Jürgen Bregenzer

Effizienter Einsatz von Multicore-Architekturen in der Steuerungstechnik

Jürgen Bregenzer

Effizienter Einsatz von Multicore-Architekturen in der Steuerungstechnik

Jürgen Bregenzer

Effizienter Einsatz von Multicore-Architekturen in der Steuerungstechnik

Würzburg University Press

Dissertation, Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik, 2014
Gutachter: Prof. Dr. Reiner Kolla, Prof. Dr. Ludwig Leurs

Impressum

Julius-Maximilians-Universität Würzburg
Würzburg University Press
Universitätsbibliothek Würzburg
Am Hubland
D-97074 Würzburg
www.wup.uni-wuerzburg.de

© 2015 Würzburg University Press
Print on Demand

ISBN 978-3-95826-010-8 (print)
ISBN 978-3-95826-011-5 (online)
URN urn:nbn:de:bvb:20-opus-106239

This document—excluding the cover—is licensed under the
Creative Commons Attribution-ShareAlike 3.0 DE License (CC BY-SA 3.0 DE):
http://creativecommons.org/licenses/by-sa/3.0/de/

The cover page is licensed under the Creative Commons
Attribution-NonCommercial-NoDerivatives 3.0 DE License (CC BY-NC-ND 3.0 DE):
http://creativecommons.org/licenses/by-nc-nd/3.0/de/

Vorwort

Jahrzehntelang befanden sich Softwareentwickler und damit auch die Entwickler der Firmware digitaler Steuerungen in der höchst komfortablen Situation, dass man ohne eigenes Zutun mit einer festen Leistungssteigerung der Software nach dem Mooreschen Gesetz rechnen konnte. Manche Engpässe erledigten sich dadurch einfach durch Abwarten auf die kommende Prozessorgeneration. Diese Zeiten sind wohl endgültig vorüber. Die Taktrate der Prozessoren lässt sich wegen des damit verbundenen Leistungsverbrauchs nicht weiter steigern, die Parallelität der Software auf Befehlsebene ist ausgereizt. Es bleibt nur noch, aus der Parallelität auf Datenebene oder der Parallelität auf Threadebene weitere Leistungssteigerungen zu schöpfen. Dies führte zur Entwicklung mehrkerniger Prozessoren und wird wohl in Prozessorarchitekturen mit vielen Kernen enden. Gleichzeitig erwächst damit die Anforderung an die Softwareentwicklung, die Systeme im Hinblick auf parallele Ausführbarkeit zu entwickeln und zu optimieren und ist, sieht man von einigen Gebieten, in denen es schon lange eine Tradition des parallelen Rechnens auf Supercomputern gibt, wie dem wissenschaftlichen Rechnen, mal ab, eine große Herausforderung für die Zukunft, insbesondere im Bereich der eingebetteten Systeme.

Aus diesem Grund war auch sofort mein Interesse geweckt, als Herr Jürgen Bregenzer an mich herantrat und mich um die Betreuung seines Promotionsvorhabens bat, welches den Einsatz von Multicore-Prozessoren im Bereich der industriellen Steuerungstechnik zum Gegenstand haben sollte. Das dabei von ihm angestrebte Vorgehen war insofern spannend, als die Arbeit im Rahmen einer Industriepromotion bei der Firma *Bosch Rexroth* in der Steuerungsentwicklung für Werkzeugmaschinen entstehen sollte. Diese Konstellation versprach, neben den theoretischen Aspekten der bearbeiteten Thematik auch zugleich den Praxisbezug zur industriellen Anwendung dieser Rechnerarchitektur zu berücksichtigen, so dass ich gerne die Betreuung der nun vorliegenden Dissertation übernahm, was mir nicht zuletzt viele interessante Einblicke in die Domäne der industriellen Automatisierung ermöglichte.

In seiner Arbeit stellt Herr Bregenzer heraus, dass der Einsatz von Multicore-Prozessoren in der Steuerungstechnik einerseits eine Notwendigkeit darstellt, um angesichts stagnierender CPU-Taktraten für anspruchsvoller werdende Steuerungsaufgaben entsprechend leistungsfähige Steuerungsplattformen zur Verfügung stellen zu können. Andererseits macht er deutlich, dass der Einsatz dieser Prozessorarchitektur insbesondere im Anwendungsfall der Systemkonsolidierung auch zahlreiche Chancen birgt. Auf diese Weise nähert er sich in seiner Arbeit dem Einsatz von Multicore-Prozessoren in der Steuerungstechnik ausgehend von zwei sehr unterschiedlichen Nutzungsstrategien und motiviert dabei sein Vorgehen mittels entsprechender Anwendungsfälle. Als wesentlicher Beitrag der Dissertation von Herrn Bregenzer kann dabei die Evaluation einer Systemkonsolidierung im Bereich der Steuerungstechnik ebenso gelten, wie der Schwerpunkt seiner Arbeit, welcher sich der Spezifikation einer modellbasierten Methodik zur Parallelisierung einer bestehenden Firmware für Multicore-Prozessoren widmet. Im Kontext der Informatik ist dabei besonders interessant, dass diese Methodik auch über den Anwendungsfall der Steuerungstechnik hinaus eingesetzt werden kann, um in anwendungsori-

entierter Weise die Parallelisierungsalternativen einer bestehenden Firmware zu evaluieren und so informierte Entscheidungen über das Vorgehen bei deren Adaption für eine Ausführung auf Multicore-Prozessoren treffen zu können.

Bedanken möchte ich mich zu guter Letzt bei der Firma *Bosch Rexroth*, insbesondere in Person von Herrn Dr. Jens Brühl, für die angenehme und unbürokratische Zusammenarbeit, welche stets durch einen hohen Grad an Professionalität geprägt war. Mein abschließender Dank gilt zudem Herrn Prof. Dr. Ludwig Leurs für das Zweitgutachten zu dieser Arbeit.

Prof. Dr. Reiner Kolla
Lehrstuhl für Technische Informatik
Institut für Informatik
Universität Würzburg

Abstract

The application of multi-core CPUs in industrial control technology holds chances as well as risks. Consequently, this thesis develops and evaluates generic strategies for using this processor architecture in due consideration of the specific framework conditions and demands of this domain.

Multi-core CPUs offer the chance of consolidating heterogeneous control subsystems currently running on dedicated hardware devices while maintaining a degree of temporal isolation in between them that has been unattainable so far. In this context, this thesis defines the specific demands an integrated execution has to meet in the domain of industrial automation. However, one precondition to this scenario is the use of an appropriate consolidation solution. Thus, two representative solutions for the domain of embedded systems are presented in terms of a virtualized and a hybrid consolidation approach, before being finally evaluated with regard to the previously defined criteria.

As CPU clock rates have reached physical boundaries, significant future performance gains in the domain of control technology will only be achieved by the application of multi-core CPUs. As a precondition, the firmware has to exploit the parallelism of this processor architecture in an appropriate manner. Unfortunately, for a sophisticated system like an automation firmware, a parallelization commonly induces significant efforts. Thus, decisions in this regard should only be made on the basis of an objective consideration of potential alternatives. However, an estimation of a specific parallel firmware design's prospective performance is challenging due to the system's complexity. This is particularly true, as a parallelization is required that fits a variety of load scenarios in terms of the machines being controlled. Thus, this thesis specifies an application-oriented method that supports the design decisions to be taken when migrating an existing single-core firmware to a homogeneous multi-core architecture. This is achieved by automatically building adequate firmware models based on dynamic firmware profiling under multiple representative load scenarios. These models are then enhanced by the firmware developers' expert knowledge before multi-objective genetic algorithms are applied for exploring the design space of parallelization alternatives. Finally, a specific solution from the retrieved Pareto front can be selected on basis of its evaluation metrics for an implementation by a developer. This thesis concludes with a case study that applies the aforementioned method to a numerical control firmware and thereby reveals its potential of supporting a firmware parallelization in a comprehensive way.

Kurzfassung

Der Einsatz von Multicore-Prozessoren in der industriellen Steuerungstechnik birgt sowohl Chancen als auch Risiken. Die vorliegende Dissertation entwickelt und bewertet aus diesem Grund generische Strategien zur Nutzung dieser Prozessorarchitektur unter Berücksichtigung der spezifischen Rahmenbedingungen und Anforderungen dieser Domäne.

Multicore-Prozessoren bieten die Chance zur Konsolidierung derzeit auf dedizierter Hardware ausgeführter heterogener Steuerungssubsysteme unter einer bisher nicht erreichbaren temporalen Isolation. In diesem Kontext definiert die vorliegende Dissertation die spezifischen Anforderungen, die eine integrierte Ausführung in der Domäne der industriellen Automatisierung erfüllen muss. Eine Vorbedingung für ein derartiges Szenario stellt allerdings der Einsatz einer geeigneten Konsolidierungslösung dar. Mit einem virtualisierten und einem hybriden Konsolidierungsansatz werden deshalb zwei repräsentative Lösungen für die Domäne eingebetteter Systeme vorgestellt, die schließlich hinsichtlich der zuvor definierten Kriterien evaluiert werden.

Da die Taktraten von Prozessoren physikalische Grenzen erreicht haben, werden sich in der Steuerungstechnik signifikante Performanzsteigerungen zukünftig nur durch den Einsatz von Multicore-Prozessoren erzielen lassen. Dies hat zur Vorbedingung, dass die Firmware die Parallelität dieser Prozessorarchitektur in geeigneter Weise zu nutzen vermag. Leider entstehen bei der Parallelisierung eines komplexen Systems wie einer Automatisierungs-Firmware im Allgemeinen signifikante Aufwände. Infolgedessen sollten diesbezügliche Entscheidungen nur auf Basis einer objektiven Abwägung potentieller Alternativen getroffen werden. Allerdings macht die Systemkomplexität eine Abschätzung der durch eine spezifische parallele Firmware-Architektur zu erwartenden Performanz zu einer anspruchsvollen Aufgabe. Dies gilt vor allem, da eine Parallelisierung gefordert wird, die für eine Vielzahl von Lastszenarien in Form gesteuerter Maschinen geeignet ist. Aus diesem Grund spezifiziert die vorliegende Dissertation eine anwendungsorientierte Methode zur Unterstützung von Entwurfsentscheidungen, die bei der Migration einer bestehenden Singlecore-Firmware auf eine homogene Multicore-Architektur zu treffen sind. Dies wird erreicht, indem in automatisierter Weise geeignete Firmware-Modelle auf Basis von dynamischem Profiling der Firmware unter mehreren repräsentativen Lastszenarien erstellt werden. Im Anschluss daran werden diese Modelle um das Expertenwissen von Firmware-Entwicklern erweitert, bevor mittels multikriterieller genetischer Algorithmen der Entwurfsraum der Parallelisierungsalternativen exploriert wird. Schließlich kann eine spezifische Lösung der auf diese Weise hergeleiteten Pareto-Front auf Basis ihrer Bewertungsmetriken zur Implementierung durch einen Entwickler ausgewählt werden. Die vorliegende Arbeit schließt mit einer Fallstudie, welche die zuvor beschriebene Methode auf eine numerische Steuerungs-Firmware anwendet und dabei deren Potential für eine umfassende Unterstützung einer Firmware-Parallelisierung aufzeigt.

Inhaltsverzeichnis

Vorwort v

Abstract vii

Kurzfassung ix

1 Einleitung 1
- 1.1 Multicore-Architekturen in der Steuerungstechnik 1
- 1.2 Einordnung der Arbeit 2
- 1.3 Aufbau der Arbeit 4
- 1.4 Danksagung 5

2 Grundlagen 7
- 2.1 Parallelität von Software 7
- 2.2 Multicore-Prozessoren 8
- 2.3 Steuerungstechnik 11
  - 2.3.1 Topologische Struktur einer Steuerung 11
  - 2.3.2 Firmware-Architektur einer Steuerung 14
- 2.4 Betriebssysteme 15
  - 2.4.1 Basismechanismen 16
  - 2.4.2 SMP-Betriebssysteme 17
- 2.5 Parallele Programmierung 18
  - 2.5.1 Entwicklung paralleler Software 18
  - 2.5.2 Paradigmen paralleler Programmierung 20
- 2.6 Modellierung 21
- 2.7 Software-Analyse 22
- 2.8 Skalare und multikriterielle Optimierung 23
- 2.9 Genetische Algorithmen 24
- 2.10 Systemvirtualisierung 27
  - 2.10.1 Definition und formale Betrachtung 27
  - 2.10.2 Strategien der Systemvirtualisierung auf x86-Architekturen 29

3 Strategien der Multicore-Nutzung in der Steuerungstechnik 33
- 3.1 Performanzsteigerung 33
  - 3.1.1 Motivation einer Performanzsteigerung 33
  - 3.1.2 Firmware-Parallelisierung in der Steuerungstechnik 35
- 3.2 Systemkonsolidierung 38

4 Bewertung der Systemkonsolidierung in der Steuerungstechnik 41
4.1 Ableitung und Abgrenzung der Vorgehensweise . . . 41
4.2 Anforderungen an eine Systemkonsolidierung . . . 41
4.3 Strategien zur Systemkonsolidierung . . . 44
4.3.1 Virtualisierter Konsolidierungsansatz . . . 45
4.3.2 Hybrider Konsolidierungsansatz . . . 46
4.4 Evaluation . . . 48
4.4.1 Reaktivität . . . 49
4.4.2 Zeitdeterminismus . . . 53
4.4.3 Zuverlässigkeit und Sicherheit . . . 56

5 Entwicklung einer Methode zur Firmware-Parallelisierung 59
5.1 Ableitung und Abgrenzung der Vorgehensweise . . . 59
5.1.1 Motivation und Zieldefinition . . . 59
5.1.2 Vorgehensmodell der Modellierung . . . 62
5.2 Verwandte Arbeiten und Stand der Technik . . . 65
5.3 Die *EEEPA*-Toolchain . . . 74
5.3.1 Architektur . . . 74
5.3.2 Profiler-Integration . . . 75
5.3.3 *GraphML*-Erweiterung . . . 77
5.3.4 *PISA*-Konformität . . . 78
5.4 Modellierung auf Systemebene . . . 80
5.4.1 Bewertungskriterien . . . 81
5.4.2 Basismodell . . . 84
5.4.3 Systeminstrumentierung . . . 85
5.4.4 Modellausprägung . . . 88
5.4.5 Validierung . . . 96
5.5 Modellierung auf Taskebene . . . 99
5.5.1 Bewertungskriterien . . . 99
5.5.2 Basismodell . . . 101
5.5.3 Systeminstrumentierung . . . 110
5.5.4 Modellausprägung . . . 113
5.5.5 Validierung . . . 120
5.6 Entwurfsraumexploration mittels genetischer Algorithmen . . . 123
5.6.1 Modellübergreifende Vorgehensweise . . . 123
5.6.2 Entwurfsraumexploration auf Systemebene . . . 128
5.6.3 Entwurfsraumexploration auf Taskebene . . . 131
5.7 Ableitung der Implementierung . . . 140
5.7.1 Parallelisierung auf Systemebene . . . 140
5.7.2 Parallelisierung auf Taskebene . . . 140
5.7.3 Kombinierte Anwendung der Methode . . . 141

6 Fallstudie **145**
6.1 Gegenstand der Fallstudie 145
6.2 Einsatz der *EEEPA*-Toolchain 146
6.2.1 Parametrierung und Konfiguration 146
6.2.2 Modellierung und Optimierung auf Systemebene 148
6.2.3 Modellierung und Optimierung auf Taskebene 156

7 Zusammenfassung und Ausblick **161**
7.1 Beitrag zum Stand der Forschung und Technik 161
7.2 Bewertung der Ergebnisse 163
7.3 Ausblick 164

Literaturverzeichnis **167**

Stichwortverzeichnis **183**

Kapitel 1
Einleitung

Dem zunehmenden internationalen Wettbewerbsdruck versuchen Unternehmen in nahezu allen Branchen unter anderem mit einer kontinuierlichen Produktivitätssteigerung in der Fertigung zu begegnen. Die dabei geforderte Reduzierung der Stückkosten bei einer zugleich konstant hohen Produktqualität lässt sich dabei oft nur mittels einer umfassenden Automatisierung industrieller Prozesse erreichen. Der Begriff der Automatisierung bezeichnet dabei den Einsatz technischer Mittel, mittels derer ein Prozess völlig oder teilweise selbsttätig gemäß einem vorgegebenen Programm abläuft [181]. Eine wesentliche Voraussetzung für eine automatisierte Fertigung bilden Maschinen, die nach heutigem Stand der Technik mit Sensoren (Messgliedern), Aktoren (Stellgliedern) und einer signalverarbeitenden Steuerung ausgerüstet sind. Sensorsignale liefern dabei die erforderlichen Informationen über den Zustand der Maschine oder des Prozesses, während unter anderem elektrisch, hydraulisch oder pneumatisch ausgeführte Aktoren die entsprechenden Verarbeitungsschritte des Prozesses ausführen. Die grundlegenden Aufgaben einer industriellen Steuerung sind schließlich die Auswertung der Sensoren und die Ansteuerung der Aktoren entsprechend dem Programm, das den zu automatisierenden Prozess beschreibt [141, 181].

1.1 Multicore-Architekturen in der Steuerungstechnik

In den Anfängen der Automatisierung wurde eine Maschinensteuerung in vollständig mechanischer Weise realisiert, so dass die meist komplexen Arbeitsabläufe ausschließlich mittels mechanischer Komponenten wie Nocken und Hebeln gesteuert wurden. In elektrischen Steuerungen wurden diese Bauteile schließlich durch elektrische Schalter ersetzt, deren Signale mittels Relais verarbeitet wurden [181]. Heute wiederum stellen insbesondere in komplexen Produktionsanlagen elektronische Steuerungen auf Basis integrierter Schaltkreise den Stand der Technik dar. Anwendungsspezifische Komponenten wie *ASICs (Application Specific Integrated Circuits)* oder *FPGAs (Field Programmable Gate Arrays)* werden dabei aber oft nur noch für ausgewählte Aufgaben genutzt. Stattdessen erfolgt die Maschinensteuerung zunehmend auf Basis einer automatisierungstechnischen Firmware, die auf einem Mikroprozessor ausgeführt wird. Dabei werden immer häufiger Standard-Mikroprozessoren eingesetzt, die in gleicher Weise beispielsweise in Personalcomputern zum Einsatz kommen. Diese zeichnen sich vor allem durch geringe Kosten bei einer inzwischen für industrielle Steuerungsaufgaben ausreichenden Leistungsfähigkeit aus [181]. Dazu hat nicht zuletzt die kontinuierliche Leistungssteigerung beigetragen, die im Bereich der Prozessortechnik in den vergangenen Jahrzehnten erzielt wurde.

Zurückführen lässt sich diese Leistungssteigerung vor allem auf einen durchschnittlichen jährlichen Zuwachs der Taktraten von Prozessoren um annähernd 15 % in den Jahren von 1978 bis 1986 und sogar um knapp 40 % in den Jahren von 1986 bis 2003 [74]. Allerdings ist die Fortführung dieser Entwicklung mit zahlreichen Herausforderungen verbunden [146]. So kann die aus der zunehmenden Leistungsaufnahme resultierende Abwärme von Transistoren bei höheren Taktraten nur noch mit sehr hohem Aufwand abgeführt werden. Gleichzeitig stellen Zugriffe auf den Hauptspeicher bei einer weiteren Steigerung der Taktraten zunehmend einen Flaschenhals für die Systemleistung dar. So benötigt ein einzelner Hauptspeicherzugriff auf einer CPU des Typs *Intel Pentium* aus dem Jahr 2006 bereits 220 Taktzyklen im Gegensatz zu 6 bis 8 Zyklen auf der Architektur *Intel i486* aus dem Jahr 1990. Darüber hinaus wird für die Übertragungsgeschwindigkeit von Signalen gemeinhin die Lichtgeschwindigkeit als physikalisch limitierender Faktor angenommen. Dies führt dazu, dass im Fall einer hypothetischen CPU-Taktrate von 30 GHz die von einem Signal innerhalb eines Taktzyklus zurückgelegte Strecke von 1 cm die Abmessungen heutiger Prozessoren erreicht. Diese Gründe haben dazu geführt, dass diese Strategie der Leistungssteigerung nicht mehr weiter verfolgt werden konnte und die Taktraten seit dem Jahr 2003 weitestgehend stagnieren [74] (vgl. Abbildung 1.1).

Eine weitere Methode zur Leistungssteigerung war die Integration größerer Cache-Speicher und duplizierter Funktionseinheiten in die Chips. Dies wurde möglich, da für die Integrationsdichte der Chips auch heute noch das 1965 von Gordon E. Moore formulierte empirische Gesetz *(Moore's Law)* gilt, welches ungefähr alle 18 bis 24 Monate eine Verdopplung der Transistorzahl pro Flächeneinheit prognostiziert [121]. Während Caches auch in Zukunft wachsen werden [164], ist die durch eine weitere Duplikation von Funktionseinheiten erzielbare Leistungssteigerung gering, da sequentielle Programme in der Regel die dafür erforderliche Parallelität auf Instruktionsebene nur in begrenztem Umfang aufweisen. Deshalb nutzen die CPU-Hersteller ungefähr seit dem Jahr 2005 [164] die zusätzlich verfügbaren Transistoren für die Integration mehrerer weitestgehend unabhängiger Verarbeitungseinheiten, sogenannter *CPU-Kerne*, auf einem Chip. Derartige Prozessoren werden als *Multicore-* respektive *Mehrkernprozessoren* oder *Single-Chip Multiprocessors (CMPs)* bezeichnet. Nachdem diese Architekturen in den vergangenen Jahren den Markt der Personalcomputer nahezu vollständig durchdrungen haben, wächst deren Verbreitung auch im Bereich eingebetteter und echtzeitfähiger Systeme kontinuierlich. So kommen sowohl bei paketverarbeitenden Systemen der Netzwerktechnik als auch bei mobilen Endgeräten in Form von Smartphones oder Tablet-PCs bereits seit einigen Jahren Multicore-Prozessoren zum Einsatz [122]. Während damit der Bereich der Kommunikationstechnik zu den Vorreitern zählt, werden inzwischen auch in vielen anderen Domänen eingebetteter Systeme Konzepte zur Nutzung dieser Prozessorarchitektur entwickelt und evaluiert. Dazu zählen unter anderem die Automobilelektronik [75, 120], die Avionik [135] und nicht zuletzt die industrielle Steuerungstechnik [38, 150].

1.2 Einordnung der Arbeit

Für eine effiziente Nutzung von Multicore-Prozessoren besteht die Erfordernis, für die einzelnen CPU-Kerne eine ausreichende Anzahl möglichst unabhängiger Ausführungskontexte bereitzustellen. Dies lässt sich auf unterschiedliche Arten erreichen [50]. So können den einzelnen CPU-Kernen im Rahmen einer *Konsolidierung* der eingesetzten Hardware unterschiedliche

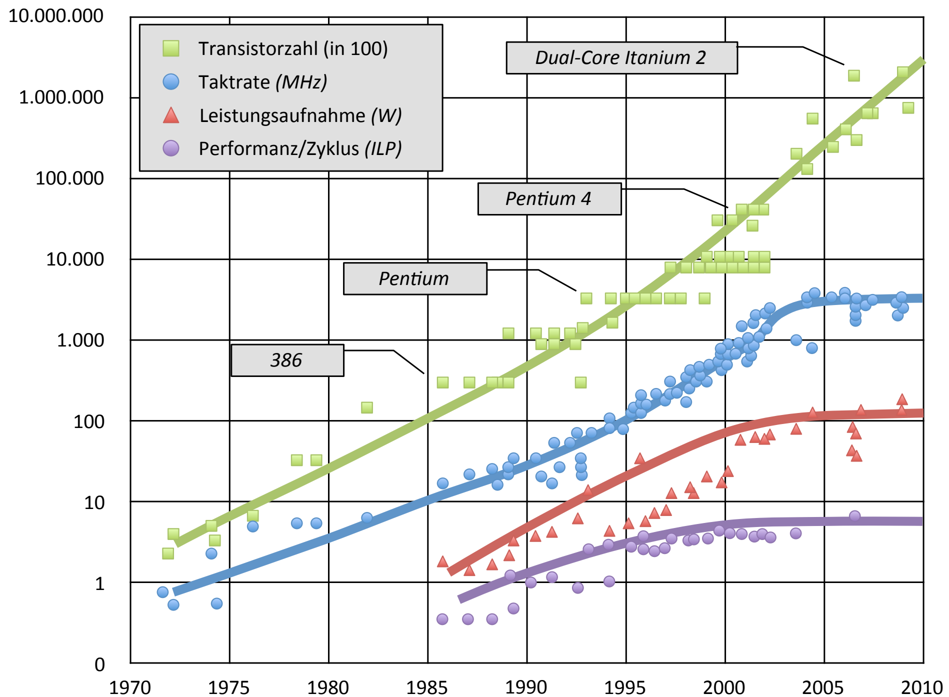


Abbildung 1.1: Entwicklung von CPU-Kenndaten am Beispiel von *Intel* (nach [165])

Betriebssysteme mit jeweils eigenen Applikationen zugewiesen werden. Eine derartige Systemkonsolidierung erfordert eine geeignete Abstraktionsschicht [20], wie sie beispielsweise durch eine *Systemvirtualisierung* realisiert wird. Alternativ dazu können auf den CPU-Kernen unter einem Betriebssystem mehrere, meist unabhängige Applikationen ausgeführt werden. Diese Strategie besitzt allerdings im Bereich eingebetteter Systeme meist weniger Relevanz, da diese in der Regel auf die Ausführung einer einzigen Applikation beschränkt sind. Schließlich besteht die Möglichkeit, eine Applikation in nebenläufige Ausführungsfäden zu partitionieren, die wiederum unterschiedlichen CPU-Kernen zugeordnet werden. Dies eröffnet das Potential, die Ausführungsdauer der zuvor sequentiell implementierten Applikation durch eine parallele Abarbeitung zu reduzieren. Zugleich stellt diese Parallelverarbeitung für das zu erwartende Szenario einer weiter anhaltenden Stagnation bei den CPU-Taktraten die einzige Möglichkeit dar, auch zukünftig signifikante Performanzsteigerungen einzelner Applikationen zu erzielen. Daraus resultiert jedoch eine Zäsur in der Software-Entwicklung, der ein tiefgreifender Paradigmenwechsel folgen muss. So wird die Leistungssteigerung einer Applikation beim Wechsel auf eine leistungsfähigere CPU-Architektur zukünftig nicht mehr allein durch die Hardware, sondern zu einem wesentlichen Teil auch durch die Applikation selbst bestimmt [164].

Als Konsequenz dieser Nutzungsszenarien gestaltet sich die aktuelle Situation der Steuerungstechnik im Hinblick auf Multicore-Prozessoren ambivalent. So eröffnen sich im Kontext der Systemkonsolidierung zahlreiche Chancen für zukünftige Steuerungsarchitekturen, mittels de-

rer Hersteller Alleinstellungsmerkmale gegenüber Wettbewerbern entwickeln können. Zugleich bergen Multicore-Prozessoren im Kontext der Performanzsteigerung für die Steuerungshersteller Risiken. Können die Firmware-Architekturen zukünftige Leistungssteigerungen der eingesetzten Hardware in Form einer zunehmenden Parallelität nicht oder nicht in dem erwarteten Umfang nutzen, droht langfristig der Verlust der Wettbewerbsfähigkeit. Dieser Herausforderung müssen sich branchen- und herstellerübergreifend alle industriellen Steuerungslösungen stellen. Somit gilt es, unter Berücksichtigung der spezifischen Rahmenbedingungen und Anforderungen im Bereich der Steuerungstechnik generische Strategien zu entwickeln und zu bewerten, um Multicore-Architekturen zukünftig auch in dieser Domäne in effizienter Weise nutzen zu können. Dieser Herausforderung widmet sich die vorliegende Arbeit.

1.3 Aufbau der Arbeit

Mit der Systemkonsolidierung und der Performanzsteigerung wurden bereits die beiden wesentlichen Strategien zur Nutzung von Multicore-Architekturen in der Steuerungstechnik genannt. Wenngleich beide Strategien im Anwendungsfall der Steuerungstechnik aufeinandertreffen, so sind diese im wissenschaftlichen Kontext weitestgehend disjunkten Forschungsbereichen zuzuordnen. Dies spiegelt der nachfolgend beschriebene Aufbau dieser Arbeit wider.

Im Anschluss an diese Einleitung werden in Kapitel 2 zunächst die theoretischen Zusammenhänge dargelegt, welche als Grundlagen für das weitere Verständnis dieser Arbeit maßgeblich sind. Dabei werden Einblicke in die Architektur von Multicore-Prozessoren ebenso gegeben wie in die eingangs nur knapp umrissene Domäne der Steuerungstechnik. Weiterhin erfolgt eine Beschreibung grundlegender Konzepte im Bereich von Betriebssystemen und der Entwicklung paralleler Software, bevor eine Einführung in die Themen Systemmodellierung und Software-Analyse sowie in die algorithmische Lösung von Optimierungsproblemen geliefert wird. Dem schließt sich eine formale Betrachtung der Systemvirtualisierung an, infolge derer verbreitete Strategien zur Realisierung einer solchen vorgestellt werden.

Kapitel 3 diskutiert schließlich die Performanzsteigerung und die Systemkonsolidierung als Strategien der Multicore-Nutzung. In beiden Fällen wird zunächst die jeweilige Motivation in Form diverser Nutzungsszenarien in der Domäne der Steuerungstechnik konkretisiert. Darüber hinaus wird ein zur Performanzsteigerung einer automatisierungstechnischen Firmware empfohlenes Vorgehen ebenso abgeleitet wie die für eine Systemkonsolidierung erforderlichen Voraussetzungen.

Der Systemkonsolidierung in der Steuerungstechnik widmet sich Kapitel 4 und nennt zunächst die Anforderungen, die in dieser Domäne an eine zu diesem Zweck eingesetzte Konsolidierungslösung gestellt werden. Darüber hinaus werden zwei alternative Konsolidierungsansätze am Beispiel kommerziell verfügbarer Produkte vorgestellt und im Anschluss daran hinsichtlich der zuvor definierten Anforderungen bewertet.

Kern der Arbeit bildet Kapitel 5, das die Entwicklung einer anwendungsorientierten Methode zum Gegenstand hat, welche die Parallelisierung einer automatisierungstechnischen Firmware für homogene Multicore-Prozessoren in umfassender Weise unterstützt. Zu diesem Zweck werden auf Basis von Laufzeitinformationen Modelle einer Firmware generiert, mittels derer sich effiziente Alternativen der Parallelisierung auf verschiedenen Ebenen der Implementierung explorieren und evaluieren lassen. Nach einer Vorstellung verwandter Arbeiten wird zunächst

eine im Rahmen der vorliegenden Arbeit entwickelte *Toolchain* beschrieben, welche diesen Ansatz in prototypischer Weise implementiert. Schließlich erfolgt eine detaillierte Spezifikation der zuvor skizzierten Methode, bevor in Kapitel 6 deren Anwendung auf eine konkrete Steuerungs-Firmware im Rahmen einer Fallstudie beschrieben wird. Dabei werden im Anschluss an eine Darstellung der entsprechenden Rahmenbedingungen und der beim Einsatz der Methode gewählten Vorgehensweise die erzielten Ergebnisse präsentiert.

Kapitel 7 beschließt die Arbeit mit einer Zusammenfassung und diskutiert in einem Ausblick noch offene Aspekte der Problemstellung, die in zukünftigen Arbeiten zu thematisieren sind.

1.4 Danksagung

Die vorliegende Dissertation entstand im Rahmen meiner Teilnahme am Doktorandenprogramm der *Robert Bosch GmbH* in der Entwicklungsabteilung für Werkzeugmaschinensteuerungen der *Bosch Rexroth AG*.

Zuallererst möchte ich deshalb meinem Doktorvater, Prof. Dr. Reiner Kolla, dafür danken, dass er meine Betreuung als externer Doktorand bereitwillig übernahm. Er hatte stets ein offenes Ohr für meine Fragen und ließ mich in zahlreichen Diskussionen an seinem umfangreichen Erfahrungsschatz teilhaben.

Mein Dank gebührt auch Prof. Dr. Ludwig Leurs, der sich ohne Zögern bereit erklärte, das Zweitgutachten zu dieser Arbeit anzufertigen.

Ebenso gilt mein Dank den Kolleginnen und Kollegen der *Bosch Rexroth AG*, die mir stets mit Rat und Tat zur Seite standen. Hervorzuheben sind dabei Dr. Thomas Schröder, Dr. Jens Brühl, Günther Landgraf und Dr. Thomas Bürger, deren Anregungen dieser Arbeit nicht zuletzt zu ihrer thematischen Vielfalt verhalfen. Danken möchte ich auch den Kollegen Wolfgang Rochau, Ralf Schäfer und Achim Kreutz, auf deren Unterstützung ich stets zählen konnte. Mein Dank gilt darüber hinaus den Kollegen der Vorausentwicklung softwareintensiver Systeme der *Robert Bosch GmbH* in Schwieberdingen, die meiner Arbeit regelmäßig neue Impulse gaben.

Weiterhin danke ich Frank Adämmer und Julian Hartmann, die mich im Rahmen ihrer Diplom- respektive Bachelorarbeit bei der Implementierung meiner Ideen unterstützten und mir dabei stets wichtige Diskussionspartner waren.

Zum Dank bin ich auch meinen Mitstreitern vom Lehrstuhl für Technische Informatik der Universität Würzburg für die zahllosen Gespräche verpflichtet. Besonders hervorheben möchte ich dabei Christian Appold und Dr. Marcel Baunach, die sich trotz eigener Verpflichtungen die Zeit für ein Korrektorat dieser Arbeit nahmen.

Ferner gebührt mein Dank auch meinen Eltern, die mein Promotionsvorhaben stets mit Interesse begleiteten und mich dabei unterstützten.

Mein abschließender und besonderer Dank gilt Sanne, die stets Verständnis für all die Stunden hatte, die ich insbesondere in den vergangenen drei Jahren berufsbegleitend in diese Arbeit investiert habe. Du motiviertest mich immer wieder neu, ließt mich nie zweifeln und gabst mir stets den erforderlichen Rückhalt, ohne den ich diese Arbeit nicht hätte schreiben können.

Kapitel 2
Grundlagen

2.1 Parallelität von Software

Die Ausführung eines Programms kann auf mehreren Ebenen respektive Granularitätsstufen in paralleler Weise erfolgen, wenn entsprechende Verarbeitungsressourcen zur Verfügung stehen. Bezüglich der dabei erzielten Parallelität ist folgende Klassifikation üblich [63]:

- Eine *Parallelität auf Instruktionsebene (Instruction-Level Parallelism, ILP)* basiert auf der Unabhängigkeit aufeinanderfolgender Instruktionen eines sequentiellen Programms, wodurch deren parallele Ausführung möglich wird. So definieren *VLIW-Architekturen* sogenannte *lange Instruktionen (Very Long Instruction Words, VLIWs)*, die durch einen Compiler in automatisierter Weise mittels einer Kombination mehrerer einfacher Instruktionen generiert werden [115]. Techniken wie das *Pipelining* realisieren hingegen eine durch die CPU zur Laufzeit gesteuerte überlappende Ausführung der Verarbeitungsschritte aufeinanderfolgender Instruktionen. *Superskalare Prozessoren* vervielfältigen darüber hinaus ausgewählte Funktionseinheiten und können auf diese Weise pro Taktzyklus mehrere Instruktionen echt parallel ausführen.
- Eine *Parallelität auf Datenebene (Data-Level Parallelism, DLP)* liegt vor, wenn eine Instruktion zeitgleich auf mehreren Bits eines Datenfelds oder auf mehreren Datenfeldern ausgeführt wird [61]. Dieser Ansatz findet heute vor allem in Grafikprozessoren *(Graphics Processing Units, GPUs)* oder in Form von Befehlssatzerweiterungen für Standard-CPUs wie beispielsweise den *Streaming SIMD Extensions (SSE)* von *Intel* Verwendung.
- Eine *Parallelität auf Task-* oder *Threadebene (Thread-/Task-Level Parallelism, TLP)* lässt sich auf Basis nebenläufiger Kontrollflüsse in Form von *Prozessen*, *Tasks* oder *Threads* erzielen[1]. Zwei Ausführungsfäden sind genau dann vollständig nebenläufig, wenn jede Anweisung eines Ausführungsfadens nebenläufig zu jeder Anweisung des anderen Ausführungsfadens ist. Für zwei Anweisungen gilt wiederum genau dann die Eigenschaft der Nebenläufigkeit, wenn zwischen diesen keine kausale Relation besteht und diese somit unabhängig voneinander abgearbeitet werden können. In der Realität wird allerdings die Nebenläufigkeit zweier Ausführungsfäden und somit auch die Parallelität auf Taskebene häufig durch Synchronisationen eingeschränkt [107]. Wird zudem aufgrund einer

[1] In Systemen mit einer virtuellen Speicherverwaltung bezeichnet ein Prozess oder eine Task in der Regel eine in einem separaten Adressraum ausgeführte Programmlogik mit gegebenenfalls multiplen Kontrollflüssen in Form sogenannter *leichtgewichtiger Threads*. In eingebetteten Systemen wird hingegen meist auf eine virtuelle Speicherverwaltung verzichtet und somit eine synonyme Verwendung dieser Termini gepflegt. Diesem Duktus folgt auch die vorliegende Arbeit.

mangelnden Ressourcenverfügbarkeit die sequentielle Ausführung nebenläufiger Ausführungsfäden mittels *Software Multithreading* oder *Hardware Multithreading* erzwungen, so wird dies als *Quasiparallelität* bezeichnet. Wenngleich die Begriffe der Parallelität und Nebenläufigkeit häufig synonym verwendet werden, so stellt folglich die Parallelität eine spezielle Form der Nebenläufigkeit dar, die sich nur bei einer entsprechenden Ressourcenverfügbarkeit realisieren lässt [77].

Darüber hinaus existieren Techniken, die mehrere der zuvor genannten Arten der Parallelität kombinieren. So nutzt das als Variante des Hardware Multithreading klassifizierte *Simultaneous Multithreading (SMT)*, das in der Realisierung durch *Intel* als *Hyper-Threading* bezeichnet wird [113], gegebenenfalls vorhandene Parallelität auf Instruktions- und Taskebene zugleich aus. Dabei werden zur Laufzeit innerhalb des gleichen Taktzyklus Instruktionen unterschiedlicher Ausführungsfäden in echt paralleler Weise auf den Funktionseinheiten einer superskalaren CPU ausgeführt [138].

2.2 Multicore-Prozessoren

Allen Multicore-Prozessoren ist gemein, dass sie als ein auf einem einzigen Chip realisiertes Multiprozessorsystem angesehen werden können. Dabei ist jeder CPU-Kern unabhängig von den übrigen Kernen in der Lage, Instruktionen und Daten zu lesen, zu verarbeiten und zu schreiben. Dies hat zur Konsequenz, dass ausgewählte Ressourcen, wie in Abbildung 2.1 am Beispiel der *Intel Core* Mikroarchitektur dargestellt, für jeden Kern dediziert vorhanden sind. Aus diesem Grund ist zur effizienten Nutzung von Multicore-Prozessoren auch eine ausreichende Parallelität auf Taskebene (TLP) erforderlich; die entsprechenden Implikationen für die Software diskutierte bereits Abschnitt 1.2. Unabhängig davon werden häufig die in Abschnitt 2.1 beschriebenen Mechanismen zur Ausnutzung gegebenenfalls vorhandener Parallelität auf Daten- und Instruktionsebene weiterhin unterstützt. Im Fall des Simultaneous Multithreading stellt sich beispielsweise eine CPU mit zwei physikalischen Kernen dem Betriebssystem als logische Quad-Core-CPU dar. Weiterhin sind vor allem im Server-Bereich Multicore-Prozessoren mit VLIW-Befehlssatz verfügbar, wie beispielsweise der *Intel Itanium 2*.

Generell lassen sich zwei Typen von Multicore-Architekturen unterscheiden: Im Desktop- und Server-Bereich kommen in der Regel *homogene Multicore-Prozessoren* zum Einsatz, bei denen die Kerne aufgrund einer äquivalenten *Befehlssatzarchitektur (Instruction Set Architecture, ISA)* in identischer Weise programmiert werden können, so dass es aus funktionaler Sicht keinen Unterschied macht, auf welchem CPU-Kern ein Programm ausgeführt wird [174]. Weiterhin sind homogene Multicore-Prozessoren in der Regel speichergekoppelt mit einem gemeinsamen globalen Adressraum, so dass der Speicher sowie die Systemperipherie allen Kernen in gleicher Weise zur Verfügung steht. In dieser Domäne besitzen vor allem die Hersteller *Intel* und *AMD* große Marktanteile. Im Embedded-Bereich hingegen finden vor allem *heterogene Multicore-Prozessoren* mit aufgabenspezifischen Kernen, bei denen Systemressourcen häufig nur exklusiv zugeordnet sind, zunehmend Verbreitung [36]. Die Kerne können dabei identische, überlappende oder disjunkte Befehlssatzarchitekturen aufweisen [174]. Der letztere Fall gilt beispielsweise für den vor allem in sicherheitskritischen *Automotive*-Applikationen eingesetzten *Infineon TriCore TC1797*, der einen *RISC*-Prozessor und eine weniger leistungsfähige CPU zur Peripheriesteuerung integriert [82].

