* in mehrere Ebenen unterteilt ist. Die einzelnen Ebenen sind über einen *Puffer* mit dem *Komponentenpool* verbunden. Konfigurationsvorgänge, die durch das Umschalten der Ebenen innerhalb einer adaptiven Komponente erledigt werden können, benötigen wenige Nanosekunden. Das Nachladen eines Moduls aus dem Puffer kann in einem Zeitrahmen von wenigen 100 Taktzyklen erfolgen. Für Schaltungen, die direkt aus dem Komponentenpool angefordert werden, ergeben sich Konfigurationszeiten, die massiv von der Größe der Schaltung abhängen und bis zu einigen Millisekunden dauern (siehe auch [30, 68, 154, 157]). Letzteres bedeutet, dass trotz einer verbesserten Infrastruktur nicht jede Konfigurationsanforderung befriedigend abgearbeitet werden kann. Aus diesem Grund muss die Adaptivität des Systems, wie in den vorangegangenen Abschnitten ausgeführt, bereits in der Betriebssystemschicht und zur Übersetzungszeit mit berücksichtigt werden.

Darüber hinaus ergeben sich zusätzliche Erweiterungen, die beim Entwurf der Konfigurationsinfrastruktur für das adaptive System eine Rolle spielen. Der Einsatz mehrerer unabhängiger adaptiver Komponenten erhöht zusätzlich die Problematik der Konfigurationszeiten, indem sich gleichzeitig auftretende Konfigurationsanforderungen gegenseitig behindern können.

Um Zugriffskonflikte zu vermeiden bzw. abzuschwächen, empfiehlt es sich die Konfigurationsinfrastruktur weiter auszubauen. Zwei grundsätzliche Möglichkeiten sind in Abbildung 5.18 skizziert. Teilbild a) zeigt diesbezüglich die Aufteilung des Puffers in zwei kleinere Puffer, die jeweils nur einer adaptiven Komponente zugeordnet sind. Des Weiteren kann der Komponentenpool in zwei (Teilbild b) oder mehrere physikalisch getrennte Pools zerlegt werden. Diese Möglichkeit bietet sich an, falls Schaltungen nur jeweils für eine Gruppe von adaptiven Komponenten nutzbar sind.

Insgesamt stehen für die Konfigurationsinfrastruktur die gleichen Mechanismen zur Verfügung, die bei Hauptspeichern eingesetzt werden. Dementsprechend ergeben sich nicht nur die in Abbildung 5.18 gezeigten Realisierungsmöglichkeiten, sondern sind darüber hinaus weitere Abstufungen der Hierarchie sowie Mischformen aus den skizzierten Vorschlägen möglich. Eine allgemein gültige Vorschrift für die Umsetzung der Konfigurationsinfrastruk-

tur wird nicht angegeben, da diese hauptsächlich von den Ressourcen und Rahmenbedingungen, wie z.B. Formfaktor, Chipfläche, Stromverbrauch usw., des Einsatzgebiets abhängt.

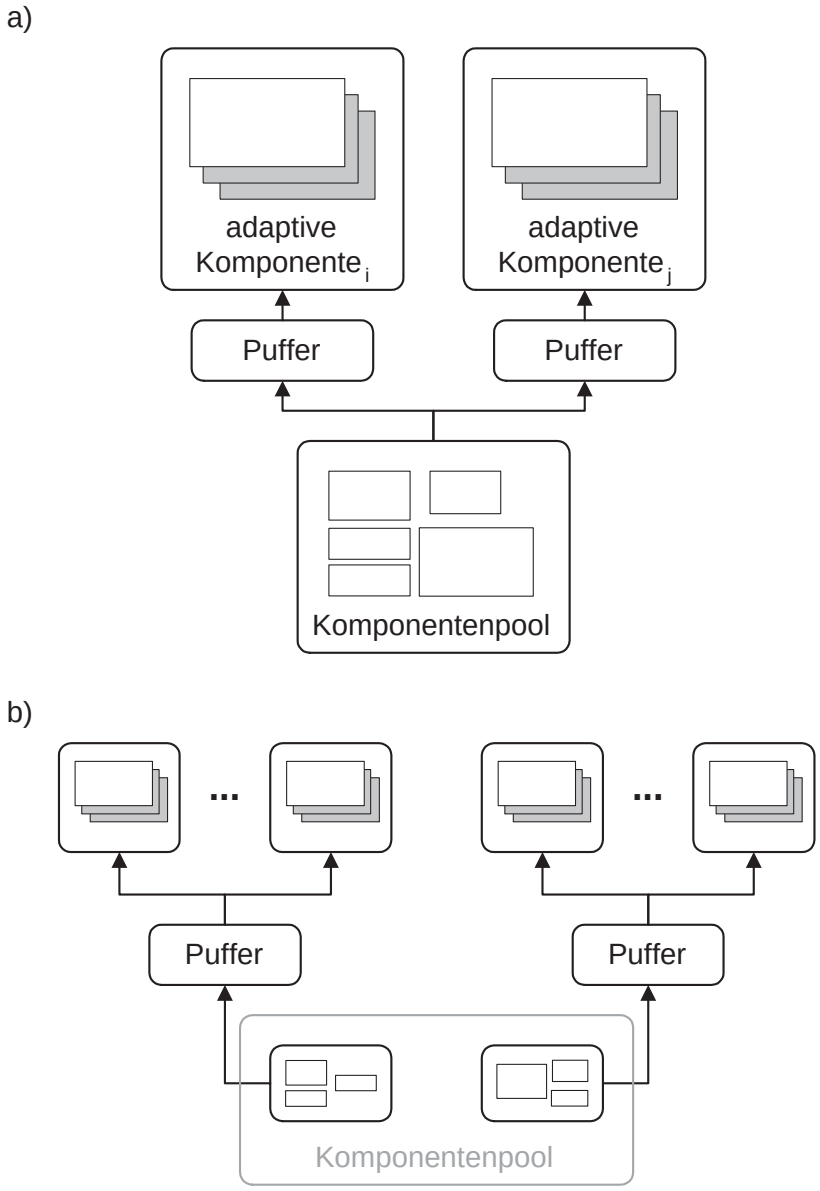

Abbildung 5.18: Konfliktvermeidung bei Konfigurationsvorgängen

5.3 Überwachung und Regelung im System

Dieser Abschnitt dient als Einführung in die Überwachung und Regelung innerhalb des vorgestellten autonomen adaptiven Systems. Hierbei sind der Energiebedarf und die Rechenleistung des Gesamtsystems von vorrangiger Bedeutung. Eine Vielzahl von Arbeiten haben sich bis dato mit diesem Themenbereich auseinandergesetzt (Überblick siehe Grundlagenabschnitt 2.4) und verschiedenste Mechanismen hervorgebracht, die zum Teil in das Konzept des adaptiven Gesamtsystems einfließen.

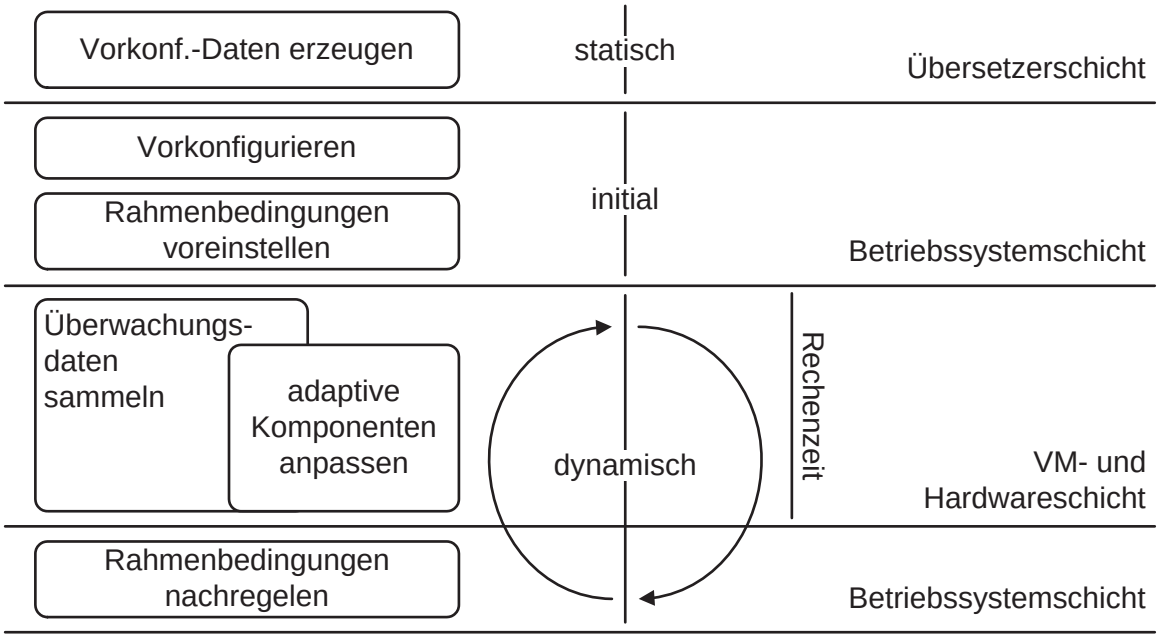

Abbildung 5.19: Systemschichten und zugehörige Regelungsabläufe

Abbildung 5.19 zeigt die Regelungsabläufe der bereits vorgestellten Systemschichten. Die Darstellung in Abbildung 5.19 beschränkt sich ausschließlich auf die Vorgänge, die in unmittelbarem Zusammenhang mit der Überwachung und Regelung stehen und die Grundvoraussetzungen für das autonome, adaptive System bilden.

Obwohl die Übersetzerschicht nicht aktiv an der dynamischen Regelung beteiligt ist, müssen bereits hier verschiedene Mechanismen zur Regelungsunterstützung ansetzen. Allen voran das Einfügen der Bibliotheks-Instruktionen, die eine weiterführende Regelung überhaupt erst ermöglichen (Details siehe Abschnitt 5.2). Maßgeblich für die dynamische Regelung ist die Betriebssystemschicht. Das Betriebssystem ist verantwortlich für die Festlegung der zur Ausführungszeit geltenden Rahmenbedingungen. Hierunter fallen z.B. die Einstellung von Frequenz und Spannung, oder das An- und Abschalten von Systemteilen. Vor der eigentlichen Anwendungsausführung kann diese Regelung nur auf Basis von statischen Information des Übersetzers (Abbildung 5.19) oder auf Grund von Benutzervorgaben getätigt werden. Entsprechend dieser Information erfolgt eine Vorkonfiguration der adaptiven Systemkomponenten und die Einstellung der Rahmenbedingungen.

Mit Beginn der Anwendungsausführung tritt die Regelung in die dynamische Phase ein und startet die Überwachungsmechanismen der Hardware-Schicht, die die Grundlagen wei-

terer Regelung darstellen. Durch entsprechende Überwachungs-Hardware wird anhand der Nutzungshäufigkeit und des Nutzungsverhaltens von Systemkomponenten eine effektive Regelung, die sowohl den Energiebedarf als auch eine adäquate Rechenleistung des Gesamtsystems berücksichtigt, implementiert.

In diesem Zusammenhang muss die VM-Schicht die Überwachungs-Hardware unterstützen. Dies bedeutet in erster Linie, dass Information über das Nutzungsverhalten von Modulen an die Hardware geleitet wird. Notwendig ist diese Vorgehensweise, da die VM etwaige Bibliotheksinstruktionen, die auf Grund einer vorliegenden Systemkonfiguration nicht durch die zugehörigen Hardware-Module beschleunigt werden können, durch die korrespondierende Software-Lösung ersetzt. Ist dies der Fall, existiert aus Sicht der Hardware keine Möglichkeit die evtl. sinnvolle Nutzung dieses Moduls zu erkennen und entsprechend in eine Regelung mit einzubeziehen. Demzufolge muss die VM die darunter liegende Überwachungs-Hardware auf eine solche Ersetzung hinweisen.

Die für software-basierte Regelungen in der Zwischenzeit häufig eingesetzten Tracking-Verfahren eignen sich ohne weitreichende Änderung für die Verwendung in der Betriebssystemschicht des vorgestellten adaptiven Systems. Zu berücksichtigen ist, dass die Verfahren meist auf Information aus den einzelnen Hardware-Komponenten angewiesen sind. Zudem müssen die physikalischen Komponenten Mechanismen bereitstellen, die eine Regelung zulassen. Zum Beispiel sind die Mechanismen für Intels *Speed-Stepping* oder die diversen *Sleep-Modi* in den Prozessoren physikalisch vorhanden (siehe Abschnitt 4.3.1). Die Einstellung muss jedoch vom Betriebssystem kontrolliert werden, da eine Reihe von Voraussetzungen erfüllt sein muss, z.B. keine Aktivität auf dem Systembus, die dem Prozessor selbst nicht bekannt ist [79, 84, 77].

Prinzipiell unterscheidet sich die Regelung des adaptiven Systems in den oberen Schichten – vor allem der Betriebssystemschicht – nicht von der aktuell eingesetzter Systeme (Vgl. Kapitel 3 und 4). Eine Ausnahme bildet die Aufgabe der Hardware-Schicht. Die Bereitstellung von Überwachungsinformation zur Auswertung in oberen Systemschichten bleibt bestehen. Darüber hinaus bietet sich aus dem Einsatz rekonfigurierbarer Hardware eine weitere Möglichkeit an, diese adaptiven Komponenten durch entsprechende Überwachung und hardware-basierte Regelung effizient zu nutzen.

Bisher ist der Einsatz rekonfigurierbarer Bauteile darauf beschränkt, dass für die Ausführung einer Anwendung die zugehörigen HW-Module zum Ausführungsbeginn eingelagert werden. Diese Vorgehensweise führt zu Beschleunigungsfaktoren von ca. zwei bis hin zu 100 und mehr [30, 154, 87, 68]. In Zusammenhang mit dem adaptiven Gesamtsystem wird gezeigt, dass durch den Einsatz von Überwachungs- und entsprechender Regelungs-Hardware die Nutzung rekonfigurierbarer Felder dynamisch optimiert werden kann. Dies erlaubt eine Verringerung des Umfangs notwendiger rekonfigurierbarer Ressourcen bei vergleichbaren Beschleunigungsfaktoren.

5.3.1 Passive und aktive adaptive Systemkomponenten

Das adaptive Gesamtsystem setzt sich, wie bestehende Systeme, aus unterschiedlichen physikalischen Systemkomponenten zusammen. Diejenigen Systemkomponenten, die über rekonfigurierbare Bereiche verfügen, werden, wie in Abschnitt 5.2 eingeführt, als adaptive Komponenten bezeichnet. Die Anzahl, Größe, Art der Integration etc. der rekonfigurier-

baren Bereiche hängt von der jeweiligen Komponente ab und ist im Allgemeinen für die Überwachung und Regelung nicht von Bedeutung. Ausschlaggebend ist die Nutzung und Nutzbarmachung dieser Bereiche durch den möglichen Einsatz verschiedener verfügbarer Hardware-Module. Es ist entscheidend, in welchen Zeitintervallen der Austausch bzw. die Nutzung von Hardware-Modulen erfolgen soll. Die Länge der zu betrachtenden Zeitintervalle ist die Maßgabe für die Überwachung, Auswertung und die maximal zulässige Reaktionszeit einer Regelung (Vgl. [122]).

Das Betriebssystem des adaptiven Gesamtsystems ist grundsätzlich als Mehrprozesssystem vorgesehen. Dementsprechend werden Anwendungen in Zeitscheiben abgearbeitet. Üblicherweise entspricht eine Zeitscheibe einem Zeitraum von wenigen Millisekunden. Die Auslegung als Mehrprozesssystem mit verhältnismäßig kurzen Zeitscheiben wurde festgelegt, um zu klären, ob sich das Konzept *nicht* nur für spezielle Anwendungen im Bereich der eingebetteten Systeme eignet sondern darüber hinaus in PC-Systemen oder Systemen des Hoch- und Höchstleistungsrechnens eingesetzt werden kann.

Für die Regelung des Gesamtsystems bedeutet dies, dass Vorgänge, die über eine oder mehrere Zeitscheiben andauern, vom Betriebssystem – oder allgemein oberen Systemschichten – überwacht und mittels bestehenden Software-Verfahren gesteuert werden können. Die Wahl einer geeigneten Zeitscheibenlänge muss dennoch in Relation zu den eingesetzten Modulen bzw. deren Rekonfigurationszeiten erfolgen. Module mit Rekonfigurationszeiten die ein vielfaches einer Zeitscheibe ausmachen, erfordern eine weitreichende Vorausplanung des Moduleinsatzes. Dies ist häufig nur bedingt möglich oder sogar unmöglich. Zudem wird dadurch die Flexibilität des dynamischen Moduleinsatzes stark beschränkt. In solchen Fällen ist entweder eine Verlängerung der Zeitscheiben oder eine Vereinfachung oder funktionale Beschränkung der Module – was in der Regel zu einer Verkleinerung eines Moduls und somit zu einer Verkürzung der Rekonfigurationszeiten führt – in Erwägung zu ziehen.

Feingranulare Abläufe, mit kürzeren Zeitintervallen als der Länge einer Zeitscheibe, entziehen sich der Überwachung und Kontrolle durch obere Systemschichten. Allgemein ist der Einsatz von software-basierten Lösungen für die Regelung innerhalb der Hardware-Schicht des adaptiven Systems abzulehnen. Diesbezüglich geeignete Software-Lösungen fallen unter die Kategorie der Tracking-Verfahren. Um mittels dieser Ansätze Regelungsberechnungen in wenigen Mikrosekunden – oder darunter – durchführen zu können, wird entweder eine sehr umfangreiche Speziallogik benötigt oder es müssen eigenständige Mikrocontroller oder Prozessoren eingesetzt werden. Gleiches gilt auch für Fuzzy-Regler oder Neuro-Fuzzy-Regler (siehe Kapitel 3).

Im Zusammenhang mit dem adaptiven Gesamtsystem würde dies bedeuten, dass jede geeignete adaptive Komponente einen eigenständigen Regelungs-Controller bzw. -Prozessor benötigt. Diese Vorgehensweise widerspricht prinzipiell den Anforderungen, die im Bereich eingebetteter und mobiler Systeme gelten (Vgl. Grundlagenabschnitt 2.3). Demzufolge erfordern Vorgänge mit kurzen Zeitintervallen einfache Überwachungs- und Regelungs-Mechanismen, die mittels logischer Schaltungen in adaptive Komponenten integriert werden können. Außerdem ergeben sich in Abhängigkeit der zu betrachtenden Vorgänge und der diesbezüglichen Regelungsmöglichkeiten zwei grundsätzliche Kategorien von adaptiven Komponenten:

1. **Passive adaptive Komponenten**, die vollständig einer Steuerung durch höhere Systemschichten unterliegen.

2. **Aktive adaptive Komponenten**, die über Überwachungs- und Regelungs-Hardware verfügen und innerhalb bestimmter Rahmenbedingungen eine eigenständige Verwaltung ihrer rekonfigurierbaren Ressourcen aufweisen.

5.3.2 Wirkungskreisläufe adaptiver Komponenten

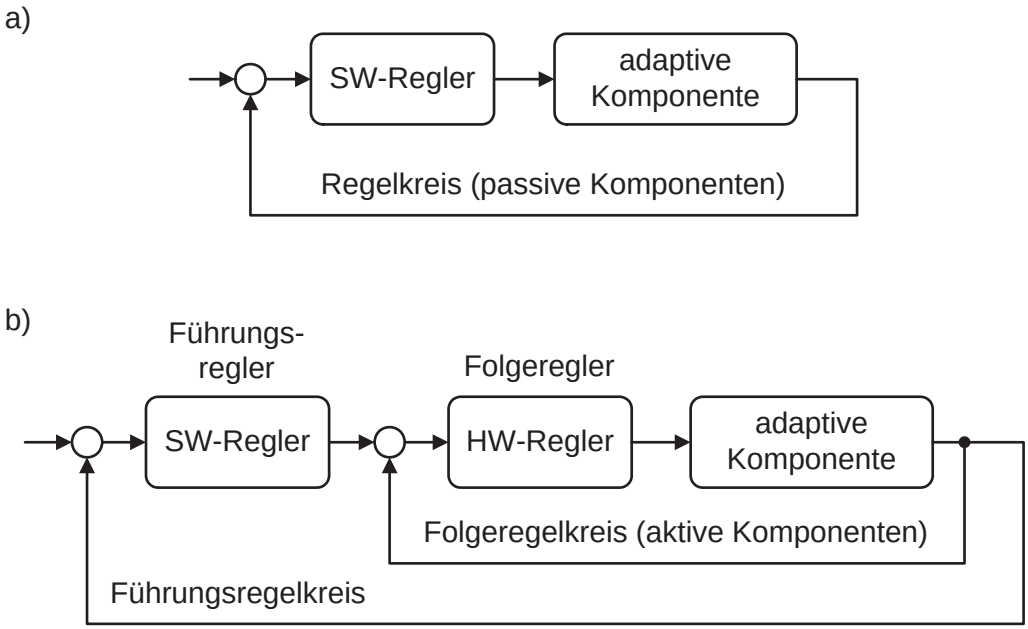

Abbildung 5.20: Wirkungskreisläufe im Gesamtsystem

Entsprechend den beiden Kategorien, passive und aktive adaptive Komponenten, sind zwei Wirkungskreisläufe im adaptiven Gesamtsystem anzutreffen (Abbildung 5.20).

Teilbild 5.20 a zeigt den Wirkungskreislauf passiver adaptiver Komponenten: Die Messdaten werden in der Komponente (Regelstrecke) gewonnen und an den Regler übermittelt. Die Regelung erfolgt außerhalb der Komponente im Betriebssystem. Bei der Verwendung aktiver adaptiver Komponenten (Teilbild 5.20 b) bilden diese eine Teilstrecke mit eigenständiger Regelung[3].

Die Regelung passiver adaptiver Komponenten beschränkt sich auf Rekonfigurationen während eines Prozesswechsels. Hierzu existieren Vorschläge, die die Rekonfigurationszeiten verringern [44, 68] oder bei denen durch entsprechendes Prozess-Scheduling der Rekonfigurationsaufwand minimiert wird [55]. Im Weiteren werden ausschließlich aktive adaptive Komponenten und deren Überwachung und interne Regelung betrachtet.

[3]Die dargestellten Wirkungskreisläufe können auf alle Systemkomponenten übertragen werden. Welcher Wirkungskreislauf zutreffend ist, hängt von der Regelungsfähigkeit der jeweiligen Komponente ab.

5.3.3 Regelungsziel aktiver adaptiver Komponenten

Der Einsatz von rekonfigurierbarer Logik in Rechensystemen oder einzelnen Systemkomponenten ist kein neuartiger Ansatz. Bisher beschränkt sich die Nutzung der rekonfigurierbaren Bereiche auf die gesamte Laufzeit einer Anwendung. Das bedeutet, dass im Allgemeinen zur Übersetzungszeit einer Anwendung eine Menge möglicher Hardware-Module ermittelt bzw. erzeugt wird. Diese werden zu Beginn der Anwendung in die verfügbaren rekonfigurierbaren Bereiche geladen. Werden mehrere Anwendungen in einem Zeitscheibenverfahren abgearbeitet, erfolgt eine entsprechende Rekonfiguration zu Beginn jeder Zeitscheibe. Das Laufzeitverhalten einer Anwendung wird nicht beachtet. Das bedeutet, dass unabhängig von der Nutzung eines Hardware-Moduls, dies zu jedem Zeitpunkt für eine Anwendung bereitsteht. In diesem Kontext problematisch ist der hohe Ressourcenaufwand der nötig ist, um alle Hardware-Module aufzunehmen. Diesbezüglich gibt es verschiedene Ansätze in der Literatur, die alle eine feste Begrenzung der möglichen Hardware-Module zugrunde legen [154, 44, 30, 68, 157, 49]. Dies führt dazu, dass entweder die Anzahl der nutzbaren Hardware-Module stark beschränkt, oder ein sehr hoher Ressourcenaufwand notwendig ist, um mehrere Module gleichzeitig bereitstellen zu können (siehe auch Abschnitt 4.4).

In Bezug auf das adaptive Gesamtsystem werden diese Nachteile durch eine dynamische Regelung innerhalb aktiver Komponenten stark reduziert. Hierbei steht ein möglichst geringer Bedarf an rekonfigurierbaren Ressourcen mit einer bestmöglichen Ausnutzung zur Laufzeit einer Anwendung im Vordergrund. Ziel ist es, den Einsatz rekonfigurierbarer Hardware in realen Systemen attraktiv zu machen.

Bestimmte Konfigurationsvorgänge sind dennoch dem Betriebssystem oder allgemein oberen Systemschichten vorbehalten. Zum Beispiel bei Prozesswechseln ist es sinnvoll, den vorhergehenden Konfigurationszustand wieder herzustellen, ohne diesen erneut durch die Hardware-Regelung dynamisch zu ermitteln. Des Weiteren kann der Einsatz von Hardware-Modulen an Laufzeitkriterien, z.B. zur Einhaltung von Echtzeitanforderungen, gekoppelt sein. Solche Vorgaben sind einer aktiven adaptiven Komponente im Allgemeinen nicht bekannt. Diese zählen zu den Rahmenbedingungen, die von oberen Systemschichten vorgegeben sind, und innerhalb derer die Hardware-Regelung arbeiten muss. Die diesbezüglichen Aufgaben und Problemstellungen der oberen Systemschichten wurden detailliert in Abschnitt 5.2 aufgeführt.

5.3.4 Anwendungsverhalten und Regelungsansätze

Der Einsatz von Hardware-Modulen in einer adaptiven Komponente erlaubt die Optimierung dieser Komponente unter Berücksichtigung des Anwendungsverhaltens. Hierbei ist die Nutzungshäufigkeit eines Moduls von vorrangiger Bedeutung. Je häufiger eine Optimierung zum Einsatz kommt, desto größer ist der positive Gesamteffekt.

Abbildung 5.21 stellt qualitativ die Ausführung einer Anwendung ohne den Einsatz von Hardware-Modulen (oben) der vollständig optimierten Ausführung (unten) gegenüber. Der angegebene Zeitraum $T_{Anwendung}$ entspricht einer einzelnen Zeitscheibe. Betrachtet man die gesamte Rechenzeit einer Anwendung zeigt sich ein zu Abbildung 5.21 äquivalentes Verhalten. Für die Regelung innerhalb aktiver adaptiver Komponenten ist jedoch das Awendungsverhalten innerhalb einer einzelnen Zeitscheibe ausschlaggebend.

Aus Sicht der adaptiven Komponenten besteht eine Anwendung aus einer Reihe nicht-optimierbarer Teilstücke c, die paarweise einen optimierbaren Bereich umschließen (Abbildung 5.21). Jeder Zeitpunkt, der den Eintritt in einen optimierbaren Bereich markiert, ist als Optimierungspunkt P gekennzeichnet. In Abbildung 5.21 stellen sich Bereiche die optimierbar sind, jedoch nicht-optimiert ausgeführt werden, mit $T_{\setminus M}$ dar. Die optimierte Ausführung wird durch T_M repräsentiert. Die gewählte Beschriftung ist hierbei an die Reihenfolge der Optimierungspunkte, nicht an die zugrunde liegenden Hardware-Module, angelehnt. Entsprechend zeigt ein Optimierungspunkt Px in Abbildung 5.21 jeweils einen nicht-optimierten Bereich $T_{\setminus M(Px)}$ sowie den korrespondierenden, optimierten Bereich $T_{M(Px)}$ an. Ein beliebiger Optimierungspunkt Px lässt in der vorliegenden Darstellung keine Rückschlüsse auf das verwendbare Modul zu. D.h. für Abbildung 5.21 sind die Module, die sich für die jeweiligen optimierbaren Bereiche eignen, so gewählt, dass $M(P0) \neq M(P1) \neq M(P[i-1])$ gilt.

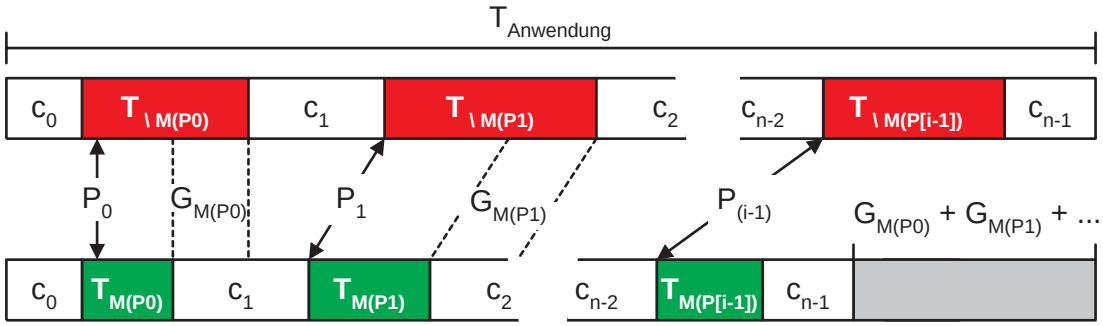

Abbildung 5.21: Ursprüngliche und vollständig optimierte Anwendungslaufzeit

Allgemein weist eine Anwendung, während des Zeitraums $T_{Anwendung}$, i Optimierungsmöglichkeiten, mit $i \in \mathbb{N}_0$, auf. Jeder Optimierungsmöglichkeit lässt sich eindeutig ein Modul X aus einer Menge von m einsetzbaren Modulen zuweisen. Ist ein Modul beim Auftreten eines Optimierungspunktes Px verfügbar, wird die hierbei entstehende Zeitersparnis mit $G_{M(Px)}$ angegeben. Zudem sind n, mit $n = i + 1$, nicht-optimierbare Bereiche c anzutreffen.

Die Gegenüberstellung der optimierten und nicht-optimierten Ausführung in Abbildung 5.21 ist nur dann korrekt, wenn jeweils identische Ausgangs- bzw. Rahmenbedingungen gelten. Das heißt, dass alle Faktoren, die das zeitliche Verhalten in der Realität mit beeinflussen können, wie z.B. Speicherzugriffe, variierende Zugriffszeiten in Bezug auf Aus-/Eingabegeräte oder Unterbrechungen und Ausnahmebehandlungen, ignoriert bzw. idealisiert betrachtet werden. Somit gilt stets, dass jedes Zeitintervall c_i der nicht-optimierten Ausführung exakt dem korrespondierenden c_i der optimierten Ausführung entspricht. Darüber hinaus sind demzufolge alle Bereiche c_i in einem betrachteten Zeitraum $T_{Anwendung}$ als konstant anzusehen. Für die Regelung der adaptiven Komponenten ist die Vereinfachung zulässig, da diese Bereiche weder beeinflusst werden können noch zur Entscheidungsfindung beitragen.

Die gesamte Ausführungszeit (innerhalb einer Zeitscheibe) ohne Optimierung lässt sich folgendermaßen beschreiben:

$$T_{Anwendung} = c_0 + T_{\setminus M(P0)} + c_1 + T_{\setminus M(P1)} + \cdots + T_{\setminus M(P[n-2])} + c_{n-1} \tag{5.1}$$

Für die vollständig optimierte Ausführung ergibt sich:

$$T_{Anwendung.optimiert} = c_0 + T_{M(P0)} + c_1 + T_{M(P1)} + \cdots + T_{M(P[n-2])} + c_{n-1} \tag{5.2}$$

Unter obiger Annahme konstanter nicht-optimierbarer Bereiche c und der Gewinnberechnung des Moduleinsatzes zum Optimierungszeitpunkt Px mit

$$G_{M(Px)} = T_{\setminus M(Px)} - T_{M(Px)} \tag{5.3}$$

sowie $i = n - 1$ Optimierungspunkten, gilt:

$$T_{Anwendung} = T_{Anwendung.optimiert} + \sum_{0 \leqq k < i} G_{M(Pk)} \tag{5.4}$$

Entsprechend beschreibt $T_{Anwendung} : T_{Anwendung.optimiert}$ den Beschleunigungsfaktor, der sich durch die Nutzung aller einsetzbaren Hardware-Module ergibt.

Die Ausführungszeit $T_{Anwendung.optimiert}$ kann nur dann erzielt werden, falls alle notwendigen Hardware-Module innerhalb der verwendeten adaptiven Komponenten vorhanden sind bzw. durch Regelungsmechanismen bis zu den zugehörigen Optimierungspunkten bereitgestellt werden.

Jede aktive adaptive Komponente ist, auf Grund des Wirkungskreislaufs, eigenständig in ihrer internen Überwachung und Regelung. Die Darstellung des Anwendungsverhaltens (Abbildung 5.21) aus Sicht des Gesamtsystems lässt sich deshalb unverändert auf die Sichtweise einer einzelnen Komponente übertragen. Hierbei ist zu berücksichtigen, dass nur die Hardware-Module der betrachteten aktiven adaptiven Komponente in die Ausführungszeit einfließen. Optimierungsmöglichkeiten anderer Komponenten stellen sich diesbezüglich als nicht-optimierbare Bereiche c dar. Demzufolge sind die Gleichungen 5.1 bis 5.4 nicht nur für das Gesamtsystem, sondern für die Einzelbetrachtung einer Komponente anwendbar.

Hält eine einzelne aktive adaptive Komponente k rekonfigurierbare Bereiche bereit, ergeben sich folgende beide Situationen, falls m Hardware-Module im Zeitintervall $T_{Anwendung}$ zum Einsatz kommen sollen:

1. Mit $m \leqq k$ liegt ein trivialer Zustand vor, der die Einlagerung aller Module zulässt.
2. Im Fall $m > k$ muss eine Auswahl über die Verwendung von Modulen getroffen werden.

Ohne Unterstützung aus oberen Systemschichten lässt sich, aus Sicht der Hardware, im Allgemeinen keiner der beiden Fälle zufriedenstellend lösen. In Abschnitt 5.2.4 wurde ein cache-ähnlicher Aufbau für die Rekonfigurationsinfrastruktur zur Verdeckung von Rekonfigurationslatenzen vorgestellt. In diesem Zusammenhang existieren folgende Rahmenbedingungen im Hinblick auf die Nutzung von Hardware-Modulen innerhalb des Zeitraums $T_{Anwendung}$:

Definition 5.1 *Die Modulrekonfigurationszeit*

> Die Modulrekonfigurationszeit T_{MR} ist diejenige Zeit, die benötigt wird, um ein Modul, von der Anforderung des Moduls bis zum Abschluss der Rekonfiguration, in einer adaptiven Komponente verfügbar zu machen. T_{MR} wird

maßgeblich durch die Größe des Bitstreams, der das Modul beschreibt, und die Übertragungsrate des Ablagespeichers bzw. des zugehörigen Bussystems bestimmt. Für die vorgeschlagene hierarchische Rekonfigurationsinfrastruktur, mit ein oder mehreren Zwischenpuffern, ergeben sich für ein Modul so viele Modulrekonfigurationszeiten, wie Hierarchiestufen eingesetzt werden. □

In Anbetracht der Tatsache, dass sich die weiteren Ausführungen stets auf die *worst-case* bzw. *best-case* Szenarien beziehen, sind, unabhängig von der tatsächlich vorliegenden Anzahl von Hierarchiestufen, nur zwei Modulrekonfigurationszeiten von Interesse: Die Modulrekonfigurationszeit $T_{MR_{Speicher}}$[4], die sich durch das Nachladen und Rekonfigurieren eines Moduls ergibt, falls sich das Modul (bzw. der Bitstream des Moduls) in keinem Zwischenspeicher aufhält. Demgegenüber steht der optimistische Fall, dass sich das Modul bereits im *schnellsten* Zwischenpuffer befindet und die Modulrekonfigurationszeit $T_{MR_{Puffer}} \ll T_{MR_{Speicher}}$ ist [30, 68].

Des Weiteren ist, für eine allgemeine Betrachtung von Rekonfigurationsvorgängen, die Art der innerhalb einer adaptiven Komponente integrierten, rekonfigurierbaren Bereiche zu erwähnen. Kommen Multi-Kontext-Bereiche zum Einsatz, erfordert das Umschalten zwischen Modulen, die sich in unterschiedlichen Kontexten befinden, eine gewisse Zeit, die so genannte Modulanwahlzeit.

Definition 5.2 *Modulanwahlzeit*

Die Modulanwahlzeit T_{MA} ist diejenige Zeit, die benötigt wird, um ein Modul, das bereits in einem rekonfigurierbaren Multi-Kontext-Bereich liegt, anzuwählen und dadurch verfügbar zu machen. Unabhängig von der Größe des Moduls ist T_{MA} konstant. Liegt das gewünschte Modul bereits im ausgewählten Kontext, ist $T_{MA} = 0$. □

Existiert nur ein einziger Kontext in einem rekonfigurierbaren Bereich, kann dies als Spezialfall angesehen werden, für den stets $T_{MA} = 0$ gilt. Darüber hinaus stellten M. Motomura *et al.* [48, 157] eine Umsetzung eines Multi-Kontext-Bereichs vor, für den $T_{MA} < 5ns \ll T_{MR_{Puffer}}$ nachweisbar ist. Bis zu einem Frequenzbereich von 200MHz, der für das adaptive Gesamtsystem realistisch erscheint, kann dementsprechend eine Kontextanwahl innerhalb eines Taktzyklus abgewickelt werden. Zusätzlich besteht die Möglichkeit, mit Hilfe von Fließbandverarbeitung, durch Verarbeitungsphasen, die der Ausführung innerhalb der Hardware-Module vorgelagert sind, eine verdeckte Umschaltung durchzuführen. Dies führt dazu, dass in der Regel die Modulanwahlzeit $T_{MA} = 0$ angenommen werden kann und demzufolge nicht in die weiteren Betrachtungen mit einfließt.

Insgesamt ist somit die Nutzung eines Hardware-Moduls im schlechtesten Fall von $T_{MR_{Speicher}}$ und im besten Fall von $T_{MR_{Puffer}}$ abhängig. Die Modulrekonfigurationszeit $T_{MR_{Speicher}}$ ist grundsätzlich in einer Größenordnung von mehreren 10 Mikrosekunden bis hin zu wenigen Millisekunden anzusetzen [44, 68, 55]. Da dies an den Zeitraum $T_{Anwendung}$ beim Einsatz eines Zeitscheibenverfahrens heranreicht bzw. diesen übertrifft, ist ein ungepuffertes Modul aus Sicht einer aktiven adaptiven Komponente innerhalb des Zeitraums $T_{Anwendung}$ nicht nutzbar. Hierfür müssen die oberen Systemschichten herangezogen werden, die dafür Sorge

[4] Hierbei handelt es sich nicht um den Hauptspeicher des Systems. Vgl. Abschnitt 5.2

tragen, dass die entsprechenden Module einer Anwendung bereits zu Beginn der Rechenzeit im Puffer liegen. Die Auswahl von Modulen basiert anfangs auf statischen Informationen des Übersetzers. Im weiteren Ausführungsverlauf bilden explizite Konfigurationsanweisungen sowie Überwachungsdaten die innerhalb der Hardware-Schicht, respektive den adaptiven Komponenten, gewonnen werden, die Grundlage für das Puffern von Modulen.

5.3.4.1 Problematik der Anpassung

Wie bereits erwähnt, ergeben sich aus der Verfügbarkeit von m Hardware-Modulen und k rekonfigurierbaren Bereichen zwei grundlegende Fälle: Zum einen, dass $m \leqq k$ und somit ohne Einschränkung alle Module gleichzeitig in einer adaptiven Komponente eingelagert sein können, und zum anderen der Fall, dass $m > k$ und sich demzufolge maximal k aus m Modulen zeitgleich in einer Komponente aufhalten können. Des Weiteren wurde festgestellt, dass, unabhängig von der Anzahl m, aus Sicht einer adaptiven Komponente nur Module für die Ausführung in Betracht kommen, die sich bereits im Puffer befinden. Das Nachladen aus dem Pool-Speicher dauert zu lange. Das Puffern von Modulen ist Aufgabe des Betriebssystems. Diesbezüglich stellen z.B. S. Ogrenci Memik *et al.* [108] und S. Ghiasi *et al.* [55] Scheduling-Algorithmen vor, die sich zur Anwendung für das adaptive Gesamtsystem eignen. Im Weiteren wird davon ausgegangen, dass die benötigten Module gepuffert vorliegen.
Ob $m \leqq k$ gilt, ist im Allgemeinen bereits zur Übersetzungszeit bekannt, da der Übersetzer die Erzeugung bzw. Auswahl von Hardware-Modulen aus der Bibliothek vornimmt (siehe Abschnitt 5.2). Aus Sicht des Übersetzers ergeben sich wiederum zwei Situationen:

- Für die gesamte Anwendung trifft $m \leqq k$ zu. Dies kann stets durch Vorkonfiguration der adaptiven Komponenten mit Hilfe des Betriebssystems optimal gelöst werden.

- Eine Anwendung kann in n, mit $n > 1$, logisch unterscheidbare Ausführungsphasen eingeteilt werden. Jede Phase lässt sich mit $m_{Phase} \leqq k$ Hardware-Modulen optimieren. Da im Allgemeinen logische Ausführungsphasen nicht mit einzelnen Zeitscheiben der Abarbeitung in Verbindung gesetzt werden können, ergibt sich $m_{Anwendung} = \sum_a m_{Phase_a}$, mit $a \epsilon \{0, ..., n-1\}$, als Anzahl der innerhalb einer Zeitscheibe benötigten Module. Diese Situation kann wiederum in die ursprünglichen Fälle $m_{Anwendung} \leqq k$ bzw. $m_{Anwendung} > k$ zurückgeführt werden.

Aus Sicht einer aktiven adaptiven Komponente ist folglich stets der Fall $m > k$ von Interesse und weitestgehend optimal zu lösen.

5.3.4.2 Anpassung bei Bedarf

Ein intuitiver Ansatz besteht darin, die Hardware-Module, die nicht zum Ausführungsbeginn (einer Zeitscheibe) in eine adaptive Komponente geladen werden können, bei Bedarf verfügbar zu machen (*demand fetching*). Obwohl dies eine einfach umsetzbare Lösung der Anpassungsproblematik darstellt, ist eine praktische Umsetzung nur bedingt möglich. Dies liegt grundsätzlich daran, dass der Bedarf eines Moduls X erst zum Zeitpunkt P_X erkennbar

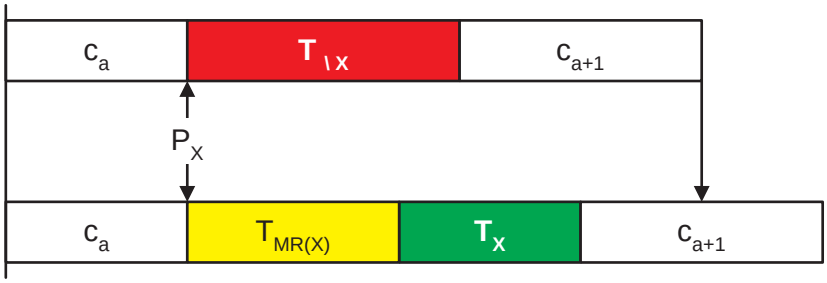

Abbildung 5.22: Anpassung im Bedarfsfall

ist (Abbildung 5.22)[5]. Zum Zeitpunkt P_X müsste eine Rekonfiguration eingeleitet werden. Nach deren Abschluss erfolgt die hardware-beschleunigte Ausführung. Wobei somit eine gesamte Ausführungzeit $T_{Bedarf}(X) = T_{MR}(X) + T_X$ angenommen werden muss. Bis auf wenige Sonderfälle ist die Rekonfigurationszeit $T_{MR}(X)$, falls sich Modul X bereits im Puffer befindet, in der Größenordnung von $T_{\setminus X}$ bzw. darüber anzusetzen[6]. Generell führt eine derartige bedarfsorientierte Lösung zu der in Abbildung 5.22 dargestellten Situation und verlängert die Verarbeitungszeit einer Anwendung.

5.3.4.3 Anpassung durch absolute Häufigkeit

Eine weitere Möglichkeit zur dynamischen Anpassung während einer Zeitscheibe besteht darin, die Häufigkeit von möglichen Hardware-Modulen zu ermitteln. Auf Basis dieser Überwachungsdaten – die mittels Hardware-Zählern [136, 27] gewonnen werden können – ist eine Rekonfiguration durchführbar. Im Gegensatz zu obiger bedarfs-orientierten Lösung, erlaubt dies den Einsatz von k aus m Modulen, unter Einbeziehung des Laufzeitverhaltens der Anwendung.

Abbildung 5.23 skizziert den möglichen Einsatz zweier Module A und B innerhalb eines gegebenen Überwachungsintervalls $T_{Überwachung}$. Eine allgemeine Regel für die Festlegung des Überwachungsintervalls wird nicht angegeben, da die maximale Länge des Intervalls maßgeblich von der Größe der eingesetzten Hardware-Zähler und den zu zählenden Ereignissen abhängt. Insgesamt sollte bei einer Implementierung darauf geachtet werden, dass ein gewählter Überwachungszeitraum $T_{Überwachung} \ll T_{Anwendung}$ ist. Da die Auswertung T_{Reg} der Überwachungsdaten stets am Ende eines Intervalls stattfindet, lassen sich mit zunehmender Länge des Intervalls immer weniger mögliche Anpassungen während der Anwendungslaufzeit durchführen. Umgekehrt liefern zu kurze Intervalle – in Bezug auf überwachte Ereignisse – keine aussagekräftigen Daten.

Obwohl in Abbildung 5.23 die Häufigkeit von Modul A die des Moduls B übersteigt, muss zudem der Effekt beider Module mit in die Entscheidungsfindung einbezogen werden. Die

[5] Hierbei handelt es sich um den allgemeinen Fall des Moduleinsatzes zur Laufzeit. In ausgesuchten Fällen ist die statische Information zur Zeit der Programmübersetzung ausreichend, um entsprechende Konfigurationsinstruktionen in den Programmfluss einzubauen und somit Module vorab verfügbar zu machen.

[6] Logische Operation oder Schiebeoperationen lassen sich sehr effizient in FPGAs umsetzen und führen häufig zu kleinen Modul-Bitstreams, die entsprechend zeitnah rekonfiguriert werden können (Vgl. [2, 11, 12, 155, 156]).

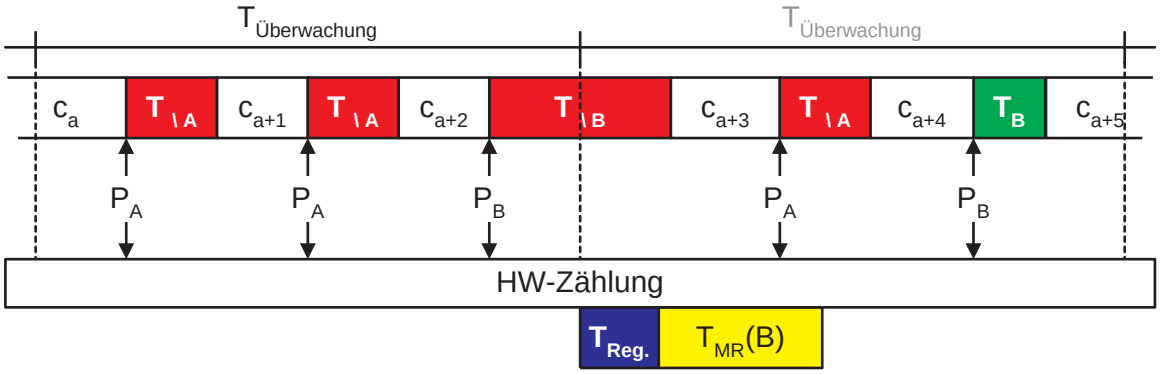

Abbildung 5.23: Häufigkeit als Grundlage der Anpassung

Häufigkeit eines Moduls bezeichnet die mögliche und tatsächliche Nutzung dieses Moduls. Die positive Auswirkung eines beliebigen Moduls X mit der absoluten Häufigkeit H_X auf die partielle bzw. gesamte Ausführungszeit einer Anwendung lässt sich folgendermaßen beschreiben:

$$T_{Effekt.gesamt}(X) = \sum_{0 \leqq i < H_X} a(i) \cdot G_X(i), \; mit \; a(i) \; \epsilon \; \{0,1\} \tag{5.5}$$

Hierbei repräsentiert $G_X(i)$ den Zeitgewinn, der durch den Einsatz eines Moduls X zum Zeitpunkt i im Gegensatz zur Ausführung in Software erzielt werden kann. Entsprechend gibt $a(i) = 1$ an, dass zu einem Zeitpunkt i das Modul X zum Einsatz kam, und $a(i) = 0$, dass das Modul X nicht genutzt wurde. Je größer der Wert von $T_{Effekt.gesamt}(X)$, desto rentabler ist der Einsatz des Moduls X und umgekehrt.

Soll neben dem Zeitaspekt der Energieaspekt der Module mit einbezogen werden, muss die Gewinnkomponente eine entsprechende Erweiterung erfahren. In diesem Zusammenhang bietet sich z.B. eine Bewertung nach S. Abdelwahed *et al.* [1] an, die zu folgender Gewinnberechnung führt:

$$G(i) = c_{Rechenzeit} \cdot g_{Rechenzeit}(i) + c_{Energie} \cdot g_{Energie}(i) \tag{5.6}$$

Sowohl $c_{Rechenzeit}$ als auch $c_{Energie}$ sind aus Sicht der adaptiven Komponenten Konstanten, die eine Gewichtung der einzelnen Gewinnbeiträge erlauben. Die Einstellung beider Werte und somit die Zielsetzung des Gesamtsystems (siehe Kapitel 4) obliegt dem Betriebssystem in Abhängigkeit von Umgebungs-, Anwendungs- oder Benutzeranforderungen. Hierbei gilt $c_{Rechenzeit}, c_{Energie} \; \epsilon \; [0,1]$ und $c_{Rechenzeit} + c_{Energie} = 1$. $g_{Rechenzeit}(i)$ und $g_{Energiebedarf}(i)$ beschreiben den Gewinn in Bezug auf die Rechenzeit bzw. den Energiebedarf, falls ein Modul zum Zeitpunkt i eingesetzt wird. Bei beiden Funktionen ist der Wertebereich $\mathbb{R}$ zulässig. Wobei ein Gesamtgewinn $G(i) \leqq 0$ den Einsatz eines Hardware-Moduls fragwürdig erscheinen lässt, und demzufolge nur Module mit $G(i) > 0$ für einen Einsatz in Betracht gezogen werden.

Unabhängig davon ergibt sich ein praktisches Problem bei der Ermittlung des Gewinns $G(i)$ zum Zeitpunkt i. Dies liegt daran, dass der reale Gewinn zu einem bestimmten Zeitpunkt von dynamischen Faktoren beeinflusst sein kann. Z.B. eine 32-Bit Ganzzahlmultiplikation er-

laubt in Abhängigkeit von den Werten der Operanden verschiedene abkürzende Maßnahmen – sowohl in Software als auch in Hardware. Hierunter fällt die Behandlung, falls ein Operand den Wert Null trägt, ebenso wie ein vorzeitiges Beenden, falls die verbleibenden Bits des Operandenwerts Null sind. Wird ein Hardware-Modul eingesetzt – wie z.B. im ARM 9 – ergeben sich Ausführungszeiten im Bereich von 1 bis 4 Taktzyklen [7]. Ungleich schwieriger gestaltet sich die Angabe der Rechenzeit für eine Software-Implementierung, da zusätzlich zu den benötigten Instruktionen noch weitere dynamische Faktoren, wie Registerabhängigkeiten, bedingte Sprünge usw. in die Ausführungszeit einfließen. Obwohl es nicht unmöglich ist, genaue Angaben über die Ausführung zu ermitteln, ist der Aufwand für eine Laufzeitregelung bzw. Anpassung, speziell einer hardware-basierten Regelung, aus praktischer Sicht abzulehnen. Der Gewinn eines Hardware-Moduls muss auf Basis von Durchschnittswerten oder *worst-case*-Abschätzungen angegeben werden, die statisch, als konstantes Attribut, einem Hardware-Modul zugeordnet sind (Vgl. Abschnitt 5.2).

In diesem Zusammenhang erweist sich die Ermittlung eines durchschnittlichen Laufzeitgewinns mittels Zeitmessung ohne und mit Einsatz eines bestimmten Hardware-Moduls als relativ einfach. Zusätzlich kann zur weiteren Vereinfachung auf die Gegenüberstellung des *worst-case* Zeitverhaltens eines Moduls und der *best-case* Verarbeitung in Software zurückgegriffen werden. Aufwendiger gestaltet sich die Erhebung des energetischen Verhaltens, sowohl von Hardware- als auch Software-Lösungen. Diesbezüglich beschreiben z.B. Haid *et al.* [64] ein Verfahren, welches sich sowohl für dynamische als auch statische vorab Schätzungen des Energiebedarfs von FPGA-basierten Schaltungen eignet. Die maximal festgestellte Abweichung der Schätzung von den Messwerten liegt unterhalb von 5%.

Im Weiteren wird davon ausgegangen, dass die für die Gewinnberechnung erforderlichen Parameter, den Hardware-Modulen als Attribute zugewiesen und den entsprechenden aktiven adaptiven Komponenten mittels der Systembibliothek bekannt sind (siehe auch Abschnitt 5.2). Demzufolge vereinfacht sich die Berechnung des Effekts eines Moduls X zu:

$$T_{Effekt.gesamt_X} = \sum_{0 \leqq i < H_X} a(i) \cdot G_X, \ mit\ a(i) \epsilon \{0,1\}\ und\ G_X\ konstant. \tag{5.7}$$

Des Weiteren ergibt sich der theoretisch mögliche Gewinn durch den Einsatz eines Moduls X zu:

$$T_{Effekt.optimal_X} = \sum_{0 \leqq i < H_X} a(i) \cdot G_X = H_X \cdot G_X, \ mit\ a(i) = 1 \tag{5.8}$$

Wird nur die Häufigkeit eines möglichen Moduleinsatzes zur Regelung herangezogen, erfolgt nach jedem Überwachungsintervall eine Gewinnberechnung anhand der vorliegenden Messwerte. Eine Änderung der Konfiguration der adaptiven Komponente wird nur dann vorgenommen, wenn sich ein oder mehrere bisher nicht eingelagerte Module als geeignet erweisen. Folgende Situationen können auftreten:

- Es sind i rekonfigurierbare Bereiche ungenutzt. Demzufolge werden bis zu maximal i Module, mit den höchsten Gewinnerwartungen $T_{Effekt.optimal}$ eingelagert.
- Ein oder mehrere Module zeigen eine höhere Gewinnerwartung, als dies bei bereits eingelagerten Modulen der Fall ist. Entsprechend werden die Module mit den niedrigsten Bewertungen bevorzugt durch die mit den höchsten Bewertungen ersetzt.

Im letzteren Fall muss die Rekonfigurationszeit T_{MR} der einzulagernden Module berücksichtigt werden. Während der Rekonfigurationsphase ist der betroffene konfigurierbare Bereich nicht rechenbereit. Welche Auswirkungen die Rekonfiguration hat, ist maßgeblich von der implementierten Konfigurationsinfrastruktur abhängig (siehe auch Abschnitt 5.2.4). Somit kann eine allgemeine Vorschrift zur Behandlung dieser Situation nicht angegeben werden. Dennoch sollten Rekonfigurationen grundsätzlich so erfolgen, dass die Auswirkungen auf die Ausführungszeit einer Anwendung gering sind.

Die Einbeziehung des sich zur Laufzeit ergebenden Effekts eines Hardware-Moduls löst dennoch nicht die Problematik, dass nur jeweils maximal k von m Modulen zum Einsatz kommen, wenn k rekonfigurierbare Bereiche zur Verfügung stehen. Zudem ist der tatsächliche Effekt der Regelung stark vom Anwendungsverhalten abhängig. Sind die Nutzungshäufigkeiten nicht über mehrere Überwachungsintervalle ähnlich, kann im schlechtesten Fall, durch ständige Rekonfigurationsvorgänge, eine Laufzeitverlängerung die Folge sein.

5.3.4.4 Regelung bei zyklischem Anwendungsverhalten

In diesem Kontext liefert das Verhalten von Anwendungen einen unterstützenden Beitrag zur Regelung. Der Großteil der gesamten Rechenzeit einer Anwendung wird in der Regel für die Abarbeitung von Schleifen oder geschachtelten Schleifen aufgewendet [99] (Vgl. auch [140] für wissenschaftliche, technische Anwendungen oder [93] im Zusammenhang mit adaptiven Systemen).

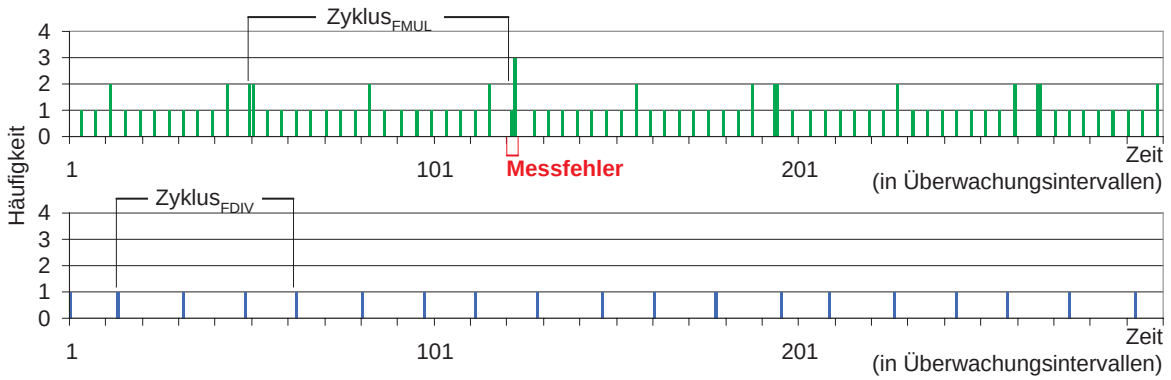

Abbildung 5.24: Exemplarische Häufigkeitsverteilung zweier Gleitkommaoperationen

Diesbezüglich lässt sich für den adaptiven Prozessor, als exemplarische aktive adaptive Komponente, für alle Beispielanwendungen ein zyklisches Nutzungsmuster eingesetzter und einsetzbarer Hardware-Module erkennen. Ohne den Ergebnissen, die ausführlich in Kapitel 6 besprochen werden, vorzugreifen, zeigt Abbildung 5.24 die durch Simulation erhaltenen Nutzungsprofile zweier Module. Bei den Modulen handelt es sich um eine Gleitkommamultiplikation (FMUL, oben) und eine Gleitkommadivision (FDIV, unten). Die beiden Profile sind aus der Simulation der BaseFP-Anwendung der EEMBC-Benchmarks entnommen. Für die Profilerstellung wurde ein Überwachungsintervall von 100 Taktzyklen vorgegeben. Folglich zeigt Abbildung 5.24 einen Zeitraum von 30.000 Taktzyklen. In y-Richtung ist die Nutzungshäufigkeit eines Moduls innerhalb eines Überwachungsintervalls abgetragen. Des

Weiteren ist ein Messfehler innerhalb des FMUL-Profils markiert. Darüber hinaus sind zwei mögliche Zyklen gekennzeichnet. Der $Zyklus_{FMUL}$ ergibt sich intuitiv. Für die Nutzung des FDIV-Moduls ist die optische Erkennung eines Zyklus schwieriger. Der in Abbildung 5.24 gewählte $Zyklus_{FDIV}$ ist der Zyklus mit dem kleinsten Zeitintervall. Insgesamt lässt sich feststellen, dass die Modulnutzung während der Anwendungsausführung in der Regel einem zyklischen Verlauf folgt[7]. Die sich bei einer Implementierung ergebenden Probleme in Verbindung mit Messfehlern, der Wahl eines Zyklus usw. sind Thema des Abschnitts 5.3.6.

Die prinzipielle Problematik einer bedarfsorientierten Rekonfiguration eines Hardware-Moduls ist die dafür benötigte Rekonfigurationszeit T_{MR}. Berücksichtigt man jedoch ein zyklisches Nutzungsverhalten eines Moduls, ergibt sich hieraus die Möglichkeit einer *Vorausplanung*. Kann die nächste Nutzung eines Moduls vorhergesagt werden, erlaubt dies eine Einlagerung dieses Moduls derart, dass die gesamte Rekonfigurationszeit T_{MR} verdeckt wird. Aus Sicht der Anwendung würde dies zu einem optimalen Einsatz aller Module führen. In diesem Zusammenhang existieren grundsätzliche Probleme der Überwachung und Regelung, unabhängig von einer Umsetzung in Hardware.

5.3.4.4.1 Grundsätzliche Probleme der Zyklenerkennung In Verbindung mit der Erkennung von zyklischem Verhalten in Anwendungen ergeben sich drei prinzipielle Problemstellungen:

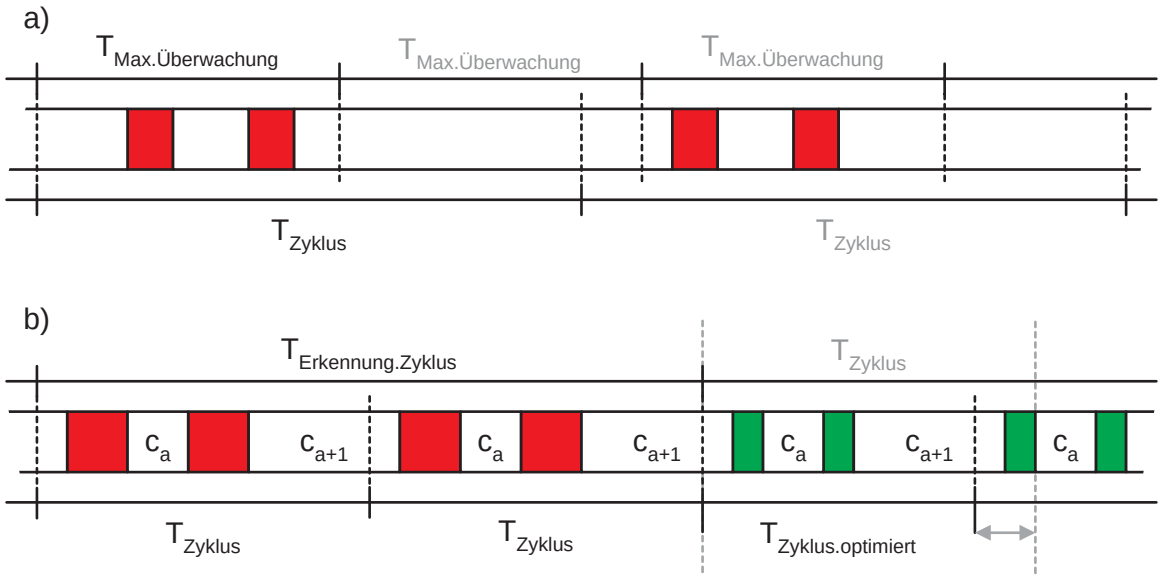

Abbildung 5.25: Prinzipielle Probleme der Zyklenerkennung

- Die **Startpunktsuche**, beschreibt die Notwendigkeit, einen Anfang bzw. ein Ende eines Zyklus zu bestimmen. Nur falls ein Startpunkt ermittelt werden kann, lässt sich ein Zyklus identifizieren.

[7] Alle Beispielanwendungen die im Verlauf der Arbeit untersucht wurden, zeigen ein zyklisches Modulnutzungsverhalten.

- Das Problem der **Zyklusüberlänge** bzw. eines zu großen Zeitlags (Vgl. Abschnitt 3.4). Zeigt das Anwendungsverhalten einen Zyklus, dessen Dauer einen gegebenen maximalen Überwachungszeitraum überschreitet, ist es nicht möglich, diesen zu erkennen (Abbildung 5.25 a).
- Als **Zyklenverkürzung** wird die Problematik bezeichnet, die entsteht, falls ein Zyklus identifiziert wurde und daraufhin optimierende Maßnahmen ergriffen werden. Durch den Einsatz von optimierenden Hardware-Modulen, wird die ursprüngliche Zykluszeit, wie in Abbildung 5.25 b illustriert, verkürzt. Das zeitliche Verhalten des ursprünglichen Zyklus stellt sich im Verhältnis zum optimierten Zyklus länger dar. Der Zeitlag der Messreihe wird verkleinert.

Zur Lösung der Startpunktproblematik wird davon ausgegangen, dass es, falls ein Zyklus vorliegt, unerheblich ist, welchen Startpunkt man wählt (Vgl. Abschnitt 3.4). Um unnötige Überwachungsdaten zu vermeiden, bietet sich an, den Startpunkt so zu wählen, dass dieser mit der ersten Nutzung bzw. Nutzungsmöglichkeit eines Moduls zusammenfällt. Somit beginnt die Überwachung mit dem ersten Auftreten eines Moduls. Der vorangehende Zeitraum wird nicht beachtet. Diese Vorgehensweise gewährleistet dennoch nicht, dass ein gültiger Startpunkt gefunden wird. Diesbezügliche Probleme und Lösungen werden in Unterabschnitt 5.3.6 behandelt.

Die in Abbildung 5.25 a skizzierte Problematik der Zyklusüberlänge kann nicht gelöst werden. Ein maximal zulässiges Überwachungsintervall $T_{Max.Überwachung}$ ist implementierungsabhängig. In Anbetracht des Ressourcenbedarfs für Überwachungsdaten ergibt sich zwangsläufig eine starke Beschränkung dieses Zeitraums. Demzufolge können Zyklen stets länger als das größtmögliche Überwachungsintervall sein. Obwohl sich keine allgemeine Lösung finden lässt, kann durch vorausgehende Rekonfigurationsmaßnahmen entscheidend auf die Zykluslänge eingewirkt werden. Hierbei sind z.B. statische Informationen zur Nutzung von Modulen aus der Übersetzerschicht von großem Wert. Das vorzeitige Einlagern von Modulen verkürzt die Zykluszeit und erhöht somit die Wahrscheinlichkeit, dass ein Zyklus, dessen Ausführungszeit ohne Optimierung länger als das Überwachungsintervall ist, dennoch identifiziert werden kann.

Im Gegensatz dazu ist eine Zyklenverkürzung, die sich aus der dynamischen Regelung (Abbildung 5.25 b) ergibt, problematisch. Dies führt häufig dazu, dass ein Zyklus T_{Zyklus} in der optimierten Variante $T_{Zyklus.optimiert}$, anhand der vorliegenden Überwachungsdaten, nicht mehr erkennbar ist. Dieser Effekt muss bei der Regelung mit berücksichtigt werden, und gegebenfalls zu einer Korrektur der Überwachungsdaten führen. Das Ausmaß solcher Korrekturmaßnahmen ist von den überwachten bzw. eingesetzten Hardware-Modulen abhängig.

5.3.4.4.2 Verhalten konkurrierender Module Das Ziel der Zyklenerkennung und der darauf aufbauenden Regelung ist der Einsatz mehrerer konkurrierender Hardware-Module innerhalb eines einzigen rekonfigurierbaren Bereichs. Im Idealfall alterniert die Nutzung der einzelnen Hardware-Module derart, dass keine Konfliktsituationen entstehen.

Im Allgemeinen zeigt sich jedoch ein Problem im Zusammenhang mit dem Auftreten mehrerer Optimierungsmöglichkeiten innerhalb eines Zyklus. Abbildung 5.26 skizziert die Problematik zweier konkurrierender Module, die sich einen einzigen rekonfigurierbaren Bereich teilen sollen. Dies lässt sich auf eine beliebige Anzahl konkurrierender Module übertragen,

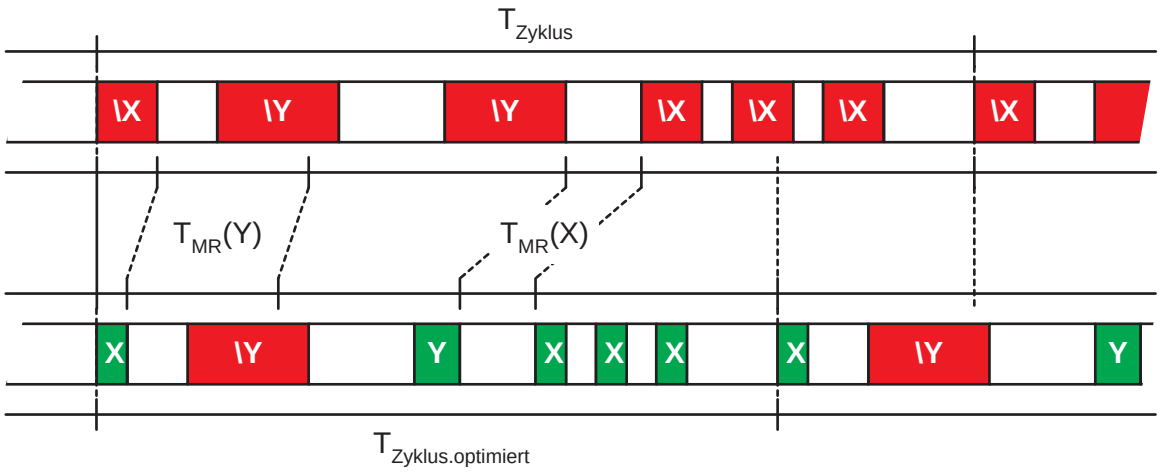

Abbildung 5.26: Verhalten konkurrierender Module

wird aber exemplarisch nur für zwei Module X und Y behandelt. Aus Platzgründen wurde in Abbildung 5.26 die Beschriftung für die nicht-optimierte Ausführungszeit $T_{\backslash X}$ des Moduls X, zu $\backslash X$ vereinfacht. Gleiches gilt für die optimierte Ausführungszeit des Moduls X, sowie entsprechend für die beiden Situationen des Moduls Y.

Es wird davon ausgegangen, dass ein Zyklus der Länge T_{Zyklus} ermittelt wurde. Somit gilt ein zulässiger Startpunkt als gefunden, und es liegt keine Zyklusüberlänge vor. Darüber hinaus wird die Behandlung der Zyklenverkürzung zu $T_{Zyklus.optimiert}$ als gelöst betrachtet. Dennoch zeigt sich in Abbildung 5.26, dass nicht alle Optimierungsmöglichkeiten genutzt werden können. Beide Module, X und Y, sollen sich einen rekonfigurierbaren Bereich teilen. D.h. entweder X oder Y ist verfügbar, aber niemals beide gleichzeitig. Dies ist nur dann möglich, wenn die Rekonfigurationszeit T_{MR} der Module durch die nicht optimierbaren Anwendungsteile verdeckt werden kann. Abbildung 5.26 demonstriert beide Fälle. Hierbei kann das erste Auftreten von Y nicht durch den Moduleinsatz beschleunigt werden ohne zumindest einen Einsatz von Modul X zu blockieren. Der Einsatz für den zweiten Optimierungspunkt ist jedoch möglich. Des Weiteren ist die Nutzung des Moduls X in Abbildung 5.26 stets zulässig.

Zusätzlich können Konstellationen entstehen, für die gilt, dass die Rekonfigurationszeit T_{MR} nur teilweise oder nicht verdeckt werden kann, jedoch zusammen mit der Modulausführungszeit weniger Zeit in Anspruch nimmt, als die nicht-optimierte Variante.

5.3.5 Resultierender Regelungsablauf

Nachdem die allgemeine Problemstellung in Bezug auf die Regelung erörtert wurde, kristallisiert sich folgender Regelungsablauf für aktive adaptive Komponenten heraus:

1. Eine software-gesteuerte Vorkonfiguration mit entsprechenden Hardware-Modulen. Dies bietet zum einen die Möglichkeit, die Einhaltung geforderter Laufzeitkriterien zu erfüllen und zum anderen werden Anwendungszyklen – falls vorhanden – bereits im Vorfeld zeitlich verkürzt.

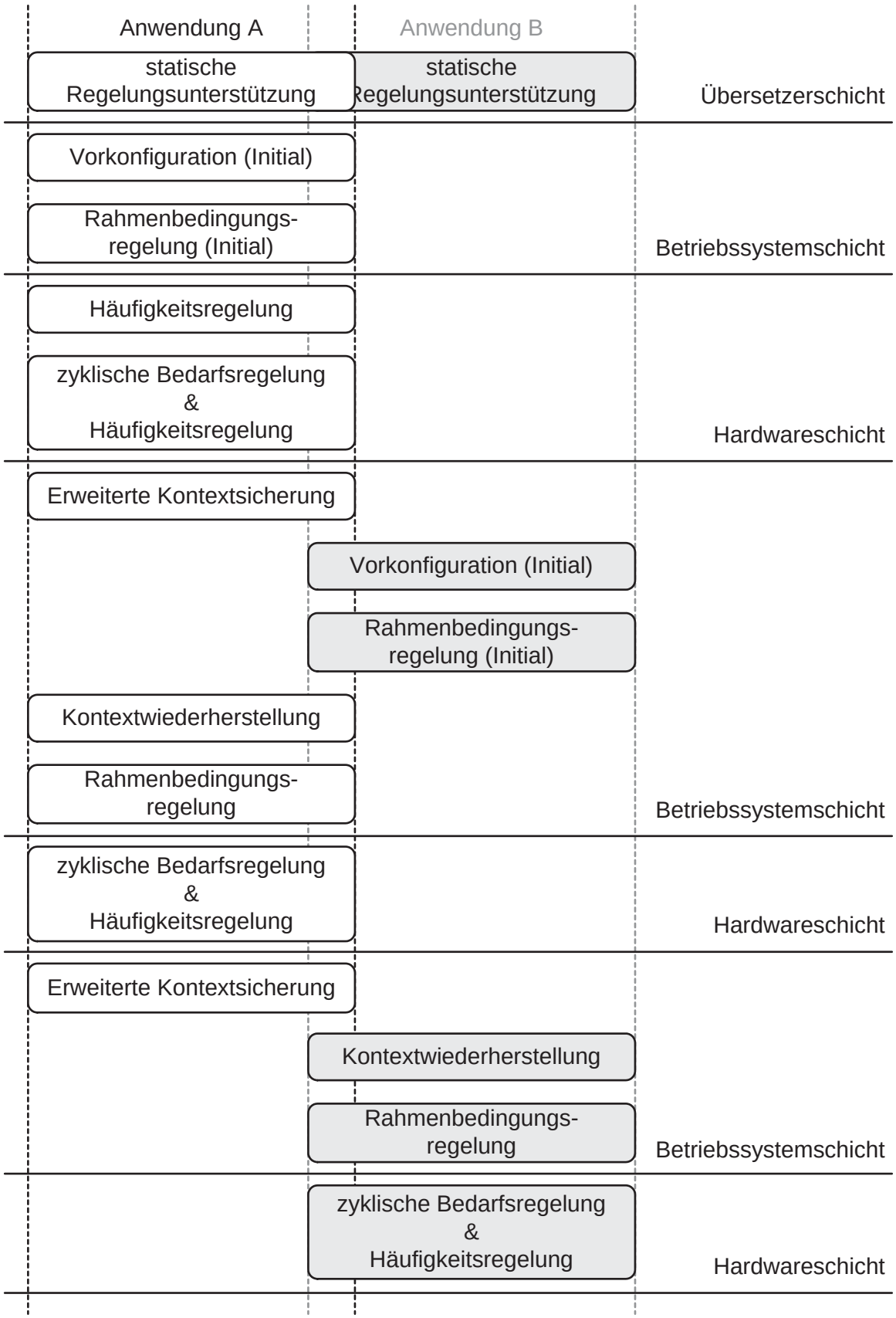

Abbildung 5.27: Übersicht über den Regelungsablauf des Gesamtsystems

2. Bei erstmaliger Ausführung einer Anwendung stehen als nächstes die Ermittlung von Optimierungspunkten und deren Häufigkeit. Erlaubt dies ein oder mehrere Module nachzuladen, kann das gegebenenfalls zu einer zusätzlichen Zyklusverkürzung beitragen.

3. Des Weiteren folgt die Zykluserkennung. Kann kein Zyklus identifiziert werden, bleibt die Modulnutzung auf die Anzahl verfügbarer rekonfigurierbarer Bereiche beschränkt. Anderenfalls wird anhand des zyklischen Verhaltens eine bedarfsorientierte Modulkonfiguration, die zusätzlich die Häufigkeit der Module berücksichtigen muss, angestrebt.

Abbildung 5.27 verdeutlicht den Regelungsablauf exemplarisch für zwei Anwendungen A und B unter Einbeziehung der Übersetzerschicht als statische Unterstützungsmaßnahme. Die dynamische Regelung beginnt mit dem Eintritt in die Betriebssystemschicht.

5.3.6 Grundlagen der Überwachungsumsetzung

Um i Module, mit $i \geqq 2$, in einem einzigen rekonfigurierbaren Bereich einsetzen zu können, muss das Nutzungsverhalten der Module betrachtet werden. Ein Modul X kann stets dann eingesetzt werden, wenn kein anderes Modul $Y \neq X$ zu diesem Zeitpunkt verfügbar sein muss. Zudem soll Modul X vom aktuell im Bereich eingelagerten Modul verschieden sein. Dies führt dazu, dass die Modulrekonfigurationszeit $T_{MR}(X)$ mit berücksichtigt werden muss. Während eines Rekonfigurationsvorgangs ist der betroffene rekonfigurierbare Bereich nicht rechenbereit. Für den Einsatz eines Moduls X muss somit die Modulrekonfigurationszeit $T_{MR}(X)$ dergestalt in eine Regelung einfließen, dass sowohl die Nutzung des Moduls X als auch die der Vorgängermodule und etwaiger Nachfolger gewährleistet wird. Möglich ist dies ausschließlich dann, wenn das Nutzungsverhalten der Vorgänger- und Nachfolgermodule entsprechende Lücken (Korrelation des Nutzungsverhaltens unterschiedlicher Module) aufweist, die eine konfliktfreie Konfiguration mit nachfolgender Nutzung zulassen.

Darüber hinaus müssen solche Lücken vorhersagbar sein und demzufolge ein zyklisches sich wiederholendes Verhalten zeigen (Autokorrelation des Nutzungsverhaltens eines Moduls mit anwendungsspezifischem Zeitlag). Kann kein Zyklus festgestellt werden, kann entsprechend keine Regelung greifen. Somit kommen nur Fälle in Betracht, die ein reproduzierbares, zyklisches Nutzungsverhalten zeigen.

In erster Linie sind die Lücken oder Nullstellen des Nutzungsverhaltens von Interesse. Solche Nullstellen ermöglichen erst den Einsatz mehrerer Module im selben rekonfigurierbaren Bereich. Hierbei existieren zwei Extremfälle, die nicht untersucht werden müssen, da keine Regelung notwendig bzw. möglich ist:

- Ein Modul wird während des gesamten Betrachtungszeitraums nicht genutzt. Das Nutzungsverhalten weist durchgehend Nullstellen auf. Es besteht also kein Bedarf, dieses Modul einzulagern und demzufolge muss dieses Modul nicht berücksichtigt werden.

- Dem gegenüber steht der Fall, dass ein Modul *ständig* zum Einsatz kommt, und entsprechend keine verwertbaren Lücken liefert, um ein oder mehrere andere Module zwischenzeitlich einzulagern.

Folglich beschränkt sich die Realisierung der Überwachung und Regelung auf die Erkennung und Nutzbarmachung von zyklisch auftretenden Nullstellen im Nutzungsverhalten von Modulen.

5.3.6.1 Allgemeine Zyklenerkennung

Prinzipiell lassen sich die Aufgaben der Überwachung und Regelung wie folgt beschreiben:

- Aufzeichnen des Nutzungsverhaltens und fortlaufende Erstellung einer Nutzungshistorie.
- Das entstehende Nutzungsprofil auf geeignete Nullstellen testen. Treten ein oder mehrere Nullstellen auf, folgt
- die Identifikation zyklischen Verhaltens. Wird ein reproduzierbares Verhalten festgestellt, dient das entsprechende Nutzungsprofil als Muster für die weitere Regelung.
- Lassen sich zwei oder mehrere Muster korrespondierender Module so überlagern, dass die Nullstellen der Module jeweils auf die genutzten Intervalle anderer Module treffen, bilden die zugehörigen Muster die Basis für einen optimierenden Regeleingriff.
- Die Regelung veranlasst in Abhängigkeit von den erhaltenen Mustern eine Rekonfiguration mit den entsprechenden Modulen. Gegebenenfalls ist eine Musterkorrektur zum Ausgleichen einer Zyklenverkürzung notwendig.

5.3.6.1.1 Erstellen einer Nutzungshistorie Zur Erstellung einer Nutzungshistorie bietet sich an, die Nutzungshäufigkeit eines Moduls innerhalb konstanter Zeitintervalle zu ermitteln, und diese am Ende jedes Intervalls in ein Nutzungsprofil zu übertragen. Die konstanten Zeitintervalle repräsentieren die Granularität der Überwachung und werden mit T_G bezeichnet. Sind $n_{Max.Profil}$ Einträge in einem Nutzungsprofil zulässig, ergibt sich die Länge $\overline{T_{Max.Überwachung}}$ des maximalen Überwachungsintervalls stets zu:

$$\overline{T_{Max.Überwachung}} = n_{Max.Profil} \cdot \overline{T_G} \; mit \; n_{Max.Profil} \; \epsilon \; \mathbb{N} \tag{5.9}$$

Die maximale Anzahl möglicher Einträge eines Nutzungsprofils hängt davon ab, in welchem Umfang Hardware-Ressourcen bei einer Implementierung für die Überwachung bereitstehen bzw. genutzt werden können.

Im Laufe der Überwachung ergibt sich der Überwachungszeitraum $T_{Überwachung}$, falls ein Zyklus innerhalb der Zeitspanne $\overline{T_{Max.Überwachung}}$ erkannt wurde. Die Länge $\overline{T_{Überwachung}}$ des Überwachungszeitraums entspricht:

$$\overline{T_{Überwachung}} = i \cdot \overline{T_G} \; mit \; i \; \leq \; n_{Max.Profil} \tag{5.10}$$

Allgemein gilt folgender Zusammenhang, falls ein Zyklus erkannt wird und dafür i Einträge eines Nutzungsprofils mit $i \leq n_{Max.Profil}$ notwendig sind:

$$i \cdot \overline{T_G} = \overline{T_{Überwachung}} \leq \overline{T_{Max.Überwachung}} \tag{5.11}$$

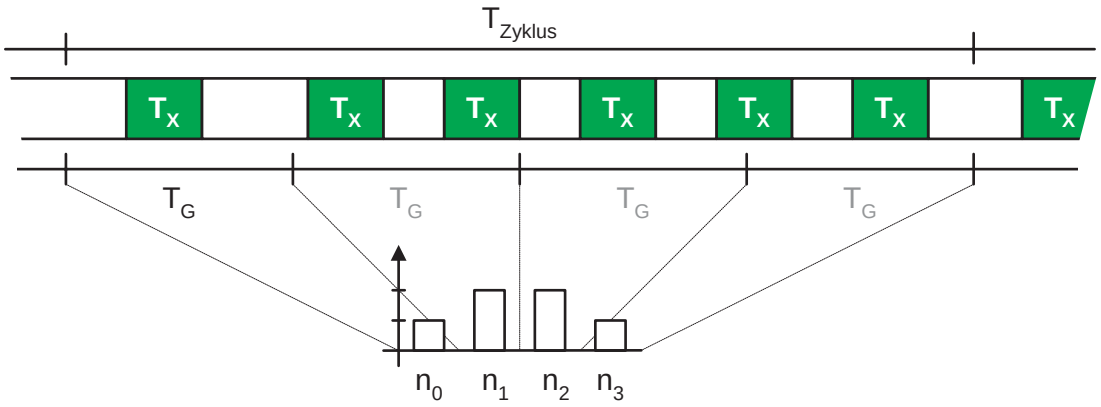

Abbildung 5.28: Erzeugung eines Nutzungsprofils

Abbildung 5.28 zeigt die Granularitätsintervalle T_G. Hierbei wurde die Länge $\overline{T_G}$ von T_G so gewählt, dass die Dauer $\overline{T_{Zyklus}}$ eines fiktiven Zyklus T_{Zyklus} ein ganzzahliges Vielfaches der Dauer von T_G darstellt. Dies ist ein Sonderfall für den nach Gleichung 5.11 zusätzlich gilt, dass $\overline{T_{Überwachung}} = \overline{T_{Zyklus}}$ ist.

Die Nutzungshäufigkeit innerhalb eines Granularitätsintervalls T_G wird mit n bezeichnet. Hierbei entspricht n_0 der *ersten* Nutzungshäufigkeit eines überwachten Moduls. Demzufolge repräsentiert n_0 die Nutzungshäufigkeit innerhalb des ersten Teilintervalls T_G des Zyklus T_{Zyklus}. Die Häufigkeit im nächsten Teilbereich T_G wird als n_1 bezeichnet usw. Mit Beginn des darauf folgenden Zyklusintervalls wird die Nummerierung wieder bei Null gestartet.

T_X ist die Ausführungszeit des überwachten Moduls X (Abbildung 5.28). Die Nutzung eines Moduls wird mit dem Startzeitpunkt von T_X in Verbindung gebracht. Das heißt, dass die Nutzung des Moduls zu dem Intervall T_G gezählt wird, das den Startzeitpunkt bzw. Optimierungspunkt von T_X beinhaltet.

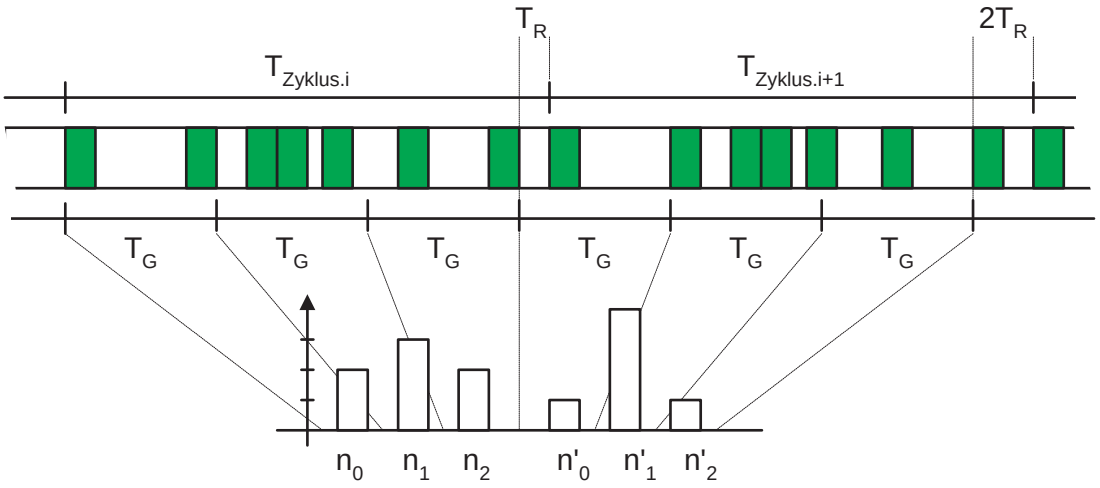

Abbildung 5.29: Problematik der Zyklen- und Überwachungsintervalle

Im Allgemeinen ergibt sich bei der Überwachung mit einer bestimmten Granularität die in Abbildung 5.29 gezeigte Problematik. Das zum Messwert n'_0 gehörige Granularitätsintervall beinhaltet zwar den Startzeitpunkt des Zyklus $T_{Zyklus.i+1}$, jedoch ergibt sich $n'_0 \neq n_0$ durch die ungünstige zeitliche Verschiebung auf Grund des Rests T_R.

Selbst wenn der Zyklusbeginn eindeutig identifiziert wurde und dementsprechend der Startzeitpunkt des ersten Überwachungsintervalls T_G mit dem des Zyklus T_{Zyklus} übereinstimmt, ist in der Regel die Zykluslänge kein ganzzahliges Vielfaches der Länge $\overline{T_G}$ (Abbildung 5.29). Das heißt es bleibt ein Rest der Länge $\overline{T_R}$. Allgemein gilt somit:

$$\overline{T_{Zyklus}} = i \cdot \overline{T_G} + \overline{T_R},\ mit\ i\ \epsilon\ \mathbb{N}\ und\ 0 \leqq \overline{T_R} < \overline{T_G} \tag{5.12}$$

Ein Rest der Länge $\overline{T_R} > 0$ beeinflusst zum einen die Messwerte, wie in Abbildung 5.29 angedeutet, und summiert sich zum anderen mit weiterführender Messung. Diese *Restproblematik* beeinträchtigt sowohl die Zyklenerkennung als auch eine darauf aufbauende Regelung.

5.3.6.1.2 Probleme der exakten Zyklenerkennung: Die Nutzungshistorie ist die Grundlage für die Mustererstellung und die weiterführenden Regelungsmaßnahmen. Um einen Zyklus in der Nutzung eines Moduls zu erkennen, muss dieser anhand der Messdaten der Historie ablesbar sein. Ein Zyklusrest $\overline{T_R} > 0$, wie in Abbildung 5.29 gezeigt, beeinträchtigt die Messungen dahingehend, dass ein n_0 im Allgemeinen vom Messwert n'_0 abweicht. Somit kann ein Zyklusende, bzw. der Anfang des Folgezyklus nicht eindeutig bestimmt werden.

Dementsprechend bietet es sich an, die Überwachungsgranularität T_G so zu wählen, dass sich $\overline{T_R} = 0$ ergibt oder $\overline{T_R} \to 0$ geht. Diese Vorgehensweise scheitert daran, dass $\overline{T_{Zyklus}}$ unbekannt, und somit T_R bzw. $\overline{T_R}$ unbekannt ist. Eine exakte Bestimmung von $\overline{T_{Zyklus}}$ zur Laufzeit ist nur dann möglich, wenn T_G die kleinstmögliche Granularität annimmt. Gibt man $\overline{T_G}$ in Taktzyklen an, würde dies einem $\overline{T_G} = 1\ Taktzyklus$ gleichkommen. Für ein solches T_G vereinfacht sich zwar die Ermittlung der Häufigkeit zu einem 1 Bit-Wert, da stets nur eine einzige Nutzung angezeigt werden kann oder nicht. Im Gegensatz dazu wird die Erzeugung einer Historie unangemessen aufwendig[8]. Zudem kann eine derartige Historie nur dünn besetzt sein, da ungeachtet anderer Faktoren bereits die Ausführungszeit $\overline{T_X}$ eines Moduls X mehrere Taktzyklen bis hin zu mehreren 100 Taktzyklen umfasst. Des Weiteren muss die Rekonfigurationszeit T_{MR} in die Betrachtung mit einbezogen werden, deren Dauer bei einer Größenordnung von 100 und mehr Taktzyklen anzusetzen ist. Demzufolge ist – selbst wenn es theoretisch möglich ist – eine Anpassung von T_G an einen zu ermittelnden Zyklus T_{Zyklus} unbrauchbar. $\overline{T_G}$ muss in Relation zur Rekonfigurationszeit und Ausführungszeit eines Moduls gewählt werden, um redundante Nutzungsdaten weitestgehend zu vermeiden und entsprechend Hardware-Ressourcen zu sparen.

Umgekehrt kann auf eine Betrachtung des Rests T_R verzichtet werden, wenn man davon ausgeht, dass sich nach einer bestimmten Zyklenwiederholung i, $(i \cdot \overline{T_R})\ mod\ \overline{T_G} = 0$, mit $i > 1$, ergeben muss. Der Faktor i ist somit das kleinste gemeinsame Vielfache von $\overline{T_G}$ und $\overline{T_R}$, respektive $\overline{T_{Zyklus}}$. Demzufolge wäre der ermittelte Zyklus der i-fache Grundzyklus T_{Zyklus}. Allgemein kann der ungünstigste Fall, mit $i = \overline{T_G}$(in Taktzyklen) nicht ausgeschlossen werden, was wiederum zu sehr großen Historien, die für eine Zyklenerkennung notwendig sind, führt und abgelehnt werden muss.

[8] Darüber hinaus muss bedacht werden, dass nicht nur eine Historie generiert werden muss.

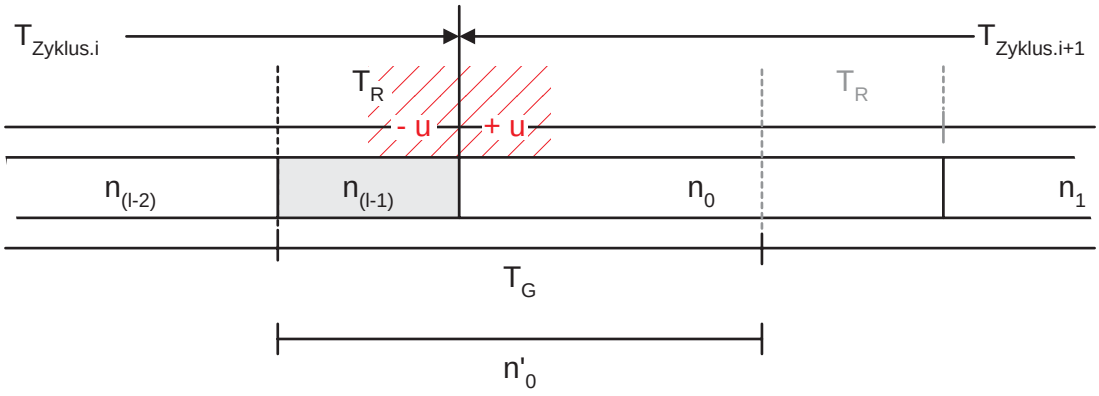

Abbildung 5.30: Auswirkung einer Unschärfe bei Messwerten

5.3.6.1.3 Zyklenerkennung mittels Unschärfe: Abbildung 5.30 zeigt das Ende der i-ten Wiederholung und den Anfang der $(i+1)$-ten Wiederholung eines Zyklus. Betrachtet wird ausschließlich das Nutzungsverhalten eines einzigen Moduls. Der Startzeitpunkt der i-ten Wiederholung ist so gewählt, dass dieser mit dem Startzeitpunkt der Granularitätsintervalle T_G übereinstimmt (nicht in der Abbildung enthalten). Für Zyklus $T_{Zyklus.i}$ ergeben sich $l-1$ Messwerte n_0 bis n_{l-2}, die jeweils genau einem Intervall T_G zuzuordnen sind. Dadurch dass das Zyklusende nicht mit dem Ende eines Intervalls T_G zusammentrifft, wird beim Übergang der i-ten zur $(i+1)$-ten Zykluswiederholung der Messwert n'_0 geliefert. Der Messwert n'_0 beinhaltet den in Abbildung 5.30 als $n_{(l-1)}$ beschrifteten Wert, der sich innerhalb des Rests T_R ergibt und zusätzlich ein Anteil des ursprünglichen n_0 ist. Zur Erkennung des zugrunde liegenden Zyklus muss folglich der Messwert n'_0 als n_0, n'_1 als n_1 usw. identifizierbar sein.

Als Alternative zu den obigen Varianten bleibt die Einführung einer Unschärfe u für die Wiedererkennung von Messwerten. Das heißt ein Messwert n_j wird als Messwert n_i erkannt, falls $n_i - u \leqq n_j \leqq n_i + u$. Abbildung 5.30 skizziert die Auswirkung der Unschärfe u für einen Zyklus mit l Messwerten. Der letzte Messwert n_{l-1} entspricht dem Wert innerhalb des Zyklusrests T_R und beeinträchtigt den Messwert n'_0. Die Unschärfe u ist in Abbildung 5.30 auf die Zeitachse übertragen. Dementsprechend zeigt der schraffierte Bereich die Abweichung an, für die stets n'_0 als n_0 erkannt wird. Ist der Rest T_R außerhalb des Unschärfebereichs von u, so ist die Erkennung von n'_0 als n_0 von den tatsächlich gemessenen Werten abhängig.

Die Werte eines beliebigen n_j liegen im Bereich von 0 bis zu einem Maximalwert n_{Max}, der sich aus der Überwachungsgranularität T_G und der Modulausführungszeit T_X des überwachten Moduls X ergibt:

$$n_{Max} = \begin{cases} T_G \; div \; T_X & falls \; T_G \; mod \; T_X = 0, \\ (T_G \; div \; T_X) + 1 & sonst \end{cases} \tag{5.13}$$

Gleichung 5.13 gilt nur für Module ohne Fließbandverarbeitung. Kommen gepipelinete Module zum Einsatz, ist der theoretische Maximalwert $n_{Max.Pipeline}$ entsprechend von der Ausführungszeit der längsten Pipelinestufe abhängig. Diese Betrachtung ist jedoch nicht praktikabel, da beim Einsatz von Fließbandverarbeitung andere Faktoren, wie z.B. Registerabhängigkeiten und die maximale Anzahl verfügbarer Register eine maßgeblich be-

schränkende Rolle spielen. Um weitestgehend von solchen Implementierungsfaktoren unabhängig zu sein, werden zur Vereinfachung nur Module ohne Pipeline behandelt. Zudem zeigt sich, dass der Wertebereich von n für die letztendlich eingesetzte Zyklenerkennung von nachrangiger Bedeutung ist.

Nach Gleichung 5.13 ist $\{0, 1, \ldots, n_{Max}\}$ der Wertebereich eines Messwertes n. Des Weiteren gilt allgemein (für Module *ohne* Pipeline), dass in einem beliebigen Zeitraum $0 < \overline{T} < \overline{T_G}$ ein Modul X maximal $n(T)$-mal eingesetzt werden kann, mit

$$n(T) = \begin{cases} \overline{T} \; div \; \overline{T_X} & falls \; \overline{T} \; mod \; \overline{T_X} = 0, \\ (\overline{T} \; div \; \overline{T_X}) + 1 & sonst. \end{cases} \tag{5.14}$$

Somit kann, innerhalb eines Restzeitraums T_R, ein Modul nur maximal $n(T_R)$-mal registriert werden. Im Weiteren werden $n(T_R)$ mit $n_{R.Max}$ und ein tatsächlicher Messwert mit n_R bezeichnet.

Dies wiederum bedeutet, dass ein Messwert n'_0, der den Zyklusrest T_R enthält, maximal um den Wert $n_{R.Max}$ vom ursprünglichen n_0 abweichen kann. Bei der Verwendung der Unschärfe u gilt somit, dass für alle $u \geqq n_{R.Max}$ ein Messwert n' als n erkannt wird. Da Messwerte niemals negativ sein können, scheint eine u-Umgebung (Abbildung 5.30) überflüssig, und z.B. $n - u \leqq n'$ ausreichend. Liegt keine Information über die Verteilung der einzelnen Modulnutzungen vor, kann z.B. folgende Situation auftreten: Der Messwert des Zyklusrests T_R, $n_R = n_{R.Max}$ und der ursprüngliche Messwert $n_0 = 1$ (Durch die Startpunktfestlegung muss stets $1 \leqq n_0 \leqq n_{Max}$ gelten). Wobei der gemessene Optimierungspunkt so liegt, dass durch die Verschiebung von T_R, der Anteil von n_0 in n'_0 gleich 0 ist. Dementsprechend ergibt sich der Messwert $n'_0 = n_R + 0 = n_{R.Max}$. Folglich gilt in diesem Fall $n'_0 \geqq n_0$ und für eine Erkennung von n'_0 als n_0, muss $n'_0 \leqq n_0 + u$ zulässig sein. Für die Annahme $n + u \geqq n'$ kann in gleicher Weise gezeigt werden, dass dies nicht ausreichend ist und $n - u \leqq n' \leqq n + u$ verwendet werden muss.

Die Verwendung einer Messwertumgebung u löst die Restproblematik in Verbindung mit der Zyklenerkennung insofern befriedigend, dass nicht nur Zyklen oder i-fache Zyklen, die sich als exakte Vielfache der Überwachungsgranularität T_G darstellen, gefunden werden. Nach wie vielen Wiederholungen ein Zyklus als solcher erkannt wird, hängt vom Zyklusrest T_R und der gewählten Umgebung u ab.

Bevor auf die Wahl eines geeigneten u eingegangen wird, muss festgelegt werden, wann ein Messwert n'_0 als ursprünglicher Messwert n_0 zulässig ist. In diesem Zusammenhang wird folgendes definiert:

Definition 5.3 *Die Messwertzugehörigkeit*

> Die Messwertzugehörigkeit wird in Abhängigkeit von der Größe des Zyklusrests T_R festgelegt. Ist $T_R > \frac{T_G}{2}$, so dominiert der Zyklenrest das Messintervall T_G. Der zugehörige Messwert wird als Zyklusende behandelt. Im Fall $T_R \leqq \frac{T_G}{2}$ gilt der Messwert als erneuter Zyklusstart und wird als n'_0 angesehen. □

Nach Definition 5.3 kann die Wahl einer Umgebung u auf $0 < u \leqq \frac{n_{Max}}{2}$ eingeschränkt werden. Theoretisch sind alle Zahlen aus dem Wertebereich $]0, \frac{n_{Max}}{2}]$ zulässig. Es liegen ausschließlich ganzzahlige Messwerte vor, weshalb nur ganzzahlige u sinnvoll sind.

Allgemein gilt, dass mit großem u eine Zyklenerkennung für den Fall eines vorliegenden Zyklusrests nach wenigen Zyklenwiederholungen möglich ist. Für $u = \frac{n_{Max}}{2}$ kann gezeigt werden, dass bei beliebigem T_R spätestens nach zwei Wiederholungen ein Zyklus als Zyklus erkannt wird. Mit $u = \frac{n_{Max}}{3}$ nach maximal drei Wiederholungen usw. Im Gegensatz dazu wird mit zunehmendem u die Wiedererkennung von Messwerten immer ungenauer. D.h. mit großem u ist im schlechtesten Fall keine Unterscheidung von Messwerten und entsprechend keine zuverlässige Zyklenerkennung mehr möglich. Demzufolge steht die Lösung der Restproblematik einer Zyklenerkennung entgegen.

Darüber hinaus zeigt sich, dass eine allgemeine dynamische Zyklenerkennung grundsätzlich scheitert. Unabhängig von der Wahl eines u sowie ohne Verwendung einer Unschärfe können stets Beispiele konstruiert werden, für die eine bestimmte Messwertfolge fälschlicherweise als Zyklus erkannt wird. Andererseits existieren Zyklen, die auf Grund der Zyklusüberlänge relativ zum maximalen Überwachungszeitraum nicht als solche erkennbar sind. Somit kann zwar der Einsatz einer Unschärfe die Restproblematik soweit vereinfachen, dass wenige Zyklenwiederholungen ausreichen würden, damit ein tatsächlicher Zyklus mit Rest wiedererkannt werden kann. Eine allgemeine Zyklenerkennung, die zuverlässig alle in Betracht kommenden Zyklen identifiziert, ist jedoch nicht möglich.

5.3.6.1.4 Messfehler, sichere Nullstellen und Zyklenverschiebung Dies bedeutet für die Überwachung und Regelung aktiver adaptiver Komponenten, dass weitere Merkmale benötigt werden, um spezielle, für eine Steuerung brauchbare, Zyklen zu ermitteln. In diesem Zusammenhang erweisen sich Nullstellen in zweierlei Hinsicht als hilfreich:

1. Ausschließlich Nullstellen im Nutzungsverhalten von Modulen ermöglichen eine Optimierung.

2. Ein eventuell vorliegender Messfehler, der sich aus einem Zyklusrest T_R ergibt, kann keine Nullstelle *überspringen*. D.h. liegt eine Folge von Nullstellen vor, kann nur die erste Nullstelle von einem etwaigen Messfehler betroffen sein. Alle nachfolgenden Nullstellen müssen erhalten bleiben.

Letzteres gilt immer, da sich nach Gleichung 5.12 bei w-facher Zyklenwiederholung die Auswirkung eines Rests T_R des ursprünglichen Zyklus wie folgt zu einem wiederholt addierten Rest $T^w_{R.gesamt}$ der Länge $\overline{T^w_{R.gesamt}}$ ergibt:

$$\overline{T^w_{R.gesamt}} = w \cdot \overline{T_R},\ mit\ 0 < \overline{T_R} < \overline{T_G} \tag{5.15}$$

$T^w_{R.gesamt}$ repräsentiert die gesamte Verschiebung des Zyklus nach der w-ten Wiederholung in Bezug zum ursprünglichen Zyklus (Abbildung 5.31). Durch die Überwachung mit konstantem Granularitätsintervall T_G lässt sich jedoch nur der ganzzahlige Anteil w', mit $w' = \overline{T^w_{R.gesamt}}\ \ div\ \overline{T_G}$, als Verschiebung registrieren. Die Länge des Rests der w-ten Wiederholung ist

$$\overline{T^w_R} = (w \cdot \overline{T_R})\ mod\ \overline{T_G}. \tag{5.16}$$

Entsprechend trägt T^w_R zu einer Veränderung der Messwerte bei. Das heißt, dass nach w Wiederholungen sich der gemessene Zyklus um w' Messintervalle verschoben zeigt und sich

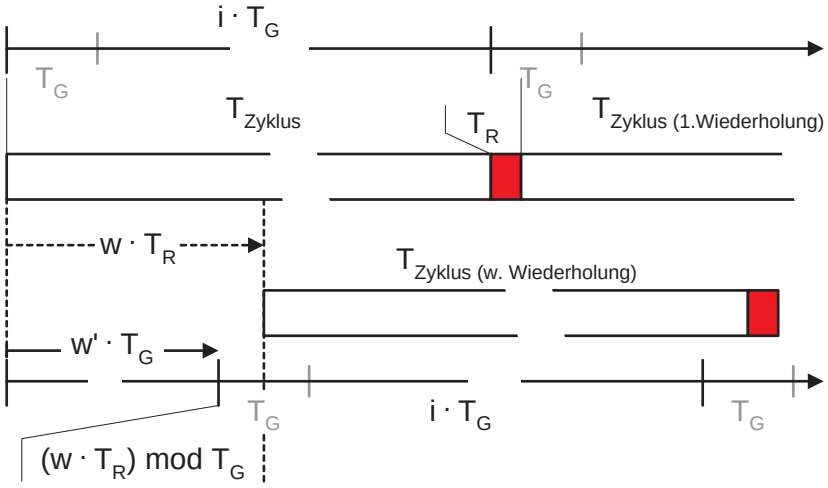

Abbildung 5.31: Zyklenverschiebung nach w-facher Wiederholung

zusätzlich die Messwerte um ein durch T_R^w festgelegtes Maximum von den ursprünglichen Messwerten unterscheiden (Abbildung 5.31).

Nach einer gewissen Anzahl w_{RV} Wiederholungen ist $w_{RV} \cdot \overline{T_R} \geqq \overline{T_G}$ und der ursprüngliche Zyklus stellt sich um ein ganzes T_G nach *rechts* verschoben – wenn der Zyklenverlauf auf einer gedachten Zeitachse von links nach rechts abgetragen wird – dar, wie in Abbildung 5.32 illustriert. Eine *Rechtsverschiebung* tritt wiederholt bei allen ganzen Vielfachen von w_{RV} auf.

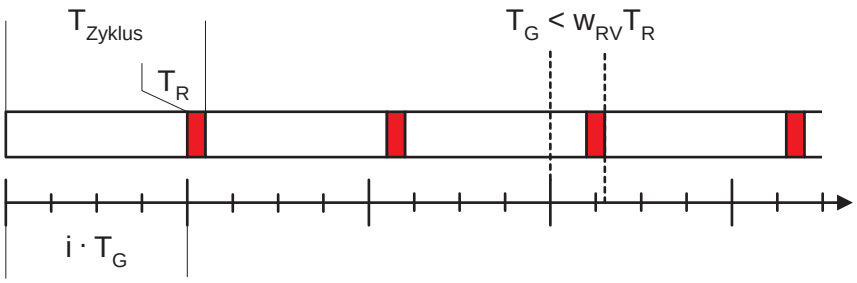

Abbildung 5.32: Rechtsverschiebung bei Zyklen mit Rest

Unabhängig davon gilt nach Gleichung 5.14 und 5.16 für eine beliebige Anzahl von Wiederholungen: Der Restanteil n_R^w des zugehörigen Messwerts n^w ist $n_R^w \leqq n_{R.Max} \leqq n_{Max}$, da $\overline{T_R^w} < \overline{T_G}$ vorliegt. Weiter bedeutet dies, dass bei einer Folge von Nullstellen mit zyklischem Verhalten, unabhängig von der betrachteten Wiederholung, stets nur die erste Nullstelle betroffen sein kann. Alle nachfolgenden Nullstellen sind von Messfehlern unberührt. Die Nullstellenfolge tritt zwar in Bezug zur ursprünglichen Zyklusmessung zeitlich verschoben auf, bleibt aber, mit Ausnahme der ersten Nullstelle, die durch n_R^w beeinflusst ist, immer erhalten. Dies gilt ausschließlich für Nullstellen. Für beliebige Messwerte $n_x^w \neq 0$, bzw. Folgen von Messwerten ungleich Null, existiert eine Abschätzung für die maximale Abweichung vom

ursprünglichen Messwert n_x. Darüber hinaus kann keine allgemeine Aussage über einzelne Messwerte getroffen werden.

Bei einer Folge von mindestens zwei Nullstellen bleiben stets die zweite Nullstelle und alle nachfolgenden erhalten. Dies ist unabhängig von einem eventuellen Zyklenrest. Treten Folgen von mindestens zwei Nullstellen auf, werden diese ab der zweiten als *sichere Nullstellen* bezeichnet.

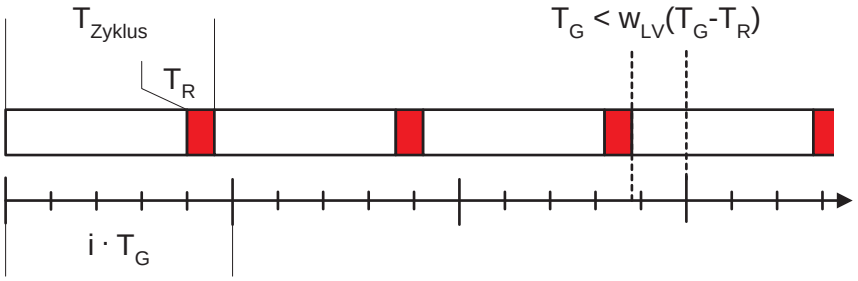

Abbildung 5.33: Linksverschiebung für dominanten Zyklusrest

Behandelt man das Nutzungsverhalten eines einzelnen Moduls unter Berücksichtigung der Startpunktbedingung sowie der Annahme, dass ein Zyklenrest stets dem nächsten Zyklenstart zugeordnet wird, kann sich ausschließlich eine Rechtsverschiebung ergeben. Jedoch wurde allgemein eine Messwertzugehörigkeit definiert, der zufolge ein Zyklenrest T_R als eigenständiger Messwert angesehen wird, falls $\overline{T_R} > \frac{\overline{T_G}}{2}$ vorliegt. In Abbildung 5.33 erkennt man, dass sich somit eine Zyklusdauer folgendermaßen beschreiben lässt:

$$\overline{T_{Zyklus}} = i \cdot \overline{T_G} - (\overline{T_G} - \overline{T_R}),\ mit\ i\ \epsilon\ \mathbb{N}\ und\ \frac{\overline{T_G}}{2} < \overline{T_R} < \overline{T_G} \tag{5.17}$$

Obwohl Gleichung 5.17 einfach in Gleichung 5.12 überführbar ist, müssen im Zusammenhang mit der Zyklenbehandlung beide Fälle gesondert betrachtet werden. Abbildung 5.33 verdeutlicht eine sich hierbei ergebende *Linksverschiebung* des Zyklus in Bezug zu den Messintervallen T_G, die nach einer gewissen Anzahl von Wiederholungen auftritt und von $(\overline{T_G} - \overline{T_R})$ abhängt.

Demzufolge muss Gleichung 5.16 unter Einbeziehung der Größe von T_R erweitert werden:

$$\overline{T_R^w} = \begin{cases} (w \cdot \overline{T_R})\ mod\ \overline{T_G} & falls\ \overline{T_R} \leqq \frac{\overline{T_G}}{2} \\ (w \cdot (\overline{T_G} - \overline{T_R}))\ mod\ \overline{T_G} & falls\ \frac{\overline{T_G}}{2} < \overline{T_R} < \overline{T_G} \end{cases} \tag{5.18}$$

Unabhängig vom Zyklusrest T_R und der betrachteten Wiederholung w gilt $\overline{T_R^w} < \overline{T_G}$ nach Gleichung 5.18. Entsprechend den Ausführungen zur Rechtsverschiebung kann bei einer Folge von Nullstellen wiederum nur die äußerste einen von Null verschiedenen Messwert annehmen. Des Weiteren kann es entweder zu einer Rechts- oder einer Linksverschiebung kommen. Das gleichzeitige oder abwechselnde Auftreten beider ist nicht möglich. Die Auswirkung einer Linksverschiebung auf Nullstellen, verhält sich entgegengesetzt der Rechtsverschiebung. Das bedeutet, bei einer Linksverschiebung kann die letzte Nullstelle einer Folge von Nullstellen angetastet werden. Alle Vorgänger bleiben stets Nullstellen. Da vorab nicht bekannt ist, welche Verschiebung vorliegt, müssen beide Arten berücksichtigt werden.

Definition 5.4 *Sichere Nullstelle*

Eine sichere Nullstelle, die sowohl Links- als auch Rechtsverschiebungen beachtet, liegt dann vor, wenn diese von zwei weiteren Nullstellen umgeben ist. □

Dies bedeutet in Bezug auf die Messwerte zur Erzeugung einer Nutzungshistorie, dass eine sichere Nullstelle existiert, wenn wenigstens drei Nullstellen in Folge anzutreffen sind. Liegt eine längere Folge vor, stellen alle inneren Nullstellen, sichere Nullstellen dar.

5.3.6.2 Mustererzeugung und spezielle Zyklenerkennung

Im Zusammenhang mit der Modulnutzung ist, wie bereits erwähnt, keine allgemeine Zyklenerkennung notwendig. Der Einsatz einer Regelung basiert auf einem zyklischen Auftreten von Nullstellen. Das heißt, dass die genauen Messwerte n nur von nachrangiger Bedeutung sind. Für die Erkennung von Nullstellen, reicht prinzipiell die Feststellung, ob innerhalb eines Überwachungsintervalls T_G ein Modul mindestens einmal genutzt wurde oder nicht.

Auch für die spezielle Zyklenerkennung können Messreihen konstruiert werden, die eine Erkennung als Zyklus zur Folge haben, wobei es sich nicht um einen Zyklus handelt. Dies ist ein grundsätzlicher Schwachpunkt einer, wie auch immer gearteten, dynamischen Zyklenerkennung. Die Verwendung von Mustern erlaubt im Gegensatz zu den in Abschnitt 3.4 vorgestellten statistischen Methoden eine sehr einfache und effiziente Behandlung von Zyklen – vorausgesetzt diese liegen vor –, die sich zudem gut für eine ressourcensparende Umsetzung in Hardware eignet.

Der kleinste Zyklus der erkannt werden kann, besteht, unter Berücksichtigung einer einzigen sicheren Nullstelle, aus mindestens vier Messwerten n_i, mit $i \epsilon \{0, 1, 2, 3\}$, wobei $n_0 > 0$ (Startpunktbedingung) und alle weiteren $n_i = 0$ gelten muss. Da die Messung erst mit dem Auftreten einer Nutzungsmöglichkeit eines Moduls beginnt, ist der erste sich ergebende Messwert stets von Null verschieden. Für alle folgenden Messwerte gibt es keine Beschränkungen.

5.3.6.2.1 Prinzipieller Ablauf der Mustererzeugung Die Bedingungen für einen kleinstmöglichen Zyklus bedeuten für ein daraus ableitbares Muster, dass frühestens nach vier Messintervallen T_G ein sinnvolles Muster vorliegen kann, oder allgemein, erst nachdem mindestens drei Nullstellen in Folge gemessen wurden. Ist i die Anzahl der Messintervalle, wobei $i \geqq 4$ gilt, die zur Erkennung der ersten Lücke führen, werden alle Messwerte n_0 bis $n_{(i-1)}$ in einem Muster abgelegt. Die einzelnen Mustereinträge m sind Elemente aus $\{0, X, G, ?\}$. 0 bezeichnet eine sichere Nullstelle. X beschreibt einen beliebigen Wert. G repräsentiert einen Wert größer Null, ? steht für einen noch nicht bestimmten Musterwert. Bei einer Zyklenerkennung werden Musterwerte $m = ?$ ignoriert. Die Mustererzeugung arbeitet folgendermaßen:

- Der erste Mustereintrag m_0 wird stets mit G belegt, da ein Zyklusbeginn und der zugehörige Messwert größer Null gefordert sind. Dies folgt aus der Startbedingung eines Zyklus.
- Alle folgenden Mustereinträge m_i ergeben sich aus der Betrachtung von $n_{(i-1)}$, n_i und $n_{(i+1)}$. Liegt der Messwert $n_{(i+1)}$ noch nicht vor, wird $m_i = ?$ gesetzt. Ansonsten gilt

$m_i = 0$ falls $n_{(i-1)} = n_i = n_{(i+1)} = 0$. Sowie $m_i = X$, falls ein Messwert von Null verschieden ist.