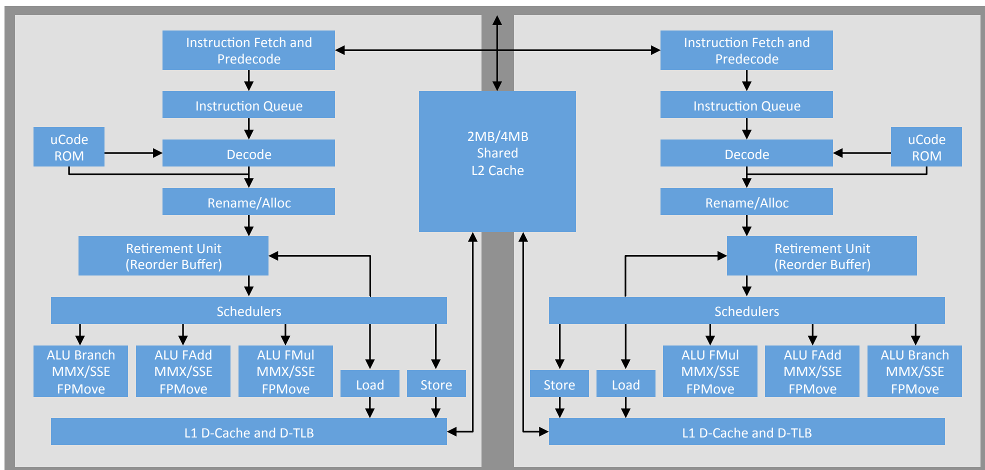


Abbildung 2.1: Architektur eines Multicore-Prozessors am Beispiel der *Intel Core* Mikroarchitektur nach [114]

Zur Anbindung des gemeinsamen Hauptspeichers in homogenen Multicore-Architekturen gibt es vielfältige Konzepte [146], wobei das *hierarchische Design* bei homogenen CMPs mit einer geringen Anzahl an CPU-Kernen zum aktuellen Zeitpunkt die größte Verbreitung besitzt. Ähnlich wie bei Prozessoren mit nur einer Verarbeitungseinheit erfolgen beim hierarchischen Design Zugriffe auf den Hauptspeicher über eine Hierarchie kleiner, aber schneller Zwischenspeicher, die sogenannten *Caches*. Diese erzeugen die Illusion eines einzelnen Speichers in der Dimension des preiswerten Hauptspeichers, der aber zugleich die Geschwindigkeit des kleinen und teuren Level-1-Cache (L1-Cache) zu besitzen scheint [138]. Bezüglich der konkreten Ausgestaltung der Cache-Hierarchie in Multicore-CPUs gibt es verschiedene Varianten, die sich im Wesentlichen in der Anzahl der Ebenen, der Größe der Speicher und der Art der Zuordnung zu den CPU-Kernen unterscheiden. Abbildung 2.2 stellt die Speicherhierarchien ausgewählter Multicore-Prozessoren dar.

Allen diesen Varianten ist gemein, dass es mindestens eine Ebene mit Caches gibt, die jeweils durch einen CPU-Kern exklusiv genutzt werden. Um die Nebenläufigkeit des Systems nicht zu stark einzuschränken, erfolgt dabei allerdings von Seiten der Hardware kein wechselseitiger Ausschluss von Speicherzugriffen unter den CPU-Kernen. Stattdessen können Kopien eines Speicherworts zu einem Zeitpunkt in mehreren Caches der gleichen Ebene enthalten sein. Als Konsequenz müssen allerdings Maßnahmen zur Sicherstellung der Gültigkeit von Speicherinhalten ergriffen werden, so dass zu jedem Zeitpunkt durch beliebige CPU-Kerne stets aktuelle Werte eines Speicherworts gelesen werden [172].

Ein mögliches Verfahren, um Gültigkeitsprobleme zu verhindern, stellt dabei die Sicherstellung der *Cache-Konsistenz* dar, bei der alle im Hauptspeicher und in den Caches vorhandenen Kopien eines Speicherworts identisch gehalten werden. Dies erfordert jedoch ein kontinuierliches Durchschreiben von Änderungen auf Speicher höherer Ebene und kann somit die Systemleistung signifikant beeinträchtigen. Gültigkeitsprobleme der Daten können allerdings auch

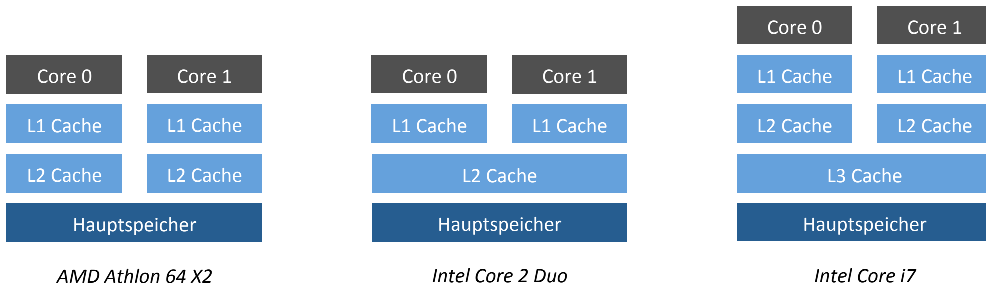


Abbildung 2.2: Speicherhierarchien ausgewählter Multicore-Architekturen [3, 85, 87]

vermieden werden, wenn Inkonsistenzen bis zu einem gewissen Grad zugelassen werden und lediglich die *Cache-Kohärenz* garantiert wird. Diese erfordert, dass bei einem Lesezugriff auf ein Speicherwort immer der Wert des letzten Schreibzugriffs auf eben dieses zurückgeliefert wird. Zu diesem Zweck wird in der Regel das sogenannte *Bus Snooping* angewandt, bei dem die Cache-Logik jedes Cores am gemeinsamen Speicherbus auf Adressen lauscht, deren Inhalte derzeit im eigenen Cache enthalten sind. Dieses Verfahren ist essentieller Bestandteil zahlreicher *Cache-Kohärenzprotokolle* wie *MESI (Modified Exclusive Shared Invalid)*, welches beispielsweise in der *Intel Core* Mikroarchitektur eingesetzt wird [117]. Das Grundprinzip des *MESI*-Protokolls besteht darin, dass eine Kopie eines Speicherworts im Fall eines Schreibzugriffs als modifiziert und alle weiteren Kopien zugleich als invalide gekennzeichnet werden. Im Fall eines Lesezugriffs auf eine invalide Kopie wird nun zunächst die modifizierte Kopie in den gemeinsamen Speicher zurückgeschrieben und dann erneut von dort eingelesen [50]. Weiterentwickelte Varianten des *MESI*-Protokolls sind unter anderem das *MOESI*-Protokoll, das Bestandteil der *AMD64*-Mikroarchitektur ist [4] und das *MESIF*-Protokoll, das von *Intel* in Prozessoren mit der *Nehalem*-Mikroarchitektur eingesetzt wird [86].

Findet zwischenzeitlich infolge einer zu geringen Cache-Dimensionierung keine Verdrängung des Cache-Inhalts statt, beschränkt sich in hierarchischen Multicore-Architekturen der Datenaustausch zwischen Anweisungsfolgen, die auf dem gleichen CPU-Kern ausgeführt werden, auf das Schreiben und Lesen des dem entsprechenden Core exklusiv zugeordneten L1-Cache. Im Gegensatz dazu müssen beim Datenaustausch zwischen Anweisungsfolgen, die auf verschiedenen CPU-Kernen ausgeführt werden, die Daten in den Hauptspeicher oder mindestens in einen langsameren Cache der niedrigsten Stufe, der den beteiligten Kernen gemeinsam zur Verfügung steht, zurückgeschrieben werden, bevor sie vom Empfänger gelesen werden können. Im Fall des *Intel Core 2 Duo* wäre dies mindestens der L2-Cache. Das damit verbundene Kopieren der Daten zwischen den Caches verschiedener Stufen und gegebenenfalls dem Hauptspeicher erhöht dabei allerdings nicht nur die Latenz der Kommunikation, sondern verursacht auch eine signifikante Systemlast in Form von Wartezyklen in der *Instruktions-Pipeline* der CPU *(Pipeline Stalls)* [74]. Ein besonders hoher Performanzverlust ist dabei beim sogenannten *Cache Ping-Pong* festzustellen. In diesem Szenario wird von zwei unterschiedlichen CPU-Kernen zyklisch und wechselseitig auf den gleichen Wert zugegriffen, der infolgedessen kontinuierlich zwischen den jeweils Core-lokalen Caches ausgetauscht werden muss [50]. Zu beachten ist,

dass dieser Effekt auch dann auftritt, wenn beide Cores auf unterschiedliche Werte zugreifen, die aber in der gleichen *Cache-Line* liegen. Dies wird als *False Sharing* bezeichnet [23].

Generell sind Multicore-Architekturen von sogenannten *Manycore*-Prozessoren abzugrenzen, zu denen entsprechend heute üblicher Klassifikationen in der Regel CPUs mit 32 und mehr Kernen gezählt werden. Bei derartigen Prozessoren, wie beispielsweise einem von *Intel* im Rahmen der *Tera-scale Computing* Initiative entwickelten Teraflop-Chip mit 80 CPU-Kernen [73], wird statt einer hierarchischen Speicherarchitektur in der Regel ein sogenanntes *Network-on-Chip (NoC)* mit entsprechendem *Routing* eingesetzt, da dieses die entsprechenden Anforderungen besser erfüllt [146]. Aktuell prognostizieren die Chip-Hersteller für die nächste Dekade alle zwei Jahre eine Verdopplung der Anzahl der CPU-Kerne pro Chip, was auf die Verfügbarkeit von CPUs mit mehreren hundert oder tausend Kernen im Jahr 2020 schließen ließe [174]. Allerdings warnen kritische Stimmen bei derartigen Prozessoren vor einem beträchtlichen, von mehreren Einflussgrößen abhängigen Anteil an sogenanntem *Dark Silicon* in Form von CPU-Kernen, die aufgrund von Beschränkungen in der Leistungsaufnahme des Prozessors und der Parallelität der Anwendungen nicht genutzt werden können [51].

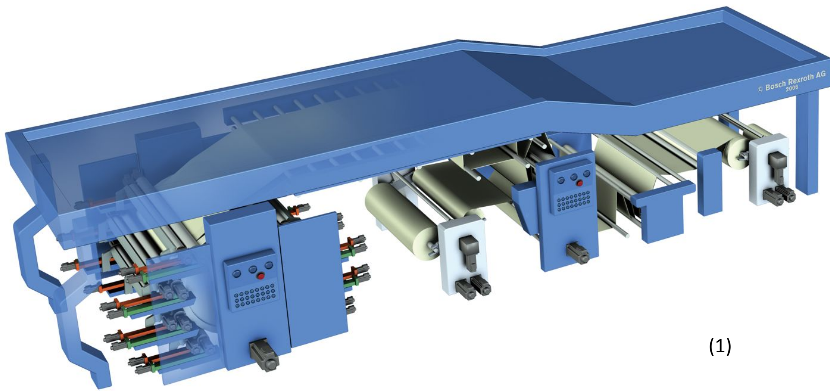
2.3 Steuerungstechnik

Generell lassen sich zwei wesentliche Voraussetzungen für jegliche Art der Automatisierung definieren: Die Mechanisierung der Anlagen und die Steuerungstechnik [181]. Dabei werden unter dem Begriff der Steuerungstechnik alle Aufgaben der Speicherung, Verarbeitung und Übertragung von Daten zusammengefasst, die mit der Ansteuerung von Aktoren einer Maschine in Zusammenhang stehen [181]. Ausgewählte Beispiele für Maschinen und somit typische Einsatzszenarien industrieller Automatisierungstechnik zeigt Abbildung 2.3.

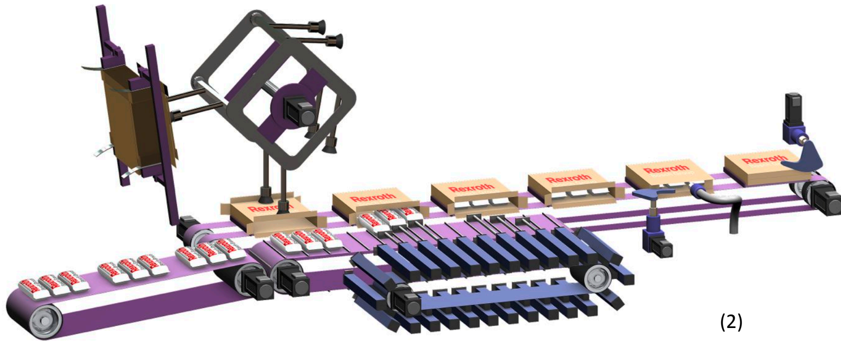
2.3.1 Topologische Struktur einer Steuerung

Eine Maschinensteuerung umfasst üblicherweise verschiedene Funktionsbereiche mit jeweils spezifischen Aufgaben. In Anlehnung an [129] sind dabei entsprechend der Darstellung in Abbildung 2.4 folgende Funktionsbereiche hervorzuheben:

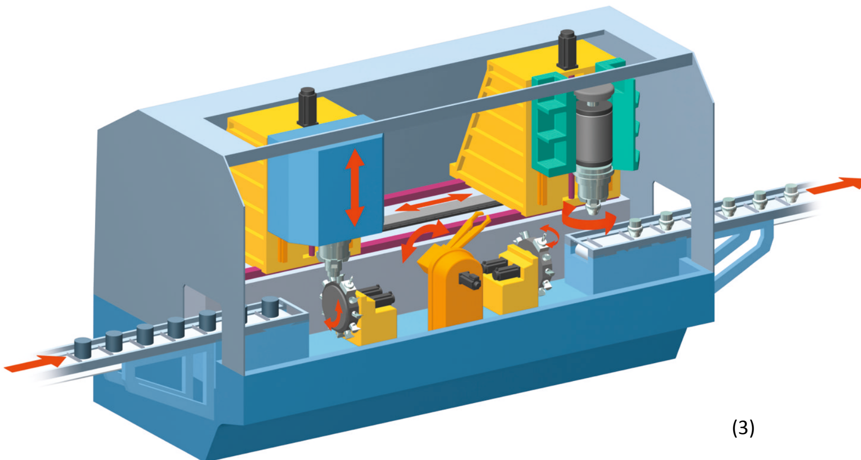
- Während Pumpen und Ventile die Aktoren der hydraulischen und pneumatischen Automatisierung darstellen, nehmen in der elektrischen Automatisierung Motoren unterschiedlicher Bauart diese Aufgabe wahr. Die Ansteuerung der Motoren erfolgt dabei durch sogenannte *Antriebsregler (Drive Controls, DCs)*, welche die Regelung und das für die Spannungsversorgung verantwortliche Leistungsteil integrieren. Üblicherweise wird in Maschinen zur Regelung jedes Motors ein dedizierter Antriebsregler eingesetzt, der inzwischen meist digital realisiert ist und somit über einen Mikroprozessor verfügt. Dies hat dazu geführt, dass Funktionen wie die Grenzwert- und Verfahrbereichsüberwachung zunehmend von der Steuerung in den Antriebsregler verlagert werden [180].
- Als *Verknüpfungssteuerung* wird eine boolesche Verknüpfung von Eingangssignalen zu Ausgangssignalen bezeichnet, während eine *Ablaufsteuerung* eine Schrittkette mit Schaltbedingungen und schrittspezifischen Aktionen realisiert [69, 157, 181]. Moderne Steuerungen realisieren beide Varianten nicht mehr mittels logisch verknüpfter Schaltkreise,



(1)



(2)



(3)

Abbildung 2.3: Anwendungsfelder industrieller Automatisierung: Flexodruckmaschine (1), Verpackungsmaschine (2) und Vertikaldrehmaschine (3) *(Bilder: © Bosch Rexroth AG 2014)*.

sondern durch entsprechende Instruktionssequenzen, die durch eine *speicherprogrammierbare Steuerung (SPS)*, im Englischen als *(Programmable) Logic Control (PLC/LC)* bezeichnet, in zyklischer Weise ausgeführt werden. Richtlinien zur Programmierung einer SPS sind in der Norm *IEC 61131-3* [132] definiert, die von den Programmiersystemen der SPS-Hersteller in einem jeweils zu dokumentierenden Umfang erfüllt werden [90]. Zu den normierten Programmiersprachen zählen dabei unter anderem der an eine Hochsprache angelehnte *Strukturierte Text (ST)* oder der von elektrischen Schaltplänen inspirierte *Kontaktplan (KOP)*. Die Programmierung wird dabei oft durch den Einsatz logischer Programmeinheiten in Form sogenannter *Funktionsbausteine* unterstützt, die vom Steuerungshersteller zur Verfügung gestellt werden.

- Die Aufgabe der *Bewegungssteuerung (Motion Control, MC)* besteht darin, die Verfahrbewegungen der Motoren einer Maschine entsprechend eines vorgegebenen Programms zu steuern. Die Art der Programmierung ist dabei vom jeweiligen Maschinentyp abhängig. Im Fall von *Werkzeugmaschinensteuerungen (Computerized Numerical Controls, CNCs)* wird beispielsweise die Kontur des zu fertigenden Werkstücks durch sogenannte *NC-Programme* beschrieben, während in *Robotersteuerungen (Robot Controls, RCs)* meist herstellerspezifische Sprachen Anwendung finden. Im Bereich der sogenannten *allgemeinen Automatisierung*, wie beispielsweise der Steuerung von Druck- und Verpackungsmaschinen, werden zudem häufig SPS-Funktionsbausteine zur Beschreibung von Achsbewegungen eingesetzt. In der Regel wird zur Steuerung eines Motors eine sogenannte *kaskadierte Regelung*, bestehend aus Strom-, Drehzahl- bzw. Geschwindigkeits- und Lageregler, eingesetzt, wobei diese Regelkreise je nach Ausführung entweder im Antriebsregler oder in der Bewegungssteuerung geschlossen werden [180].
- Die *Benutzerschnittstelle (Human Machine Interface, HMI)* stellt alle Funktionen bereit, die von den verschiedenen Benutzergruppen einer Maschine, wie den Maschinenbedienern oder den Inbetriebnehmern, zur Durchführung ihrer Aufgaben benötigt werden. Dazu zählt neben der Simulation und Implementierung von Bearbeitungsprogrammen auch die Diagnose und Wartung der Maschine [141]. Die HMI basiert heute in der Regel auf der graphischen Benutzeroberfläche eines Standardbetriebssystems wie beispielsweise *Microsoft Windows*.

Diese verschiedenen Funktionsbereiche sind in der Regel nicht zentral, sondern durch eine Vielzahl dezentraler, entsprechend vernetzter Hardware-Komponenten realisiert. Eine mögliche physikalische Topologie einer automatisierungstechnischen Steuerung zeigt Abbildung 2.4 am Beispiel typischer Komponenten der Firma *Bosch Rexroth*. Hier sind sowohl die Bewegungssteuerung als auch die SPS auf einem zur Hutschienenmontage geeigneten Controller kompakter Baugröße integriert. Die HMI ist hingegen in abgesetzter Form als PC mit integriertem Bildschirm realisiert, der über eine Ethernet-Schnittstelle mit der Steuerung kommuniziert. Es gibt allerdings Topologien, bei denen auch die Steuerungsfunktionen in die HMI integriert sind, so dass auf einen dedizierten Controller verzichtet werden kann. Die Anbindung der Antriebsregler und der E/A-Module an die Steuerung erfolgt in beiden Fällen mittels sogenannter *Feldbusse*, die einen zyklischen Datenaustausch zwischen den Teilnehmern unter Echtzeitbedingungen ermöglichen. Diese Forderung impliziert neben der garantierten Übertragung die strikte Einhaltung enger zeitlicher Schranken, die durch die konfigurierte Zykluszeit des Feldbusses vorgegeben werden; typische Werte liegen hier im Bereich weniger Millisekunden oder

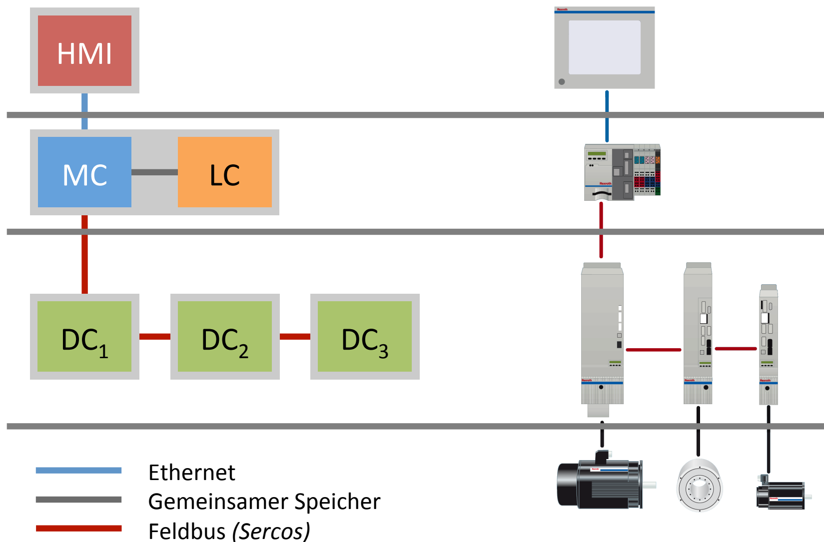


Abbildung 2.4: Logische und physikalische Topologie einer automatisierungstechnischen Steuerung am Beispiel von Komponenten der Firma *Bosch Rexroth*. Zu den Funktionsbereichen zählen die Benutzerschnittstelle (HMI), die Bewegungssteuerung (MC), die speicherprogrammierbare Steuerung (LC) und die Antriebsregler (DC_1, DC_2, DC_3) *(Bilder: © Bosch Rexroth AG 2014)*.

darunter [190]. Im Rahmen der Kommunikation zwischen der Steuerung und den Antriebsreglern werden über den Feldbus je nach Ausführung der kaskadierten Regelung Strom-, Drehzahl- bzw. Geschwindigkeits- oder Lagesollwerte übertragen [180].

2.3.2 Firmware-Architektur einer Steuerung

Zu den wesentlichen Herausforderungen einer automatisierungstechnischen Steuerungs-Firmware zählt in der Regel die bereits im vorherigen Abschnitt erwähnte Integration verschiedener Funktionsbereiche der Steuerungstechnik. Diese lassen sich wiederum in einzelne Funktionen untergliedern, die sich durch stark heterogene Prioritäten und Zeitanforderungen auszeichnen. So umfassen im Fall einer Werkzeugmaschinensteuerung die Logik- und Bewegungssteuerung unter anderem die folgenden Funktionen [53, 181]:

- Da die Werkstückkontur mittels eines beispielsweise gemäß *DIN 66025* [130, 131] erstellten NC-Programms beschrieben wird, ist dieses zunächst zu parsen und zu interpretieren. Dies ist die Aufgabe der *Satzaufbereitung*, die das Programm einer syntaktischen Analyse unterzieht und in die entsprechende interne Struktur der Steuerung transformiert.
- Im Rahmen der *Satzvorbereitung* findet die sogenannte *Geometriedatenverarbeitung* statt, die neben der Geschwindigkeitsführung geometrische Transformationen und Korrekturen wie die Nullpunktverschiebung oder Werkzeugradiuskorrekturen umfasst. Die Aufgabe der Geschwindigkeitsführung besteht bei der Bearbeitung des Werkstücks in der

Anpassung der Vorschubgeschwindigkeit an die Randbedingungen der Maschine, wie beispielsweise Dynamikbegrenzungen der Antriebe. Dabei erfolgt häufig auch ein *Look-Ahead* auf spätere NC-Sätze, um die Geschwindigkeitsführung weiter zu optimieren.

- Die Aufgabe der *Interpolation* besteht in der Überführung der zu verfahrenden Achsbewegungen in zeitdiskrete Lagesollwerte, die schließlich an die Antriebe weitergegeben werden. Zu diesem Zweck werden unterschiedliche Interpolationsarten wie die Geraden-, Kreis- oder Spline-Interpolation eingesetzt. Dabei ist eine achsspezifische Zerlegung der Bewegung zu leisten, die in einer synchronisierten Bewegung abhängiger Achsen resultiert. Kritisch ist im Rahmen der Interpolation, dass die Sollwerte zwingend in einem definierten Takt vorgegeben werden müssen, der wiederum durch die konfigurierte Zykluszeit des Feldbusses determiniert wird. Das Zeitintervall zwischen zwei Sollwertvorgaben wird dabei als *Interpolationstakt* (IPO-Takt) bezeichnet.
- Eine SPS wird im Kontext von Werkzeugmaschinen als *Anpasssteuerung* bezeichnet, da sie maschinenunabhängige Funktionen einer CNC in maschinenspezifischer Weise realisiert. Zu diesen zählen beispielsweise Sicherheitsfunktionen wie die Überwachung von Schutztürverriegelungen sowie die Steuerung von Werkzeug- und Werkstückwechseln oder Kühl- und Schmiermittelzuschaltungen.
- Zu den *Hintergrunddiensten* zählen unter anderem von der Steuerung bereitgestellte Server, wie beispielsweise *FTP* für den Zugriff auf das steuerungsinterne Dateisystem oder *OPC-UA* zum Austausch von Prozessdaten mit der Betriebsleitebene. Weitere Dienste sind die Datenaufbereitung für die Visualisierung oder die Überwachung von Maschinenfunktionen.

Zusammenfassend definiert eine Automatisierungs-Firmware somit eine Vielzahl verschiedener und in der Regel kooperierender Ausführungsfäden. Dabei handelt es sich sowohl um hochpriore, periodische respektive sporadische Aufgaben mit harten Echtzeitanforderungen als auch um aperiodische Hintergrundaufgaben geringer Priorität, die nur die Einhaltung weicher Echtzeitbedingungen erfordern.

2.4 Betriebssysteme

Im Gegensatz zu einfachen Anwendungen kann bei komplexen, mehrfädigen Applikationen wie der im vorherigen Abschnitt geschilderten Steuerungs-Firmware die Verwaltung der meist zahlreichen Ressourcen und Ausführungsfäden in der Regel nicht mehr unter vertretbarem Aufwand im Rahmen der Applikationsentwicklung geleistet werden. Stattdessen empfiehlt es sich, derartige Aufgaben einem *Betriebssystem* zu überlassen. Tanenbaum [168] definiert dabei als die wesentlichen Aufgaben eines Betriebssystems die Bereitstellung einer sauberen Hardware-Abstraktion für den Applikationsentwickler und die Verwaltung der Hardware-Ressourcen.

Ein Betriebssystem wird als *Echtzeitbetriebssystem (Real-Time Operating System, RTOS)* bezeichnet, wenn das Design des Systems auf die Einhaltung vorgegebener Reaktionszeiten ausgerichtet ist. Zu diesem Zweck definieren Tasks ihre Zeitanforderungen meist in Form von *Deadlines*, bis zu denen die Aufgabe erledigt sein muss. Zu unterscheiden sind dabei Aufgaben mit *weichen Echtzeitbedingungen*, bei denen das Überschreiten einer Deadline zwar unerwünscht, aber teilweise akzeptabel ist und Aufgaben mit *harten Echtzeitbedingungen*, bei denen ein sol-

cher Fall unter allen Umständen vermieden werden muss. Eine alternative Strategie zur Definition von Zeitanforderungen in einem Echtzeitbetriebssystem stellen sogenannte *Time/Utility-Functions (TUFs)* dar. Dabei wird der durch die Erledigung einer Aufgabe für das Gesamtsystem erzielte Nutzen statt über eine Deadline in generalisierter Form als frei definierbare Funktion über dem Zeitpunkt, zu dem die Aufgabe abgeschlossen wird, definiert [108]. Generell sind in Echtzeitbetriebssystemen Schutzmaßnahmen wie zum Beispiel die Kontrolle von Ressourcenzugriffen meist gering ausgeprägt, da sich entsprechende Mechanismen nur selten ohne Seiteneffekte auf die Systemreaktivität realisieren lassen [190].

Als *Standardbetriebssystem (General Purpose Operating System, GPOS)* werden hingegen meist Desktop-Betriebssysteme bezeichnet, die stets von nur einem Anwender gleichzeitig genutzt werden. Somit muss ein Standardbetriebssystem alle Funktionen bieten, die im Rahmen von typischerweise an einem Desktop-Computer durchgeführten Arten der Datenverarbeitung benötigt werden. Da hierbei durch den Anwender beliebige Applikationen zur Laufzeit dynamisch nachgeladen werden können, ist der Schutz von Betriebsmitteln vor Zugriffen unberechtigter Anwendungen ein essentielles Entwurfsziel.

2.4.1 Basismechanismen

Mit Ausnahme von Interrupts wird jeder durch ein Betriebssystem ausgeführte sequentielle Kontrollfluss durch eine Task repräsentiert. Gängige Betriebssysteme definieren mindestens drei Zustände, in denen sich eine Task zur Laufzeit des Systems befinden kann: *Running*, *Ready* und *Blocked* [168]. Eine Task befindet sich dabei im Zustand *Running*, wenn sie gerade auf der CPU oder einem CPU-Kern ausgeführt wird, während eine Task im Zustand *Ready* vom Betriebssystem-Scheduler nicht zur Ausführung gebracht wird, da die CPU gegenwärtig durch eine oder mehrere andere Tasks belegt ist. Ist eine Task *Blocked*, so ist diese im Gegensatz zu einer Task im Zustand *Ready* selbst bei einer nicht blockierten CPU nicht ausführungsbereit, da sie auf ein Ereignis oder Betriebsmittel wartet. Die möglichen Übergänge zwischen den Task-Zuständen zeigt Abbildung 2.5.

Im Bereich von Echtzeitsystemen werden meist folgende Task-Typen unterschieden [94]:

- *Periodische Tasks*, die zu regelmäßig wiederkehrenden Zeitpunkten ausführungsbereit werden und harten Echtzeitbedingungen mit periodischer Deadline genügen müssen.
- *Aperiodische Tasks*, die zu unregelmäßigen Zeitpunkten ausführungsbereit werden, dann jedoch keinen harten Echtzeitanforderungen unterliegen.
- *Sporadische Tasks*, die zu unregelmäßigen Zeitpunkten ausführungsbereit werden und dann harten Echtzeitbedingungen genügen müssen.

Gängige Prozessoren definieren häufig unterschiedliche Privilegierungsstufen; im Fall von *Intel*-CPUs werden diese als *Ringe* bezeichnet. In der Regel sind vier Stufen im Bereich von Ring 0 *(Supervisor Mode)* bis Ring 3 *(User Mode)* definiert, die sich bezüglich der Menge der in der jeweiligen Privilegierungsstufe ausführbaren Instruktionen unterscheiden [124]. So sind im User Mode beispielsweise keine die Konfiguration des virtuellen Speichermanagements betreffenden Instruktionen zulässig. Betriebssysteme können nun für jede Task den Ring der Ausführung und auf diese Weise deren Rechte definieren. Während *Windows* und *Linux* die Ringe 0 und 3 nutzen, laufen unter Echtzeitsystemen zugunsten geringerer Reaktionszeiten und eines geringeren Overheads häufig alle Tasks mit den vollen Rechten des Ring 0. Generell wird

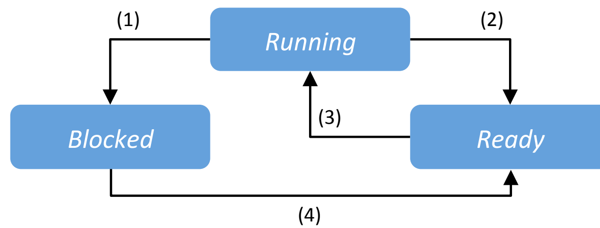


Abbildung 2.5: Übergänge zwischen den Task-Zuständen *Running*, *Ready* und *Blocked*: (1) Task wartet auf Ereignis. (2) Scheduler wählt andere Task zur Ausführung. (3) Scheduler wählt diese Task zur Ausführung. (4) Erwartetes Ereignis liegt vor [168].

ein Wechsel der Privilegierungsstufe von den Ringen 1 bis 3 in den Ring 0 mittels eines sogenannten *Trap* ausgelöst, der wiederum die Konsequenz eines Aufrufs einer Betriebssystemfunktion durch eine Task mittels eines Systemaufrufs *(Systemcall, Syscall)* ist. Die Menge der von einem Betriebssystem zur Verfügung gestellten Syscalls bildet schließlich zusammen mit den im *User Mode* ausführbaren CPU-Instruktionen die sogenannte *Binärschnittstelle (Application Binary Interface, ABI)* [161].

Um ein kooperatives Multitasking zu ermöglichen, stellen Betriebssysteme verschiedene Mechanismen zur Kommunikation und Synchronisation zwischen Prozessen bereit. Diese Mittel der *Interprozesskommunikation und -synchronisation* umfassen beispielsweise bei *UNIX*-basierten Systemen *Pipes*, *Messages*, *Semaphore*, *Signale* und *Shared Memory* [162]. Zu unterscheiden sind dabei Kommunikationsmechanismen der direkten Art, bei denen der Sender den Empfänger explizit definiert und der indirekten Art, bei denen der Sender eine Nachricht ohne Angabe eines Empfängers einem Medium übergibt [184].

2.4.2 SMP-Betriebssysteme

Beim Task-Scheduling auf Multicore-Prozessoren gibt es neben der temporalen Dimension zusätzlich die räumliche Dimension in Form des CPU-Kerns, auf dem eine Task ausgeführt wird. Ein Betriebssystem, das in der Lage ist, Tasks auf unterschiedlichen CPU-Kernen zu verwalten, wird als *SMP-Betriebssystem (Symmetric Multiprocessing-Betriebssystem)* bezeichnet. Dabei liegt auch das Betriebssystem selbst als eine Menge unabhängiger Prozesse vor, die parallel auf den CPU-Kernen ausgeführt werden [20].

Beim Scheduling eines SMP-Systems lassen sich generell *statische Verfahren* mit einer zum Entwicklungszeitpunkt geplanten Ausführung der Tasks und *dynamische Verfahren*, bei denen die Planung zur Laufzeit des Systems erfolgt, unterscheiden. Als Varianten des dynamischen Schedulings in SMP-Systemen sind das *partitionierte Scheduling* und das *globale Scheduling* verbreitet [13]. So erfolgt beim partitionierten Scheduling eine statische Zuordnung der Tasks zu den CPU-Kernen und somit nur ein lokales dynamisches Scheduling. Dieses Verfahren wird häufig in Echtzeitsystemen eingesetzt, da sich auf diese Weise ein besser vorhersagbares und analysierbares Systemverhalten erzielen lässt [126]. Weiterhin verhindert ein partitioniertes Scheduling bei Task-Unterbrechungen Migrationen auf andere CPU-Kerne, in deren Folge die benötigten Daten zunächst in die lokalen Caches nachgeladen werden müssten. Dabei ist zwar

in einem stark ausgelasteten System der Overhead einer Unterbrechung und Fortsetzung der Task-Ausführung auf dem gleichen CPU-Kern nahezu identisch zu dem einer Migration, da hier meist bereits bei einer kurzen Unterbrechung der Ausführung die Cache-Inhalte verdrängt werden. Dies kann jedoch bei einer geringen Systemlast in der Regel vermieden werden und führt hier zu einem signifikant geringeren Overhead, wenn eine Task auf dem Kern fortgesetzt wird, auf dem sie zuvor unterbrochen wurde [19]. Ein partitioniertes Scheduling hat darüber hinaus noch einen positiven Effekt auf den Implementierungsaufwand und somit auf die Effizienz der Parallelisierung: Auf die Sicherstellung der echt parallelen Ausführbarkeit zweier Tasks kann im Fall eines partitionierten Schedulings genau dann verzichtet werden, wenn beide dem gleichen CPU-Kern zugeordnet werden. Dies ist möglich, da diese Tasks dann quasiparallel, aber nicht echt parallel ausgeführt werden können. Als Konsequenz wird ein partitioniertes Scheduling gerne als Brücke von einem Singlecore- auf ein Multicore-System genutzt. Zu beachten ist allerdings, dass die optimale Partitionierung der Tasks ein sogenanntes *Bin-Packing*-Problem darstellt und als solches *NP-schwer* ist. Zudem gibt es Task-Systeme, für die ein partitioniertes Scheduling nicht möglich ist [39]. Beim globalen Scheduling wird hingegen eine zweidimensionale Ausführungsplanung durchgeführt, indem zusätzlich zu den Ausführungszeitpunkten einer Task auch stets der jeweilige CPU-Kern neu bestimmt wird [168]. Ein globales Scheduling kann unter Umständen zu besseren Lastverteilungen führen als ein partitioniertes Scheduling, da die beim partitionierten Scheduling möglichen *Idle*-Zustände eines Cores bei einem gleichzeitigen Warten ausführungsbereiter Tasks auf einem anderen Core verhindert werden. Allerdings ist jede Migration einer Tasks zwischen CPU-Kernen mit einem Overhead für das Befüllen des Cache des neuen Cores verbunden [71], so dass eine häufige Migration die Systemperformanz beeinträchtigen kann.

Ein weiterer, für SMP-Betriebssysteme spezifischer Aspekt ist die Synchronisation und Kommunikation über Kerngrenzen. Um Ereignisse und Informationen im System asynchron zwischen CPU-Kernen zu propagieren, setzen SMP-Betriebssysteme in der Regel sogenannte *Inter-Processor Interrupts (IPIs)* ein [26]. Diese können mittels des lokalen Interrupt-Controllers *(Local Advanced Programmable Interrupt Controller, Local-APIC)* eines CPU-Kerns generiert und über den *APIC Bus* an einen spezifischen anderen CPU-Kern signalisiert werden, um dort beispielsweise ein Rescheduling zu veranlassen. Ein in diesem Zusammenhang häufiges Szenario ist in Abbildung 2.6 dargestellt: Eine Task auf dem CPU-Kern 1 hat zu einem früheren Zeitpunkt synchron eine Message aus einer *Message-Queue* abgefragt, die allerdings noch leer war. Als Konsequenz wurde die entsprechende Task daraufhin in den Zustand *Blocked* versetzt. Nun sendet eine weitere Task auf Kern 0 eine Message an eben diese Queue, woraufhin die Empfänger-Task zunächst *Ready* wird und daraufhin vom Betriebssystem-Scheduler in den Status *Running* versetzt wird. Zu erkennen ist dabei der zur Signalisierung genutzte IPI auf Kern 1.

2.5 Parallele Programmierung

2.5.1 Entwicklung paralleler Software

Um Multicore-CPUs in adäquater Weise zu nutzen, ist, wie in Abschnitt 2.2 beschrieben, eine auf Taskebene parallel ausführbare Software erforderlich, welche die einzelnen CPU-Kerne mit jeweils separaten Kontrollflüssen versorgen kann. Ein möglicher Weg zu einem derart parallelen System besteht darin, dieses von Grund auf neu zu entwerfen und zu implementieren, wobei im

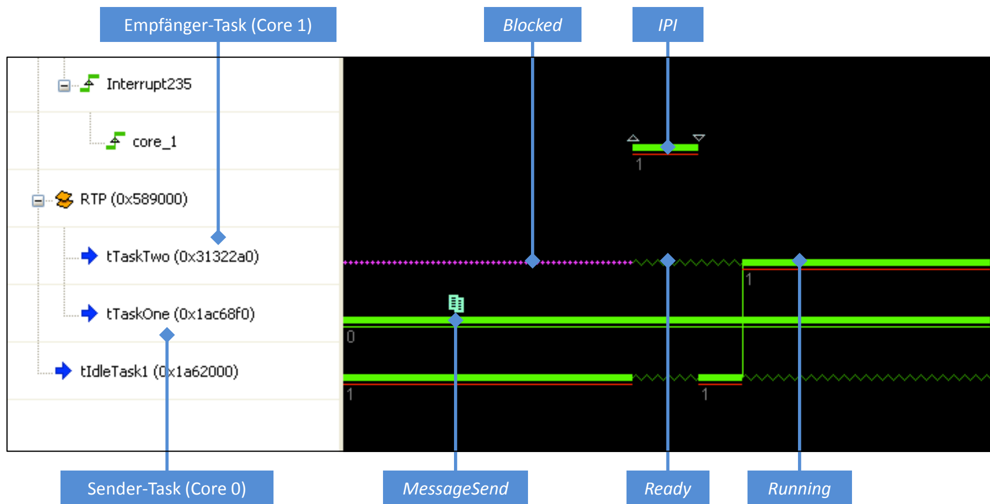


Abbildung 2.6: Inter-Processor Interrupt (IPI) im Echtzeitbetriebssystem *Wind River® VxWorks®*

Bereich eingebetteter Systeme häufig Verfahren des sogenannten *Model Driven Software Development (MDSD)* angewandt werden. Beim *modellzentrierten Ansatz* werden dabei sogar formal beschriebene Struktur- und Verhaltensspezifikationen von Software in automatisierter Weise in lauffähigen Code überführt [37]. Derartige Verfahren bilden die Grundlage einer Vielzahl kommerzieller Tools, wie beispielsweise *IBM Rational Rhapsody* [81], *MATLAB/Simulink* [116] oder *ASCET* [52]. Die modellbasierte Entwicklung fand in den letzten Jahren vor allem im Bereich paralleler Systeme zunehmend Verbreitung, da formale Notationen wie *UML* oder *SysML* eine vereinfachte Spezifikation paralleler Abläufe versprechen. Eine weitere Alternative für eine Neuimplementierung, die vereinzelt auch im Bereich eingebetteter Systeme Anwendung findet [167], ist der Einsatz funktionaler Programmiersprachen [146] wie *Haskell*, die sich durch eine implizite Parallelität auszeichnen.