Formal besteht folgende Vorschrift zur Erzeugung der Mustereinträge m_i, bei L vorliegenden Messwerten, n_0 bis n_{L-1} und $L \geqq 4$:

$$m_i = \begin{cases} G & \textit{für } i = 0, \\ 0 & \textit{für } 1 \leqq i \leqq (L-2) \textit{ und } n_{(i-1)} = n_i = n_{(i+1)} = 0, \\ X & \textit{für } 1 \leqq i \leqq (L-2) \textit{ und } n_{(i-1)} \neq 0 \textit{ oder } n_i \neq 0 \textit{ oder } n_{(i+1)} \neq 0, \\ ? & \textit{für } i = (L-1) \end{cases} \tag{5.19}$$

Nachdem eine sichere Nullstelle erkannt wurde, liegen $L \geqq 4$ Messwerte n vor. Das zugehörige Muster enthält ebenfalls L Einträge, für die nach obiger Vorschrift gilt: $m_0 = G$ sowie $m_{L-1} = ?$. Zudem muss genau ein $m_x = 0$, mit $2 \leqq x \leqq (L-2)$, vorliegen. Alle übrigen Musterwerte y weisen $m_y = X$ auf, wobei $1 \leqq y \leqq (L-2)$ und $x \neq y$ gilt.

5.3.6.2.2 Zyklenerkennung mit Hilfe von Mustern Wurde ein vorläufiges Muster der Länge L mit genau einer Nullstelle gefunden, werden alle nachfolgenden Messwerte n_l, mit $l \geqq L$, mit dem angenommenen Pendant im Muster verglichen. Das heißt n_L wird mit m_0 in Relation gesetzt, n_{L+1} mit m_1 usw. Dies bedeutet weiter, dass, falls ein Zyklus vorliegt, der durch das Muster der Länge L beschreibbar ist, frühestens nach der ersten Wiederholung, somit nach $2 \cdot L$ Messungen respektive Granularitätsintervallen erkannt wird. Mit Hilfe des bisherigen Musters wird ein Messwert n'_x als ein bereits vorliegender Messwert n_x interpretiert, falls anhand des zugehörigen Musterwertes m_x, $n'_x \sim n_x$ festgestellt wird.

$$n'_x \sim n_x \textit{ gilt dann, wenn } \begin{cases} m_x = X, \\ n'_x = 0 \textit{ und zugleich } m_x = 0, \textit{ oder} \\ n'_x > 0 \textit{ und zugleich } m_x = G \end{cases} \tag{5.20}$$

Nach Vorschrift 5.20 ergibt sich der prinzipielle Ablauf der Zyklenerkennung wie folgt:

Ein Muster der Länge L mit genau einer sicheren Nullstelle liegt vor. Das bedeutet, dass genau L Messwerte, n_0 bis n_{L-1} in der Nutzungshistorie eingetragen sind. Für den darauf folgenden Messwert n_L wird mittels Vorschrift 5.20 die Ähnlichkeit zu Messwert n_0 untersucht, bzw. festgestellt, ob n_L zum Mustereintrag m_0 passt. Zeigt sich $n_L \sim n_0$ wird nach Erhalt des Messwerts n_{L+1} ebenso verfahren usw. Wird eine Abweichung vom Muster festgestellt, bevor $2 \cdot L$ Messwerte vorliegen, führt dies zu einer Erweiterung des Ausgangsmusters um einen neuen Musterwert m_L. Dieser Musterwert wird entsprechend Vorschrift 5.19 generiert. Ein neues Muster der Länge $L+1$ liegt vor. Der Musterwert m_0 wird diesmal mit Messwert n_{L+1} getestet, m_1 mit n_{L+2} usw. Folglich muss sich nach $2 \cdot (L+1)$ ein zyklisches Verhalten ergeben, oder das Muster wird erneut um einen Wert verlängert. Liegt nun ein Zyklus vor, zeigt sich dies nach $2 \cdot (L+2)$ Messungen usw.

Diese Vorgehensweise erlaubt es, zyklisch auftretende Nullstellen im Nutzungsverhalten eines Moduls zu erkennen. Da hierfür nur die Startpunktbedingung und sichere Nullstellen Verwendung finden, kann die Restproblematik bei der Zyklenerkennung, und nur bei der Zyklenerkennung, vernachlässigt werden.

5.3.6.2.3 Auswirkung der Restproblematik Die beschriebene Methode eignet sich, um zyklisch auftretende Nullstellen zu identifizieren und diese entsprechend im weiteren Verlauf vorherzusagen. Die Restproblematik führt jedoch nach einer bestimmten Anzahl von Zyklenwiederholungen zu der bereits angesprochenen Links- oder Rechtsverschiebung. Demzufolge wird ein richtig erkanntes Muster irrtümlich für falsch gehalten, da sich entsprechend einer Links- oder Rechtsverschiebung, das Muster entweder als einen Musterwert zu lang oder zu kurz erweist.

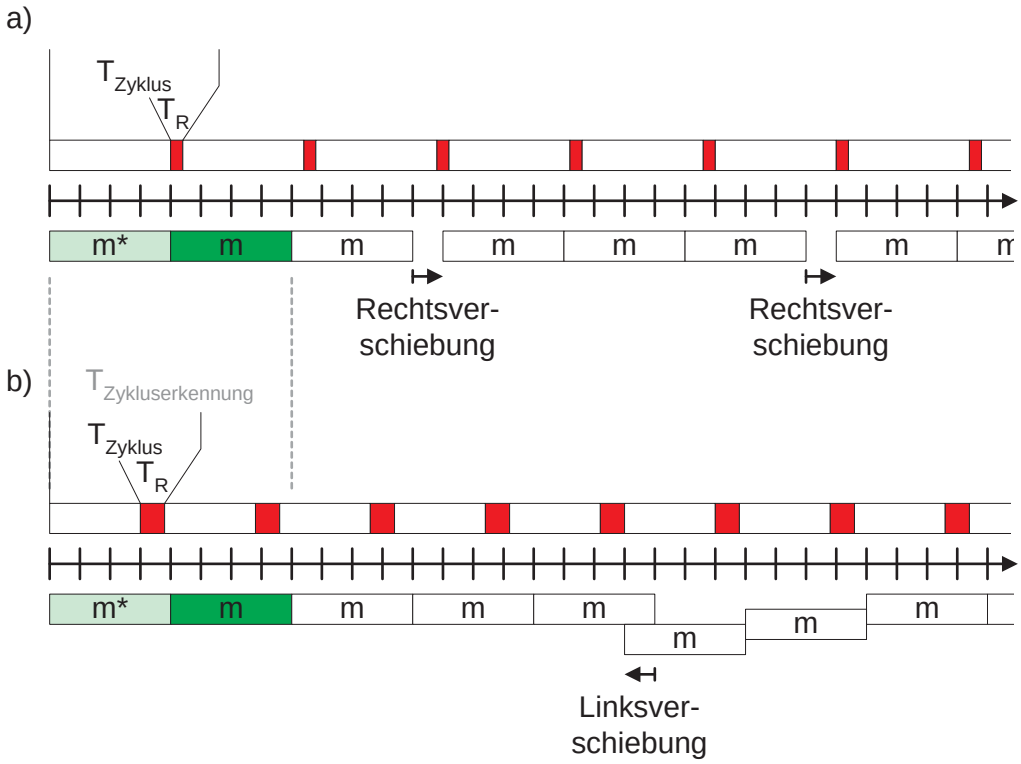

Abbildung 5.34: Auswirkung von Rechts- und Linksverschiebung beim Einsatz von Mustern

In jedem Fall führt eine Verschiebung dazu, dass sich eine sichere Nullstelle des Musters nicht mit den erhobenen Messwerten deckt. Die Verschiebung des Zyklenverhaltens ist insofern problematisch, da auf Basis gewonnener Muster die Vorhersage künftiger Nullstellen beruht. Wird eine Musterverschiebung erst dann erkannt, wenn eine sichere Nullstelle erwartet wurde, sind sämtliche darauf gegründeten Regelungsmaßnahmen hinfällig. Darüber hinaus ist nicht auszuschließen, dass sich durch bereits eingeleitete Rekonfigurationen das Laufzeitverhalten einer Anwendung verschlechtert.

In diesem Zusammenhang skizziert Abbildung 5.34 den Idealfall: Eine Rechts- und Linksverschiebung wird bereits zu Beginn bzw. Ende eines Zyklus erkannt und entsprechend eine Verschiebung des zugehörigen Musters durchgeführt. Zusätzlich ist die Zeit für die Zyklenerkennung in Abbildung 5.34 angedeutet, wobei m^* das vorläufig angenommene Muster repräsentiert, das sich nach der ersten Zyklenwiederholung als gültiges Muster m erweist.

Der Vorteil einer musterbasierten Erkennung zyklisch auftretender Nullstellen liegt im geringen Aufwand. Es muss jeweils nur ein Musterwert m mit einem Messwert n verglichen

werden. Da anhand eines einzelnen Messwertes nur die Identifikation von sicheren Nullstellen möglich ist, existiert diesbezüglich keine allgemeine Lösung, die zu einer vorzeitigen Verschiebungsbehandlung eingesetzt werden kann. Selbst die Startpunktbedingung, mit $n_0 \geqq 1$ bzw. $m_0 = G$, garantiert die Verschiebungserkennung zu Beginn bzw. Ende eines Zyklus nicht. Es ist immer möglich einen Zyklus so zu konstruieren, dass die Startpunktbedingung solange aufrechterhalten bleibt, bis eine sichere Nullstelle von einer auftretenden Verschiebung betroffen ist.

Eine allgemein gültige Verschiebungserkennung ist wiederum ausschließlich durch Nullstellen realisierbar. Hierbei muss jeweils die äußerste sichere Nullstelle herangezogen werden. Bei einer Rechtsverschiebung, ist diese erkennbar, wenn die erste sichere Nullstelle im Muster auf einen von Null verschiedenen Messwert trifft. Umgekehrt zeigt sich eine Linksverschiebung bei der letzten sicheren Nullstelle. Damit bezüglich der Regelung, keine Konflikte entstehen, müssen die äußeren sicheren Nullstellen zur Verschiebungserkennung reserviert werden. Demzufolge ist die obere Feststellung, dass ein minimales, geeignetes Muster nach vier Messwerten vorliegen kann, dahingehend zu erweitern, dass mindestens sechs Messwerte benötigt werden, um eine sichere Regelung zu ermöglichen. Hierbei muss durch die Startpunktbedingung $n_0 > 0$ bzw. $m_0 = G$ gelten. Alle weiteren Messwerte n_1 bis n_5 müssen jeweils Null sein. Dementsprechend ergibt sich ein Muster $M = [GX000?]$, wobei $m_2 = 0$ für die Erkennung von Rechtsverschiebungen, bzw. $m_4 = 0$ für Linksverschiebungen, im Zusammenhang mit einer Regelung beachtet werden muss. Somit bleibt nur m_3 als Möglichkeit eines sicher prognostizierbaren Regeleingriffs bestehen. Sowohl die erste sichere Nullstelle als auch die letzte sichere Nullstelle werden zur Verschiebungserkennung benötigt. Dies gilt allgemein für beliebige Muster mit mehr als fünf Musterwerten.

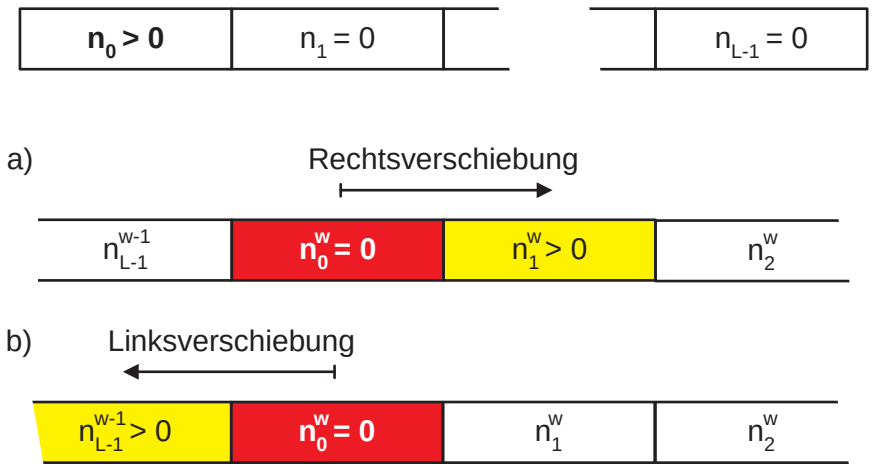

Abbildung 5.35: Frühzeitige Verschiebungserkennung auf Grund von Startpunktsonderfällen

5.3.6.2.4 Sonderfälle der Startpunktbedingung Ein Regeleingriff kann ausschließlich auf sicheren Nullstellen erfolgen. Dadurch, dass sich ein Muster im Bezug zum korrespondierenden Zyklus verschieben kann, werden die jeweils äußeren Nullstellen eines Musters zur Verschiebungserkennung benötigt. Diesbezüglich existieren Sonderfälle in Verbindung mit

der Startpunktbedingung eines Musters, die eine vorzeitige Verschiebungserkennung ohne sichere Nullstellen erlauben.

Abbildung 5.35 zeigt eine Messreihe der Länge L (oben) und die relevanten Messwerte n_0 bis n_{L-1}. Im Beispiel repräsentiert $n_{L-1} = 0$ einen Messwert, der den Zyklusrest $\overline{T_R} > \frac{\overline{T_G}}{2}$ enthält. Für $\overline{T_R} \leqq \frac{\overline{T_G}}{2}$ muss zusätzlich der Messwert $n_{L-2} = 0$ gelten. Durch die Startpunktbedingung ist $n_0 > 0$. Des Weiteren soll $n_1 = 0$ vorliegen.

Nachdem dieser Zyklus erkannt wurde, liegt nach w Wiederholungen entweder (Abbildung 5.35 a) eine Rechtsverschiebung oder (Abbildung 5.35 b) eine Linksverschiebung vor. Dadurch, dass die Messwerte $n_{L-1} = n_1 = 0$ sind, kann der Messwert n_0 nicht von etwaigen Verschiebungen der umgebenden Messwerte verändert werden. Nur für n_0 ist es möglich, die angrenzenden Messwerte zu beeinflussen. Für den Fall, dass eine Verschiebung – unabhängig von der Richtung – um ein ganzes Messintervall auftritt, muss sich $n_0 = 0$ ergeben. Dies widerspricht der Startpunktbedingung, mit $n_0 > 0$ und erlaubt eine zuverlässige und frühzeitige Erkennung von Verschiebungen ohne die Einbeziehung von sicheren Nullstellen.

Die Verschiebungserkennung ist ausschließlich im Zusammenhang mit der Startpunktbedingung respektive $n_0 > 0$ zulässig, da die Startpunktbedingung zur Mustererzeugung benutzt wird. Für andere Messwerte n_1 bis n_{L-1} ist dieses Verhalten nicht zwingend ein Hinweis auf eine Verschiebung um ein ganzes Messintervall.

5.3.6.3 Nutzung zyklischen Nullstellenverhaltens

Nachdem allgemein die Zyklenerkennung sowie die Grundsätze der musterbasierten Zyklenerkennung behandelt wurden, widmet sich dieser Abschnitt der Überwachung mehrerer Module und den sich daraus ergebenden Problemen und Regelungsmöglichkeiten.

5.3.6.3.1 Überwachung mehrerer Module: Für eine Regelung ist der Einsatz mindestens zweier Module während der betrachteten Anwendungsrechenzeit – vornehmlich einer Zeitscheibe – erforderlich. Allgemein wird der konkurrierende Einsatz von m Modulen in k konfigurierbaren Bereichen, wobei im Weiteren nur der Fall $m > k$ Beachtung findet, betrachtet. In diesem Zusammenhang ist anzunehmen, dass bereits k der insgesamt m Module in einer adaptiven Komponente eingelagert sind. Demzufolge müssen k Nutzungshistorien, respektive Muster, der einsatzbereiten Module, sowie $(m - k)$ Profile und Muster für die mögliche Nutzung der übrigen Module erstellt werden.

Diesbezüglich existieren zwei Sonderfälle, die keine Erstellung von Nutzungsprofil und Muster benötigen:

1. Wurde ein Modul vom Anwendungsprogrammierer, Übersetzer oder dem Betriebssystem, z.B. zur Einhaltung von Laufzeitkriterien, entsprechend markiert, entfällt die Überwachung dieses Moduls. Solche Module müssen stets eingelagert sein und sind im Allgemeinen nicht in die Regelung mit einbezogen.

2. Module die selten oder nur ein einziges Mal, z.B. für die Beschleunigung von Initialisierungsvorgängen, eingesetzt werden, können durch Konfigurationsinstruktionen zu den jeweiligen Nutzungszeitpunkten manuell verfügbar gemacht werden. Hierbei sollten diese so gekennzeichnet sein, dass keine Überwachung stattfindet und diese nach ihrem Einsatz bei Bedarf wieder aus der adaptiven Komponente entfernbar sind.

Beide Sonderfälle bedürfen Information, die entweder bereits dem Anwendungsprogrammierer bekannt ist oder durch den Übersetzer oder das Betriebssystem ermittelt wurde. Unabhängig von der Informationsquelle sollten solche Informationen, falls vorhanden, genutzt werden, um z.B. ein sicheres *worst-case* Laufzeitverhalten zu garantieren (Vgl. auch Abschnitt 5.2).

Allgemein bleibt die Situation mit m Modulen, die sich k konfigurierbare Bereiche teilen, wobei $k \geqq 1$ sein muss, und $m > k$ vorliegt, bestehen. Wann und wie oft ein bestimmtes Modul genutzt wird, ist zu Beginn der Rechenzeit einer Anwendung nicht bekannt. Zudem muss, wie bereits erwähnt, ein zyklisches Nutzungsverhalten feststellbar sein, um mehr als k von m Modulen während einer Zeitscheibe einsetzen zu können. Andernfalls kann ausschließlich mit Hilfe von Häufigkeiten eine etwaige Rekonfiguration vorgenommen werden (Details siehe Unterabschnitt 5.3.4.3).

Da zum Anwendungsbeginn keine Information über die zu observierenden Module vorliegt, muss mit Auftreten der ersten Modulnutzung bzw. Nutzungsmöglichkeit die Aufzeichnung aller Nutzungsprofile gestartet werden. Diese Vorgehensweise dient der Synchronisation der zu erwartenden Muster. Nur falls die Nutzungsprofilerstellung aller Module zum gleichen Zeitpunkt beginnt, kann gewährleistet werden, dass die resultierenden Muster ohne zusätzliche Veränderung untereinander korreliert werden können. Andererseits bedeutet dies, dass für alle Module, die innerhalb des ersten Messintervalls keine Nutzung aufweisen, die Startpunktbedingung nicht erfüllt ist. Hierbei wird das Muster des Moduls, welches die Überwachung initiiert hat, als *Führungsmuster* bezeichnet. Die Startpunkte aller anderen Muster werden jeweils dann unabhängig voneinander festgelegt, wenn die erste Nutzung bzw. Nutzungsmöglichkeit registriert wird. Dies ist notwendig, da nicht sichergestellt ist, dass das als Führungsmuster angenommene Muster, tatsächlich als Referenz für die weitere Regelung verwendet werden kann. Zum Beispiel ist die einmalige Nutzung dieses Moduls, während der betrachteten Laufzeit zulässig. Somit ist dieses Muster für die Regelung unbrauchbar. Die Erkennung dieser Situation führt zur Festlegung eines neuen Führungsmusters. Das Muster mit dem frühesten Startpunkt wird ausgewählt.

Dies führt im Allgemeinen dazu, dass der Startpunkt des später ausgesuchten Führungsmusters nicht exakt mit der ersten Nutzungsmöglichkeit zusammenfällt. Diesbezüglich sind Abweichungen im Bereich von 0 und $< \overline{T_G}$ möglich. Hierbei kann eine weitere Situation auftreten, die eine zusätzliche Variante einer Linksverschiebung darstellt. Diese ist entsprechend der obigen allgemeinen Ausführungen zu behandeln und wird deshalb nicht genauer erläutert. Das Phänomen, bei dem der eigentliche Optimierungspunkt, der den Startpunkt darstellt, nicht exakt mit dem Beginn des ersten Granularitätsintervalls T_G zusammenfällt, wird als *relativierte Startpunktbedingung* bezeichnet, da dennoch stets der zugehörige Messwert größer Null ist. Insgesamt bleibt sowohl die Vorgehensweise bei der weiteren Mustererzeugung, der Zyklenerkennung sowie der Verschiebungserkennung für die relativierte Startpunktbedingung unverändert.

Darüber hinaus muss ein geeignetes Granularitätsintervall T_G gefunden werden, damit die Muster verschiedener Module untereinander in Beziehung gesetzt werden können.

5.3.6.3.2 Wahl der Messintervalle: Bisher wurden zwei Probleme in Bezug auf passende Messintervalle angesprochen:

1. Unter Berücksichtigung des Ressourcenaufwands sowie der Aussagekraft einer sich ergebenden Nutzungshistorie, ist ein Messintervall T_G relativ zur Modulausführungszeit T_X eines Moduls X und der Modulrekonfigurationszeit $T_{MR}(X)$ zu wählen (Vgl. Abschnitt 5.3.6.1.2).

2. Damit Muster verschiedener Module untereinander vergleichbar sind, muss für die Messungen dieser Module jeweils ein – zumindest – äquivalentes Messintervall T_G Verwendung finden (siehe vorhergehenden Abschnitt 5.3.6.3.1).

Da sich die Ausführungszeiten und Rekonfigurationszeiten einzelner Module stark unterscheiden können, ist die Forderung, ein Messintervall in Abhängigkeit dieser Größen zu wählen im direkten Widerspruch dazu zu sehen, dass die Messintervalle der Module äquivalent sein sollen.

Hierbei ist es nicht möglich allgemein gültige Aussagen zu treffen, da die Rekonfigurations- und Ausführungszeiten zum einen von den verwendeten bzw. erzeugten Modulen und zum anderen von zusätzlichen Implementierungsaspekten abhängen. Diesbezüglich wird eine Gruppierung unterschiedlicher Module vorgeschlagen, die in erster Linie den Rekonfigurationszeiten der Module Rechnung trägt. Module mit Rekonfigurationszeiten in vergleichbarer Größenordnung bilden jeweils eine Gruppe. Für eine Gruppenbildung ist zu beachten, dass das Modul mit der längsten Rekonfigurationszeit, und dementsprechend das Modul mit dem längsten Bitstream, maßgeblich bei der Wahl des zugehörigen Messintervalls T_G ist. Bezeichnet man dieses Modul mit X, sollte T_G so gewählt werden, dass $\overline{T_{MR}}(X) + \overline{T_X} < \overline{T_G} + \overline{T_{\setminus X}}$ gilt. Diesbezüglich muss eine Rekonfiguration zu Beginn einer erwarteten sicheren Nullstelle erfolgen und kann im ungünstigsten Fall, dass die Ausführung von X zu Beginn des darauf folgenden Messintervalls gefordert ist, dennoch keine Laufzeitverlängerung zur Folge haben. Zusätzlich ist zu Berücksichtigen, ob ein Modul Y, mit $\overline{T_{MR}}(Y) \leqq \overline{T_{MR}}(X)$ und $\overline{T_{MR}}(Y) + \overline{T_Y} > \overline{T_{MR}}(X) + \overline{T_X}$, in der gleichen Gruppe existiert. Kann ein solches Modul ausgemacht werden, ist T_G entsprechend anzupassen.

Darüber hinaus ist eine Gruppierung von Modulen nur dann sinnvoll, wenn mehr als ein konfigurierbarer Bereich vorliegt. In diesem Zusammenhang kann es sich anbieten, verschiedene konfigurierbare Bereiche mit unterschiedlicher Größe, in Anlehnung an die ermittelten oder zu erwartenden Modulgruppen, innerhalb einer einzelnen adaptiven Komponente zuzulassen.

Zusätzlich sollten die Messintervalle einzelner Modulgruppen einfach ineinander überführbar sein. Dies bietet einer Regelung die Möglichkeit, über Modulgruppen hinweg Entscheidungen zu treffen. Ein geeigneter Zusammenhang zwischen den Messintervallen, der zudem effizient in Hardware behandelt werden kann, ist die Verwendung Vielfacher von zwei dergestalt, dass das größtmögliche Messintervall $\overline{T_{G.Max}} = 2n \cdot \overline{T_{G.Min}}$, mit $n \epsilon \mathbb{N}$, gilt. Dementsprechend lassen sich z.B. für $n = 1$ jeweils zwei Mess- oder Musterwerte des kürzeren Messintervalls zu einem einzelnen Mess- oder Musterwert des längeren Intervalls umwandeln usw.

5.3.6.3.3 Musterwerterweiterung und Auflösung von Konkurrenzsituationen:

Entscheidend für die Regelung sind die sicheren Nullstellen der generierten Muster. Können hierbei zwei oder mehrere Muster unterschiedlicher Module so übereinander gelegt werden, dass stets sichere Nullstellen auf genutzte Bereiche eines oder mehrere Module treffen, eignen sich diese Module für den Einsatz in einem einzigen rekonfigurierbaren Bereich. In diesem

Zusammenhang treten Situationen auf, die eine konfliktfreie Überlagerung nicht zulassen. Zwei grundsätzliche Konstellationen sind allgemein möglich, die zur Vereinfachung anhand von zwei Modulen X und Y behandelt werden, aber für eine beliebige Anzahl von Modulen gelten:

1. Sowohl Modul X als auch Modul Y werden in den gleichen Messintervallen genutzt. Eine Regelung auf Basis der zugehörigen Muster ist nicht möglich. Als Entscheidungsgrundlage muss die Gewinnberechnung aus Abschnitt 5.3.4.3 herangezogen werden. Entweder Modul X oder Modul Y kommt zum Einsatz.

2. Es existiert mindestens ein Messintervall, das den Einsatz beider Module anzeigt. Alle übrigen Messintervalle lassen eine konfliktfreie Nutzung beider Module zu. Dementsprechend muss entschieden werden, welches Modul innerhalb der betroffenen Messintervalle eingesetzt werden soll. Hierbei bietet sich wiederum die oben erwähnte Gewinnberechnung an, wobei nur die Häufigkeit der Module in den problematischen Messintervallen beachtet werden darf. Das bedeutet, dass für diese Messintervalle entweder Modul X oder Y zur Verfügung gestellt wird. Die übrigen Optimierungsmöglichkeiten können voll ausgeschöpft werden.

Letztere Vorgehensweise birgt jedoch eine Schwierigkeit. Die zugehörigen Muster der Module X und Y zeigen zwar, dass eine Konfliktsituation vorliegt, die Information über die Häufigkeit ist aber nicht in den Mustern verfügbar. Demzufolge muss die Häufigkeit auf Grund der zugehörigen Nutzungshistorie festgestellt werden. Um den zugehörigen Messwert der Historie zu ermitteln, muss anhand der Position der Musterwerte, die einen Konflikt aufweisen, rückwärts gesucht werden. Eine Suche kann z.B. durch eine vorangegangene Verschiebung erschwert werden. Darüber hinaus wird nicht zugesichert, dass die Nutzungshistorie die benötigten Häufigkeiten verfügbar hat. Es kann nur eine beschränkte Anzahl von Messwerten abgelegt werden. Somit ist es möglich, dass ein benötigter Messwert aus Platzgründen bereits durch einen neuen Messwert überschrieben wurde. Insgesamt bietet es sich dementsprechend an, die Messwerte in die Mustererzeugung mit einzubeziehen. Folglich lässt sich eine Messwertsuche vermeiden und es ist keine zusätzliche Pflege älterer Messwerte der zugehörigen Nutzungshistorie notwendig.

Dies führt zu einer Erweiterung der Mustererzeugung, die in Abschnitt 5.3.6.2 eingeführt wurde:

$$m_i = \begin{cases} 0 & \textit{für } 1 \leqq i \leqq (L-2) \textit{ und } n_{(i-1)} = n_i = n_{(i+1)} = 0, \\ ? & \textit{falls } n_{(i+1)} \textit{ nicht verfügbar} \\ n_i & \textit{sonst} \end{cases} \tag{5.21}$$

Der Musterwert m_0 wird nicht eigenständig behandelt, da durch die Festlegungen in Bezug auf ein Führungsmuster der zugehörige Messwert $n_0 > 0$ vorliegen muss. Dies gilt auch, falls ein neues Führungsmuster gewählt wird (Vgl. Abschnitt 5.3.6.3.1).

Im Zusammenhang mit erweiterten Musterwerten ist die Feststellung der Ähnlichkeit zweier Messwerte, als Basis der Zyklenerkennung, folgendermaßen anzupassen:

$$n'_x \sim n_x \textit{ gilt dann, wenn } \begin{cases} n'_x = 0 \textit{ und zugleich } m_x = 0, \textit{ oder} \\ m_x \geqq 0, \end{cases} \tag{5.22}$$

Durch die Erweiterung der Musterwerte ändert sich die spezielle Zyklenerkennung nicht. Zusätzlich erlaubt dies aber eine einfache Vorgehensweise, um Häufigkeiten bei der Lösung von Konfliktsituationen mit einzubeziehen. Da sich Abweichungen der tatsächlichen Messwerte von den Musterwerten mit steigender Anzahl von Zyklenwiederholungen ergeben, kann es bei der Gewinnberechnung nützlich sein, die umgebenden Musterwerte mit zu berücksichtigen. Somit lässt sich die Häufigkeit einer Umgebung von Musterwert m_i mit $m_{i-1} + m_i + m_{i+1}$ berechnen.

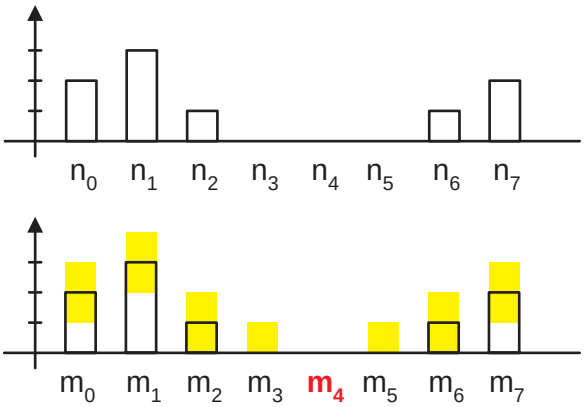

Abbildung 5.36: Einsatz von Häufigkeiten als Musterwerte

Obwohl es grundsätzlich nicht erforderlich ist, wird für die Zyklenerkennung in adaptiven Komponenten eine spezielle Art der Ähnlichkeitsfeststellung eingesetzt. Diese führt zu einer verbesserten Zyklenerkennung. Da nicht nur auf sichere Nullstellen getestet wird, beugt dies der Erkennung von zyklischem Verhalten, das sich innerhalb eines relevanten Zyklus anhand von Nullstellen zeigen kann, vor. Zudem können Verschiebungen auf der Grundlage von Sonderfällen (siehe Unterabschnitt 5.3.6.2.4) erkannt werden. Dennoch ist es keine allgemein gültige Lösung der beiden Problematiken. Diese Vorgehensweise zeigt sich jedoch im Zusammenhang mit dem adaptiven Prozessor als praktikabel (siehe Kapitel 6). Zur Feststellung von Ähnlichkeiten dient erneut eine Unschärfe, die zu folgender Vorschrift führt:

$$n'_x \sim n_x \; gilt \; dann, \; wenn \begin{cases} n'_x = 0 \; falls \; m_x = 0, \; sonst \\ m_x - u \leqq n'_x \leqq m_x + u \end{cases} \tag{5.23}$$

Abbildung 5.36 zeigt exemplarisch eine Messreihe n_0 bis n_7 und das resultierende Muster. Zudem sind in Abbildung 5.36 die zulässigen Abweichungen vom Muster angedeutet. Sichere Nullstellen erlauben keine Abweichung.

Die Problematik der Zyklenerkennung in Bezug auf eine festgelegte Unschärfe u wurden in Abschnitt 5.3.6.1.2 detailliert behandelt. Hieraus ergibt sich eine starke Limitierung bei der Wahl einer Unschärfe. Damit Zyklen nach wenigen Wiederholungen erkannt werden, ist ein Ganzzahliges u im Bereich $[\frac{n_{Max}}{2}, \frac{n_{Max}}{3}]$ für das in dieser Arbeit behandelte System günstig. Dies ist eine Beschränkung, die sich aus dem Hardware-Aufwand ergibt. Für Systeme, die diesbezüglich mehr Ressourcen bereitstellen (können), kann eine weitaus kleinere Unschärfe gewählt werden. Hierbei entstehen wiederum andere Probleme, die auf verschiedene Laufzeitkriterien, wie z.B. Zugriffslatenzen usw., zurückzuführen sind, wodurch sich

ein zu kleines u als unpraktikabel erweist. Die Wahl einer geeigneten Unschärfe u muss auf Grund realer Systemeigenschaften erfolgen.

5.3.6.3.4 Auswirkung von Laufzeiteinflüssen: Unter dem Begriff Laufzeiteinflüsse sind all diejenigen Faktoren zusammengefasst, die in einem realen System die Ausführungzeit einer Anwendung in Abhängigkeit von aktuellen Systemzuständen verändern. Zum Beispiel Registerabhängigkeiten, falsch vorhergesagte Sprünge oder Wartezeiten durch Speicherzugriffe etc. Diesbezüglich problematisch ist ausschließlich eine Veränderung der Laufzeiteinflüsse. Wird z.B. stets ohne die Ausnutzung der Caches auf den Hauptspeicher zugegriffen, beeinflusst dies die Zyklenerkennung und Mustererzeugung nicht. Allgemein können Laufzeiteinflüsse, deren zeitliche Auswirkung wesentlich kleiner als das Überwachungsintervall T_G sind, vernachlässigt werden. Dies liegt grundsätzlich daran, dass die Mustererzeugung und Zyklenerkennung in Bezug auf eine mögliche Restproblematik bereits einen Schwankungsbereich zulassen muss. Ob eine zeitliche Veränderung nun tatsächlich auf der Restproblematik beruht oder dies durch Laufzeiteinflüsse hervorgerufen wurde, muss nicht gesondert berücksichtigt werden. In diesem Zusammenhang muss jedoch die Behandlung von Links- oder Rechtsverschiebung dahingehend erweitert werden, damit stets beide Verschiebungsrichtungen zulässig sind.

Laufzeitveränderungen, deren Auswirkungen im Bereich des Messintervalls T_G oder darüber liegen, können sich als durchaus problematisch, sowohl für die Mustererzeugung als auch die Zyklenerkennung, erweisen. Insbesondere Speicherzugriffe, die nicht mittels Caches abgewickelt werden können, führen häufig zu sehr langen Latenzen. Dies ist wiederum nur dann zu beachten, wenn sich daraus unregelmäßige zeitliche Veränderungen ergeben. Ist dieses Verhalten mit einem erkannten Zyklus vereinbar, muss dies nicht berücksichtigt werden. Da in Bezug auf Speicherzugriffe bzw. das Speicherzugriffsverhalten keine allgemein gültigen Aussagen möglich sind, wird für das adaptive System eine Vorgehensweise eingeführt, die diese Faktoren eliminiert: Ein Speicherzugriff führt zum Anhalten der Messintervalle. Die Überwachung bleibt bestehen. Das heißt, dass die Nutzungsmöglichkeiten, die sich zwischenzeitlich ergeben, weiterhin gezählt werden. Die Messwerte werden dem aktuell vorliegendem Messintervall zugeschrieben. Nach Abschluss des Speicherzugriffs wird die Weiterführung der Messintervalle wieder erlaubt. Demzufolge sind die erhaltenen Messwerte unabhängig von der tatsächlichen Dauer eines Speicherzugriffs. Dies ist zulässig, da davon ausgegangen werden kann, dass die weitere Ausführung einer Anwendung – solange die Speicherinhalte nicht verfügbar sind – im schlechtesten Fall nach wenigen Taktzyklen angehalten werden muss. Nur falls die angeforderten Speicherinhalte keinen Bezug zur weiteren Anwendungsausführung hätten, könnte die Anwendung längerfristig fortgeführt werden und es würde diesbezüglich eine nicht hinnehmbare Messwertverfälschung auftreten. Erfahrungsgemäß werden solche Instruktionen, bzw. Speicherzugriffe, von Compilern erkannt und entfernt.

Das Anhalten der Messintervalle kann grundsätzlich für alle laufzeitverändernden Faktoren angewendet werden. Um Ressourcen zu sparen, wird jedoch abgelehnt, alle Beeinflussungen nach obiger Methode zu eliminieren. Ausschließlich die Faktoren, die die Mustererzeugung und Zyklenerkennung massiv beeinträchtigen, werden entsprechend behandelt.

5.3.6.3.5 Unechte Zyklen: Des Weiteren kann nicht ausgeschlossen werden, dass ein als Zyklus erkanntes Verhalten kein Zyklus ist. Diese Problematik ist zur Laufzeit nur dann

feststellbar, falls sich nach ein oder mehreren Wiederholungen das gefundene Muster nicht mehr anwenden lässt. In diesem Fall muss die Regelung auf Basis des Musters abgebrochen und erneut eine Zyklenerkennung gestartet werden.

Die Fälle, dass kein echter Zyklus vorliegt, aber ein gewonnenes Muster dennoch stets auf auftretende Nullstellen anwendbar ist, müssen nicht gesondert behandelt werden. Für die Regelung ist nur das Verhalten der Nullstellen von Bedeutung. Hierbei kommt es im schlechtesten Fall zu der Situation, dass mehr sichere Nullstellen verfügbar wären, diese aber nicht in der Regelung berücksichtigt werden.

5.4 Implementierungsaspekte der Überwachung und Regelung

Dieser Abschnitt beschreibt die grundlegenden Aspekte der Implementierung der in der Hardware-Schicht des Gesamtsystems eingesetzten Überwachungs- und Regelungskomponenten. Im Gegensatz zum vorherigen Abschnitt, der sich überwiegend mit der Überwachung von adaptiven Komponenten bzw. Modulen beschäftigt, wird hier auf die Realisierung für die gesamte Hardware-Schicht, unabhängig von einzelnen Systemkomponenten, eingegangen. In diesem Zusammenhang existiert keine ausgezeichnete Gruppe von Systemteilen oder -komponenten, die überwacht wird. Die Überwachung ist allgemein auf Ereignisse ausgerichtet. Ob eine Ereignisauslösung durch ein einzelnes Hardware-Modul, eine adaptive Komponente oder durch mehrere Systemkomponenten erfolgt, ist ohne Bedeutung.

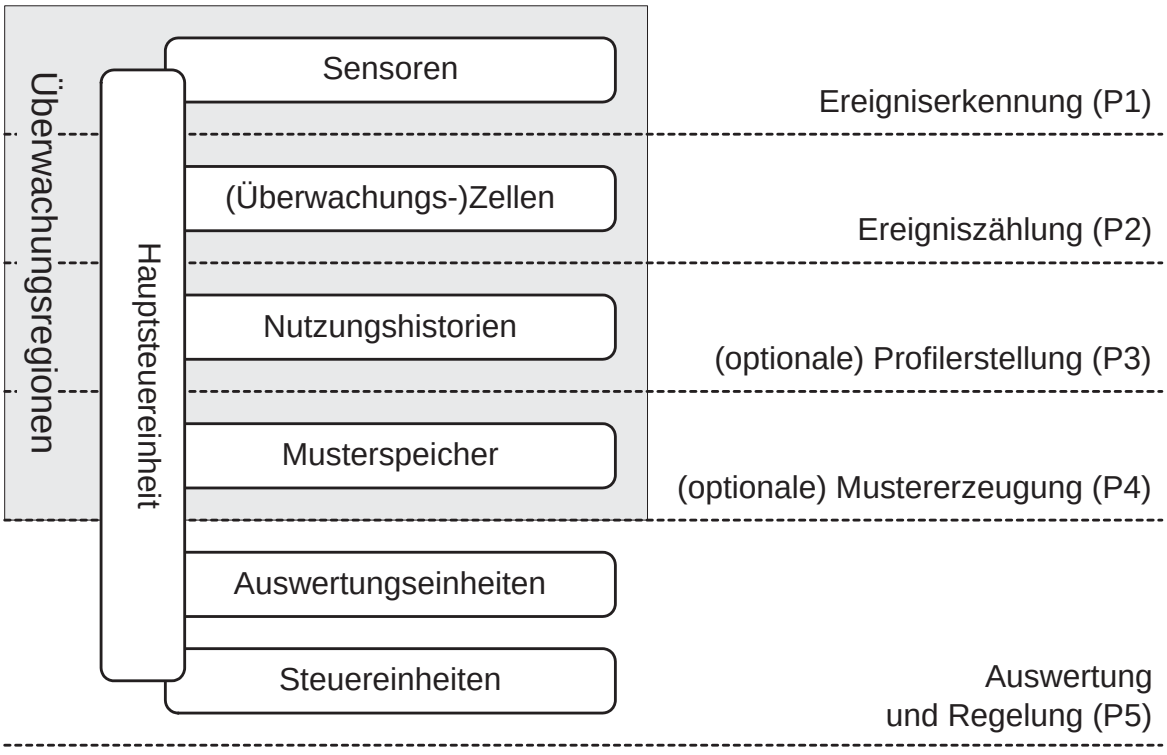

Abbildung 5.37: Phasen und zugehörige Schichten der Überwachung und Steuerung

Abbildung 5.37 skizziert die fünf Phasen (*P1 bis P5*), in die sich der Überwachungs- und Regelungsablauf unterteilt. Den Phasen 1 bis 4 lässt sich jeweils eine Schicht der Überwachungs- und Regelungsmechanismen zuordnen. Phase 5 umfasst zwei Schichten. Alle Schichten werden von der Ebene *Hauptsteuereinheit* geschnitten. Sowohl die Schichten als auch die Ebene bestehen jeweils aus ein oder mehreren gleichnamigen Einheiten.

Dementsprechend existieren folgende Überwachungseinheiten (Überwachungselemente) mit unterschiedlicher Funktionalität: Sensoren, Überwachungszellen, Nutzungshistorien, Musterspeicher, Auswertungseinheiten, Steuereinheiten und eine Hauptsteuereinheit.

Die Ereigniserkennung wird durch *Sensoren* realisiert. Für die Ereigniszählung sind *Überwachungszellen* eingesetzt. Diejenigen Überwachungselemente, die einem Ereignis oder einer logischen Gruppe von Ereignissen zugeordnet sind, bilden eine Überwachungsregion. Diesbezüglich ist es zulässig, dass mehrere Überwachungsregionen über gemeinsame *Auswertungseinheiten* und *Steuereinheiten* verfügen.

Die Sensoren und Überwachungszellen repräsentieren die Überwachungsgrundelemente, die stets vorhanden sein müssen. Umgekehrt ist sowohl die Profilerstellung als auch die Mustererzeugung eine optionale Phase der Überwachung. Das heißt, dass weder *Nutzungshistorien* noch *Musterspeicher* innerhalb einer Überwachungsregion zwingend erforderlich sind. Existiert mindestens eine Nutzungshistorie, erhält diese die entsprechenden Daten aus einer oder mehreren Überwachungszellen. Dies gilt ebenfalls für Musterspeicher. Zudem können Musterspeicher auch Daten aus vorgeschalteten Nutzungshistorien übernehmen.

Eine *Hauptsteuereinheit* kontrolliert alle zusammengehörigen Überwachungsregionen und die notwendigen Auswertungs- und Steuereinheiten. Vorrangig ist hier die Vorgabe des Granularitätsintervalls zu erwähnen. Darüber hinaus fallen die Aktivierung und Deaktivierung einzelner physikalischer Komponenten der Überwachungsregion(en) in den Aufgabenbereich der Hauptsteuereinheit. Zusätzlich steuert diese die Kommunikation und den Datenaustausch der Überwachungselemente untereinander.

5.4.1 Spezielle Anforderung an eine Hardware-Umsetzung

Für die Implementierung der oben beschriebenen Überwachungs- und Regelungselemente gelten spezielle Anforderungen, um dem Einsatz im adaptiven System gerecht zu werden. Die Anforderungen ergeben sich zum einen aus der vorgegebenen Beschränkung des Ressourcenverbrauchs für Überwachungs- und Regelungsmaßnahmen in Bezug auf Hardware und Energie und zum anderen aus der Notwendigkeit kurzer Reaktionszeiten der Regelungen:

- Die Grundelemente der Überwachung müssen einfach strukturiert und stark spezialisiert sein.
- Eine primäre Datenreduktion muss, falls dies möglich ist, bereits in den Grundelementen durchgeführt werden.
- Alle weiterverarbeitenden Elemente müssen eindeutig zu den korrespondierenden *Überwachungselementen* zuzuordnen sein. Das bedeutet, Ereignisse werden durch zwei oder mehrere Elemente, die eine gemeinsame Region bilden, überwacht. Die Kaskadierung mehrerer Grundelemente ist zulässig.
- Durch spezialisierte *Überwachungsregionen* ist die Information über die Herkunft und Beschaffenheit der Daten implizit vorhanden.
- *Steuerelemente* greifen steuernd, auf Basis der Daten einer oder mehrerer Überwachungsregionen, in das System ein.

5.4.2 Grundlegende Überwachungsbausteine

Im Folgenden werden die Überwachungselemente, die die grundlegenden Bausteine des Überwachungssystems bilden, in ihrer Grundstruktur vorgestellt. Unter

Berücksichtigung des Einsatzgebiets[9] von Überwachungsregionen sind zusätzliche spezialisierte Überwachungskomponenten zulässig. Hierfür existiert keine Einschränkung der Schnittstellen, Funktionalität oder Struktur. Solche Komponenten, die als *Überwachungshilfselemente* bezeichnet werden, dürfen in jeder Überwachungsschicht auftreten und unterstützen die dort angesiedelten Überwachungselemente in ihrer Funktion.

Im Gegensatz dazu ist der Aufbau der grundlegenden Überwachungselemente weitestgehend modular, d.h. die Schnittstellen bleiben grundsätzlich erhalten, wobei die Funktionalität der Komponenten je nach Einsatzgebiet variieren kann. Eine Einschränkung der Modularität ist auf Grund der Anforderungen an eine Hardware-Umsetzung akzeptabel. Dies erlaubt eine Optimierung der Überwachungseinheiten in Abhängigkeit vom jeweiligen Einsatzgebiet, in Bezug auf die Energieeffizienz und den Ressourcenbedarf. Der im Weiteren vorgestellte Aufbau der einzelnen Bausteine ist als Richtlinie für eine Realisierung zu verstehen.

Alle Überwachungskomponenten sind mit dem Systemtakt synchronisiert und können über ein asynchrones Reset-Signal zurückgesetzt werden. Die hierfür benötigten Eingangssignale fehlen in den Abbildungen. Darüber hinaus wurde aus Gründen der Übersichtlichkeit darauf verzichtet, die Signalleitungen der korrespondierenden Hauptsteuereinheit darzustellen. Diese müssen ebenfalls an alle zugehörigen Überwachungs- und Regelungselemente angeschlossen sein. Diesbezüglich sind folgende Signale notwendig:

Kürzel	Signalname	Kurzbeschreibung
IRst	Interner Reset	Zurücksetzen aller Komponenten der zugehörigen Überwachungsregion
Pause	Pause	Anhalten der Überwachungsregion. Die Inhalte der einzelnen Komponenten bleiben erhalten.
EMi	Messintervallende	Signalisiert das Ende bzw. den Neustart eines Überwachungsintervalls.
Ak	Aktivierung	Aktiviert oder deaktiviert eine oder mehrere Überwachungskomponenten.
SE	Signal zulassen	Betrifft die Kommunikation zwischen Überwachungselementen. Einzelne Kommunikationskanäle können explizit an- und abgeschaltet werden.

Tabelle 5.1: Erforderliche Steuersignale der Überwachungs- und Regelungselemente

Zudem sollten grundsätzlich alle Überwachungskomponenten *clock-gating* unterstützen, um den Energiebedarf der Überwachung zu senken (Vgl. Grundlagenabschnitt 2.4). Mit Ausnahme der Überwachungsgrundelemente lässt sich *clock-gating* einfach durch die Kopplung an das Ende der Messintervalle realisieren. Hierbei muss die Hauptsteuereinheit gewährleisten, dass die Komponenten einen entsprechenden Zeitraum zur Verfügung haben, der ausreicht, um die benötigten Vorgänge abzuwickeln. Anschließend können die Komponenten bis zum nächsten Messintervallende deaktiviert werden.

[9] Einsatzgebiet bezeichnet im Zusammenhang mit der Überwachung und Regelung, den oder die Systemteile, bzw. Komponenten, die überwacht und geregelt werden.

5.4.2.1 Sensoren und Überwachungszellen

Mindestens ein Sensor und eine zugehörige Überwachungszelle sind in jeder Überwachungsregion anzutreffen. Die Aufgabe des Sensors besteht darin, ein oder mehrere Signale, die das zu überwachende Ereignis beschreiben, zu einem einzelnen Signal zu bündeln, das als Eingang der Überwachungszelle dient und das Auftreten des observierten Ereignisses anzeigt. Zum Beispiel die Durchführung der Dekodierung einer Instruktion kann ein Ereignis sein, dass mittels eines einzelnen Sensors festgestellt wird. Jedes zu überwachende Ereignis muss durch einen Sensor erkannt werden. Lassen sich i Ereignisse gemeinsam überwachen, müssen i Sensoren an eine einzelne Überwachungszelle angeschlossen sein.

Wie bereits erwähnt, wird im Weiteren bei der Beschreibung davon ausgegangen, dass sowohl der Systemtakt als auch ein globaler Reset stets vorhanden sind. Diese fließen nicht in die Schnittstellenbeschreibung der Überwachungselemente ein. Die Schnittstelle der Sensoren lässt sich entsprechend mit $n \epsilon \mathbb{N}_0$ Eingangsleitungen und einer Ausgangsleitung beschreiben. Der Spezialfall $n = 0$ ist prinzipiell erlaubt. Zum Beispiel kann ein Sensor eine Systemtaktzählung implementieren. Das zugehörige Ereignis wird dann angezeigt, wenn der Zähler einen gewissen Grenzwert erreicht oder einen Überlauf aufweist. Hierbei sind keine zusätzlichen Eingangssignale notwendig.

Eine zugehörige Überwachungszelle registriert die von i Sensoren, mit $i \geqq 1$, wahrgenommenen Ereignisse. Die Funktionalität und der Aufbau von Überwachungszellen sind vergleichbar mit den in gängigen Prozessoren eingesetzten Ereigniszählern. Überwachungszellen verfügen jedoch über keine Benutzerschnittstelle und benötigen deshalb sowie auf Grund kleinerer Zählerregister weniger Hardware-Ressourcen (Vgl. z.B. Ereigniszähler Itanium [76], Itanium 2 [78], DEC Alpha [35] usw.).

Der Aufbau der Überwachungszellen ist in Abbildung 5.38 skizziert. Die zugehörigen Sensoren fehlen in der Abbildung. *SE* repräsentiert die Ausgänge von i Sensoren, die einer Überwachungszelle als Eingänge dienen. Die *Eingangsbewertung* bestimmt, in welcher Weise die Zählung der Eingangssignale (*SE*) erfolgt. Grundsätzlich sind vier Implementierungen vorgesehen, die zu einer Zählererhöhung führen:

1. **Oder-Verknüpfung**: Mindestens ein Eingangssignal muss aktiv[10] sein.
2. **Und-Verknüpfung**: Alle Eingangssignale müssen aktiv sein.
3. **Filter**: Die Eingangssignale müssen spezielle Kriterien (boolesche Verknüpfungen) erfüllen.
4. **Akkumulation**: Die Anzahl aktiver Eingänge wird zum aktuellen Zählerstand addiert.

Das Zurücksetzen des *Zählers* erfolgt über ein Reset-Signal (*IRst*). Der Reset unterliegt der externen Hauptsteuereinheit und ist in Abbildung 5.38 nicht eingezeichnet. Zudem bleibt der Zählerwert bei Erreichen des Maximalwerts erhalten. Ein Überlauf oder internes Zurücksetzen ist nicht vorgesehen. Mit Hilfe von Schwellwerten wird anhand des Zählerstands ein Ausgangsdatum generiert. Allgemein sind $m - 1$ Schwellwerte zulässig,

[10] Aktiv bedeutet, dass ein Ereignis angezeigt wird.

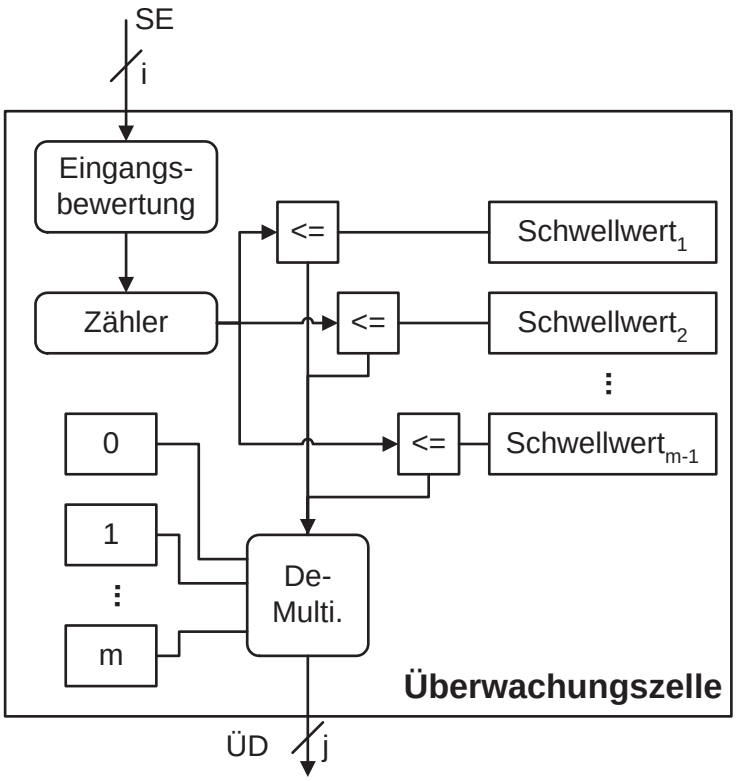

Abbildung 5.38: Aufbau einer Überwachungszelle

die zu einem ganzzahligen Ausgangswertebereich von $[0, m]$ führen. Soll nur der Zählerwert ausgegeben werden, kann auf die Verwendung von Schwellwerten und den notwendigen *Demultiplexer* verzichtet werden. Der Einsatz von Schwellwerten eröffnet die Möglichkeit, Filter zur Elimination saisonaler Schwankungen zu integrieren (siehe Abschnitt 3.4).

Die Größe des Zählers ist vom Einsatzgebiet der Überwachungsregion und den überwachten Ereignissen abhängig. Hierbei sind das Überwachungsintervall T_G sowie das resultierende Maximum von Ereignissen n_{Max} während T_G maßgebend. Bei i an eine Überwachungszelle angeschlossenen Sensoren, ergibt sich allgemein ein maximaler Wertebereich des Zählers mit $\sum_{0 \leqq k<i} n_{Max}(k)$, wobei $n_{Max}(k)$ die maximale Anzahl möglicher Ereignisse des Sensors k im Intervall T_G darstellt (Vgl. auch Abschnitt 5.3.6).

Die Breite des Ausgangsdatums $\ddot{UD}$ hängt ausschließlich vom Ausgangswertebereich, der sich aus der Anzahl eingesetzter Schwellwerte ergibt, ab. Zudem muss bei der Implementierung darauf geachtet werden, dass der Ressourcenaufwand der Überwachungszellen minimiert wird. Überwachungszellen und Sensoren stellen die am häufigsten verwendeten Überwachungselemente dar. Sind diesbezüglich Vereinfachungen bzw. Optimierungen möglich, gilt es diese umzusetzen.

5.4.2.2 Nutzungshistorien und Musterspeicher

Für die Überwachung von Ereignissen, deren Verlauf die Basis einer Regelung darstellt, werden Nutzungshistorien und/oder Musterspeicher eingesetzt. Insbesondere die in Abschnitt

5.3 behandelten Überwachungs- und Regelungsmechanismen, in Bezug auf aktive adaptive Komponenten, sind auf beide Überwachungsbausteine angewiesen.

Eine Nutzungshistorie speichert Überwachungsdaten in chronologischer Reihenfolge. Am Ende jedes Granularitätsintervalls können die aktuellen Überwachungsdaten einer einzelnen Überwachungszelle an eine Nutzungshistorie zur Ablage weitergereicht werden.

Im Gegensatz dazu kann ein Musterspeicher sowohl Überwachungsdaten in zeitlicher Reihenfolge als auch zu einem Zeitpunkt aus mehreren unabhängigen Datenquellen aufnehmen. Darüber hinaus werden die im Musterspeicher abgelegten Daten im Allgemeinen korreliert und soweit möglich reduziert. Die Datenreduktion muss in dem Umfang stattfinden, dass das resultierende Muster mit möglichst geringem Aufwand ausgewertet bzw. direkt zur Steuerung herangezogen werden kann.

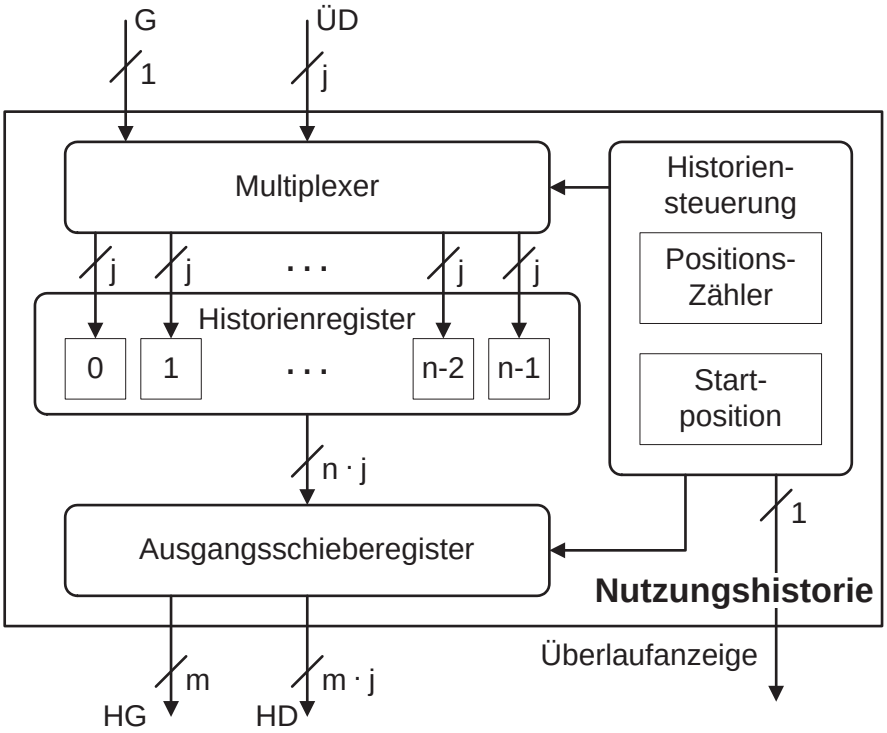

Abbildung 5.39: Aufbau einer Nutzungshistorie

Abbildung 5.39 zeigt den Aufbau einer Nutzungshistorie, die in der Lage ist, n Messwerte der Länge j aufzunehmen. Hierbei repräsentiert der Eingang *ÜD* die Messwertdaten und *G* zeigt die Gültigkeit des Eingangs an. Als Ablage dient das *Historienregister*. Dieses ist als Ringpuffer organisiert. Innerhalb der *Historiensteuerung* werden die aktuelle *Startposition*, sowie die Position für neue Eingangsdaten verwaltet. Durch die Ringpufferorganisation müssen neu erhaltene Messdaten entsprechend verteilt (*Multiplexer*) und anhand des *Positionszählers* an der richtigen Position im Historienregister abgelegt werden. Tritt ein Überlauf auf, wird dieser durch einen 1 Bit Ausgang angezeigt. Intern erfolgt keine Behandlung eines Überlaufs. Das bedeutet, dass alte Messdaten durch neue überschrieben werden. Entsprechend verschiebt sich die Startposition des Historienregisters. Damit sich eine erzeugte Nutzungshistorie für nachfolgende Überwachungskomponenten ohne weitere Eingriffe stets

in der richtigen Reihenfolge darstellt, ist das *Ausgangsschieberegister* vorgelagert. Unter der Kontrolle der Historiensteuerung wird das Historienregister soweit verschoben, dass die Ausgangsleitungen *HD* die aufgezeichneten Daten so präsentieren, als ob die Startposition mit dem ersten Datum des Historienregisters übereinstimmt. Entsprechend zeigen die *HG*-Leitungen, die Gültigkeit der einzelnen Einträge an, und somit implizit die Länge der Historie.

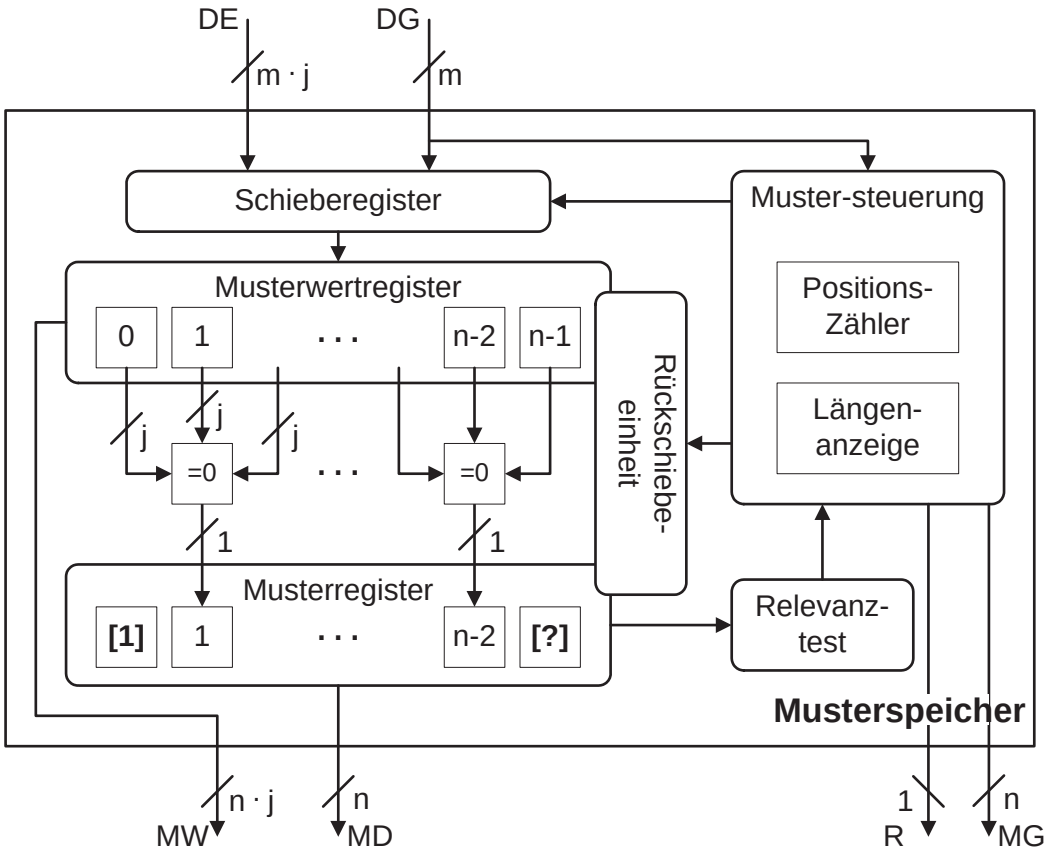

Abbildung 5.40: Aufbau eines Musterspeichers

Ähnlich einer Nutzungshistorie zeigt sich der Aufbau eines Musterspeichers (Abbildung 5.40). Ein wesentlicher Unterschied besteht jedoch in der Art der Eingangsdaten (*DE*). Hierbei sind m Daten mit je j Bit Breite zulässig. Die Gültigkeit der einzelnen Daten signalisiert *DG*. Mit Hilfe eines Schieberegisters und der *Mustersteuerung* werden neu ankommende Eingangsdaten in die korrespondierenden Registerteile des *Musterwertregisters* abgelegt. Die maximale Länge eines Musters und die Anzahl der Musterwerte ist auf n beschränkt. Das Muster wird exemplarisch nach Vorschrift 5.21 (siehe Abschnitt 5.3.6.3) aus den Musterwerten generiert[11]. Es ergeben sich zwei konstante Einträge im Muster, die in Abbildung 5.40 von eckigen Klammern umschlossen sind. Ansonsten repräsentiert die Nummerierung die Reihenfolge der Einträge. Sowohl die Musterwerte als auch das Muster sind über die Ausgänge *MW* bzw. *MD* für andere Komponenten verfügbar.

[11] Je nach Überwachungsregion sind unterschiedliche Vorschriften zur Mustererzeugung zulässig.

Ist eine Nutzungshistorie mit den Eingängen eines Musterspeichers verbunden, sind die Musterwerte eine lokale Kopie der Werte der Nutzungshistorie. In diesem Fall kann das Musterwertregister bei einer Implementierung weggelassen werden. Dienen die Daten ein oder mehrerer Überwachungszellen als Eingang des Musterspeichers, werden diese im Musterwertregister abgelegt. Für die Überwachung der Nutzung von Hardware-Modulen bedeutet das, dass ein Musterspeicher entweder die Nutzung eines Moduls über mehrere Granularitätsintervalle hinweg ablegen oder die Nutzung mehrerer unabhängiger Module während eines einzigen Granularitätsintervalls darstellen kann. Auf welche Weise ein Musterspeicher genutzt wird, hängt davon ab, ob die Daten für eine Autokorrelation eines Moduls oder für die Korrelation mehrerer Module eingesetzt werden soll. In dieser Arbeit sind ausschließlich Musterspeicher zur Erzeugung eines chronologischen Musters eines einzelnen Moduls von Interesse.

Die Gültigkeit einzelner Einträge wird mit MG signalisiert und unterliegt der Kontrolle der Mustersteuerung. MG repräsentiert somit implizit die aktuelle Länge des Musters. Diesbezüglich ist anzumerken, dass die Musterwerte eine optionale Erweiterung eines Musterspeichers sind. Ist ausschließlich das Muster für eine Überwachungsregion von Bedeutung, entfällt das Musterwertregister und der zugehörige Ausgang. Die Verknüpfung einzelner Musterwerte zum eigentlichen Muster kann und darf auch ohne Musterwertregister erfolgen.

Der *Relevanztest* prüft das vorhandene Muster auf verschiedene Sonderfälle und leitet das Ergebnis an die Mustersteuerung weiter. Mit Hilfe des Relevanztests, kann vorab entschieden werden, ob ein Muster für die weitere Auswertung in Betracht gezogen werden muss oder nicht. In Zusammenhang mit der in Abschnitt 5.3.6.3 eingeführten Regelung auf Basis von Zyklen, würde ein Muster, für den Fall, dass keine sicheren Nullstellen auffindbar sind, als irrelevant eingestuft. Der Relevanztest ist jedoch nur in Verbindung mit der aktuellen Länge des Musters sinnvoll. Deshalb wird das Ergebnis erst der Mustersteuerung übergeben. Diese legt in Verbindung mit der Länge MG des vorliegenden Musters den resultierenden 1-Bit Wert an den Ausgang R.

Eine weitere Aufgabe der Mustersteuerung ist die Einleitung einer etwaigen Verschiebung im Musterwertregister und zugehörigem Musterregister. Durchgeführt wird eine Verschiebung von der *Rückschiebeeinheit*, die stets eine einzige Rechtsverschiebung dergestalt ausführt, dass ein Eintrag an der Position i mit dem Datum der Position $i+1$ belegt wird usw. Eine Verschiebung im Musterregister ist in Abhängigkeit von der tatsächlichen Implementierung nicht notwendig, da die Einträge aus den verschobenen Musterwerten neu erzeugt werden können.

5.4.2.3 Auswertungseinheiten

Bereits bei der Vorstellung der Musterspeicher kristallisiert sich heraus, dass die Funktionalität stark von der Überwachungsregion abhängig ist. In diesem Zusammenhang ergibt sich jedoch eine weitestgehend allgemein gültige Struktur. In Bezug auf die Auswertungseinheiten gilt dies nicht mehr. Sowohl die Funktionalität als auch die Struktur dieser Einheiten ist nicht festgelegt. In diesem Fall steht eine energieeffiziente und ressourcensparende Implementierung im Vordergrund. Dennoch können in Bezug auf eine Schnittstelle zwei prinzipielle Aussagen getroffen werden:

- Mindestens zwei Eingangsdaten unterschiedlichen Ursprungs müssen vorliegen.

- Die Ausgangsdaten benötigten in der Regel wenige Leitungen im Vergleich zu den Eingangsdaten und zeigen den Zusammenhang der Eingangsdaten an.

Letzteres führt dazu, dass Auswertungseinheiten als *Indikatoren* bezeichnet werden. Da sowohl die Funktionalität als auch die Struktur nicht festgelegt sind, wird exemplarisch eine Indikatoreinheit vorgestellt, die für die Erkennung von Zyklen auf der Grundlage von Mustern eingesetzt werden kann (Abbildung 5.41).

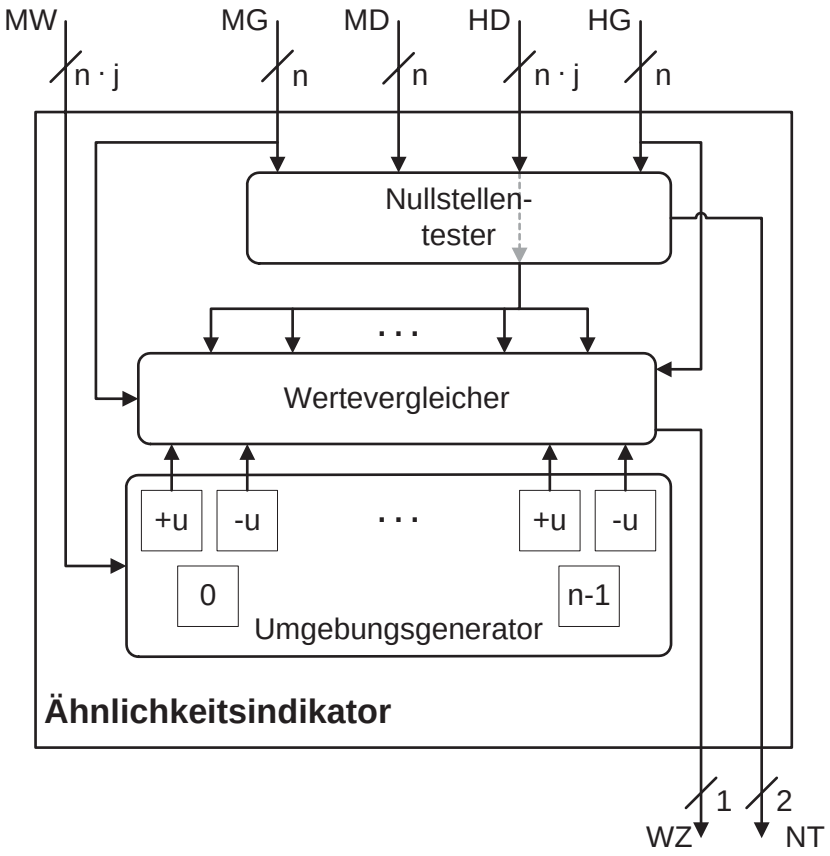

Abbildung 5.41: Exemplarischer Aufbau eines Indikators zur Zyklenerkennung

Als Eingänge dienen die einzelnen Musterwerte MW und das Muster selbst (MD). Die Gültigkeit wird wiederum durch MG signalisiert. Zudem werden die Daten einer Nutzungshistorie am Eingang HD und entsprechend die Gültigkeit an HG bereitgestellt.

Der *Nullstellentester* vergleicht das Muster und die Daten der Nutzungshistorie auf übereinstimmende, sichere Nullstellen. Der zugehörige Ausgang NT kann vier Werte annehmen: $NT = 0$ signalisiert, dass alle sicheren Nullstellen des Musters in der Nutzungshistorie auffindbar sind. $NT = 1$ zeigt eine Linksverschiebung und $NT = 2$ eine Rechtsverschiebung an. Bei $NT = 3$ können die sicheren Nullstellen nicht identifiziert werden. Für eine Regelung auf Basis sicherer Nullstellen ist diese Information bereits ausreichend.

Zudem findet sich in Abbildung 5.41 ein zusätzlicher *Wertevergleicher*. Dieser realisiert eine Ähnlichkeitsfeststellung mit Hilfe einer Messwertumgebung u. Ausgang WZ zeigt somit entweder, dass die Musterwerte und die Werte der Nutzungshistorie ähnlich sind, oder dass

diese um mehr als die Umgebung u für mindestens ein Wertepaar abweichen. Dies ist eine zusätzliche Information, die den Zusammenhang zwischen Muster und Historie genauer beschreibt. Für eine Regelung kann dies unterstützend eingesetzt werden, um z.B. eine frühzeitige Verschiebungserkennung (Details siehe Abschnitt 5.3.6) zu implementieren.

Insgesamt hängt jedoch die Funktionalität, Struktur und Schnittstelle der Auswertungseinheiten massiv von der zugehörigen Überwachungsregion und dem Ziel einer darauf aufbauenden Regelung ab.

5.4.2.4 Steuerungselemente

Die *Steuerungselemente* sind nur der Vollständigkeit halber erwähnt, da diese keine allgemein gültige Struktur, Funktionalität oder Schnittstelle aufweisen. Dieser Einheitentyp ergibt sich ausschließlich aus den Faktoren, die das jeweils überwachte Verhalten und dessen Steuerung vorgeben. Einzig und allein eine Mindestanforderung für die Schnittstelle kann bezugslos festgehalten werden: Es existiert wenigstens ein Eingang, der ein Steuerungselement mit einem Indikator verbindet. Als Ausgang muss mindestens ein Steuersignal vorhanden sein. Genauere allgemeine Angaben sind zu den Steuerungselementen nicht möglich und darüber hinaus durch die Forderung nach Spezialisierung eines Regelungssystems, bestehend aus Auswertungs- und Steuerungseinheiten, nicht erwünscht.

6.

Der adaptive Prozessor

Der adaptive Prozessor ist die Umsetzung einer aktiven adaptiven Komponente des in Kapitel 5 vorgestellten autonomen adaptiven Gesamtsystems. Seine grundsätzliche Befehlssatzarchitektur stellt sich als 32 Bit RISC-Architektur dar. Diese basiert auf der Befehlssatzarchitektur des Prozessorkerns ARMv4 ohne Thumb-Unterstützung (siehe Grundlagenabschnitt 2.3.3.1).

Im folgenden Abschnitt wird der konzeptionelle Aufbau des adaptiven Prozessors behandelt. Der anschließende Abschnitt widmet sich den Implementierungsaspekten sowie der Integration der Überwachungs- und Regelungsmechanismen. Des Weiteren wird der Einsatz von speziellen Hardware-Modulen, die so genannte *Operationskombinationen* implementieren, betrachtet. Im darauf folgenden Abschnitt findet sich die Darstellung der im Laufe der Arbeit erhaltenen Simulationsergebnisse.

6.1 Entwurf des adaptiven Prozessors

Dieser Abschnitt stellt das Konzept des adaptiven Prozessors vor, der speziell für den Einsatz im autonomen adaptiven Gesamtsystem entworfen wurde. Abbildung 6.1 zeigt eine Übersicht der logischen Komponenten des adaptiven Prozessors. Darüber hinaus erkennt man die Anbindung an ein hierarchisches Hauptspeichersystem. Hierbei sind drei Adressbereiche hervorgehoben, die für die Kommunikation zwischen Prozessor und oberen Systemkomponenten, vornehmlich der VM, benötigt werden (siehe auch Zugriff auf adaptive Komponenten in Abschnitt 5.2.3.1):

1. Die **Systembibliothek** stellt diejenige Information über verfügbare Hardware-Module bereit, die prozessorintern zur Überwachung und Regelung benötigt wird.

2. Der Adressbereich **Zustand** repräsentiert den aktuellen Konfigurationszustand[1] und den Konfigurationsstatus[2] und erlaubt einzelne Hardware-Module nachzuladen. Dieser Bereich unterliegt der Kontrolle des Betriebssystems.

[1] Der Konfigurationszustand beinhaltet die Indices der aktuell geladenen Module, sowie die Zuordnung der Module zu den verfügbaren Kontexten der rekonfigurierbaren Bereiche.

[2] Der Konfigurationsstatus zeigt an, ob ein Modul gerade in einen Kontext geladen wird oder ob der Konfigurationsvorgang abgeschlossen und das Modul verfügbar ist.

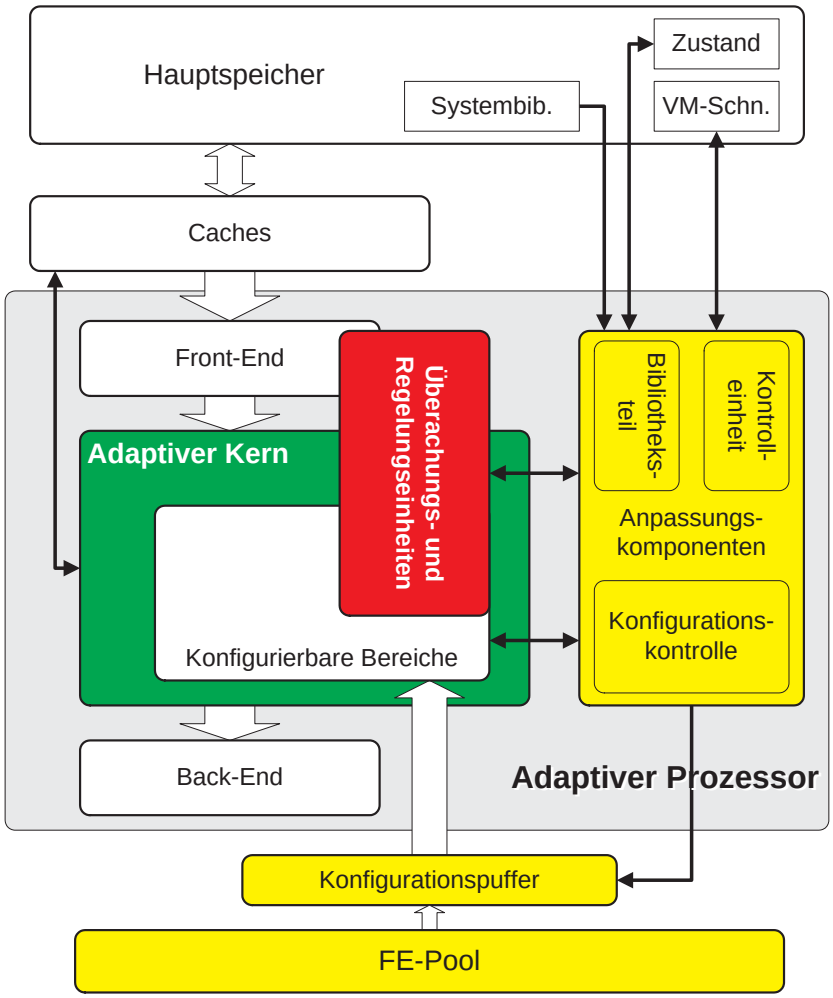

Abbildung 6.1: Schematisches Grundmodell des adaptiven Prozessors

3. Als **VM-Schnittstelle** ist der Adressbereich gekennzeichnet, der ausschließlich dem Informationsaustausch zwischen VM und Prozessor dient[3]. Hierbei existieren Ähnlichkeiten zu dem mit *Zustand* betitelten Adressbereich. Für die Kommunikation zwischen VM und Prozessor gelten jedoch andere Anforderungen, die gesondert behandelt werden müssen. Für die VM ist z.B. das Stattfinden von etwaigen Konfigurationsvorgängen ebenso von Bedeutung, wie die Möglichkeit ein bereits konfiguriertes Modul temporär für Rekonfigurationen zu sperren. Hierbei handelt es sich um essentielle Vorgänge in Bezug auf die Programmausführung mittels Software oder Hardware, die sich auf Grund ihrer kurzen Dauer sowohl der Überwachung als auch der Steuerung durch das Betriebssystem entziehen.

[3]Einen Adressbereich für die Kommunikation vorzusehen, hat den Vorteil dass keine Erweiterungen des zugrunde liegenden ARM-Kerns um ein oder mehrere Statusregister und folglich zusätzliche Instruktionen notwendig sind. Werden die Signal- und Datenleitungen direkt vom Prozessor, z.B. über zusätzliche Pins, nach außen geführt und in einen Adressbereich abgebildet, ist es möglich sowohl eine Software- als auch eine dem adaptiven Prozessor vorgelagerte Hardware-Implementierung der VM zu integrieren. Erstere realisiert die Kommunikation über den Adressbereich und letztere kann die Signalleitungen direkt abgreifen.

Man erkennt in Abbildung 6.1, dass alle drei Adressbereiche über eigene Kommunikationskanäle verfügen. Dies ist eine Empfehlung für die Implementierung des adaptiven Prozessors, die aus Leistungsgründen in Erwägung gezogen wurde. Wobei sich die Behandlung des Zustands weniger zeitkritisch als die Kommunikation zwischen VM und Prozessor darstellt, da der Zustand ausschließlich in Zusammenhang mit Prozesswechseln von Bedeutung ist. Die Daten der Systembibliothek werden in Form einer Kopie innerhalb des Prozessors verfügbar gemacht und erfahren bei jedem Prozesswechsel eine entsprechende Anpassung. Dies ist über die Speicherinfrastruktur durchführbar, sollte jedoch aus Performanzgründen eine eigenständige Anbindung (z.B. $memory - mapped$) an den Adressraum haben.

Die Kommunikation zwischen VM und Prozessor ist auf Grund ihrer Häufigkeit problematisch[4]. Letzteres kann, falls die vorhandene Speicherinfrastruktur genutzt wird, bei der Ausführung von Anwendungen zu massiven Beeinträchtigungen der Laufzeit führen. Deshalb ist die VM-Schnittstelle grundsätzlich von der Hauptspeicherinfrastruktur zu entkoppeln und wenn möglich, über eigene Steuer- und Datenleitungen des Prozessors zu realisieren.

Alle Kommunikationskanäle, die unmittelbar mit der Konfiguration und den sich daraus ergebenden Maßnahmen in Verbindung stehen, enden in dem logischen Bestandteil *Anpassungskomponenten*. Dieser besteht aus den drei Komponenten *Bibliotheksteil*, *Kontrolleinheit* und *Konfigurationskontrolle*, die bereits in Abschnitt 5.2.3 eingeführt wurden. Die Anpassungskomponenten regeln die Anforderung neuer Hardware-Module aus dem *Konfigurationspuffer* und stehen in direktem Kontakt zu den *Überwachungs- und Regelungseinheiten* sowie den *konfigurierbaren Bereichen* des adaptiven Prozessors.

Die konfigurierbaren Bereiche liegen im so genannten *adaptiven Kern* des Prozessors, der darüber hinaus die Schnittstellen zum *Front-End* und *Back-End* des Prozessors beinhaltet.

Neben der Verwendung einer VM-Schnittstelle als Grundlage der Modulnutzungserkennung, wurde eine alternative Variante untersucht, die ohne die Unterstützung einer VM umsetzbar ist. In diesem Zusammenhang wird ein CISC-ähnlicher Ansatz verfolgt, der den Einsatz von Modulen, die so genannte *Operationskombinationen* ausführen, favorisiert. Diejenigen Module, die Operationskombinationen implementieren, bilden eine Teilmenge der für das Gesamtsystem vorgesehenen Hardware-Module und erlauben sequentielle Maschinenbefehle mit echten Registerabhängigkeiten als eine einzige, kombinierte Instruktion zu verarbeiten. Diesbezüglich sind diverse Erweiterungen, die hauptsächlich das *Front-End* des Prozessors betreffen, notwendig. Im Weiteren wird die Integration beider Varianten im adaptiven Prozessor gezeigt. Auf Erweiterungen in der Funktionalität oder Struktur von Komponenten, die ausschließlich eine Variante betreffen, wird entsprechend hingewiesen. Insgesamt muss, auf Basis der bis dato erlangten Erkenntnisse, der VM-gestützten Realisierung der Vorzug gegeben werden. Da bisher nur Simulationsergebnisse sowie Implementierungen einzelner physikalischer Komponenten vorliegen, die nur bedingt Rückschlüsse auf den Energiebedarf des gesamten adaptiven Prozessors zulassen, kann nicht ausgeschlossen werden, dass sich die Alternative *ohne* VM-Unterstützung, bzw. der gleichzeitige Einsatz beider Varianten, in ausgesuchten Einsatzgebieten dennoch als rentabel erweist. Dies muss in weiterführenden Arbeiten untersucht werden.

[4]Der Entwurf des adaptiven Prozessors sieht aus Performanzgründen eine Hardware-Implementierung der VM vor.

6.1.1 Aufbau und Befehlsfluß

Im Gegensatz zu bestehenden Prozessoren gilt für den adaptiven Prozessor, dass die Überwachung nicht als gesonderte Zusatzfunktionalität betrachtet werden kann, sondern mit den internen Komponenten fest verbunden ist. Ebenso sind die durch die Überwachungseinheiten erhobenen Daten nicht für den externen Gebrauch, z.B. zur Unterstützung von Programmierern oder Werkzeugen für die Anwendungsoptimierung, gedacht, sondern müssen direkt im Prozessor – bzw. teilweise durch das Betriebssystem – ausgewertet und zweckdienlich verarbeitet werden. In diesem Zusammenhang kommt das in Abschnitt 5.3 allgemein vorgestellte Überwachungs- und Regelungssystem im adaptiven Prozessor zum Einsatz.

6.1.1.1 Der Befehlsfluss

Abbildung 6.2 zeigt eine Übersicht der, für die Befehlsverarbeitung notwendigen, Komponenten des Prozessors. Als Orientierungshilfe findet sich am rechten Rand die Einordnung der gezeigten Komponenten in den für Prozessoren üblichen Maschinenbefehlszyklus. Zudem ergibt sich eine weitere Unterteilung durch die Reihenfolge der Befehlsausführung. Propagiert wird eine Mischung aus *out-of-order Execution* und *in-order Completion*, wobei die Reihenfolge der Befehle in einem Reihenfolgepuffer (*Reorder Buffer*), vor der Vollendung der Rückschreibphase, wieder hergestellt wird. Den Grund für diese Entscheidung behandelt Abschnitt 6.1.1.1.3. Entsprechend der Ausführungsreihenfolge lässt sich das Schema in ein *Front-End* und *Back-End*, jeweils mit *in-order* Charakteristik und den *adaptiven Kern* mit *out-of-order* Charakteristik, zerlegen.

Die in Abbildung 6.2 schraffiert dargestellten Komponenten sind ausschließlich für die Modulrekonfiguration *ohne* VM-Unterstützung notwendig. Wird nur die VM-gestützte Methode zur Modulrekonfiguration herangezogen, entfallen die schraffierten Komponenten.

6.1.1.1.1 Front-End: Das *Front-End* umfasst die *Instruktionsholeinheit*, bis zu drei *Dekoder*, einen *Wandler* sowie die *Reservierungseinheit*. Die *Registerdatei* und das *Bypass-Netzwerk* sind ebenfalls dem Front-End zugeordnet.

Die Instruktionsholeinheit ist über einen 32 bzw. 96 Bit breiten Bus[5] an den Instruktionscache angeschlossen. Dies ermöglicht das gleichzeitige Lesen von bis zu drei Befehlsworten[6]. Alle geholten Maschinenbefehle werden in der Dekodierphase entsprechend in ein internes Befehlsformat übersetzt. Falls die hardware-basierte Modulrekonfiguration unterstützt wird, durchlaufen die internen Befehlsworte eine zusätzliche Wandlungs-Phase, die logisch der Dekodierphase des Maschinenzyklus zugeordnet ist. In Abhängigkeit von der aktuellen Konfiguration des adaptiven Kerns erfahren die internen Befehlsworte Veränderungen, die eine optimierte Ausführung in speziellen Hardware-Modulen ermöglichen. Die Reservierungseinheit sorgt anhand der vorliegenden internen Befehlsworte für das Lesen, Reservieren oder Sperren der korrespondierenden Register. Die angeforderten Registerinhalte werden entweder direkt durch die *Registerdatei* oder über das *Bypass-Netzwerk* bezogen.

[5] Die Busbreite hängt von der Anzahl der tatsächlich verwendeten Dekoder ab.

[6] Der adaptive Prozessor verwendet RISC oder unter Einbeziehung einer übergeordneten VM, RISC-basierte Maschinenbefehle mit konstanter Länge.

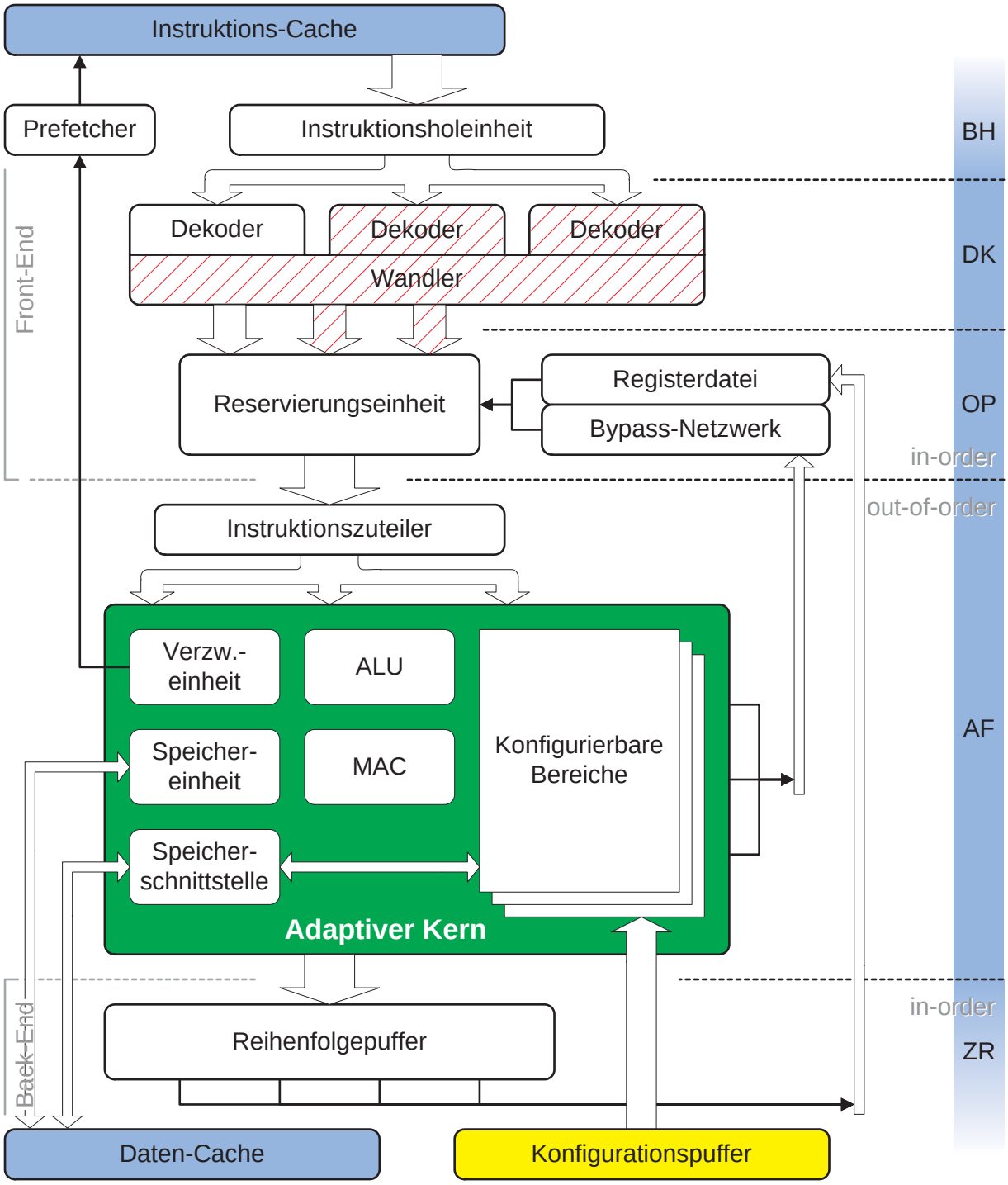

Abbildung 6.2: Komponentenschema des adaptiven Prozessors

6.1.1.1.2 Adaptiver Kern: Unter dem Begriff adaptiver Kern werden alle Funktionseinheiten und rekonfigurierbaren Ressourcen sowie die zugehörigen Überwachungs- und Regelungseinheiten des Prozessors zusammengefasst. Die *Instruktionszuteilungseinheit* gehört logisch dem Prozessorkern an, wird aber nicht zum adaptiven Kern gezählt. Wenn Befehle die Reservierungseinheit verlassen, werden diese in einer Warteschlange der Instruktionszuteilungseinheit eingeordnet. Im darauf folgenden Schritt erfolgt die Zuteilung ein oder mehrerer interner Befehlsworte zu einer entsprechenden Funktionseinheit, und deren Ausführung.

Vier Funktionseinheiten stehen permanent im adaptiven Kern bereit (Abbildung 6.2):

1. Eine *Verzweigungseinheit* zur Behandlung von Sprungbefehlen. Hierbei wurde in der Simulation eine statische Sprungvorhersage realisiert, die in Zukunft durch eine dynamische ersetzt werden sollte. Des Weiteren ist die Verzweigungseinheit über einen Kommunikationskanal mit dem *Instruction Prefetcher* verbunden, der zur Verringerung von Speicherzugriffslatenzen dient.

2. Die *Speichereinheit* stellt eine parametrisierbare Verbindung zum *Daten-Cache* über einen bis zu 256 Bit breiten Bus her. Entsprechend können maximal acht 32 Bit Maschinenworte gleichzeitig transferiert werden. Für die Beschleunigung von Schreiboperationen ist die Verwendung eines zusätzlichen Schreibpuffers vorgesehen (fehlt in Abbildung 6.2).

3. Zur Verarbeitung logischer Operationen sowie Integer-Additionen bzw. Subtraktionen steht eine *ALU* bereit.

4. Zusätzlich wird eine *Multiply Accumulate*-Einheit (MAC) für 32 und 64 Bit Multiplikationen bzw. MAC-Operationen bereitgestellt.

Darüber hinaus umfasst der adaptive Kern alle verfügbaren konfigurierbaren Bereiche, die als eigenständige rekonfigurierbare Funktionseinheiten integriert sind. Zudem existiert eine zusätzliche Speicherschnittstelle, die ausschließlich der Nutzung durch in die konfigurierbaren Bereiche eingelagerten Hardware-Module vorbehalten ist.

6.1.1.1.3 Back-End: Bis zu vier abgeschlossene Befehle können gleichzeitig vom adaptiven Kern an den *Reihenfolgepuffer* übergeben werden. Nachdem die Befehle in die von der Anwendung vorgegebenen Reihenfolge gebracht sind, erfolgt in einem nächsten Schritt das Zurückschreiben der Ergebnisse.

In aktuellen Prozessoren wird neben der Verwendung eines Reihenfolgepuffers auch eine *out-of-order Completion* praktiziert. Dies erfordert einen so genannten *roll-back*-Mechanismus, um im Falle von Unterbrechungen oder Ausnahmen eine Rückführung des Prozessorstatus in den Zustand zur Zeit derer Auslösung zu gewährleisten. Dieses Verfahren ist bei fest verdrahteten Prozessoren äußerst komplex und erfordert die Führung unterschiedlicher Ablauf- und Zustandshistorien. Der Einsatz im adaptiven Prozessor erhöht diese Komplexität zusätzlich. Obwohl es unwahrscheinlich ist, ist es dennoch möglich, dass während einer Unterbrechung eine Rekonfiguration stattfindet bzw. stattgefunden hat. Somit müsste bei einem *roll-back*-Verfahren speziell für diese Fälle, die Möglichkeit geschaffen werden, die Hardwarekonfiguration zum Zeitpunkt der Unterbrechung wieder herzustellen. Es kann nicht ausgeschlossen werden, dass hierbei ein ungepuffertes Modul direkt aus dem Funktionseinheitenpool, bzw.

Komponentenpool, geladen werden muss. Diesbezüglich ergeben sich Latenzen im Bereich von Mikro- bis hin zu Millisekunden. Demzufolge ist ein *roll-back*-Mechanismus für den adaptiven Prozessor abzulehnen.

6.1.1.2 Die Monitoringschicht

Abbildung 6.3 zeigt die logischen Komponenten der Monitoringschicht an den korrespondierenden Einheiten der Befehlsflussschicht. Jeder eingezeichnete Monitor besteht aus mindestens einem Sensor und einer zugehörigen Überwachungszelle. Zudem sind, in stark vereinfachter Form, die Verbindungen zwischen den einzelnen Überwachungskomponenten erkennbar. Die schraffierten Komponenten werden ausschließlich für die hardware-gestützte Modulrekonfiguration benötigt. Die Ereigniszählung mit Hilfe von Überwachungszellen arbeitet unabhängig von der Variante der Modulrekonfiguration. Der Ursprung der Information kann jedoch variieren. Entweder werden die *Instruktions-Register-Sensoren* (*I-R-Sensoren*) oder die *VM-Sensoren* zur Anzeige von Modulnutzungsereignissen eingesetzt. Darüber hinaus ist der gleichzeitige, gemischte Einsatz möglich. Beide Sensorentypen, sowohl die Instruktions-Register-Sensoren als auch die VM-Modulsensoren, signalisieren zum einen die mögliche Nutzung von Modulen und zum anderen die tatsächliche Nutzung. Letztere Information muss an die Anpassungskomponenten weitergeleitet werden, damit die entsprechenden Hardware-Module zur Ausführung bereitstehen und keine Verdrängung durch andere Module erfolgt. In den Überwachungszellen werden ausschließlich die Nutzungsmöglichkeiten registriert. Die tatsächliche Nutzung von Modulen wird mit Hilfe der *Nutzungsmonitore* erfasst. Die Generierung von Nutzungshistorien und zugehörigen Mustern erfolgt aus den Daten der Nutzungsmöglichkeit und der Nutzung. Entsprechend wertet die *Nutzungsauswertung* die gesammelte Information aus und übergibt die Ergebnisse an die *Nutzungssteuerung*. Beide gemeinsam bilden logisch die *Regelungseinheit*. Zusätzlich werden Überwachungsdaten über entstehende Wartezyklen mittels dem *Wartezyklenmonitor* an die Nutzungssteuerung übergeben. Die Regelung basiert auf den in Abschnitt 5.3 behandelten Häufigkeiten sowie Mustern bei zyklischem Anwendungsverhalten.

Der an den Instruktionszuteiler gekoppelte Wartezyklenmonitor liefert Daten über Wartezeiten, die durch die Einlagerung zusätzlicher Funktionseinheiten reduziert werden können. Instruktionen, die bereits in der Zuteilerwarteschlange liegen, liefern nur dann Wartezyklen, wenn die benötigte Funktionseinheit nicht rechenbereit ist. Die entstehenden Latenzen können durch das Nachladen entsprechender Module verringert bzw. beseitigt werden. Ob entsprechende rekonfigurierbare Ressourcen für diesen Zweck eingesetzt werden können, muss entsprechend unter Berücksichtigung der Modulnutzung bzw. -nutzungsmöglichkeit innerhalb der Nutzungssteuerung abgeglichen werden. Wartezyklen die sich innerhalb des Prozessor-Front-Ends ergeben, fließen – bis dato – nicht in die Regelung mit ein.

6.1.2 Dekodierung und Wandlung

Alternativ zur VM-basierten Erkennung der Modulnutzungsmöglichkeiten wurde eine ausschließlich hardware-basierte in den adaptiven Prozessor integrierte Variante untersucht. Durch die beschränkte Sichtweise des Prozessors auf den ausgeführten Anwendungscode kann die hardware-basierte Modulnutzungserkennung jedoch nur so genannte

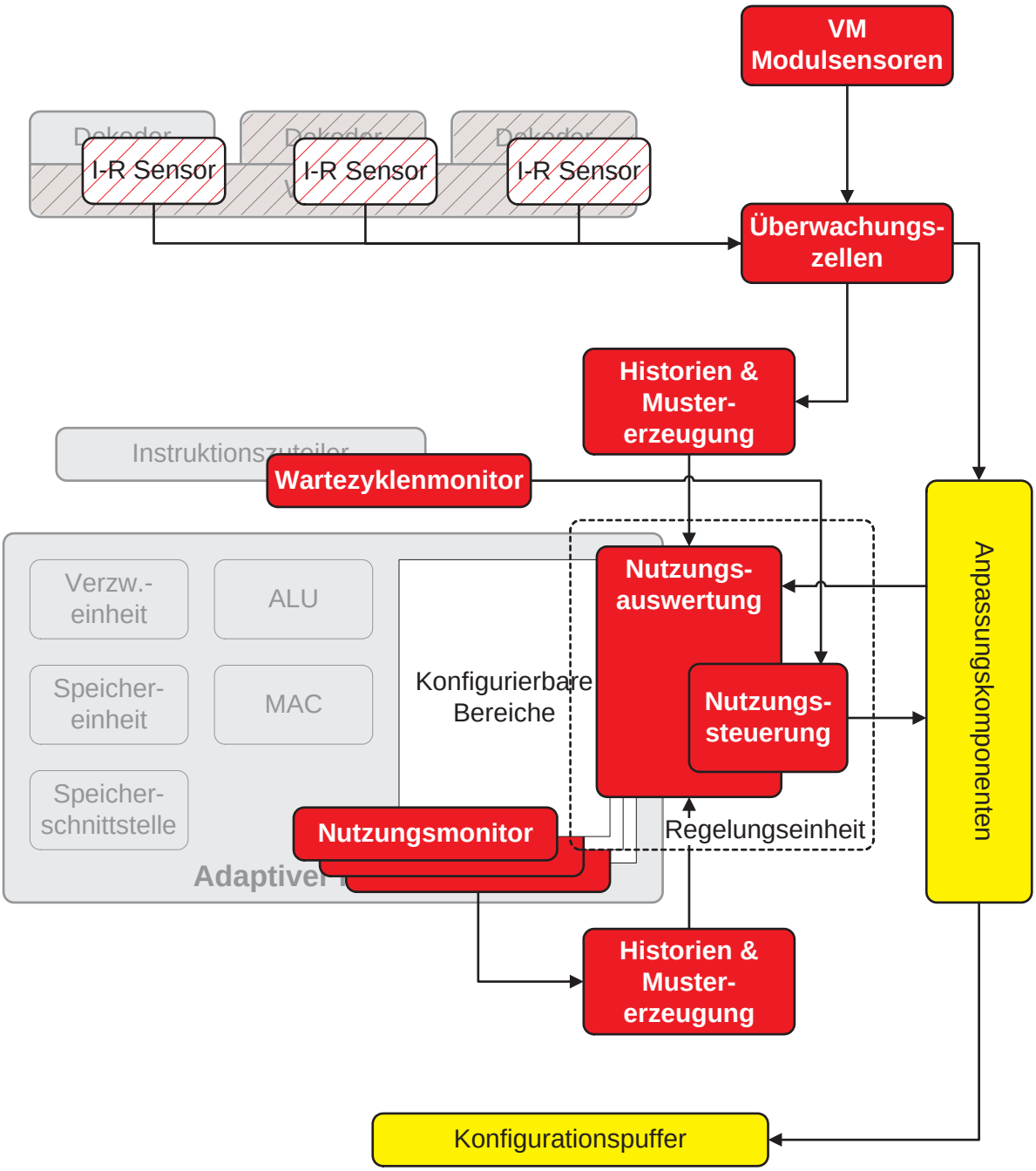

Abbildung 6.3: Komponentschema der Monitoringschicht

Operationskombinationen identifizieren. Operationskombinationen nutzen echte Registerabhängigkeiten sequentieller Maschinenbefehle (siehe Abbildung 6.4).

Hierbei wird versucht solche Sequenzen durch die Einlagerung geeigneter Hardware-Module zu beschleunigen. Ziel ist eine Optimierung der Anwendungsausführung ohne die Anwendung für den adaptiven Prozessor erneut zu übersetzen. Statische Codeanalysen zeigen, dass ca. 10% der erzeugten Maschinenbefehle in Form von Zweierkombinationen und ca. 1% in Dreierkombinationen zusammengefasst werden können. Diese Befehlssequenzen gilt es zur Laufzeit zu erkennen und durch geeignete Hardware-Module zu beschleunigen. Die Daten zur Identifikation der Operationskombinationen werden innerhalb der Dekodierungsphase des Prozessors gewonnen. Zusätzlich ist eine so genannte Wandlungsphase im Anschluss an die Befehlsdekodierung notwendig. Im Folgenden wird die Vorgehensweise, benötigte Erweiterungen und die Umsetzung der hardware-basierten Modulnutzungserkennung beschrieben.

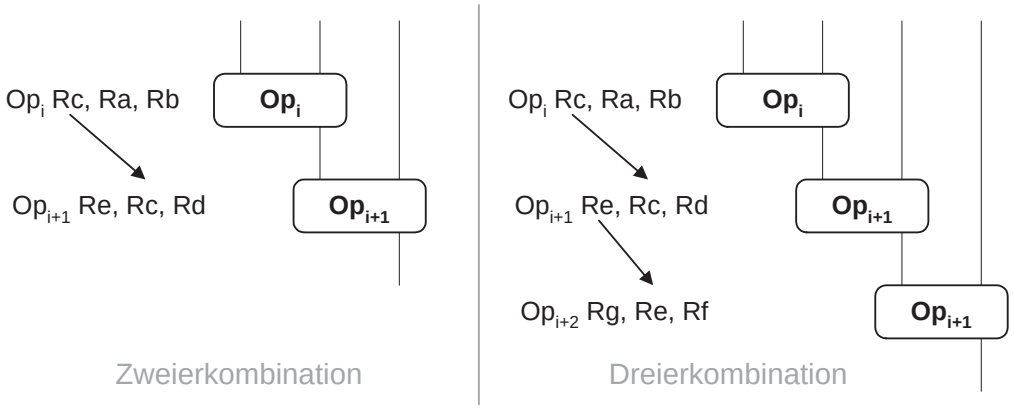

Abbildung 6.4: Einfaches Schema möglicher Operationskombinationen

Der Aufbau der Überwachung und Regelung zur optimierten Nutzung der rekonfigurierbaren Ressourcen (Abschnitt 6.2.5) ist nicht durch den Einsatz der Wandlungsphase beeinflusst. Der Ursprung der Überwachungsdaten über die Modulnutzungsmöglichkeiten ist jedoch nicht die VM-Schnittstelle, sondern liegt bei den drei Dekodern[7] des adaptiven Prozessors (Vgl. Abschnitt 6.1.1).

6.1.2.1 Prinzip der Operationskombinationen

Mit Hilfe ausgesuchter Beispiele wird das Prinzip der Operationskombinationen erschlossen und in den folgenden Abschnitten ausgearbeitet. Abbildung 6.5 zeigt fünf Fälle, welche häufig in dieser oder einer ähnlichen Art und Weise in Anwendungen anzutreffen sind[8]. Die linke Spalte zeigt jeweils einen ARM-Maschinenbefehl oder eine Sequenz. Korrespondierend findet sich in der mittleren Spalte eine vereinfachte Darstellung des internen Befehlswortes nach der Dekodierung. Der rechte Abbildungsteil enthält entsprechend die resultierenden internen Befehlsworte nach der Wandlung. In diesem Zusammenhang repräsentiert die linke Spalte das Wandlungsergebnis, falls keine Operationskombinationen möglich sind und jeder

[7] Drei Dekoder sind ausschließlich für die Unterstützung der Operationskombinationen notwendig.

[8] Die Maschinenbefehle sind aus Testanwendungen entnommen, die mittels *gcc* mit Codeoptimierung *-O3* übersetzt wurden.

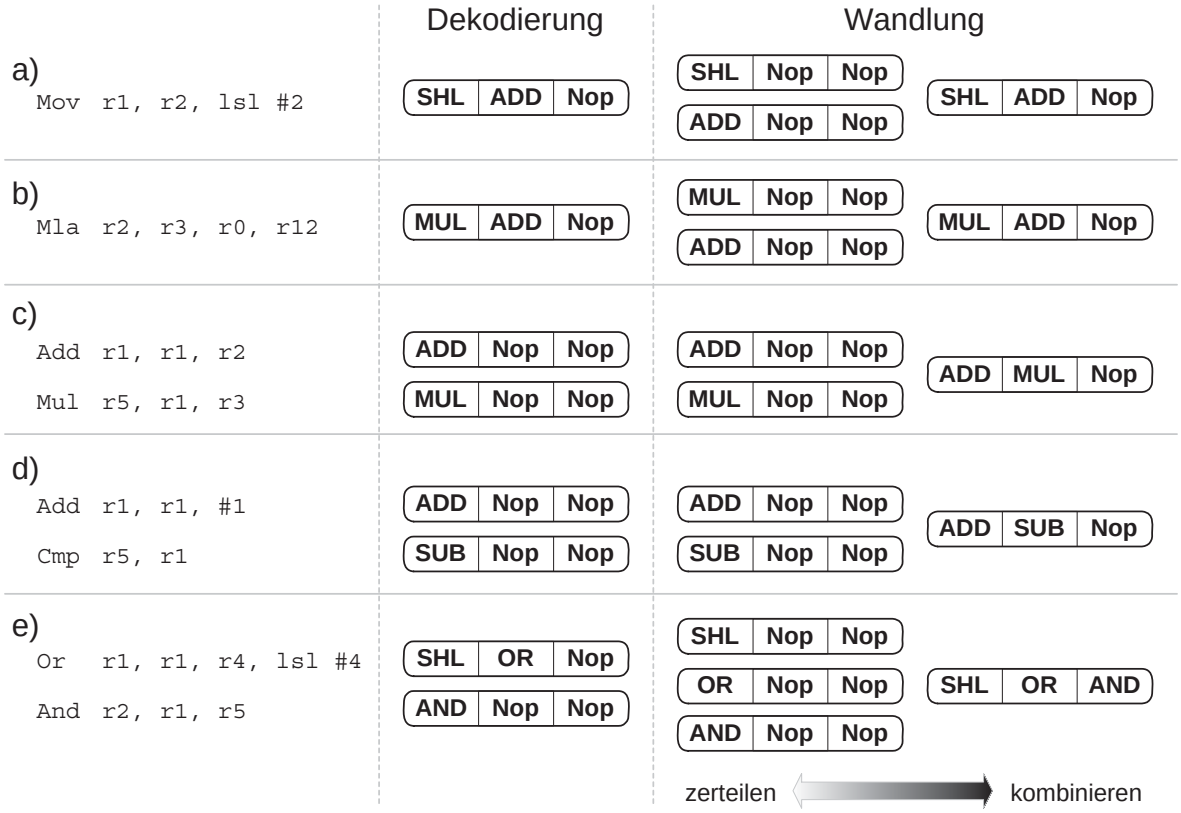

Abbildung 6.5: Dekodierung und Wandlung anhand ausgesuchter Beispiele

Maschinenbefehl in einzelne Operationen *zerteilt* werden muss. Die rechte Spalte hingegen illustriert eine Wandlung unter der Annahme, dass sämtliche Operationskombinationen durch die aktuelle Konfiguration des adaptiven Prozessors zugelassen sind und somit Operationen *kombiniert* werden.

Teilbild 6.5 a) behandelt einen `Mov`-Befehl, der für die Ausrichtung von Datenzugriffen auf Wort-Grenzen eingesetzt wird. Hierbei ist `r2` der Index auf ein Datum in einem Datenfeld. Für die Adressierung wird der Index mit vier multipliziert (`lsl #2`) und nach `r1` geschoben. Das interne Befehlswort trägt nach der Dekodierung die entsprechenden zwei Operationen. Die Position einer Operation im Befehlswort hängt von der Ausführungsreihenfolge ab. Ist der adaptive Kern mit einer Funktionseinheit bestückt, welche die Operationskombination verarbeiten kann, durchläuft das interne Befehlswort die Wandlungsphase unverändert. Falls keine Funktionseinheit dies leistet, wird eine Zerteilung in Einzeloperationen vollzogen und das ursprüngliche Befehlswort durch zwei ersetzt.

Für alle weiteren Beispiele gilt die Regelung für Zerteilung und Kombination in gleicher Form und wird nicht im einzelnen durchexerziert. Teilbild 6.5 b) zeigt einen *Multiply Accumulate*-Befehl der `r2 = r3 * r0 + r12` berechnet. In c) erkennt man eine Addition gefolgt von einer Multiplikation, die folgenden Ausdruck verkörpert: `r5 = (r1 + r2) * r3`. Die Befehlssequenz in Teilbild d) wird typischer Weise in Programmschleifen erzeugt, d.h. im konkreten Beispiel ist `r1` der Schleifenzähler und `r5` die zugehörige Begrenzung. Die in e) aufgeführten Maschinenbefehle ergeben sich bei Operationen auf Bitfeldern. Hier z.B. ein

Bitfeld `r4`, das um vier Bit nach links geschoben und anschließend in `r1` eingefügt wird. Der darauf folgende `And`-Befehl maskiert `r1` mit `r5` und legt das Ergebnis in `r2` ab.

Die Beispiele in Abbildung 6.5 verdeutlichen, dass die Zerteilung von Befehlen in der Regel zu einem erhöten Verarbeitungsaufwand führt. Im Gegensatz dazu verringert die Möglichkeit der Kombination den Aufwand. Die Verwendung des einen oder anderen Mechanismus ist eine Frage der Ausrichtung des Systems. Soll das System auf Leistung getrimmt werden, müssen demzufolge verhältnismäßig viele Operationskombinationen ausführbar sein. Wird mehr Wert auf die Systemeffizienz gelegt, muss zur Laufzeit eine Entscheidung über die Wichtigkeit und Nutzbarkeit einer entsprechenden Funktionseinheit getroffen werden. Dies bedeutet, dass nur ausgewählte Operationskombinationen unterstützt werden können und alle übrigen Befehle einer Zerteilung unterliegen.