Umfangreiche und langjährige Investitionen in bestehende Software schließen jedoch gerade bei komplexen und anspruchsvollen Systemen in der Regel eine vollständige Neuentwicklung aus ökonomischen Gründen aus. Die Alternative besteht deshalb in den meisten Fällen in der Parallelisierung der bestehenden, imperativ implementierten Applikation, wobei ein quasiparalleles Multitasking-System mit nebenläufigen Tasks und Interrupts hierfür die idealen Voraussetzungen bietet. Der Vorteil eines solchen Systems besteht darin, dass dessen Funktionalität bereits in eine Menge nebenläufig ausführbarer Kontrollflüsse partitioniert ist. Die Parallelisierung eines quasiparallelen Systems für Multicore-CPUs erfolgt schließlich, indem die Tasks und Interrupts in geeigneter Weise auf die verfügbaren CPU-Kerne verteilt werden, wobei allerdings zunächst die im nachfolgenden Abschnitt beschriebenen Voraussetzungen zu schaffen sind.

Zeichnet sich das bestehende System hingegen durch eine fehlende oder noch nicht zufriedenstellende Nebenläufigkeit aus, ist zunächst dessen Aufspaltung in eine ausreichende Menge nebenläufiger Kontrollflüsse erforderlich. Während Parallelität auf Daten- und Instruktionsebe-

ne durch geeignete Compiler oder durch die CPU selbst zur Laufzeit extrahiert werden kann, konnten bislang trotz intensiver Forschung keine zufriedenstellenden Verfahren zur automatischen Parallelisierung auf Taskebene entwickelt werden [146]. Somit muss dieser Vorgang entweder manuell oder semiautomatisch durchgeführt werden. Im manuellen Fall muss jeder Aspekt der Parallelisierung, wie die Generierung, Kommunikation und Synchronisation der Tasks, durch den Entwickler selbst organisiert und implementiert werden. Im semiautomatischen Fall hingegen werden diese Aufgaben zwar durch Bibliotheken wie *OpenMP* [48] oder *Cilk* [22] übernommen, die vorbereitenden Maßnahmen für eine Parallelisierung durch den entsprechenden Compiler sind allerdings dennoch durch den Entwickler in Form von teils umfangreichen Quellcode-Annotationen zu leisten.

2.5.2 Paradigmen paralleler Programmierung

Bereits bei der Implementierung quasiparalleler Programme für Architekturen mit gemeinsamem Speicher sind einige potentielle Fallstricke zu umgehen: Während Codesequenzen in einem einfädigen *(single-threaded)* System ohne Interrupts immer unterbrechungsfrei ausgeführt werden, gilt diese Annahme bereits in quasiparallelen Systemen nicht mehr, da Tasks jederzeit suspendiert und zu einem späteren Zeitpunkt fortgeführt werden können. In echt parallelen Systemen ist darüber hinaus sogar die gleichzeitige Ausführung von Tasks möglich. Um ein dadurch verursachtes Fehlverhalten des Systems zu vermeiden, gibt es in parallelen und quasiparallelen Systemen folgende Strategien:

- *Multilaterale (mehrseitige) Synchronisation*: Die Task-Präemption und die gleichzeitige Task-Ausführung stellen eine Fehlerquelle dar, wenn Tasks gemeinsame Ressourcen nutzen. Um dabei konsistente Systemzustände zu gewährleisten, müssen Transaktionen auf diesen Ressourcen in der Regel unter einem wechselseitigen Ausschluss aller beteiligten Tasks ausgeführt werden. Die entsprechenden Codesequenzen werden dabei als *kritische Abschnitte (Critical Sections)* bezeichnet. Mittels geeigneter Synchronisationsmechanismen wie Spinlocks oder Semaphoren lässt sich der Eintritt in kritische Abschnitte wechselseitig verriegeln: Der entsprechende Abschnitt kann erst dann durch eine Task betreten werden, wenn sich keine andere darin befindet. Andernfalls findet ein aktives respektive passives Warten statt.
- *Wiedereintrittsfähigkeit (Reentrancy):* Kann eine Funktion vor dem Ende ihrer Ausführung bereits ein weiteres Mal aufgerufen werden, so ist dies genau dann problematisch, wenn diese auf gemeinsam genutzte Variablen oder Ressourcen zugreift [58]. In diesem Fall ist die Wiedereintrittsfähigkeit der Funktion sicherzustellen, indem diese entweder auf lokalen Kopien von Daten arbeitet oder Zugriffe auf globale Daten synchronisiert werden.

Zudem ist in nebenläufigen Programmen an ausgewählten Stellen die Reihenfolge der Task-Ausführung zu koordinieren. Dies ist erforderlich, wenn zwei Tasks kausal voneinander abhängig sind, da die Ausführung einer Task das Resultat einer anderen Task erfordert. Diese Art der Synchronisation wird als *unilaterale (einseitige) Synchronisation* oder *Bedingungssynchronisation* bezeichnet. Eine besondere Form der einseitigen Synchronisation ist die sogenannte *Barrierensynchronisation*. Hier wartet eine Gruppe von Tasks an einer Barriere, bis diese von allen Tasks der Gruppe erreicht wurde.

Selbst ein System, welches in quasiparalleler Weise fehlerfrei ausgeführt wird, kann im Fall einer echt parallelen Ausführung durch eine Verteilung der bestehenden Tasks auf die Kerne einer Multicore-CPU zu Problemen wie *Race Conditions* oder *Deadlocks* führen. So wird in quasiparallelen Systemen unter Umständen auf die Implementierung expliziter mehrseitiger Synchronisationen oder wiedereintrittsfähiger Funktionen verzichtet. Stattdessen wird mittels Methoden wie dem vorübergehenden Sperren von Interrupts und Kontextwechseln eine unterbrechungsfreie Ausführung kritischer Abschnitte und somit in impliziter Weise der wechselseitige Ausschluss sichergestellt. In SMP-Systemen sind derartige Sperrmechanismen aufgrund der echt parallelen Task-Ausführung hingegen nicht geeignet, um einen wechselseitigen Ausschluss zu erzwingen und wirken aus diesem Grund hier in der Regel nur auf spezifische CPU-Kerne [20]. Weiterhin werden in quasiparallelen Systemen bei der Implementierung nicht selten Annahmen über die Task-Ausführung getroffen. So kann hier die Ausführung von Interrupts ebenso wie die Ausführung der höchstprioren Task als atomar bezüglich anderer Tasks angenommen werden. Außerdem kann im Fall eines Verzichts auf ein Zeitscheiben-Scheduling die Ausführung von Tasks gleicher Priorität als unterbrechungsfrei zueinander gelten. Auch diese Annahmen gelten in einem echt parallelen System nicht mehr, so dass bei quasiparallelen Systemen gegebenenfalls noch umfangreiche Anpassungen erforderlich sind, um eine fehlerfreie echt parallele Ausführung zu gewährleisten.

2.6 Modellierung

Die Motivation zur Modellierung eines Systems besteht darin, dass die zur Analyse des Systemverhaltens erforderlichen Modifikationen des realen Systems entweder nicht möglich oder im erforderlichen Umfang zu aufwändig beziehungsweise zu teuer sind [106]. Andernfalls könnte man auf eine Modellierung verzichten, da diese immer die Herausforderungen einer Modellvalidierung mit sich bringt.

Generell werden physikalische Modelle, wie beispielsweise ein miniaturisierter Nachbau, und mathematische Modelle, die ein System mittels logischer und quantitativer Relationen beschreiben, unterschieden. Die Untersuchung eines mathematischen Modells kann dabei entweder analytisch oder simulativ erfolgen. Eine analytische Untersuchung wird zwar im Allgemeinen bevorzugt, ist allerdings häufig aufgrund einer zu hohen Systemkomplexität nicht möglich. Zur analytischen Untersuchung besonders geeignet sind Modelle, die auf Graphen basieren. Diese zeichnen sich durch folgende Eigenschaften aus [92]:

- Modelle bilden meist Objekte und deren Beziehungen zueinander ab. Dies ist in Graphenmodellen mittels Knoten und Kanten in sehr intuitiver Weise möglich.
- Graphenmodelle sind mathematisch präzise, da sie auf Relationen basieren. Somit ist im Rahmen der Modellanalyse die formale Ableitung vielfältiger Eigenschaften möglich.
- Auf Graphen basierende Modelle lassen sich systematisch als Datenstrukturen implementieren. Zudem existieren effiziente Algorithmen zur Analyse und Visualisierung.

Eine algorithmische Analyse erfordert allerdings zunächst die Implementierung als Computermodell. Der Prozess der Generierung von Simulationsmodellen ist nach Sargent [154] in der einfachen Form entsprechend Abbildung 2.7 definiert, kann aber auf analytisch auswertbare Modelle übertragen werden: Zunächst wird auf Basis des realen Systems *(Problem-Entität)* ein

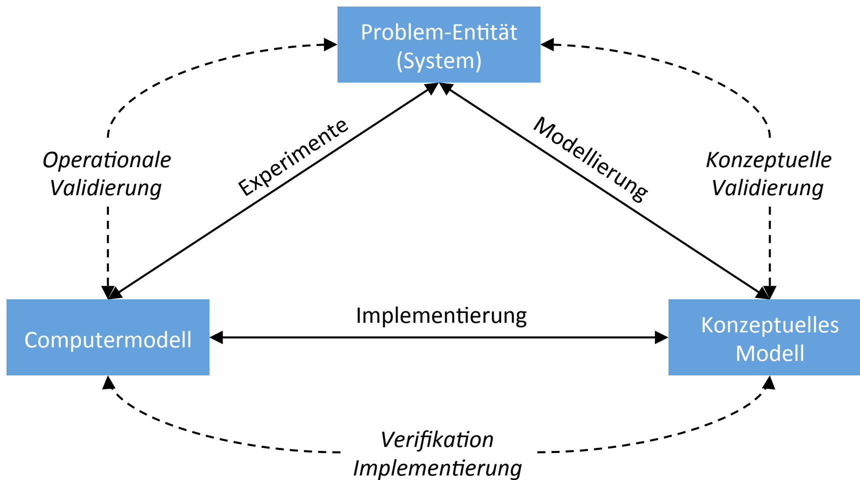


Abbildung 2.7: Einfaches Vorgehensmodell der Modellierung nach Sargent [154]

konzeptuelles Modell erstellt, welches schließlich in ein Computermodell überführt wird. Dieses dient schließlich der modellbasierten Untersuchung des Realsystems. Im Rahmen der konzeptuellen Validierung ist dabei der Nachweis zu erbringen, dass das Modell für die gewünschte Systemanalyse geeignet ist, während die Verifikation des Computermodells die Verifikation der Implementierung des konzeptuellen Modells zum Gegenstand hat. Die operationale Validierung führt durch den Vergleich von Resultaten des Realsystems und des Modells den Nachweis einer ausreichenden Genauigkeit der Modellergebnisse für die gewünschte Untersuchung.

2.7 Software-Analyse

Um die zur Modellierung eines bestehenden Software-Systems erforderlichen Informationen zu erhalten, ist eine Analyse desselben unumgänglich. Bezüglich des dabei angewandten Vorgehens werden folgende Alternativen unterschieden:

- Die Grundlage einer *statischen Analyse* bildet stets der Quellcode einer Implementierung. Zu den Verfahren der statischen Codeanalyse zählt unter anderem das sogenannte *Program Slicing*, bei dem sich beispielsweise alle Instruktionen eines Programms extrahieren lassen, die den Wert einer Variable zu einem bestimmten Zeitpunkt beeinflussen [183]. Eine Herausforderung bei der statischen Analyse stellen allerdings dynamische Systemeigenschaften wie Speicherreferenzen, die mittels Zeigerarithmetik zur Laufzeit modifiziert werden, dar. Um diesem Problem zu begegnen, wurden verschiedene Verfahren der sogenannten *Pointer Alias Analyse* entwickelt [78], die allerdings aufgrund der Unentscheidbarkeit einer exakten Alias-Analyse [102, 143] stets nur eine Approximation liefern können, die in den meisten Fällen konservativ und somit potentiell überschätzend sein muss. Dies führt dazu, dass eine statische Analyse Eigenschaften eines Systems extrahiert, die so in der Realität möglicherweise nicht existieren [179].

- Eine *dynamische Analyse* erfasst die Eigenschaften eines laufenden Programms [15], so dass diese Analyse zur Laufzeit des Systems durchgeführt wird. Als Konsequenz bezieht sich eine dynamische Analyse allerdings immer auf das Verhalten des Systems unter einem spezifischen Ausführungsfall und liefert somit nur Modelle mit einer partiellen Abdeckung. Um ein vollständiges Modell zu erhalten, ist hingegen eine dynamische Analyse unter allen denkbaren Ausführungsszenarien durchzuführen, was zu einer schlechten Skalierbarkeit dieses Verfahrens führt. Als Kompromiss erfolgt eine dynamische Analyse häufig auf repräsentativen Ausführungsfällen, wohl wissend, dass die Resultate aufgrund der potentiellen Unvollständigkeit mit einer gewissen Skepsis zu interpretieren sind [179].

Eine Form der dynamischen Analyse stellt das sogenannte *Software-Profiling* dar, bei dem zur Laufzeit eines Systems angenommene Zustände und aufgetretene Ereignisse aufgezeichnet werden. Hierzu gibt es verschiedene Möglichkeiten, von denen nachfolgend ohne Anspruch auf Vollständigkeit eine Auswahl genannt wird. Die erste besteht darin, die Software zunächst mittels geeigneter Instruktionen so zu instrumentieren, dass relevante Daten aufgezeichnet werden. Die Instrumentierung kann dabei entweder statisch in automatisierter oder manueller Weise im Source-, Assembler- oder Binärcode oder aber auch dynamisch zur Laufzeit erfolgen. Der wesentliche Nachteil dieses Verfahrens besteht allerdings darin, dass die Aufzeichnung der Daten häufig einen signifikanten Overhead generiert, der wiederum das Laufzeitverhalten der Software beeinflusst. Alternativ zur Software-Instrumentierung ist es möglich, das Profiling der darunterliegenden Hardware zu überlassen, was allerdings neben der *Intel Itanium*-Plattform [41] nur wenige Architekturen unterstützen. Ein weiteres Verfahren ist das *Sampling*-basierte Profiling, bei dem das laufende System in definierbaren Intervallen unterbrochen wird, um relevante Zustände aufzuzeichnen [161]. Eine weniger verbreitete Alternative zur Instrumentierung stellt schließlich die Ausführung der Software in einer virtuellen Maschine oder einem sogenannten *Instruction Set Simulator* dar [42]. Im Rahmen der Interpretation der Instruktionen durch den Hypervisor oder den Simulator kann auch hier ein Profiling des Laufzeitverhaltens eines Systems durchgeführt werden.

2.8 Skalare und multikriterielle Optimierung

Als *Optimierung* wird generell die Suche nach den Elementen eines Entwurfsraums bezeichnet, die sich hinsichtlich definierter Kriterien durch die höchste Güte auszeichnen. Nach [182] lässt sich ein *skalares Optimierungsproblem* wie folgt definieren:

Definition 1 (Skalare Optimierung) *Sei Ω der Entwurfsraum möglicher Lösungen, $f\colon \Omega \to \mathbb{R}$ eine Bewertungs- oder Zielfunktion und $\succ \in \{<, >\}$ eine Vergleichsrelation. Die Menge $X \subseteq \Omega$ der globalen Optima definiert sich nun wie folgt:*

$$X = \left\{ x \in \Omega \,\middle|\, \forall_{x' \in \Omega} \colon f\left(x\right) \succeq f\left(x'\right) \right\} \tag{2.1}$$

Ein *multikriterielles Optimierungsproblem*, auch häufig als *Pareto*- oder *Vektoroptimierung* bezeichnet, liegt hingegen genau dann vor, wenn eine potentielle Lösung $x \in \Omega$ für ein Problem unter n Kriterien $F(x) = (f_1(x), \ldots, f_n(x))$ zu bewerten ist. Häufig stehen die Kriterien dabei zueinander in Konflikt, so dass eine Verbesserung bei einem Kriterium mit einer Verschlechterung bei einem anderen Kriterium erkauft werden muss. Um ein multikriterielles

Optimierungsproblem zu lösen, gibt es verschiedene Alternativen. Die erste besteht darin, das Problem in ein skalares Optimierungsproblem zu transformieren, indem eine Zielfunktion $f(x)$ gebildet wird, die alle Kriterien $F(x)$ in gewichteter Weise aggregiert:

$$f(x) = m_1 \cdot f_1(x) + \cdots + m_n \cdot f_n(x) \tag{2.2}$$

Im Rahmen der Optimierung wird nun ausschließlich diese Zielfunktion $f(x)$ optimiert, so dass entsprechende Verfahren der skalaren Optimierung angewandt werden können. Dies hat jedoch zur Konsequenz, dass mittels der Faktoren m_i vor der Optimierung eine subjektive Präferenz bezüglich der einzelnen Kriterien zu definieren ist. Somit ist im Fall einer Adaption der Präferenz die Optimierung erneut durchzuführen. Alternativ dazu kann die multikriterielle Bewertung auch im Rahmen der Optimierung beibehalten werden, dies wird als *Pareto-Optimierung* bezeichnet. In Anlehnung an [175] gilt:

Definition 2 (Pareto-Dominanz) *Eine Lösung $x \in \Omega$ Pareto-dominiert eine Lösung $x' \in \Omega$ hinsichtlich ihrer Bewertungsfunktionen $F(x) = (f_1(x), \ldots, f_n(x))$ genau dann, wenn gilt:*

$$\forall i \in \{1, \ldots, n\} : f_i(x) \geq f_i(x') \wedge \exists i \in \{1, \ldots, n\} : f_i(x) > f_i(x') \tag{2.3}$$

Darauf basierend sei die Menge der *globalen Pareto-optimalen* Lösungen definiert [49]:

Definition 3 (Globale Pareto-optimale Menge) *Sei $X \subseteq \Omega$ eine Menge von Lösungen. Wenn im Entwurfsraum Ω keine Lösung $x \in \Omega$ existiert, die eine Lösung $x' \in X$ Pareto-dominiert, dann bildet X eine globale Pareto-optimale Menge.*

Weiterhin sei die Menge der *lokalen Pareto-optimalen* Lösungen definiert [49]:

Definition 4 (Lokale Pareto-optimale Menge) *Sei $\hat{X} \subseteq \Omega$ eine Menge von Lösungen. Wenn zu jeder Lösung $x \in \hat{X}$ keine Lösung $y \in \Omega$ mit der Eigenschaft $\|F(y) - F(x)\|_\infty \leq \varepsilon$, wobei ε eine kleine positive Zahl ist, existiert, die eine Lösung $x' \in \hat{X}$ Pareto-dominiert, dann bildet $\hat{X}$ eine lokale Pareto-optimale Menge.*

Die Bewertungen $F(x) = (f_1(x), \ldots, f_n(x))$ aller Lösungen $x \in X$ einer globalen Pareto-optimalen Menge X werden schließlich, wie in Abbildung 2.8 dargestellt, als *globale Pareto-Front* bezeichnet [175]. Analog dazu bilden die Bewertungen $F(x)$ aller Lösungen $x \in \hat{X}$ einer lokalen Pareto-optimalen Menge $\hat{X}$ die *lokale Pareto-Front*.

2.9 Genetische Algorithmen

Sowohl bei skalaren als auch bei multikriteriellen Optimierungsproblemen verhindert oft die Komplexität des Problems eine systematische oder gar vollständige Durchmusterung des Entwurfsraums innerhalb einer vertretbaren Rechenzeit. Geeignete Methoden zur Lösung derartiger Optimierungsprobleme sind deshalb sogenannte *lokale Suchverfahren* wie das *Hill-Climbing* oder das *Simulated Annealing* [153]. Ein durch die biologische Evolution inspiriertes lokales Suchverfahren stellen darüber hinaus die sogenannten *genetischen Algorithmen* dar, die in den 1970er Jahren von John Holland entwickelt wurden [79]. Diese bilden neben den *Evolutionsstrategien* sowie dem *evolutionären* und *genetischen Programmieren* eine Teilmenge der

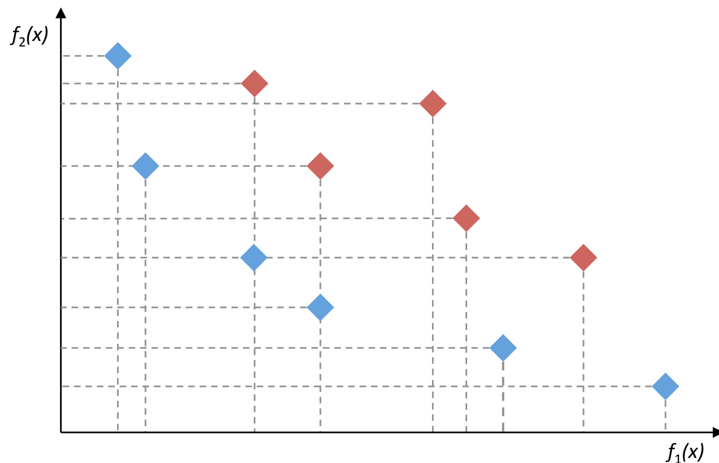


Abbildung 2.8: Darstellung der globalen Pareto-Front (blaue Marken) und der Pareto-dominierten Lösungen (rote Marken) einer multikriteriellen Optimierung mit dem Ziel der Minimierung zweier Kriterien $F(x) = (f_1(x), f_2(x))$

evolutionären Algorithmen [182], wobei die Grenzen zwischen diesen historisch gewachsenen Domänen in den letzten Jahren zunehmend verschwimmen.

Das grundlegende Konzept eines genetischen Algorithmus besteht darin, zu jedem Zeitpunkt eine Menge potentieller Problemlösungen *(Population)* zu verwalten, die über eine Vielzahl an Generationen hinweg den Mechanismen der Evolution unterworfen werden. Die Population besteht dabei aus einer Menge von Individuen, den sogenannten *Genotypen*. Jeder Genotyp repräsentiert dabei eine konkrete, nachfolgend als *Phänotyp* bezeichnete, potentielle Problemlösung, die in geeigneter Weise durch eine spezifische Anzahl an *Genen* kodiert wird. Die möglichen Ausprägungen eines Gens werden als *Allele* bezeichnet, wobei sich die Kodierung bei genetischen Algorithmen häufig auf nur zwei Allele beschränkt und somit mittels des Binäralphabets erfolgen kann. Allerdings sind auch beliebige andere problemspezifische Repräsentationen möglich. Den prinzipiellen Ablauf, der den meisten genetischen Algorithmen gemein ist, skizziert Abbildung 2.9.

In einem ersten Schritt wird eine Initialpopulation erzeugt, deren Elemente in der Regel zufällig generiert werden. Der weitere Ablauf des Algorithmus gestaltet sich nun in zyklischer Weise. Zunächst findet eine Bewertung der generierten Individuen hinsichtlich ihrer Eignung als Problemlösung statt, indem jeweils ein skalarer oder multikriterieller *Fitnesswert* berechnet wird, der diese Güte widerspiegelt. Auf Basis dieser Bewertung erfolgt nun unter Einsatz geeigneter Strategien die Selektion einer Teilmenge der Population als Elternindividuen der nachfolgenden Generation. Dieser Vorgang bildet die erstmals durch Charles Darwin beschriebene natürliche Selektion insofern ab, als Individuen mit besonders guter Fitness bevorzugt zur Reproduktion ausgewählt werden. Die zur Fortpflanzung ausgewählten Elternindividuen enthalten nun das genetische Material, auf Basis dessen die Individuen der nachfolgenden Generation generiert werden. Besonders verbreitet ist dabei zunächst die *Rekombination* mittels Kreuzung *(Cross-*

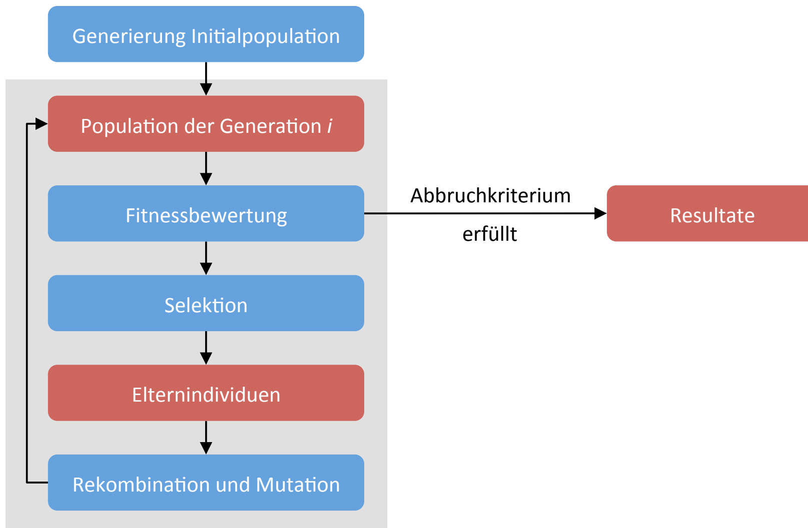


Abbildung 2.9: Darstellung des grundlegenden Ablaufs eines genetischen Algorithmus

over), bei der jeweils zwei Individuen die Basis für ebenso viele Nachkommen liefern. Dieser Operator wird in der Regel mit einer gewissen Wahrscheinlichkeit $0 < p_c < 1$ angewandt, so dass einzelne Individuen im Gegensatz zur natürlichen Fortpflanzung auch unverändert in die nachfolgende Generation übernommen werden können. Der alleinige Einsatz der Rekombination zur Fortpflanzung birgt allerdings die Gefahr, dass die nachfolgenden Generationen ausschließlich beschränkte Bereiche des Entwurfsraums abdecken und der Algorithmus sich somit in einem lokalen Optimum verfängt. Aus diesem Grund wird häufig noch eine *Mutation* mit einer Wahrscheinlichkeit $0 < p_m < 1$ pro Gen eines Genotyps durchgeführt, bei der im Fall einer binären Repräsentation eine Invertierung und im Fall komplexerer Repräsentationen eine Ersetzung durch ein zufällig gewähltes Allel stattfindet. Auf diese Weise wird sichergestellt, dass in regelmäßigen Abständen auch neue Regionen des Entwurfsraums erreicht werden können. Die auf diese Weise generierten Individuen können nun entweder die jeweils vorhergehende Generation vollständig ersetzen oder in geeigneter Weise in diese integriert werden. Unabhängig von der gewählten Integrationsstrategie steigert sich allerdings aufgrund der bevorzugten Selektion von Individuen mit hoher Fitness die durchschnittliche Fitness der Population über eine ausreichende Zahl an Iterationen hinweg und nährt so die Hoffnung auf eine stetige Annäherung an die optimalen Problemlösungen des Entwurfsraums.

Der zyklische Prozess des genetischen Algorithmus wird, wie in Abbildung 2.9 dargestellt, so lange wiederholt, bis eine definierte Abbruchbedingung erfüllt ist. Als entsprechendes Kriterium kann beispielsweise die Konvergenz des genetischen Materials der Population oder das Ausbleiben einer Steigerung der Fitness über mehrere Generationen hinweg dienen. Als Lösungen des Algorithmus werden schließlich in der Regel die Individuen der letzten Generation ausgegeben. Es gibt allerdings auch Strategien, bei denen ein vollständiges, generationenübergreifendes Archiv der bis zum Abbruchzeitpunkt generierten Individuen mitgeführt wird.

Generell gibt es bei der Entwicklung genetischer Algorithmen einen enormen Parameterraum. Dieser umfasst nicht nur die Populationsgröße oder die jeweilige Anzahl der Elternindividuen, sondern auch die Ausgestaltung der zuvor erwähnten Strategien der Selektion, Rekombination, Mutation und Integration. Darüber hinaus sind häufig weitere Maßnahmen zu definieren, wie beispielsweise der Umgang mit Genotypen, die sich auf keinen oder keinen validen Phänotyp abbilden lassen.

2.10 Systemvirtualisierung

Der Begriff der *Virtualisierung* wird in der Informatik in vielfältigen Kontexten verwendet. Einen zentralen Aspekt stellt dabei in der Regel die Bereitstellung einer Standard-Software-Schnittstelle für die Interaktion mit einem realen System dar, die im Gegensatz zu einer *Abstraktion* nicht zwangsläufig den Detailgrad reduziert [138, 161]. Der nachfolgende Abschnitt widmet sich der Technik der sogenannten *Systemvirtualisierung*, mittels derer sich eine virtualisierte Ausführung von Betriebssystemen realisieren lässt.

2.10.1 Definition und formale Betrachtung

Eine Systemvirtualisierung ermöglicht die Ausführung von Betriebssystemen in einer als *virtuelle Maschine (VM)* bezeichneten Laufzeitumgebung, die von einem sogenannten *Virtual Machine Monitor (VMM)* oder *Hypervisor* verwaltet wird. Jede virtuelle Maschine stellt dabei ein Abbild einer realen Hardware dar und definiert somit als Schnittstelle in der Regel eine Befehlssatzarchitektur [161]. Diese Eigenschaften treffen auch für eine *Hardware-Emulation* zu, allerdings mit dem signifikanten Unterschied, dass sich bei einer Emulation im Gegensatz zur Virtualisierung die Befehlssatzarchitekturen der realen Hardware und der virtuellen Maschine unterscheiden. Eine formale Definition der Virtualisierung lieferten Gerald J. Popek und Robert P. Goldberg im Jahr 1974, im Rahmen derer sie folgende Eigenschaften fordern [140]:

- *Äquivalenz:* Die Ausführung in einer virtuellen Maschine muss im Vergleich zur nichtvirtualisierten Ausführung im Wesentlichen identische Rahmenbedingungen bieten. Dies gilt auch bei gleichzeitigen weiteren Aktivitäten auf dem realen System, so dass diese Forderung auch die Isolation gegenüber anderen Ausführungsumgebungen impliziert. Ausnahmen hierzu stellen nur Abweichungen im Zeitverhalten oder Unterschiede aufgrund der fehlenden Verfügbarkeit von Systemressourcen dar.
- *Effizienz:* Die Performanz der virtuellen Maschine darf sich von der Performanz der darunterliegenden realen Maschine nur geringfügig unterscheiden. In Abgrenzung zu einer Emulation bedingt diese Forderung die direkte Ausführung eines Großteils der Instruktionen der virtuellen Maschine auf der realen CPU ohne Intervention des Hypervisors.
- *Ressourcenkontrolle:* Der Hypervisor muss die volle Kontrolle über alle Systemressourcen besitzen. Dies impliziert, dass nur der Hypervisor eine Ressourcenallokation zu Gastsystemen vornehmen oder wieder aufheben darf und dass aus einer virtuellen Maschine heraus nur der Zugriff auf Ressourcen möglich ist, die dieser virtuellen Maschine zugewiesen wurden.

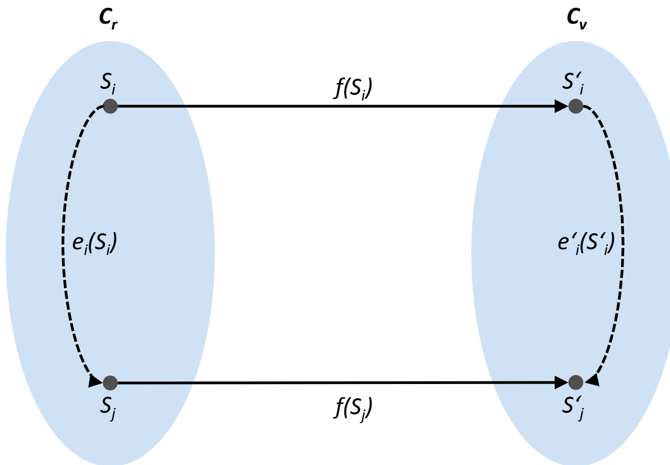


Abbildung 2.10: Darstellung einer Virtualisierung als Isomorphismus nach [140]

Die Äquivalenzforderung formalisieren Popek und Goldberg als die Existenz eines Isomorphismus $f: C_r \rightarrow C_v$ zwischen zwei Zustandsmengen C_r und C_v, welche eine Partitionierung der Menge C aller Maschinenzustände bilden. Dabei umfasse C_v alle Zustände aus C, bei denen der VMM im Speicher hinterlegt ist und C_r alle übrigen Zustände aus C. Weiterhin sei auf C_r ein unärer Operator $e_i: C_r \rightarrow C_r$ und auf C_v ein unärer Operator $e'_i: C_v \rightarrow C_v$ definiert. Dabei repräsentiere e_i eine Sequenz von Instruktionen, die in einem Zustand $S_i \in C_r$ der realen Maschine ausgeführt werden und diese in einen Zustand $S_j \in C_r$ überführen: $e_i(S_i) = S_j$. Dies gelte analog für die Instruktionsfolgen e'_i der virtuellen Maschine: $e'_i(S'_i) = S'_j$ mit $S'_i, S'_j \in C_v$. Somit lassen sich folgende Anforderungen an eine Virtualisierung ableiten:

- Es existiert ein als *VM-Map* bezeichneter Isomorphismus $f: C_r \rightarrow C_v$, der jeden Zustand $S \in C_r$ der realen Maschine auf einen Zustand $S' \in C_v$ der virtuellen Maschine abbildet.
- Zu jeder Instruktionsfolge e_i der realen Maschine muss eine korrespondierende Instruktionsfolge e'_i der virtuellen Maschine existieren und ausgeführt werden können, so dass entsprechend der Darstellung in Abbildung 2.10 gilt: $f(e_i(S_i)) = e'_i(f(S_i))$.

Als weitere Voraussetzung für die Virtualisierbarkeit einer Maschine definieren Popek und Goldberg die Verfügbarkeit von mindestens zwei unterschiedlichen Privilegierungsstufen der CPU. In diesem Zusammenhang erfolgt eine Klassifikation der im Befehlssatz einer Maschine definierten Instruktionen in sogenannte *privilegierte* und *nicht privilegierte Instruktionen*. Ein Befehl gilt als privilegiert, wenn seine Ausführung im User-Modus einen Trap auslöst, als dessen Konsequenz der Systemzustand vor dem Trap gesichert wird, bevor das System den Zustand wechselt. Ein derartiger Systemzustand definiert sich unter anderem über die jeweilige Privilegierungsstufe und den Inhalt des *Befehlszählers (Program Counter, PC)*. Bei der Ausführung einer privilegierten Instruktion im Supervisor-Modus findet hingegen kein Trap statt.

Weiterhin erfolgt eine Klassifikation des Instruktionssatzes in *sensitive* und *nicht sensitive Instruktionen*. Die Klasse der sensitiven Instruktionen wiederum lässt sich in *verhaltenssensitive*

und *kontrollsensitive Instruktionen* untergliedern. Zur ersten Kategorie zählen Instruktionen, deren Verhalten von der aktuellen Privilegierungsstufe der CPU oder der Adresse der Instruktion im physikalischen Programmspeicher abhängig ist. Entsprechende Verhaltensabweichungen können dazu führen, dass ein in einer virtuellen Maschine ausgeführtes Gastsystem erkennen kann, dass es nicht auf der realen Maschine läuft. Als kontrollsensitiv wird eine Instruktion hingegen dann bezeichnet, wenn diese in der Lage ist, Änderungen am Zustand einer Maschine vorzunehmen, die im Kontext der Virtualisierung nicht zulässig sind. Dazu zählt beispielsweise die Änderung der aktuellen Privilegierungsstufe der CPU oder die Konfiguration der virtuellen Speicherverwaltung der Maschine. Somit würde die Ausführung einer kontrollsensitiven Instruktion durch ein Gastsystem dieses in die Lage versetzen, Aktionen außerhalb seiner virtuellen Maschine zu veranlassen, die weitere Gastsysteme beeinflussen können.

Das zentrale Theorem von Popek und Goldberg besagt nun, dass eine Rechnerarchitektur nur dann virtualisiert werden kann, wenn deren sensitive Instruktionen eine Teilmenge der privilegierten Instruktionen darstellen[2]. Eine Skizze des entsprechenden Beweises liefert [140].

Sind die zuvor definierten Bedingungen erfüllt, lässt sich eine Virtualisierung mittels *Trap-and-Emulate* realisieren [161]. Dabei besitzt jeglicher innerhalb einer virtuellen Maschine ausgeführte Code lediglich User-Rechte auf der CPU, wobei zur Steigerung der Effizienz alle nicht zur Klasse der sensitiven Befehle zählenden Instruktionen direkt ausgeführt werden. Die Ausführung einer sensitiven Instruktion löst hingegen einen Trap in den Supervisor-Modus aus, dessen Sprungziel der Virtual Machine Monitor ist. Dessen Aufgabe besteht nun in der Emulation des Verhaltens der entsprechenden Instruktion in einer Weise, die den zuvor geforderten Eigenschaften einer Virtualisierung entspricht, bevor schließlich ein Rücksprung in die virtuelle Maschine erfolgt.

2.10.2 Strategien der Systemvirtualisierung auf x86-Architekturen

Die am weitesten verbreitete Mikroprozessorarchitektur im Bereich von Desktop-Computern und Servern ist zweifellos die im Jahr 1978 von *Intel* entwickelte und seitdem auch von zahlreichen anderen Herstellern wie beispielsweise *AMD* oder *IBM* eingesetzte *x86-Architektur* [138]. Zu deren Entwurfszielen zählte allerdings nicht die Virtualisierbarkeit, so dass die im vorherigen Abschnitt beschriebene zentrale Forderung von Popek und Goldberg an eine virtualisierbare Rechnerarchitektur von x86-Prozessoren nicht ohne Weiteres erfüllt wird. So definierten beispielsweise im Jahr 2000 verbreitete Prozessoren der *Intel Pentium*-Familie insgesamt 17 sensitive, aber nicht privilegierte Befehle, wie beispielsweise POP, PUSH, CALL, JMP und RET [151]. Um die klassische x86-Architektur trotzdem virtualisieren zu können, haben sich verschiedene Strategien durchgesetzt.

Bei der *dynamischen Binärübersetzung (Dynamic Binary Translation, DBT)* wird der Binärcode des virtualisierten Gastsystems zur Laufzeit durch einen VMM in sogenannte *dynamische Basisblöcke* zerlegt, die nun analysiert, modifiziert und schließlich ausgeführt werden. Ein dynamischer Basisblock beginnt dabei stets mit der ersten, einer Verzweigung *(Branch)* oder

[2] Popek und Goldberg beziehen diese Forderung auf ein Modell eines Rechners, der unter anderem eine virtuelle Speicherverwaltung und eine Rechteverwaltung mit einem Supervisor- und einem User-Mode definiert. Dieses Modell ist an die Architektur sogenannter *Rechner der 3. Generation*, wie z.B. der *IBM 360*, angelehnt. Aufgrund seines hohen Abstraktionsgrades besitzt dieses Modell allerdings auch für heutige Rechnerarchitekturen eine ausreichende Gültigkeit.

einem Sprung *(Jump)* folgenden Anweisung und endet wiederum mit einem Verzweigungs- oder Sprungbefehl [161]. Im Gegensatz zu *statischen Basisblöcken* definieren potentielle Sprungziele im Code nicht zwingend den Beginn eines dynamischen Basisblocks, so dass je nach dynamischer Ausführung Codesequenzen Bestandteil mehrerer dynamischer Basisblöcke sein können [54]. Ein dynamischer Basisblock wird nun modifiziert, indem zunächst sensitive Instruktionen durch entsprechende Aufrufe von Hypervisor-Funktionen substituiert werden, um den entsprechenden, im User-Modus nicht erlaubten Befehl zu emulieren. Weiterhin wird vor der Ausführung die jeweils letzte Instruktion eines dynamischen Basisblocks durch einen Aufruf des Hypervisors ersetzt, um schließlich stets die Kontrolle an diesen zurückzugeben [168]. Zur Steigerung der Leistung werden bereits übersetzte Basisblöcke in einem Cache vorgehalten, um sie ohne erneute Übersetzung ausführen zu können. Das Konzept der dynamischen Binärübersetzung wird unter anderem in den Produkten von *VMware* eingesetzt [5].

Wie bereits zu Beginn des vorigen Abschnitts erwähnt, ist die Schnittstelle zwischen einer VM und einem VMM allerdings nicht zwingend eine Befehlssatzarchitektur. Dies trifft insbesondere für die Strategie der *Paravirtualisierung* zu, bei der vom Hypervisor eine Schnittstelle definiert wird, die aus dem virtualisierten Gastbetriebssystem heraus genutzt werden kann. Die Forderung nach einem Trap sensitiver Instruktionen wird nun umgangen, indem diese Instruktionen durch sogenannte *Hypercalls*, also explizite Funktionsaufrufe in den Hypervisor, ersetzt werden. Da derartige Substitutionen statisch im Quellcode des Betriebssystems durchgeführt werden müssen, ist diese Alternative auf quelloffene oder bereits vom Hersteller adaptierte Betriebssysteme beschränkt. Zugleich müssen die Adaptionen jedoch auf den Betriebssystemcode beschränkt bleiben, um beliebige, gegebenenfalls nur als Binärcode verfügbare Applikationen ausführen zu können. Dies wiederum erfordert die Paravirtualisierung aller Merkmale einer realen Hardware, die durch existierende Binärschnittstellen genutzt werden können [17].

Für beide der zuvor genannten Strategien stellt die Virtualisierung der Speicherverwaltung eine Herausforderung dar. Zum einen müssen den Gastsystemen selbst unterschiedliche virtuelle Speicherbereiche zugewiesen werden, zum anderen soll auch innerhalb der Gastsysteme eine virtuelle Speicherverwaltung unterstützt werden. Dies erfordert eine zweistufige Adressübersetzung von den virtuellen Adressen des Gastsystems *(Guest Virtual Address, GVA)* über die physikalischen Adressen des Gastsystems *(Guest Physical Address, GPA)*, die den virtuellen Adressen des Hypervisors entsprechen, in die physikalischen Adressen der Maschine *(Host Physical Address, HPA)*. Da den Gastsystemen ein direkter Zugriff auf die zur Adressübersetzung genutzten *Seitentabellen (Page Tables)* des Hypervisors nicht erlaubt werden kann, erfolgt die Übersetzung entsprechend Abbildung 2.11 meist mittels vom Hypervisor verwalteter sogenannter *Schattentabellen (Shadow Page Tables)*. Diese Tabellen konsolidieren die zweistufige Übersetzung und werden schließlich von der Hardware-*MMU* zur Adressübersetzung genutzt [74]. Dies hat allerdings zur Konsequenz, dass jede Modifikation, die ein Gastsystem an seinen Seitentabellen vornimmt, eine VMM-Intervention erfordert, um diese Adaption in den Schattentabellen zu reflektieren.

Über eine lange Zeit hinweg stellten die dynamische Binärübersetzung und die Paravirtualisierung die einzigen Strategien dar, um auf x86-Architekturen eine Systemvirtualisierung zu realisieren. Erst mit der Einführung einer *Hardware-Virtualisierungsunterstützung* durch *Intel* und *AMD* im Jahr 2005 wurde diese CPU-Architektur in einer Weise erweitert, die das zuvor beschriebene klassische Konzept des Trap-and-Emulate unterstützt [168]. Die entsprechenden Erweiterungen werden seitdem unter den Begriffen *Intel VT* [50, 125] respektive *AMD-V* [4]

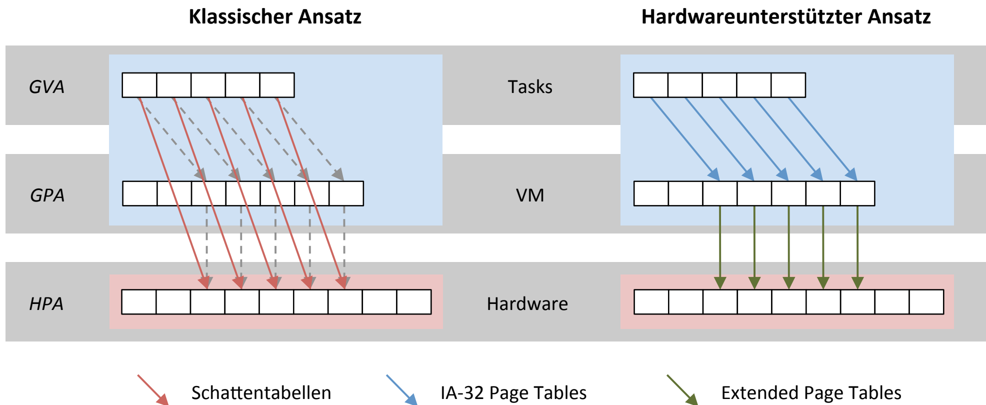


Abbildung 2.11: Adressübersetzung beim klassischen Ansatz und bei Verfügbarkeit einer Hardware-Virtualisierungsunterstützung

vermarktet. Da sich diese auf konzeptueller Ebene sehr ähnlich sind, beschränken sich die nachfolgenden Beschreibungen auf das Beispiel *Intel VT*. Um die Privilegien eines VMM von denen eines Gastsystems zu differenzieren, definiert *Intel VT* eine neue Dimension der Privilegierung mit den Modi *VMX Root* für die Ausführung des Hypervisors und *VMX Non-Root* für die Ausführung der Gastsysteme. Beide Modi umfassen wiederum die in Abschnitt 2.4.1 beschriebenen Ringe 0 bis 3, so dass die ursprüngliche Einordnung eines Gastbetriebssystems und seiner Applikationen in das Ringmodell beibehalten werden kann, allerdings im Modus *VMX Non-Root*. Ein weiterer Aspekt von *Intel VT* ist die Verfügbarkeit einer als *Virtual Machine Control Structure (VMCS)* bezeichneten Datenstruktur, die den Prozessorzustand sowohl für ein Gastsystem *(Guest State Area)* als auch für den Hypervisor *(Host State Area)* sichert und Konfigurations-Bits für das Verhalten der Virtualisierungsunterstützung definiert. So lässt sich in der *VMCS* unter anderem definieren, welche Instruktionen und Exceptions einen Wechsel des Ausführungsmodus von *VMX Non-Root* nach *VMX Root* auslösen *(VM Exit)*, um den Eingriff des Hypervisors zu ermöglichen, bevor ein Wechsel zurück nach *VMX Non-Root* erfolgt *(VM Entry)*. Weiterhin lassen sich mittels der *VMCS* externe Interrupts vom Hypervisor zur Behandlung in die VM weiterreichen, insbesondere dann, wenn die VM zum Zeitpunkt des Interrupts nicht ausgeführt wird. Dabei ist zu beachten, dass die *VMCS* die VM-spezifische Maskierung von Interrupts ermöglicht. Weiterhin ergänzt *Intel VT* ab der zweiten Generation die Virtualisierungsunterstützung um sogenannte *Extended Page Tables (EPTs)*. Diese übernehmen, wie in Abbildung 2.11 dargestellt, die Adressübersetzung von den physikalischen Adressen der Gastsysteme (GPA) in die physikalischen Maschinenadressen (HPA). Die Übersetzung der virtuellen (GVA) in die physikalischen Adressen des Gastsystems (GPA) wird dabei in den gewohnten Seitentabellen realisiert, so dass die Gastsysteme nun ihre Seitentabellen ohne eine per *VM Exit* angestoßene Intervention des Hypervisors selbst modifizieren und Zugriffsfehler *(Page Faults)* direkt behandeln können [125]. Eine weitere, von *Intel* unter der Bezeichnung *VT-d* vermarktete Funktionalität realisiert eine *I/O-MMU* in Hardware, mittels derer I/O-Geräte voneinander isolierten Domänen wie beispielsweise virtuellen Maschinen zugeordnet werden können. So lassen sich

unter anderem per *DMA*-Transfer realisierte unberechtigte Zugriffe einer VM auf den Speicher einer anderen VM verhindern. Als Konsequenz muss nicht mehr bei jedem DMA-Transfer mit I/O-Geräten mittels einer VMM-Intervention die Isolation der Gastsysteme sichergestellt werden. Stattdessen werden Zugriffe von I/O-Geräten auf Speicherbereiche virtueller Maschinen, für die keine Berechtigung vorliegt, durch die Hardware-I/O-MMU unterbunden. Darüber hinaus realisiert *Intel VT-d* ein sogenanntes *DMA-Remapping*, mittels dessen für DMA-Transfers eine Adressübersetzung von physikalischen Adressen eines Gastsystems (GPA) in physikalische Maschinenadressen (HPA) realisiert werden kann [1, 88].

Kapitel 3
Strategien der Multicore-Nutzung in der Steuerungstechnik

Für den Einsatz von Multicore-Architekturen in der Steuerungstechnik existieren vielfältige Motivationen, hinsichtlich derer nicht zuletzt unter dem Aspekt der jeweiligen technischen Realisierung eine Klassifikation in zwei Kategorien vorgenommen werden kann. Dabei handelt es sich um die Strategie der *Performanzsteigerung* und die Strategie der *Systemkonsolidierung*.

3.1 Performanzsteigerung

Eine wesentliche Motivation eines Einsatzes von Multicore-Prozessoren in der Steuerungstechnik besteht in der Steigerung der allgemeinen Leistungsfähigkeit der ausgeführten Firmware. Insbesondere in der Automatisierungstechnik existieren dabei vielfältige Aspekte, hinsichtlich derer sich die Performanz einer Firmware manifestiert.

3.1.1 Motivation einer Performanzsteigerung

Wie im vorigen Abschnitt beschrieben, definiert eine typische Steuerungs-Firmware diverse zyklisch unter Echtzeitbedingungen auszuführende Funktionen und eine Vielzahl azyklischer Hintergrunddienste mit weichen oder keinen Echtzeitanforderungen. Eine zu hohe Auslastung eines Multitasking-Systems führt generell dazu, dass Tasks niedriger und gegebenenfalls mittlerer Priorität verdrängt werden und somit nicht mehr ihre Zeitanforderungen erfüllen können. Dies kann sich beispielsweise bei einer numerischen Steuerung in Form eines regelmäßigen Leerlaufs des Puffers vorbereiteter NC-Sätze und somit in Verzögerungen bei der Werkstückbearbeitung äußern. Zudem kann es zu einer mangelnden Reaktivität der Steuerung bei der Bereitstellung von Statusinformationen oder Netzwerkdiensten kommen. Eine Ausführung der Firmware auf einer Multicore-CPU hat nun zunächst zur Folge, dass die zur Verfügung stehende freie Rechenzeit im System *(Idle Time)* insgesamt erhöht und im Fall einer Parallelisierung mit einer balancierten Lastverteilung auf jedem CPU-Kern maximiert wird. Als Konsequenz stehen für niedriger priorisierte Aufgaben mehr Reserven an Rechenzeit zur Verfügung und im Fall sporadischer Lastspitzen werden Systemlatenzen signifikant reduziert. Diese erste Stufe der Leistungssteigerung ist in Abbildung 3.1 dargestellt.

Nun ist allerdings zu beachten, dass eine automatisierungstechnische Steuerung in den meisten Fällen mit statisch definierten Konfigurationen, wie beispielsweise den Verarbeitungstakten oder der Anzahl der gesteuerten Achsen, betrieben wird. Die diesbezügliche Performanz der

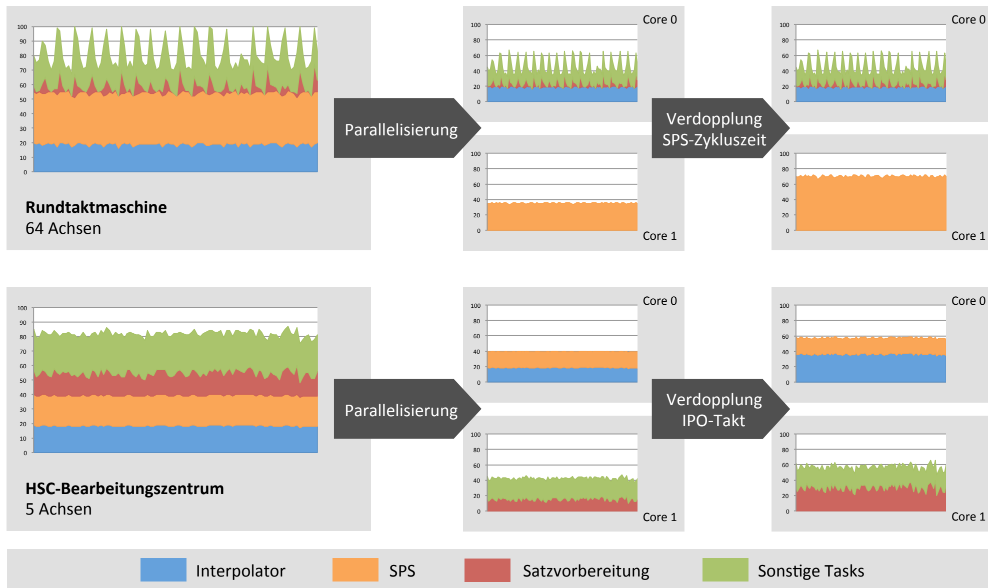


Abbildung 3.1: Szenarien der Performanzsteigerung einer Werkzeugmaschinensteuerung auf einer *Dual-Core*-CPU (nach [34]). Die Darstellung der parallelen Lastprofile hat lediglich illustrativen Charakter und basiert auf einer einfachen Extrapolation der Messwerte der nichtparallelen Ausführung.

Steuerung profitiert somit nicht unmittelbar von einer Parallelisierung. Aus diesem Grund muss, wie in Abbildung 3.1 als mögliche zweite Stufe der Leistungssteigerung dargestellt, eine explizite Anpassung der Steuerungskonfiguration erfolgen, die im Fall einer numerischen Steuerung unter anderem folgende Parameter zum Gegenstand haben kann [32, 34]:

- Von einer Reduzierung des Interpolationstakts (IPO-Takt) lässt sich auf verschiedene Arten profitieren: Bleibt die Vorschubgeschwindigkeit der Achsen konstant, kann insbesondere bei der Bearbeitung von Freiformflächen eine höhere Oberflächengüte des Werkstücks erzielt werden. Alternativ lässt sich bei einem geringeren IPO-Takt auch die Vorschubgeschwindigkeit erhöhen, was wiederum die Bearbeitungszeit eines Werkstücks verkürzt und so den Teiledurchsatz erhöht.
- Komplexere NC-Funktionen mit höherem Rechenaufwand wie zum Beispiel *NURBS (Non-Uniform Rational B-Splines)* [100] lassen sich nutzen, ohne dabei die Bearbeitungsgeschwindigkeit signifikant reduzieren zu müssen.
- Die Zahl der von einer Steuerung kontrollierbaren Achsen und Spindeln kann erhöht werden. Vor allem komplexe Rundtaktmaschinen besitzen nicht selten mehr als 100 Achsen und erfordern heute oft noch eine Vernetzung mehrerer Steuerungen. Mittels einer diesbezüglichen Konsolidierung können sich Kosteneinsparungen ebenso realisieren lassen wie Produktivitätssteigerungen, da eine Querkommunikation zwischen Steuerungen nicht mehr erforderlich ist und Latenzzeiten somit reduziert werden.

- Die Zykluszeit einer in die numerische Steuerung integrierten SPS kann reduziert werden. Dies verringert die Reaktionszeiten der Steuerung und steigert somit potentiell die Produktivität der Maschine.

Ein weiterer Aspekt eines Steuerungssystems ist dessen Funktionsumfang, also die Leistungsfähigkeit, die sich nicht durch die zuvor genannten Leistungsdaten manifestiert. Dies sind beispielsweise kontinuierlich oder aperiodisch neben der eigentlichen Maschinensteuerung ausgeführte Sicherheits-, Überwachungs- und Diagnosefunktionen oder intuitivere Methoden der Mensch-Maschine-Interaktion:

- Eine *sensorlose Online-Kollisionsüberwachung* verhindert in numerischen Steuerungen zur Laufzeit eine Kollision zwischen dem Werkzeug und anderen Objekten des Arbeitsraums, ohne dass dazu separate Geber erforderlich sind. Zu diesem Zweck werden relevante Objekte mittels geometrischer Körper approximiert und zur Laufzeit der Maschine drohende Kollisionen mittels geometrischer Berechnungen eines sogenannten *Kollisionsrechners* erkannt. Dieses Verfahren erzeugt allerdings einen nicht unerheblichen Rechenaufwand [181].
- Beim *Condition Monitoring* wird mittels einer Aufzeichnung und Auswertung von Maschinenstatusinformationen, wie beispielsweise Vibrationsfrequenzen, versucht, drohende Ausfälle von Maschinenkomponenten zu prädizieren, um durch einen rechtzeitigen Austausch betroffener Komponenten Stillstandzeiten der Maschine zu reduzieren. Zur Auswertung der Daten werden dabei unter anderem rechenzeitintensive Algorithmen wie *Fourier-Analysen* angewandt [144].
- Eine zunehmend vernetzte Fertigung erfordert umfassende *Security-Mechanismen* zur Verhinderung unbefugter externer Zugriffe auf produktionsnahe Anlagen. Neben Authentifizierungsmechanismen zählen hierzu auch rechenintensivere Maßnahmen wie die Datenverschlüsselung oder eine detaillierte Protokollierung und Analyse von Ereignissen.
- Bislang nur prototypisch angewandte *biometrische Methoden* der Mensch-Maschine-Interaktion wie beispielsweise die Programmierung eines Roboters mittels Sprache und Gesten erfordern anspruchsvolle Algorithmen der Audio- und Bildanalyse [128].

Die eigentlichen Steuerungsaufgaben lasten die heute in CNCs eingesetzten Prozessoren insbesondere bei komplexen Maschinen bereits zu einem hohen Grad aus, so dass derartige Funktionen höchstens auf einem separaten Prozessor ausgeführt werden können. Dies erzeugt allerdings nicht nur zusätzliche Kosten, sondern erfordert auch eine datenintensive Kommunikation zwischen eigenständigen Systemen. Multicore-Prozessoren bieten nun die Möglichkeit, die zusätzliche Idle-Zeit des Systems für derartige Aufgaben zu nutzen.

3.1.2 Firmware-Parallelisierung in der Steuerungstechnik

Von den in Abschnitt 2.5.1 dargestellten Alternativen zur Entwicklung einer parallelen Software stellt im Fall der komplexen Firmware automatisierungstechnischer Systeme die Parallelisierung eines bestehenden Systems in der Regel die einzige rationale Alternative dar. Dabei gilt es, das bestehende System so anzupassen, dass unter möglichst geringen Entwicklungsaufwänden eine ausreichend hohe Leistungssteigerung erzielt wird. Für ein komplexes und anspruchsvolles

System wie eine automatisierungstechnische Steuerung, deren Quellcode nicht selten mehrere Millionen Programmzeilen umfasst, ist jedoch auch eine solche Adaption mit signifikanten Entwicklungsaufwänden und Herausforderungen verbunden. Dies ist nicht zuletzt dadurch begründet, dass das System höchsten Anforderungen hinsichtlich Echtzeitfähigkeit und Zuverlässigkeit genügen muss.

Wie in Abschnitt 2.3.2 beschrieben, besteht eine moderne Automatisierungs-Firmware in der Regel aus einer Vielzahl von Tasks unterschiedlicher Priorität, welche die einzelnen Funktionsbereiche implementieren. Somit besteht der erste Schritt einer Parallelisierung in der verteilten Ausführung dieser Tasks auf den Kernen einer Multicore-CPU. Dies ist allerdings nicht zwingend ausreichend, so dass für ausgewählte Tasks eine Dekomposition erwogen werden sollte. Resultat der Dekomposition ist dann eine Menge von Tasks, die in kooperativer Weise die Funktionalität der ursprünglichen Task implementieren, allerdings in einer parallel auf mehreren CPU-Kernen ausführbaren Weise. Eine derartige Dekomposition empfiehlt sich im Umfeld der Steuerungstechnik vor allem für Tasks, die einer Auswahl der folgenden Bedingungen genügen:

- Die Task beansprucht einen signifikanten Anteil der Rechenzeit des Systems und kann somit die Möglichkeiten zur gleichmäßigen Verteilung der Rechenlast auf den CPU-Kernen einschränken. Eine Dekomposition und anschließende Verteilung der resultierenden Tasks würde somit zu einer Performanzsteigerung des Systems führen.
- Die Task ist Bestandteil des zyklischen Echtzeitteils der Firmware, so dass ihre Ausführungsdauer spezifische Leistungsdaten der Steuerung, wie beispielsweise den Interpolationstakt oder die SPS-Zykluszeit, unmittelbar beeinflusst. Eine Parallelisierung würde somit die Möglichkeit zur Steigerung der diesbezüglichen Systemperformanz bieten, sofern dies aufgrund der übrigen Systemauslastung möglich ist.
- Die Task zeichnet sich durch einen hohen Grad an Datenparallelität aus, indem identische Berechnungen auf disjunkten Daten ausgeführt werden. In nicht implizit parallelen Programmiersprachen liegt eine Datenparallelität häufig in Form von Schleifen vor.
- Der Codeumfang der Task bewegt sich in einem Rahmen, der eine detaillierte und umfassende Analyse sowie als Konsequenz auch eine Dekomposition unter vertretbarem Aufwand ermöglicht.

Während die eigentliche Task-Dekomposition ohnehin einmalig zum Entwurfszeitpunkt durchgeführt wird, gibt es für das Scheduling der resultierenden Tasks auf den Kernen der CPU die in Abschnitt 2.4.2 beschriebenen Alternativen. Zu den wichtigsten Zielkriterien einer Parallelisierung im Umfeld automatisierungstechnischer Steuerungen zählt dabei die Performanz und der Determinismus der Ausführung, so dass sich hier ein partitioniertes Scheduling empfiehlt. Entsprechend Abschnitt 2.4.2 besteht der wesentliche Nachteil eines partitionierten Schedulings allerdings darin, dass die Task-Allokation zur Laufzeit nicht an die Lastdynamik des Systems angepasst wird und so die CPU-Kerne gegebenenfalls nicht zu jedem Zeitpunkt optimal genutzt werden. Es ist allerdings zu vermuten, dass dieser Effekt in der Automatisierungstechnik weitestgehend vernachlässigt werden kann, da sich das Lastprofil einer Maschinensteuerung aufgrund von deren zyklischer Arbeitsweise im Zeitverlauf nicht signifikant ändert. Dieser Verdacht lässt sich exemplarisch anhand umfangreicher Laufzeitanalysen der CNC-Plattform *IndraMotion MTX* [24] von *Rexroth*, die als repräsentativ für gängige Automatisierungslösungen angesehen werden kann, bestätigen. Die dabei gewonnenen Ergebnisse zeigt Abbildung 3.2.

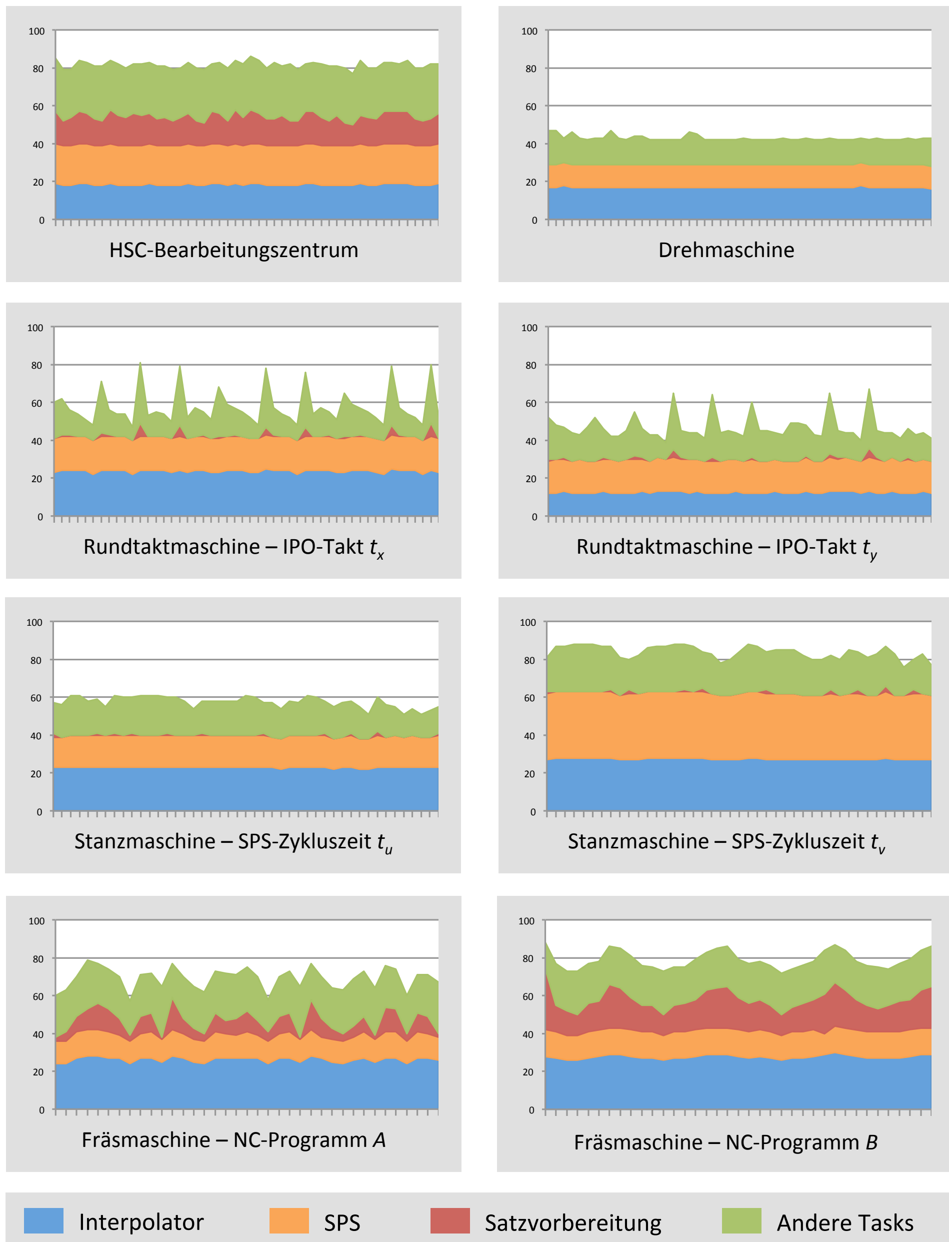


Abbildung 3.2: Anteile der Funktionsbereiche an der CPU-Last über einen vom Maschinentyp abhängigen Zeitraum zwischen 74 s und 100 s bei einem Abtastintervall von 2 s. Jeder Wert entspricht dem Lastanteil des entsprechenden Funktionsbereichs in dem seit der jeweils letzten Abtastung verstrichenen Zeitintervall.

Alle hier beobachteten Lastanteile der Funktionsbereiche sind entweder wie im Fall der Dreh- und Stanzmaschine nur sehr geringen Schwankungen im einstelligen Prozentbereich unterworfen oder schwanken mit Zyklen von wenigen Sekunden innerhalb fester Intervalle. Der letztere Fall trifft insbesondere für die Lastprofile der Rundtaktmaschine zu, bei denen wiederkehrende Lastspitzen im Rotationstakt des Schalttellers auftreten.

Um darüber hinaus bei der Parallelisierung von dem in Abschnitt 2.4.2 beschriebenen geringeren Implementierungsaufwand eines partitionierten Schedulings profitieren zu können, muss die Task-Allokation bereits zum Entwurfszeitpunkt der Firmware statisch definiert werden. Das Problem besteht nun darin, dass die zuvor geschilderten und in Abbildung 3.2 dargestellten Laufzeitanalysen signifikante Korrelationen zwischen dem jeweiligen Lastprofil der Firmware und folgenden Faktoren gezeigt haben:

- Einen wesentlichen Einfluss auf das Lastprofil hat der Typ der gesteuerten Maschine und deren konkrete Ausprägung, wie beispielsweise die Anzahl der Achsen oder die Komplexität der NC-Programme. So steuern Rundtaktmaschinen häufig mehr als 100 Achsen und weisen demzufolge eine hohe Auslastung seitens des Interpolators auf, da dieser in jedem Interpolationstakt für alle Achsen neue Sollwerte berechnen muss. Im Gegensatz dazu wird bei einer 5-Achs-Fräsbearbeitung von Freiformflächen ein signifikanter Anteil der Rechenzeit für die Satzaufbereitung und die Satzvorbereitung aufgewendet. Dies ist erforderlich, da die Werkstückkontur hier entweder mittels komplexer Splines beschrieben oder durch zahlreiche kurze Geraden- und Kreissegmente approximiert wird [181].
- Anpassungen der Zykluszeit bei getakteten Berechnungen der Steuerung, wie beispielsweise dem Interpolator oder der SPS, führen zu einer nahezu proportionalen Änderung des Lastanteils der jeweiligen Funktionsbereiche.
- Das ausgeführte Steuerungsprogramm hat einen nicht zu vernachlässigenden Einfluss auf das Lastprofil: Je größer beispielsweise bei einer numerischen Steuerung die Anzahl der innerhalb eines definierten Zeitintervalls zu verarbeitenden NC-Sätze ist, desto größer ist der Lastanteil der Satzvorbereitung, während unter anderem die SPS oder der Interpolator hiervon nicht betroffen sind.

Als Konsequenz ist es bei der Parallelisierung automatisierungstechnischer Steuerungen mittels eines zum Entwurfszeitpunkt definierten partitionierten Schedulings von großer Relevanz, eine Task-Allokation zu wählen, die sich unter einer Vielzahl potentieller Anwendungsfälle der Steuerung durch eine möglichst hohe Performanz auszeichnet. Dieses Vorgehen beinhaltet jedoch zahlreiche Herausforderungen, denen es im Rahmen dieser Arbeit in geeigneter Weise zu begegnen gilt.

3.2 Systemkonsolidierung

Neben der im vorherigen Abschnitt beschriebenen qualitativen und quantitativen Performanzsteigerung können Multicore-CPUs auch genutzt werden, um zuvor auf separater Hardware ausgeführte Funktionsbereiche zu konsolidieren. Auf diese Weise kann die Anzahl der eingesetzten Hardware-Komponenten reduziert werden, so dass sich unter Umständen nicht nur bei den Anschaffungs-, sondern auch bei den Betriebskosten der Hardware-Infrastruktur deutliche Einsparungen erzielen lassen. Ein derartiges Nutzungsszenario rückt in den Bereich des Möglichen,

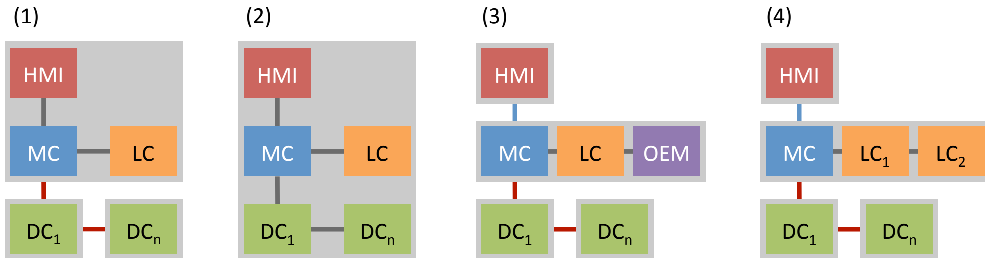


Abbildung 3.3: Konsolidierungsszenarien der Funktionsbereiche Benutzerschnittstelle (HMI), Bewegungssteuerung (MC), speicherprogrammierbare Steuerung (LC), Antriebsregler (DC) und OEM-Firmware (OEM)

da die echt parallelen Ausführungseinheiten von Multicore-Architekturen erstmals die Möglichkeit bieten, mehrere der in Abschnitt 2.3.1 beschriebenen Funktionsbereiche einer Steuerung auf einem Prozessor zu integrieren und trotzdem einen bisher nicht erreichten Grad an temporaler Isolation zu wahren. Abbildung 3.3 skizziert eine Auswahl möglicher Konsolidierungsszenarien der Funktionsbereiche Benutzerschnittstelle (HMI), Bewegungssteuerung (MC), speicherprogrammierbare Steuerung (LC), Antriebsregler (DC) und OEM-Firmware (OEM), die sich wie folgt darstellen [34]:

- *Szenario 1: Integration der Benutzerschnittstelle.* Bei einer Integration der Funktionsbereiche MC, LC und HMI auf einer CPU und somit einer gemeinsamen Hardware kann der separate HMI-PC auf ein Display mit Eingabeperipherie wie einer Tastatur und *Softkeys* reduziert werden.
- *Szenario 2: Integration von Antriebsreglern.* Eine Integration der Funktionsbereiche MC, LC, HMI und DCs ermöglicht für eine begrenzte Anzahl an Achsen den Verzicht auf separate Antriebsregler. Dies kann vor allem für kompakte Maschinen mit einer geringen Anzahl an Achsen deutliche Einsparungen bei den Hardware-Kosten generieren.
- *Szenario 3: Integration einer OEM-Firmware.* Manche Maschinenhersteller erweitern die Funktionalität einer Steuerung mittels einer eigenen *OEM-Firmware*. Seitens des Steuerungsherstellers besteht dabei die Anforderung, diesen Code von der eigenen Firmware zu isolieren und Interaktionen auf definierte Schnittstellen zu beschränken. Mittels einer Integration dieser OEM-Firmware unter einer ausreichenden Isolation ist es möglich, eine Offenheit der Steuerung für OEM-Funktionalität zu realisieren, ohne die Komplexität der Steuerungs-Firmware weiter zu erhöhen und deren Ausführungsdeterminismus zu beeinträchtigen. Dies ist insbesondere vor dem Hintergrund wachsender Anforderungen an die *Safety* und *Security* automatisierungstechnischer Systeme von Relevanz[3].

[3] In der deutschen Sprache hat sich sowohl für die *Safety* als auch für die *Security* der unspezifische Begriff der *Sicherheit* etabliert. Insbesondere im Kontext der Automatisierungstechnik wird unter der Safety allerdings der Schutz von Mensch und Umwelt gegenüber unbeabsichtigten Fehlfunktionen eines Systems verstanden, während sich die Security eines Systems auf dessen Schutz gegen beabsichtigte Angriffe von außen bezieht. Der Versuch, diese Differenzierung durch die Begriffe der *Betriebssicherheit* und der *Angriffssicherheit* vorzunehmen, ist bislang auf geringe Akzeptanz gestoßen, so dass in dieser Arbeit die englischen Begriffe verwendet werden.

- *Szenario 4: Integration einer Safety-SPS.* Die Safety-Eigenschaft eines Systems impliziert insbesondere in der Automatisierungstechnik im Allgemeinen nicht die Forderung nach der Vermeidung, sondern nach der zuverlässigen Erkennung von Fehlern. Um dies zu gewährleisten, stellt eine redundante Auslegung mittels einer zweikanaligen Ausführung den derzeitigen Stand der Technik bei sicherheitskritischen Steuerungskomponenten wie einer Safety-SPS dar. Diese Redundanz wird üblicherweise nicht nur für die Software, sondern auch für die Hardware, die in Form zweier separater Prozessoren zu realisieren ist, gefordert. Mittels einer Multicore-CPU ist es nun prinzipiell denkbar, die redundanten Kanäle unter einer entsprechenden Isolation auf unterschiedlichen CPU-Kernen zu integrieren und so die Hardware-Kosten zu reduzieren. Zu beachten ist allerdings, dass in einem solchen Szenario der Prozessor beispielsweise mit dem Taktgeber, der Spannungsversorgung oder diversen Bussen über diverse nicht duplizierte Ressourcen verfügt und auf diese Weise einen *Single-Point-of-Failure (SPoF)* darstellt. Somit gilt es, geeignete Strategien zu entwickeln, die auch ohne eine redundante Auslegung entsprechender Hardware-Ressourcen eine sichere Fehlererkennung und somit eine Safety-Zertifizierung ermöglichen.

Eine Möglichkeit, derartige Konsolidierungsszenarien zu realisieren, stellt der Einsatz einer entsprechenden Basis-Software dar, die unter anderem die Ressourcenarbitrierung und die Isolation zwischen den integrierten Funktionsbereichen gewährleistet. Entsprechende Lösungen präsentiert und bewertet Kapitel 4.

Kapitel 4

Bewertung der Systemkonsolidierung in der Steuerungstechnik

In Kapitel 3.2 wurde eine Systemkonsolidierung in Form einer Integration ausgewählter Funktionsbereiche einer automatisierungstechnischen Steuerung als relevantes Nutzungsszenario für Multicore-Prozessoren in der Steuerungstechnik abgeleitet. Dies motiviert die im nachfolgenden Kapitel vorgenommene Bewertung von Strategien, die eine derartige Integration ermöglichen, hinsichtlich der spezifischen Anforderungen dieser Domäne [33].

4.1 Ableitung und Abgrenzung der Vorgehensweise

Generell steht ein Steuerungshersteller bei der Realisierung einer Systemkonsolidierung vor der Entscheidung, die dafür erforderliche Basis-Software selbst zu entwickeln oder eine am Markt verfügbare Standardlösung *(Commercial Off-The-Shelf, COTS)* [170] zuzukaufen. Bei einer solchen *Make-or-Buy*-Entscheidung sind eine Vielzahl verschiedener Faktoren aus dem technischen und nichttechnischen Bereich zu berücksichtigen. Dabei sind beispielsweise die im Fall der einzelnen Alternativen anfallenden direkten und indirekten Kosten ebenso zu evaluieren wie die Zeitdauer bis zur Fertigstellung respektive Marktverfügbarkeit einer geeigneten Lösung. Bei den in der Automatisierungstechnik eingesetzten Betriebssystemen verstärkte sich in den letzten Jahren der Trend, herstellerspezifische Entwicklungen durch am Markt verfügbare Standardlösungen zu ersetzen [181]. Als Konsequenz ist zu erwarten, dass Hersteller industrieller Automatisierungstechnik auch bei der Systemkonsolidierung auf Standard-Software zurückgreifen, statt sich die zur Entwicklung einer derartigen Lösung erforderliche Expertise im eigenen Haus aufzubauen. Diese Tatsache begründet wiederum die Entscheidung, die nachfolgende Evaluation anhand zweier aktuell am Markt verfügbarer Konsolidierungslösungen vorzunehmen, wobei zugleich so weit wie möglich von der konkreten Implementierung des jeweiligen Produkts abstrahiert werden soll.

4.2 Anforderungen an eine Systemkonsolidierung

Während Abschnitt 2.10.1 die von Gerald J. Popek und Robert P. Goldberg allgemein für eine Systemvirtualisierung definierten Kriterien beschreibt, gilt es nun, diese auf eine Systemkonsolidierung im Steuerungsumfeld zu übertragen. Dies ist nötig, da sich die Anforderungen, die im Umfeld automatisierungstechnischer Steuerungen an eine Systemkonsolidierung gestellt werden, signifikant von denen im Bereich der Server-Konsolidierung oder Desktop-Virtualisierung

unterscheiden. In erster Linie ist dies der Tatsache geschuldet, dass eine Systemkonsolidierung in der industriellen Automatisierung mit Standard- und Echtzeitbetriebssystemen stark heterogene Betriebssystemtypen integriert. Daraus ergeben sich folgende Anforderungen:

- Eine Systemkonsolidierung darf den *Zeitdeterminismus* der integrierten Echtzeitbetriebssysteme nicht auf unzulässige Weise, wie beispielsweise durch einen zu hohen *Jitter* in der Ausführungsdauer von Programmsequenzen, beeinträchtigen. Diese Forderung bezieht sich neben der CPU-internen Verarbeitung auch auf die Interaktion mit der Systemperipherie, wie beispielsweise dem Speicher oder I/O-Ressourcen.
- Die *Reaktivität* der integrierten Echtzeitbetriebssysteme in Form von deren Latenz bei der Reaktion auf externe Stimuli darf auch im Kontext einer Systemkonsolidierung nicht auf unzulässige Weise beeinträchtigt werden.

Zu beachten ist, dass beide Forderungen bezüglich der beim Determinismus und der Reaktivität tolerierbaren Beeinträchtigungen bewusst vage formuliert sind, da diesbezügliche quantitative Schranken von zahlreichen Randbedingungen abhängig sind, die nicht zuletzt auch von den jeweiligen Gastsystemen und dem konkreten Anwendungsfall der Steuerung determiniert werden. Unabhängig davon gilt jedoch, dass die im Umfeld der industriellen Automatisierung geltenden Anforderungen an das temporale Verhalten der integrierten Betriebssysteme eine signifikante Verschärfung der Forderungen von Popek und Goldberg [140] darstellen, deren Äquivalenzforderung Abweichungen im Zeitverhalten explizit toleriert.

Generell lassen sich in einem konsolidierten Szenario zwei wesentliche Arten des Einflusses auf die zuvor genannten Aspekte des temporalen Verhaltens differenzieren. Zunächst sind hier Einflüsse zu nennen, die als Folge der Konkurrenz mehrerer integrierter Gastsysteme um nicht exklusiv verfügbare Betriebsmittel des Systems, wie beispielsweise Hardware-Ressourcen oder Dienste der Konsolidierungslösung, verursacht werden. Diese Problematik wird durch die echt parallele Ausführung von Gastsystemen auf den Kernen einer Multicore-CPU noch verschärft, da hier im Gegensatz zu einer Singlecore-CPU keine durch den Hypervisor-Scheduler implizit erzwungene Serialisierung der Betriebsmittelzugriffe der virtuellen Maschinen stattfindet. Eine zweite Quelle potentieller Einflüsse auf das temporale Verhalten sind Maßnahmen, die von der Hardware und der Konsolidierungslösung ergriffen werden, um Gastsysteme gegenüber Fehlfunktionen oder unberechtigten Zugriffen anderer Gastsysteme zu isolieren. Dabei kann es sich um das Einfrieren oder den Absturz eines Gastsystems ebenso handeln wie um einen gezielten Angriff von Seiten eines der integrierten Systeme. Die daraus resultierende Forderung nach der *Zuverlässigkeit und Sicherheit* der integrierten Ausführung ist im automatisierungstechnischen Umfeld von wesentlich größerer Relevanz als bei der Integration von Standardbetriebssystemen, da eine kompromittierte Isolation weitreichende Folgen für Mensch und Maschine haben kann. Dies betrifft vor allem die Integration von Standardbetriebssystemen, da diese aufgrund ihrer Komplexität, der Möglichkeit zum dynamischen Laden von Programmen aus unbekannten Quellen und der Gefahr von Fehlbedienungen durch den Anwender das potentiell größte Risiko darstellen. Im Steuerungsumfeld versuchte man deshalb bislang, die Sicherheit und Zuverlässigkeit des Gesamtsystems durch den Einsatz dedizierter CPUs für verschiedene Funktionsbereiche der Steuerung zu erhöhen [181], um so die Anzahl von Berührungspunkten der Systeme, die entsprechende Schutzmaßnahmen erfordern, zu reduzieren.