6.1.2.2 Anforderungen an das interne Befehlswort

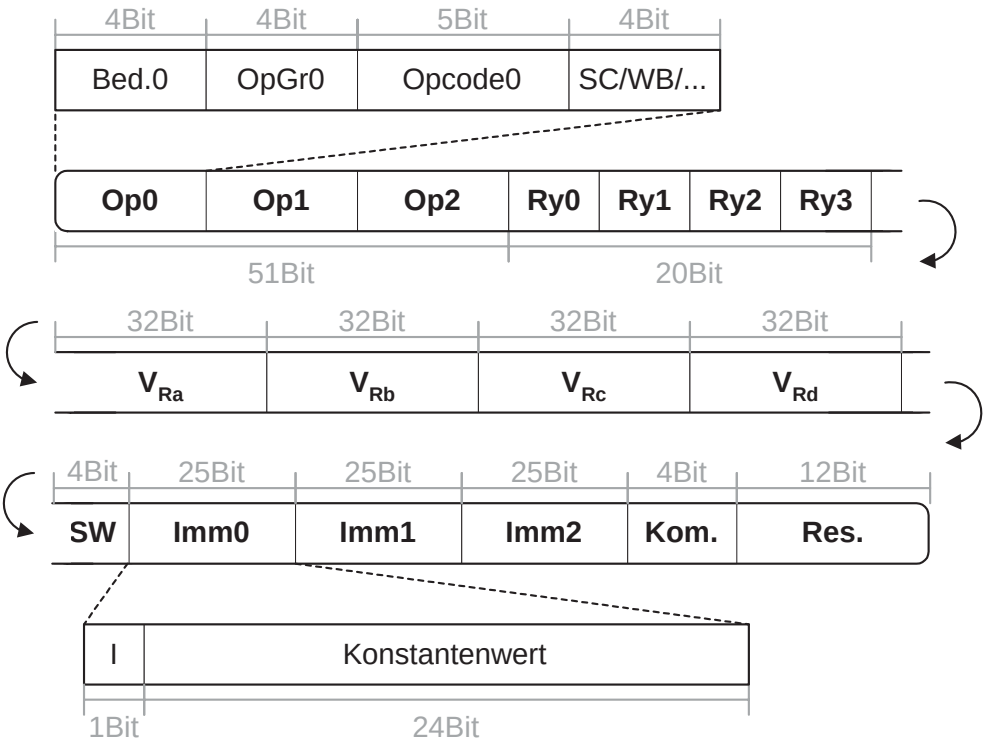

Abbildung 6.6: Schematische Darstellung des erweiterten internen Befehlswortes

Um Befehlssequenzen von bis zu drei Maschinenbefehlen bilden zu können, muss das interne Befehlswort, wie in Abbildung 6.6 dargestellt, aufgebaut sein. Die Länge des vorliegenden Befehlswortes beläuft sich auf 294 Bit. Es werden bis zu drei Operationen (*Op0, Op1, Op2* aus unterschiedlichen Operationsgruppen (*OpGr*) mit unterschiedlichen Bedingungen (*Bed.*) unterstützt. Jede Operation benötigt zusätzliche Information über das Verhalten in Bezug auf Setzen von Statusbits und das Verhalten beim Zurückschreiben von Ergebnissen (*SC/WB*). Insgesamt sind vier Bits vorgesehen, wobei vorerst nur zwei genutzt werden. Das erweiterte Befehlswort erlaubt bis zu vier Ausgangsregisterindices (*Ry0, ..., Ry3*) sowie vier Eingangsregisterwerte ($V_{Ra}, ..., V_{Rd}$). Drei zusammengefasste Operationen können zusätzlich

bis zu drei Konstantenwerte (*Imm*) nutzen. Die Nutzung eines Konstantenwerts wird jeweils durch das höchstwertige Bit angezeigt. Darüber hinaus sind zwei 4 Bit Steuersignale (*SW* und Komm.) für die Abarbeitung möglicher Operationskombinationen notwendig (siehe Abschnitt 6.1.2.4). Weitere zwölf Bits sind bisher für Erweiterungen reserviert (*Res.*).

6.1.2.3 Technische Aspekte der Operationskombinationen

Die Ergebnisse der STRATIX II Synthesen werden zur Analyse des Ressourcenbedarfs der Operationskombinationen und des zeitlichen Verhaltens herangezogen. Tabelle 6.1 zeigt einen ausgewählten Teil implementierter Operationseinheiten und deren resultierende Taktfrequenzen:

Kombination	Bezeichnung	ALUTs	MHz
Single	Log. Op.	143	340
	Add./Sub.	106	260
	Log. Sh.	166	210
	Arith. Sh.	171	210
	Multiplikation	1202	150
Dual	Log. + Log.	183	330
	AS + AS	113	180
	Log + AS	109	220
Tripel	Log. + Log. + Log.	210	270
	AS + AS + AS	179	130

Tabelle 6.1: Auszüge aus Syntheseberichten von Operationskombinationen

Alle verfügbaren logischen Operationen wurden in einer Einheit, die als *Log. Op.* abgekürzt ist, zusammengefasst. Des Weiteren sind kombinierte Addierer und Subtrahierer, sowie eine logische und eine arithmetische Schiebeeinheit und eine Multiplikationseinheit aufgelistet. Die Operationseinheiten sind in *Single*, *Dual* und *Triple* unterteilt. *Single* repräsentiert eine Operationseinheit, die nur eine Operation bzw. Operationsgruppe in einem Zug abarbeitet. *Dual* entsprechend, erlaubt die Ausführung einer Operationskombination. Eine Operationskombination aus drei Einzeloperationen kann mit den als *Triple* bezeichneten Einheiten verarbeitet werden.

Für einen realen Einsatz sind Taktfrequenzen von 200MHz und mehr anzustreben. Betrachtet man Tabelle 6.1, erkennt man, dass alle Varianten logischer Operationen oberhalb dieser Marke liegen. Bei den Addierern und Subtrahierern ist bereits eine Zweierkombination ein Grenzfall. Für die beiden Arten von Schiebeoperationen liegen die Ergebnisse mit einer einzelnen Operation am Limit[9].

Konkret bedeuten diese experimentellen Ergebnisse, dass bei Operationskombinationen logischer Operationen die Ausführungsphase – im schlechtesten Fall einer Operation, im besten Fall von zwei Operationen – überdeckt wird, d.h. für deren Ausführung keine zusätzlichen

[9]Die Implementierungen wurden ohne optimierende Maßnahmen durchgeführt, d.h. speziell bei den Addierern und Subtrahierern ist eine Zweierkombination unter Einhaltung der Zeitkriterien durchaus vorstellbar.

Prozessortakte benötigt werden. Aufwendigere Operationskombinationen müssen jedoch in mehreren Takten abgearbeitet werden. Die Anzahl der Takte hängt von der Anzahl der kombinierten Operationen ab.

Zusätzlich kann festgehalten werden, dass sich die auf einem Prozessor ausführbaren Einzeloperationen durch deren Komplexität und die resultierenden Zeitkriterien in folgende Operationsgruppen unterteilen:

- Alle logischen Operationen
  - AND, NAND, OR, EOR und NOT
- Einfache arithmetische Operationen auf Ganzzahlwerten
  - Addition und Subtraktion
- Schiebe- und Rotationsoperationen
  - Logischer SHL und SHR
  - Arithmetischer SHL und SHR
  - Rotation
- Komplexe arithmetische Operationen auf Ganzzahlwerten
  - Multiplikation und Division
- Die Gruppe von Gleitkommaoperationen

6.1.2.4 Verhalten und Funktionalität von Operationskombinationen

Berücksichtigt man das erweiterte interne Befehlsformat und die obigen Varianten der Operationskombinationen, gelangt man zu folgenden prinzipiellen Anforderungen für diejenigen Hardware-Module die Operationskombinationen unterstützen:

- Ausführung einer Operation oder Operationssequenz in Ein- und Mehrzyklenverfahren.
- Verarbeitung einer einzelnen Operation sowie Sequenzen von zwei und drei Operationen.

6.1.2.4.1 Klassen von Operationskombinationen: Bisher wurde stets davon ausgegangen, dass Operationskombinationen von zwei und drei Operationen vom adaptiven Prozessor erkannt und mit Hilfe entsprechender Funktionseinheiten effektiv abgearbeitet werden können. Damit dies tatsächlich erreicht wird, sind verschiedene funktionale Eigenschaften der möglichen Operationen und Probleme, die unter Umständen bei der Kopplung von Operationen auftreten, zu beachten.

Die Festlegung des internen Befehlswortes liefert die grundsätzlichen Rahmenbedingungen für die einzelnen Module zur Operationskombinationsverarbeitung des adaptiven Prozessors (Abbildung 6.7):

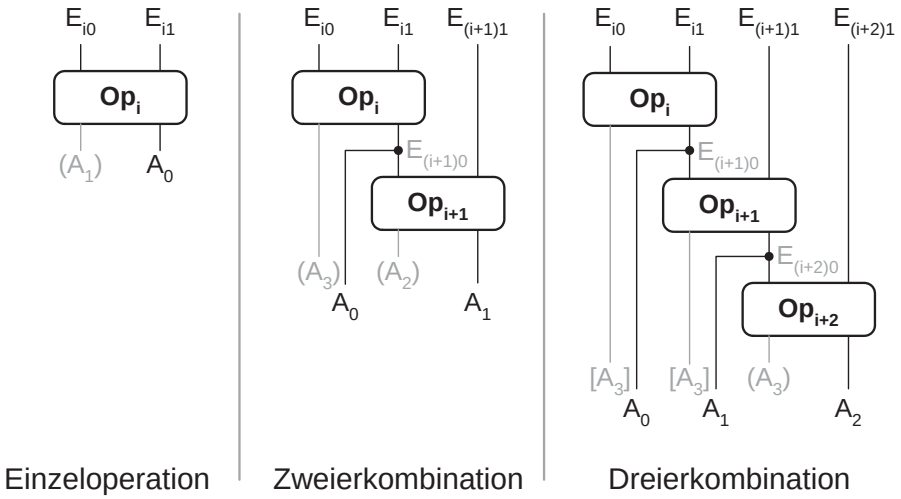

Abbildung 6.7: Rahmenbedingungen für Operationskombinationen

- 1 - 4 32 Bit Eingangswerte
- 1 - 4 32 Bit Ausgangswerte und korrespondierende Registerindices
- 1 - 3 Operationen die auf den Eingangswerten ausgeführt werden oder keine Operation (*NoOp*)

Die Ein- und Ausgänge von Modulen mit Operationskombinationen werden folgendermaßen gekennzeichnet:

$$E_{Fn} \; und \; A_m \; mit \; n \; \epsilon \; \{0,1\} \; und \; m \; \epsilon \; \{0,1,2,3\}$$

F gibt die Operation bzw. Funktion an, der der Eingang E zugeordnet ist. n kennzeichnet, ob es sich um den ersten oder zweiten Eingang einer Operation handelt. Die Ausgänge A werden von 0 bis 3 durchnummeriert. Sind mehr als vier Ausgänge möglich, kann diese Operationskombination auf Grund des internen Befehlswortes nicht umgesetzt werden.

6.1.2.4.2 Funktionalität der Operationskombinationsausführung: Beim Zusammenfassen zweier Einzeloperationen der Form $g(f(r_a, r_b), r_c)$, wobei an E_{f0} r_a und an E_{f1} r_b anliegt, können folgende zwei Probleme auftreten:

1. Beide Eingänge von g beziehen sich auf das Ergebnis von f, d.h. r_c referenziert ebenfalls das Ergebnis von f und somit den Ausgangswert A_{f0}, was im Weiteren als **Eingangsidentitätsproblem** bezeichnet wird. Exemplarisch kann man hier den Ausdruck $(a+b)^2$, mit $f : r_{y0} = r_a + r_b$ und $g : r_{y1} = r_{y0} \cdot r_{y0}$, angeben.
2. Das **Kommutativitätsproblem** entsteht dann, wenn $g(f(r_a, r_b), r_c) \neq g(r_c, f(r_a, r_b))$. Zum Beispiel falls es sich bei g um eine Subtraktion oder Division handelt.

Beide Probleme können durch die in Abbildung 6.8 veranschaulichten Maßnahmen gelöst werden. Diese zusätzliche Funktionalität von Funktionseinheiten wird für den Einsatz im

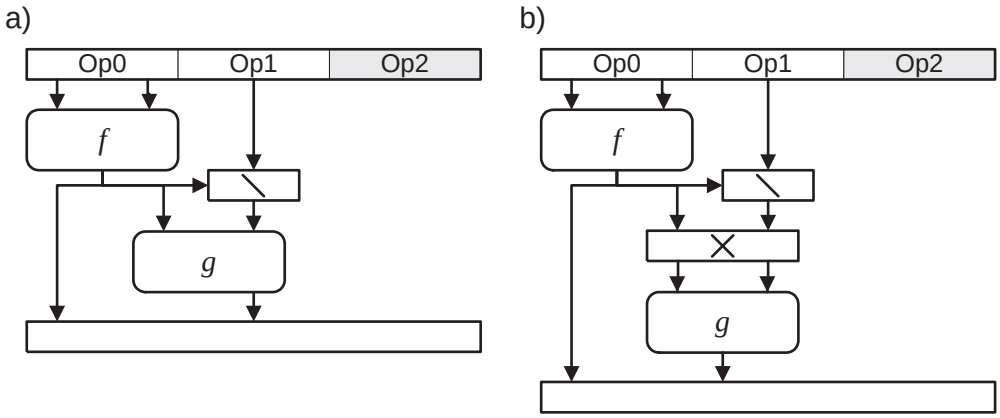

Abbildung 6.8: Anforderungen an interne Abläufe von Funktionseinheiten

adaptiven Prozessor vorausgesetzt, da so bei der Verwendung von Operationskombinationen stets nur die Reihenfolge von Operationen und nicht zusätzlich die der Register geprüft werden muss.

Abbildung 6.8 a) zeigt die Auflösung des Eingangsidentitätsproblems bei kommutativen Operationen. Die Funktionseinheit muss lediglich über einen Demultiplexer verfügen, der den zweiten Eingang der Folgeoperation zusätzlich mit dem Ergebniswert der ersten Operation versorgt. Um während der Ausführungsphase keine Auswertungen der Eingangsidentität vornehmen zu müssen, sind diese Informationen bereits innerhalb der Wandlungsphase zu erzeugen und mittels des internen Befehlsworts bereit zu halten. In Abbildung 6.8 b) ist die Lösung bei nicht kommutativen Operationen illustriert. Hierbei ist ebenfalls ein Demultiplexer und darüber hinaus ein Vertauscher notwendig. Die Steuersignale beider eingefügter Komponenten werden gleichfalls bei der Wandlung erzeugt.

6.1.3 Leistungsbewertung und Vergleich mit der Basisarchitektur

Obwohl der Befehlssatz des adaptiven Prozessors auf der Befehlssatzarchitektur des ARMv4-Kerns beruht, ist der interne Aufbau an den ARMv6-Kern, der unter anderem in den ARM10-Prozessoren zu finden ist, angelehnt. Für die Leistungsbewertung der grundlegenden Architektur des adaptiven Prozessors (Nutzung einer einzelnen Pipeline und ohne den Einsatz der Hardware-Module) wurden dementsprechend als Vergleichswerte die Angaben der EEMBC-Benchmarks für den ARM10-Prozessor herangezogen.

Das Diagramm 6.9 zeigt für vier Beispielanwendungen den erzielten Befehlsdurchsatz des adaptiven Prozessors im Vergleich zum ARM10. Alle Angaben liegen in Prozent vor und sind auf die Angaben des ARM10 normiert. Der jeweils linke Balken repräsentiert die, aus der Simulation (siehe Anhang A) gewonnen, Leistungsdaten des adaptiven Prozessors. In Bezug auf den eingesetzten Simulator existieren zwei Einschränkungen, deren Auswirkungen sich rechnerisch ermitteln lassen und jeweils die rechten Balken in Diagramm 6.9 ergeben: Zum einen unterstützt der Simulator aktuell eine statische Sprungvorhersage und zum anderen werden Änderungen des Befehlszählers, die nicht auf Sprungbefehle zurückzuführen sind, in der Sprungvorhersage ignoriert. Das Auftreten beider kann anhand der Simulation eindeu-

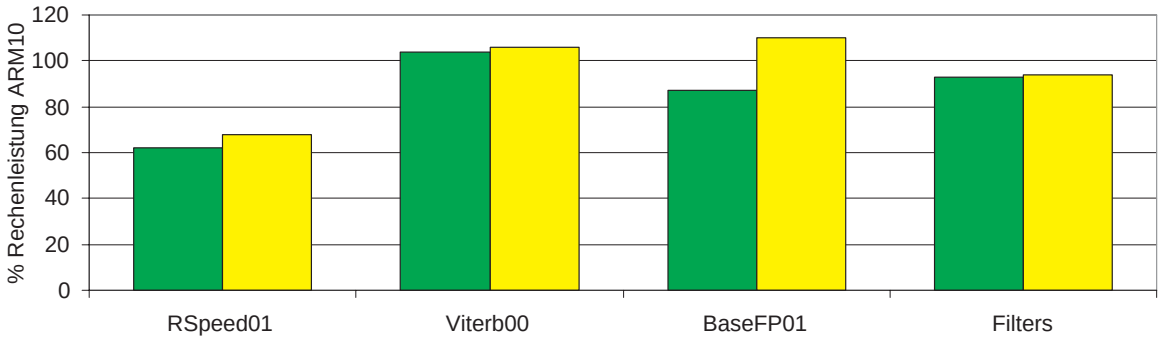

Abbildung 6.9: Vergleich adaptiver Prozessor und ARM10

tig identifiziert und zahlenmässig erfasst werden. Darüber hinaus sind die diesbezüglichen Latenzen des ARM10-Prozessors verfügbar (Vgl. [6]) und die entsprechenden simulierten Latenzen bekannt.

Die Abweichung der simulierten Leistungsmessung des adaptiven Prozessors liegt generell im Bereich von ±10%. In diesem Zusammenhang sind Messwerte, die eine bessere Leistung im Vergleich zum ARM10 aufweisen, auf Speicherzugriffe zurückzuführen. Speicherzugriffe werden in der Simulation stets als Zugriffe auf den Cache durchgeführt. Des Weiteren erkennt man in Diagramm 6.9 für die Anwendung *RSpeed01* eine stark verminderte Rechenleistung (61,9% simuliert und 68,1% rechnerisch). Dies ist auf das Fehlen eines *Register Renaming*-Mechanismus in der Simulation zurückzuführen. Situationen, die *Register Renaming* nutzen könnten, werden in der Simulation bis dato nicht erfasst. Aus diesem Grund ist auch eine rechnerische Ermittlung nicht möglich. Detaillierte Ausführungen zum verwendeten Simulator sind in Anhang A nachzulesen.

6.2 Implementierungsaspekte des adaptiven Prozessors

Dieser Abschnitt beschreibt die notwendigen Veränderungen und Erweiterungen des zugrunde liegenden ARMv6-Prozessorkerns. VHDL-basierte Implementierungen des ARMv6-Kerns werden z.B. von der Firma Altera unter dem Namen Nios bzw. Nios II angeboten (Vgl. Grundlagenabschnitt 2.2). Diese eignen sich als Basisimplementierung für den adaptiven Prozessor. Diesbezüglich müssen der Prozessorkern zum adaptive Kern ausgebaut und die Überwachungs- und Regelungskomponenten integriert werden. Unabhängig davon bedarf es zur Nutzung der Operationskombinationen – und ausschließlich derer – Änderungen am Front-End des Prozessors.

6.2.1 Integration der Wandlung

Zur Unterstützung der hardware-basierten Nutzungserkennung von Modulen mit Operationskombinationen muss die Wandlungsphase im Prozessor-Front-End direkt an die Dekodierer angeschlossen werden. Hierfür werden drei Dekodierer benötigt. Die VM-basierte Nutzungserkennung sowie die Überwachung und Regelung sind von diesen Änderungen in keinem Fall betroffen.

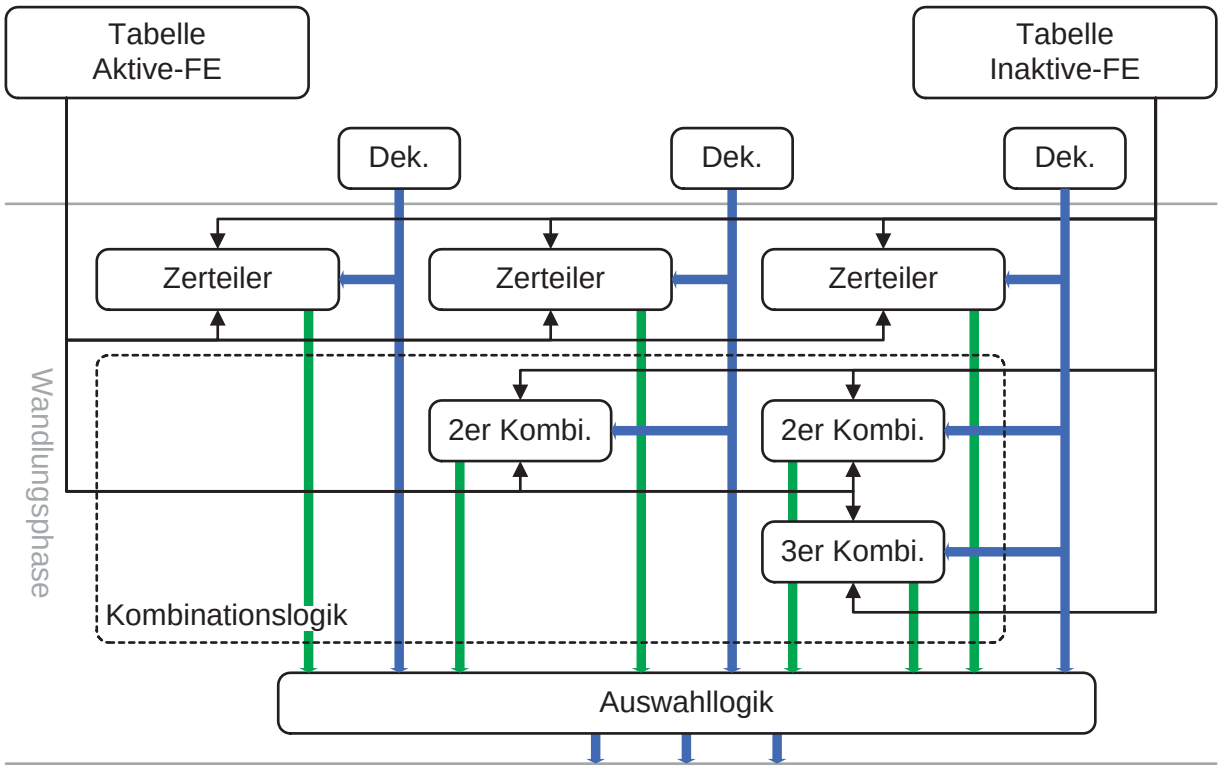

Abbildung 6.10: Dekodierung mit anschließender Wandlung in Komponentendarstellung

Abbildung 6.10 schematisiert die Dekodierung mit anschließender Wandlung. Wohingegen die Bildung von Operationskombinationen ausschließlich der Leistungssteigerung dient, ist

die Zerteilung ein essentieller Bestandteil des Prozessor, der die laufende Verarbeitung von Maschinenbefehlen sicherstellt. Das heißt, kann ein Befehl nicht direkt von einer Funktionseinheit abgearbeitet werden, muss die Zerlegung in etwaige Einzeloperationen diesbezüglich Abhilfe schaffen.

Man erkennt in Abbildung 6.10, dass jeder Ausgang der Dekodierkomponenten jeweils mit einer Zerteilerkomponente verbunden ist. Zudem verfügen die Zerteilereinheiten über eine umfassende Sichtweise in Bezug auf aktive zurzeit eingelagerte Hardware-Module – mittels der *Tabelle aktive-Funktionseinheiten* – sowie den für das System insgesamt zugänglichen Modulen des Funktionseinheitenpools über die *Tabelle inaktive-Funktionseinheiten*. Beide Tabellen sind eine lokale und eingeschränkte Kopie der Systembibliothek. Ausschließlich diejenigen Module, die sich als Funktionseinheiten zur Ausführung von Operationskombinationen eignen, werden dort abgelegt. Zum Beispiel Informationen über eine etwaige Software-Ausführung, wie diese die Systembibliothek bereithält, fehlen in den Funktionseinheitentabellen. Die Kopplung der Zerteilungs- und Kombinationseinheiten an die Tabelle aktiver Funktionseinheiten dient der Observation des tatsächlichen Nutzungsverhaltens eines Moduls. Dementgegen liefert die Anbindung an die Tabelle inaktiver Funktionseinheiten die Daten über mögliche Modulnutzungen.

Kann ein Zerteiler das interne Befehlswort, dass er aus seinem zugeordneten Dekodierer erhält, weder direkt, noch durch eine Zerlegung in Einzeloperationen, einer Funktionseinheit zuordnen, liegt ein ungültiges Befehlswort vor. Wird für die Befehlsverarbeitung ein Modul aus den inaktiven Funktionseinheiten identifiziert, muss dem adaptiven Kern signalisiert werden, dass dieser mit höchster Priorität die entsprechende Einheit einlagert. Diese Situation tritt theoretisch dann auf, wenn bei der Ausführung einer oder mehrerer Anwendungen über einen *gewissen* Zeitraum eine Befehlsgruppe nicht benötigt wird und deshalb aus dem adaptiven Kern verdrängt oder erst gar nicht eingelagert wurde.

Für die Kombination von Operationen wird eine Reihe von Komponenten eingesetzt. Das Erkennen von Zweierkombinationen erfordert die Kopplung an jeweils zwei Dekoder. Die Zusammenfassung von drei Befehlen muss über drei interne Befehlsworte aus drei Dekodern erfolgen. Dadurch, dass der ARM-Befehlssatz Operationskombinationen von zwei Operationen – in der Regel Schiebeoperationen in Verbindung mit arithmetischen Operationen – standardmäßig bereitstellt, können in diesen Fällen Dreierkombinationen bereits durch die Verknüpfung von zwei internen Befehlsworten entstehen und dementsprechend Zweierkombinationen durchaus nach der Dekodierung – ohne weitere Wandlung – vorliegen. Dies gilt es, bei der Wandlung von Operationen zu beachten.

Die *Auswahllogik* ist die Ausgangskomponente in Abbildung 6.10. Anhand der in den Zerteilungs- und Kombinationskomponenten gewonnenen Daten wird die Entscheidung gefällt, welche Befehlsworte in Richtung Prozessorkern weitergeleitet werden. Entsprechend wird das *ungewandelte* Befehlswort des Dekodierers, mehrere Befehlsworte nach einer Zerteilung oder ein kombiniertes Befehlswort weitergegeben.

6.2.1.1 Die Mechanismen der Wandlung

Die Wandlerphase ist für zwei Arten der Weiterverarbeitung eines dekodierten Maschinenbefehls verantwortlich:

1. Die *Zerteilung* eines internen Befehlswortes, um gegebenfalls eine Abarbeitung mittels Einzeloperationen zu ermöglichen.

2. Die *Kombination* von zwei oder mehreren Befehlsworten zur Ausführung in speziell dafür vorgesehenen Funktionseinheiten.

Der Zerteilungsmechanismus

Dass während der Zerteilung eine Zerlegung des dekodierten Befehlswortes erfolgt, wurde bereits erläutert. Für den realen Einsatz im adaptiven Prozessor reicht dies jedoch nicht aus. Das Problem, welches in diesem Kontext auftreten kann, beruht auf der Forderung Hardware-Module mit Operationskombinationsunterstützung so flexibel als möglich einzusetzen. Das heißt, um den Nutzungsgrad solcher Module zu verbessern, müssen diese in der Lage sein neben den Operationskombinationen auch die korrespondierenden Einzeloperationen auszuführen. Konkret bedeutet dies, dass bei der Ausführung von Einzeloperationen auf Modulen mit Operationskombinationen das interne Befehlswort adäquat aufbereitet werden muss. Der einfachste Weg das zu realisieren, ist die Ersetzung nicht genutzter Operationsfelder des Befehlswortes durch *No Operation (Nop* oder *NoOp)* Anweisungen[10].

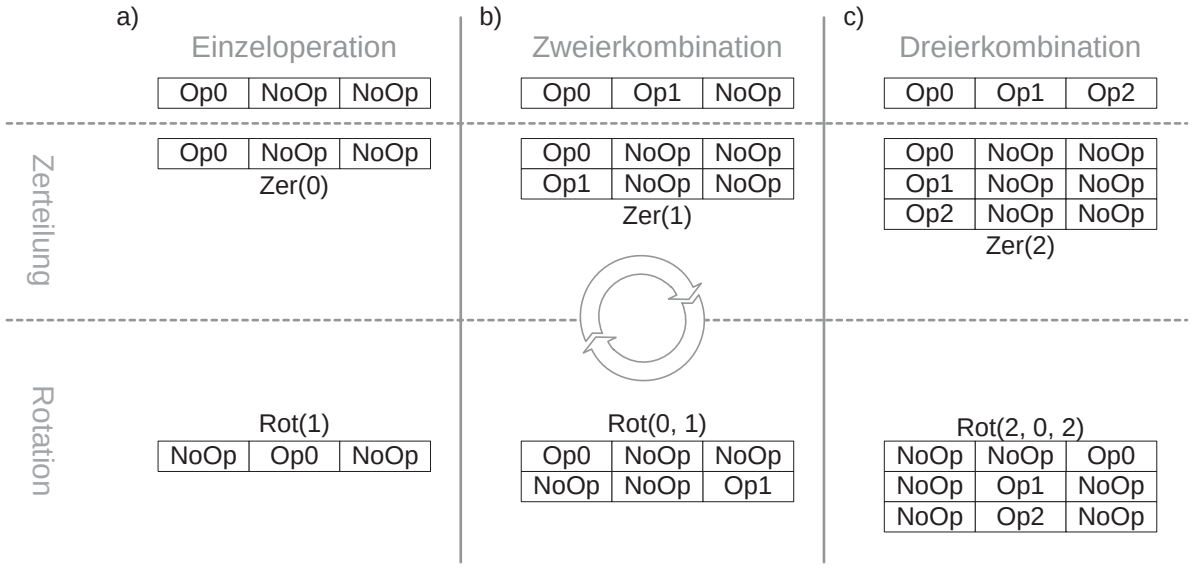

Abbildung 6.11: Der Zerteilungsmechanismus in der Praxis

Abbildung 6.11 zeigt jeweils exemplarisch ein Ergebnis der Wandlung für eine Einzeloperation, eine Zweierkombination und eine Dreierkombination beliebiger Operationen die als *Op0*, *Op1* und *Op2* bezeichnet sind. Der Zerteilungsmechanismus wird in zwei Schritten durchgeführt, der Zerlegung in Einzeloperationen und eine anschließende Rotation der resultierenden Befehlsworte. Die Rotation sorgt dafür, dass ein internes Befehlswort direkt an ein Modul mit Operationskombinationsverarbeitung weitergeleitet werden kann, ohne dass zusätzliche Veränderungen vorgenommen werden müssen. Dies erlaubt die Ausführung von Einzeloperationen auf Einheiten, die eigentlich nur Operationskombinationen unterstützen.

[10]Dies ist an die Vorgehensweise bei VLIW/EPIC Architekturen angelehnt.

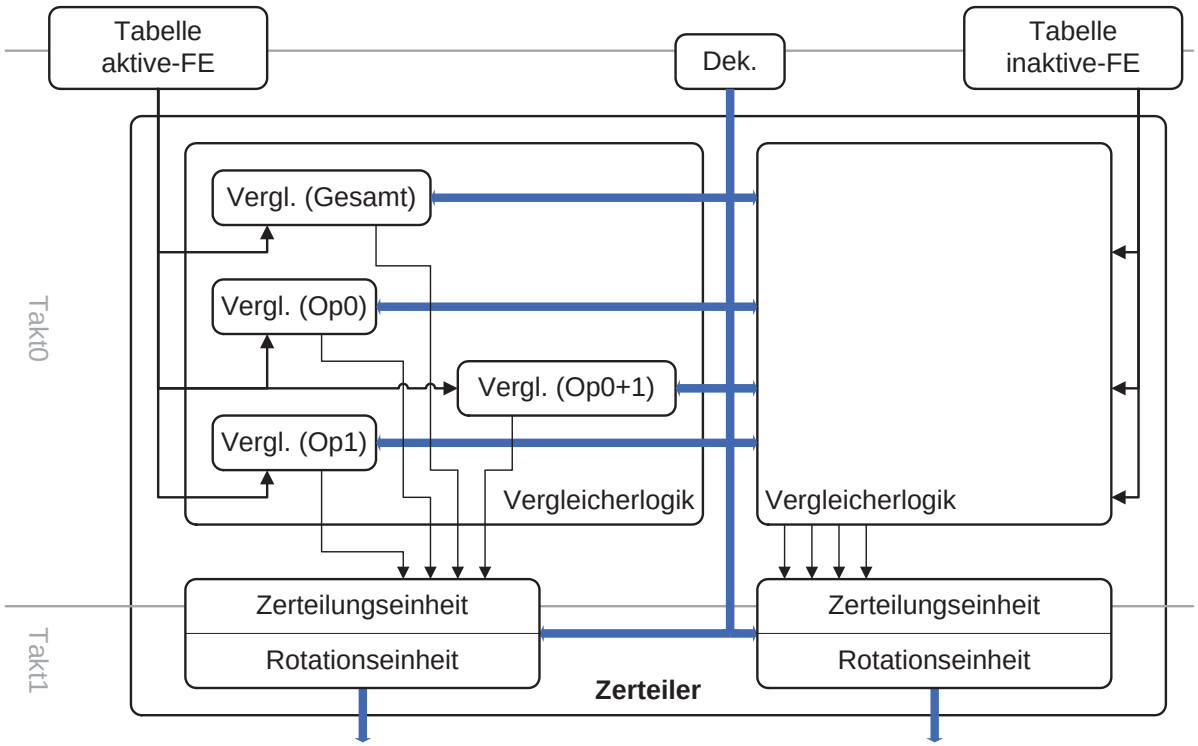

Abbildung 6.12: Die logischen Komponenten der Zerteilung

Sowohl der Ablauf der Zerteilung als auch die der Rotation sind in Abbildung 6.11 anhand dreier Beispiele dargestellt. Für die Zerteilung ist jeweils der Grad der angewendeten Zerteilung angegeben. Möglich sind Werte aus $\{0, 1, 2\}$ entsprechend der Anzahl der vorgenommenen Zerteilungen des ursprünglichen Befehlswortes. Rotationen können auf jedem einzelnen der nach der Zerteilung vorliegenden Befehlsworte ausgeführt werden. Es wird stets eine Rechtsrotation der Operationsfelder durchgeführt. Eine Rotation eines Befehlswortes um drei Felder liefert das ursprüngliche Befehlswort. Demzufolge sind für die Rotation ebenfalls Werte aus $\{0, 1, 2\}$ sinnvoll.

Die Zerteilung einer Einzeloperation (Abbildung 6.11 a) führt zu keiner Veränderung und liefert deshalb das ursprüngliche Befehlswort. Im Beispiel aus Abbildung 6.11 erfolgt anschließend eine Rotation um 1, d.h. die Operation *Op*0 wird um eine Position nach rechts geschoben. Die ursprünglich letzte NoOp erhält die erste Position im Befehlswort. Aus Sicht der Hardware-Module verkörpert das Befehlswort nun eine Zweierkombination aus einer NoOp und der Operation *Op*0 oder eine Dreierkombination aus NoOp, *Op*0 und NoOp. Eine Rotation anzuwenden ist nur dann sinnvoll, wenn keine im adaptiven Kern befindliche Funktionseinheit in der Lage ist, das Befehlswort nach der Zerteilung in unveränderter Form zu verarbeiten. Darüber hinaus ist zu beachten, dass nur NoOp über das rechte Ende eines Befehlswortes hinausgeschoben und auf der linken wieder hineingeschoben werden darf. Bei anderen Operationen würde dies zu einer Veränderung der Ausführungsreihenfolge und demzufolge einer Verfälschung der Berechnungen führen, da Operationskombinationen auf der Verarbeitung von Operationen mit echten Registerabhängigkeiten beruhen.

Exemplarisch illustriert Abbildung 6.11 zwei weitere Möglichkeiten, eine Zweierkombination und eine Dreierkombination jeweils komplett zu zerlegen und zu rotieren. Dreierkombinationen können bei der Verwendung des ARM-Befehlssatzes nach der Dekodierung nicht vorliegen. Das Prinzip der Zerlegung und Rotation wäre jedoch auch in einem solchen Fall anwendbar.

Abbildung 6.12 skizziert die logischen Komponenten des Zerteilungsmechanismus und deren wichtigsten Datenpfade. Der Aufbau des Zerteilers zeigt sich symmetrisch, entsprechend der Kopplung an die Tabelle aktiver bzw. inaktiver Funktionseinheiten. Der Unterschied beider Hälften liegt im Inhalt der angeschlossenen Tabellen. Darüber hinaus muss der Zerteiler den adaptiven Kern informieren, falls ein Befehl nur durch ein Modul der inaktiven Funktionseinheiten abgearbeitet werden kann. Der Kern muss dementsprechend mit einer Einlagerung reagieren. Beide Hälften müssen parallel abgearbeitet werden. Eine sequentielle Schaltung erfüllt die geforderten Zeitkriterien (> 200 MHz) nicht. Am linken Rand der Abbildung ist eine Unterteilung in *Takt 0* und *Takt 1* angefügt. Für die Implementierung hat sich gezeigt, dass die Vergleiche im Taktzyklus der Dekodierung erfolgen müssen. Die Zerteilung und Rotation wird im darauf folgenden Takt durchgeführt.

Die Vergleicherlogik: Alle Vergleicher in Abbildung 6.12 werden zu einer logischen Einheit zusammengefasst, die als *Vergleicherlogik* beschriftet ist. Es sind vier Vergleichskomponenten notwendig, die wiederum aus einer Reihe von einzelnen Vergleichern, in Abhängigkeit von der maximalen Anzahl von Tabelleneinträgen, bestehen. Der *Gesamtvergleicher* liefert als Ergebnis, ob das Befehlswort unverändert ausgeführt werden kann. Zwei Vergleicher testen die zwei einzelnen Operationen des Befehlswortes. Ein weiterer Vergleicher überprüft

die mögliche Ausführung der ersten zwei Operationen als Operationskombination, falls eine Rotation angewendet wird.[11].

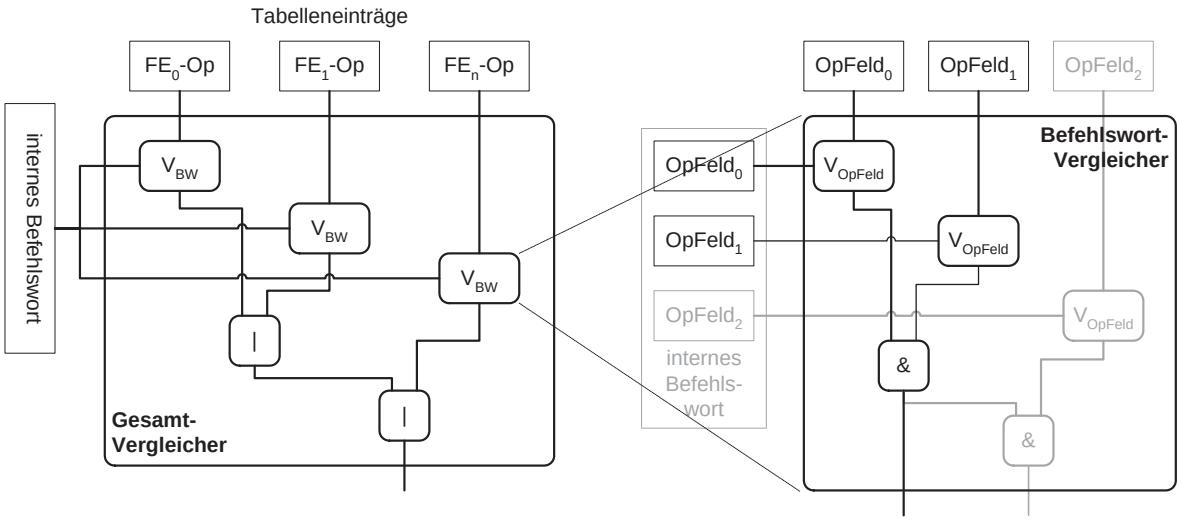

Abbildung 6.13: Aufbau des Gesamtvergleichers

Jedes Befehlswort erfährt nach der Dekodierung einen Vergleich auf Übereinstimmung mit den, durch die verfügbaren Module unterstützten, Befehlsworten, wie in Abbildung 6.13 dargestellt. Es werden *Befehlswortvergleicher* in der Anzahl der Funktionseinheiten einer Tabelle benötigt. Zur Vereinfachung sind in Abbildung 6.13 Vergleicher für nur drei Funktionseinheiten skizziert. Das Ergebnis eines Befehlswortvergleichers ist ein 1 Bit Signal. Das untersuchte Befehlswort kann dann unverändert ausgeführt werden, falls der Vergleich mit einer einzigen Funktionseinheit positiv verläuft. Der rechte Teil der Abbildung zeigt den Aufbau der verwendeten Befehlswortvergleicher aus einzelnen *Operationsfeldvergleichern*.

Die drei Operationsfelder müssen mit den Daten der Funktionseinheiten verglichen werden. Falls alle drei Vergleiche erfolgreich sind, wird eine Übereinstimmung angezeigt. Das dritte Operationsfeld muss bei der Verwendung des ARM-Befehlssatzes nicht beachtet werden, weshalb die zugehörige Logik grau gezeichnet ist.

Den Aufbau der Operationsfeldvergleicher skizziert Abbildung 6.14. Jeder Operationsfeldvergleicher besteht aus einem *Operationsgruppenvergleicher* und einem *Operationsvergleicher*. Nur falls die Operationsgruppen passen, wird das Ergebnis durch die Auswertung der Operation beeinflusst.

Sowohl der Operationsgruppen- wie auch der Operationsvergleicher basieren auf *And*-Verknüpfungen der jeweiligen Bitfelder (rechter Teil Abbildung 6.14). Diese Vorgehensweise ist nur dann korrekt, falls jeder Operationsgruppe stets ein Bit zugeordnet ist. Gleiches gilt für die Operationen einer Gruppe. Damit NoOp innerhalb eines internen Befehlswortes als gültige Operationsgruppe, wie auch Operation, aller Funktionseinheiten erkannt wird, muss demzufolge NoOp durch ein gesetztes Bitfeld repräsentiert werden.

Ob ein Befehlswort nach einer Zerlegung in Einzeloperationen verarbeitet werden kann, wird mittels der Vergleiche der einzelnen Operationsfelder mit den Funktionseinheitentabellen

[11] Eine Einzeloperation $Op2$ und demzufolge eine Operationskombination aus $Op1$ und $Op2$ kann nicht auftreten, da der ARM-Befehlssatz keine Befehle mit mehr als zwei Einzeloperationen liefert.

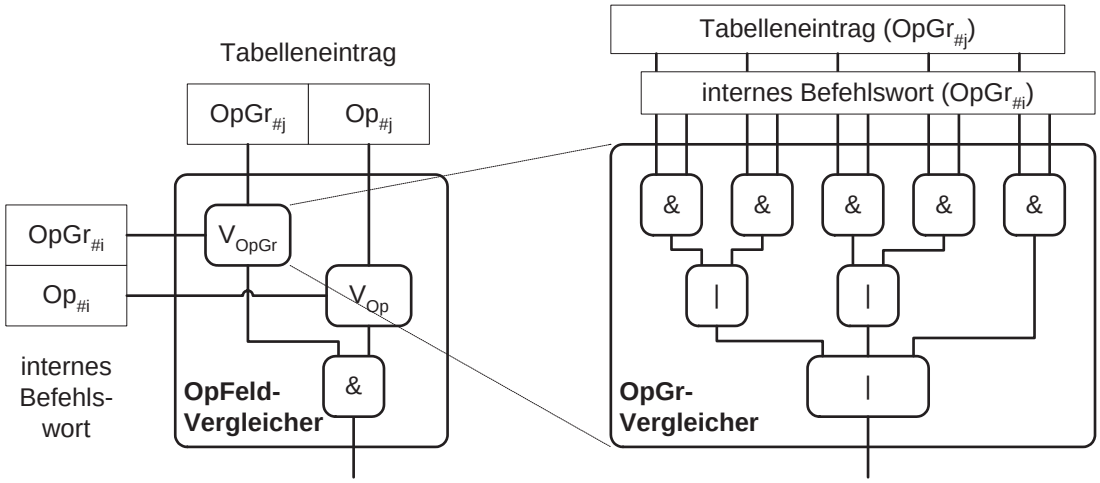

Abbildung 6.14: Aufbau der Operationsfeldvergleicher

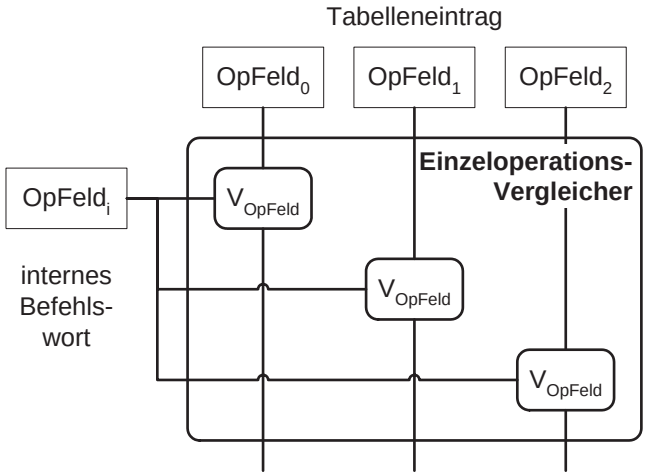

Abbildung 6.15: Aufbau der Einzeloperationsvergleicher

analysiert. Abbildung 6.15 stellt die *Einzeloperationsvergleicher* vor. Man erkennt ein beliebiges Operationsfeld des internen Befehlswortes, dass an einen Eintrag einer Tabelle gebunden ist. Als Ergebnis erhält man einen 3 Bit Wert. Sind alle Bit logisch Null, wird die Operation nicht unterstützt. Ansonsten spiegelt der Ausgangswert die Ausführungsmöglichkeiten wieder und dient als Eingabe für die evtl. später durchzuführende Rotation. Ist das erste Bit gesetzt, muss, falls überhaupt eine Einzeloperationsausführung erforderlich ist, keine Rotation erfolgen. Des Weitern entspricht z.B. die Bitfolge 010 einer Rotation. Bei 011 kann optional ein- oder zweimal geschoben werden usw.

Als letztes bleibt ein weiterer Fall zu überprüfen. Kann eine Funktionseinheit das Befehlswort aus $Op0$ und $Op1$ ausführen, wenn eine Rotation nach rechts ausgeführt wird. Der *Operationskombinationsvergleicher* aus Abbildung 6.16 testet diese Konstellation und liefert zudem implizit die Rotationseinstellung.

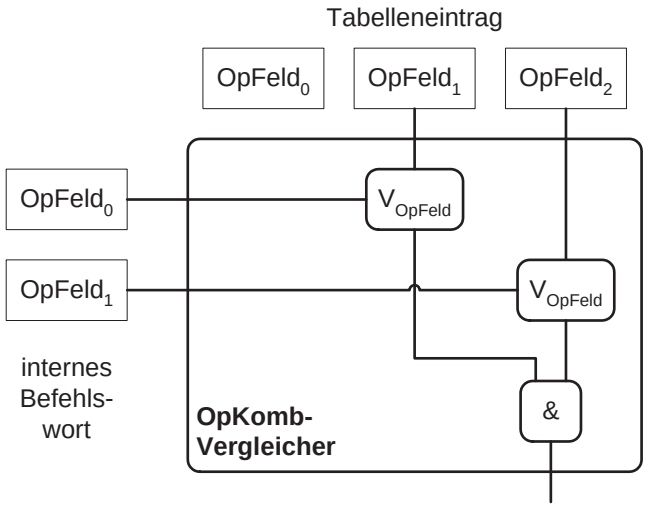

Abbildung 6.16: Aufbau der Operationskombinationsvergleicher

Zusammenfassend kann festgehalten werden, dass die Vergleichslogik aus folgenden Komponenten besteht:

- Dem *Gesamtvergleicher*, der das gesamte Befehlswort auf Ausführbarkeit ohne weitere Veränderungen prüft.

- Zwei *Einzeloperationsvergleichern*, die, falls eine Zerteilung nötig ist, zum einen den Grad der Zerteilung und zum anderen die jeweilige Anzahl der Rotationen liefern.

- Einem *Operationskombinationsvergleicher* dessen Aufgabe darin besteht, zu testen, ob das Befehlswort als Operationskombination betrachtet durch eine Rechtsrotation verarbeitet werden kann. In diesem Zusammenhang wird ein internes Befehlswort mit nur einer Operation $Op0$ als Kombination aus $Op0$ und NoOp behandelt. Dies ist keine Einschränkung der Vorgehensweise, muss aber bei der Auswertung aller Vergleichsergebnisse beachtet werden.

Die Zerteilungs- und Rotationseinheit: Bei der funktionalen Betrachtung der Zerteilung ist es sinnvoll eine logische Trennung in die Zerteilungs- und Rotationsphase zu vollziehen. Für die Implementierung bietet sich jedoch ein Ineinandergreifen beider Phasen an, wie in Abbildung 6.17 gezeigt wird. Unabhängig von der Auswertung der Vergleicherlogik werden immer zwei interne Befehlsworte aus einem eingehenden Befehlswort erzeugt. Die Gültigkeit der Worte wird durch die Vergleicherlogik beeinflusst. Kann das Befehlswort unverändert oder durch eine einzelne Rechtsrotation verarbeitet werden, ist das resultierende zweite Befehlswort ungültig. Der Inhalt bleibt unbeachtet und ist mit X gekennzeichnet. Da der ARM-Befehlssatz keine Befehle, die in drei Einzeloperationen zerlegbar sind, anbietet, wird in beiden Fällen entweder die erste oder letzte Operation des ersten Befehlswortes konstant auf *NoOp* gesetzt. Bei der Zerlegung in Einzeloperationen müssen beide Befehlsworte, in Abhängigkeit von den zugehörigen Vergleicherergebnissen, komplett generiert werden.

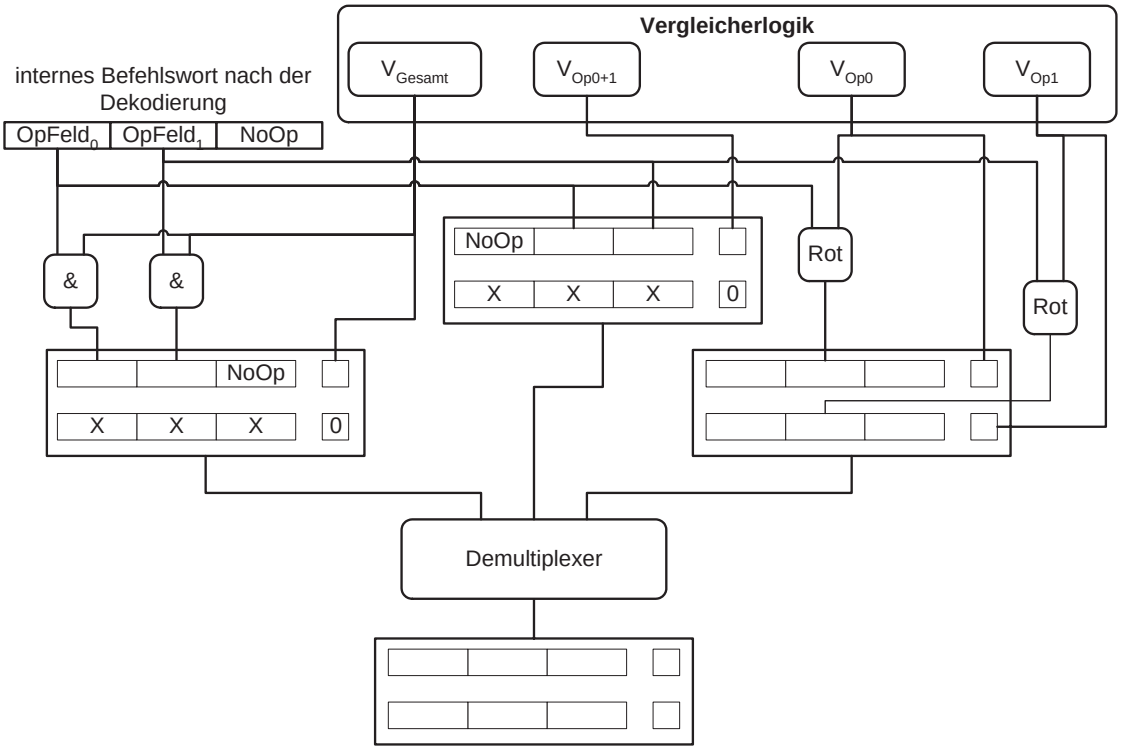

Abbildung 6.17: Schematische Darstellung der Zerteilung

Für die Weiterverarbeitung im Prozessor wird aus den drei Ergebnissen eines ausgewählt, wobei die unveränderte Version die höchste Priorität genießt und die Zerlegung die niedrigste.

Der Kombinationsmechanismus

Hauptbestandteil der *Kombinationslogik* sind zwei unterschiedliche Komponenten, die zum einen auf die Verknüpfung von zwei Befehlsworten und zum anderen auf die Kombination von drei Befehlsworten abzielen. Da nach der Dekodierung eines Maschinenbefehls des ARM-Befehlssatzes eine oder zwei Einzeloperationen vorliegen können, folgt daraus, dass die Kombination von zwei Befehlsworten zu einer Zweierkombination, falls beide Befehle Einzeloperationen verkörpern, oder einer Dreierkombination, falls einer der beiden Befehle bereits eine Zweierkombination darstellt, möglich ist. Die Zusammenfassung von drei Befehlen beruht andererseits auf drei Einzeloperationen. Die Maximalzahl von drei Operationen pro internem Befehlswort darf in keinem Fall überschritten werden.

Der grundsätzliche Aufbau der verschiedenen Kombinationseinheiten ist identisch. Die Komponenten variieren in der Anzahl der nötigen Vergleiche zur Erkennung von Operationskombinationen.

Die Kombinationseinheiten: Die in Abbildung 6.18 skizzierte Kombinationseinheit realisiert die Umwandlung von zwei Maschinenbefehlen in eine Zweier- oder Dreierkombination in ein einzelnes Befehlswort. Der NoOp-Vergleich dient der Erkennung, ob eine Kombinati-

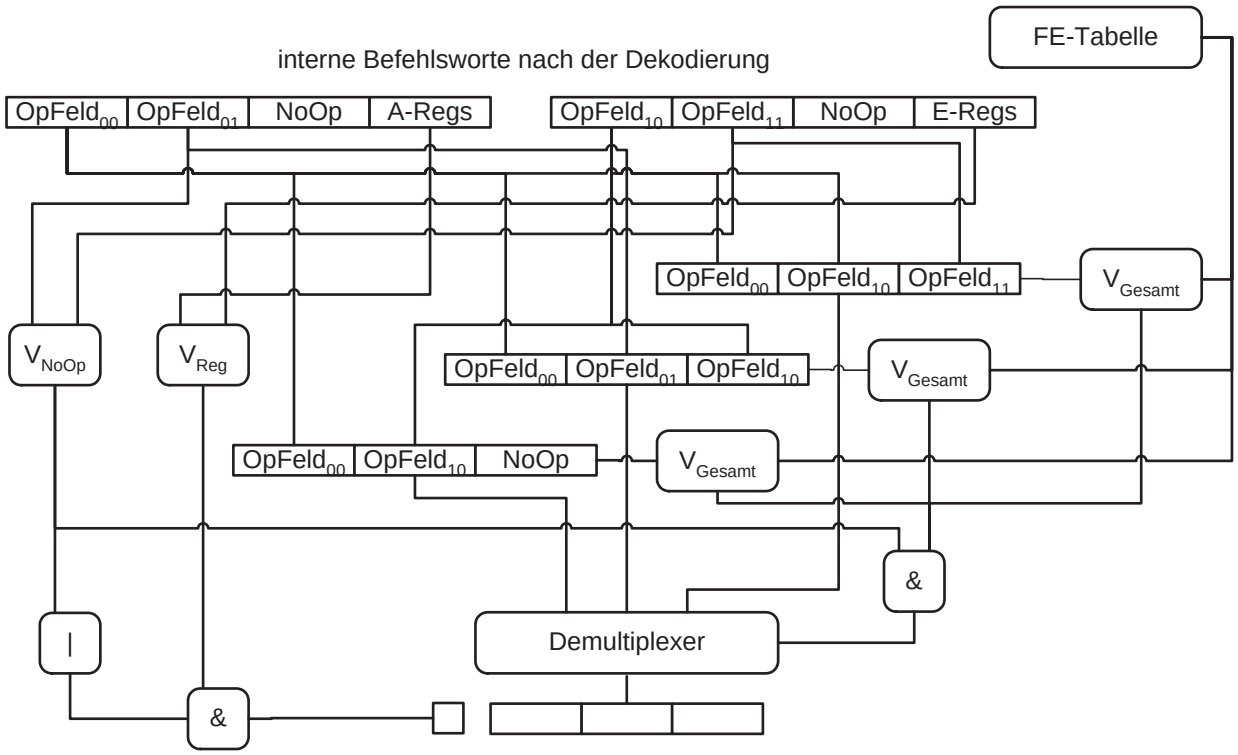

Abbildung 6.18: Aufbau einer Kombinationseinheit

on anhand der Operationsfelder der beiden am Eingang anliegenden Befehlsworte möglich ist. Aus den Operationsfeldern der beiden Befehlsworte werden vorab – unabhängig von der Auswertung der NoOp-Vergleicher – drei Befehlsworte gebildet (siehe Abbildung 6.18). Diese erfahren einen Vergleich mit den Inhalten der Tabellen aktiver und inaktiver Funktionseinheiten. Zur Vereinfachung ist exemplarisch nur eine Funktionseinheitentabelle eingezeichnet.

Der resultierende drei Bit Wert des NoOp-Vergleichers erlaubt eine Und-Verknüpfung mit den Ausgängen der drei *Gesamtvergleicher*. Dieses Ergebnis fungiert zusätzlich als Steuersignal des *Demultiplexers*. Das Befehlswort am Ausgang des Demultiplexers wird abschließend durch das Ergebnis des *Registervergleichers* verifiziert, d.h. falls der NoOp-Vergleich mindestens eine Kombinationsmöglichkeit als zulässig identifiziert, ist eine Registerabhängigkeit die letzte Anforderung, die erfüllt sein muss.

Ob eine Operationskombination zur Ausführung kommt, entscheidet sich an der Auswertung der Tabelle aktiver Funktionseinheiten. Die Signale, die mittels der Tabelle inaktiver Funktionseinheiten erzeugt werden, dienen ausschließlich der Identifikation der Modulnutzungsmöglichkeiten und fließen entsprechend in die Regelung des adaptiven Kerns ein.

6.2.2 Hardware-Implementierung der Wandlung

Tabelle 6.2 zeigt einen Ausschnitt aus den Syntheseergebnissen der Wandlungskomponenten und der gesamten Wandlungsphase. Die Tabelle enthält die Angaben zum Ressourcenbedarf in *ALUTs* (*Adaptive Look-up Tables*)[12] und den erreichten Taktfrequenzen für die Kompo-

[12] Als *ALUTs* werden die Logikelemente (Logik und Speicher) der STARIX-Bausteine bezeichnet.