Darüber hinaus lassen sich weitere Kriterien definieren, deren Berücksichtigung sich bei der Wahl einer Konsolidierungslösung im Steuerungsumfeld empfiehlt:

- *Ressourcenpartitionierung:* Um die im Kontext des temporalen Verhaltens als kritisch bewerteten konkurrierenden Zugriffe mehrerer Gastsysteme auf gemeinsame Systemressourcen zu reduzieren, sollte in einem Konsolidierungsszenario die Verfügbarkeit insbesondere der Ressourcen, die für die Echtzeitfähigkeit eines Gastsystems besonders relevant sind, maximiert werden können. Dazu sollte der Hypervisor die Möglichkeit bieten, ausgewählte Betriebsmittel den Gastbetriebssystemen exklusiv zur Verfügung stellen zu können. Dies gilt für die Kerne einer Multicore-CPU ebenso wie für I/O-Ressourcen, wie beispielsweise den Feldbus-Controller. Eine Partitionierung beeinträchtigt zwar in der Regel die effiziente Auslastung der Systemressourcen, erhöht aber zugleich die zeitliche Entkopplung der Gastsysteme [91]. Zudem kann eine Ressourcenpartitionierung das System-Debugging erleichtern, da deutlich weniger Seiteneffekte zwischen den Gastsystemen auftreten.
- *Ressourcenvirtualisierung:* Da im Bereich automatisierungstechnischer Steuerungen aus Kostengründen statt spezifischer Hardware immer häufiger Industrie-PCs eingesetzt werden, haben Ressourcenbeschränkungen hier eine deutlich geringere Relevanz als in typischen eingebetteten Systemen. Mit Blick auf eine effiziente Ressourcennutzung, die in der Regel durch die Forderung nach reduzierten Komponentenkosten und einem geringen Energieverbrauch motiviert ist, sollte eine Systemkonsolidierung trotzdem für geeignete Ressourcen die Möglichkeit der Virtualisierung bieten, um diese effektiv durch mehrere Gastbetriebssysteme nutzen zu können. Vertreter dieser Kategorie sind beispielsweise der Netzwerkadapter, der Feldbus-Controller [156] oder die Kerne einer Multicore-CPU. Im letzteren Fall sind entsprechende Scheduling-Verfahren erforderlich, die auch bei einer Koexistenz mehrerer Betriebssysteme auf einem CPU-Kern deren jeweilige zeitliche Anforderungen erfüllen [91].
- *Multicore-Ausführungsumgebungen:* Um auch in einem konsolidierten Szenario die Ausführung von SMP-Betriebssystemen zu ermöglichen, muss der Hypervisor die Möglichkeit bieten, einem Gastsystem mehrere Kerne einer Multicore-CPU zuzuweisen. Nur auf diese Weise ist sichergestellt, dass mit einer steigenden Zahl der verfügbaren CPU-Kerne trotz einer Systemkonsolidierung langfristig Performanzsteigerungen durch die parallelisierte Ausführung bestehender Applikationen auf mehreren Kernen möglich sind.
- *Gastsystemflexibilität:* Die im Steuerungsumfeld eingesetzten Echtzeitbetriebssysteme sind zwar in der Regel COTS-Lösungen [181], aber dennoch vielfältig. Im Bereich der Benutzerschnittstelle (HMI) werden hingegen zunehmend vollwertige Standardbetriebssysteme wie *Microsoft Windows* eingesetzt [181]. Da die Wahl einer Konsolidierungslösung aus ökonomischen Gründen meist eine langfristige Entscheidung darstellt, sollte eine solche bezüglich der unterstützten Gastsysteme eine hohe Flexibilität und Zukunftssicherheit gewährleisten. Dies gilt insbesondere für die Unterstützung quelloffener ebenso wie nicht quelloffener Betriebssysteme oder sogenannter *Bare Metal Applications* ohne darunter liegendes Betriebssystem.
- *Inter-OS-Kommunikation:* Wie in Abschnitt 3.2 dargestellt, sind die Funktionsbereiche einer automatisierungstechnischen Steuerung in der Regel durch einen hohen Grad an

Interaktion und demzufolge ein hohes Maß an Inter-OS-Kommunikation geprägt. Basierend auf dem gemeinsamen Speicher als physikalisches Übertragungsmedium einer Inter-OS-Kommunikation im konsolidierten Fall stellen Hypervisor-Lösungen im Echtzeitbereich meist bereits Protokolle verschiedener Schichten des *ISO/OSI-Referenzmodells* [134] zur Kommunikation zwischen den Gastsystemen zur Verfügung. Zum einen muss der Hypervisor dabei auf Basis der vordefinierten Schichten die Möglichkeit zur Implementierung eigener Protokolle, wie beispielsweise Feldbusprotokolle, bieten, zum anderen müssen die bereits implementierten Schichten für einen Einsatz im Steuerungsumfeld geeignet sein. Je nach Kommunikationstyp wird dabei entweder ein hoher Durchsatz bei gleichzeitig gelockerten Zeitbedingungen oder aber ein deterministisches zeitliches Verhalten mit geringen Latenzen und hoher Zuverlässigkeit gefordert.

- *Ressourcenanforderungen:* Auch wenn automatisierungstechnische Steuerungen in der Regel nicht durch strikte Ressourcenbeschränkungen geprägt sind, unterscheidet sich ein im industriellen Umfeld eingesetzter PC in wesentlichen Aspekten von einem Standard-PC, um den Anforderungen an Temperatur-, Vibrations- und elektromagnetischer Verträglichkeit gewachsen zu sein [181]. Hierzu zählt der Verzicht auf verschleißanfällige Lüfter und die damit erforderliche Reduzierung der *Thermal Design Power*, beispielsweise mittels einer Beschränkung der CPU-Taktrate, ebenso wie der Einsatz sogenannter *Solid State Disks* ohne mechanische Komponenten. Die Konsequenz dieser Maßnahmen für eine Konsolidierungslösung ist somit die Forderung nach einer Funktionsfähigkeit unter gegebenenfalls moderaten Einschränkungen hinsichtlich der verfügbaren Rechen- und Speicherressourcen.
- *Kosten:* Da Kosteneinsparungen die maßgebliche Motivation einer Systemkonsolidierung im Steuerungsumfeld darstellen, dürfen die durch den Einsatz einer Konsolidierungslösung verursachten direkten und indirekten Kosten über die Nutzungsdauer des Systems *(Total Cost of Ownership, TCO)* [163] nicht die eines Einsatzes dedizierter Hardware-Komponenten für die einzelnen Funktionsbereiche übersteigen. Während sich die direkten Kosten im Wesentlichen als Beschaffungs- und Lizenzkosten für das jeweilige Produkt manifestieren, werden indirekte Kosten wie Entwicklungs- und Wartungskosten vor allem dann generiert, wenn das eingesetzte Produkt die Systemkomplexität im Vergleich zu einer nicht konsolidierten Ausführung signifikant erhöht.

Zusammenfassend steht eine Konsolidierungslösung im Steuerungsumfeld somit vor der Herausforderung, Ausführungsumgebungen zur Koexistenz heterogener Gastsysteme auf einer Hardware-Plattform unter hohen Anforderungen an das temporale Verhalten bereitzustellen. Zugleich ist ein hohes Maß an Zuverlässigkeit und Isolation zwischen den Systemen ebenso gefordert wie die Unterstützung flexibler Nutzungsszenarien der Hardware-Ressourcen.

4.3 Strategien zur Systemkonsolidierung

Insbesondere mit der wachsenden Verbreitung von Multicore-Prozessoren im Umfeld echtzeitkritischer Systeme hat auch die Zahl der Produkte, die eine für diese Domäne geeignete Systemkonsolidierung ermöglichen sollen, in den letzten Jahren stark zugenommen. Zu diesen häufig als *Embedded Hypervisor* klassifizierten Lösungen sind unter anderem der *Green*

Hills INTEGRITY Multivisor [67], *Sysgo PikeOS* [166], *LynxSecure* [111], der *Wind River Hypervisor* [185] und der *Real-Time Hypervisor* [147] zu zählen. Diese Lösungen unterscheiden sich allerdings zum Teil signifikant hinsichtlich der zugrunde liegenden Architektur, weshalb der nachfolgende Abschnitt unterschiedliche Strategien der Systemkonsolidierung am Beispiel zweier derartiger COTS-Lösungen vorstellt.

4.3.1 Virtualisierter Konsolidierungsansatz

Die erste der in dieser Evaluation betrachteten Konsolidierungslösungen verfolgt für alle Gastsysteme konsequent die Strategie der Systemvirtualisierung[4]. Der Hypervisor selbst basiert dabei auf einem nur wesentliche Funktionen umfassenden *Microkernel*, während jegliche übrige Funktionalität außerhalb des *Kernels* implementiert ist. Motivation eines solchen Konzepts ist nach [168] im Allgemeinen eine höhere Zuverlässigkeit infolge einer Dekomposition in kleinere, definierte Module, von denen ausschließlich diejenigen in der höchsten Privilegierungsstufe der CPU ausgeführt werden, welche die entsprechenden Rechte zwingend erfordern. Um zudem die Skalierbarkeit des Hypervisors für Multicore-CPUs zu erhöhen, basiert der virtualisierte Ansatz auf einem verteilten Kernel. So wird auf jedem CPU-Kern ein Kernel-Modul ausgeführt, das die diesen Kern betreffenden Aufgaben des Hypervisors wahrnimmt.

Für die Ausführung der Gastsysteme stellt der virtualisierte Konsolidierungsansatz deprivilegierte Ausführungsumgebungen in Form virtueller Maschinen (VMs) zur Verfügung, die vom Hypervisor verwaltet werden. Die Zuordnung der Gastsysteme zu den virtuellen Maschinen erfolgt über die Hypervisor-Konfiguration. Ein Gastsystem kann entweder ein Betriebssystem zusammen mit den darin ausgeführten Applikationen oder aber auch eine Bare Metal Application sein, die auf ein Betriebssystem verzichtet. Jeder virtuellen Maschine wird nun in einem weiteren Schritt eine Teilmenge der verfügbaren CPU-Kerne zugewiesen, so dass auch SMP-Gastsysteme unter dem virtualisierten Konsolidierungsansatz ausführbar sind. Dabei lassen sich zwei Modi unterschieden: Im einfachsten Fall wird jedem Kern maximal eine virtuelle Maschine zugeordnet, so dass eine statische Partitionierung der CPU-Kerne unter den Gastsystemen erfolgt. Alternativ dazu können einem CPU-Kern auch mehr als eine virtuelle Maschine und somit mehrere Gastsysteme zugeordnet werden, so dass der Kern selbst virtualisiert wird. Dies hat zur Konsequenz, dass ein Scheduling virtueller Maschinen geleistet werden muss. Zur Auswahl stehen dabei ein statisches Partitionen-Scheduling mit zyklischen, spezifisch für jede virtuelle Maschine definierbaren Zeitschlitzen und ein Prioritäten-Scheduling auf Basis von Prioritäten, die den virtuellen Maschinen zugeordnet werden.

Eine der wesentlichen Konsequenzen der Virtualisierung von CPU-Kernen betrifft den Umgang mit Interrupts im System. Wird ein Betriebssystem nichtvirtualisiert ausgeführt, so kann dieses eine Interrupt-Maskierung direkt im realen Local-APIC des jeweiligen CPU-Kerns vornehmen, um beispielsweise die unterbrechungsfreie Ausführung einer *Interrupt-Service-Routine (ISR)* zu gewährleisten. Dies hat zur Konsequenz, dass ein eintretender, aber derzeit mas-

[4] Die Evaluation wurde auf Basis eines *Early Access Release (EAR)* des Produkts durchgeführt, für welches durch die Firma *Bosch Rexroth* ein *Non-Disclosure Agreement (NDA)* unterzeichnet wurde. Auf Basis dieses NDA wurde durch den Hersteller eine Veröffentlichung der Evaluationsergebnisse unter Nennung des Produktnamens untersagt. Dies ist allerdings für die folgenden Darstellungen der Ergebnisse unkritisch, da ohnehin eine Evaluation der dem Produkt zugrunde liegenden Konzepte und nicht das Produkt selbst im Fokus dieser Arbeit steht. Allerdings muss aus diesem Grund bei der Beschreibung des virtualisierten Konsolidierungsansatzes auf die Angabe entsprechender Quellen verzichtet werden.

kierter Interrupt im Local-APIC einmalig als *pending* markiert und erst beim Aufheben der Maskierung zugestellt wird. Aufgrund der möglichen Virtualisierung der CPU-Kerne muss beim virtualisierten Konsolidierungsansatz allerdings eine direkte Konfiguration der realen Interrupt-Controller durch ein Gastsystem verhindert werden, da dies alle Gastsysteme betreffen würde, die dem CPU-Kern des entsprechenden Local-APIC zugeordnet sind. Stattdessen muss für jede virtuelle Maschine des Systems ein virtuelles Abbild des Interrupt-Controllers, der sogenannte *Virtual APIC*, durch den Hypervisor bereitgestellt und zur Laufzeit verwaltet werden. Eine Konfiguration durch ein Gastsystem, beispielsweise in Form einer Interrupt-Maskierung, wirkt nun ausschließlich auf das jeweils zugeordnete virtuelle Controller-Abbild. Als Konsequenz muss nun die Logik des realen Interrupt-Controllers durch den Hypervisor nachgebildet werden. So muss jeder am realen Controller eintreffende Interrupt zunächst durch den Hypervisor entgegengenommen und nach einem Routing am entsprechenden virtuellen APIC zugestellt oder im Falle einer dortigen Maskierung als *pending* markiert werden. Dies erfordert jedoch stets einen Kontextwechsel von der Ausführung einer virtuellen Maschine in den Hypervisor sowie einen Kontextwechsel zurück zum Gastsystem.

Die durch diesen Konsolidierungsansatz geleistete Virtualisierung basiert zunächst auf der Methode der Paravirtualisierung, bei der Gastsysteme im User-Modus laufen und privilegierte Befehle durch Hypercalls in den Hypervisor ersetzt werden. Ist zudem eine Virtualisierungsunterstützung in Hardware, wie beispielsweise *Intel VT* [50, 125], verfügbar, so werden deren Dienste zusätzlich in Anspruch genommen, um den Umfang der für eine Paravirtualisierung erforderlichen Betriebssystemadaptionen zu reduzieren.

Ebenso wie dem Hypervisor selbst wird auch den virtuellen Maschinen im Rahmen einer Speichervirtualisierung jeweils eine Teilmenge des verfügbaren Hauptspeichers exklusiv zur Verfügung gestellt, so dass jede VM einen zusammenhängenden, bei der Adresse 0x0 beginnenden, virtuellen Adressraum hat. Gemäß Abschnitt 2.10.2 wird damit eine Adressübersetzung von den physikalischen Adressen der Gastsysteme (GPA) in die physikalischen Adressen der Maschine (HPA) eingeführt. Zu beachten ist, dass auf eine solche in nativ ausgeführten Echtzeitsystemen häufig verzichtet wird, um deterministische Zugriffszeiten zu erzielen. Diese Übersetzung ist auch bei DMA-Transfers eines Gastsystems mit I/O-Geräten erforderlich, so dass der virtualisierte Konsolidierungsansatz zu diesem Zweck entsprechende Hypercalls zur Nutzung durch die Gastsysteme definiert. Eine I/O-MMU, beispielsweise in Form von *Intel VT-d* [1], die unter anderem diese Adressübersetzung in Hardware realisiert, wird durch das vorliegende Release des virtualisierten Konsolidierungsansatzes noch nicht unterstützt.

4.3.2 Hybrider Konsolidierungsansatz

Einen hybriden Konsolidierungsansatz hingegen verfolgt der *RTS Hypervisor* der Firma *Real-Time Systems*, der sich ebenfalls als Konsolidierungslösung im Bereich echtzeitfähiger Systeme versteht [149]. Dabei nimmt er jedoch für sich in Anspruch, die jeweilige Hardware so weit wie möglich nur unter den Gastsystemen zu partitionieren, statt diese zu virtualisieren. Dies äußert sich zunächst dadurch, dass mit dem *Virtualized Mode* und dem *Privileged Mode* zwischen zwei Ausführungsmodi für Gastsysteme differenziert wird, die in der Hypervisor-Konfiguration statisch definiert werden. Gastsysteme im *Privileged Mode* werden dabei grundsätzlich mit vollen Rechten und somit im Supervisor-Modus ausgeführt und sollen auf diese Weise in ihrem Zeitdeterminismus und ihrer Reaktivität nicht beeinträchtigt werden. Dieser Modus

wird vom Hersteller allerdings nur für vertrauenswürdige Systeme mit Echtzeitanforderungen empfohlen [149], da bei diesen unterstellt wird, dass der Code in ausreichender Weise durch die Applikationsentwickler getestet und vor Modifikationen durch den Anwender geschützt ist. Gastsysteme im *Virtualized Mode* werden hingegen mittels der Hardware-Virtualisierungsunterstützung *Intel VT* in eine virtuelle Maschine verlagert und somit deprivilegiert. Dieser Modus empfiehlt sich nach Angaben des Herstellers für Betriebssysteme, auf die der Anwender direkten Zugriff hat und die somit als potentiell nicht vertrauenswürdig einzustufen sind.

Unter Verweis auf die angestrebte Eignung für einen Einsatz im Echtzeitumfeld verfolgt der hybride Ansatz das Ziel, die Auswirkungen der Integration auf das temporale Verhalten der Gastsysteme im *Privileged Mode* so weit wie möglich zu reduzieren. Konsequenz dieser Motivation ist eine ausschließlich räumliche statt einer temporalen Partitionierung der CPU-Kerne und somit deren exklusive Zuordnung zu den Gastbetriebssystemen. Dies ermöglicht den Verzicht auf eine Virtualisierung der Kerne und somit auf eine Virtualisierung des Interrupt-Controllers zum Routen eintreffender Interrupts. Stattdessen werden die Interrupts statisch den CPU-Kernen zugeordnet, so dass das Routing an die entsprechenden Local-APICs durch den *I/O-APIC* in Hardware geleistet werden kann. Weiterhin wird für Systeme im *Privileged Mode* auf eine Speichervirtualisierung, die den Gastsystemen jeweils nur eine Teilmenge des physikalisch verfügbaren Speichers bereitstellen würde, verzichtet, um das Risiko nichtdeterministischer Zugriffszeiten infolge der Adressübersetzung zu reduzieren.

Da Gastsysteme im *Privileged Mode* mit vollen Rechten laufen, können Sie den kompletten Instruktionssatz der CPU nutzen, ohne dass dem Hypervisor dabei die Möglichkeit zur Intervention eingeräumt wird. Da der Hypervisor somit deren Verhalten nicht überwachen kann, muss er diesen Gastsystemen zumindest die Möglichkeit zu einem kooperativen Verhalten im Kontext des Gesamtsystems und somit zu einer aktiven Interaktion mit dem Hypervisor bieten. Dazu wird der Code privilegierter Gäste, der die koexistente Ausführung der Gastsysteme beeinträchtigen kann, durch Funktionsaufrufe in ein Hypervisor-Modul ersetzt, das beim Booten für jedes Gastsystem genau einmal in den Speicher geladen wird. Die Ausführung der angeforderten Aktion erfolgt nun durch das entsprechende Hypervisor-Modul in Abhängigkeit vom globalen Systemzustand, der vom *RTS Hypervisor* in einem als *Registry* bezeichneten gemeinsamen Speicherbereich verwaltet wird [148].

Im einfachsten Fall reduzieren sich die erforderlichen Anpassungen zur Sicherstellung der koexistenten Ausführung auf die Hardware-Abstraktionsschicht eines Gastsystems, die häufig als sogenanntes *Board Support Package (BSP)* realisiert ist. Dies ist allerdings nur dann möglich, wenn sich alle Zugriffe der Applikation auf entsprechende kritische Hardware-Konfigurationen auf diese Schnittstelle beschränken; andernfalls ist auch die Applikation selbst zu adaptieren. Die erforderlichen BSP-Adaptionen der im *Privileged Mode* ausgeführten Gastsysteme sind unter anderem eine Konsequenz der fehlenden Speichervirtualisierung für diese Systeme: Während manche Betriebssysteme dynamisch relokiert werden können, lässt sich die Relokation beispielsweise bei *VxWorks* nur durch eine Rekompilierung mit angepasster Basisadresse realisieren. Weiterhin darf das Gastsystem nicht wie bei einer nativen Ausführung den maximal zur Verfügung stehenden Speicher dynamisch ermitteln, sondern muss beim Booten mittels eines Funktionsaufrufs in das Hypervisor-Modul den ihm zugewiesenen Speicherumfang abfragen. Wenngleich Ressourcen beim hybriden Ansatz den Gastbetriebssystemen generell exklusiv zugewiesen werden, gibt es Hardware-Komponenten, die von mehreren Gastsystemen implizit gemeinsam genutzt werden. Zu diesen zählen beispielsweise der *PCI*-Bus und der I/O-APIC.

Aus diesem Grund sind Konfigurationsänderungen dieser Komponenten auch sensitiv bezüglich des Gesamtsystems, so dass das Hypervisor-Modul spezifische Funktionen zur Konfiguration dieser Komponenten implementiert. Um den Reboot eines Gastsystems ohne Seiteneffekte auf das Gesamtsystem durchführen zu können, darf zudem kein Hardware-Reset stattfinden. Stattdessen muss im Code eines Gastsystems unmittelbar vor einem solchen Reset ein Funktionsaufruf in das Hypervisor-Modul erfolgen, der in der Regel dazu führt, dass der Hypervisor das Gastsystem neu lädt und startet.

4.4 Evaluation

Während in den vorangegangenen Abschnitten zunächst die Anforderungen an eine Konsolidierungslösung im automatisierungstechnischen Umfeld definiert und im Anschluss daran zwei entsprechende Ansätze vorgestellt wurden, widmen sich die nachfolgenden Abschnitte einer Bewertung dieser Ansätze unter einer Auswahl der zuvor definierten Anforderungen. Sofern möglich, wird dabei nicht eine Bewertung der Produkte, sondern der jeweils zugrunde liegenden Konsolidierungsstrategie angestrebt.

Bezüglich der Reaktivität und des Zeitdeterminismus findet dabei eine quantitative Evaluation der Konsolidierungsansätze statt. Da eine solche aufgrund der Systemkomplexität analytisch nicht möglich ist, stellt eine empirische Analyse mittels geeigneter Messungen mit einer ausreichenden Anzahl an Iterationen die einzige Möglichkeit einer solchen Bewertung dar. Die Implementierung der Messlogik sowie die Durchführung der Messungen wurden dabei im Rahmen einer Diplomarbeit [2] geleistet. Generell sind folgenden Aspekte zu beachten:

- Da es sich bei den Hypervisor-Produkten nicht um quelloffene Lösungen handelt, muss man sich hinsichtlich der Details der technischen Realisierung wie der Konfiguration der Hardware oder der Nutzung von Funktionen der Hardware-Virtualisierungsunterstützung auf Herstellerangaben verlassen, sofern diese verfügbar sind.
- Die der Evaluation zugrunde liegende x86-Plattform weist eine Vielzahl von Konfigurationsmöglichkeiten auf, was einen enormen Parameterraum für die Messungen generiert.

Diese Aspekte führen letztlich dazu, dass eine absolut identische Konfiguration der Hardware bei der Durchführung der Messungen nicht garantiert werden kann, was wiederum die Interpretation der Ergebnisse erschwert.

Für die empirische Evaluation wurden schließlich beide Konsolidierungsansätze entsprechend der Darstellung in Abbildung 4.1 auf einer Dual-Core-CPU zur Ausführung gebracht. Bei der entsprechenden Hardware-Plattform handelt es sich um eine typische Automatisierungssteuerung in Form der *Rexroth IndraControl L85* [25]. Dieser Controller ist mit einer CPU vom Typ *Intel Core 2 Duo* ausgestattet, die über die Virtualisierungsunterstützung *Intel VT* verfügt und deren Taktrate bei 1,2 GHz liegt. Als Chipsatz kommt ein *Intel GS45 Express Chipset* mit in die *Northbridge* integriertem Grafik-Controller zum Einsatz. Obwohl der virtualisierte Konsolidierungsansatz die Zuordnung mehrerer virtueller Maschinen zu einem CPU-Kern ermöglicht, wurde auf diese Option verzichtet, um eine bessere Vergleichbarkeit der Ergebnisse zu erzielen. Als Konsequenz wurden hier auf dem ersten CPU-Kern das Echtzeitbetriebssystem *VxWorks 6.7* und auf dem zweiten CPU-Kern *Wind River Linux 2.0.3* zur Ausführung konfiguriert. Beim hybriden Konsolidierungsansatz wurden hingegen auf dem ersten Kern *VxWorks 6.3* im *Privileged*

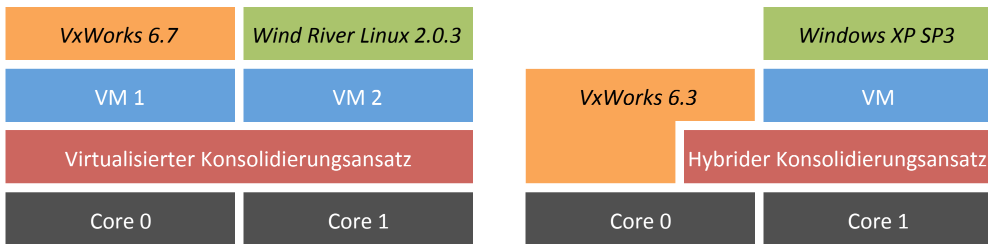


Abbildung 4.1: Messkonfigurationen zur Evaluation der Konsolidierungsansätze

Mode und auf dem zweiten Kern *Windows XP Service Pack 3* im *Virtualized Mode* ausgeführt. Zu berücksichtigen ist, dass die Heterogenität der *VxWorks*-Versionen sowie der Standardbetriebssysteme in den Messkonfigurationen infolge von Kompatibilitätseinschränkungen der evaluierten Hypervisor-Lösungen unvermeidbar war. Die somit fehlende Äquivalenz der Konfigurationen stellte bei den Messungen stets eine potentielle Einflussgröße dar, welche die direkte Vergleichbarkeit der unter den Konsolidierungsansätzen erzielten Ergebnisse bis zu einem gewissen Grad beeinträchtigen kann. Dies wird im Folgenden nicht in jedem Kontext explizit erwähnt, ist allerdings bei der Interpretation der Ergebnisse stets zu berücksichtigen.

4.4.1 Reaktivität

In Abschnitt 4.2 wurde die Wahrung der Reaktivität eines Echtzeitsystems als eine der wesentlichen Anforderungen an eine Konsolidierung im Bereich der industriellen Automatisierung definiert. Somit ist eine Evaluation der in Abschnitt 4.3 vorgestellten Konsolidierungsansätze unter diesem Aspekt von besonderer Relevanz. Als Metrik für die Reaktivität eines Systems sei dessen Reaktionszeit *(Response Time)* t_r als das Zeitintervall zwischen einem Stimulus an das System und der Reaktion des Systems darauf definiert:

$$t_r := t_i + t_p + t_o \tag{4.1}$$

Wie in Abbildung 4.2 dargestellt, entspreche dabei t_i der Interrupt-Latenz, t_p der Ausführungszeit einer zur Bearbeitung des Stimulus implementierten Interrupt-Service-Routine und t_o der benötigten Ausgabelatenz, bis die in der ISR veranlasste Reaktion die Peripherie erreicht.

Messaufbau

Um nicht nur den Einfluss einer integrierten Ausführung, sondern auch den Einfluss eines auf dem zweiten Kern ausgeführten Standardbetriebssystems (GPOS) auf die Reaktivität des Echtzeitsystems zu evaluieren, wurden unter diesem diverse Lastszenarien implementiert. Diese sind als synthetische Workloads konzipiert, stellen jedoch eine nach Lasttypen klassifizierte Verdichtung der in einem realen Standardbetriebssystem potentiell auftretenden Lasten dar:

- Beim Szenario *Idle* wird nach dem Booten des GPOS kein Anwenderprozess explizit gestartet. Somit bildet dieses Szenario den Fall eines zwar konfigurierten und parallel ausgeführten Standardbetriebssystems ab, in dem jedoch keine zusätzliche Last neben

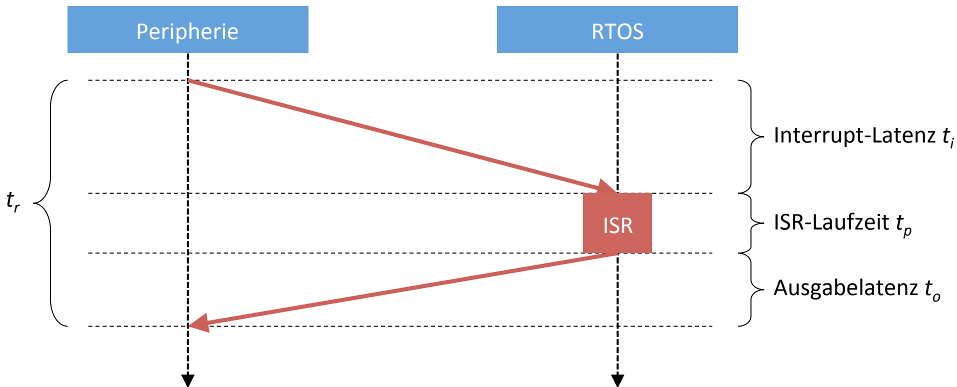


Abbildung 4.2: Zusammensetzung der zur Evaluation der Systemreaktivität definierten Response Time t_r nach Formel 4.1

den typischen Hintergrundprozessen des Betriebssystems erzeugt wird. Die Wahl dieses Szenarios ist dadurch motiviert, dass hier die geringsten Seiteneffekte des Standardbetriebssystems auf das Echtzeitsystem unterstellt werden.

- Das Szenario *Integer Load* führt im Standardbetriebssystem einen hochprioren Prozess mit *Integer*-Berechnungen aus. Sowohl der Programmcode als auch die von diesem genutzten Daten sind dabei so gewählt, dass sie die Kapazität des für jeden CPU-Kern exklusiv vorhandenen L1-Cache nicht übersteigen. Da dieses Lastszenario als Anwenderprozess realisiert ist, können allerdings sporadische Kontextwechsel zu hochprioren Kernel-Tasks und somit Zugriffe auf gemeinsam genutzte Cache-Level oder andere nicht-exklusive Hardware-Ressourcen nicht ausgeschlossen werden.
- Die im Szenario *Memory Load* erzeugte Last generiert hochpriore Lese- und Schreibzugriffe auf den Hauptspeicher: Es werden Speicherblöcke unterschiedlicher Größe zunächst alloziert, beschrieben, gelesen und abschließend wieder freigegeben. Auch die Prozesspriorität dieses Workloads liegt unterhalb der Priorität der Kernel-Tasks, so dass auch hier Unterbrechungen und Kontextwechsel nicht ausgeschlossen werden können.
- Beim Szenario *Forkbomb* wird ein Prozess ausgeführt, der in einer Schleife Kopien seiner selbst startet *(Forking)*. Da somit der Kindprozess die gleiche Programmlogik wie der Elternprozess abarbeitet, liegt einer Forkbomb eine rekursive Ausführung zugrunde. Als Konsequenz sind die in eine Prozessgenerierung involvierten Systemressourcen wie die CPU, die Speicheranbindung und die Prozesstabellen des Betriebssystem-Kernels einer hohen Last ausgesetzt [98] und somit meist zeitnah nicht mehr verfügbar. Eine Forkbomb realisiert somit eine *Denial-of-Service*-Attacke. Zu beachten ist, dass sich sowohl die Implementierung der Forkbomb [2] als auch das Systemverhalten unter einer Forkbomb in Abhängigkeit vom jeweiligen Betriebssystem stark unterscheiden. Somit kann eine Forkbomb nur den Versuch darstellen, hinsichtlich der Systemlast unter dem jeweiligen Standardbetriebssystem ein *Worst-Case*-Szenario zu generieren, ohne zugleich den Anspruch auf eine Äquivalenz dieser Lasten zu erheben.

Zur Messung der Reaktivität des Echtzeitbetriebssystems wurde ein über den PCI-Bus angebundenes FPGA verwendet, welches über eine dedizierte *Clock* getaktet ist. Somit konnte das FPGA als externe Messinstanz genutzt werden, da es periodisch einen Interrupt der CPU triggert und daraufhin die Zeitdauer t_r misst, bis die CPU ein *Acknowledge*-Bit in den Speicher des FPGAs schreibt. Insgesamt wurden für jeden Konsolidierungsansatz pro Lastszenario des Standardbetriebssystems 50 000 Messiterationen durchgeführt, von denen jede aus 5 Messungen bestand. Dies ergab somit für jede Iteration i und jede Messung n eine Response Time $t_r(i, n)$. Auf Grundlage dieser Werte seien die Werte $t_{r_{max}}(i)$ definiert, welche pro Iteration i nur die jeweils maximale Response Time $t_{r_{max}}(i)$ berücksichtigen:

$$t_{r_{max}}(i) \coloneqq \max\left(t_r(i, n) \,|\, 0 \leq n < 5\right) \tag{4.2}$$

Die Wahl von $t_{r_{max}}(i)$ ist dadurch motiviert, dass in der Steuerungstechnik vor allem das Verhalten im Worst Case von Relevanz ist, weil daraufhin die Parametrierung und Taktung der Steuerung optimiert werden.

Messergebnisse

Die nach Lastszenarien differenzierten Messwerte der Response Times $t_{r_{max}}$ sind in den Abbildungen 4.3 und 4.4 für den virtualisierten respektive hybriden Konsolidierungsansatz dargestellt. Eine Aggregation der Werte $t_{r_{max}}(i)$ der Iterationen i erfolgt dabei mittels zweier Werte:

- Das *Maximum* berechnet sich als $\max(t_{r_{max}}(i) | 0 \leq i < 50\,000)$ und entspricht somit der längsten gemessenen Response Time aller Iterationen i. Für die Evaluation der Echtzeitfähigkeit einer Konsolidierungslösung besitzt dieser Wert die meiste Relevanz.
- Das *Minimum* berechnet sich als $\min(t_{r_{max}}(i) | 0 \leq i < 50\,000)$ und liefert somit die in allen Iterationen i im besten Fall erzielte Response Time. Dieser Wert definiert zusammen mit dem Maximum die Spannweite der erfassten Response Times, besitzt aber für die Bewertung der Echtzeitfähigkeit keine Relevanz.

Unter dem Echtzeitbetriebssystem *VxWorks 6.7* ist zunächst im nativen Fall eine maximale Response Time von 5 µs messbar. Dieser Wert steigt beim virtualisierten Konsolidierungsansatz unter einer *Forkbomb*-Last im Standardbetriebssystem auf einen Wert von bis zu 43 µs[5], so dass dieses Lastszenario den größten Seiteneffekt auf das Echtzeitsystem ausübt. Davon abgesehen ist auffällig, dass in allen Lastszenarien der virtualisierten Ausführung das Minimum der Response Times bei 15 µs und somit deutlich über der maximalen Response Time der nativen Ausführung liegt. Der Grund für diesen Effekt ist vermutlich die bei diesem Konsolidierungsansatz vorgenommene APIC-Virtualisierung. So unterliegen eintreffende Interrupts stets einem Routing durch den Hypervisor und die beim ISR-Eintritt durchgeführte Interrupt-Maskierung muss im virtuellen APIC erfolgen. Weiterhin ist ein deutlicher Einfluss der Virtualisierung auf die Spannweite der Response Times zu beobachten: Während sich die Response Times im nativen Fall

[5] Beim virtualisierten Konsolidierungsansatz waren je nach Lastszenario zwischen 0,01 % und 0,12 % extreme Ausreißer in den Response Times feststellbar, die sich in einem Intervall zwischen 264 µs und 997 µs bewegten. Diese konnten jedoch mittels einer Maskierung aller maskierbaren und nicht dem FPGA zugeordneten Interrupts verhindert werden, so dass als Ursache eine fehlerhafte APIC-Virtualisierung des Hypervisors vermutet wird. Da es sich somit hierbei mit hoher Wahrscheinlichkeit nicht um ein inhärentes Problem dieses Konsolidierungsansatzes handelt, werden diese sporadischen und extremen Ausreißer in der nachfolgenden Evaluation nicht berücksichtigt.

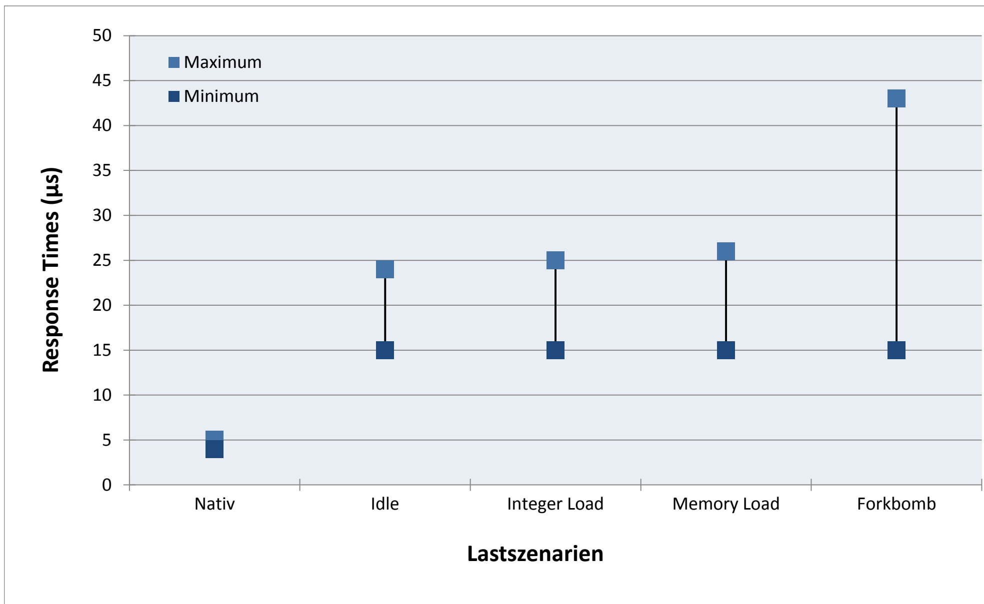


Abbildung 4.3: Response Times des virtualisierten Konsolidierungsansatzes

immer im Intervall zwischen 4 µs und 5 µs bewegen, unterscheiden sich das Minimum und das Maximum beim virtualisierten Konsolidierungsansatz mit einer Differenz von mindestens 9 µs deutlich. Als Ursache dieser Schwankungen sind sporadische Kontextwechsel von der Interrupt-Service-Routine des Gastsystems zum Kernel-Modul des Hypervisors zu vermuten, welche die Ausführungsdauer t_p der ISR deutlich verlängern. Wie in Abschnitt 4.3.1 beschrieben, wird ein solcher Kontextwechsel beispielsweise erforderlich, wenn während der ISR-Ausführung ein weiterer Interrupt eintrifft. Somit kann zusammenfassend ein signifikanter Einfluss der virtualisierten Systemkonsolidierung auf die Reaktivität des Echtzeitbetriebssystems festgestellt werden. Sofern die hier gemessenen Werte allerdings sichere obere Schranken und somit *Worst Case Response Times (WCRTs)* darstellen, kann ein Overhead dieser Größenordnung unter Umständen in einem entsprechend angepassten und robusteren Applikationsdesign berücksichtigt und damit kompensiert werden.

Bei dem Echtzeitbetriebssystem *VxWorks 6.3* ist schließlich im nativen Fall eine maximale Response Time von 6 µs beobachtbar. Das Minimum von 5 µs wiederum wird auch bei einer Ausführung unter dem hybriden Konsolidierungsansatz unter allen Lastszenarien im Minimum erzielt, so dass hier im Gegensatz zum virtualisierten Konsolidierungsansatz kein obligatorischer Overhead generiert wird. Darüber hinaus ist beachtenswert, dass unter keinem Lastszenario eine Response Time von mehr als 8 µs feststellbar ist. Dies lässt sich auf die nichtvirtualisierte Ausführung des Echtzeitsystems und den damit einhergehenden Verzicht auf eine APIC-Virtualisierung beim hybriden Konsolidierungsansatz zurückführen. Somit müssen hier bei der Behandlung eines Interrupts keine zeitaufwändigen Kontextwechsel in den Hypervisor erfol-

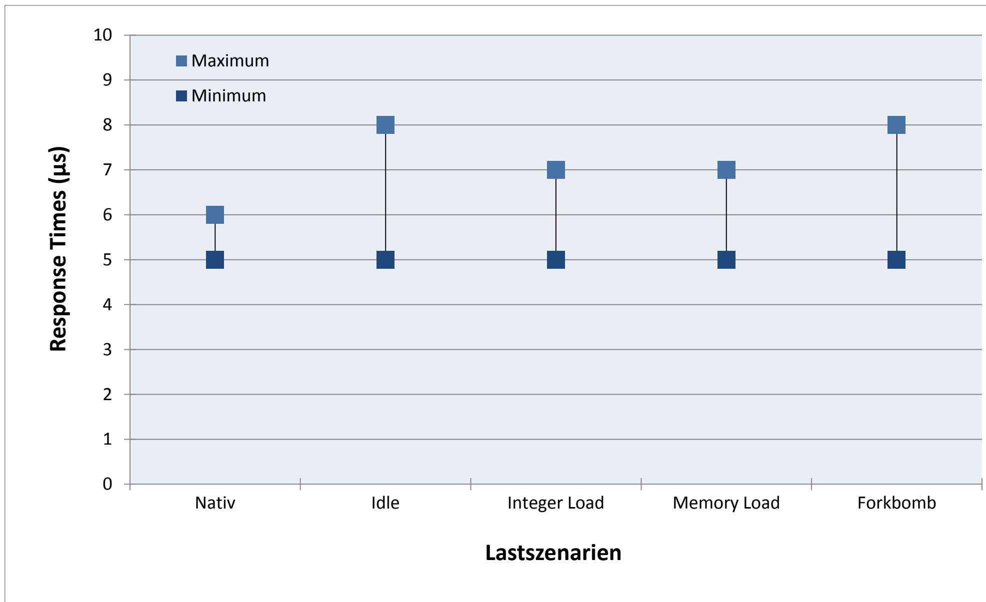


Abbildung 4.4: Response Times des hybriden Konsolidierungsansatzes

gen, da beispielsweise das Interrupt-Routing und die Interrupt-Maskierung im realen APIC-Subsystem vorgenommen werden können. Die hier erzielten Ergebnisse zeigen zudem, welche Reaktivität im Echtzeitsystem unter einer Systemkonsolidierung bewahrt werden kann, selbst wenn das Standardbetriebssystem zur gleichen Zeit die zuvor definierten Lastszenarien auf gemeinsam genutzten Betriebsmitteln generiert. Dies wiederum lässt die Schlussfolgerung zu, dass konkurrierende Hardwarezugriffe als Ursache für die beim virtualisierten Konsolidierungsansatz gemessenen Einflüsse auf die Reaktivität des Echtzeitsystems weitestgehend ausgeschlossen werden können.

4.4.2 Zeitdeterminismus

Als weiteres Bewertungskriterium einer Systemkonsolidierung wurde der Zeitdeterminimus definiert, den entsprechend Abschnitt 4.2 konkurrierende Ressourcenzugriffe und Maßnahmen zur Sicherstellung der Gastsystem-Isolation potentiell beeinträchtigen.

Messaufbau

Um den Einfluss der zuvor genannten Faktoren auf den Zeitdeterminismus der von den Konsolidierungslösungen bereitgestellten Ausführungsumgebungen zu evaluieren, wurden die Ausführungszeiten ausgewählter synthetischer Workloads unter dem jeweiligen Echtzeitbetriebssystem gemessen. Um eine detaillierte Bewertung zu ermöglichen, bilden diese die typischen Charakteristika einer realen Firmware in Form separater Workloads ab:

- Der Workload *Integer Load* umfasst einen hochprioren Prozess mit ganzzahligen arithmetischen Berechnungen, deren Ausführungsdauer für 1 000 000 Iterationen erfasst wird.
- Der Workload *Memory Load* besteht aus einem hochprioren Prozess, der ausschließlich Lese- und Schreibzugriffe auf den Hauptspeicher durchführt. Aufgrund der signifikant längeren Laufzeit des Workloads werden hier nur 50 000 Iterationen durchgeführt.
- Auch der Workload *I/O* besteht aus Lese- und Schreibzugriffen einer hochprioren Task. Das Ziel der Zugriffe ist allerdings nicht der Hauptspeicher, sondern der Speicher von I/O-Ressourcen, der in den globalen Adressraum abgebildet wird *(Memory Mapped I/O)*. Eine Messung umfasst 200 000 Iterationen.

Zu beachten ist, dass diese Workloads gegenüber einigen Kernel-Tasks des Echtzeitbetriebssystems niedriger priorisiert sind, so dass sporadische Unterbrechungen der entsprechenden Prozesse nicht ausgeschlossen werden können. Die Hardware-Plattform der Messung stellte ebenso wie bei der Messung der Response Times in Abschnitt 4.4.1 die Automatisierungssteuerung *Rexroth IndraControl L85* dar. Aufgrund der längeren Laufzeiten war allerdings in diesem Fall eine externe Messung über das FPGA nicht möglich, so dass der *Time Stamp Counter* der CPU zur internen Messung genutzt wurde. Pro Workload wurde dabei die Ausführungsdauer im nativen Fall gemessen und der Ausführungsdauer im konsolidierten Szenario gegenübergestellt, wobei im GPOS erneut die in Abschnitt 4.4.1 beschriebenen Lastszenarien *Idle*, *Integer Load*, *Memory Load* und *Forkbomb* ausgeführt wurden.

Messergebnisse

Die Abbildungen 4.5 und 4.6 zeigen schließlich die Messergebnisse für den virtualisierten respektive hybriden Konsolidierungsansatz. Die Ausführungszeiten sind dabei stets in Relation zur minimalen Ausführungsdauer des entsprechenden Workloads im nativen Fall gesetzt; diese Bezugspunkte sind im Diagramm rot markiert. Ebenso wie bei den Reaktionszeiten sind bei der Auswertung der Ergebnisse vor allem die jeweiligen Maxima von besonderer Relevanz, da diese für die Wahl der Maschinentaktung maßgeblich sind, um eine zuverlässige Ausführung zu gewährleisten. Somit ist für jede Kombination aus Workload und Lastszenario stets die maximal gemessene Ausführungsdauer des Workloads dargestellt. Darüber hinaus visualisieren die Diagramme auch die jeweils minimal gemessene Ausführungsdauer in relativer Weise, um die Spannweite der Resultate zu verdeutlichen.

Beim virtualisierten Konsolidierungsansatz stellt der signifikante Einfluss der *Forkbomb* auf den *I/O*-Workload das bemerkenswerteste Ergebnis dar, da hier die Ausführungsdauer des Workloads im Maximum gegenüber dem nativen Fall um 94 % steigt[6]. Dabei handelt es sich auch nicht um einen sporadischen Ausreißer, da hier selbst das im Diagramm nicht dargestellte arithmetische Mittel einen Zuwachs von 89,6 % aufweist. In allen übrigen Lastszenarien werden die jeweiligen Workloads durch die Lastszenarien hingegen deutlich geringer in ihrer Ausführungsdauer beeinflusst. Die Zuwächse gegenüber dem jeweils besten Resultat der nativen Ausführung liegen hier, wie im Diagramm dargestellt, maximal im unteren einstelligen

[6] Bei einigen Messreihen wurde jeweils eine einzige Iteration mit einer von den übrigen Messwerten um Größenordnungen abweichenden Ausführungsdauer gemessen. Aufgrund der somit je nach Workload sehr geringen relativen Häufigkeit von 0,002 % respektive 0,0001 % werden diese Messungen in der Auswertung nicht berücksichtigt. Dem liegt die Annahme zugrunde, dass es sich hier um Messfehler oder um einen Fehler in der Hypervisor-Implementierung handelt und nicht um ein dem jeweiligen Konsolidierungsansatz inhärentes Merkmal.

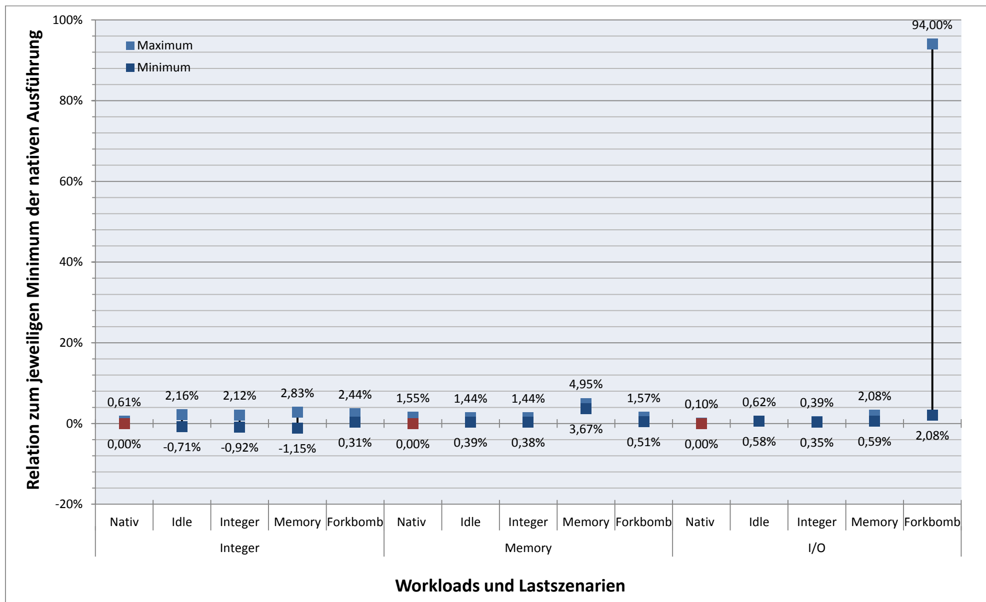


Abbildung 4.5: Ausführungszeiten der Workloads beim virtualisierten Konsolidierungsansatz

Prozentbereich. Nicht abschließend erklärbar ist die Tatsache, dass die für die Ausführungsdauer des *Integer*-Workloads im Minimum erzielten Werte bei einer virtualisierten Ausführung geringer sind als bei einer nativen Ausführung. Als Ursache lässt sich allerdings die standardmäßig durch den Hypervisor gewählte Partitionierung der Interrupts unter den Gastsystemen vermuten. Dies kann dazu führen, dass im nativen Fall dem Echtzeitsystem zugeordnete Interrupts nun durch das Standardbetriebssystem behandelt werden und so die Rechenlast des Echtzeitsystems reduziert wird. Bei der Spannweite der gemessenen Workload-Laufzeiten zeigt sich mit Ausnahme des zuvor beschriebenen Szenarios des *I/O*-Workloads bei gleichzeitiger *Forkbomb*-Last kein signifikanter Einfluss des virtualisierten Konsolidierungsansatzes. So ist hier je nach Workload und Lastszenario sowohl bei einer nativen als auch bei einer virtualisierten Ausführung eine Spannweite der Ausführungsdauer im unteren einstelligen Prozentbereich feststellbar.

Beim hybriden Konsolidierungsansatz liefert erneut die Kombination aus dem *Forkbomb*-Lastszenario und dem *I/O*-Workload das Messergebnis mit der größten Signifikanz. Allerdings fällt hier der Zuwachs der Ausführungsdauer gegenüber dem nativen Fall mit 21,5 % deutlich geringer aus als beim virtualisierten Konsolidierungsansatz. In den übrigen Konstellationen aus Workload und Lastszenario liegen hingegen die Zuwächse der Ausführungsdauer im Worst Case erneut im unteren einstelligen Prozentbereich. Auch bezüglich der Spannweite der Ausführungsdauer ist mit Ausnahme der Konstellation aus *I/O*-Workload und *Forkbomb*-Last kein signifikanter Einfluss des hybriden Konsolidierungsszenarios feststellbar; so ist im Fall des *Memory*-Workloads die Spannweite der nativen Ausführung sogar größer als die der integrierten

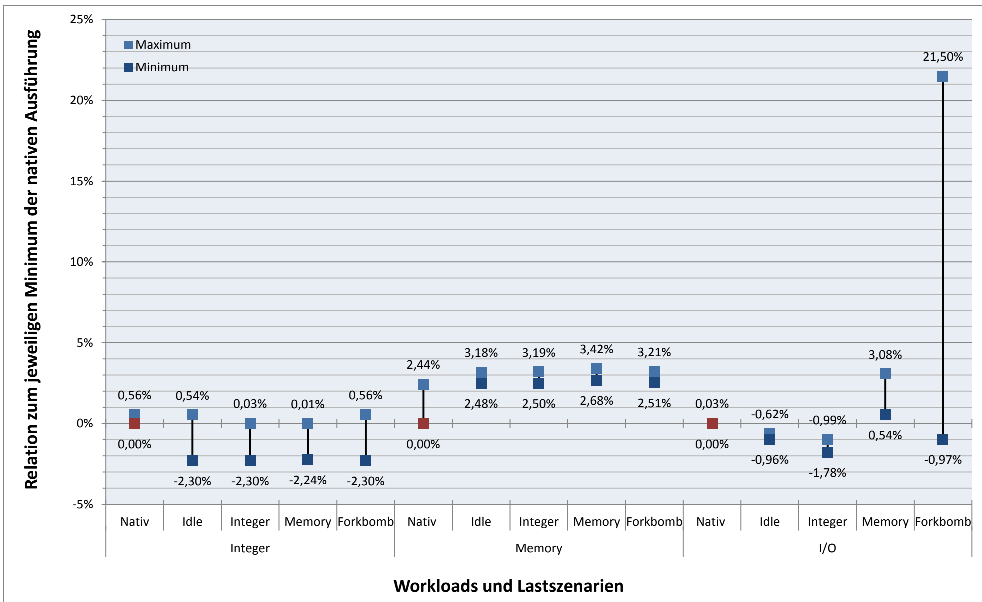


Abbildung 4.6: Ausführungszeiten der Workloads beim hybriden Konsolidierungsansatz

Ausführung. Darüber hinaus ist in ausgewählten Konstellationen bei einer integrierten Ausführung erneut eine geringere Ausführungsdauer feststellbar als bei der nativen Ausführung; auch hier wird die Interrupt-Partitionierung unter den Gastsystemen als Ursache vermutet.

Zusammenfassend sind hinsichtlich des Zeitdeterminismus die zuvor beschriebenen Gemeinsamkeiten zwischen einer Ausführung im virtualisierten und im hybriden Konsolidierungsansatz hervorzuheben. Dies lässt die Interpretationen zu, dass entweder beide Ansätze ähnliche Strategien beim Umgang mit Ressourcenkonflikten verfolgen oder dass diese Effekte ihre gemeinsame Ursache in der Architektur der zugrunde liegenden Multicore-Hardware haben. Hervorzuheben ist generell, dass die Einflüsse der Systemkonsolidierungen auf den Zeitdeterminismus der Workloads bei den meisten Lastszenarien im unteren einstelligen prozentualen Bereich liegen. Dennoch lassen sich bei beiden Konsolidierungsansätzen spezifische Konstellationen aus Workload und Lastszenario generieren, bei denen die temporale Isolation zwischen den Gastsystemen signifikant beeinträchtigt wird. Diese Tatsache ist vor allem vor dem Hintergrund des in Abschnitt 4.2 beschriebenen Szenarios eines dynamischen Nachladens von Programmen im Standardbetriebssystem als äußerst kritisch zu bewerten.

4.4.3 Zuverlässigkeit und Sicherheit

Wie in Abschnitt 3.2 beschrieben, werden in der industriellen Automatisierung hohe Anforderungen an die Zuverlässigkeit und Sicherheit der eingesetzten Systeme gestellt, um Schaden von Personen oder der Umwelt abzuwenden und die Integrität der verarbeiteten Informationen zu gewährleisten. Als Konsequenz müssen auch die Implikationen einer Systemkonsolidierung

auf alle Komponenten einer automatisierungstechnischen Steuerung, die zur Wahrung der an das Gesamtsystem gestellten Safety- und Security-Anforderungen beitragen, evaluiert werden.

Im Kontext informationsverarbeitender Systeme stellt die Grundlage jedes vertrauenswürdigen Systems die sogenannte *Trusted Computing Base (TCB)* dar. Diese umfasst alle Mechanismen, die für die Durchsetzung der Sicherheitsrichtlinien eines Systems und für die Isolation der Module, auf denen die Sicherheitsmaßnahmen eines Systems basieren, erforderlich sind [173]. Kann durch eine Verifikation der Nachweis erbracht werden, dass die TCB entsprechend ihrer als korrekt unterstellten Spezifikation implementiert ist, so kann garantiert werden, dass die Systemsicherheit auch unter beliebigen Umgebungseinflüssen, wie beispielsweise bei gezielten Angriffen, nicht beeinträchtigt werden kann [168]. Die Trusted Computing Base eines Systems umfasst dabei mit wenigen Ausnahmen die komplette Hardware sowie in der Regel auch jeglichen mit Supervisor-Rechten ausgeführten Code des Systems, da dieser alle durch das System implementierten Sicherheitsmechanismen unterwandern kann. Somit sind bei der nativen Ausführung eines Betriebssystems nicht nur zentrale Bestandteile des Betriebssystem-Kernels wie die Task- und Speicherverwaltung Bestandteil der TCB, sondern auch nahezu alle Prozesse mit Supervisor-Rechten [168]. Von wesentlicher Relevanz für die Verifikation der Trusted Computing Base ist dabei deren Umfang, da eine Korrelation zwischen diesem und dem für eine Verifikation erforderlichen Aufwand unterstellt werden kann.

Da beim virtualisierten Konsolidierungsansatz alle Gastsysteme in ihren Rechten deprivilegiert sind, beinhaltet die Trusted Computing Base hier, wie in Abbildung 4.7 dargestellt, neben der Hardware lediglich den privilegierten Code des Hypervisors, der bei einem konsequenten Design entsprechend des Microkernel-Ansatzes einen geringen Umfang hat. Somit ist hier unter Umständen eine formale Verifikation der durch den virtualisierten Konsolidierungsansatz bereitgestellten Dienste möglich; bei einem einfachen Hypervisor [7] und einem Betriebssystem-Microkernel [99] konnten hier beispielsweise bereits entsprechende Erfolge erzielt werden. Die TCB des hybriden Konsolidierungsansatzes hat hingegen einen deutlich größeren Umfang in Form der Hardware sowie des kompletten Hypervisors und aller Gastsysteme, die im *Privileged Mode* und somit mit vollen Rechten ausgeführt werden. Dies trifft in der Regel auf die integrierten Echtzeitsysteme zu, die, wie zuvor erwähnt, wiederum selbst häufig auf eine sichere Rechteverwaltung zugunsten einer höheren Echtzeitfähigkeit verzichten. Als Konsequenz werden somit häufig auch die Tasks des Echtzeitsystems mit vollen Rechten ausgeführt und sind deshalb auch Bestandteil der TCB. Zudem ist die in Abschnitt 3.2 beschriebene Konstellation zu betrachten, in der einem OEM durch den Steuerungshersteller die Möglichkeit geboten wird, weitere Gastbetriebssysteme mit eigenen Applikationen in die bereitgestellte Konsolidierungslösung zu integrieren. In diesem Fall würde die TCB beim hybriden Konsolidierungsansatz auch den Code der nachträglich integrierten Systeme umfassen, sofern diese im *Privileged Mode* ausgeführt werden müssen. Da sich eine Verifikation dieses Codes den Möglichkeiten des Steuerungshersteller entzieht, scheint eine formale Verifikation der Zuverlässigkeit des hybriden Konsolidierungsansatzes mit den heute bekannten Mitteln somit ausgeschlossen.

Lässt sich die Zuverlässigkeit der vom Hypervisor bereitgestellten Isolation zwischen den Gastsystemen als Resultat einer entsprechenden Verifikation zertifizieren, so bringt dies in Verbindung mit einer entsprechend ausgelegten Hardware auch potentielle Vorteile bei der Safety-Zertifizierung eines konsolidierten Systems mit Komponenten unterschiedlicher *Sicherheits-Integritätslevel (Safety Integrity Level, SIL)* mit sich. Dies kann beispielsweise eine sicherheitskritische Steuerungsapplikation und eine HMI sein. Generell wird hier durch die für eine

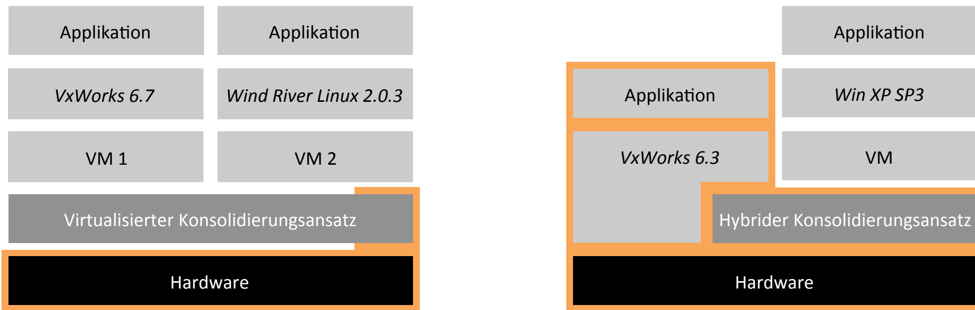


Abbildung 4.7: Darstellung der Konsolidierungsansätze unter dem Aspekt der *Trusted Computing Base (TCB)* (orange markiert)

Zertifizierung der funktionalen Sicherheit im Bereich der Automatisierung relevante Norm *IEC 61508* [133] gefordert, dass alle Komponenten des Systems in gleicher Weise zu zertifizieren sind wie die Komponente mit dem höchsten Sicherheits-Integritätslevel. Davon ausgenommen ist nur der Fall, dass eine räumliche und zeitliche Isolation zwischen Systemkomponenten mit unterschiedlichen Sicherheits-Integritätsleveln nachgewiesen oder zumindest eine Verletzung dieser Isolation detektiert und in kontrollierter Weise behandelt werden kann [133]. Als Konsequenz wäre im Fall eines Hypervisors mit einer zertifizierten Isolation der Gastsysteme nur die Zertifizierung der Komponenten erforderlich, die die eigentlichen Safety-Funktionen implementieren. Weitere, davon isolierte und nicht in das Safety-Konzept involvierte Funktionsbereiche wie die HMI oder ein weiteres Echtzeitsystem des OEM müssten im Fall einer nachweislich sicheren Partitionierung des Hypervisors nicht berücksichtigt werden. So ließen sich auch Änderungen an diesen Bereichen des Gesamtsystems durchführen, ohne zugleich die eigentliche Safety-Implementierung einer erneuten Zertifizierung unterziehen zu müssen. Weiterhin wäre eine zertifizierte Isolation der Gastsysteme durch den Hypervisor auch eine wesentliche Voraussetzung für das in Abschnitt 3.2 beschriebene Szenario, das die Realisierung einer zweikanaligen Safety-SPS auf einer Multicore-CPU beschreibt.

Kapitel 5

Entwicklung einer Methode zur Firmware-Parallelisierung

5.1 Ableitung und Abgrenzung der Vorgehensweise

In Abschnitt 3.1.2 wurden die beiden zentralen Merkmale einer Firmware-Parallelisierung im Bereich der Steuerungstechnik abgeleitet. Zu diesen zählt die Tatsache, dass der Parallelisierung eines bestehenden Systems, in das langjährige Entwicklungsaufwände geflossen sind, im Allgemeinen der Vorzug gegenüber der vollständigen Neuentwicklung einer parallelen Firmware-Architektur gegeben wird. Weiterhin wurde ein partitioniertes Scheduling mit statisch zum Entwicklungszeitpunkt definierten Task-Verteilungen und Task-Dekompositionen als geeignete Strategie identifiziert, um sowohl den Systemdeterminismus zu erhöhen als auch den Parallelisierungsaufwand zu reduzieren.

5.1.1 Motivation und Zieldefinition

Der Wahl einer konkreten Task-Verteilung und -Dekomposition liegt üblicherweise ausschließlich das Wissen der Entwickler über die Architektur der von ihnen entwickelten Firmware zugrunde. Diese Informationen stellen jedoch bei weitem keine zufriedenstellende Entscheidungsgrundlage für das weitere Vorgehen bei der Parallelisierung dar: Aufgrund der Systemkomplexität besitzen einzelne Entwickler meist nur hinsichtlich ausgewählter Firmware-Module detailliertes Wissen und somit eine lokale Systemsicht, während bezüglich der globalen Wechselwirkungen im Gesamtsystem in der Regel nur ein abstraktes Verständnis besteht. Zudem lassen sich die für die Evaluation verschiedener Parallelisierungsalternativen erforderlichen Erkenntnisse über das Laufzeitverhalten einer Automatisierungs-Firmware oft erst im Rahmen einer Interaktion der Firmware mit einer realen Umgebung in Form einer automatisierungstechnischen Anlage gewinnen. Dabei besitzt die Lastcharakteristik der Firmware, wie in Abschnitt 3.1.2 dargestellt, eine hohe Dynamik bezüglich unterschiedlicher Maschinentypen und Maschinenkonfigurationen. Somit lässt sich die zu erwartende Leistungsfähigkeit eines spezifischen parallelen Firmware-Designs unter verschiedenen Lastprofilen nur sehr schwer prognostizieren, so dass die naheliegende Vorgehensweise aus einer experimentellen Implementierung eines parallelen Designs und einer anschließenden Performanzanalyse an realen Maschinen besteht. Dies bringt jedoch zwei essentielle Probleme mit sich: Zum einen ist es sehr unwahrscheinlich, dass auf diese Weise aus den zuvor genannten Gründen bereits die optimale Parallelisierung des Systems gefunden wird, zum anderen ist aber auch nicht bekannt,

welche Performanzsteigerung auf einer vorgegebenen Zielplattform theoretisch überhaupt möglich wäre. Somit müssten eine Vielzahl experimenteller Parallelisierungen implementiert und evaluiert werden, was aufgrund der anfallenden Entwicklungsaufwände in der Realität nicht praktikabel ist.

Als Konsequenz dieser Randbedingungen wird im Rahmen dieser Arbeit eine Methode spezifiziert, die eine Parallelisierung einer automatisierungstechnischen Firmware für homogene Multicore-Architekturen unterstützt. Der Anspruch dieser Methode besteht darin, den Workflow von der Analyse des bestehenden Systems über dessen Optimierung bis hin zur Implementierung der Parallelisierung so umfassend wie möglich zu unterstützen. Dabei kann diese Methode insofern als anwendungsorientiert bewertet werden, als bei deren Entwicklung eine der zentralen Prämissen darin bestand, die im Rahmen eines Einsatzes anfallenden Aufwände zu reduzieren. Der Kern der Methode besteht dabei aus einer Exploration und Evaluation effizienter Parallelisierungsalternativen einer automatisierungstechnischen Firmware auf Basis geeigneter Systemmodelle. Dabei ist hervorzuheben, dass die Methode zwar durch die spezifischen Anforderungen der Automatisierungstechnik motiviert ist, sich aber in weiten Teilen auch für eine Vielzahl weiterer Software- und Firmware-Systeme nutzen lässt.

Diese in Abbildung 5.1 in einem Überblick skizzierte Methode stellt sich wie folgt dar: Da das parallele System hier nicht von Grund auf neu, sondern mittels einer Adaption einer bestehenden Implementierung entwickelt wird, darf das Modell im Gegensatz zu der in Abschnitt 2.5.1 beschriebenen modellbasierten Entwicklung nicht auf Basis einer abstrakten Systemspezifikation erstellt werden. Stattdessen muss es die existierende Implementierung auf einer geeigneten Abstraktionsebene abbilden. Da eine Systemmodellierung auf Basis statischer Informationen über das zu parallelisierende System aufgrund dessen Laufzeitdynamik nicht zielführend ist, wird dabei eine dynamische Systemanalyse durchgeführt. In diesem Rahmen werden umfangreiche Daten bezüglich des Laufzeitverhaltens eines Systems mittels eines Profilings unter Ausführung der Firmware an realen Maschinen aufgezeichnet. Aus diesen Laufzeitdaten werden unter Einbeziehung des Expertenwissens von Entwicklern in einem weiteren Schritt geeignete Modelle der Firmware in Form von Graphen G_i auf unterschiedlichen Modellierungsebenen extrahiert.

Definition 5 (Modellierungsebene) *Als Modellierungsebene sei die Granularitätsstufe bezeichnet, die in einem spezifischen Modell der Firmware abgebildet wird.*

Als Konsequenz aus den in Abschnitt 3.1.2 beschriebenen Strategien der Firmware-Parallelisierung in der Steuerungstechnik wird im Rahmen der in dieser Arbeit entwickelten Methode zwischen einer Modellierung auf System- und Taskebene differenziert.

Definition 6 (Systemebene) *Eine Modellierung auf Systemebene bildet eine vollständige Multitasking-Firmware mit dem Ziel ab, Alternativen eines partitionierten Schedulings zu explorieren und zu evaluieren. Als Modell wird ein sogenannter* Task-Graph *gewählt.*

Definition 7 (Taskebene) *Eine Modellierung auf Taskebene bildet eine Task einer Multitasking-Firmware mit dem Ziel ab, deren alternative Dekompositionen in eine Menge neuer Tasks zu explorieren und zu evaluieren. Als Modell wird ein sogenannter* Codeblock-Graph *gewählt.*

Da die Lastprofile der Firmware und somit auch die Modelle auf System- und Taskebene spezifisch für die jeweiligen Maschinentypen und deren Konfigurationen sind, sollte hierbei ein

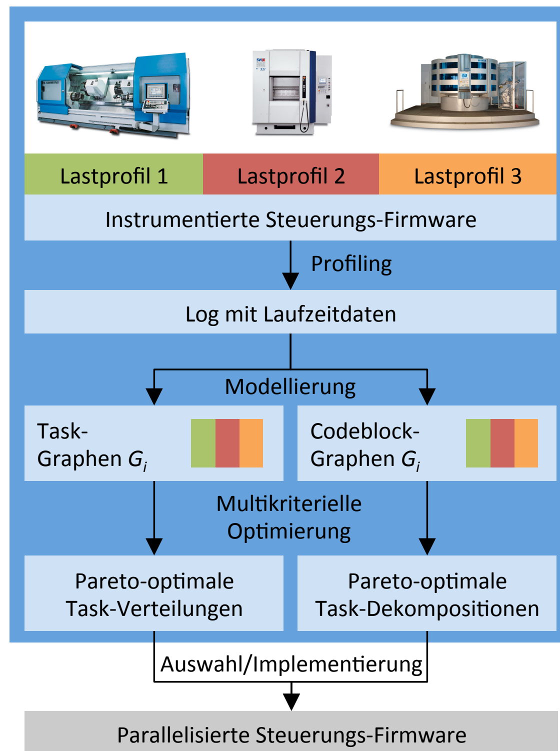


Abbildung 5.1: Der Prozess zur Entwicklung einer parallelen Steuerungs-Firmware aus einer bestehenden Firmware. Die Schritte, die von der in der vorliegenden Arbeit entwickelten Methode zur Exploration und Evaluation effizienter Parallelisierungsalternativen abgedeckt werden, sind blau hinterlegt *(Bilder: © Bosch Rexroth AG 2014)*.

Profiling unter verschiedenen Lastprofilen erfolgen; in Abbildung 5.1 sind exemplarisch die Lastprofile 1, 2 und 3 dargestellt. Die Menge der durch das Profiling generierten Modelle G_i bildet nun zunächst die Grundlage für eine Bewertung möglicher Alternativen der Parallelisierung für eine vorgegebene Anzahl an CPU-Kernen. Da es sich bei den zuvor genannten Modellen um Graphen handelt, ist dies in analytischer Weise möglich. Dabei wird zu jeder potentiellen Lösung eine dreidimensionale und somit multikriterielle Bewertung definiert, auf die in den nachfolgenden Abschnitten detailliert eingegangen wird.

Wenngleich eine Bewertung manuell definierter Parallelisierungsalternativen auf Basis der abgeleiteten Modelle möglich ist, besteht das wesentliche Ziel in der automatisierten Exploration der Pareto-Optima des Entwurfsraums paralleler Firmware-Designs. Dabei handelt es sich um das in Abschnitt 2.8 beschriebene Problem einer globalen, multikriteriellen Optimierung. Besonders erschwert wird diese durch die Tatsache, dass der Entwurfsraum möglicher paralleler Firmware-Designs selbst bei Systemen geringer Komplexität enorm groß ist und einem exponentiellen Wachstum unterliegt. So gibt es beispielsweise bereits für die Allokation einer

typischen, aus 50 Tasks bestehenden Automatisierungs-Firmware auf einer CPU mit 4 Kernen $4^{50} \approx 1,27 \cdot 10^{30}$ Möglichkeiten. Als Konsequenz ist eine erschöpfende Durchmusterung des Entwurfsraums nicht zu leisten, so dass geeignete Verfahren erforderlich sind, die innerhalb eines vertretbaren Berechnungszeitraums ausreichend gute Problemlösungen herleiten. Eine besondere Herausforderung der gewählten Zielstellung stellt die Tatsache dar, dass jede Lösung in Form eines konkreten statischen parallelen Firmware-Designs unter einer Vielzahl von Problemausprägungen in Form von Firmware-Modellen G_i bewertet werden muss. Um ein derartiges Problem zu lösen, sind verschiedene Möglichkeiten denkbar. So können die Problemausprägungen zunächst auf ein einziges Problem reduziert werden, auf das sich schließlich entsprechende Lösungsmethoden anwenden lassen. Im Fall der Graph-basierten Modelle G_i würde dies der Generierung eines einzigen Graphen entsprechen, der die Merkmale der Graphen G_i aggregiert. Alternativ dazu kann die Generierung potentieller Lösungen zunächst unabhängig von den Problemausprägungen erfolgen, so dass diese erst bei der Bewertung einer Lösung Berücksichtigung finden. Dies lässt sich so gestalten, dass eine Lösung zunächst unter jedem Graphen G_i separat bewertet wird und abschließend die einzelnen Bewertungen in geeigneter Weise aggregiert werden, um die Eignung der Lösung für alle Problemausprägungen zu bewerten. Dies ist zugleich der Ansatz, den auch die im Rahmen dieser Arbeit entwickelte Methode nutzt, da die Aggregation metrischer Bewertungen die einfache Möglichkeit einer Gewichtung der einzelnen Graph-basierten Modelle entsprechend ihrer Relevanz bietet.

In Abschnitt 2.9 dieser Arbeit wurden genetische Algorithmen als eine Methode zur Exploration exponentiell wachsender Entwurfsräume innerhalb einer akzeptablen Rechenzeit vorgestellt. Diese stellen zudem ein geeignetes Mittel für die bei diesem Problem geforderte multikriterielle Optimierung dar und ermöglichen zugleich die problemübergreifende Generierung und problemabhängige Bewertung potentieller Lösungen. Als Konsequenz setzt die in der vorliegenden Arbeit entwickelte Methode genetische Algorithmen zur Exploration effizienter Parallelisierungsalternativen ein. Dabei erfolgt die Evaluation potentieller Task-Verteilungen und -Dekompositionen stets unter allen zuvor generierten Firmware-Modellen G_i, so dass der genetische Algorithmus die entsprechende Parallelisierung stets hinsichtlich dieser Lastprofile optimiert. Infolgedessen hat die Auswahl der Lastprofile, unter denen die Modelle G_i gewonnen werden, zentralen Einfluss auf die erzielten Ergebnisse, so dass sich die Wahl von Szenarien empfiehlt, die für den Einsatzzweck der Steuerung repräsentativ sind.

Das Wissen über die zu erwartenden multikriteriellen Performanzwerte der durch den genetischen Algorithmus als Pareto-optimal identifizierten Lösungen stellt schließlich die Grundlage für eine Entscheidung hinsichtlich des weiteren Vorgehens bei der Firmware-Parallelisierung dar. Dazu muss zunächst entsprechend der jeweiligen Präferenzen bezüglich der Performanzkriterien eine Lösung ausgewählt werden, bevor mit deren Implementierung im Realsystem entsprechend der durch die Lösung beschriebenen Weise begonnen werden kann.

5.1.2 Vorgehensmodell der Modellierung

Die Definition geeigneter Vorgehensmodelle ist nicht nur in der Software-Entwicklung, sondern auch in der Modellentwicklung und -analyse ein geeignetes Mittel, um Fehler frühzeitig zu identifizieren und die Validierung und Verifikation des gewählten Ansatzes zu unterstützen (vgl. Abschnitt 2.6). Da die im Rahmen dieser Arbeit entwickelte Methode auf der Generierung und Auswertung von Firmware-Modellen basiert, ist die nachfolgende Definition eines korrek-

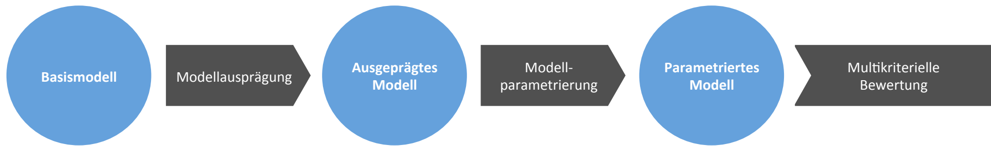


Abbildung 5.2: Prozess der Ausprägung und Parametrierung eines Basismodells

ten Vorgehensmodells eine wesentliche Voraussetzung für die kommenden Schritte. In diesem Zusammenhang findet auch eine Zuordnung der weiteren Abschnitte dieser Arbeit zu den einzelnen Phasen des Vorgehensmodells statt, um das Verständnis der Methodenbeschreibung zu erleichtern. Eine wesentliche Voraussetzung für die weitere Darstellung des Vorgehensmodells ist die Definition nachfolgender Termini, deren Beziehungen in Abbildung 5.2 dargestellt sind.

Definition 8 (Basismodell) *Als Basismodell einer konkreten Modellierungsebene sei die Spezifikation eines konkreten Graphentyps bezeichnet, welche die Definition aller Knoten- und Kantentypen des Graphen und die jeweils zu einem Graphenelement annotierbaren Daten umfasst.*

Definition 9 (Modellausprägung) *Als Modellausprägung sei der Vorgang bezeichnet, der auf Basis von in einem realen System aufgezeichneten Laufzeitdaten und von durch Experten formalisiertem Wissen über das System einen Graphen generiert, dessen Typ einem Basismodell entspricht. Im Rahmen der Ausprägung wird dabei die konkrete Anzahl und Art der Knoten und Kanten des Graphen sowie eine Teilmenge der zu diesen annotierten Daten definiert.*

Definition 10 (Ausgeprägtes Modell) *Als ausgeprägtes Modell sei das aus einer Modellausprägung in der zuvor definierten Weise resultierende Modell der Ausführung eines Systems unter einem konkreten Lastprofil bezeichnet.*

Definition 11 (Modellparametrierung) *Im Rahmen der Modellparametrierung wird ein ausgeprägtes Modell um alle Daten ergänzt, die eine konkrete Parallelisierungsvariante des modellierten Systems für einen Prozessor mit mehreren CPU-Kernen in eindeutiger Weise definieren.*

Definition 12 (Parametriertes Modell) *Als parametriertes Modell wird das aus einer Modellparametrierung in der zuvor definierten Weise resultierende Modell eines Systems bezeichnet. Ein parametriertes Modell liefert alle Informationen, um eine multikriterielle Bewertung der dadurch repräsentierten Parallelisierung eines Systems vornehmen zu können.*

Als Erweiterung des in Abschnitt 2.6 beschriebenen einfachen Vorgehensmodells von Robert F. Sargent [154] sei zunächst das in Abbildung 5.3 dargestellte Vorgehensmodell der Modellierung definiert. Die Motivation für eine Erweiterung des einfachen Modells ist die Tatsache, dass die spezifikationsgemäße Modellimplementierung im Fall der hier entwickelten Methode kein einmaliger Prozess ist, sondern durch eine Software in automatisierter Weise auf Basis von Laufzeitaufzeichnungen erfolgt.

Ausgangssituation des Vorgehensmodells ist das Realsystem, an dessen Parallelisierung eine Reihe von Anforderungen gestellt werden, die sich wiederum aus den Randbedingungen ergeben, unter denen das System entwickelt und eingesetzt wird. Diese Anforderungen sind in

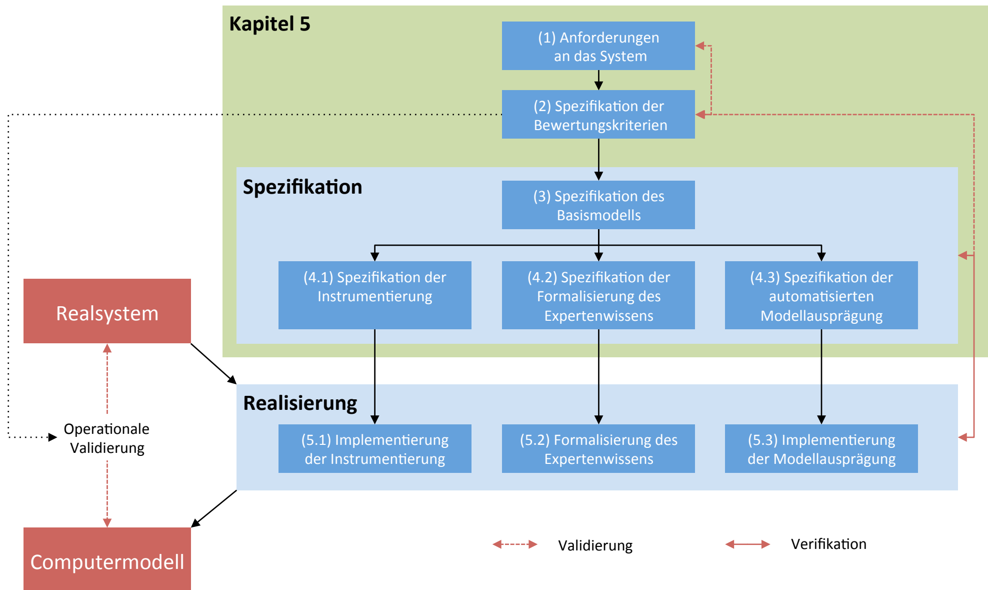


Abbildung 5.3: Das Vorgehensmodell der Modellentwicklung zur Exploration und Evaluation effizienter Parallelisierungsalternativen. Die Inhalte der in dieser Arbeit spezifizierten und in Kapitel 5 beschriebenen Methode sind entsprechend markiert.

Abbildung 5.3 mit (1) markiert. Für eine Evaluation verschiedener Modellparametrierungen gilt es nun in einem ersten Schritt, geeignete Bewertungskriterien und deren Quantifizierung mittels geeigneter Formeln zu spezifizieren (2). Diese Phase des Vorgehens wird für die Modellierung auf Systemebene in Abschnitt 5.4.1 und für die Modellierung auf Taskebene in Abschnitt 5.5.1 beschrieben. Die Validierung dieser Phase muss den Nachweis erbringen, dass die quantifizierten Kriterien geeignet sind, um eine Modellparametrierung im Hinblick auf die zuvor definierten Anforderungen zu bewerten.