Komponente	Subkomponente(n)	ALUTs	MHz	Pfad
FE-Tabelle	8 Tabelleneinträge	240	272,33	
	Tabelleneintrag	29	> 500,00	
Zerteiler	Zerteilerlogik, Vergleicherlogik	459	149,08	
	Zerteilerlogik	241	(148,7)	6,726 ns
	Vergleicherlogik	224	(191,9)	5,210 ns
Komb.Einheit	2 2erKomb.Logik, 3erKomb.Logik, 3 Gesamtvergleicher	189	229,15	
	2erKomb.Logik	44	(212,0)	4,716 ns
	3erKomb.Logik	38	(207,0)	4,830 ns
	Gesamtvergleicher	9	(236,0)	4,236 ns
Wandlung	FE-Tabelle, 3 Zerteiler, Komb.Einheit	1.713	132,68	

Tabelle 6.2: Hardware-Implementierung der Wandlung

nenten und Teilkomponenten der Wandlung. Zusätzlich gibt die Spalte *Pfad* den zeitkritischsten Leitungspfad (in Nanosekunden) der rein kombinatorischen Logikschaltungen an. Die diesbezüglich in Klammern angegebenen Frequenzwerte ergeben sich rechnerisch aus dem Zeitverhalten der Leitungspfade. Die Synchronisation zum Systemtakt übernimmt jeweils die übergeordnete Komponente.

Entsprechend den in Abschnitt 6.2.6 aufgeführten Implementierungsvarianten der ARM-Vergleichsarchitektur liegt der Ressourcenbedarf der gesamten Wandlungsphase im Vergleich zur Basisarchitektur in einem Bereich von 7,3% und 10,9%. In diesem Zusammenhang ist die notwendige Logik für zwei zusätzliche Dekodereinheiten und der Mehraufwand bezüglich des breiteren internen Befehlswortes nicht berücksichtigt. Derzeit liegen darüber noch keine Daten vor.

6.2.3 Aufbau des adaptiven Kerns

Der Aufbau des adaptiven Kerns, der unter anderem die rekonfigurierbaren Bereiche beinhaltet, wird im Folgenden erklärt. Im Vordergrund steht die Umsetzung auf der Grundlage der VM-basierten Modulnutzungserkennung. Mit Ausnahme der Integration der rekonfigurierbaren Bereiche, die vom verwendeten internen Befehlswort beeinflusst wird, ist die Variante der Modulnutzungserkennung für den adaptiven Kern irrelevant.

In Abbildung 6.19 erkennt man die Befehlswarteschlange des Instruktionszuteilers und einen Ausschnitt des adaptiven Kerns. Für den adaptiven Kern werden mindestens zwei Warteschlangen empfohlen: Eine Befehlswarteschlange, die ausschließlich der Ablage der ARM-basierten Befehle dient, und eine Warteschlange, die der Verwendung im Zusammenhang mit Hardware-Modulen vorbehalten ist.

Abbildung 6.19 zeigt die Integration zweier unabhängiger rekonfigurierbarer Bereiche sowie deren Anbindung an die zugehörige Befehlswarteschlange und die zusätzlich benötigten

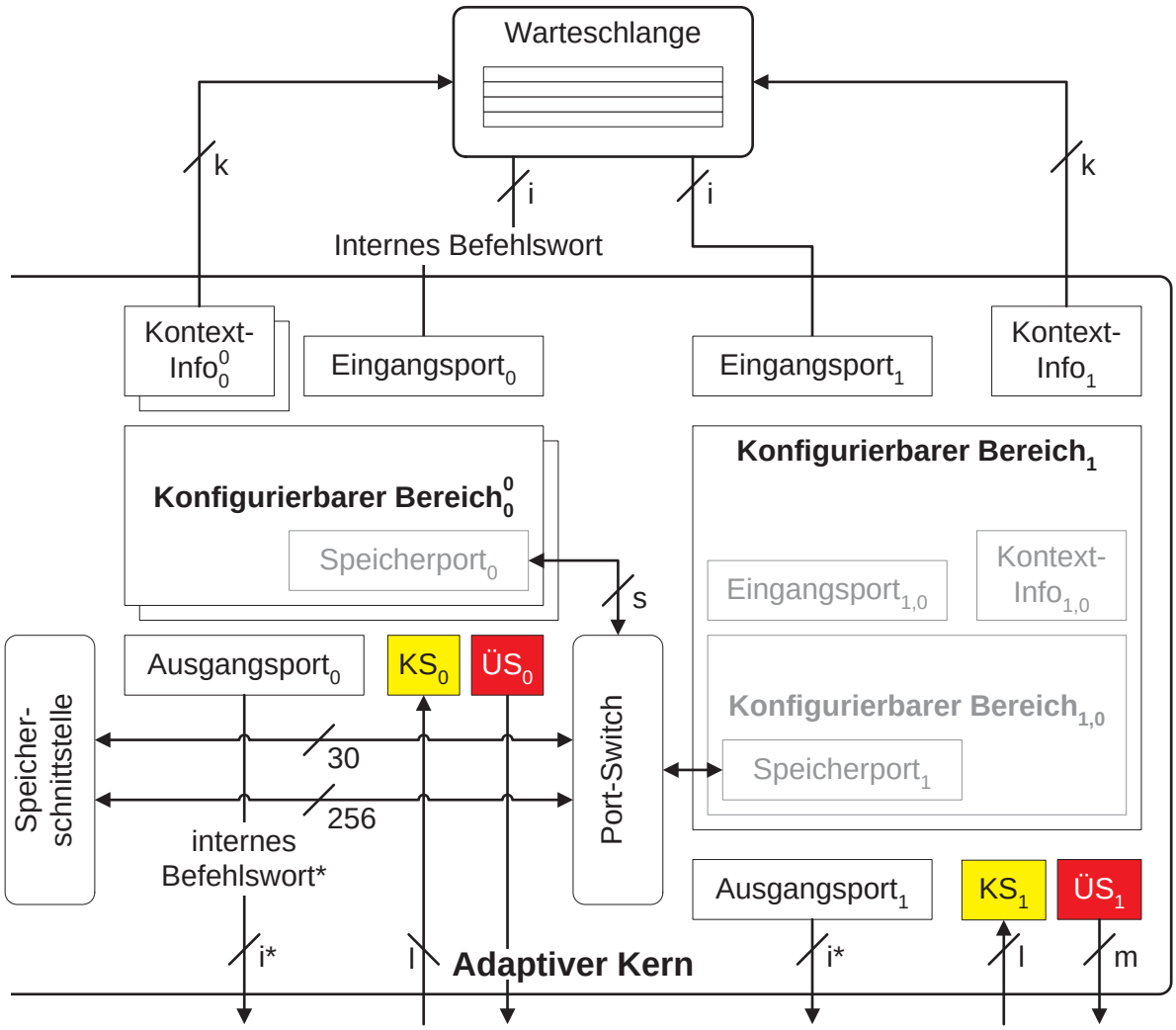

Abbildung 6.19: Integration der rekonfigurierbaren Bereiche in den adaptiven Kern

Schnittstellen. Die Beschriftung der rekonfigurierbaren Bereiche (KB) wird folgendermaßen vorgenommen:

$$KB^{Kontext}_{Index,Unterbereich},\ wobei\ Kontext,\ Index\ und\ Unterbereich\ \epsilon\ \mathbb{N}_0\ sind.$$

Existieren k konfigurierbare Bereiche, werden diese mit jeweils einem Index $\epsilon\ \{0,\ldots,k-1\}$ durchnummeriert. Entsprechendes gilt für die Angaben des Kontextes. Zudem kann ein konfigurierbarer Bereich in ein oder mehrere Unterbereiche aufgeteilt sein.

Die Darstellung in Abbildung 6.19 lässt zwei konfigurierbare Bereiche (KB) unterschiedlicher Größe erkennen:

KB_0 umfasst zwei Kontexte KB^0_0 und KB^1_0. Physikalisch liegt ein einzelner konfigurierbarer Bereich KB_0 vor. Die Nutzung zweier Kontexte ist mit Hilfe geeigneter Speicherressourcen innerhalb des Bereichs realisiert (Vgl. Grundlagenabschnitt 2.2). Da das Umschalten von Kontexten während eines Prozessortaktes abgewickelt werden kann [48, 157], ist aus Sicht des Prozessors keine Unterscheidung notwendig, ob zwei logische Kon-

texte in einem Bereich oder zwei physikalisch getrennte Bereiche mit jeweils einem Kontext vorliegen. Im Weiteren werden Kontexte so betrachtet, als handle es sich um jeweils physikalisch getrennte Ressourcen.

KB_1 hingegen ist in einen zusätzlichen Unterbereich $KB_{1,0}$ zerlegbar, verfügt jedoch nur über einen einzigen Kontext. Somit können drei unabhängige Module gleichzeitig im adaptiven Kern verfügbar sein. Die Anzahl integrierter Bereiche sollte als Richtlinie verstanden werden. Für die bisher durchgeführten Simulationen zeigt sich diese Anzahl rekonfigurierbarer Bereiche als ausreichend. Hierzu muss angemerkt werden, dass bis dato keine automatische Modulgenerierung für den adaptiven Prozessor vorhanden ist und sowohl das Erzeugen von Modulen, wie auch das Einfügen der notwendigen Bibliotheksinstruktionen manuell durchgeführt wurde. Demzufolge kann sich mit entsprechendem Übersetzer durchaus eine andere Anzahl rekonfigurierbarer Bereich ergeben. Zudem muss dies evtl. durch den Einsatz des Prozessors in einem speziellen Anwendungsumfeld angepasst werden. Dies gilt ebenfalls für die Größe der einzelnen rekonfigurierbaren Bereiche. In dieser Arbeit zeigte sich eine Größe von 1024 ALMs (Adaptive Logic Elements[13]) jeweils für die beiden Kontexte von KB_0 als ausreichend[14]. KB_1 hingegen besteht aus 12.288 ALMs (siehe Abschnitt 6.2.3.1).

Sowohl für den rekonfigurierbaren Bereich KB_0 als auch KB_1 steht ein eigenständiger *Eingangs-* und *Ausgangsport* zur Verfügung. Zudem existieren für jeden Bereich eine Konfigurationsschnittstelle (*KS*) und eine Überwachungsschnittstelle (*ÜS*). Jeder Bereich enthält einen *Speicherport*. Der Zugriff auf die *Speicherschnittstelle* wird über einen *Port-Switch* ermöglicht, der beiden konfigurierbaren Bereichen wechselweisen, gemeinsamen oder exklusiven Zugriff erlaubt. Darüber hinaus muss sich jedes eingelagerte Modul beim Instruktionszuteiler respektive der Befehlswarteschlange mit Angaben über die zugehörigen Opcodes anmelden. Die Angaben werden, während des Konfigurationsvorgangs, über den Bibliotheksteil (ohne Abbildung) bezogen, lokal im zugehörigen *Kontext-Info*-Bereich abgelegt und so an den Instruktionszuteiler weitergeleitet. Bei der Integration zusätzlicher rekonfigurierbarer Bereiche ist der Aufwand für die notwendige Infrastruktur stets zu berücksichtigen. Der Leitungsaufwand für sämtliche Ports steigt linear. Dementgegen steigt der Aufwand für die Instruktionszuteilung quadratisch!

Eingangs-, Ausgangs- und Speicherports repräsentieren, ebenso wie die Konfigurations- und Überwachungsschnittstelle, keinerlei Logik. Diese dienen der Festlegung der für die Modulrekonfiguration notwendigen Anschlusspunkte, zur Integration in den adaptiven Prozessor. Die in Abbildung 6.19 vorgeschlagene Positionierung ist nicht zwingend. Jedoch muss eine Festlegung vor der Erzeugung der eingesetzten Hardware-Module vorgenommen werden. Die Positionierung der Ports bzw. Schnittstellen ist für alle eingesetzten Module bereits bei der Erzeugung unbedingt erforderlich. Die Festlegungen müssen sowohl für manuell als auch automatisch erzeugte Module mit Hilfe so genannter *Constraints* für alle Module gleichermaßen gelten. Dementsprechend umfasst der in Abbildung 6.19 angedeutete Unterbereich $KB_{1,0}$, wie auch die Kontexte des Bereichs KB_0, exakt 1024 ALMs in identischer Anordnung. Dies gilt auch für die Positionierung des $Eingangsports_{1,0}$ und die Lage des $Ausgangsports_1$. Die Lage des $Speicherport_1$ innerhalb des Unterbereichs muss exakt der Lage entsprechen, die der $Speicherport_0$ im Bereich KB_0 einnimmt. Nur so kann gewährleistet werden, dass

[13] Die Nomenklatur entspricht den von der Firma Altera verwendeten Bezeichnungen. Die Zahlenangaben beziehen sich auf ALMs der Stratix II Serie.

[14] Physikalisch existieren 1024 ALMs. Diese stehen beiden Kontexten abwechselnd zur Verfügung.

alle Module ohne Einschränkung – mit Ausnahme des maximalen Ressourcen-Bedarfs – in allen konfigurierbaren Bereichen untergebracht werden können.

6.2.3.1 Struktur der konfigurierbaren Bereiche

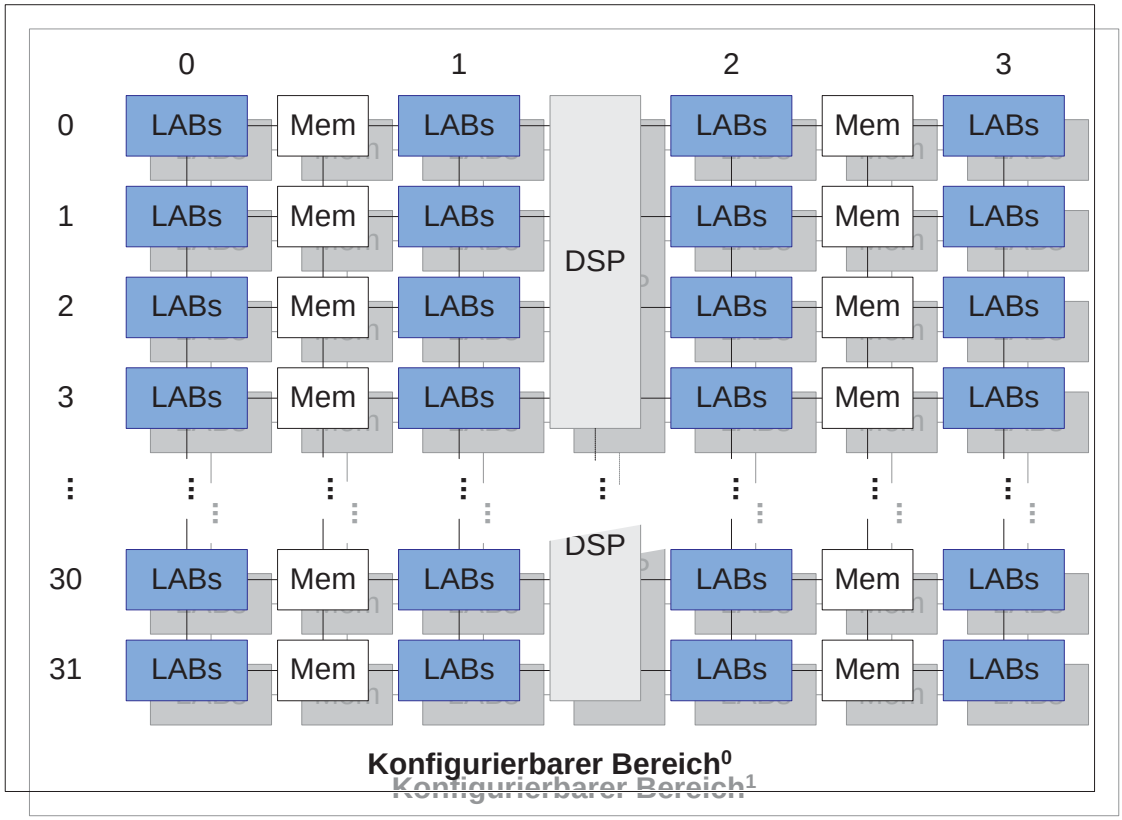

Abbildung 6.20: Struktur eines exemplarischen konfigurierbaren Bereichs mit zwei Kontexten

Abbildung 6.20 veranschaulicht exemplarisch den internen Aufbau des im vorherigen Abschnitt behandelten rekonfigurierbaren Bereichs KB_0. Jedes *LAB* (*Logic Array Block*) beinhaltet acht ALMs. Die LABs sind in vier Spalten und 32 Reihen organisiert. Sowohl den LAB-Spalten 0 und 1 als auch 2 und 3 sind jeweils 32 512 Bit Speicherblöcke zwischengelagert. Jeweils vier LAB-Reihen haben Zugriff auf einen der insgesamt acht verfügbaren *DSP-Blocks* (*Digital Signal Processing-Blocks*). Der Bereich KB_0 stellt demzufolge 1024 ALMs, 32 KBit Speicher und acht DSP-Blöcke für ein einziges Hardware-Modul bereit. Auf Grund der direkten Kopplung der konfigurierbaren Bereiche an den adaptiven Prozessor, verfügt kein Bereich über so genannte IOEs (*Input Output Elements*). Eine detaillierte Beschreibung der einzelnen verwendeten Elemente, sowie der zugehörigen Leitungsinfrastruktur und der Switch-Matrizen kann im Handbuch des Startix II [11] nachgelesen werden (Vgl. auch Grundlagenabschnitt 2.2).

Der in Abschnitt 6.2.3 vorgestellte rekonfigurierbare Bereich KB_1 ist vergleichbar zu Bereich KB_0 aufgebaut. KB_1 verfügt jedoch über 12 LAB-Spalten mit 128 Reihen. Zudem sind zwei der sechs Speicherreihen mit M4K Elementen bestückt, die jeweils 4 KBit Speicher re-

präsentieren. Insgesamt ergeben sich somit 12.288 ALMs, 1280 KBit (256 KBit + 1024 KBit) Speicher und 32 DSP-Blöcke, die maximal als rekonfigurierbare Ressourcen bereitstehen.

Die gewählten Größen der beiden rekonfigurierbaren Bereiche basieren zum Teil auf eigenen Syntheseergebnissen und zum Teil auf Herstellerangaben zu angebotenen Hardware-Lösungen ausgesuchter (Teil-)Algorithmen. Letztere bezeichnet die Firma Altera als so genannte *MegaCores*. Tabelle 6.3 stellt einen Auszug aus verschiedenen MegaCore-Handbüchern dar [8, 9, 13, 10][15].

Name	Variante	ALUTs	Speicher	DSPs	Zyklen	Durchsatz
FFT	1024 Punkte	5.137	152	9	1.024	
	4096 Punkte	7.429	576	18	4.096	
	8192 Punkte	7.351	1.216	16	8.192	
FIR	Seriell	357	26,5			4,26
	Parallel	2.115	180			36,96
	Multi-cycle variable decimation by 4	1.300	52	25		31,82
Viterbi	1 ACS-Einheit	1.381	46,125			2,03
	2 ACS-Einheit	1.532	46,125			3,92
	16 ACS-Einheit	3.471	73,125			25,00
Reed-Solomon	Discrete decoder	1.477	6			372
	Continuous decoder	5.463	14			1.320
	Standard encoder	169	ca. 0,41			>2.000

Tabelle 6.3: Auszug aus den Herstellerangaben zu ausgesuchten MegaCores

Die erste Spalte beinhaltet den Namen des jeweiligen MegaCores. Jeder MegaCore wird in unterschiedlichen Varianten mit diversen Einstellungsmöglichkeiten angeboten. Eine entsprechende Kurzbeschreibung enthält die Spalte *Variante*. Die folgenden drei Spalten zeigen den Ressourcenbedarf der Implementierungen: *ALUTs* repräsentiert diesbezüglich die benötigten Logikelemente. Der Speicherbedarf ist in KBit angegeben. *DSPs* zeigt den Bedarf an DSP-Blöcken. Abschließend zeigen die Spalten *Zyklen* und *Durchsatz* die Leistung der Hardware-Implementierung: Zyklen gibt die Anzahl der Taktzyklen der Abarbeitung an. Zum Beispiel benötigt die erste Variante des FFT-MegaCores 1024 Taktzyklen für die Berechnung von 1024 Punkte. Die Werte der Spalte Durchsatz sind für den FIR-MegaCore in *Giga Multiply Accumulates Per Second* angegeben. Für Viterbi und Reed-Solomon liegen die Durchsatzwerte in $MBits/Sekunde$ vor.

Tabelle 6.4 präsentiert einen Auszug aus eigenen Syntheseläufen für Hardware-Module des adaptiven Prozessors. Eingesetzt wurde die von der Firma Altera vertriebene Quartus II Entwicklungsumgebung. Die Tabellen 6.3 und 6.4 geben eine Vorstellung von den Größenordnungen der notwendigen Hardware-Module. Unter Berücksichtigung dieser Anga-

[15] ACS: Add/Compare/Select

ben und Ergebnisse wurden die Ressourcen der rekonfigurierbaren Bereiche des adaptiven Prozessors festgelegt.

Name	Variante	ALUTs	Speicher	DSPs	Zyklen
FMUL	5-stufig	879	0	0	6
	5-stufig (mit DSPs)	422	0	8	6
MAC	MUL + MUL	1.885	0	0	5
	MUL + MUL (mit DSPs)	67	0	16	5

Tabelle 6.4: Auszug aus eigenen Syntheseläufen mittels QUARTUS II Entwicklungsumgebung

6.2.3.2 Integration der konfigurierbaren Bereiche

Die Betrachtung des internen Befehlsformats des adaptiven Prozessors ist wegen der Art der Integration der rekonfigurierbaren Bereiche notwendig. Jeder integrierte rekonfigurierbare Bereich ist als eigenständige Funktionseinheit mit adaptiver Funktionalität an die fest verdrahteten Elemente des adaptiven Prozessors angeschlossen. Entsprechend muss die benötigte Leitungsinfrastruktur, die dem Transport des internen Befehlswortes dient, unveränderlich als Schnittstelle der rekonfigurierbaren Bereiche vorhanden sein. Um möglichst wenig Ressourcen durch die Leitungsinfrastruktur zu belegen, ist es sinnvoll das interne Befehlsformat den Anforderungen der einzelnen Phasen des Maschinenzyklus anzugleichen. Diese Vorgehensweise ist durch die Kopplung von fest verdrahteter und konfigurierbarer Logik unbedingt notwendig. Im Allgemeinen wird beim Schaltungsentwurf, z.B. mit Hilfe von VHDL, eine Optimierung des Ressourcenverbrauchs durch die Übersetzungs- bzw. Synthesewerkzeuge durchgeführt. Dadurch, dass die Funktionalität der rekonfigurierbaren Bereiche und demzufolge die benötigte Leitungsinfrastruktur während der Implementierung des adaptiven Prozessors nicht bekannt ist, muss eine manuelle Optimierung vorgenommen werden. Hierbei gilt es zu beachten, die Funktionalität der möglichen Hardware-Module nicht einzuschränken. Das heißt, es müssen stets alle relevanten Daten des internen Befehlsformates – unabhängig von der Verwendung in einzelnen Hardware-Modulen – an den Eingängen der rekonfigurierbaren Bereiche verfügbar sein. Entsprechend sind die Ausgänge der rekonfigurierbaren Bereiche so zu gestalten, dass eine Weiterverarbeitung ungeachtet der Modulfunktionalität gewährleistet ist.

Diesbezüglich illustriert Abbildung 6.21 die relevanten Daten des internen Befehlsformates in Abhängigkeit der Phasen des Maschinenbefehlszyklus. Der Vollständigkeit halber ist das 32 Bit Befehlswort der Befehlsholphase, das der Ableitung des internen Befehlswortes dient, eingezeichnet.

Das in diesem Abschnitt behandelte interne Befehlsformat gilt ausschließlich für den Einsatz der VM-gestützten Modulrekonfiguration. Erweiterungen und Anpassungen, die auf die zusätzliche HW-gestützte Variante zurückzuführen sind, wurden in Abschnitt 6.1.2 erläutert.

Abbildung 6.21 zeigt jeweils die Daten des internen Befehlswortes, die für die Übergabe an die darauf folgende Pipeline-Stufe von Bedeutung sind. Zusätzliche Steuersignale und temporäre Daten, die in einzelnen Phasen zur Verarbeitung benötigt werden, fehlen. Hierunter

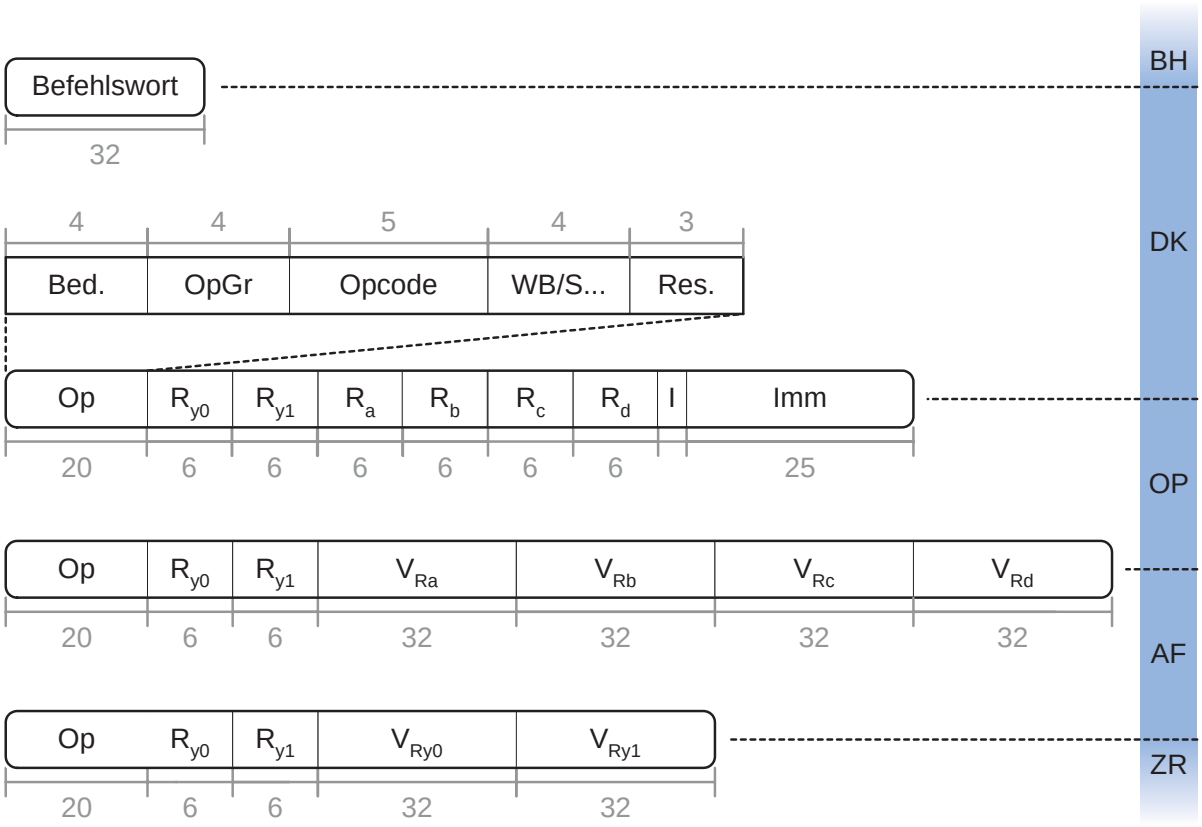

Abbildung 6.21: Phasenspezifische Anpassung des internen Befehlswortes

fallen auch sämtliche Informationen die z.B. für das Holen oder Reservieren von Operanden zuständig sind sowie die Daten für die Wiederherstellung der Ausführungsreihenfolge usw. Alle Daten, die hierbei für die Rückschreibphase des Prozessors von Bedeutung sind, werden an den rekonfigurierbaren Bereichen vorbei geleitet. Für die Weiterverarbeitung müssen diese an den Eingängen der nachfolgenden Pipeline-Stufe entsprechend verfügbar gemacht werden. Diese Vorgehensweise belegt keine rekonfigurierbaren Ressourcen durch die Ablage von Information, die für die Funktionalität der Hardware-Module keine Relevanz besitzen.

Der 32 Bit Maschinenbefehl wird innerhalb der Dekodierphase in ein 82 Bit breites internes Befehlsformat übersetzt (Abbildung 6.21):

- Das **Opcode**-Feld (*OP*) umfasst 20 Bit und ist selbst in 5 Teilfelder aufgegliedert:
  - Die **Bedingung** (*Bed.*) ist ein vier Bit Wert, der die Ausführungsbedingung des ursprünglichen Maschinenbefehls beeinhaltet.
  - **Opcode-Gruppe** (*OpGr*) und **Opcode** halten die Information über die auszuführende Operation. Die Opcode-Gruppe unterteilt die einzelnen Operationen in Gruppen, die eine Zuweisung zu den korrespondierenden physikalischen Funktionseinheiten erlaubt. Dies ermöglicht eine einfache Instruktionszuteilung, da diese anhand der Opcode-Gruppe durchführbar ist und nicht den eigentlichen gesamten Opcode (aus Opcode-Gruppe und Opcode) benötigt.

  - Zusätzlich sind vier Bit für die Übergabe spezieller Eigenschaften vorgesehen (*WB/S*). Hierbei ist eine Doppelbelegung der einzelnen Bits oder Bitkombinationen möglich. Innerhalb einer Opcode-Gruppe muss die Belegung jedoch eindeutig sein. Z.B. existiert bei arithmetischen Operationen die Möglichkeit, das Zurückschreiben von Ergebnissen zu unterdrücken, um Vergleichsbefehle auszuführen, die die Registerinhalte unverändert lassen. Im Gegensatz dazu ist diese Option für *Load*-Instruktionen belanglos. Hierbei sind Angaben zur Datenübertragung, z.B. ob ein gesamtes 32 Bit Maschinenwort oder nur Teilworte geladen werden, von Interesse.
  - Der mit *Res.* gekennzeichnete Drei-Bit-Wert ist bisher ungenutzt.

- R_{y0} ist, wie alle mit R beschrifteten Werte, ein Sechs-Bit-Wert, der den Index eines physikalischen Registers trägt. Der Index wird mit einem fünf Bit Wert beschrieben. Demzufolge können 32 unabhängige Register adressiert werden. Das oberste Bit dient dazu, die Gültigkeit des Index anzuzeigen. Ist ein Eintrag als ungültig markiert, wird nicht versucht einen dem Index zugehörigen Registerinhalt zu lesen oder zu schreiben. R_{y0} und R_{y1} repräsentieren zwei zulässige Ausgangsregister. Dies ist mit dem zugrunde liegenden ARM-Befehlssatz konform und erlaubt entweder die Ausgabe zweier unabhängiger 32 Bit Werte oder eines zerlegten 64 Bit Werts. Entsprechend stehen R_a bis R_d stellvertretend für bis zu vier Eingangsregisterindices.

- Zusätzlich ist ein 25 Bit Konstantenfeld vorgesehen (*Imm*). Die Verwendung eines konstanten Eingangswertes wird durch den 1 Bit Wert I signalisiert. Die Nutzung dieses Feldes ist befehlsabhängig und repräsentiert bei Sprungbefehlen einen möglichen Offset oder bei arithmetischen Operationen einen konstanten Wert, der zusätzlich die Anwendung einer Schiebeoperation zulässt und anschließend auf einen 32 Bit Eingangswert erweitert wird (Vgl. Grundlagenabschnitt 2.3.3.1).

Nach Abschluss der Dekodierung wird das erzeugte 82 Bit breite interne Befehlswort an die Reservierungseinheit der Operandenholphase weitergegeben. Die Reservierungseinheit bezieht bei Verfügbarkeit die Registerinhalte in Abhängigkeit der übergebenen Eingangsregisterindices und legt das resultierende 160 Bit breite interne Befehlswort an die Eingänge der folgende Pipeline-Stufe[16]. Für die Weiterverarbeitung sind die Registerindices der Eingangsregister bedeutungslos und werden entsprechend weggelassen. Ausschließlich die Werte der korrespondierenden Register werden weitergeleitet. Wird das Konstantenfeld als Eingangswert genutzt, liegt dieser Wert in einem der mit V beschrifteten Feldern. Welcher Registerwert für die Ersetzung durch den Konstantenwert zulässig ist, ergibt sich aus der zugrunde liegenden Operation. Damit ein Konstantenwert bereits am Ende der Operandenholphase verfügbar ist, muss mindestens eine zusätzliche Schiebeeinheit, die der Nutzung durch die Reservierungseinheit vorbehalten ist, eingesetzt werden.

Das nunmehr 160 Bit lange interne Befehlswort wird mit Hilfe des Instruktionszuteilers an die entsprechenden Funktionseinheiten bzw. rekonfigurierbaren Bereiche verteilt. Mit Abschluss der Ausführungsphase reduziert sich das interne Befehlswort auf 96 Bit (Abbildung 6.21). Davon entfallen 64 Bit auf die Übergabe des oder der Ergebnisse.

[16] Multiple Load/Store-Instruktionen müssen gesondert behandelt werden. Diese sind als einzige in der Lage bis zu 16 Registerwerte gleichzeitig zu lesen bzw. zu schreiben.

Da sich der adaptive Kern und somit auch die rekonfigurierbaren Bereiche eindeutig der Ausführungsphase zuordnen lassen, sind sowohl das interne Befehlswort mit 160 Bit als auch 96 Bit Breite für die Integration der rekonfigurierbaren Bereiche relevant.

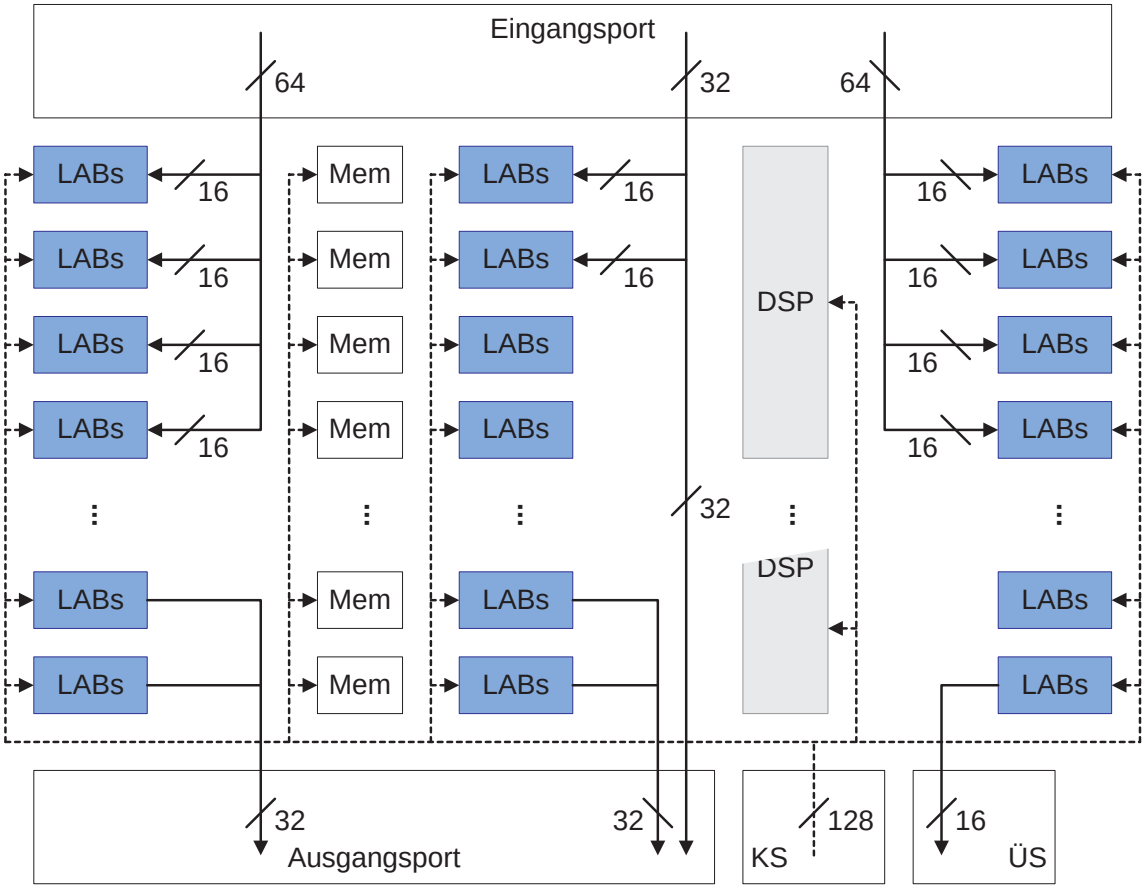

Abbildung 6.22: Vorgaben für die Integration konfigurierbarer Bereiche

Abbildung 6.22 stellt die für die Integration eines einzelnen rekonfigurierbaren Bereichs notwendigen Datenpfade dar. Die gezeigte Anbindung der Eingangs- und Ausgangsleitungen muss mittels *Constraints* bei der Erzeugung von Hardware-Modulen berücksichtigt werden. Als Grundlage für die Datenverteilung dienen die Fähigkeiten der Startix II Logikelemente [11]. Die Übergabe des internen Befehlsformats an die rekonfigurierbaren Bereiche erfordert 160 Eingangsleitungen. Es werden 96 Leitungen für den Transport des internen Befehlsformats, am Ende der Ausführungsphase, an das Prozessor-Back-End benötigt. Zusätzlich erfordern die Eingangs- und Ausgangsleitungen mindestens ein Steuersignal zur Kommunikation mit den vor- und nachgelagerten Prozessorkomponenten.

Um eine Rekonfiguration in angemessenen Zeiträumen zu bewältigen muss die Konfigurationsschnittstelle 128 Bit Konfigurationsdaten pro Taktzyklus übertragen können. Die Konfiguration erfolgt spaltenweise. Entsprechend muss die Konfigurationsinformation in Abhängigkeit vom Konfigurationsfortschritt an die passende Spalte umgeleitet werden. Als Anbindung an die Überwachungseinheiten des Prozessors dient eine 16 Bit Schnittstelle.

Sowohl das Systemtaktnetz als auch die Versorgung der rekonfigurierbaren Elemente mit einem asynchronem Reset fehlen aus Gründen der Übersichtlichkeit in Abbildung 6.22. Dies gilt ebenfalls für die innerhalb eines Bereichs bereitgestellte Leitungsinfrastruktur.

6.2.4 Aufbau der Anpassungskomponenten

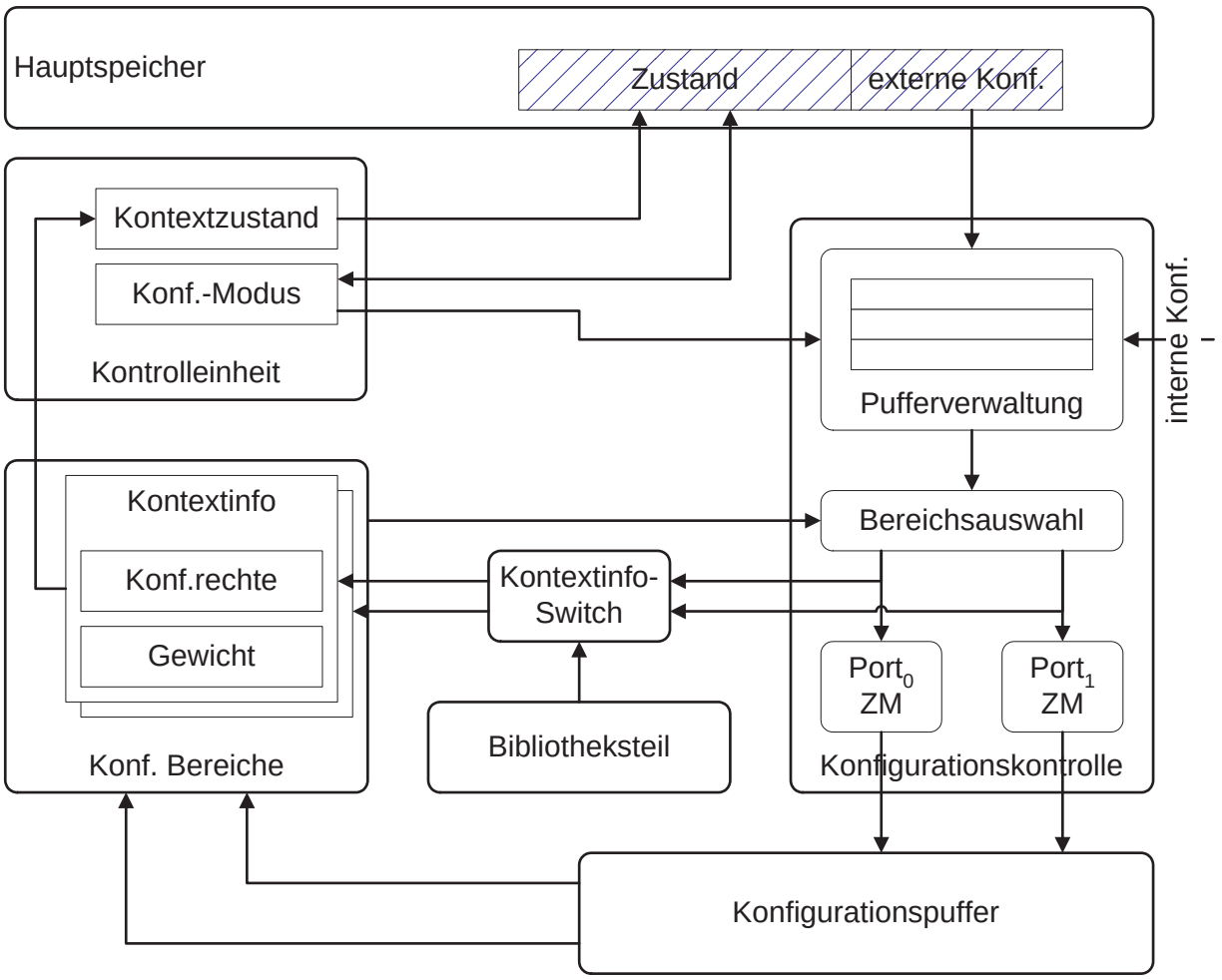

Abbildung 6.23: Aufbau der Anpassungskomponenten im adaptiven Prozessor

In Abbildung 6.23 erkennt man die Anpassungskomponenten sowie die essentiellen Kommunikationskanäle. Zudem bildet der Hauptspeicher über die beiden Speicherbereiche *Zustand* und *externe Konfiguration* die Kommunikationsschnittstelle zwischen Betriebssystem und dem adaptiven Prozessor. Diesbezüglich legt die Kontrolleinheit die aktuellen Zustandsdaten der prozessorinternen rekonfigurierbaren Bereiche in den vorgesehenen Speicherbereich. Darüber hinaus kann das Betriebssystem den Konfigurationsmodus des Prozessors einstellen und auslesen. In Abhängigkeit vom Konfigurationsmodus ist die externe Konfiguration erlaubt oder wird unterbunden.

Konfigurationsanfragen, die auf Grund des Konfigurationsmodus zulässig sind, werden im Puffer der Pufferverwaltung abgelegt. Die Konfigurationsanfragen beinhalten folgende Felder:

- **Bereich**: Gibt den zu verwendenden rekonfigurierbaren Bereich an.
- **Kontext**: Legt den Kontext des Bereichs fest.
- **Unterbereich**: Legt entsprechend den Unterbereich fest.
- **Index**: Repräsentiert den eindeutigen Index, der genau ein Hardware-Modul identifiziert.
- **Kontextrechte**: Beschreibt die Dringlichkeit der Einlagerung und ob das zugehörige Modul durch interne Regelungsmaßnahmen entfernt werden darf.

Eingelagerte Konfigurationsanfragen übernimmt die Bereichsauswahl und versucht diese an eine der beiden verfügbaren Port-Zustandsmaschinen (*Port-ZM*) weiterzuleiten. Die Zuteilung erfolgt anhand der Information der Konfigurationsanfrage und auf Basis der aktuell vorliegenden Kontextinformation. Sind die Felder Bereich, Kontext oder Unterbereich nicht gesetzt, kann die Bereichsauswahl ohne Einschränkung einen rekonfigurierbaren Bereich für das angeforderte Modul auswählen. Hierbei ist ausschließlich die Größe des Moduls zu beachten. Jede der beiden Port-Zustandsmaschinen ist einem rekonfigurierbaren Bereich des Prozessor zugewiesen.

Zusätzlich leitet die *Bereichsauswahl* die Konfigurationsanfrage an den *Kontextinfo-Switch*, der die zugehörigen Bibliotheksdaten in die passenden *Kontextinfo* der rekonfigurierbaren Bereiche umleitet. Eine Änderung der Kontextinfo wird an die Kontrolleinheit weitergeleitet, damit diese die aktuelle Zustandsinformation zur Veröffentlichung vorbereitet.

Innerhalb der Kontextinfo sind zwei Register hervorgehoben: Die Kontextrechte signalisieren der Bereichsauswahl, ob der zugehörige Kontext überhaupt zur Rekonfiguration freigegeben ist. Zudem liefert *Gewicht* eine Bewertung, die für die Ersetzung genutzt wird. Neben den Kontextrechten zeigt das Gewicht, den zu erwartenden Gewinn des Moduls an. Somit wird das Modul mit kleinstem Gewicht bevorzugt ersetzt[17]. Während der Laufzeit einer Anwendung erfährt der Eintrag Gewicht eine Anpassung durch neu gewonnene Überwachungsdaten. Zu Beginn der Einlagerung ist der Wert Null. Demzufolge muss die Einlagerung eines Moduls so gekennzeichnet sein, dass bei einer darauf folgenden Konfigurationsanforderung nicht das jüngst eingelagerte Modul wieder entfernt wird.

6.2.4.1 Aufbau des Bibliotheksteils

Der *Bibliotheksteil* beinhaltet die lokale Kopie der relevanten Modulinformation der Systembibliothek. Der Aufbau des Bibliotheksteils stellt sich tabellarisch dar und umfasst bis zu neun Einträge (siehe Abbildung 6.24).

Zu den Daten der Systembibliothek enthält der Bibliotheksteil zusätzliche Steuerinformation:

- **Zähler aktivieren**: Zeigt an, ob gemessene Häufigkeiten des zugehörigen Eintrags mittels Überwachungsregistern dem Betriebssystem zur Verfügung gestellt werden. Falls dieses Signal aktiviert ist, werden die gemessenen Häufigkeiten in gesonderten Registern akkumuliert und können vom Betriebssystem ausgelesen werden.

[17] Die Gewinnberechnung erfolgt nach dem in Kapitel 5 angegebenen Schema.

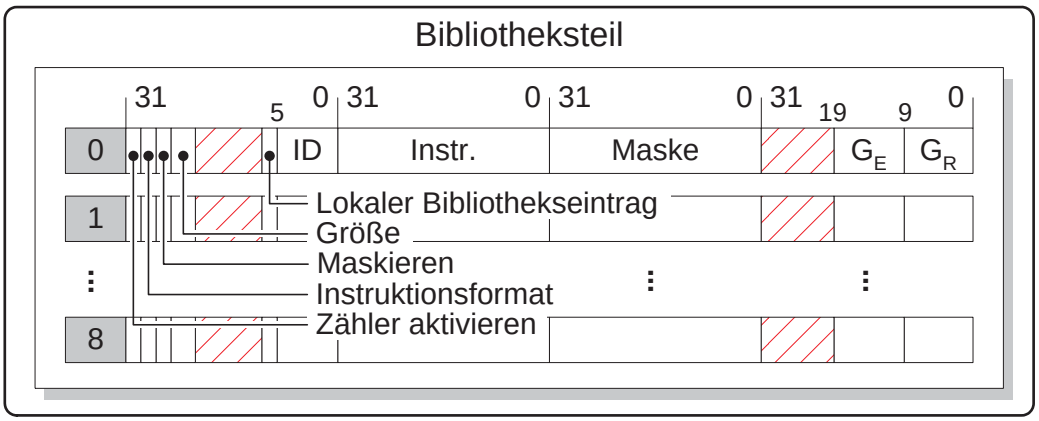

Abbildung 6.24: Schematischer Aufbau des Bibliotheksteils

- **Instruktionsformat**: Ein 2 Bit Wert, der die Interpretation des Instruktions- und Masken-Eintrags regelt. Der Wert 00 bezeichnet den normalen Modus. Hierbei enthält Instruktion *Instr.* den Opcode, der dem Modul zugeordnet wurde. *Maske* kann in Abhängigkeit von *Maskieren* genutzt werden, um die Eindeutigkeit des Opcodes im Feld Instruktion zu gewährleisten[18]. 01 erlaubt die Bildung eines erweiterten Opcodes aus den beiden Feldern Instruktion und Maske. Dies ist im Zusammenhang mit Operationskombinationen notwendig. Die Werte 10 und 11 sind bisher ungenutzt.

- **ID**: Eindeutiger 6 Bit Index, der im Software- und Hardware-Teil der Systembibliothek genau einen Eintrag referenziert und zudem im Funktionseinheitenpool respektive Komponentenpool genau einem Hardware-Modul zugeordnet ist.

- **Lokaler Bibliothekseintrag**: Markiert einen Eintrag als lokalen oder globalen Bibliothekseintrag.

- **Instruktion**: In Abhängigkeit vom Signal Instruktionsformat enthält des Feld Instruktion entweder einen eindeutigen Opcode, der einem Bibliothekseintrag zuzuordnen ist oder repräsentiert den ersten Teil eines erweiterten internen Befehlswortes zur Unterstützung der Operationskombinationen.

- **Maske**: Dient entweder als Maske oder Erweiterung des Opcodes.

- **G**: Beide stellen die zusätzlichen Attribute der Module dar, die für die prozessorinterne Regelung benötigt werden. G_E steht für den zu erwartenden Gewinn in Bezug auf den Energiebedarf. G_R repräsentiert entsprechend den zu erwartenden Gewinn bezüglich der Rechenzeit. Beide Werte sind 10 Bit vorzeichenbehaftete Zweierkomplementzahlen. Beide Attribute werden nach der in Abschnitt 5.3.4.3 vorgestellten Gewinnberechnung ermittelt. Die Berechnung nimmt das Betriebssystem anhand der aktuellen Systemzielsetzung während der Prozesswechsel vor.

[18] Tritt aktuell nicht auf, da Koprozessor-Instruktionen als Bibliotheksinstruktionen genutzt werden und kein zusätzlicher Koprozessor eingesetzt wird. Ist ein Koprozessor vorhanden, müssen etwaige Probleme bezüglich der Eindeutigkeit der Instruktionen mit Hilfe der Maskierung gelöst werden.

- **Größe**: Dieses 2 Bit Feld gibt die Größe des Hardware-Moduls an. Die Angabe erfolgt in den Größenordnungen der rekonfigurierbaren Bereiche. Der kleinste Bereich und somit Module die sich für diesen eignen, wird mit dem Größenwert 00 angegeben. Der nächstgrößere mit 01 usw. Für die beiden Bereiche des adaptiven Prozessors ist ein Ein-Bit-Wert ausreichend. Das zweite Bit bleibt vorerst ungenutzt.

6.2.5 Integration der Überwachung und Regelung

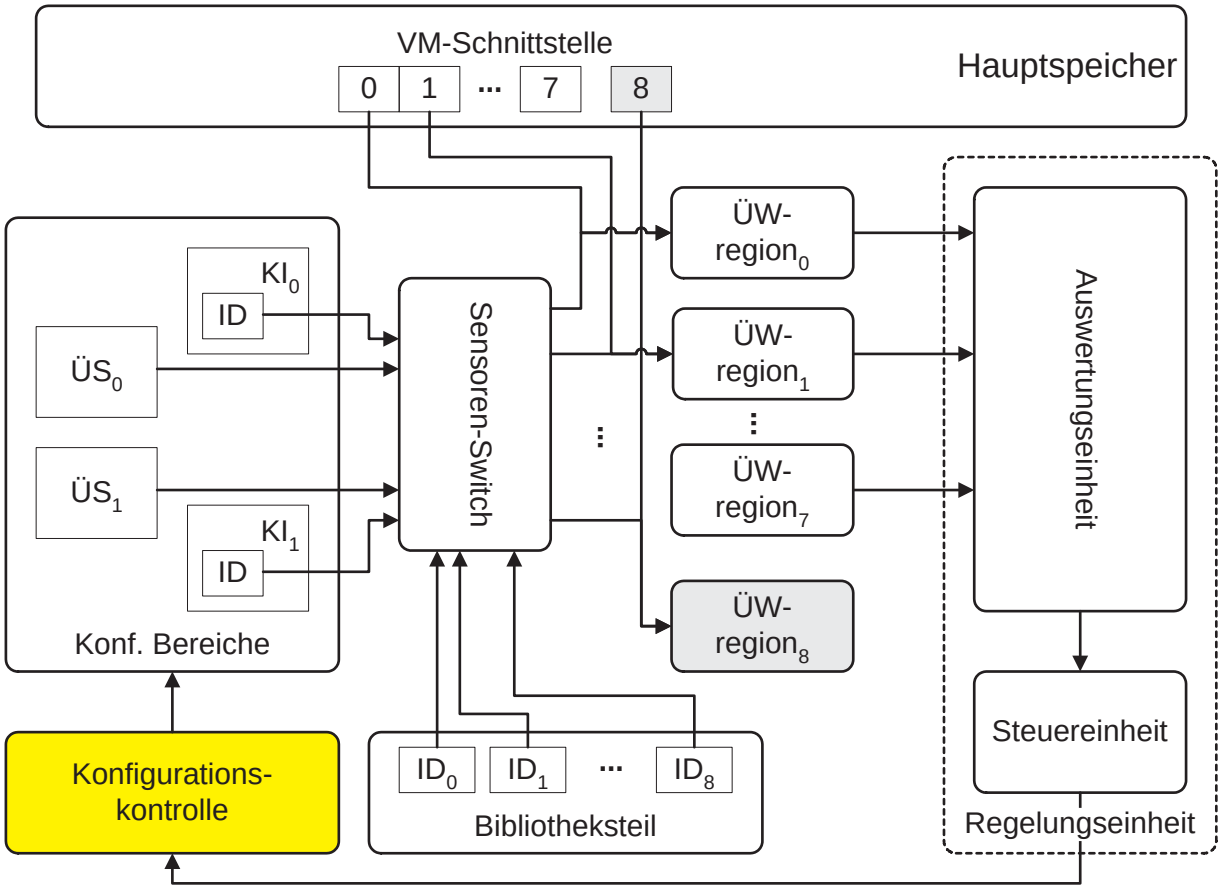

Abbildung 6.25: Integration der Überwachung und Regelung

Die Darstellung in Abbildung 6.25 gibt eine schematische Übersicht über die Kopplung der logischen Überwachungs- und Regelungskomponenten an den adaptiven Kern des Prozessors. Zu jedem Zeitpunkt sind 9 Hardware-Module zulässig. Im adaptiven Kern können nur drei davon gleichzeitig eingelagert sein. Jedem möglichen Hardware-Modul ist eine Überwachungsregion (*ÜW-Region*), bestehend aus Überwachungszelle, Nutzungshistorie und Musterspeicher, zugeordnet. Der achte Eintrag ist für eine permanente Modul-Einlagerung, die durch das Betriebssystem vorgenommen werden kann, reserviert. Die zugehörige Überwachungsregion liefert Daten mit Hilfe von Registern an das Betriebssystem. Solche Module unterliegen *nicht* der Regelung.

Die Zählung von Nutzungsereignissen der Module erfolgt zum einen über die VM-Schnittstelle und zum anderen über die Überwachungsschnittstellen (*ÜS*) der konfigurierba-

ren Bereiche im Kern. Hierbei liefert die VM-Schnittstelle Daten über eine mögliche Nutzung und die Information aus dem adaptiven Kern zeigt die tatsächliche Nutzung eines Moduls an. Die Überwachungsregionen sind entsprechend der Reihenfolge der Module im *Bibliotheksteil* angeordnet. Demzufolge muss die Nutzungsinformation aus dem Kern das Wechseln von Modulen zur Laufzeit berücksichtigen. Um keine zusätzlichen Daten zur Identifizierung einzelner Module innerhalb der Überwachungskomponenten verwalten zu müssen, wurde den Überwachungsregionen ein *Sensoren-Switch* vorgelagert. Mit Hilfe des Bibliotheksteils wird die Verbindung der zugehörigen Sensorleitungen so vorgenommen, dass die Daten zur entsprechenden Überwachungsregion gelangen. Die Auswertung der gesammelten Daten übernimmt die *Auswertungseinheit*. Diese beliefert die *Steuereinheit* mit verwertbaren Konfigurationsinformationen. Konfigurationsanforderungen schickt die Steuereinheit an die Konfigurationskontrolle.

6.2.5.1 Aufbau der Auswertungseinheit

Wird in den Überwachungsregionen ein zyklisches Verhalten mit *sicheren Nullstellen* (siehe Abschnitt 5.3.6) signalisiert, beginnt die Auswertungseinheit mit der Verarbeitung der aktuell vorliegenden Muster.