Nun gilt es, aus der Spezifikation der Bewertungskriterien eine geeignete Vorgehensweise zur Modellierung abzuleiten. Dies umfasst unter anderem die Spezifikation von Basismodellen (3), die für die Evaluation einer Ausprägung und Parametrierung unter den zuvor spezifizierten Kriterien geeignet sind. Hierbei handelt es sich je nach Modellierungsebene um den *Task-Graphen* (vgl. Abschnitt 5.4.2) oder den *Codeblock-Graphen* (vgl. Abschnitt 5.5.2). Die Spezifikation dieser Basismodelle bildet nun die Grundlage einer Reihe weiterer Spezifikationen, beginnend mit der einer geeigneten Systeminstrumentierung (4.1) für das in Abschnitt 5.1.1 beschriebene Profiling. Dies erfordert, alle relevanten Ereignisse und Vorgänge innerhalb des Realsystems zu identifizieren, die für eine Ausprägung eines Basismodells mit den definierten Daten notwendig sind. Für die Modellierung auf Systemebene wird diese Phase in Abschnitt 5.4.3 und analog dazu für die Taskebene in Abschnitt 5.5.3 beschrieben. Ausgehend von den Basismodellen wird zudem die Formalisierung des Expertenwissens (4.2) spezifiziert. Schließlich wird die Spezifikation eines Programms abgeleitet, das die automatisierte Modellausprägung (4.3) mit den

Daten des Profilings und dem formalisierten Expertenwissen realisiert. Jeweils beide Schritte des Vorgehensmodells werden für die Modellierung auf Systemebene in Abschnitt 5.4.4 und für die Modellierung auf Taskebene in Abschnitt 5.5.4 beschrieben.

Schließlich ist im Rahmen einer Validierung der Nachweis zu erbringen, dass die zuvor spezifizierte Instrumentierung, Formalisierung und Modellausprägung geeignete Modelle der Firmware generieren, die eine für den Zweck der Methode ausreichend genaue Bewertung parametrierter Modelle unter den zuvor spezifizierten Kriterien ermöglichen. Diesen Schritt beschreibt Abschnitt 5.4.5 für die Systemebene und Abschnitt 5.5.5 für die Taskebene. Gemäß Abschnitt 5.1.1 liegt der Fokus dieser Arbeit auf der Entwicklung eines geeigneten methodischen Vorgehens, das somit alle zuvor genannten Schritte des Vorgehensmodells umfasst. Dies ist auch in Abbildung 5.3 visualisiert.

Im Rahmen einer Realisierung ist schließlich eine Implementierung der zuvor spezifizierten Konzepte zu leisten und im Anschluss daran zu verifizieren, dass diese der zuvor definierten Spezifikation entspricht. Dies betrifft die Instrumentierung (5.1) und die Formalisierung des Expertenwissens (5.2) ebenso wie die automatisierte Modellausprägung (5.3). Diese Realisierungen sind allerdings stets für ein Betriebssystem oder eine konkrete Firmware spezifisch und somit nicht generisch leistbar. Abschnitt 5.3 stellt zwar mit der *EEEPA*-Toolchain eine prototypische Implementierung der automatisierten Modellausprägung vor, die schließlich im Rahmen der in Kapitel 6 beschriebenen Fallstudie zum Einsatz kommt. Da aber diese Realisierung nicht der Spezifikation des methodischen Vorgehens zuzuordnen ist, kommt ihr in dieser Arbeit nur eine untergeordnete Rolle zu, so dass in diesem Kontext auf eine Verifikation verzichtet wird.

Der letzte Schritt des Vorgehensmodells sieht nun in Anlehnung an das Modell von Sargent die operationale Validierung des ausgeprägten Modells vor. Dabei ist der Nachweis zu erbringen, dass die auf Basis eines ausgeprägten und parametrierten Modells erstellten Prädiktionen mit dem Verhalten des Realsystems eine ausreichende Übereinstimmung aufweisen. Eine operationale Validierung eines Modells kann mittels verschiedener Methoden wie beispielsweise dem Vergleich mit anderen, bereits validierten und verifizierten Modellen oder der statistischen Auswertung von Vergleichen zwischen Prädiktionen des Modells und experimentell gewonnenen Resultaten des realen Systems erfolgen [154]. Ein Vergleich empfiehlt sich dabei generell unter den zu Beginn spezifizierten Metriken, sofern die zur Berechnung erforderlichen Daten im Realsystem beobachtbar sind. Eine operationale Validierung der zuvor entwickelten Methode ist allerdings nicht Gegenstand dieser Arbeit, da alle im Rahmen der Fallstudie generierten Modelle auf den zuvor beschriebenen Realisierungen basieren. Da diese allerdings nicht verifiziert wurden, sind auch die Voraussetzungen für eine operationale Validierung nicht erfüllt. Im Rahmen eines Ausblicks auf noch offene Themenstellungen werden in Kapitel 7.3 die im Rahmen einer operationalen Validierung zu leistenden Aufgaben diskutiert.

5.2 Verwandte Arbeiten und Stand der Technik

Die Software-Parallelisierung ist kein junges Forschungsgebiet, sondern hat bereits in den vergangenen Jahrzehnten viel Aufmerksamkeit erfahren. Dabei haben sich vor allem die frühen Arbeiten häufig auf die Domäne des wissenschaftlichen Rechnens und somit auf die Extraktion von Parallelität auf Datenebene fokussiert [105]. Das Forschungsgebiet hat jedoch in den vergangenen Jahren vor allem infolge der zunehmenden Etablierung von Multicore-Prozesso-

ren, die eine Parallelisierung auf Taskebene erfordern, erneut großes Interesse geweckt. Dieser Abschnitt präsentiert eine Auswahl neuerer verwandter Arbeiten im Bereich der Software-Parallelisierung, ohne sich dabei auf Lösungen zu beschränken, die eine Firmware-Parallelisierung unter den Rahmenbedingungen der Automatisierungstechnik unterstützen. Stattdessen werden auch explizit auf andere Anwendungsdomänen spezialisierte Lösungen vorgestellt, sofern diese in der Wahl ihrer Mittel eine entsprechende Nähe zu der im Rahmen dieser Arbeit entwickelten Methode aufweisen. So haben beispielsweise einige der vorgestellten Arbeiten statt eines Multicore-Prozessors ein sogenanntes *MPSoC (Multi-Processor-System-on-Chip)* als Zielarchitektur, das sich mit einer Vielzahl potentiell heterogener *Verarbeitungseinheiten (Processing Elements, PEs)* durch einen signifikant höheren Parallelitätsgrad auszeichnet. Zugleich haben jedoch alle vorgestellten Arbeiten als gemeinsamen Ausgangspunkt stets eine in der Programmiersprache *C* realisierte Implementierung der zu parallelisierenden Software, so dass keine darüber hinausgehenden Modelle oder Spezifikationen der Programme als existent vorausgesetzt werden.

Die Rekonstruktion der Kommunikation zwischen Threads in bereits mehrfädig implementierten Linux-Applikationen ist die Anwendungsdomäne der Software *CETA* [109]. Das Tool kann gerichtete Kommunikationsgraphen extrahieren, deren Knoten die Threads und deren gerichtete und gewichtete Kanten den Umfang der Thread-Kommunikation mittels Lese- und Schreibzugriffen auf gemeinsame Speicherbereiche abbilden. Somit lässt sich prinzipiell auch die Kommunikation mittels Betriebssystemmechanismen wie Messages rekonstruieren, sofern diese als Zugriffe auf gemeinsame Speicherbereiche implementiert sind. Die dynamische Rekonstruktion der Graphen erfordert allerdings die Ausführung der Applikation unter dem Hardware-Simulator *Simics* [112], so dass die chronologische Aufzeichnung aller Lese- und Schreibzugriffe auf den Speicher möglich ist. Um diese einem spezifischen Thread zuzuordnen, wird jeder Aufruf der Betriebssystemfunktion, die einen Kontextwechsel auslöst, aufgezeichnet und die entsprechende Prozess-ID extrahiert. Die auf diese Weise gesammelten Daten werden schließlich mittels Skripten zu Kommunikationsgraphen unterschiedlichen Typs aggregiert.

Auf die Evaluation grobgranularer Alternativen der funktionalen Parallelisierung eingebetteter Streaming-Applikationen für potentiell heterogene Multiprozessorarchitekturen fokussiert sich das *SPRINT*-Tool [43]. Dabei werden aus sequentiellem *C*-Code in semiautomatischer Weise nebenläufige, funktional äquivalente *SystemC-Transaktionsebenenmodelle* [68] generiert. Die jeweils zu evaluierende Partitionierung eines sequentiellen *C*-Programms in nebenläufige Aufgaben wird dabei durch den Entwickler in einer separaten Datei mittels entsprechender Direktiven definiert, welche Funktionsbezeichner und Marken im Code referenzieren. Die Transformation des *C*-Programms in das *SystemC*-Modell erfolgt schließlich mittels einer statischen Codeanalyse, die aber in [43] nicht weiter detailliert wird. Da die generierten Modelle zunächst keine temporale Dimension besitzen, ist für eine quantitative Evaluation einer spezifischen Partitionierung zunächst die Annotation von Berechnungs- und Kommunikationszeiten erforderlich. Erstere können beispielsweise mittels einer statischen oder dynamischen Codeanalyse oder in Form von Entwicklerabschätzungen ergänzt werden, während letztere ein Modell der Hardware-Plattform erfordern. Die Kommunikation zwischen den Aufgaben wird im *SystemC*-Modell in Form sogenannter *Kanäle* abgebildet, die im Rahmen der statischen Analyse des sequentiellen Codes auf Basis von Variablenreferenzen unter Anwendung einer Pointer-Analyse nach Andersen [8] extrahiert werden. Abschließend kann ein durch *SPRINT* generiertes Transaktionsebenenmodell eine Verhaltensspezifikation darstellen, auf Basis derer die Aufgaben in Software-Prozesse oder FPGA-Konfigurationen überführt werden.

Eine mittels einer Bibliothek wie *OpenMP*, *Cilk* oder *Intel Threading Building Blocks* realisierte Parallelisierung sequentieller *C*-Programme wird durch das Tool *Prospector* [95] unterstützt. Dieses deckt mit der Rekonstruktion von Abhängigkeiten im Code, einer modellbasierten *Speedup*-Prognose und automatisch abgeleiteten Implementierungsratschlägen wesentliche Schritte des Parallelisierungsprozesses ab. Die abschließende Implementierung der Parallelisierung mittels der zuvor genannten Bibliotheken bleibt jedoch die Aufgabe des Entwicklers. Für die Abhängigkeitsanalyse wurde im Kontext von *Prospector* das Tool SD^3 entwickelt, welches ausgehend von einer statischen Analyse des Eingabeformats in Form von Quellcode oder Binärdateien zunächst eine Instrumentierung vornimmt, um im Anschluss daran mittels eines dynamischen Profilings Statistiken der Schleifenausführung und Datenabhängigkeiten der Applikation zu extrahieren [96]. Diese Informationen können nun wiederum in dem ebenfalls im Kontext von *Prospector* entwickelten Tool *Parallel Prophet* Verwendung finden, das ausgehend von einem instrumentierten sequentiellen *C*-Programm Prognosen für den bei einer parallelen Ausführung erzielbaren Speedup generiert [97]. Die erforderlichen Annotationen umfassen dabei jeweils den Anfang und das Ende paralleler Abschnitte (z.B. Schleifen), paralleler Aufgaben (z.B. Schleifeniterationen) und gegebenenfalls unter wechselseitigem Ausschluss auszuführender Berechnungen. Nun wird aus dem *Trace* einer dynamischen Ausführung des instrumentierten Programms ein sogenannter *Programmbaum* generiert, der für jedes Paar von Annotationen einen Knoten definiert. Die Blätter des Baums stellen stets Berechnungen dar, die mit oder ohne wechselseitigen Ausschluss zu anderen Berechnungen ausgeführt werden können. Für die Gewichtung eines Knotens wird die Ausführungsdauer aller Instruktionen zwischen den beiden entsprechenden Annotationen vorgeschlagen, welche beim Profiling mittels des Timestamp-Counters der CPU ermittelt wird. Nun werden die Gewichte parallel ausführbarer Blätter mit einem Faktor für den Overhead einer parallelen Ausführung verrechnet, der sich aus einem einfachen Speicherperformanzmodell der Ziel-Hardware ergibt. Die Prädiktion des Speedups einer parallelen Ausführung des Codes erfolgt abschließend für eine vorgegebene Anzahl an CPU-Kernen und verschiedene Scheduling-Strategien der *OpenMP*-Bibliothek. Die Prognosen werden dabei entweder in analytischer Weise mittels einer Traversierung des Programmbaums abgeleitet oder unter einer per Parallelisierungsbibliothek realisierten Ausführung eines Programms gemessen, das als um die eigentlichen Berechnungen reduziertes Codeskelett aus dem Programmbaum generiert wurde.

Auf die automatische Extraktion von Abhängigkeitsgraphen für *OpenMP*-parallelisierte *C*-Programme fokussieren sich Larsen et al. [103, 104]. Statt einer dynamischen Codeanalyse befürworten sie eine statische Analyse, definieren aber zu deren Unterstützung eine *OpenMP*-Erweiterung in Form zweier Compiler-Direktiven, die durch einen Programmierer manuell im Code eingefügt werden müssen. Mittels dieser Direktiven soll die einer statischen Analyse inhärente konservative Überschätzung der Abhängigkeiten reduziert werden, um auf diese Weise exaktere Abhängigkeitsgraphen zu erhalten. So werden beispielsweise mittels der `depends`-Direktive die Datenabhängigkeiten einer Task zu anderen Tasks in expliziter Weise definiert. Dabei müssen für jede Task sowohl die als Eingangswerte erwarteten Zeigervariablen als auch die Bezeichner der Tasks, die auf diese Variablen zuvor schreibend zugegriffen haben, definiert werden. Weiterhin sind die Ausgangswerte einer Task und die Bezeichner aller auf diese Werte danach lesend zugreifenden Tasks ebenfalls explizit mittels `depends`-Direktiven zu deklarieren. Zur Validierung dieser umfangreichen Programmannotationen schlagen Larsen et al. schließlich Laufzeitprüfungen auf Basis einer instrumentierten Version des *C*-Programms vor.

Die Parallelisierung sequentieller *C*-Programme für MPSoCs hat das *MAPS*-Framework [40] sich zur Aufgabe gemacht, das allerdings keinen generischen parallelen *C*-Code generiert. Stattdessen wird für die sogenannte *Tightly-Coupled-Thread*-Plattform [192] spezifischer Code erzeugt, der potentielle Parallelität in Form entsprechender Codeannotationen deklariert. Die Partitionierung dieses Codes in parallele Threads, die Generierung entsprechender Kommunikations- und Synchronisationsmechanismen zwischen den Threads und das Mapping der Threads auf die PEs der *TCT*-Plattform können schließlich durch das *TCT*-Framework ebenso geleistet werden wie eine zyklengenaue Simulation und eine Speedup-Prädiktion. Der automatisierten Annotation des sequentiellen *C*-Codes durch *MAPS* gehen allerdings mehrere davon unabhängige Verarbeitungsschritte voraus. Zunächst wird mittels eines Compilers eine geeignete Zwischenrepräsentation des sequentiellen *C*-Codes erzeugt, auf Basis derer eine statische Kontroll- und Datenflussanalyse durchgeführt wird. Zugleich wird der *C*-Code in automatisierter Weise so instrumentiert, dass im Rahmen eines dynamischen Profilings die Ausführungsreihenfolge von Basisblöcken ebenso aufgezeichnet werden kann wie auf Zeigern basierende Speicherzugriffe, die wiederum für eine anschließende dynamische Datenflussanalyse genutzt werden. Anhand aller zuvor extrahierten Informationen wird schließlich ein spezieller gewichteter Kontroll-Datenfluss-Graph generiert, der als Knoten die einzelnen Anweisungen der Zwischenrepräsentation und als Kanten die zuvor extrahierten Kontroll- und Datenflüsse definiert. Mittels eines heuristischen Clustering-Algorithmus erfolgt nun nach dem *Bottom-Up*-Prinzip eine Agglomeration von Anweisungen zu sogenannten *Coupled Blocks*, im Rahmen derer sowohl die Dominanz- und Postdominanzbeziehungen der Anweisungen als auch der Umfang der Datenabhängigkeiten zwischen den Anweisungen berücksichtigt werden. Dieser Prozess wird iterativ durchgeführt und erhöht auf diese Weise sukzessive die Granularität der *Coupled Blocks*, die nach der Terminierung des Algorithmus direkt auf die Threads der finalen *TCT*-Annotation abgebildet werden.

Einen pragmatischen Ansatz zur Parallelisierung sequentieller, in *C* implementierter Streaming-Applikationen präsentieren Thies et al. [169] und zielen dabei auf die Ausnutzung von Pipeline-Parallelität ab. Dabei wird unterstellt, dass sich die Anwendung im eingeschwungenen Zustand ausschließlich in der zyklischen Ausführung einer Schleife oberster Ebene befindet, die sich in mehrere Partitionen in Form von Codeabschnitten unterteilen lässt. Diese Partitionen bilden nun die Stufen der Pipeline, wodurch sich selbst bei sogenannten *schleifengetragenen Datenabhängigkeiten (Do-Across-Schleifen)* eine parallele Ausführung mittels einer Überlappung von Schleifeniterationen erzielen lässt. Das Konzept sieht nun vor, dass die Partitionierung des sequentiellen Codes in manueller Weise erfolgt, indem mittels entsprechender Codeannotationen der Anfang und das Ende der Hauptschleife sowie die Grenzen der Pipeline-Stufen definiert werden. Dabei ist die Annotation in tieferen Ebenen geschachtelter Schleifen ebenso verboten wie die Nutzung ausgewählter, den Kontrollfluss beeinflussender Instruktionen wie `break` und `continue` innerhalb des annotierten Codes. Die Extraktion der Datenflüsse zwischen den einzelnen Partitionen erfolgt nun in dynamischer Weise, indem das annotierte Programm für eine ausgewählte Menge an Eingabewerten unter Nutzung des auf *dynamische Binärinstrumentierung (Dynamic Binary Instrumentation, DBI)* spezialisierten *Valgrind*-Frameworks [127] interpretiert wird. Dies ermöglicht das Profiling aller Lese- und Schreibzugriffe auf den Speicher und eine anschließende Rekonstruktion der Produzenten-Konsumenten-Relationen zwischen den Instruktionen respektive Pipeline-Partitionen. Allen weiteren Schritten wird nun die Annahme zugrunde gelegt, dass sich die Anwendungen dieser Domäne hinsichtlich ihres Datenflusses

während der kompletten Laufzeit durch eine hohe Regelmäßigkeit auszeichnen, so dass selbst ein kurzes Profiling alle relevanten Abhängigkeiten extrahieren kann. Aus den aufgezeichneten Daten wird nun ein sogenannter *Stream-Graph* generiert, dessen Knoten den mit ihrem Anteil an der Rechenzeit gewichteten Partitionen und dessen gewichtete Kanten den Datenflüssen zwischen den Partitionen entsprechen. Anhand dieses Graphen lässt sich nun durch den Entwickler die unter der gewählten Partitionierung erzielte Parallelität evaluieren. Im Anschluss an eine gegebenenfalls erforderliche Überarbeitung der Partitionierung kann nun das ursprüngliche *C*-Programm in automatisierter Weise um Makros ergänzt werden, welche die parallelen Prozesse und deren Interprozesskommunikation mittels Pipes definieren und auf diese Weise eine Pipeline-Parallelisierung entsprechend der gewählten Partitionierung realisieren.

In Erweiterung der Arbeit von Thies et al. präsentieren Rul et al. in [152] einen Ansatz für eine grobgranulare Pipeline-Parallelisierung, ohne dabei auf manuelle Annotationen der Schleifen und Pipeline-Stufen durch den Entwickler angewiesen zu sein. Die Vorgehensweise ist zunächst der von Thies et al. insofern sehr ähnlich, als im Rahmen eines dynamischen Profilings eines entsprechend instrumentierten Programms unter repräsentativen Eingabewerten Graphen generiert werden. Diese bilden die dynamischen Kontroll- und Datenflüsse des Programms ab, wobei als Knoten nicht manuell definierte Pipeline-Stufen, sondern zuvor mittels einer statischen Codeanalyse rekonstruierte Codeabschnitte in Form von Funktionen, Schleifen und Basisblöcken gewählt werden. Die Definition der Pipeline-Stufen erfolgt nun mittels eines Clusterings von Codeabschnitten zu nicht weiter partitionierten Arbeitspaketen, wobei hierfür die zuvor extrahierten Kontroll- und Datenflüsse maßgeblich sind. Die beim Clustering erzielten Ergebnisse werden nun dem Entwickler in Verbindung mit einer in [152] nicht weiter detaillierten Speedup-Prädiktion zur Evaluation, Verifikation und Selektion präsentiert. Abschließend wird der sequentielle *C*-Code entsprechend der Wahl des Entwicklers transformiert, indem die Threads der Pipeline-Parallelisierung ebenso wie deren Synchronisation und Kommunikation in automatisierter Weise generiert werden.

Ebenfalls in Erweiterung der Arbeit von Thies et al. beschreiben Tournavitis et al. in [171] einen sehr ähnlichen, ebenfalls auf dynamischem Profiling basierenden Ansatz zur Extraktion von Pipeline-Parallelität in sequentiellen *C*-Programmen. Zur Definition der Pipeline-Stufen auf Basis eines aus den Traces generierten Abhängigkeitsgraphen wird statt eines *Bottom-Up*-Clusterings allerdings eine iterative *Top-Down*-Partitionierung gewählt, womit grobgranularere Parallelisierungen erzielt werden sollen.

Auch wenn bislang keine Compiler existieren, die generischen *C*-Code auf Thread-Ebene in automatisierter Weise parallelisieren können, so haben sich doch in den vergangenen Jahren Nischen mit gewissen Einschränkungen entwickelt, in denen sich entsprechende Erfolge erzielen lassen. Mit dem sogenannten *Decoupled Software Pipelining (DSWP)* schlagen Ottoni et al. [137] einen automatisch parallelisierenden Compiler für sequentiellen *C*-Code vor, der Code mit feingranularer Pipeline-Parallelität generiert. Die geringe Granularität, die sich durch Pipeline-Stufen in der Größenordnung einzelner Instruktionen ergibt, erfordert jedoch eine effiziente Kommunikation und Synchronisation zwischen den parallelen Verarbeitungseinheiten. Auf dem gemeinsamen Hauptspeicher gängiger Architekturen basierende Betriebssystemmechanismen können dies nicht leisten, so dass hier spezifische, in Hardware realisierte Kommunikationsmechanismen erforderlich sind, beispielsweise in Form sogenannter *Synchronisations-Arrays* [145]. Ein wesentlicher Schritt des Kompilierens ist zunächst die Generierung eines Graphen, der alle Kontroll- und Datenabhängigkeiten als gerichtete Kanten zwischen Instruktionsknoten defi-

niert. Die hierfür erforderlichen Informationen werden mittels einer statischen Analyse des sequentiellen Codes generiert, so dass Abhängigkeiten generell konservativ und somit potentiell überschätzend abgebildet werden. Die auf dem Abhängigkeitsgraphen basierende Generierung der parallelen Threads zielt nun darauf ab, den kritischen Pfad der Ausführung vollständig in einen einzelnen Thread abzubilden, um die Laufzeit des parallelen Codes nicht durch Kommunikationslatenzen zu erhöhen. Für eine detaillierte Beschreibung der dabei eingesetzten Algorithmen sei auf [137] verwiesen.

Weiterhin präsentieren Rul et al. mit *Paralax* [176] einen automatisch parallelisierenden Compiler für sequentielle *C*-Programme mit äußeren Schleifen, der im Gegensatz zum *DSWP*-Konzept eine grobgranulare Pipeline-Parallelisierung zum Ziel hat. Die Analyse des Codes erfolgt prinzipiell in ausschließlich statischer Weise, muss allerdings durch einen Entwickler mittels Quellcode-Annotationen, die semantische Eigenschaften von Funktionen, Variablen und Funktionsargumenten beschreiben, unterstützt werden. So kennzeichnet beispielsweise eine `STATELESS`-Annotation eine Funktion als zustandslos und somit als nur mit den übergebenen Argumenten arbeitend. Weiterhin definieren die Annotationen `REF` und `MOD`, ob auf die Zeigerargumente einer Funktion lesend oder schreibend zugegriffen wird und `KILL` zeigt die Invalidierung des durch einen Zeiger referenzierten Speichers an. Da der Annotationsaufwand komplexer Programme sehr hoch sein kann, können Hinweise für das Setzen ausgewählter Annotationen wiederum mittels eines dynamischen Profilings abgeleitet und dem Entwickler unterbreitet werden. Bezüglich der exakten Vorgehensweise bei der anschließenden Codegenerierung bleibt [176] vage, die Autoren verweisen allerdings auf Algorithmen, die auch beim *DSWP*-Compiler zum Einsatz kommen.

Mit Fokus auf signalverarbeitende Applikationen präsentieren Baert et al. [12] respektive Mignolet et al. [118] im Kontext des *ATOMIUM*-Frameworks mit *MPA (MPSoC Parallelization Assist)* ein Tool für die semiautomatische Parallelisierung sequentieller *C*-Quellen für MPSoC-Plattformen. Zur Unterstützung der Codegenerierung muss dabei in einer separaten Datei eine sogenannte *Parallelisierungsspezifikation (ParSpec)* bereitgestellt werden, in der die zu realisierende Parallelisierung in abstrakter Weise definiert ist. Die Spezifikation sieht nach dem *Fork-Join*-Prinzip die Definition sogenannter *paralleler Sektionen* mit abschließenden Barrierensynchronisationen sowie Abschnitte der sequentiellen Verarbeitung vor. Innerhalb der parallelen Sektionen kann eine Parallelisierung sowohl auf funktionaler Ebene als auch auf Datenebene definiert werden, so dass entweder komplette Funktionen oder spezifische Iterationen einer Schleife als nebenläufige Ausführungsfäden gewählt werden können. Im letzteren Fall müssen jedoch die nebenläufigen Iterationen bereits statisch definiert werden, so dass eine Datenparallelisierung nur für Schleifen mit einer bereits im Voraus bekannten Anzahl an Iterationen möglich ist. Somit haben die *MPA*-Annotationen eine gewisse Ähnlichkeit mit *OpenMP*-Pragmas, setzen aber im Gegensatz zu *OpenMP* nicht die Unabhängigkeit der Berechnungen innerhalb der parallelen Abschnitte voraus. Unter Berücksichtigung der *ParSpec* erfolgt nun eine automatisierte Transformation des sequentiellen Codes in parallelen Code. Diesem Schritt geht fakultativ noch eine interaktive, durch ein Tool gesteuerte Transformation des *C*-Codes in sogenannten *CleanC*-Code voraus [119]. *CleanC* definiert dabei eine Reihe von Syntaxeinschränkungen und Programmierrichtlinien, die den Code gegenüber *C* in seiner Mächtigkeit reduzieren und auf diese Weise die statische Analysierbarkeit verbessern. Im Rahmen der automatisierten Parallelisierung werden nun zunächst Thread-Funktionen für die parallelen Ausführungsfäden generiert. Zuvor globale Daten werden dabei entweder in den Threads mittels lokaler Kopien repliziert und über

FIFO-Kanäle zwischen den Threads kommuniziert oder sie bleiben global und eine entsprechende Synchronisation wird manuell implementiert. Für das Thread-Management sowie die Thread-Kommunikation und -Synchronisation wird die API einer speziellen *MPA*-Laufzeitbibliothek genutzt, die Plattformdienste abstrahiert und somit für eine eigene Plattform entsprechend portiert werden muss. Die Transformation selbst basiert auf einer statischen Datenflussanalyse aller skalaren und nicht manuell synchronisierten Variablen des sequentiellen *C*-Codes. Abschließend kann mittels einer auf dynamischen Profiling-Resultaten des sequentiellen Codes basierenden Simulation des parallelen Codes eine Evaluation der Parallelisierungsspezifikation für eine spezifische Plattformspezifikation erfolgen, aus der sich gegebenenfalls Adaptionen der *ParSpec* ableiten lassen.

Statt die für das *MPA*-Tool erforderlichen Parallelisierungsspezifikationen manuell zu definieren, werden durch Cordes et al. Methoden der *ganzzahligen linearen Programmierung (Integer Linear Programming, ILP)* genutzt, um diese Beschreibungen in automatisierter Weise sowohl für feingranulare Pipeline-Parallelität [44] als auch für grobgranulare Task-Parallelität [46] zu generieren. Die ILP-basierte Parallelisierung arbeitet dabei auf einer geeigneten Zwischenrepräsentation in Form eines hierarchischen Task-Graphen, der zwei Arten von Knoten definiert: Einzelne *C*-Anweisungen repräsentierende einfache Knoten und hierarchische Knoten, die komplexe Kontrollstrukturen wie Schleifen oder Funktionsrümpfe abbilden. Jeder hierarchische Knoten enthält zudem je einen Knoten für eingehende respektive ausgehende Datenflüsse sowie eine beliebige Anzahl einfacher oder erneut hierarchischer Knoten. Kontrollflüsse sind somit mit Ausnahme spezieller Sprunganweisungen wie `break` bereits implizit über die Hierarchie repräsentiert. Im Anschluss an aus dem Compilerbau bekannte Optimierungen des sequentiellen *C*-Codes wie beispielsweise dem Abrollen von Schleifen erfolgt nun die automatisierte Generierung eines hierarchischen Task-Graphen. Zu diesem Zweck wird eine Kombination aus einer statischen Analyse und einem dynamischen Profiling des sequentiellen Codes angewandt. Dabei werden sowohl die Ausführungszeiten der Knoten als auch die Datenflüsse und deren Umfang extrahiert, wobei Cordes et al. die dabei angewandte Vorgehensweise nicht weiter detaillieren. Neben der ganzzahligen linearen Programmierung schlagen Cordes et al. in [45] auch einen auf genetischen Algorithmen basierenden Ansatz zur automatisierten Extraktion von Task-Parallelität in sequentiellen *C*-Programmen vor, der auf dem in Abschnitt 5.3.4 beschriebenen *PISA*-Framework [21] basiert. Dabei erfolgt eine multikriterielle Optimierung potentieller Lösungen stets hinsichtlich einer auf einfachen Modellen basierenden Bewertung der Ausführungsdauer, des Energieverbrauchs und des Kommunikations-Overheads. Der genetische Algorithmus generiert nach dem Bottom-Up-Prinzip für die Ebenen des hierarchischen Task-Graphen iterativ Pareto-optimale Zuordnungen von Knoten zu Tasks und liefert schließlich als Ergebnis eine hinsichtlich der zuvor genannten Kriterien Pareto-optimale Menge an Parallelisierungsspezifikationen für das *MPA*-Tool.

Ein Tool zur Unterstützung der Explorations- und Planungsphase bei der Parallelisierung sequentiellen *C*-Codes präsentieren Garcia et al. mit *Kremlin* [60]. Die Analyse arbeitet dabei ausschließlich mit unmodifiziertem Quellcode, der in geeigneter Weise durch einen Compiler instrumentiert und anschließend im Rahmen einer dynamischen Analyse unter repräsentativen Eingabewerten ausgeführt wird. Basierend auf den dabei generierten Daten stellen nun *Kritische-Pfad-Analysen (Critical Path Analyses, CPAs)* des sequentiellen Codes eine zentrale Komponente des *Kremlin*-Tools dar. Eine CPA generiert zunächst einen Abhängigkeitsgraphen für den betrachteten Code, der Instruktionen als mit ihrer Laufzeit annotierte Knoten und

Kontroll- und Datenabhängigkeiten als Kanten abbildet. Nun wird der für diesen Code maximal erzielbare Speedup als Quotient aus dessen sequentieller Ausführungsdauer und der Länge des kritischen Pfades des Abhängigkeitsgraphen bestimmt. Dabei bleiben somit Ressourcenbeschränkungen ebenso unberücksichtigt wie jeglicher Overhead in Form von Kommunikations- und Synchronisationslatenzen. Zudem wird durch eine CPA für den kompletten Code nur ein einziger und nicht ein auf einzelne Codeabschnitte bezogener skalarer Speedup-Faktor ermittelt. In Summe führen diese Faktoren dazu, dass zielgerichtete Hinweise für eine Parallelisierung des Codes unter Ressourcenbeschränkungen nicht möglich sind. Aus diesem Grund schlagen Garcia et al. eine *Hierarchische-Kritische-Pfad-Analyse (Hierarchical Critical Path Analysis, HCPA)* vor, die auf mehrfachen CPAs basiert. Dabei wird eine unabhängige CPA für einzelne Codeabschnitte wie Schleifen oder Funktionen durchgeführt und dabei jeweils deren sogenannte *Selbstparallelität* ermittelt. Dieser Wert beschreibt den um den Einfluss tieferer Ebenen der Programmhierarchie bereinigten Parallelitätsgrad eines Codeabschnitts und berechnet sich dabei als Quotient aus der kumulierten Länge der kritischen Pfade aller untergeordneten Abschnitte und der Länge des kritischen Pfads des jeweils betrachteten Abschnitts. Nun werden mittels einer Bottom-Up-Durchmusterung der Codeabschnitthierarchie die im parallelen Fall erzielbaren Ausführungszeiten der Codeabschnitte unter einem vereinfachten Plattformmodell approximiert. Dieser Schritt erfolgt unter Berücksichtigung der sequentiellen Ausführungszeiten, einem plattformspezifischen Verrechnungsfaktor für den Overhead und dem auf einer konkreten Hardware-Architektur erzielbaren Speedup als Minimum der Selbstparallelität des jeweiligen Codeabschnitts und der verfügbaren Anzahl an CPU-Kernen. Die abschließende Ableitung einer Parallelisierungsempfehlung und deren Speedup-Prädiktion erfolgt nun unter Berücksichtigung der zuvor approximierten Ausführungszeiten spezifisch für die durch die jeweils gewählte Parallelisierungsbibliothek wie beispielsweise *OpenMP* bereitgestellten Arten der Parallelisierung. Für eine detaillierte Beschreibung der dabei zur Optimierung des Speedups und zur Reduzierung der Anzahl der zu parallelisierenden Codeabschnitte eingesetzten Algorithmen sei auf [89] und [59] verwiesen.

Eine kommerzielle Software zur Unterstützung der Parallelisierung von in den Sprachen *C* und *C++* implementierten Programmen bietet *Vector Fabrics* mit *Pareon* [177] an, welche aus den Produkten *vfEmbedded* und *vfThreaded-x86* hervorgegangen ist. Zur Analyse des sequentiellen Codes wird durch *Pareon* sowohl eine statische als auch eine dynamische Analyse durchgeführt, wobei dem Anwender die Eignung der für die dynamische Analyse gewählten Eingabewerte für eine allgemeingültige Rekonstruktion des Programmverhaltens aufgezeigt wird. Dazu wird die unter den entsprechenden Eingabewerten erzielte Codeabdeckung als Anteil des in diesem Fall durchlaufenen Programmcodes berechnet. Als zentrales Merkmal der Software werden schließlich ausgewählte Darstellungen der Codestruktur generiert, die beispielsweise die Rechenanteile und die Verschachtelungen der Programmabschnitte ebenso visualisieren wie Abhängigkeiten innerhalb des Programmflusses. Ausgehend von dieser Visualisierung lassen sich nun Alternativen der Parallelisierung durch den Anwender auswählen, für die *Pareon* schließlich Speedup-Prädiktionen für *x86*- und *ARM*-Architekturen liefert, die nach Herstellerangaben auf Modellen der parallelen Zielplattform basieren. Statt den sequentiellen Code in automatisierter Weise entsprechend der gewählten Parallelisierungsalternative zu transformieren, werden dem Anwender abschließend lediglich die im Rahmen der *Code-Refaktorierung* erforderlichen Schritte als Handlungsempfehlung für eine manuelle Parallelisierung präsentiert.

Tool/Autoren	**Codeanalyse**	**Entwurfsraumexploration**	**Codeparallelisierung**	**Speedup-Prognose**	**Anwenderannotationen**
CETA [109]	dynamisch	-	-	-	-
SPRINT [43]	statisch	-	-	-	✓
Prospector [95]	statisch, dynamisch	✓	-	✓	-
Larsen et al. [103, 104]	statisch	-	-	-	✓
MAPS [40]	statisch, dynamisch	✓	✓	✓	-
Thies et al. [169]	dynamisch	-	✓	-	✓
Rul et al. [152]	statisch, dynamisch	✓	✓	✓	-
Tournavitis et al. [171]	dynamisch	✓	✓	-	-
DSWP [137]	statisch	✓	✓	-	-
Paralax [176]	statisch, dynamisch	✓	✓	✓	✓
MPA [12, 118]	statisch, dynamisch	-	✓	✓	✓
Cordes et al. [44, 45, 46]	statisch, dynamisch	✓	-	-	✓
Kremlin [60]	statisch, dynamisch	✓	-	✓	-
Pareon [177]	statisch, dynamisch	-	-	✓	-
Intel Advisor XE [84]	dynamisch	-	-	✓	-
EEEPA [30, 32, 35]	dynamisch	✓	-	✓	✓

Tabelle 5.1: Vergleichende Darstellung der in Abschnitt 5.2 vorgestellten verwandten Arbeiten unter Einbeziehung der in der vorliegenden Arbeit entwickelten Methode, welche in der *EEEPA*-Toolchain ihre prototypische Implementierung findet.

Eine dynamische Analyse bildet ebenfalls den Kern der Software *Intel Advisor XE* [84], welche die Parallelisierung von in *C*, *C++*, *C#* und *Fortran* implementierten sequentiellen Programmen unterstützt. In einem ersten Schritt werden dabei mittels eines dynamischen Profilings die Funktionen und Schleifen des Programms identifiziert, die signifikante Anteile der Rechenlast generieren und sich somit besonders für eine Parallelisierung empfehlen. Die dynamische Analyse basiert dabei auf dem *Pin*-Framework [110], welches mit Hilfe eines *Just-in-Time-Compilers* die Binärdateien zur Laufzeit instrumentiert und somit ähnlich wie das von Thies et al. genutzte *Valgrind* eine dynamische Binärinstrumentierung leistet. Die beim Profiling gewonnenen Informationen werden nun dem Anwender präsentiert und unterstützen diesen bei der Definition einer spezifischen Parallelisierung mittels manuell einzufügender Quellcode-Annotationen. Für

das damit spezifizierte parallele Design verspricht der Hersteller nun sowohl eine Prädiktion des Speedups und der Skalierbarkeit als auch eine Prüfung auf Korrektheit hinsichtlich typischer Aspekte bei der Nutzung gemeinsamer Daten durch parallele Threads. Eine abschließende automatische Parallelisierung des sequentiellen Codes findet zwar nicht statt, allerdings unterstützt das Tool die manuelle Überführung der zuvor gesetzten Annotationen in die Direktiven gängiger Parallelisierungsbibliotheken wie *OpenMP*, *Cilk* oder *Intel Threading Building Blocks*.

Eine vergleichende Übersicht der zuvor vorgestellten Arbeiten im Bereich der Software-Parallelisierung liefert Tabelle 5.1. Dabei ist zu beachten, dass Funktionalitäten auch dann als unterstützt gekennzeichnet sind, wenn diese gewissen Einschränkungen oder Anforderungen unterliegen. Dazu zählt beispielsweise die Fokussierung auf eine spezifische Art der Parallelität ebenso wie eine auf eine spezifische Laufzeitbibliothek oder Zielplattform angewiesene Parallelisierung. Für eine diesbezügliche Konkretisierung der jeweiligen Funktionalitäten sei auf die zuvor gelieferten detaillierten Beschreibungen der einzelnen Arbeiten verwiesen.

5.3 Die *EEEPA*-Toolchain

Zu der in Abschnitt 5.1 skizzierten Methode zur Exploration effizienter Parallelisierungsalternativen erfolgte in Form der sogenannten *EEEPA*[7]-Toolchain [30, 32, 35] für die Firma *Bosch Rexroth* eine prototypische Implementierung, die zum Teil im Rahmen einer Bachelorarbeit [72] geleistet wurde. Auch wenn Details der Implementierung in dieser Arbeit weitestgehend ausgeklammert werden sollen, liefern die nachfolgenden Abschnitte einen kurzen Überblick über die Architektur dieser Toolchain. Deren Entwicklung hatte als ein wesentliches Ziel die Nutzung existierender Applikationen, Dateiformate und Schnittstellen, sofern sich diese als geeignet erwiesen. Eine wesentliche Motivation der Implementierung ist ein Eignungsnachweis für die in dieser Arbeit entwickelte Methode und deren Evaluation im Rahmen der in Kapitel 6 beschriebenen Fallstudie. Dabei konnten wertvolle Ergebnisse für die Weiterentwicklung der CNC-Plattform *Rexroth IndraMotion MTX* gewonnen werden, so dass dort auch zukünftig ein Einsatz der *EEEPA*-Toolchain im Rahmen der Entwicklungsplanung angestrebt wird.

5.3.1 Architektur

Wie in Abbildung 5.4 dargestellt, besteht die *EEEPA*-Toolchain aus unabhängig voneinander ausführbaren Kommandozeilenprogrammen, deren Gesamtheit die in Abbildung 5.1 skizzierte Methode zur Exploration und Evaluation effizienter Parallelisierungsalternativen realisiert. Die Implementierung dieser Toolchain basiert auf der Programmiersprache *C#*, so dass die resultierenden Applikationen innerhalb einer *.NET*-Laufzeitumgebung ausführbar sind.