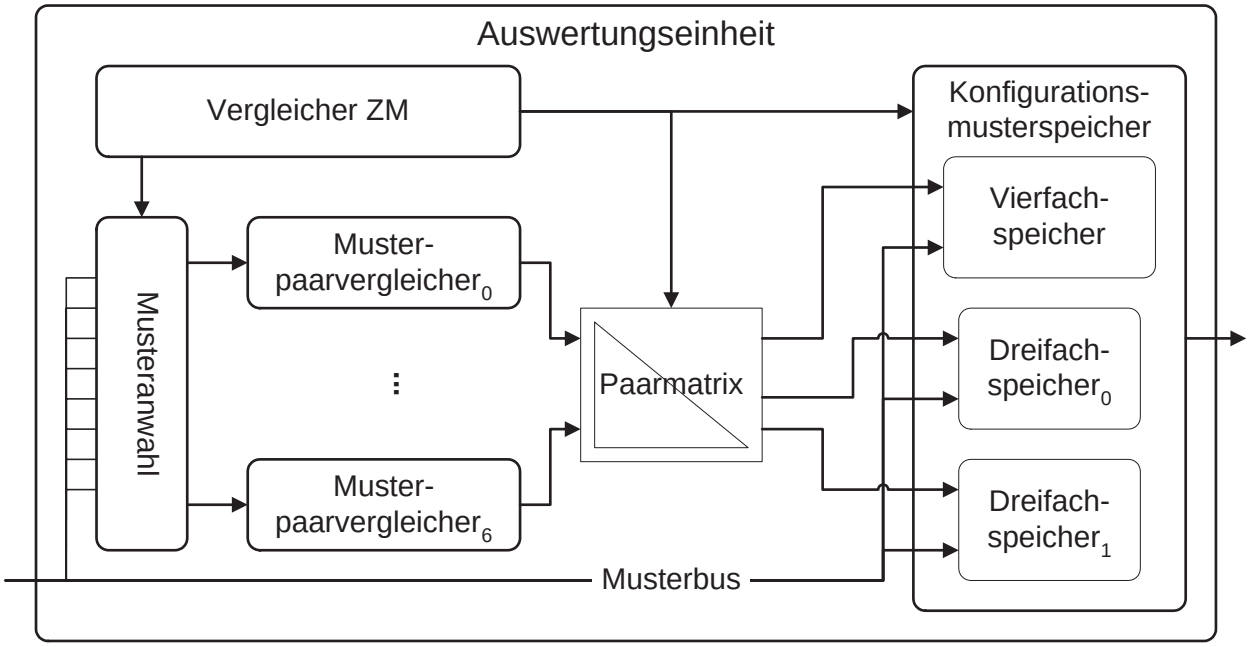

Abbildung 6.26: Aufbau der Auswertungseinheit

In Abbildung 6.26 ist der interne Aufbau der Auswertungseinheit ersichtlich. Die einzelnen Muster der Überwachungsregionen werden über einen Bus an die *Musteranwahl* und den *Konfigurationsmusterspeicher* geleitet. Die Muster werden paarweise verglichen und die Ergebnisse in der *Paarmatrix* abgelegt. Die Paarmatrix ist eine 7×7 Matrix. Hierbei repräsentiert die Spalte 0, alle sieben paarweisen Vergleiche des Musters der $\text{Überwachungsregion}_0$ mit den verbleibenden sieben Mustern usw. Auf Grund der Symmetrie der Vergleichsergebnisse wird nur die untere Dreiecksmatrix erzeugt. Diesbezüglich steuert

die *Vergleicher Zustandsmaschine* sowohl die Auswahl der Muster als auch die Platzierung der Ergebnisse in der Paarmatrix.

Mit Abschluss aller Vergleiche wird der Konfigurationsmusterspeicher angewiesen, entsprechend den Vergleichsergebnissen, mehrfache Kombinationen von Mustern zu bilden. Hierbei berücksichtigt die aktuelle Implementierung ein konfliktfreies Musterverhalten. Das bedeutet, dass ausschließlich Muster kombiniert werden, die jeweils dann eine Modulnutzung anzeigen, wenn alle übrigen betrachteten eine sichere Nullstelle aufweisen.

Innerhalb des Konfigurationsmusterspeichers muss, anhand der Moduldaten des Bibliotheksteils, die eindeutige Zuordnung der Muster erfolgen, da in den *Vierfach-* und *Dreifachspeichern* keine implizite Identifizierung eines Muster mehr möglich ist. Entsprechend der Bezeichnung Vierfach- und Dreifachspeicher können jeweils maximal vier bzw. drei unabhängige, konfliktfreie Muster kombiniert werden.

6.2.5.2 Aufbau der Steuereinheit

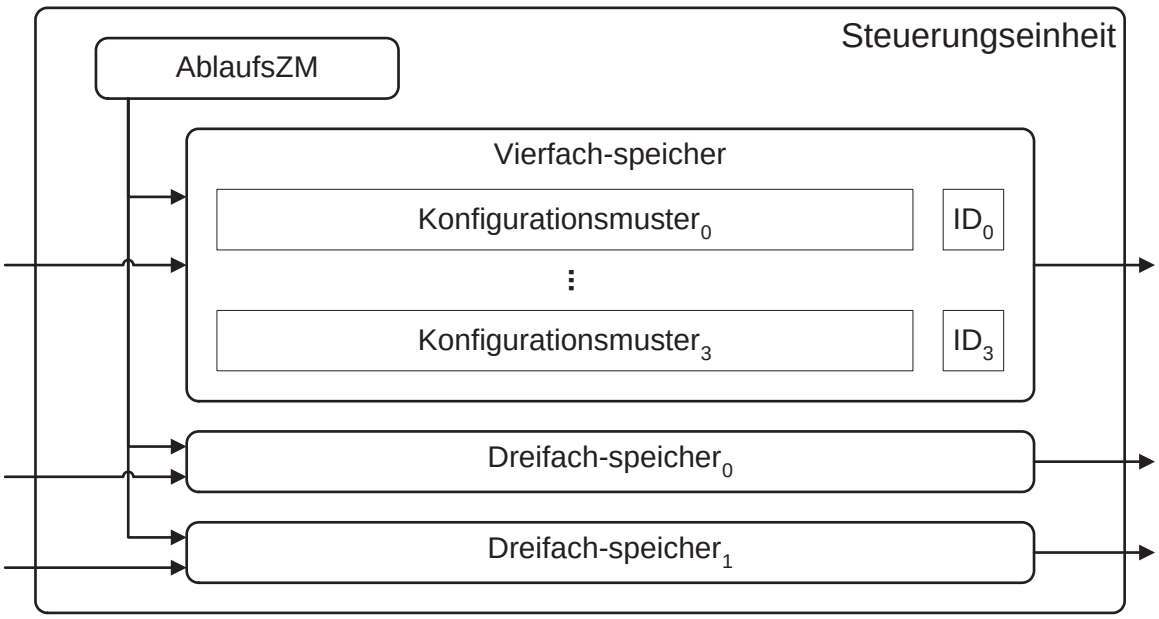

Abbildung 6.27: Aufbau der Steuereinheit

Die *Steuereinheit* (Abbildung 6.27) hält eine Kopie der Konfigurationsmuster, die die Auswertungseinheit erzeugt hat. Der Hauptbestandteil der Steuereinheit ist die Ablaufszustandsmaschine (*AblaufsZM*). Diese durchläuft alle Konfigurationsmuster in Abhängigkeit von der Granularität mit der die Muster entstanden sind. Anhand der aktuellen Musterwerte sendet die AblaufsZM die Konfigurationsanweisungen an die Anpassungskomponenten. Eine Bearbeitung der vorliegenden Muster zur Nutzung als Konfigurationsmuster ist nicht notwendig, da die Anpassungskomponente redundante Konfigurationsanfragen filtert.

Wird der AblaufsZM ein Messfehler bzw. eine Abweichung vom Musterverlauf signalisiert (Leitungen fehlen in Abbildung 6.27), muss entsprechend eine (Links-/Rechts-)Verschiebung durchgeführt werden. Beides bedeutet, dass die aktuell betrachtete Position innerhalb der Muster um eins verschoben wird.

6.2.6 Hardware-Implementierung der Überwachung und Regelung

Die Implementierung der Überwachungskomponenten ist grundsätzlich nicht für den Einsatz in einem FPGA vorgesehen. Diese gehören zu den fest verdrahteten Prozessorteilen. Um einen Größenvergleich der Überwachungs- und Regelungseinheiten in Relation zur Basisarchitektur des ARMv6-Kerns durchführen zu können, werden jedoch die Syntheseergebnisse herangezogen. Gleiches gilt für die erreichten maximalen Taktfrequenzen. Als Vergleichsarchitektur steht eine parametrisierbare VHDL-Implementierung des ARM-Prozessorkerns unter der Bezeichnung Nios II zur Verfügung.

Komponente	**Subkomponente(n)**	**ALUTs**	**MHz**	**Stufen**
Überwachungszelle		9	482,63	-
Historie		107	> 500,00	2
Muster	Musterspeicher, Musterregister	309	142,73	3
	Musterspeicher	143	191,53	[1]
Überwachungsregion	Überwachungszelle, Historie, Muster	338	156,20	5
Paarvergleicher	Musteranwahl, 7 Musterpaarvergl., Paarmatrix	720	198,26	14
	Musterpaarvergl.	42	375,80	[2]
Region&Auswertung	7 Überwachungsregionen, Paarvergleicher	3.149	134,37	19

Tabelle 6.5: Hardware-Implementierung der Überwachungskomponenten

Tabelle 6.5 zeigt Auszüge aus den Hardware-Implementierungsergebnissen der einzelnen Überwachungskomponenten.

Die erste Spalte beinhaltet den Namen der Komponente. Die darauf folgende Spalte zeigt die Subkomponenten der Komponente. Unter der mit *ALUTs* bezeichneten Spalte finden sich die für eine Implementierung benötigten Logikelemente. Die nächste Spalte zeigt die erreichte Taktfrequenz der Schaltungen. Beide Angaben beziehen sich auf die Syntheseergebnisse unter Verwendung eines Stratix II - FPGAs. In der letzten Spalte sind die für die Bearbeitung notwendigen Stufen (in Prozessortakten) angegeben. Werte in eckigen Klammeren bedeuten, dass diese in der Stufenanzahl der übergeordneten Komponente bereits eingerechnet sind.

Die einzelnen Komponenten einer Überwachungsregion stehen am Anfang der Tabelle 6.5. Hierbei fehlt die Angabe Stufen für die Überwachungszelle. Diese liefert zu jedem Zeitpunkt gültige Daten an ihren Ausgängen und muss bei der Regelungsumsetzung nicht berücksichtigt werden. Eine gesamte Überwachungsregion, bestehend aus einer Überwachungszelle, einer Historie und einem Muster, liefert am Ende eines Überwachungsintervalls nach weiteren fünf Prozessortakten ein vollständig ausgewertetes Muster. Das heißt, dass alle Häufigkeiten und ein etwaiges vorliegendes zy-

klisches Anwendungsverhalten an den jeweiligen Ausgängen angezeigt werden. Für eine Überwachungsregion sind 338 ALUTs notwendig. Das Akkumulieren des Logikbedarfs der Einzelteile ist z.B. als *worst-case* Abschätzung des gesamten Logikbedarfs anwendbar. Grundsätzlich bieten größere Schaltungen mehr Möglichkeiten die vorhandenen Ressourcen der Logikelemente besser zu nutzen. Dementsprechend verringert sich der Logikbedarf beim Zusammensetzen einzelner Subkomponenten zu einer umfangreicheren Komponente. Mit einer zugesicherten Taktfrequenz von ca. 156 MHz liegt eine Überwachungsregion unterhalb der angestrebten 200 MHz Grenze. Da weder eine Optimierung des VHDL-Codes angestrebt noch optimierende Syntheseeinstellungen vorgenommen wurden, kann davon ausgegangen werden, dass mit geeigneten Maßnahmen die Frequenzgrenze erreicht wird.

Des Weiteren bilden sieben Überwachungsregionen zusammen mit einem Paarvergleicher die gesamten Überwachungs- und Auswertungskomponenten, die innerhalb des adaptiven Prozessors benötigt werden. Die Auswertung ist spätestens 19 Takte nach Erreichen des Endes eines Überwachungsintervalls abgeschlossen.

Bei der Implementierung der eigenen Komponenten wurde auf die Verwendung spezieller FPGA-Ressourcen verzichtet, damit ein Vergleich anhand der Logikelemente bzw. ALUTs im Gegensatz zum Logikbedarf des Nios-Prozessorkerns durchführbar ist. Zur Verdeutlichung der Auswirkung auf den Ressourcenbedarf einer Komponente wurde für die *Historie* eine identische Implementierung, die zur Ablage der Nutzungshistorie die Speicherressourcen des Stratix II nutzt, angefertigt: Dies reduziert die notwendigen Logikelemente auf 20 ALUTs. Zusätzlich wird ein einziger 512 Bit Speicherblock (nur ca. zu einem Achtel) genutzt. 87 ALUTs (siehe Tabelle 6.5) des ursprünglichen Verbrauchs an Logikelementen entfallen. Hierbei werden 60 ALUTs, die genau 60 1 Bit Registern entsprechen, zur Ablage der Nutzungshistorie eingespart. Die zusätzliche 27 ALUTs entfallen auf Grund der nicht mehr notwendigen Adressierung innerhalb der registerbasierten Historie. Die Adressierungsmechanismen werden ebenfalls durch die Speicherblöcke des FPGA bereitgestellt.

Als Vergleichsimplementierung des Prozessorkerns dienen die Ergebnisse zweier Syntheseläufe des Nios-Kerns, die eine obere und untere Schranke des Ressourcenbedarfs angeben: Zum einen wurde die so genannte *Standard-Core*-Ausführung des Kerns umgesetzt die 23.305 ALUTs (11.017 ALUTs + 3 x 4096 Bits) entspricht und zum anderen die so genannte *Fast-Core*-Ausführung mit insgesamt 15.656 ALUTs[19]. Mit einem Ressourcenbedarf von 3149 ALUTs liegt die vorgestellte Überwachung und Regelung zwischen 13,5% bzw. 20% des Ressourcenbedarfs des zugrunde liegenden Prozessorkerns. Zählt man die rekonfigurierbaren Ressourcen zur Basisarchitektur, ergibt sich der Mehraufwand durch die Überwachung und Regelung zu ca. 11%. Im Gegensatz zu einem statisch rekonfigurierbaren Prozessor, wie z.B. OneChip umgerechnet auf identische Kapazitäten, benötigt der adaptive Prozessor ca. 20% weniger Hardware-Ressourcen.

[19]Die Angaben der Logikelemente beziehen sich jeweils ausschließlich auf den Prozessorkern. Caches usw. wurden nicht berücksichtigt.

6.3 Simulationsergebnisse

Dieser Abschnitt stellt die wesentlichen Simulationsergebnisse des adaptiven Prozessors vor. Überwiegend wird auf die konkreten Resultate im Zusammenhang mit der in Abschnitt 5.3 theoretisch behandelten Überwachung und Regelung adaptiver Komponenten eingegangen. Im Weiteren folgt die Betrachtung des regelungsbasierten Moduleinsatzes sowie diesbezügliche Leistungsmessungen auf der Grundlage der VM-gestützten Modulnutzungserkennung und der hardware-basierten Variante unter Einbeziehung der Wandlung. Abschließend werden die Ergebnisse beider Vorgehensweisen bewertet.

6.3.1 Allgemeine Ergebnisse zur Überwachung und Regelung

Thema dieses Teilabschnitts sind die allgemeinen Rahmenbedingungen der Überwachung, die sich beim Einsatz im adaptiven Prozessor auf die Hardware-Implementierung auswirken: Zuerst wird die Wahl der Überwachungsgranularität und deren Auswirkung auf die erhaltenen Messwertreihen betrachtet. Die Datenreduktion in Bezug auf die gesammelten Nutzungshäufigkeiten ist das darauf folgende Thema. Abschließend wird die Zyklenverkürzung durch den Einsatz von Hardware-Modulen angesprochen.

Messwerte und Überwachungsgranularität

In Abschnitt 5.3.4 des autonomen adaptiven Systems wird die Wahl eines geeigneten Überwachungsintervalls T_G erörtert. Exemplarisch zeigt Abbildung 6.28 für die EEMBC-Anwendung *bezier01* Überwachungsdaten mit drei unterschiedlichen Überwachungsgranularitäten. Jedes Diagramm umfasst 150 Überwachungsintervalle. Im Diagramm 6.28 a) wurden 20, in b) 100 und in c) 200 Taktzyklen als Überwachungsintervall genutzt. Dargestellt ist jeweils das Nutzungsverhalten eines eingelagerten FMUL-Moduls. Das jeweils obere Diagramm kann als Vergrößerung des darunter liegenden interpretiert werden. Es wurde jedoch bei der Diagrammerstellung nicht darauf geachtet, dass die Startzeitpunkte identisch sind. Ziel der Gegenüberstellung ist ausschließlich die Veranschaulichung der Auswirkung unterschiedlicher Überwachungsintervalle auf die Aussagekraft der resultierenden Überwachungsdaten.

Unter Verwendung des kleinsten Intervalls ergeben sich viele sichere Nullstellen, die jedoch für eine Regelung nicht brauchbar sind, da keine Zyklenerkennung auf Grund der hohen Redundanz der Daten mehr möglich ist. Im Gegensatz dazu zeigt sich für Diagramm 6.28 c), dass das Überwachungsintervall zu lang ist, um brauchbare Daten zu erhalten. Am günstigsten erweist sich die Überwachung mit einer Granularität von 100 Taktzyklen. Die Wahl der Granularität kann auf Basis der Modulrekonfigurationszeit (ca. 110 Takte[20]) und der Modulausführungszeit (10 Takte) angenähert werden. In der Praxis zeigt sich, dass eine Überwachungsgranularität, die sich an der Modulrekonfigurationszeit und Ausführungszeit orientiert – Abweichungen von bis zu ±30% sind akzeptabel –, stets zu geeigneten Überwachungsdaten führt.

[20] Dieser Wert ergibt sich rechnerisch aus der, auf Grund der Größe des zugehörigen Modul-Bitstreams und der Konfigurationsschnittstelle des adaptiven Prozessors zu erwartenden, Rekonfigurationszeit.

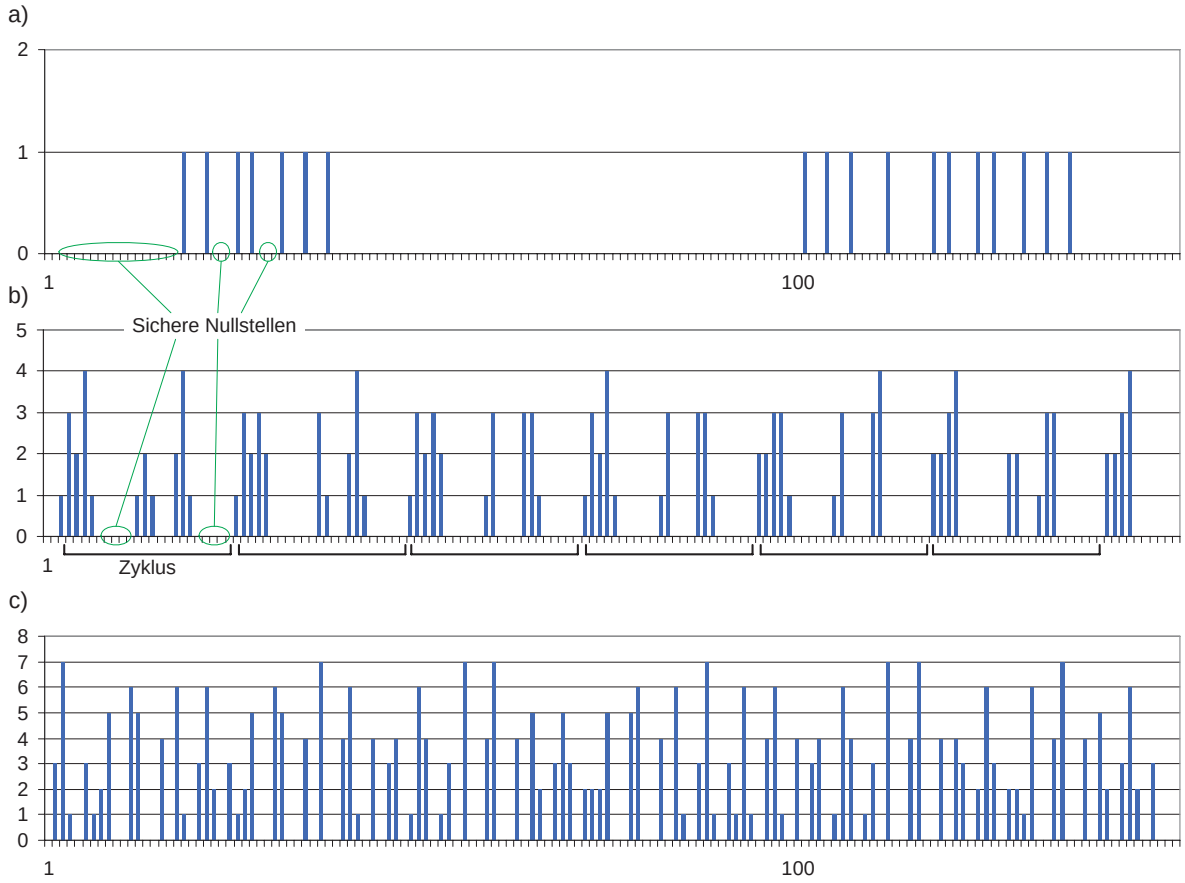

Abbildung 6.28: Auswirkung der Messintervalle auf die Überwachungsdaten

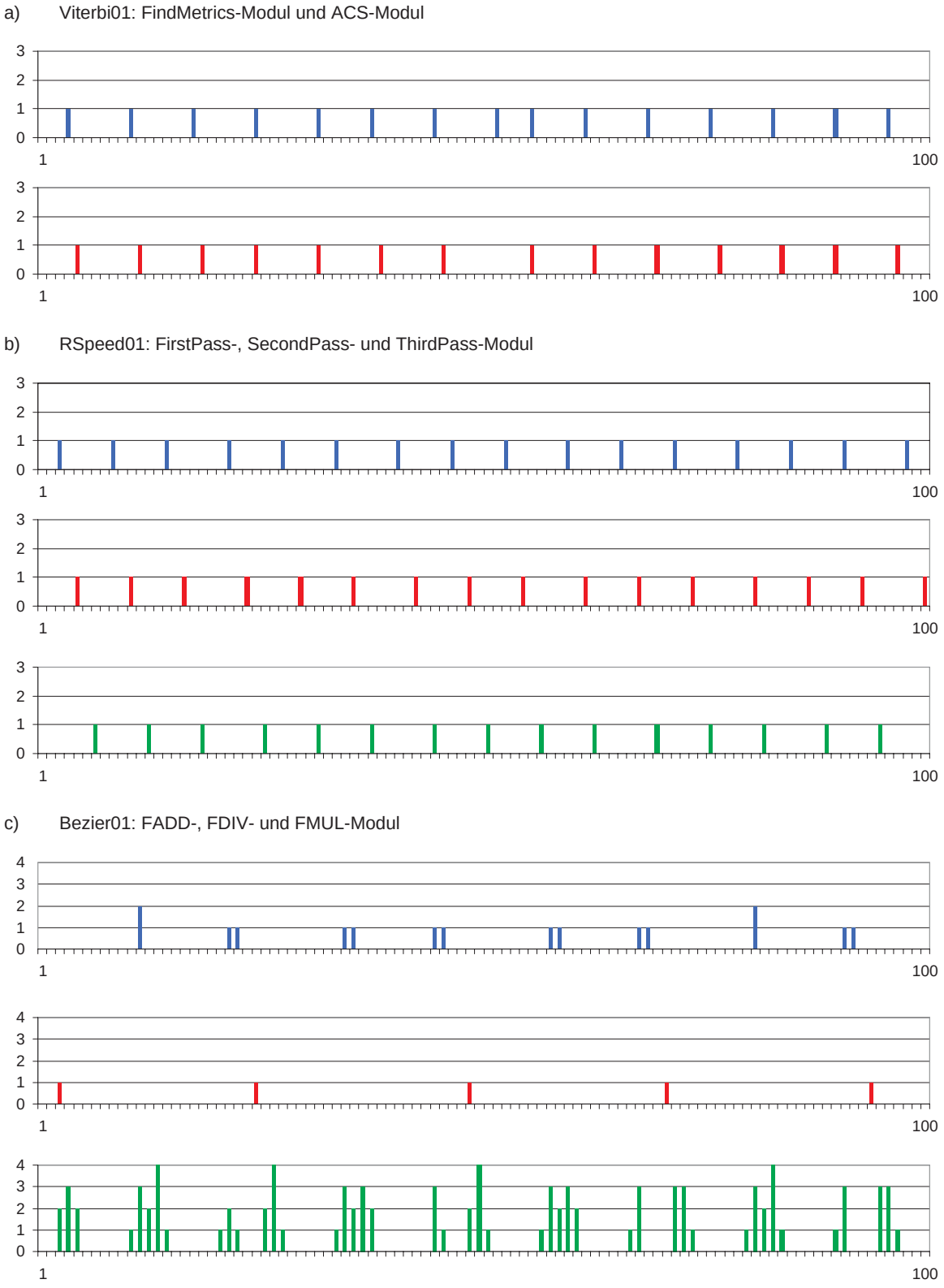

Abbildung 6.29: Beschränkung der Nutzungshäufigkeit bei geeigneter Granularität

Zudem ist in Diagramm 6.28 b) erkennbar, dass das Auftreten der sicheren Nullstellen gut prognostizierbar ist. Alle übrigen Messwerte unterliegen stets Schwankungen, die das Resultat von Messfehlern, bedingt durch die Überwachungsintervalle, sind.

Beschränkung der Messwertzähler

Abbildung 6.29 zeigt einen Ausschnitt der Überwachungsdaten dreier Beispielanwendungen. Aus Platzgründen wurden für die Anwendung *Viterbi01* nur die Daten zweier Module aufgeführt. Mit Ausnahme des *FMUL*-Moduls der Anwendung *Bezier01* sind die gemessenen Nutzungshäufigkeiten bei geeignetem Überwachungsintervall in der Regel in einem Bereich von 0 bis 3. Dies wurde bei der Implementierung der Überwachungseinheiten berücksichtigt und die korrespondierenden Überwachungsspeicher auf 2 Bit pro Intervall beschränkt. Entsprechend treten zusätzliche Messfehler auf Grund von Zählerbeschränkungen auf. Diese sind für die Erkennung zyklischen Nutzungsverhaltens nicht von Bedeutung. Bei der Korrelation des Nutzungsverhaltens zweier Module kann dies die Gewinnberechnung beeinträchtigen. In den Anwendungsbeispielen zeigt sich jedoch, wie auch in Abbildung 6.29 für die *Bezier01*-Diagramme erkennbar, dass, falls ein Modul innerhalb eines Messintervalls sehr häufig zum Einsatz kommt, andere Module stets seltener oder nie genutzt werden. Insgesamt ist bisher *kein* Fall aufgetreten, der durch die Zählerbeschränkungen zu einem unakzeptablen oder unbrauchbaren Resultat geführt hat.

Zyklenverkürzung durch Moduleinsatz

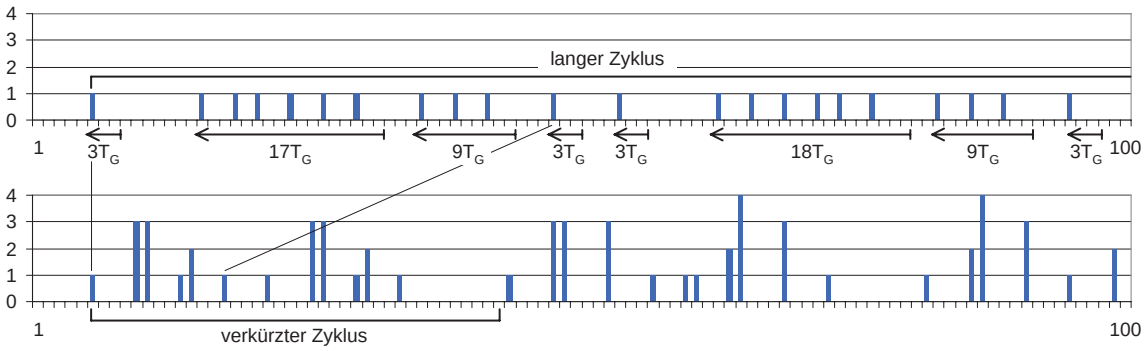

Abbildung 6.30: Zyklenverkürzung auf Grund des Einsatzes eines FMUL-Moduls

Abbildung 6.30 stellt zwei Diagramme mit jeweils 100 Takten Überwachungsgranularität gegenüber. Das obere Diagramm zeigt die aufgezeichneten Nutzungsmöglichkeiten des FMUL-Moduls. Berechnet wird die Gleitkommamultiplikation jedoch mittels Software. Im unteren Diagramm ist ein FMUL-Modul verfügbar. Angezeigt werden somit die tatsächlichen Nutzungen des Moduls. Zum Vergleich beider Datenreihen wurde jeweils der selbe Startzeitpunkt gewählt. Darüber hinaus sind in beiden Diagrammen die sich ergebenden Nutzungszyklen eingezeichnet. Zudem ist im oberen Diagramm die zeitliche Verkürzung durch den Einsatz des FMUL-Moduls mit Pfeilen angedeutet. Eine einzelne Modulnutzung beschleu-

nigt die korrespondierende Ausführung in Software um den Faktor 28,75[21]. Der Einsatz eines FMUL-Moduls bringt für die gesamte Anwendung eine Beschleunigung um den Faktor 2,36.

In diesem Zusammenhang stellt der Speedup eines Moduleinsatzes eine gute Basis für die rechnerische Annäherung an einen zu erwartenden Speedup für die gesamte Anwendung dar. Bleibt das Anwendungsverhalten gleich oder vergleichbar, kann anhand eines einzelnen Zyklus und mit Hilfe der Nutzungsmöglichkeiten und dem zu erwartenden Ausführungsbonus (in Prozessortakten), falls ein Hardware-Modul eingesetzt wird, der Beschleunigungsfaktor B der gesamten Anwendung ermittelt werden:

$$B = \frac{T_{ZA} - (N_{Zyklus} \cdot T_{Modulbonus})}{T_{ZA}} \tag{6.1}$$

T_{ZA} ist die Ausführungszeit eines Zyklus in Prozessortakten. N_{Zyklus} repräsentiert die Nutzungshäufigkeit des Moduls innerhalb eines Zyklus. Den Zeitgewinn (in Prozessortakten) durch den Einsatz eines Moduls stellt $T_{Modulbonus}$ dar. Für das FMUL-Modul ergibt sich $T_{Modulbonus} = 287,5 Takte - 10 Takte$. T_{ZA} des Zyklus ergibt sich für das Anwendungsbeispiel in Abbildung 6.30 auf Grundlage der Überwachungsintervalle zu $102 \cdot 100\ Takte$. Demzufolge ergibt sich rechnerisch $B = 2,49$ mit einer Abweichung $<5\%$ im Gegensatz zum messwertbasierten Wert von $2,36$.

6.3.2 Regelungsbasierter Moduleinsatz

Der regelungsbasierte Moduleinsatz zur Beschleunigung von Anwendungen bildete den Schwerpunkt der Untersuchungen. Um einen Vorteil gegenüber den statischen Methoden anderer Projekte zu erlangen, muss die Modulnutzung zum einen prognostizierbar und zum anderen korrelierbar sein, derart dass wenige rekonfigurierbare Bereiche ausreichen, um eine anwendungsabhängige Menge an Hardware-Modulen optimal einsetzen zu können. Die Vorhersage der Modulnutzungen ist hierbei für alle Beispielanwendungen gleichermaßen mit den vorgestellten Überwachungs- und Regelungsmethoden möglich. Andererseits verhält sich die Korrelation mehrerer prognostizierter Nutzungsreihen nicht immer optimal. Generell zeigen sich Anwendungen mit zwei Charakteristiken:

Zum einen Anwendungen, die sich auf Grund ihrer Codestruktur dergestalt in Phasen aufteilen lassen, dass jede Phase durch ein Hardware-Modul beschleunigt werden kann und die Phasen stets alternierend auftreten. Dies stellt für die regelungsbasierte Modulnutzung das optimistischste Szenario dar, da prinzipiell beliebig viele Module in zwei rekonfigurierbaren Bereichen abwechselnd eingelagert werden können, wenn man davon ausgeht, dass die Ausführungszeit jedes einzelnen Moduls die Rekonfigurationszeit des jeweiligen Nachfolgemoduls überschreitet. Zum Beispiel die Anwendung *RSpeed01* weist ein solches Verhalten auf, verfügt jedoch nur über drei Phasen respektive drei Hardware-Module. Ein Ausschnitt des gemessenen Nutzungsverhaltens ist in Abbildung 6.31 grafisch aufbereitet.

Zum anderen existieren Anwendungen mit einer Verhaltenscharakteristik die sich wesentlich diffiziler darstellt. Solches Verhalten liegt in der Regel dann vor, falls sich eine Anwendung nicht in einzelne disjunkte Phasen zerlegen lässt oder Hardware-Module, z.B. durch Ressour-

[21] Die benötigte Zeit der software-basierten Ausführung wurde durch mehrere Simulationsläufe erhoben und gemittelt. Die Ausführungszeit des FMUL-Moduls wurde mit 10 Takten angenommen (und simuliert). Dies ergibt sich aus den Syntheseergebnissen für FMUL-Module.

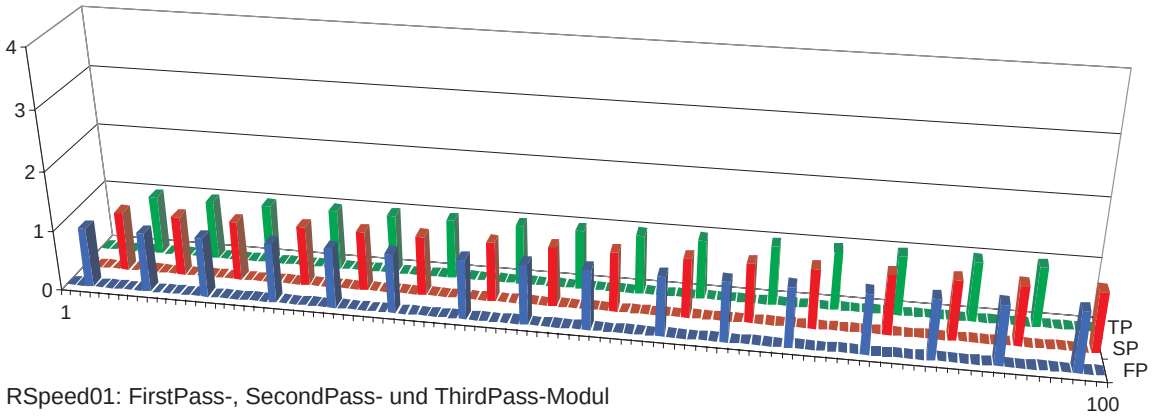

Abbildung 6.31: Modulnutzungen der RSpeed01-Beispielanwendung

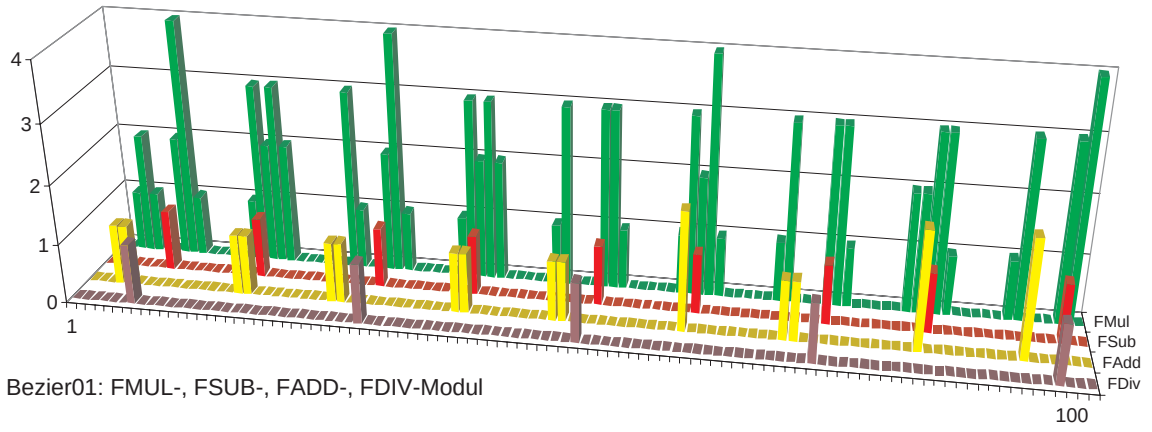

Abbildung 6.32: Modulnutzungen der Bezier01-Beispielanwendung

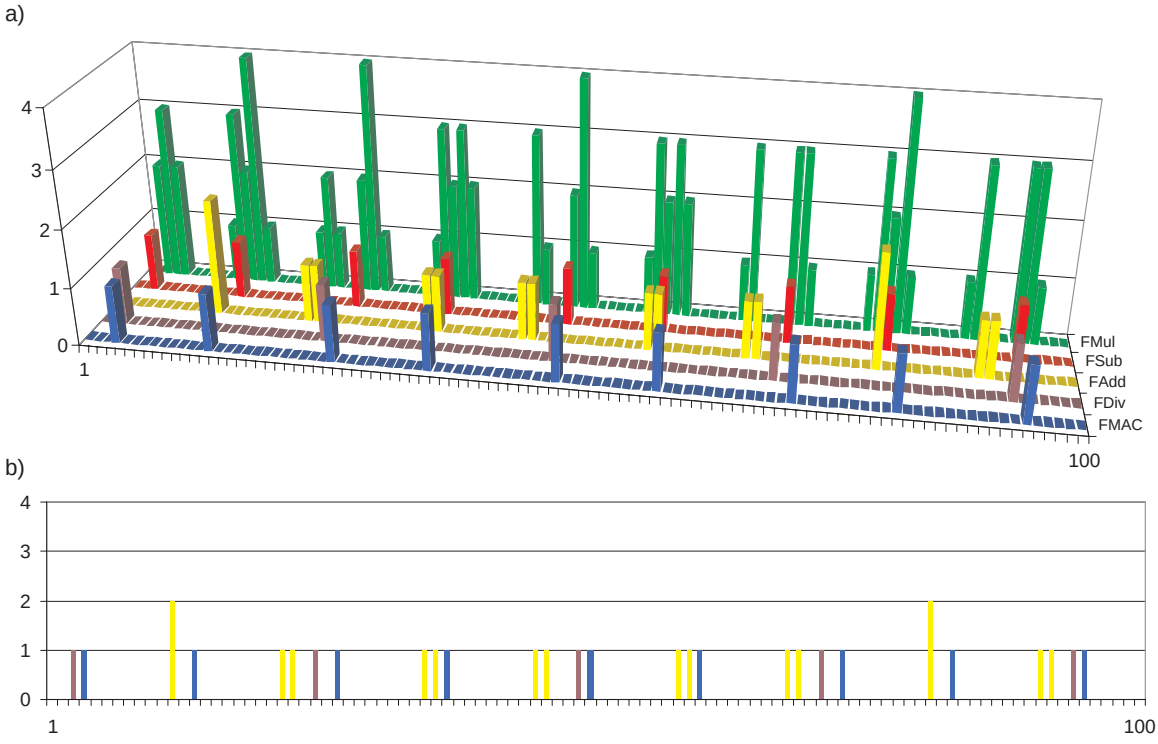

Abbildung 6.33: Zusätzliche Modulnutzungsmöglichkeit

cenbeschränkungen, keine gesamte Phase sondern nur einen Teil oder einzelne Funktionen beschleunigen können. Exemplarisch ist in Abbildung 6.32 die Modulnutzung der *Bezier01*-Anwendung illustriert. Ausschließlich die Gleitkommaoperationen wurden durch den Einsatz von Hardware-Modulen optimiert. Das resultierende Nutzungsverhalten erlaubt einen konfliktfreien Moduleinsatz für die Module *FDIV* und *FADD*. *FMUL* kann durch den häufigen Einsatz mit keinem anderen Modul sinnvoll korreliert werden. Gleiches gilt für das *FSUB*-Modul, obwohl dies im Verhältnis zur Gleitkommamultiplikation sehr selten verwendet wird.

Im Zusammenhang mit dem letzteren Anwendungsverhalten muss erwähnt werden, dass es sehr häufig die Möglichkeit gibt, einen Algorithmus manuell aufzuteilen, um ein brauchbares Modulnutzungsverhalten zu erreichen. Liegt eine günstige Aufteilung vor, zeigt sich ein zu Abbildung 6.31 vergleichbares Nutzungsverhalten. Ein derart künstlich herbeigeführtes Verhalten verlangt jedoch Hardware-Module, die in der Regel *keine* logische Einheit bilden. Zudem muss der Einsatz eines Moduls, welches sich mehrmals einsetzen ließe, unterbunden werden, damit keine Nutzungskonflikte auftreten. Für die *Bezier01*-Anwendung wurde exemplarisch eine solche Vorgehensweise versucht: Dies benötigt z.B. ein Modul für die Ausführung zweier Multiplikationen und einer nachfolgenden Addition, wobei die Multiplikationen parallel ausgeführt werden und nur die Ausgänge eines Multiplikators einen Eingang des Addierers darstellen. Dessen zweites Datum muss über einen eigenen Moduleingang von außen bezogen werden. Zusätzlich muss ein Quadrierer-Modul erzeugt werden. Dieses darf jedoch nur für eine der beiden Quadratberechnungen verwendet werden usw. Insgesamt stellt sich diese Vorgehensweise, um ein alternierendes Modulnutzungsverhalten zu erzwingen, als zu aufwendig dar. Darüber hinaus ist es unwahrscheinlich, dass eine automatische Modulerzeugung solche – teilweise unlogischen – Aufteilungen durchführt oder durchführen kann.

Dementgegen bietet sich jedoch häufig eine Lösung, die wiederverwertbare Hardware-Module berücksichtigt. Im Falle der *Bezier01*-Anwendung eröffnet sich z.B. der zusätzliche Einsatz eines *FMAC*-Moduls (*Floatingpoint MAC*). In Abbildung 6.33 a) sind die gemessenen Nutzungsmöglichkeiten des FMAC-Moduls angezeigt und den bereits eingesetzten Modulen vorgelagert. Zur Verdeutlichung des Nutzungsverhaltens präsentiert Abbildung 6.33 b) die Nutzungen des FDIV- und FADD-Moduls sowie die Nutzungsmöglichkeiten eines FMAC-Moduls für das gleiche Zeitintervall in der zweidimensionalen Ansicht.

Auswirkung auf die Rechenleistung

Im Zuge der Untersuchungen der Modulnutzungen wurden eine Reihe von Leistungsmessungen durchgeführt. Die Leistungsbewertung der VM-basierten Modulnutzungserkennung und entsprechend eingesetzter Module wurden getrennt von der hardware-basierten Modulnutzungserkennung vorgenommen.

Das Diagramm in Abbildung 6.34 trägt die erhaltenen Beschleunigungsfaktoren in y-Richtung ab. Der jeweils linke Balken repräsentiert den Beschleunigungsfaktor im Verhältnis zum simulierten adaptiven Prozessor ohne Modulnutzung. Die rechten Balken zeigen die Leistung relativ zum ARM10-Prozessor. Für die Anwendung *Bezier01* ist keine Leistungsangabe des ARM10 verfügbar. Durch die Modulnutzung ergeben sich Beschleunigungsfaktoren von ca. 2,0 bis 9,5.

Im Gegensatz dazu sind die Werte für die hardware-basierte Modulnutzungserkennung wesentlich schlechter. Abbildung 6.35 zeigt die Ergebnisse wiederum in Form eines Diagramms.

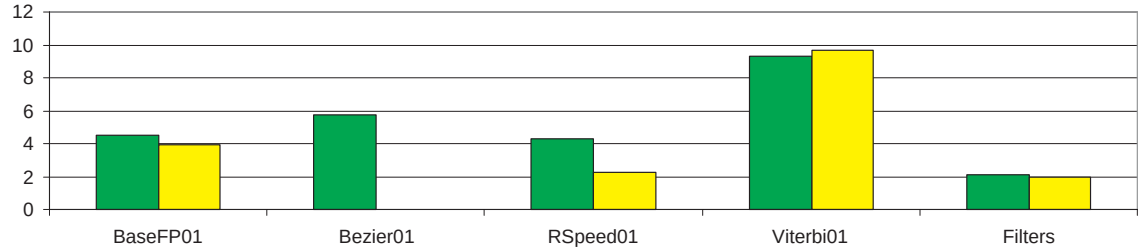

Abbildung 6.34: Beschleunigungsfaktoren der VM-basierten Modulnutzung

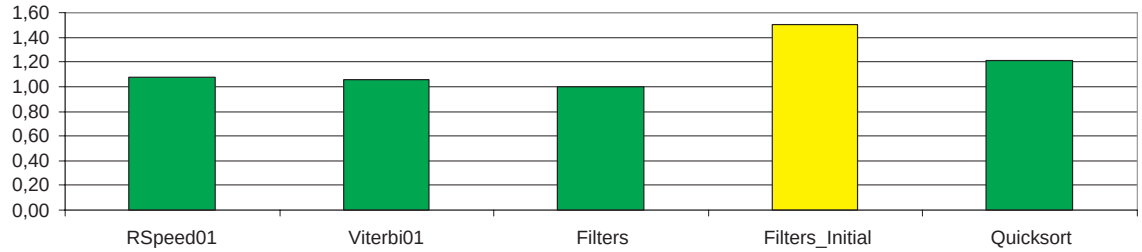

Abbildung 6.35: Beschleunigungsfaktoren der hardware-basierten Modulnutzung (Operationskombinationen)

Angeführt wurden ausschließlich diejenigen Anwendungen, die überhaupt eine Auswirkung aufweisen. Bis auf wenige Ausnahmen konnte durch den Einsatz von Modulen zur Unterstützung von Operationskombinationen keine signifikante Verbesserung der Rechenleistung nachgewiesen werden.

Das Ziel, Anwendungscode ohne erneute Compilierung für den adaptiven Prozessor zu beschleunigen wurde nicht erreicht. Statische Codeanalysen zeigen, dass üblicherweise bis zu 10% der Instruktionen einer für den ARM10 übersetzten Anwendung in Form von Operationskombinationen zusammengefasst werden können. Diesbezügliche Optimierungsmöglichkeiten stellen sich jedoch bei der Ausführung als äußerst begrenzt dar. Der Compiler versucht Registerabhängigkeiten, die für den Einsatz von Operationskombinationen notwendig sind, zu vermeiden (z.B. durch das Zwischenlagern von Speicherzugriffen etc.) und dies gelingt häufig innerhalb der für die Ausführungszeit relevanten Codeteile. Dementgegen finden sich nicht aufgelöste Registerabhängigkeiten öfters in Funktionen, die zur Anwendungs- bzw. Dateninitialisierung genutzt werden. Exemplarisch ist die Betrachtung der Dateninitialisierung der *Filters*-Anwendung in Abbildung 6.35 abgetragen. Die Initialisierung ist um 49,8% beschleunigt worden. Auf die gesamte Ausführungszeit von *Filters* hat dies jedoch keinen messbaren Effekt. Die einzige – bisher identifizierte – Anwendung, die vom Einsatz der Operationskombinationen profitiert, ist eine Implementierung des *Quicksort*-Algorithmus. Der Beschleunigungsfaktor beträgt 1,209 und somit eine ca. 20% Laufzeitverbesserung.

6.3.3 Ergebnisbewertung

Grundsätzlich zeigen alle untersuchten Anwendungsprogramme ein zyklisches und prognostizierbares Verhalten. Unabhängig von einer VM- oder hardware-basierten Modulnutzungserkennung ist stets eine zuverlässige Vorhersage des zukünftigen Einsatzes von Hardware-Modulen möglich.

Die hardware-basierte Modulnutzungserkennung beschränkt sich auf die optimierte Ausführung von Operationskombinationen mittels geeigneter Hardware-Module. Das Ziel, bereits übersetzte Anwendungsprogramme ohne zusätzliche Manipulation des Maschinencodes optimiert auszuführen, wurde nicht erreicht. Die Simulation zeigt, dass einzelne Programmphasen, wie z.B. die Initialisierung von Datenbereichen, um bis zu 50% beschleunigt werden können. Dennoch ist im Allgemeinen, auf die gesamte Rechenzeit einer Anwendung bezogen, keine oder nur eine geringe Beschleunigung erkennbar. Ausschließlich ausgesuchte Anwendungen zeigen mit einer bis zu 20% verkürzten Rechenzeit eine geringe Laufzeitverbesserung. In diesem Zusammenhang muss zukünftig untersucht werden, ob mit vertretbarem Aufwand eine Anpassung eines Compilers sinnvoll ist, damit diese Art der Modulnutzung gewinnbringend eingesetzt werden kann.

Im Gegensatz dazu zeigt die VM-gestützte Modulnutzung stets eine Leistungssteigerung um Faktoren zwischen 2,0 und 9,5. Die erzielte Leistungssteigerung ist maßgeblich von der Anzahl der einsetzbaren Hardware-Module sowie deren Funktionsumfang abhängig. Sowohl die Anzahl verfügbarer Module als auch der Funktionsumfang eines Moduls ist grundsätzlich durch die verfügbaren rekonfigurierbaren Ressourcen beschränkt. Speziell im hier behandelten Einsatzgebiet eingebetteter Systeme sind die notwendigen Ressourcen möglichst minimal zu halten. Gelten keine oder wenige Restriktionen bezüglich des Ressourcenbedarfs sind weitaus höhere Beschleunigungsfaktoren zu erwarten. Darüber hinaus verringert sich der Aufwand für die Überwachungs- und Regelungs-Hardware anteilig, wenn rekonfigurierbare Ressourcen in größerem Umfang bereitgestellt werden. Dementsprechend muss in Zukunft die Rentabilität des Einsatzes rekonfigurierbarer Hardware, wie dieser in der vorliegenden Arbeit vorgestellt wurde, auf die Eignung innerhalb von General-Purpose-Prozessoren untersucht werden.

Abschließend lässt sich festhalten, dass der vorgestellte Ansatz für die Nutzung rekonfigurierbarer Hardware in einem Prozessor für eingebettete Systeme im Gegensatz zu bestehenden Ansätzen ca. 20% weniger Ressourcen in Anspruch nimmt und vergleichbare Beschleunigungsfaktoren liefert.

7.

Zusammenfassung und Ausblick

Im Verlauf der Untersuchungen feingranularen Anwendungsverhaltens mit dem Ziel einer geeigneten Prognose zur Nutzung dieses Verhaltens für den dynamischen, optimierten Einsatz rekonfigurierbarer Ressourcen ist das Konzept des autonomen adaptiven Gesamtsystems entstanden.

Das autonome adaptive Gesamtsystem stellt sich in fünf Schichten dar. Die vier horizontalen Schichten, Anwendungs- und Übersetzer-, Betriebssystem-, Virtuelle-Maschinen- und Hardware-Schicht, werden durch die vertikale *Performance and Power Management*-Schicht, die alle Überwachungs- und Regelungsmechanismen der einzelnen Ebenen umfasst, geschnitten. In Bezug auf die Überwachung und Regelung werden die Systembibliothek und der Komponentenpool als zusätzliche Systemkomponenten, die für den Einsatz adaptiver Komponenten notwendig sind, eingeführt. Beide dienen der Verwaltung der *Hardware-Module*, zum Teil anwendungsspezifische, teils einsatzbereichsspezifische logische Schaltungen, die dem gesamten System zur Rekonfiguration einzelner adaptiver Komponenten bereitstehen. Der Komponentenpool beinhaltet alle im System verfügbaren Hardware-Module. Die Systembibliothek übernimmt die systemweite Veröffentlichung der Module und Moduleigenschaften.

Die Nutzung der Anpassungsfähigkeit erfordert verschiedene Erweiterungen in den einzelnen Systemschichten. Besonders für die Anwendungsübersetzung stellt die Gruppe adaptiver Recheneinheiten, die eine Teilmenge adaptiver Komponenten repräsentieren, eigene Anforderungen, die zum einen notwendig sind und zum anderen unterstützend zur dynamischen Anpassung beitragen. Das Einfügen so genannter *Bibliotheksinstruktionen* in den VM-Code einer übersetzten Anwendung stellt in diesem Zusammenhang die Grundvoraussetzung der dynamischen Adaptivität dar. Darüber hinaus ist eine automatisierte Modul-Generierung vorgesehen, die sowohl eine hardware-beschleunigte als auch reine softwarebasierte Ausführung in Abhängigkeit vom Konfigurationszustand des Systems erlaubt. Des Weiteren können Konfigurationsinstruktionen zur Einhaltung von Laufzeitkriterien bzw. Echtzeitanforderungen genutzt werden.

Der Einsatz rekonfigurierbarer Logik bringt diverse Probleme mit sich, die ein Resultat der zeitaufwendigen Konfigurationsvorgänge sind. Diesbezüglich bietet das Konzept Betriebssystemerweiterungen, z.B. in Form spezieller Scheduling-Techniken, und Vorschläge zur Gestaltung einer eigenen Konfigurationsinfrastruktur der betroffenen adaptiven Komponenten, die zu einer Minimierung und teilweise einer Elimination der Latenzproblematik führen.

Alle Systemschichten übernehmen Aufgaben in Bezug auf die Überwachung und Regelung des adaptiven Gesamtsystems: Die Übersetzerschicht bereitet auf Grund statischer Anwendungsinformation und Kenntnis der Systemeigenschaften den resultierenden VM-Code ent-

sprechend mit geeigneten Bibliotheksinstruktionen auf. Das Betriebssystem legt dynamisch die Zielsetzung des Systems fest und schafft die Rahmenbedingungen für eine feingranulare Anpassung der darunter liegenden Hardware-Komponenten. Zu den vorrangigen Aufgaben des Betriebssystems zählen z.B. die Regulierung der Spannung und Frequenz einzelner Systemteile und Komponenten unter Berücksichtigung der eigenen Zielsetzung, die sich an den aktuellen Umgebungsbedingungen orientiert. Zusätzliche Rahmenbedingungen die durch das Betriebssystem bedient werden müssen, sind z.B. die Auswahl geeigneter Hardware-Module und die rechtzeitige Pufferung der vordringlich benötigten Module. Diese Vorbereitungen bilden die Grundlage der Überwachung und Regelung innerhalb der so genannten *aktiven adaptiven Komponenten* der Hardware-Schicht, die eigenständig zur optimierten Ausführung einer Anwendung beitragen.

Anhand des adaptiven Prozessors wird die prinzipielle Struktur adaptiver Komponenten sowie die Integration und Umsetzung der notwendigen Überwachungs- und Regelungseinheiten aufgezeigt. Diesbezüglich wird erläutert, wie der Ressourcenbedarf möglicher Hardware-Module den Aufbau und Umfang der im Prozessor verfügbaren rekonfigurierbaren Ressourcen beeinflusst. Darüber hinaus wird eine Konfigurationsinfrastruktur vorgestellt, die den Anforderungen adaptiver Komponenten genügt. Den Hauptbestandteil bildet jedoch die Integration der Überwachung und Regelung, die den feingranularen dynamischen Einsatz der Hardware-Module, mit weniger rekonfigurierbaren Ressourcen als dies statische Lösungen erfordern, realisiert. Hierfür werden die Nutzungsmöglichkeiten sowie die tatsächlichen Nutzungen der Module überwacht. Das Überwachungsergebnis jedes einzelnen Moduls wird zuerst autokorreliert und bei geeignetem Verhalten mit den Überwachungsdaten anderer Module korreliert. Entsprechend dieser Auswertung werden die Module *getrackt* und auf dieser Basis die stark begrenzten rekonfigurierbaren Ressourcen weitgehend optimal durch mehrere unabhängige Module nutzbar.

Obwohl die Prognose der Modulnutzungen zum Beispiel durch den Einsatz von ARMA-Prozessen durchführbar ist, scheitert diese Vorgehensweise an dem notwendigen rechnerischen Aufwand. Deshalb wurde ein neuartiger Überwachungsansatz auf Basis von Nutzungshistorien und Mustern umgesetzt, der sowohl die Haltung der Überwachungsdaten als auch die Prognose zukünftigen Modulnutzungsverhaltens kombiniert. Diese Vorgehensweise erlaubt die Auswertung der Überwachungsdaten und die zugehörige Regelung in Zeiträumen von 20 Prozessortakten und folglich eine minimale Überwachungsgranularität unterhalb von 100 Prozessortakten. Dies ist selbst mit einer Hardware-Implementierung eines stark vereinfachten ARMA-Prozesses nicht zu erzielen. Zudem skaliert der gewählte Ansatz sehr gut und kann prinzipiell für beliebig große Überwachungsintervalle in Abhängigkeit von den zu observierenden Modulen bzw. deren Eigenschaften eingesetzt werden[1]. Der Ressourcenbedarf der Hardware-Implementierung der gesamten Überwachungs- und Regelungseinheiten beläuft sich auf ca. 20% im Vergleich zur Basisarchitektur des adaptiven Prozessors. Zählt man die rekonfigurierbaren Ressourcen zur Basisarchitektur, ergibt sich durch die Überwachung und Regelung ein Mehraufwand von ca. 11%. Im Gegensatz zu einem statisch rekonfigurierbaren Prozessor, wie z.B. OneChip, benötigt der adaptive Prozessor ca. 20% weniger Hardware-Ressourcen. Die gemessenen Beschleunigungsfaktoren zwischen 2 und 10,

[1]Module, die eine Überwachungsgranularität von mehreren 10.000 Takten fordern, führen auf Grund des Zeitscheibenverfahrens der Anwendungsabarbeitung zu keiner Regelung. Dies liegt an der Länge einer Zeitscheibe, nicht an den Fähigkeiten der Hardware-Überwachung.

durch den geregelten Einsatz von Hardware-Modulen, weisen zu vergleichbaren Prozessoren ähnliche Leistungssteigerungen auf.

Insgesamt kann festgestellt werden, dass die Überwachung feinkörnigen Anwendungsverhaltens und eine entsprechende Regelung des Moduleinsatzes zu einer optimierten Nutzung rekonfigurierbarer Bereiche und dies zu einer Reduzierung der notwendigen rekonfigurierbaren Ressourcen führt. Dass das Modulnutzungsverhalten mit adäquatem Aufwand beobachtet, vorhergesagt und zu einer Regelung beitragen kann, belegen die Untersuchungen an diversen Bewertungsprogrammen der EEMBC-Benchmark.

Der vorgestellte Ansatz zum Einsatz rekonfigurierbarer Hardware lässt sich leistungssteigernd in einen Prozessor für eingebettete Systeme integrieren. Durch die steigende Leistungsfähigkeit rekonfigurierbarer Hardware und das zunehmende *Platzangebot* in General-Purpose-Prozessor-Chips muss in Zukunft untersucht werden, ob sich das dargestellte Konzept für die Integration in General-Purpose-Prozessoren eignet. Hierfür spricht zum einen dass mit umfangreicheren rekonfigurierbaren Ressourcen leistungsfähigere Hardware-Module möglich werden und zum anderen sich der Hardware-Aufwand für die Überwachungs- und Regelungsmechanismen anteilig zum Prozessorkern und zu den rekonfigurierbaren Ressourcen verringert.

Unabhängige Forschungsarbeiten zeigten zwischenzeitlich, dass bereits statisch eingesetzte *FPGA-Acceleratoren* die Leistung gängiger Prozessoren im Bereich des Hoch- und Höchstleistungsrechnens übertreffen. In diesem Zusammenhang bietet sich das in dieser Arbeit vorgestellte dynamische Vorgehen an, um entscheidend zur Verringerung der eingesetzten FPGA-Ressourcen und einem effizienteren Einsatz der Acceleratoren in Form von Hardware-Modulen beizutragen.

7.1 Ausblick und Weiterführendes

Die bisher durchgeführten Untersuchungen und deren Ergebnisse bilden das Fundament für den Einsatz feingranularer Überwachung und Regelung innerhalb der Hardware-Komponenten adaptiver Systeme. Um eine Eignung der vorgestellten Mechanismen für kommerzielle Systeme – über die bisher erfolgreiche grundsätzliche Eignung hinaus – zu zeigen, müssen weitere Untersuchungen auf umfangreichere und komplexere Anwendungen ausgedehnt werden. In diesem Zusammenhang ist es z.B. erforderlich, die Energieeffizienz von Hardware-Modulen verstärkt in die Regelung mit einzubeziehen.

7.1.1 Nächste Arbeitsschritte

Konkret liegen die nächsten Schritte weiterführender Arbeiten im Ausbau des aktuell vorliegenden Simulators für den adaptiven Prozessor. Für komplexere reale Anwendungen ist das bisher manuelle Verfahren zur Erstellung und Integration von Modulen zu zeitaufwendig. Dies bedeutet, dass die Erzeugung von VM-basiertem Code und die automatisierte Generierung von Hardware-Modulen vorangetrieben werden muss. Hierbei können z.B. die Ergebnisse der Garp-Nachfolgeprojekte (Vgl. [97]) diesen Schritt entscheidend beschleunigen. Speziell das in diesem Zusammenhang entstandene Software-Hardware-Co-design eignet sich grundsätzlich für die Code- und Modulgenerierung des autonomen adaptiven Systems.

Im weiteren Verlauf sind zusätzlich verschiedene Punkte zu eruieren. Hierzu zählen folgende:

- Zur Einhaltung von Echtzeitanforderungen sind aktuell zwei Prioritäten von Konfigurationsbefehlen vorgesehen. Diesbezüglich muss geklärt werden, ob eine feinere Prioritätsabstufung für den Einsatz in realen eingebetteten Systemen sinnvoll und notwendig ist.
- Die aktuell im adaptiven Prozessor eingesetzten Hardware-Module beschränken sich auf maximal je vier Eingangs- und Ausgangsregister. Dies bietet sich an, da diese Vorgehensweise einfach mit dem Befehlssatz der ARM-Basisarchitektur in Einklang gebracht werden kann. Andererseits beschränkt dies die Möglichkeiten der Modulgenerierung. Es müssen weitere Wege untersucht werden, wie z.B die Einführung eines Register-Stacks für Hardware-Module, um hier eine größere Flexibilität zu erreichen.
- Für den Zugriff auf Überwachungsdaten des adaptiven Prozessors wurden zusätzliche Register vorgeschlagen. In diesem Kontext sollte untersucht werden, ob es nicht sinnvoll ist einen zusätzlichen Prozessormodus einzuführen, der neben den wenigen auslesbaren Registern zusätzliche Statusinformation der Überwachung und Regelung verfügbar bzw. manipulierbar macht.

7.1.2 Zukünftige Arbeitsschritte

Neben den vordringlichen Aufgaben existieren weitere, die Bestandteil zukünftiger Untersuchungen sein sollten. In diesem Abschnitt wird lediglich ein einzelner Aspekt, der sich auf Grund aktueller Entwicklungen der kommerziellen Prozessorarchitekturen abzeichnet, herausgegriffen:

Zur weiteren Reduktion des relativen Anteils rekonfigurierbarer Ressourcen zum Gesamtressourcenbedarf eines Systems sollte eine *Dual-Core*-Ausführung des adaptiven Prozessors mit geteilten gemeinsamen rekonfigurierbaren Kapazitäten in Betracht gezogen werden. Abbildung 7.1 zeigt den prinzipiellen Aufbau einer solchen Architektur. Jeder der beiden adaptiven Prozessoren verfügt über einen oder mehrere lokale rekonfigurierbare Bereiche. Darüber hinaus wird ein umfangreicher Bereich zur wechselweisen gemeinsamen Nutzung bereitgestellt. In diesem Zusammenhang muss die Nutzungsüberwachung und Regelung des gemeinsamen Bereichs prozessorübergreifend durchgeführt werden. Zur Verwaltung der lokalen rekonfigurierbaren Bereiche bleibt die Überwachung und Regelung unverändert.

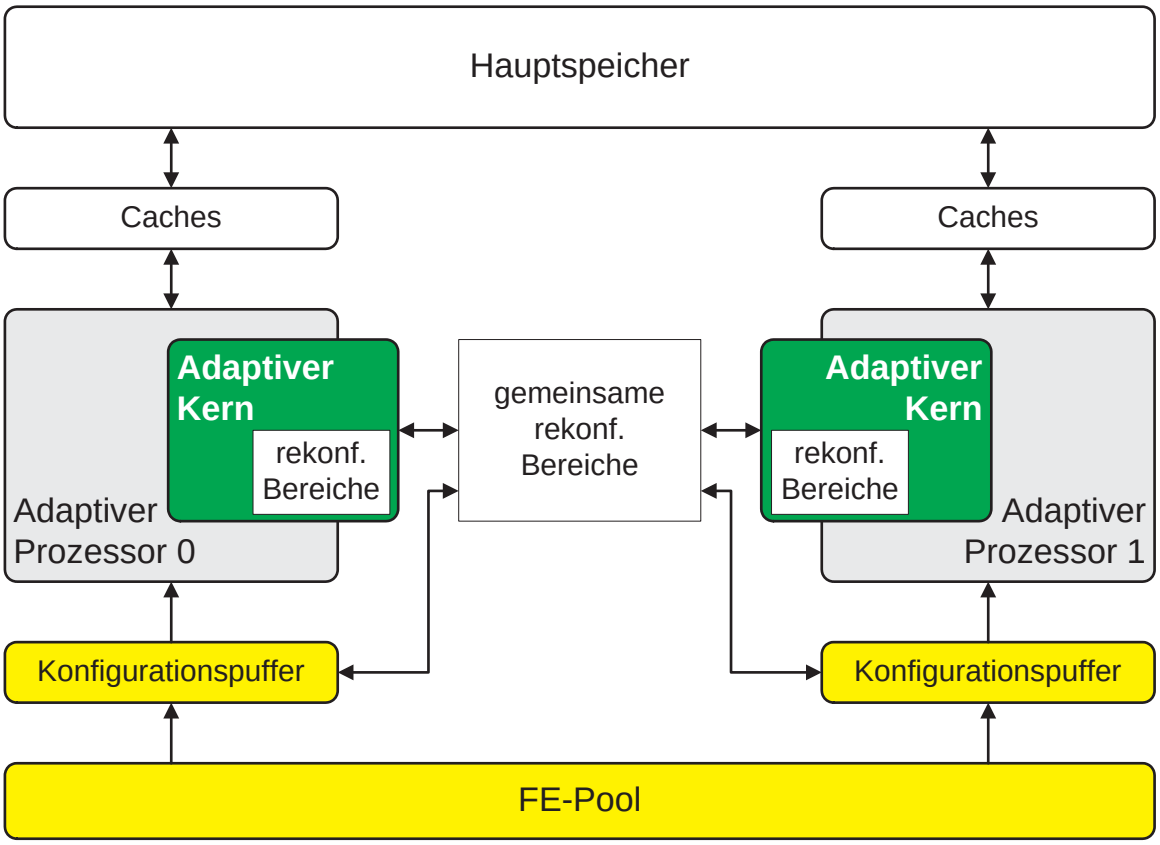

Abbildung 7.1: Geteilter gemeinsamer Rekonfigurationsbereich

A.

Simulation des adaptiven Prozessors

Als Grundlage der Simulationsdaten sowohl zur Erhebung der Daten bezüglich der Überwachung und Regelung als auch der Leistungsbewertung wurde ein eigener Prozessorsimulator entworfen.

Der Simulator bildet grundsätzlich die Architektur des ARMv6-Prozessorkerns mit den notwendigen Erweiterungen zum adaptiven Prozessor (Kapitel 6) nach. Die Prozessormodi sowie diverse Befehlssatzerweiterungen des ARMv6-Kerns stehen bisher noch nicht zur Verfügung. Dies begründet sich darin, dass vordringlich die für diese Arbeit relevanten Architekturmerkmale implementiert wurden. Hierzu zählen eine möglichst genaue Abbildung der Hardware, umfangreiche Überwachungs- und Anpassungsmöglichkeiten sowie die Unterstützung des ARM-Basisbefehlssatzes, der bis zum ARMv4-Befehlssatz aufwärtskompatibel ist. Oberste Priorität bei der Realisierung hatte die Ausführung von übersetztem C/C++ Anwendungscode. Dies ist mit wenigen Einschränkungen gelungen.

Der Einsatz eines bestehenden Prozessorsimulators, wie z.B. SimpleScalar [28], scheiterte unter anderem daran, dass der adaptive Prozessor über bis zu drei Pipelines verfügt. Die Pipelines müssen dynamisch an- und abgeschaltet werden. Des Weiteren erfordert der Einsatz von Modulen mit Operationskombinationen eine Manipulation des internen Befehlswortes nach der Dekodierungsphase. Die Anpassungsfähigkeit des adaptiven Kerns erfordert das dynamische An- und Abkoppeln von Hardware-Modulen usw. Ein zusätzlicher Punkt, der gegen die Verwendung eines bestehenden Simulators spricht, ist die Art und der Umfang der Überwachungseinrichtungen. Die Überwachung ist ein essentieller Bestandteil des adaptiven Prozessors und muss entsprechend in die internen Abläufe integriert werden. Aus dieser Situation heraus entstand ein eigener Prozessorsimulator.

A.1 Aufbau des Simulators

Für die Implementierung des Simulators wurde C++ gewählt. Jedes physikalische Teilwerk wird durch ein eigenes Objekt repräsentiert. Einzelne Komponenten innerhalb der Teilwerke sind wiederum als Objekte eingebettet. Zum Beispiel beinhaltet das Objekt der Dekoderphase vier Objekte[1], welche die einzelnen Dekoder darstellen. Diese Vorgehensweise erlaubt eine globale Verarbeitungslatenz für die Prozessorpipeline festzulegen und darüber hinaus einzelne Komponenten mit zusätzlichen zum Teil instruktionsabhängigen Latenzen zu versehen.

[1] Im Gegensatz zum adaptiven Prozessor können in der Simulation bis zu vier Pipelines aktiviert werden.

Die Überwachungseinheiten sind ebenfalls in der zu Kapitel 5 korrespondierenden Weise als Objekte organisiert.

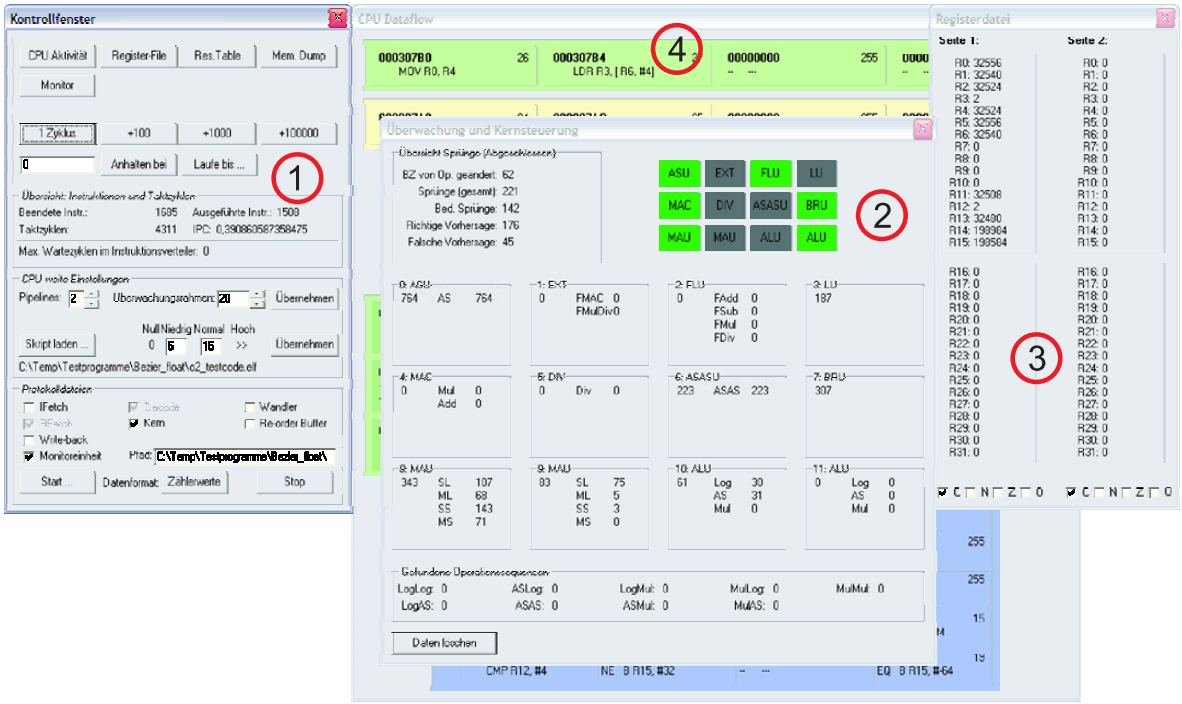

Abbildung A.1: Screenshot des Simulator-GUI

Zur besseren Handhabung wurde der Simulator mit einer grafischen Benutzerschnittstelle für *Microsoft Windows* versehen. Die wichtigsten Fenster der Oberfläche sind in Abbildung A.1 erkennbar:

1. **Kontrollfenster:** Das Kontrollfenster ermöglicht den Zugang zu den wichtigsten Funktionalitäten des Simulators. Alle weiteren Fenster sind dem Kontrollfenster untergeordnet. Vorrangig können hier alle globalen dynamischen Einstellungen des Simulators verändert werden. Hierzu zählen die zyklengenaue Weiterschaltung des Prozessors, eine einfache *Breakpoint*-Behandlung sowie Pipeline-Einstellungen und Überwachungseinstellungen. Alle Überwachungseinheiten die auf Zählerbasis arbeiten, unterstützen zwei Betriebsmodi: Zum einen die Zählung in Form von 32 Bit Registern und zum anderen die 2 Bit Datenhaltung, die für die Hardwareimplementierung der Überwachungs- und Regelungselemente herangezogen wurde. Letztere kann zusätzlich durch die Angabe von Schwellwerten beeinflusst werden.

2. **Überwachung und Kernsteuerung:** Das Fenster Überwachung und Kernsteuerung zeigt die grundlegenden Überwachungsinformationen zu den einzelnen verfügbaren Funktionseinheiten des adaptiven Kerns. Im oberen rechten Fensterbereich sind die verfügbaren Hardware-Module angezeigt. Die Anzeige basiert auf Schaltflächen, die ein manuelles An- und Abschalten der Module zur Laufzeit erlauben. Das Abschalten eines Moduls erfolgt mit einem Takt Verzögerung, vorausgesetzt, das Modul

führt zu diesem Zeitpunkt keine Operation aus. Das Anschalten wird mit moduleigenen Latenzen derart vollzogen, dass diese mit entsprechender Verzögerung rechenbereit sind. Alle weiteren Anzeigen in diesem Fenster repräsentieren die aktuellen Überwachungsinformationen der einzelnen Module. Zusätzlich werden im unteren Fensterbereich mögliche Operationskombinationen angezeigt.

3. **Registerdatei:** Im Registerdatei-Fenster sind stets die aktuellen Registerwerte (Allzweckregister) und die Statusbits dargestellt. Die Registerdatei des Simulators ist in zwei Seiten zu je zwei Bänken organisiert. Genutzt wird aktuell ausschließlich die erste Seite der ersten Bank. Die Statusbits des Prozessors lassen sich manuell ändern.

4. **CPU Dataflow:** Das *CPU Dataflow*-Fenster dient der Beobachtung der internen Abläufe und wurde vorrangig zu deren Verifikation eingesetzt. Aus Platzgründen kann dieses Fenster stets nur einen Ausschnitt der gesamten Prozessorpipeline anzeigen.

Zusätzlich zu den vier beschriebenen Fenstern des GUI, existieren drei weitere Fenster: Das **Res.Table**-Fenster erlaubt einen detaillierten Einblick in den aktuellen Status der Reservierungstabelle. Mit Hilfe des **Mem.Dump**-Fensters kann der gesamte Speicherbereich, der einer simulierten Anwendung bereitgestellt wird, auszugsweise eingesehen werden. Der **Skript laden**-Dialog ermöglicht ein skript-gesteuertes Laden einer Binärdatei, die nach der Anwendungsübersetzung im so genannten *ELF*-Format verfügbar ist.

A.2 Anwendungsübersetzung und Ausführung

Alle Beispielanwendungen liegen in ANSI C vor und wurden mit dem GNU *gcc*-Compiler übersetzt. Ein Lader, der die Positionierung im simulierten Hauptspeicher vornimmt, ist im Simulator integriert. Darüber hinaus können mit Hilfe einer einfachen Skript-Datei vorab einzelne Register- und Speicherinhalte gesetzt werden. Dies ist vor allem für die Positionierung des Befehlszählerstandes notwendig.

Als Einsprungadresse wurde für die Simulation stets die Speicheradresse, die dem Aufruf der vom *gcc* eingebauten `__gccmain` -Funktion folgt, gewählt. Dies beugt evtl. auftretenden Umschaltungen in einen anderen Prozessormodus vor. Andererseits schränkt dies die Nutzung von Datenstrukturen ein. Deshalb wurden zum Beispiel dynamische Datenstrukturen in den EEMBC-Benchmarkprogrammen durch statische Strukturen ersetzt. Bei eigenen Anwendungsimplementierungen wurde generell auf dynamische Strukturen verzichtet. In weiterführenden Arbeiten muss diese Einschränkung behoben werden.

A.2.1 Einschränkungen und bekannte Fehler

Der Simulator unterstützt den kompletten AMRv4-Befehlssatz. Die Koprozessor-Instruktionen sind bisher für die Nutzung von Hardware-Modulen reserviert.

Speicherzugriffe, die einen 16 Bit Wert (*halfword*) in ein Register laden, werden in seltenen Fällen nicht korrekt durchgeführt. Obwohl fehlerhafte Zugriffe ein reproduzierbares Verhalten aufweisen, konnte der Fehler bisher nicht behoben werden. Aus diesem Grund wurde bei Beispielanwendungen versucht die Halfword-Zugriffe durch 32 Bit Zugriffe zu ersetzen.

Des Weiteren zeigt sich in Ausnahmefällen ein Problem, falls zwei gleichberechtigte Speichereinheiten im adaptiven Kern verfügbar sind. In diesem Zusammenhang werden scheinbar WAW- und WAR-Hazards nicht immer richtig aufgelöst.

Von der simulierten Anwendung unabhängig kommt es in seltenen Fällen zu einer Speicherzugriffsverletzung des Simulators. Dieser Fehler ist bisher nicht reproduzierbar und konnte nicht behoben werden.

A.2.2 Modulnutzung

Die Modulnutzung wird durch die manuelle Integration – solange kein Compiler verfügbar ist – der korrespondierenden Bibliotheksinstruktion realisiert. Um den Aufwand der Änderungen zu minimieren, werden generell Sprungbefehle durch Bibliotheksinstruktionen ersetzt. Liegt in einer Anwendung ein zu optimierendes Codestück nicht als Funktionsaufruf vor, wurde dies entsprechend im C-Code abgeändert und der Quellcode neu übersetzt. Anschließend folgt die Ersetzung des Sprungbefehls durch die Bibliotheksinstruktion.

Zur Markierung von Modulnutzungsmöglichkeiten, wie diese die VM über die VM-Schnittstelle des adaptiven Prozessors vornimmt, wurden ebenfalls Sprungbefehle eingesetzt. Hierzu wird das zweit höchste Bit des Sprungoffsets im Maschinenbefehl verwendet. Ist das Bit im Originalcode gesetzt, wird es gelöscht und umgekehrt. Manipulierte Offsets erkennt die Instruktionsholeinheit des Simulators und macht die Änderung rückgängig. Zudem wird die Markierung an die Überwachungseinheiten weitergeleitet. Damit mehrere Modulnutzungsmöglichkeiten aufgezeigt werden können, müssen derzeit wiederholte Simulationsläufe der Anwendungscodes mit unterschiedlich markierten Sprungbefehlen durchgeführt werden. Diese Vorgehensweise ist für größere Anwendungen als die bisher untersuchten nicht immer zulässig und muss demzufolge im Weitern verändert werden.

A.3 Verifikation

Zur Verifikation des Simulators wurden zum einen Vergleiche zum ARM10 gezogen und zum anderen die Ergebnisdaten der einzelnen simulierten Anwendungen mit den Daten, die eine IA32-Übersetzung des gleichen Codes liefert, verglichen.

Abweichungen im Laufzeitverhalten gegenüber dem ARM10 Prozessor sind mit Ausnahme des Effekts, der sich durch den Einsatz von *Register Renaming* zeigt, nachvollziehbar und können anhand der Überwachungsdaten einer Simulation rechnerisch zurückverfolgt werden:

- Ausgeführte Speicherzugriffe erfolgen in der Simulation stets auf dem *First-Level*-Cache und unterliegen einer Latenz von einem Taktzyklus. Eine Anbindung an einen Speichersimulator ist vorgesehen aber nicht umgesetzt.
- Die Sprungvorhersage des Simulators erfolgt statisch. Entsprechend schlechter stellt sich diese im Vergleich zur dynamischen Sprungvorhersage des ARM10 dar. Darüber hinaus löscht der Simulator bei nicht richtig vorhergesagtem Sprung die gesamte Prozessorpipline. Hierdurch entsteht eine Latenz von sieben Takten. Weitere Maßnahmen zur Verringerung der Latenz wurden bisher nicht verfolgt. Im Vergleich dazu kann der ARM10 eine falsche Vorhersage mit einer Strafzeit von einem einzigen Prozessortakt abwickeln.

- Die Änderung des Befehlszählers durch Instruktionen, die keine Sprungbefehle sind, führt in der Simulation stets zum Leeren der Pipeline. Für den ARM10 werden auch solche Befehle in die Sprungvorhersage mit einbezogen.

A.4 Anmerkungen zur Implementierung des Simulators

Grundsätzlich müssen zwei Aspekte der Implementierung für weitere Arbeiten verbessert werden:

- Die Dekodierung der ARM-Befehle basiert auf der Simulatorimplementierung *Operator* [57]. Diese erlaubte eine sehr schnelle Implementierung des Prozessor-Front-Ends des Simulators. Im Verlauf der weiteren Implementierung stellte sich jedoch heraus, dass die Anforderungen für den adaptiven Prozessor weitreichende Änderungen der Dekodierung verlangen. Um in möglichst kurzer Zeit eine adäquate Lösung zu erhalten, wurde der ursprünglichen Dekodierung eine zusätzliche Dekodierung nachgeschaltet. Hierbei hat sich eine äußerst *unschöne* und komplexe Implementierung ergeben, die zusätzlich durch die zweifache Dekodierung unnötig langsam ist. Zudem erfordert dies bei der Integration eines neuen Hardware-Moduls die Änderung zweier Objekte des Simulators. Durch das fortwährende Erweitern des Standardbefehlssatzes um neue Bibliotheksinstruktionen sind in der Zwischenzeit für die Erweiterung um neue Instruktionen umfangreiche Codeänderungen des Simulators notwendig.

- Ein weiterer Aspekt, der unbedingt vereinfacht werden muss, ist die Integration eines neuen Hardware-Moduls in den adaptiven Kern. Derzeit wird ein Modul durch ein Container-Objekt repräsentiert, welches die Funktionalität des Moduls mit Hilfe zusätzlicher Objekte beinhaltet. Dies erlaubt eine sehr hardware-nahe Realisierung des simulierten Moduls, da z.B. einzelne physikalische Teilwerke eines Moduls entsprechend auf Software abgebildet werden können. Insgesamt stellt sich diese Vorgehensweise als zu unflexibel heraus und sollte im Weiteren optimiert werden.

A.5 Ausgabe der Überwachungsdaten

Die in der grafischen Benutzerschnittstelle (siehe Abschnitt A.1) angezeigten Überwachungsdaten repräsentieren nur einen Teil der gesamten gesammelten Information. Zusätzlich zur internen Nutzung für die Regelung des adaptiven Prozessors eignen sich die Daten als Debugging-Information.

Zur Umleitung der Überwachungsdaten in Log-Dateien können die Einstellungen im Kontrollfenster der Benutzerschnittstelle verwendet werden.

Die Daten werden in mehreren Textdateien ausgegeben und liegen in einem *csv*-konformen[2] Format vor. Dies erleichtert die Übernahme der Daten in Tabellenkalkulationsanwendungen, wie z.B. Excel, zur grafischen Aufbereitung oder gezielten manuellen Auswertung.

[2] Einträge sind stets durch Semikolon getrennt.

Glossar

Adaptive Komponente: Adaptive Komponenten sind Hardware-Komponenten eines Systems, die über ein oder mehrere interne rekonfigurierbare Bereiche verfügen. Die Funktionalität und Schnittstelle einer adaptiven Komponente kann auf Grund der Rekonfigurationsfähigkeit zur Laufzeit verändert werden.

Adaptive Recheneinheit: Adaptive Recheneinheiten sind eine echte Teilmenge adaptiver Komponenten. Eine adaptive Recheneinheit ist im Gegensatz zu anderen adaptiven Komponenten in der Lage Maschinenbefehle auszuführen.

Bibliotheksinstruktion: Eine Bibliotheksinstruktion ist ein Maschinenbefehl der direkt mittels eines zugehörigen Hardware-Moduls ausgeführt werden kann. Die Zuordnung von Hardware-Modulen zu Bibliotheksinstruktionen wird mit Hilfe der Systembibliothek durchgeführt. Zur Ausführungszeit muss vor der Abarbeitung jeder Bibliotheksinstruktion festgestellt werden, ob die aktuelle Hardware-Konfiguration der betroffenen adaptiven Komponente ein entsprechendes Hardware-Modul enthält und eine Verarbeitung zulässt. Ist dies nicht der Fall, muss die Bibliotheksinstruktion durch die zugehörige Maschinenbefehlssequenz der Systembibliothek ersetzt und demzufolge ohne Hardware-Beschleunigung abgearbeitet werden.

Bitstream: Ein Bitstream bezeichnet im Zusammenhang mit Hardware-Modulen stets die Konfigurationsdaten eines Moduls.

Funktionseinheitenpool: Der Funktionseinheitenpool umfasst alle Hardware-Module, die sich ausschließlich für den Einsatz in einer oder mehreren adaptiven Recheneinheiten eignen. Er bildet eine Teilmenge des Komponentenpools.

FPGA-Kontext: Kontexte eines rekonfigurierbaren Bereichs (oder FPGAs) sind integrierte Speicherbereiche, die Konfigurationsdaten der rekonfigurierbaren Ressourcen aufnehmen können. Die rekonfigurierbaren Ressourcen können stets nur mit den Daten eines einzigen Kontextes konfiguriert werden. Ein Kontext wird als aktiv bezeichnet, wenn die rekonfigurierbaren Ressourcen mit den Daten dieses Kontextes konfiguriert sind. Bei einem Kontextwechsel erfolgt eine Umkonfiguration der Ressourcen anhand der Daten eines anderen Kontextes. Durch diese zeitnahe Art der Rekonfiguration entsteht der Eindruck, dass gleichzeitig mehrere physisch unabhängige, rekonfigurierbare Ressourcen vorhanden sind. Ein Kontext kann ein oder mehrere Hardware-Module aufnehmen. Die Beschränkung der

Anzahl von Modulen innerhalb eines Kontextes ergeben sich aus bestehenden Schnittstellenvorgaben und der Größe der Konfigurationsdaten der Module. Im Bereich der Betriebssysteme wird die Zustandsinformation eines Prozesses ebenfalls als Kontext bezeichnet. Bei einem FPGA-Kontext handelt es sich sozusagen um den Behälter, der die Zustandsinformation für die Konfiguration rekonfigurierbarer Ressourcen beinhaltet.

Granularitätsintervall: Die Sammlung von Überwachungsdaten erfolgt in vorgegebenen Granularitätsintervallen. Treten mehrere überwachte Ereignisse innerhalb eines Granularitätsintervalls auf, sind diese zeitlich nicht unterscheidbar.

Komponentenpool: Im Komponentenpool befinden sich alle Hardware-Module des Systems. Die Module sind im Komponentenpool in Form von Konfigurationsdaten (Modul-Bitstreams) abgelegt.

Konfigurationsinstruktion: Konfigurationsinstruktionen stellen eine Gruppe von Maschinenbefehlen dar, die ausschließlich die Hardware-Konfiguration einer adaptiven Komponente beeinflussen. Konfigurationsinstruktionen können von einem Compiler in den regulären Maschinencode integriert werden, um an ausgesuchten Codepositionen das Einlagern von Hardware-Modulen zu erzwingen.

Musterspeicher: Musterspeicher dienen der Erzeugung und Verwaltung von Ereignismustern. Die Erzeugung eines Musters kann wahlweise anhand der Daten einer Nutzungshistorie oder direkt mittels der Daten einer oder mehrerer Überwachungszellen erfolgen.

Modul: Ein Modul oder Hardware-Modul erlaubt die Ausführung gewisser Funktionalität in Hardware. Bezüglich des Funktionsumfangs existieren prinzipiell keine Einschränkungen. Das heißt, ein Modul kann die Hardware-Implementierung eines einzelnen Maschinenbefehls, eines Algorithmus, einer gesamten Funktionseinheit zur Ausführung unterschiedlicher Operationen usw. repräsentieren. Einschränkungen existieren jedoch auf Grund verschiedener Schnittstellenvorgaben sowie Beschränkungen der verfügbaren Hardware-Ressourcen. Ein Modul entspricht somit einer logischen Schaltung, deren Schnittstelle vordefiniert und Ressourcenbedarf begrenzt ist, damit dieses innerhalb adaptiver Komponenten eingesetzt werden kann. Das Laden eines Moduls in einen rekonfigurierbaren Bereich bedeutet, dass dieser mit den entsprechenden Daten konfiguriert wird und abschließend die Funktionalität des Moduls als Hardware-Implementierung bereitstellt.

Nutzungshistorie: Eine Nutzungshistorie wird anhand der Überwachungsdaten einzelner Granularitätsintervalle erstellt. Die Auswertung einer Nutzungshistorie erfolgt am Ende eines Überwachungsintervalls. Hardware-Komponenten, die der Erstellung und Verwaltung von Nutzungshistorien dienen, werden ebenfalls als Nutzungshistorien bezeichnet.

Operationskombination: Eine Operationskombination bezeichnet die Fähigkeit mehrere einzelne Maschinenbefehle mit echten Registerabhängigkeiten zu einem einzelnen Maschinenbefehlswort zu kombinieren. Die Erkennung und Kombination wird nicht von einem

Compiler durchgeführt, sondern erfolgt zur Laufzeit in Hardware. Die Ausführung eines solchen kombinierten Befehlswortes ist nur dann möglich, wenn ein Hardware-Modul mit der notwendigen Funktionalität verfügbar ist.

(Re-)Konfiguration: Der Begriff Rekonfiguration beschreibt das wiederholte Konfigurieren rekonfigurierbarer Hardware-Ressourcen. Eine Konfiguration bedeutet, dass die rekonfigurierbaren Ressourcen mit den Konfigurationsdaten einer logischen Schaltung derart beschrieben werden, dass nach Abschluss der Konfiguration deren Funktionalität verfügbar ist. Die Begriffe Rekonfiguration und Konfiguration werden synonym verwendet, da eine Unterscheidung zwischen einmaliger Konfiguration und wiederholter Konfiguration für diese Arbeit unnötig ist.

Überwachungselemente: Die einzelnen Hardware-Komponenten die ausschließlich der Überwachung und Steuerung des Systems dienen sind Überwachungselemente.

Überwachungsintervall: Ein Überwachungsintervall ist der Zeitraum innerhalb dessen ausschließlich Überwachungsdaten gesammelt werden. Am Ende jedes Überwachungsintervalls beginnt die Auswertung der Daten und etwaige Steuermechanismen werden angestoßen. Ein Überwachungsintervall umfasst mindestens ein Granularitätsintervall.

Überwachungszelle: Überwachungszellen registrieren die mit Hilfe von Sensoren festgestellten Ereignisse. Der Hauptbestandteil einer Überwachungszelle ist ein Ereigniszähler. Die Ereigniszähler werden stets am Ende eines Granularitätsintervalls ausgelesen und zurückgesetzt.

Sensor: Sensoren sind diejenigen Überwachungselemente, die die Erkennung überwachter Ereignisse realisieren.

Systembibliothek: Die Systembibliothek referenziert alle dem System verfügbaren Hardware-Module sowie die korrespondierenden Software-Umsetzungen.

Literaturverzeichnis

[1] S. Abdelwahed u. N. Kandasamy. Online Control for Self-Management in Computing Systems.

[2] G. Acher. *JIFFY - Ein FPGA-basierter Java Just-in-Time Compiler für eingebettete Systeme*. Doktorarbeit, Technische Universität München, 2003.

[3] Advanced Microdevices Inc. AMD Alchemy Processor Family. http://www.amd.com/us-en/ConnectivitySolutions.

[4] Advanced Microdevices Inc. Elan SC520 Microcontroller. http://www.amd.com/epd/processors/4.32bitcont/14.lan5xxfam/24.lansc520/index.html.

[5] Advanced RISC Machines. *ARM 7TDMI Data Sheet*, 1995. http://www.arm.com/.

[6] Advanced RISC Machines. *ARM10 Thumb Family - 300MHz low-power system-on-a-chip processor*, 2000. http://www.arm.com/.

[7] Advanced RISC Machines. *ARM9E-S Technical Reference Manual*, 2002. http://www.arm.com/.

[8] Altera Corporation. *FFT - MegaCore Function User Guide*, Juni 2004.

[9] Altera Corporation. *FIR Compiler - MegaCore Function User Guide*, Dezember 2004.

[10] Altera Corporation. *Reed-Solomon Compiler - User Guide*, Juli 2004.

[11] Altera Corporation. *Stratix II Device Handbook*, 2005. Dokumentnummer: SII5V1-4.0.

[12] Altera Corporation. *Stratix II GX Device Handbook, Preliminary Information*, 2005. Dokumentnummer: SIIGX5V1-2.0.

[13] Altera Corporation. *Viterbi Compiler - User Guide*, Februar 2005.

[14] Ravi Ambatipudi u. Clive Watts. Leistung nach Bedarf: Dynamisches Energie-Management für Embedded-Systeme. *World of Embedded ARM*, Februar 2004. http://www.elektroniknet.de.

[15] Annapolis Micro Systems, Inc. *Wildfire Reference Manual*, 1998.

[16] ARM. Intelligent Energy Manager: Battery life extension through Intelligent Energy Conservation, 2003. ARM DOI 0171-1/07.03(4).

[17] ARM. ARM Core Families, 2005. http://www.arm.com/ARM%20Core%20Families.htm.

[18] ARM. ARM Core Selector, 2005. http://www.arm.com/ARM%20Core%20Selector.htm.

[19] ARM. Intelligent Energy Manager (IEM), 2005. http://arm.convergencepromotions.com/catalog/178.htm.

[20] ARM. The ARM Instruction Set Architecture, 2005. http://www.arm.com/The%20ARM%20Instruction%20Set%20Architecture.htm.

[21] K. J. Åström. *Control System Design*. 2002. http://www.cds.caltech.edu/ murray/courses/cds101/fa02/caltech/astrom.html.

[22] R. I. Bahar u. S. Manne. Power and energy reduction via pipeline balancing. In *Proceedings of 28th International Symposium on Computer Architecture (ISCA)*, S. 218–229, Juli 2001.

[23] Iris Bahar. VLIW and EPIC: Advanced Computer Architecture, Oktober 2002. http://www.amigau.com/aig/index.html.

[24] H.-G. Beyer, E. Brucherseifer, W. Jakob, H. Pohlheim u. a. Evolutionäre Algorithmen - Begriffe und Definitionen, Juni 2001. Universität Dortmund.

[25] Kiran Kumar Bondalapati. *Modeling and Mapping for dynamically reconfigurable Hybrid Architectures*. Doktorarbeit, University of Southern California, August 2001.

[26] D. Brash. The ARM Architecture Version 6 (ARMv6), Januar 2002. ARM White Paper.

[27] R. Buchty, J. Jeitner, J. Tao, W. Karl u. a. ASoCS - An Architecture Concept for Self-optimizing Parallel and Distributed Computer Systems. In *Proceedings of PARS 2005*, Juni 2005.

[28] D. Burger u. T. M. Austin. The SimpleScalar tool set, version 2.0. Technischer Bericht, Computer Sciences Department,University of Wisconsin, Madison, Juni 1997.

[29] T. J. Callahan, J. R. Hauser u. J. Wawrzynek. The Garp Architecture and C Compiler. In *IEEE Computer*, S. 62–69, April 2000.

[30] Timothy J. Callahan, John R. Hauser u. John Wawrzynek. The Garp Architecture and C Compiler - Overview, 2000. http://www.computer.org.

[31] Chameleon Systems, Inc. *CS2000 Reconfigurable Communications Processor Family Product Brief*, 2000.

[32] O. Cordon, F. Herrera, F. Hoffmann u. L. Magdalena. Genetic Fuzzy Systems, Evolutionary Tuning and Learning of Fuzzy Knowledge Basis, 2001.

[33] D. K. Day. Genetische Algorithmen und ihre Anwendung zur Prognose finanzwirtschaftlicher Daten. Diplomarbeit, Ludwig-Maximilians-Universität München, Februar 1999.

[34] D.Brooks u. M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, S. 171–182, Januar 2001.

[35] Digital Equipment Cooperation. *Alpha 21164 Microprocessor Hardware Reference Manual*, 1995. Document Number: EC-QAEQB-TE.

[36] EEMBC. EEMBC, 2005. http://www.eembc.org.

[37] D. Eisenbach. Künstliche Neuronale Netze zur Prognose von Zeitreihen. Diplomarbeit, Westfälische Wilhelms-Universität Münster, März 2005.

[38] Embedded Expert. embedded Motorola PowerPC, 2004. http://www.embeddedexpert.com/processor/motorola_mpc.html.

[39] Embedded Expert. embedded systems expert technology, 2004. http://www.embeddedexpert.com/.

[40] Embedded Expert. IBM PowerPC Prozessoren, 2004. http://www.embeddedexpert.com/processor/ibm_power.html.

[41] Embedded Expert. Renesas Hitachi Super H Prozessoren, 2004. http://www.embeddedexpert.com/processor/hitachi.html.

[42] H. Engesser [Hrsg.]. *Informatik Duden*. BI Wissenschaftsverlag, 2. Aufl., 1993. ISBN 3-411-05232-5.

[43] ePanorama. General information on embedded systems, Januar 2006. http://www.epanorama.net/links/embedded.html.

[44] J. E. C. Esparza. *Evaluation of the OneChip Reconfigurable Processor*. Doktorarbeit, University of Toronto, 2000.

[45] M. Weiser et al. Scheduling for reduced CPU energy. In *proceedings of the first USENIX Symposium on operating systems design and implementation*, S. 13–23, November 1994.

[46] H. Sanchez et al. Thermal management system for high-performance powerPC microprocessors. In *Proceedings of COMPCON'97*, Februar 1997.

[47] M. Koegst et al. Low power design of FSMs by state assignment and disabling self-loops. In *Euromicro 97*, S. 323–330, September 1997.

[48] M. Motomura et al. An Embedded DRAM-FPGA Chip with Instantaneous Logic Reconfiguration. In *Proceedings of Symposium on VLSI Circuits*, S. 55–56, 1997.

[49] S. Trimberger et al. A Time-Multiplexed FPGA. In *Proceedings of Symposium on FCCM*, S. 22–28, April 1997.

[50] W.H. Mangione-Smith et al. Seeking solutions in configurable computing. In *Proceedings of IEEE Computer*, S. 38–43, Dezember 1997.

[51] U. Frank, H. Giese, F. Klein, O. Oberschelp u. a. Arbeitspapier - Definition der Selbstoptimierung, August 2004.
http://wwwmath.upb.de/ agdellnitz/papers/DefinitionSelbstoptimierung_V106.pdf.

[52] Chr. W. Frey, M. Sajidman u. H.-B. Kuntze. Fuzzy-basierte Prozeßphasenerkennung und regelung komplexer Batchprozesse, 1999.
http://www.iitb.fraunhofer.de/servlet/is/1635/FryFuzzyDo99.pdf.

[53] FSMLabs. *RTLinux*.
http://fsmlabs.com/.

[54] R. Ghanea-Hercock. Autonomous Computing, 2002.

[55] Soheil Ghiasi u. Majid Sarrafzadeh. Optimal Reconfiguration Sequence Management.
http://www.overcite.lcs.mit.edu/576949.html.

[56] VDE / ITG / GI. Der Organische Computer, Juli 2003.

[57] M. Gleirscher. Entwicklung und Implementierung eines ARM7-Prozessorsimulators, November 2003. Systementwicklungsprojekt.

[58] M. Gokhale u. J. Stone. NAPA C: Compiling for a Hybrid RISC/FPGA Architecture. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, S. 126–135, April 1998.

[59] Benjamin Goldberg. Compiler Optimizations for Modern VLIW/EPIC Architectures, April 2002. New York University.

[60] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu u. a. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *Proceedings of the 26th. Annual International Symposium on Computer Architecture*, S. 28–39, 1999.

[61] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu u. a. Piperench: A reconfigurable Architecture and Compiler. In *Computer: Innovative Technology for Computer Professionals*, Bd. 33, S. 70–77, April 2000.

[62] R. Gonzalez u. M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.

[63] Ed Grochowski u. Murali Annavaram. Energy per Instruction Trends in Intel Microprocessors. Technischer Bericht, Microarchitecture Research Lab, Intel Corporation, 2005. ftp://download.intel.com/technology/EEP/epi-trends.pdf.

[64] J. Haid, G. Käfer, Ch. Steger, R. Weiss u. a. Run-Time Energy Estimation in System-On-a-Chip Designs. In *Asia South Pacific - Design Automation Conference*, 2003.

[65] Hannibal. RISC vs. CISC: the Post-RISC Era – A historical approach to the debate, Januar 2005.
http://www.arstechnica.com/cpu/4q99/risc-cisc/rvc-1.html.

[66] J. Hartung, B. Elpelt u. K.-H. Klösener. *Statistik*. Oldenbourg Verlag, München, Wien, 7. Aufl., 1989.

[67] S. Hauck, T. W. Fry, M. M. Hosler u. J. P. Kao. The Chimaera Reconfigurable Functional Unit. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, S. 87–96, April 1997.

[68] John R. Hauser u. John Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. Technischer Bericht, University of California, Berkeley.

[69] P.J.M. Havinga u. G.J.M. Smit. Minimizing energy consumption for wireless computers in Moby Dick. In *IEEE International Conference on Personal Wireless Communication ICPWC'97*, Dezember 1997.

[70] P.J.M. Havinga u. G.J.M. Smit. *Design techniques for low power systems*, Bd. 46. 1. Aufl., 2000.

[71] S. D. Haynes, J. Stone, P. Y. K. Cheung u. W. Luk. Video Image Processing with the Sonic Architecture. In *Computer: Innovative Technology for Computer Professionals*, Bd. 33, S. 50–57, April 2000.

[72] J.L. Hennessey u. D.A. Patterson. *Computer Architecture: A Quantitive Approach*. Morgan Kaufmann Publishers, Inc., 2. Aufl., 1996. ISBN 1-55860-372-7.

[73] Hitachi Semiconductor. *SH7708 Series Hardware Manual*, Mai 1999.

[74] R. Isermann (Hrsg.). *Überwachung und Fehlerdiagnose*. VDI Verlag, 1994.

[75] C. Im, H. Kim u. S. Ha. Dynamic voltage scheduling technique for low-power multimedia applications using buffers. In *In International Symposium on Low Power Electronics and Design*, S. 34–39, August 2001.

[76] Intel. *Intel IA-64 Architecture Software Developer's Manual*, 2000. Document Number: 245319-001.

[77] Intel. *Intel Celeron Processor in the 478-Pin Package up to 1.70 GHz*, Mai 2002. Document Number: 290748-001.

[78] Intel. *DP Optimized Intel Itanium 2 Processor Datasheet*, September 2003. Document Number: 253795-001.

[79] Intel. *Intel Pentium M Datasheet*, März 2003. Document Number: 252612-001.

[80] Intel. *IA-32 Intel Architecture Software Developers Manual, Volume 2a*, 2004. Document Number: 253666.

[81] Intel. *IA-32 Intel Architecture Software Developers Manual, Volume 2b*, 2004. Document Number: 253667.

[82] Intel. *Intel Celeron M Processor*, Juni 2004. Document Number: 300302-003.

[83] Intel. *Dual-Core Intel Xeon Processor 2.80 GHz*, Oktober 2005. Document Number: 309158-001.

[84] Intel. *IA-32 Intel Architecture Software Developers Manual*, September 2005. Document Number: 253665-017.

[85] Intel. Intels New Ultra-Low Power Manufacturing Process will Stretch Battery Life, Oktober 2005.
http://www.intel.com/technology/magazine/silicon/low-power-process-1005.pdf.

[86] Intel. Intel Core microarchitecture, 2006.
http://www.intel.com/Intel Core Microarchitecture.html.

[87] J. A. Jacob. *Memory Interfacing for the OneChip Reconfigurable Processor*. Doktorarbeit, University of Toronto, 1998.

[88] Nelly Jacqueson. Windows CE for Industrial Computing. In *Proceedings Embedded Intelligence '99*, S. 532–542. Design & Elektronik, WEKA Fachzeitschriftenverlag, März 1999.

[89] Udo Jakobza. StrongARM-1100 Board für universelle Anwendungen. In *Proceedings Embedded Intelligence '99*, S. 352–362. Design & Elektronik, WEKA Fachzeitschriftenverlag, März 1999.

[90] N. K. Jha. Low power system scheduling and synthesis. In *IEEE Int. Conf. on Computer-Aided Design*, November 2001.

[91] J. Wetzel and E. Silha and C. May and B. Frey. *PowerPC User Instruction Set Architecture*.

[92] W. Karl. *Parallele Prozessorarchitekturen: Codegenerierung für Superskalare Superpipelined und VLIW-Prozessoren*. BI Wissenschaftsverlag, 1993. ISBN 3-411-16451-4.

[93] N. Kasprzyk, A. Koch, U. Golze u. M. Rock. Eine effiziente Kontrollfluss-Repräsentation für die Erzeugung von Datenpfaden, 2003. http://www.esa.informatik.tu-darmstadt.de/twiki/pub/Staff/AndreasKochPublications/2003_EIS03.pdf.

[94] A. Kawai u. T. Fukushige. $158/Gflops Astrophysical N-Body Simulation with Reconfigurable Add-in Card and Hierarchical Tree Algorithm. In *Proceedings of Supercomputing 2006*, November 2006.

[95] Gareth Knight. AMIGA History Guide – RISC, CISC, Post-RISC, 2005.
http://www.amigau.com/aig/index.html.

[96] A. Koch, N. Kasprzyk, M. Rock u. H. Lange. Adaptive Rechensysteme und ihre Entwurfswerkzeuge, 2004.
http://www.eis.cs.tu-bs.de/eis/research/current/researchAK.htm.

[97] A. Koch. Adaptive Rechensysteme und ihre Entwurfswerkzeuge, 2001.
http://www.eis.cs.tu-bs.de/eis/people/koch/2001_eis_ws.pdf.

[98] J. Kopf. Arbeitspapiere zur Zeitreihenanalyse, 2004. http://www.wifak.uniwuerzburg.de/ewf/doku/zra/ap-zra.htm.

[99] Nectarios Koziris, Alexandros Pappas, George Papakonstantinou u. Panayotis Tsanakas. Free Scheduling of General Nested Loops For Distributed Memory Architectures. http://citeseer.ist.psu.edu/22480.html.

[100] T. Kröger. Einführung in die Regelungstechnik I - Kontinuierliche Systeme, 2005. http://www.cs.tu-bs.de/rob/download/tkr/pi_ss05/teil1.pdf.

[101] T. Küfer. Optimierung von Fuzzy-Controllern mit Hilfe von Genetischen Algorithmen. Diplomarbeit, Westfälische Wilhelms-Universität Münster, Juni 2004.

[102] R. Laufer, R. R. Taylor u. H. Schmit. PCI-PipeRennch and the SwordAPI: A system for Stream-based Reconfigurable Computing. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, S. 200–208, April 1999.

[103] Hai Li, Swarup Bhunia, Yiran Chen, T. N. Vijaykumar u. Kaushik Roy. Deterministic Clock Gating for Microprocessor Power Reduction. Technischer Bericht, ECE Department, Purdue University, 2003. http://www.ece.purdue.edu/ vijay/papers/2003/dcg.pdf.

[104] G. Lu, H. Singh, M. Lee, N. Bagherzadeh u. a. The MorphoSys Parallel Reconfigurable System. In *Proceedings of Euro-Par*, September 1999.

[105] Zhijian Lu, Jason Hein, Marty Humphrey, Mircea Stan u. a. Control-Theoretic Dynamic Frequency and Voltage Scaling for Multimedia Workloads. In *Proceedings of CASES*, S. 156–163, Oktober 2002.

[106] Zhijian Lu, John Lach, Mircea Stan, John Lach u. Kevin Skadron. Reducing Multimedia Decode Power using Feedback Control. In *Proceedings of the 21st International Conference on Computer Design (ICCD03)*, 2003.

[107] W.H. Mangione-Smith u. B.L. Hutchings. Configurable computing: the road ahead. In *Reconfigurable Architectures Workshop*, 1997.

[108] S. Ogrenci Memik, E. Bozorgzadeh, R. Kastner u. M. Sarrafzadeh. A Super-Scheduler for Embedded Reconfigurable Systems. In *International Conference on Computer-Aided Design (ICCAD)*, November 2001.

[109] T. Miyamori u. K. Olukotun. A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, S. 2–11, April 1998.

[110] S. Mohanty u. V. K. Prasanna. Energy Efficient Application Design using FPGAs, Oktober 2004. http://www.fpgajournal.com/index.htm.

[111] Motorola Inc. *Motorola MC68328 Users Manual*, 1999.

[112] Motorola Inc. *Motorola ColdFire VL RISC Processors White Paper*, 2001. http://e-www.motorola.com/brdata/PDFDB/docs/MCF5XXXWP.pdf.

[113] Motorola Semiconductor Products Sector. *MPC565/MPC566 Users Manual*, September 2002. Order Number: MPC565UM/D.

[114] Motorola Semiconductor Products Sector. *MPC603e RISC Microprocessor Users Manual*, 2002. Order Number: MPC603EUM/AD.

[115] National Semiconductor. Geode Products. http://www.national.com/appinfo/solutions/0,2062,396,00.html.

[116] S. Neema, T. Bapty u. J. Scott. Adaptive Computing and Runtime Reconfiguration. 1999.

[117] S. Niendieck. *Optimierung von Fuzzy-Controllern*. Doktorarbeit, Westfälische Wilhelms-Universität Münster, 2003.

[118] Sujay Parekh, Neha Gandhi, Joe Hellerstein, Dawn Tilbury u. a. Using Control Theory to Achive Service Level Objectives in Performance Management, Oktober 2000.

[119] Pioneer-Standard Electronics. *Windows CE Overview*, 1999. http://mypioneer.com/msembedded/ce/overview.asp.

[120] R. Razdan u. M. D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *International Symposium on Microarchitecture*, S. 172–180, November 1994.

[121] P. Rechenberg u. G. Pomberger (Hrsg.). *Informatik-Handbuch*. Carl Hanser Verlag München Wien, 1999. ISBN 3-446-19601-3.

[122] M. Reuter u. S. Zacher. *Regelungstechnik für Ingenieure - Analyse, Simulation und Entwurf von Regelkreisen*. Vieweg, 11. Aufl., Februar 2004. ISBN 3-528-05004-7.

[123] C. R. Rupp, M. Languth, T. Garverick, E. Gomersall u. a. The NAPA Adaptive Processing Architecture. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, S. 28–37, April 1998.

[124] N. R. Saxena u. E. J. McCluskey. Dependable Adaptive Computing Systems - The ROAR Project, 1998. http://crc.stanford.edu/crc_papers/saxenaSMC98.pdf.

[125] Josef Schira. *Statistische Methoden der VWL und BWL*. Pearson Studium, 2003.

[126] Manfred Schlett. SH-4: 200MHz Superscalar Embedded Processor. In *Proceedings Embedded Intelligence '99*, S. 249–258. Design & Elektronik, WEKA Fachzeitschriftenverlag, März 1999.

[127] R. Schlittgen u. B. H. J. Streitberg. *Zeitreihenanalyse*. Oldenbourg Verlag, München, Wien, 8. Aufl., 1999.

[128] H. Schmeck. Organic Computing, Juli 2003.

[129] J. Scott, S. Neema u. T. Bapty. Hardware/Software Runtime Environment for Dynamically Reconfigurable Systems. Technischer Bericht, Institute for Software Integrated Systems, Vanderbilt University, Juni 2000.

[130] S. Guccione. List of FPGA-based Computing Machines, August 2000.

[131] T. Simunic u. S. Boyd. Managing power consumption in networks on chips, 2002.

[132] T. Simunic, L. Benini, A. Acquaviva, P. Glynn u. G. D. Micheli. Dynamic voltage scaling for portable systems. In *Proceedings of the Design Automation Conference*, Juni 2001.

[133] K. Skadron, T. Abdelzaher u. M. R. Stan. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, S. 17–28, Februar 2002.

[134] M. Sturm. *Neuronale Netze zur Modellbildung in der Regelungstechnik*. Doktorarbeit, Technische Universität München, Februar 2000.

[135] A. Tanenbaum. *Moderne Betriebssysteme*. Pearson Studium, 2. Aufl., 2001. ISBN 3-8273-7019-1.

[136] J. Tao, J. Jeitner, C. Trinitis, W. Karl u. J. Weidendorfer. Comprehensive Cache Inspection with Hardware Monitors. In *Proceedings of PaCT 2005*, S. 331–345, September 2005.

[137] L. Thiele. Unterlagen zur Lehrveranstaltung Technische Informatik I. ETH Zürich.

[138] Triscend Corporation. *Triscend E5 Configurable System-on-Chip Family Product Description*, 2000.

[139] Keith D. Underwood, K. Scott Hemmert u. Craig Ulmer. Architectures and APIs: Assessing Requirements for Delivering FPGA Performance to Applications. In *Proceedings of Supercomputing 2006*, November 2006.

[140] Suvas Vajracharya. *Runtime Loop Optimization for Locality and Parallelism*. Doktorarbeit, University of Colorado at Bouldern, 1997.

[141] G. Varghese, Z. Hui u. R. Jan. The Design of a Low Energy FPGA, 1999. http://bwrc.eecs.berkeley.edu/Research/Configurable_Architectures/papers/islped99.pdf.

[142] K. Weicker. *Evolutionäre Algorithmen*. Teubner Verlag, Stuttgart, Leipzig, 2002.

[143] Wikipedia. Sequentielle Monte-Carlo-Methode, November 2005. http://www.wikipedia.org/Sequentielle Monte-Carlo-Methode.

[144] Wikipedia. Tracking, Oktober 2005. http://www.wikipedia.org/tracking.

[145] Wikipedia. ARMA-Modell, 2006. http://www.wikipedia.org/regler/arma-modell.

[146] Wikipedia. Echtzeit, Januar 2006.
http://www.wikipedia.org/echtzeit.

[147] Wikipedia. EchtzeitSysteme, Februar 2006.
http://www.wikipedia.org/echtzeit-systeme.

[148] Wikipedia. Eingebettete Systeme, Juli 2006.
http://www.wikipedia.org/kalman-filter.

[149] Wikipedia. Kalman-Filter, 2006.
http://www.wikipedia.org/kalman-filter.

[150] Wikipedia. Regelungstechnik, August 2006.
http://www.wikipedia.org/regelungstechnik.

[151] Wikipedia. Regler, August 2006.
http://www.wikipedia.org/regler.

[152] Steven J.E. Wilton, Su-Shin Ang u. Wayne Luk. The Impact of Pipelining on Energy per Operation in Field-Programmable Gate Arrays. In *Proceedings of FPL*, 2004. http://www.ece.ubc.ca/ stevew/papers/pdf/fpl04.pdf.

[153] WindRiver Inc. *VxWorks 5.x Operating Systems*.
http://www.windriver.com/products/vxworks5/vxworks 54.pdf.

[154] R. D. Wittig. *OneChip: An FPGA Processor with Reconfigurable Logic*. Doktorarbeit, University of Toronto, 1995.

[155] Xilinx. *Virtex-II Platform FPGAs: Complete Data Sheet*, März 2003. Dokumentnummer: DS031 (v3.4).

[156] Xilinx. *Virtex-II Pro Platform FPGAs: Advance Product Specification*, März 2003. Dokumentnummer: DS083-1 (v2.4.1).

[157] Masakazu Yamashina u. Masato Motomura. Reconfigurable computing: its concept and a practical embodiment using newly developed dynamically reconfigurable logic (DRL) LSI. In *Proceedings of the 2000 conference on Asia South Pacific design automation*, S. 329–332, 2000. ISBN: 0-7803-5974-7.

[158] A. Zell. *Simulation Neuronaler Netze*. Addison-Wesley, 1996.

Index