Arbeitsgrundlage der Toolchain sind mittels des Software-Profilers *Wind River System Viewer* aufgezeichnete Laufzeitdaten des zu modellierenden Systems, die jeweils in eine Log-Datei exportiert werden. Aus diesen werden nun durch den sogenannten *Log-Parser EEEPA.LP* partielle Firmware-Modelle in Form von Graphen extrahiert, bevor diese durch den *Graph-Enhancer EEEPA.GE* um Expertenwissen und Benchmark-Resultate erweitert und dabei zu finalen Modellen ausgeprägt werden. Auf deren Basis wird schließlich mittels genetischer Algorithmen der Entwurfsraum potentieller Parallelisierungen exploriert und evaluiert, um Pareto-optimale

[7] ***E**xploration and **E**valuation of **E**fficient **P**arallelization **A**lternatives*

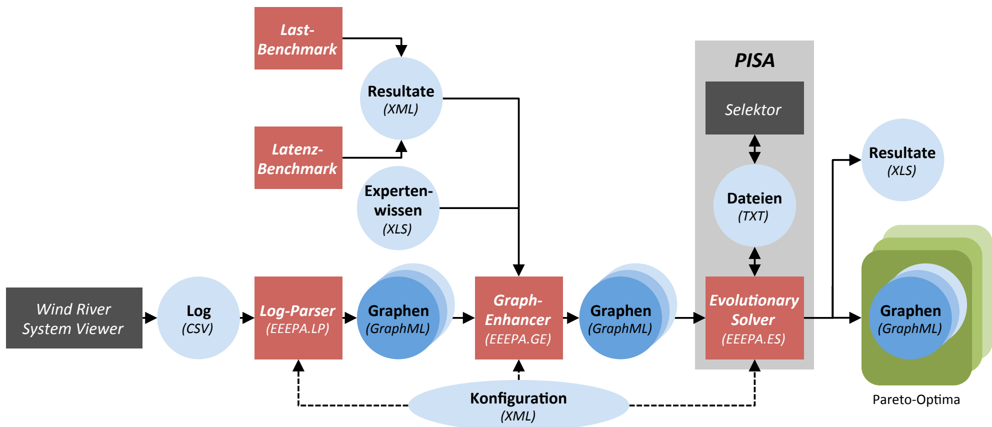


Abbildung 5.4: Übersicht über die *EEEPA*-Toolchain

Alternativen für eine Multicore-CPU mit einer vorgegebenen Anzahl an CPU-Kernen abzuleiten. Dabei muss nur der problemspezifische Teil des genetischen Algorithmus durch den *Evolutionary Solver EEEPA.ES* implementiert werden, da mittels einer Konformität mit der in Abschnitt 5.3.4 beschriebenen Schnittstellenspezifikation *PISA* auf bereits existierende Implementierungen von Selektoren zurückgegriffen werden kann. Abschließend werden die explorierten Pareto-optimalen Alternativen der Parallelisierung als parametrierte Graphen (vgl. Definition 12) ausgegeben. Eine tabellarische Darstellung fasst zusätzlich die Eigenschaften dieser Resultate in quantitativer Form zusammen.

5.3.2 Profiler-Integration

Wenngleich die Implementierung der *EEEPA*-Toolchain möglichst generisch ausgelegt ist, ist beim Profiling der Laufzeitinformationen eine Anpassung an das konkrete Betriebssystem der zu modellierenden Firmware unvermeidlich. Dies ist im Fall der CNC-Plattform *Rexroth IndraMotion MTX* das Echtzeitbetriebssystem *Wind River VxWorks 6.7* [186]. Aus diesem Grund wird zum Profiling auch das Programm *Wind River System Viewer 3.2* eingesetzt, das der Hersteller als "Logic Analyzer für Embedded-Software“ bezeichnet [188]. Dieses Tool ist Bestandteil der *Wind River Workbench*, einer auf dem *Eclipse*-Framework basierenden Umgebung zur Firmware-Entwicklung für die Betriebssysteme *VxWorks* und *Wind River Linux* [189].

Die Architektur des *System Viewers* stellt sich wie folgt dar: Auf der *VxWorks*-Plattform werden während der Ausführung der Firmware durch eine spezielle Logging-Task zunächst alle entsprechend der Konfiguration aufzuzeichnenden Laufzeitdaten in einen als Puffer genutzten Speicherbereich geschrieben. Voraussetzung dafür ist die Instrumentierung entsprechender Systembestandteile wie beispielsweise des Betriebssystem-Kernels oder der Applikation selbst, so dass diese die geforderten Informationen an die Logging-Task weiterleiten. Besonders wichtig ist, dass bei der Aufzeichnung eines Ereignisses auch dessen Zeitstempel gesichert wird. Im Anschluss an die Aufzeichnung erfolgt durch eine weitere Task der Upload der gesammelten Da-

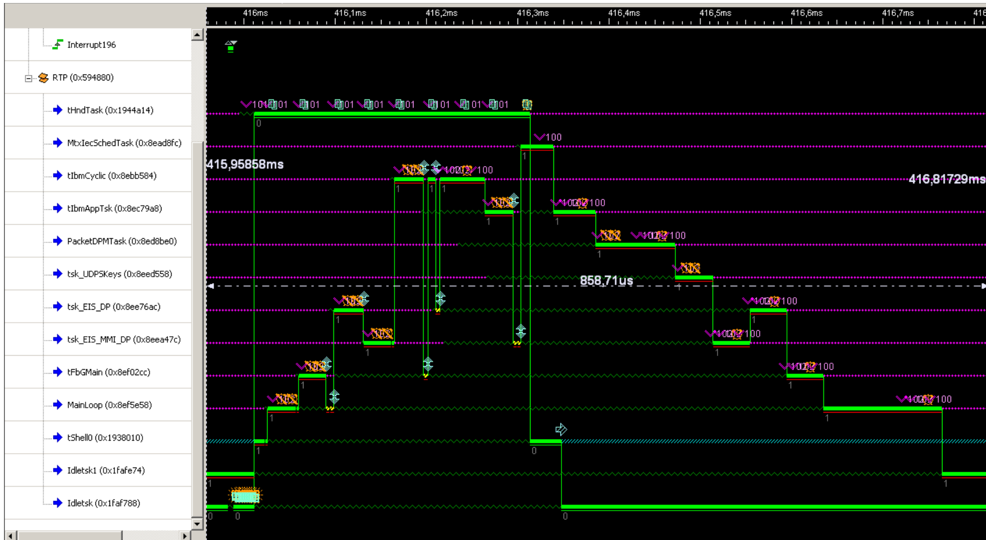


Abbildung 5.5: Visualisierung eines durch den *System Viewer* auf einer Dual-Core-CPU aufgezeichneten Firmware-Logs als *Event Graph* mit parallelen Kontrollflüssen. Task-Zustände werden als Linien visualisiert, Ereignisse mittels entsprechender Symbole. Die Legende zeigt die jeweiligen Tasks in absteigender Prioritätenordnung.

ten zu einem Host-Rechner, der mit dem Target über ein Netzwerk verbunden ist. Der Upload kann dabei entweder kontinuierlich während des Loggings stattfinden *(Continuous Upload)* oder im Anschluss daran *(Deferred Upload)*, wenn der zur Verfügung stehende Puffer voll ist. Prinzipbedingt ist dabei der Laufzeit-Overhead des kontinuierlichen Uploads signifikant höher, so dass im Rahmen der *EEEPA*-Implementierung von dieser Option kein Gebrauch gemacht wird. Die Aufbereitung, Visualisierung und Analyse der gesammelten Daten findet wiederum auf dem Host statt, wodurch nach Angaben von *Wind River* der Overhead des Loggings auf dem Target reduziert wird. Allerdings kann ein entsprechend detailliertes Profiling durchaus eine Aufzeichnungsdatenrate von 10 MB/s und mehr verursachen, was natürlich selbst bei einem nachgelagerten Upload nicht ohne Seiteneffekte auf das Systemverhalten bleibt.

Die aufgezeichneten Daten lassen sich auf verschiedene Arten analysieren; Abbildung 5.5 zeigt eine Darstellung als *Event Graph*. Für die Modellierung durch die *EEEPA*-Toolchain werden allerdings die aufgezeichneten Rohdaten benötigt, die sich als kommaseparierte Textdatei (*CSV*-Datei) exportieren lassen. Mit Ausnahme der *CSV*-basierten Schnittstelle des *Log-Parser* sind dann alle weiteren Schritte der Toolchain unabhängig vom Profiling durch den *System Viewer*. Somit kann die Toolchain durch eine Adaption des beim Import der *CSV*-Dateien eingesetzten Parsers relativ einfach an ein anderes Profiling-Tool angepasst werden, sofern dieses in der Lage ist, die nachfolgend definierten Informationen zur Laufzeit aufzuzeichnen.

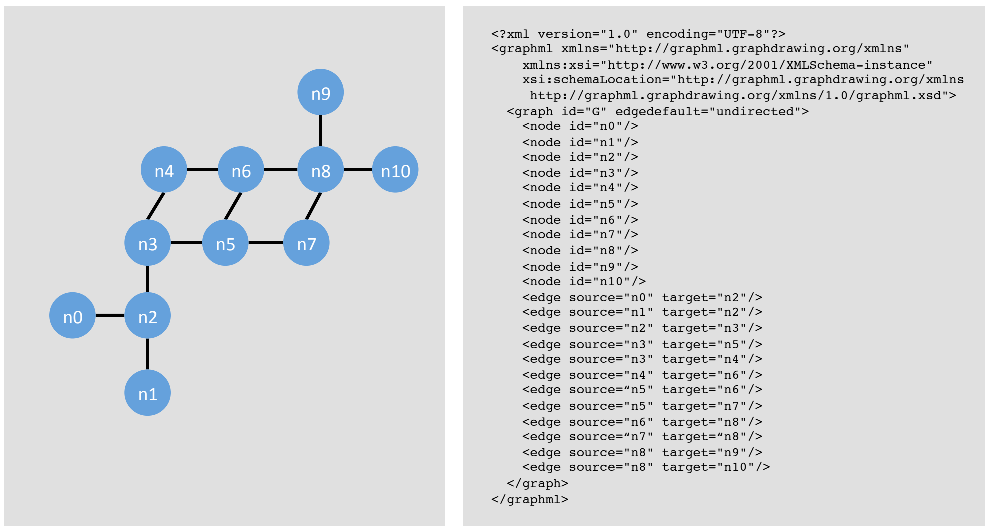


Abbildung 5.6: *GraphML*-Darstellung eines einfachen Graphen nach [28]

5.3.3 *GraphML*-Erweiterung

Die zuvor beschriebene *EEEPA*-Architektur erfordert nicht nur einen Austausch der Firmware-Modelle zwischen den Modulen der Toolchain, sondern auch deren effiziente Auswertung, Verarbeitung und Archivierung. Entsprechend Abschnitt 2.6 werden diese Anforderungen insbesondere von Graphen erfüllt. Einfache Arten der persistenten Speicherung von Graphen sind die Adjazenzmatrix, die Inzidenzmatrix oder die Adjazenzlistenrepräsentation [101], die jedoch diverse Nachteile mit sich bringen. So ist die Interpretation eines in dieser Weise gespeicherten Graphen nur durch einen Computer, nicht aber durch einen Menschen in effizienter Weise möglich. Aus diesem Grund wurde zur Speicherung von Firmware-Modellen im Rahmen der *EEEPA*-Implementierung das auf *XML* basierende Format *GraphML* [66] gewählt, dessen Spezifikation an der *Universität Konstanz* entwickelt wurde [27]. *GraphML* kann zudem durch zahlreiche gängige Programme zur Graphenvisualisierung und -analyse interpretiert werden, wie beispielsweise *yEd* [191] oder *Gephi* [18]. Dies stellt in Anbetracht der Tatsache, dass die qualitativ hochwertige Visualisierung von Graphen einen eigenen Forschungsbereich begründet [76] und somit nicht im Fokus dieser Arbeit liegt, einen großen Vorteil dar. Wie in Abbildung 5.6 gezeigt, definiert *GraphML* drei Elemente, um Graphen zu repräsentieren: `graph`, `node` und `edge`. Verpflichtend ist dabei die Definition einer ID pro Knoten, mittels derer die zu einer Kante inzidenten Knoten referenziert werden.

Ein weiterer Grund für die Wahl von *GraphML* ist die Möglichkeit zur Speicherung applikationsspezifischer Informationen im Graphen. Beliebige Attribute mit einfachen Datentypen wie `boolean`, `int`, `long`, `float`, `double` oder `string` lassen sich dabei in Form sogenannter *GraphML*-Attribute für alle zuvor genannten *GraphML*-Elemente deklarieren und in Form des *XML*-Elements `data` definieren. Dies zeigt das nachfolgende, aus [28] entnommene Beispiel,

welches für das Element `edge` zunächst ein Attribut `weight` vom Typ `double` deklariert und für dieses schließlich den Wert `1.0` definiert:

```
<key id="d1" for="edge" attr.name="weight" attr.type="double"/>
...
<edge id="e0" source="n0" target="n2">
  <data key="d1">1.0</data>
</edge>
```

Da ein Graph ein umfassendes Modell der Laufzeitcharakteristik eines Systems abbilden soll, besteht darüber hinaus die Anforderung, dessen Elemente um strukturierte Daten unterschiedlichen Typs zu erweitern. Zur Speicherung strukturierter Elementattribute definiert das *GraphML*-Schema den komplexen Datentyp `data-extension.type`. Dieser wird durch den Typ `data.type` erweitert, der wiederum den Basistyp jedes `data`-Elements des Graphen darstellt. Als Konsequenz einer Erweiterung von `data-extension.type` in einem separaten *XML*-Schema mittels Redefinition lassen sich somit auch *GraphML*-Attribute mit komplexem Datentyp deklarieren und definieren. Im Zuge der *EEEPA*-Implementierung wurde von dieser Möglichkeit Gebrauch gemacht [31], indem ein *XML*-Schema `eeepa-graphml.xsd` definiert wurde, welches diese Erweiterungen vornimmt. Alle Graphen der *EEEPA*-Toolchain referenzieren nun dieses Schema mittels `xsi:schemaLocation`. Da die strukturelle Definition des Graphen durch diese Erweiterung nicht beeinflusst wird, kann ein *EEEPA-GraphML*-Graph trotzdem durch jedes Programm interpretiert werden, das *GraphML*-Dateien verarbeiten kann. Die entsprechenden für *EEEPA* spezifischen Erweiterungen werden dann ignoriert.

5.3.4 *PISA*-Konformität

Eine detaillierte Analyse genetischer Algorithmen zeigt, dass diese in der Regel aus einem für das jeweilige Optimierungsproblem spezifischen Teil und einem von diesem Problem unabhängigen Teil bestehen. Selbst im Fall standardisierter Genotyp-Repräsentationen und Operatoren muss zumindest die Ableitung und Bewertung der Phänotypen an den jeweiligen Anwendungsfall angepasst werden. Selektionsstrategien hingegen sind in der Regel unabhängig vom Optimierungsproblem und berücksichtigen ausschließlich die Fitnessbewertungen der Individuen. Somit sollte sich die in dieser Arbeit entwickelte Methode auf die Spezifikation und Implementierung der für das Problem der Firmware-Parallelisierung spezifischen Aspekte des genetischen Algorithmus fokussieren.

Dieser Anforderung wird die unter dem Namen *PISA* an der *ETH Zürich* entwickelte Schnittstellenspezifikation für evolutionäre Algorithmen gerecht [21], die Abbildung 5.7 skizziert. *PISA* spezifiziert dabei lediglich eine auf Textdateien basierende Schnittstelle, mittels derer zwei unabhängige Anwendungen, der problemabhängige *Variator* und der problemunabhängige *Selektor*, kommunizieren. Wie in Abbildung 5.7 dargestellt, kann auf diese Weise ein selbst entwickelter Problemlösungsalgorithmus auf unterschiedliche Probleme, wie beispielsweise ein Netzwerkdesign oder das *Knapsack*-Problem angewendet werden. Alternativ dazu können für ein vorhandenes Problem wie die multikriteriell zu optimierende Task-Partitionierung und -Dekomposition *(EEEPA)* eine Vielzahl verschiedener Problemlöser genutzt werden. Aufgrund der Kommunikation mittels Dateien ist die Schnittstelle von der Plattform, der Programmiersprache und dem Betriebssystem der beiden Anwendungen unabhängig, so dass diese diesbezüglich divergieren

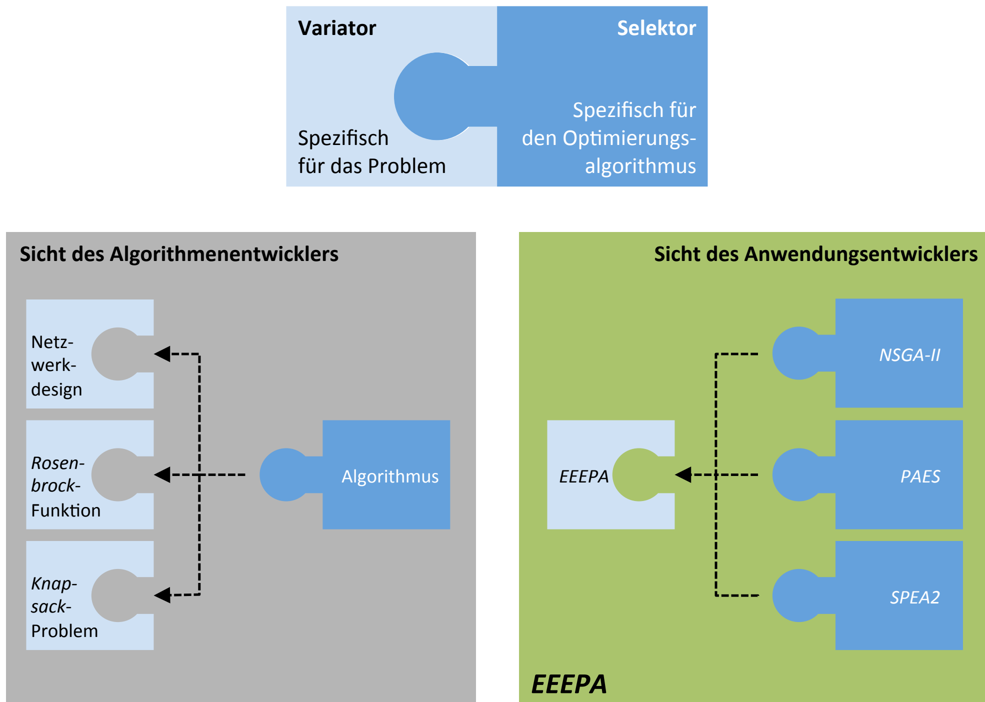


Abbildung 5.7: Das *PISA*-Konzept nach [21]. Das *EEEPA*-Tool hat eine anwendungsorientierte Sicht auf die Problemstellung der evolutionären Optimierung.

können. Zudem können von der *PISA*-Homepage [83] ausführbare Variatoren und Selektoren geladen und mit dem jeweils eigenen Programm kombiniert werden.

Die Kommunikation zwischen diesen Modulen auf Basis von Dateien stellt Abbildung 5.8 dar. Um dabei das Datenvolumen gering zu halten, werden Individuen ausschließlich über einen ganzzahligen Index referenziert. Dies ist möglich, da der Selektor unabhängig von der problemspezifischen Individuenrepräsentation arbeitet. Der Variator schreibt jeweils nach dem Generieren der Initialpopulation und der Reproduktion die mit einem Index versehenen multikriteriellen Fitnessbewertungen der Individuen in eine Textdatei. Die in der Abbildung dargestellte Datei `ini` wird dabei für die Initialpopulation und die Datei `var` für die Nachkommen einer Reproduktion genutzt. Diese Dateien werden entweder einmalig (`ini`) oder zyklisch (`var`) durch den Selektor ausgelesen und auf Basis dieser Werte eine Selektion durchgeführt. Im Anschluss daran wird mittels zweier weiterer Dateien das Ergebnis der Selektion an den Variator übermittelt: Die Datei `sel` speichert die Indizes der Individuen, die als Nachkommen für die nächste Generation selektiert wurden, während die Datei `arc` optional alle Individuen referenziert, die in ein generationenübergreifend mitgeführtes Archiv übernommen werden. Um sicherstellen zu können, dass die jeweiligen Dateien vollständig gelesen und geschrieben

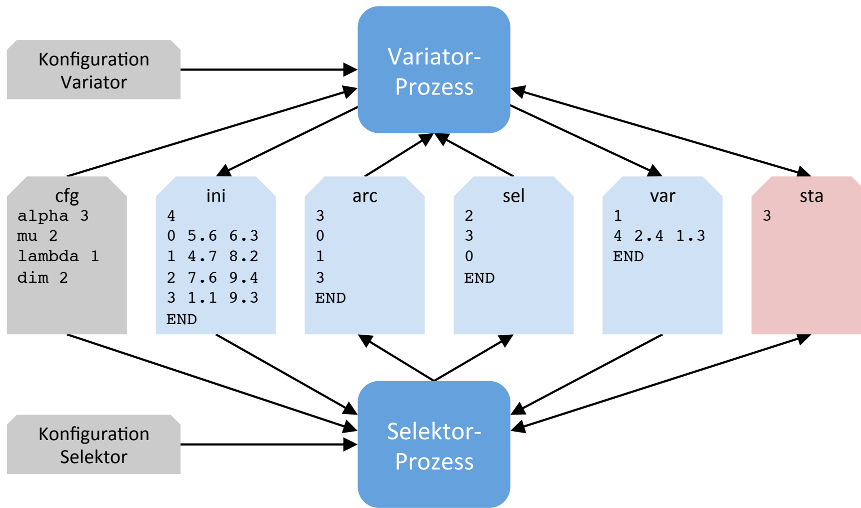


Abbildung 5.8: Darstellung der Kommunikation auf Dateibasis in *PISA* nach [21]

werden, ist in *PISA* eine Präambel mit der Anzahl der nachfolgenden Werte ebenso spezifiziert wie ein `END`-Tag, welches das Ende der Datei markiert.

Sowohl der Variator als auch der Selektor werden zyklisch ausgeführt. Zur Synchronisation der Aktivitäten ist eine weitere Textdatei `sta` definiert, die den jeweils aktuellen Zustand des Algorithmus speichert. Diese Datei wird von den beteiligten Prozessen zyklisch abgefragt und bei Vorliegen des entsprechenden Status mit den zugeordneten Aktionen begonnen, wie beispielsweise dem Einlesen der Indizes der Nachkommen und der Reproduktion. Weiterhin ist eine Konfigurationsdatei `cfg` definiert, welche die Parameter des Algorithmus speichert, die sowohl für den Variator als auch für den Selektor relevant sind. Dazu gehört der Umfang der Initialpopulation (α), der Eltern (μ) und der Nachkommen (λ) ebenso wie die Dimension `dim` der multikriteriellen Optimierung. Die modulspezifischen Konfigurationen werden hingegen entweder in einer weiteren separaten Datei oder im Fall des *EEEPA*-Variators in der allgemeinen Konfigurationsdatei der Toolchain definiert.

Im Rahmen der *EEEPA*-Toolchain wurde der problemabhängige *EEEPA*-Variator als *C#*-Applikation unter Berücksichtigung der Schnittstellenspezifikation *PISA* entwickelt, eine detaillierte Beschreibung liefert Abschnitt 5.6. Auf Seiten des Selektors lassen sich hingegen die Implementierungen verbreiteter Selektoren als kompilierte Binärdateien von der Webseite des *PISA*-Projekts laden und mit dem Variator kombinieren.

5.4 Modellierung auf Systemebene

Entsprechend Abschnitt 5.1.1 besteht das Ziel einer Modellierung auf Systemebene in der Exploration und Evaluation verschiedener Alternativen eines partitionierten Schedulings eines Multitasking-Systems auf einer Multicore-CPU.

Definition 13 (Task-Verteilung) *Als Task-Verteilung oder Task-Mapping wird eine Abbildung $f_m: T \rightarrow C$ einer Menge von Tasks[8] T auf eine Menge von CPU-Kernen C bezeichnet.*

Eine Task-Verteilung definiert somit ausschließlich die räumliche Dimension des Task-Schedulings, während die temporale Ausführungsplanung nach wie vor Aufgabe des Betriebssystem-Schedulers zur Laufzeit des Systems ist.

5.4.1 Bewertungskriterien

Die algorithmische Exploration geeigneter Task-Verteilungen erfordert die Möglichkeit zur Evaluation potentieller Lösungen unter definierten Kriterien. Für eine algorithmische Evaluation ist für jedes Kriterium dessen Quantifizierung in Form einer geeigneten Metrik nötig.

Lastverteilung

Sei $C^T := \{f_m | f_m: T \rightarrow C\}$ die Menge aller Mappings f_m einer Menge von Tasks T auf eine Menge von CPU-Kernen C. Die Grundlage aller nachfolgend definierten Metriken bildet stets die durchschnittliche Auslastung $f_{l_c}: C^T \times C \rightarrow [0,1]$ eines CPU-Kerns $c \in C$ unter einem Mapping $f_m \in C^T$ innerhalb eines spezifischen Zeitintervalls. Diese berechne sich auf Basis der mittleren relativen Lastanteile $f_{l_t}: T \rightarrow [0,1]$ der Tasks T innerhalb dieses Zeitintervalls:

$$f_{l_c}\left(f_m, c\right) := \sum_{t \in T_c} f_{l_t}\left(t\right) \quad \text{mit} \quad T_c := \left\{t \in T \middle| f_m\left(t\right) = c\right\} \tag{5.1}$$

Entsprechend Abschnitt 3.1.1 ist ein wesentliches Zielkriterium der Lastverteilung in Multicore-Architekturen eine gleichmäßige mittlere Auslastung der CPU-Kerne, um auf diese Weise die Idle-Zeiten des Systems äquivalent auf den Kernen zu verteilen und Wartezeiten der Tasks zu reduzieren. Insbesondere in der Automatisierungstechnik wird auf diese Weise die Performanz der Steuerung ebenso erhöht wie deren Robustheit gegenüber sporadischen Lastspitzen.

Die deskriptive Statistik definiert eine Reihe von Metriken zur Bewertung der Streuung skalarer Größen [10], die somit auch für die Quantifizierung einer Lastverteilung auf Basis durchschnittlicher Core-Auslastungen $f_{l_c}(f_m, c)$ geeignet sind. Dies gilt insbesondere für die Metrik der *Spannweite (Range)* $f_r: C^T \rightarrow [0,1]$, die als die Lastdifferenz zwischen dem am stärksten und dem am geringsten ausgelasteten CPU-Kern unter einem Mapping $f_m \in C^T$ definiert sei:

$$f_r\left(f_m\right) := \max_{c \in C}\left(f_{l_c}\left(f_m, c\right)\right) - \min_{c \in C}\left(f_{l_c}\left(f_m, c\right)\right) \tag{5.2}$$

Somit bewertet die Spannweite das Ungleichgewicht der Lastverteilung, so dass die Auslastung der CPU-Kerne umso gleichmäßiger ist, je kleiner der durch die Metrik berechnete Wert ist. Da die Spannweite nur die Extrema der Auslastung berücksichtigt, ist sie in effizienter Weise berechenbar. Die damit einhergehende Vernachlässigung der zwischen den Extrema liegenden Core-Auslastungen ist dabei insofern unproblematisch, als eine auf die Minimierung der Spannweite

[8] Sofern keine explizite Differenzierung erfolgt, bezieht sich die Verwendung des Begriffs *Task* im Folgenden sowohl auf Tasks als auch auf Interrupts. Auf eine Differenzierung wird hier verzichtet, da eine solche für die Methode nicht erforderlich ist und dadurch die Lesbarkeit der weiteren Ausführungen deutlich verbessert wird.

abzielende Optimierung des Mappings zugleich das Gleichgewicht der Core-Auslastungen maximiert. Der Grund hierfür ist die Tatsache, dass eine minimale Spannweite genau dann erzielt wird, wenn die Auslastung aller CPU-Kerne identisch ist.

Das der Spannweite zugrunde liegende Zielkriterium in Form einer gleichmäßigen mittleren Auslastung der CPU-Kerne kann jedoch durchaus kritisch bewertet werden, da die absolute Auslastung des Systems unberücksichtigt bleibt. So kann aber bei einem gering ausgelasteten System ein deutlich größeres Lastungleichgewicht tolerierbar sein als bei einem stark ausgelasteten System, bei dem bereits ein relativ geringes Ungleichgewicht der Core-Auslastungen bei sporadischen Lastspitzen kritisch werden kann. Somit sei als alternative Metrik der Lastverteilung die *Maximallast* $f_x\colon C^T \to [0,1]$ definiert, die dem mittleren Lastanteil entspricht, den in einem Mapping $f_m \in C^T$ der CPU-Kern mit der höchsten Auslastung aufweist:

$$f_x\left(f_m\right) \coloneqq \max_{c \in C}\left(f_{l_c}\left(f_m, c\right)\right) \tag{5.3}$$

Die Metrik der Maximallast bewertet somit ein Task-Mapping ausschließlich hinsichtlich der Last des Kerns mit der geringsten verbleibenden Idle-Zeit und lässt demgegenüber die Auslastung der übrigen CPU-Kerne unberücksichtigt. Eine auf diese Metrik abzielende Optimierung des Task-Mappings führt allerdings ebenfalls zu einer möglichst gleichmäßigen Core-Auslastung, da die maximale Auslastung eines CPU-Kerns genau dann am geringsten ist, wenn alle Kerne identisch ausgelastet sind.

Zuvor wurde gezeigt, dass sowohl die Spannweite als auch die Maximallast bei einer diesbezüglichen Optimierung des Task-Mappings das Ungleichgewicht der Core-Auslastungen reduzieren. Der wesentliche Unterschied der Metriken wird deutlich, wenn zur Bewertung eines Mappings unter mehreren Lastprofilen deren Aggregation (vgl. Abschnitt 5.1.1) erforderlich wird und die einzelnen Lastprofile in ihrer Gesamtauslastung divergieren. Die Konstruktion eines entsprechenden Beispiels an dieser Stelle würde allerdings den nachfolgenden Abschnitten zu weit vorweggreifen, so dass an dieser Stelle für eine detaillierte Erläuterung auf den Abschnitt 6.2.2 der Beschreibung der Fallstudie verwiesen sei.

Inter-Core-Interaktionen

Die unilaterale und multilaterale Task-Synchronisation sowie die Kommunikation zwischen Tasks sind ein wesentliches Merkmal eines Multitasking-Systems (vgl. Abschnitt 2.5.2). Generell lassen sich Task-Interaktionen hinsichtlich mehrerer Aspekte klassifizieren [65], wobei im Folgenden eine Beschränkung auf Interaktionen mit einer Produzenten-Konsumenten-Semantik erfolgen soll. Diese zeichnen sich dadurch aus, dass eine Datenstruktur von einer Task explizit angefordert und von einer anderen Task mittels eines entsprechenden Übertragungskanals explizit zur Verfügung gestellt wird. Unsynchronisierte Lese- und Schreibzugriffe auf gemeinsam genutzte Daten werden somit nicht berücksichtigt. Werden diese Tasks nun auf unterschiedliche CPU-Kerne verteilt, muss die entsprechende Interaktion als sogenannte *Inter-Core-Interaktion* über Kerngrenzen hinweg abgewickelt werden. Dies erzeugt jedoch zusätzlichen Overhead und reduziert so die Performanz des Systems.

Wie in Abschnitt 2.2 dargestellt, ist in hierarchischen Multicore-Architekturen eine Kommunikation über die Grenzen von CPU-Kernen hinweg gegenüber einem Datenaustausch innerhalb eines Kerns tendenziell mit einem größeren Overhead verbunden, der sich sowohl auf die

Kommunikationslatenz als auch auf die Systemlast auswirkt. Dies ist irrelevant, wenn zwischen dem Schreiben und Lesen der Daten eine Verdrängung derselben in einen durch die an der Kommunikation beteiligten Cores gemeinsam genutzten Cache höherer Ebene erfolgt. Dann müssen die Daten beim Lesen unabhängig davon, ob die Empfänger-Task auf dem gleichen Kern ausgeführt wird wie die Sender-Task oder nicht, ohnehin zunächst wieder in die Core-lokalen Caches kopiert werden. Ob es zwischen dem Schreiben und Lesen der Daten allerdings zu einer Verdrängung kommt, ist von zahlreichen Faktoren des Laufzeitverhaltens des Systems abhängig, wie beispielsweise der dazwischen verstrichenen Zeitdauer, der generellen Auslastung des Systems [19] oder sporadischen *Cache-Flushes*.

Darüber hinaus sind *Inter-Processor Interrupts (IPIs)* ein gängiges Mittel der Synchronisation über Kerngrenzen hinweg (vgl. Abschnitt 2.4.2). Vor allem die Interaktion zwischen Tasks auf verschiedenen CPU-Kernen mittels Betriebssystemmechanismen ist dabei eine häufige Quelle von IPIs, die jedoch nicht nur die Latenz der Kommunikation, sondern auch die Systemlast erhöhen. In welchem Umfang bei Task-Interaktionen IPIs generiert werden, ist jedoch vom Laufzeitverhalten des Systems abhängig. So entscheidet sich erst in Abhängigkeit von dem jeweiligen Zustand der Empfänger-Task einer spezifischen Inter-Core-Interaktion, ob im Kontext dieser Interaktion ein per IPI veranlasstes Rescheduling stattfinden muss.

Zusammenfassend kann aus den zuvor genannten Gründen unterstellt werden, dass in einem System der aufgrund von Inter-Core-Interaktionen generierte Overhead und die Anzahl der Inter-Core-Interaktionen positiv korreliert sind. Eine allgemeingültige Aussage über den Korrelationskoeffizienten ist dabei allerdings nicht möglich, da der Umfang, mit dem die zuvor beschriebenen Arten des Overheads entstehen, sowohl spezifisch für ein konkretes System als auch für eine konkrete Ausführung desselben ist. Als Konsequenz würde selbst eine Auswertung empirisch erhobener Daten keine generische Aussage über diesen Korrelationskoeffizienten erlauben. Um jedoch den Overhead einer Task-Verteilung so gering wie möglich zu halten, sollte dennoch aufgrund der zuvor als positiv unterstellten Korrelation die Anzahl an Inter-Core-Interaktionen so weit wie möglich reduziert werden, indem intensiv interagierende Tasks dem gleichen CPU-Kern zugeordnet werden [65].

Implementierungsaufwand

Als Aufgabenstellung dieser Arbeit wurde definiert, effiziente Firmware-Designs zur Nutzung von Multicore-Architekturen in der Steuerungstechnik zu explorieren. Dies impliziert zugleich, der Leistungsbewertung einer Lösung auch die Kosten gegenüberzustellen, die im Rahmen der Realisierung anfallen. Da Personalkosten den größten Anteil der Kosten von Software ausmachen, gilt es, diese in Form von Aufwänden abzuschätzen, die zur Implementierung einer Lösung zu erbringen sind. Derartige Aufwandsabschätzungen stellen bei der Software-Entwicklung ein gängiges Mittel der Entwicklungsplanung dar [16]. Für eine Parallelisierung auf Systemebene bietet ein Multitasking-System zwar die idealen Voraussetzungen, allerdings sind zur Sicherstellung der echt parallelen Ausführbarkeit der Tasks unter Umständen noch signifikante Implementierungsaufwände zu leisten. Deren Umfang stellt das dritte Evaluationskriterium der hier entwickelten Methode dar.

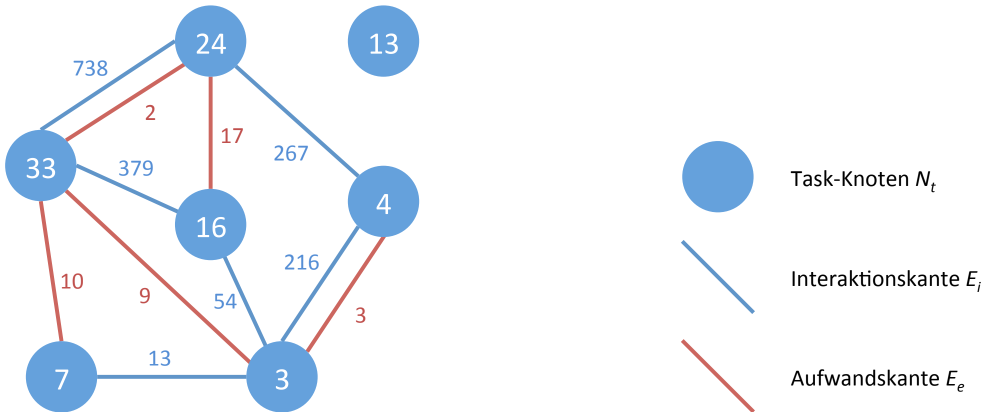


Abbildung 5.9: Visualisierung eines Task-Graphen

5.4.2 Basismodell

Um alternative Task-Mappings hinsichtlich der zuvor definierten Kriterien zu bewerten, wird für die Modellierung auf Systemebene der sogenannte *Task-Graph* $G = (N_t, E_i, E_e)$ vorgeschlagen; ein Beispiel zeigt Abbildung 5.9. Ein Task-Graph definiert die folgenden Elemente:

- Gewichtete *Task-Knoten* N_t repräsentieren die Tasks T des Systems. Das Gewicht eines Task-Knotens entspricht dabei dem relativen Lastanteil $f_{l_t}(t)$, den die Ausführung der Task t im Zeitraum des Loggings in Anspruch genommen hat. Dieser Lastanteil bezieht sich auf die durch einen einzelnen CPU-Kern zur Verfügung gestellte Rechenleistung.
- Ungerichtete, gewichtete *Interaktionskanten* E_i repräsentieren den Umfang der Task-Interaktionen zwischen den jeweils inzidenten Tasks innerhalb des Logging-Intervalls. Da die Richtung der Interaktionen für die Bewertung eines Mappings hinsichtlich der Inter-Core-Interaktionen irrelevant ist, handelt es sich hierbei um ungerichtete Kanten.
- Ungerichtete, gewichtete *Aufwandskanten* E_e repräsentieren Abschätzungen der Entwicklungsaufwände, die erforderlich sind, um die jeweils inzidenten Tasks echt parallel auf einer Multicore-CPU auszuführen. Ein gängiges Maß der Aufwandsabschätzung sind beispielsweise die bei der Implementierung durch einen Entwickler zu leistenden Personenstunden.

Die in Abschnitt 5.1.2 generisch definierten Schritte der Modellausprägung und -parametrierung stellt Abbildung 5.10 konkret für den Task-Graphen dar. Im Rahmen der Ausprägung werden dabei basierend auf den Laufzeitdaten eines Systems und entsprechendem Expertenwissen die Instanzen gewichteter Task-Knoten, Interaktionskanten und Aufwandskanten abgeleitet. Eine Parametrierung des Task-Graphen besteht schließlich darin, mittels eines Mappings $f_m\colon T \to C$ jeder durch einen Knoten repräsentierten Task $t \in T$ eine statische *Affinität* zu einem CPU-Kern $c \in C$ zuzuweisen.

Nun gilt es, die Bewertung eines ausgeprägten und parametrierten Task-Graphen unter den in Abschnitt 5.4.1 definierten Kriterien zu spezifizieren. Auf Basis des Mappings f_m lässt sich für je-

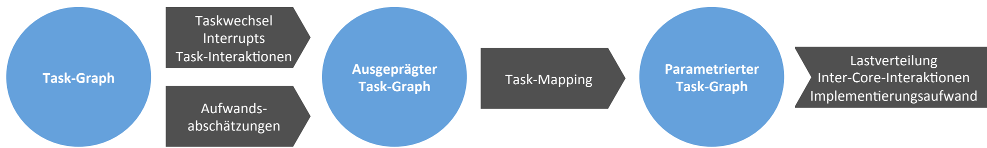


Abbildung 5.10: Prozess der Modellierung und Evaluation auf Systemebene

den CPU-Kern $c \in C$ die in Abschnitt 5.4.1 definierte Funktion f_{l_c} der Core-Auslastung unter f_m berechnen. Diese wiederum stellt die Grundlage für die Berechnung der in Abschnitt 5.4.1 definierten Metriken zur Bewertung der Lastverteilung dar. Die Bewertung eines Mappings f_m hinsichtlich der Inter-Core-Interaktionen und Implementierungsaufwände ermittelt sich jeweils in Abhängigkeit von den Gewichten der Interaktions- respektive Aufwandskanten, deren inzidente Task-Knoten $n_{t_1}, n_{t_2} \in N_t$ unter f_m unterschiedlichen CPU-Kernen zugewiesen sind. Eine detaillierte Beschreibung der dabei angewandten Logik würde allerdings zu weit vorausgreifen, so dass an dieser Stelle auf Abschnitt 5.4.4 verwiesen sei.

5.4.3 Systeminstrumentierung

Bei der Ausprägung von Task-Graphen sieht die im Rahmen dieser Arbeit entwickelte Methode die Instanziierung von Task-Knoten und Interaktionskanten auf Basis von Laufzeitdaten des jeweiligen Systems vor. Zur Erfassung der benötigten Daten beim Profiling ist jedoch zunächst eine geeignete Instrumentierung des Systems erforderlich, für deren Umfang es einen Kompromiss zu finden gilt. So besteht der Vorteil einer umfassenden Instrumentierung darin, dass sich Modelle mit einer höheren Modellierungstiefe generieren lassen, deren Prädiktionen sich infolgedessen durch eine potentiell höhere Güte auszeichnen. Zugleich beeinflusst jedoch der einem Profiling inhärente Overhead das Laufzeitverhalten des Systems und somit wiederum die Validität der Modelle in negativer Weise.

Der Aufwand einer Instrumentierung ist vergleichsweise gering, wenn sich diese auf das der Firmware zugrunde liegende Betriebssystem beschränkt, indem dort zentrale Funktionen in der für das Profiling erforderlichen Weise erweitert werden. Signifikant aufwändiger gestaltet sich dieses Vorgehen, wenn auch der meist umfangreiche Firmware-Code instrumentiert werden muss, da die Identifikation und Adaption entsprechender Stellen sehr zeitaufwändig ist. Die im Rahmen dieser Arbeit genutzte Instrumentierung zur Extraktion von Task-Graphen beschränkt sich deshalb ausschließlich auf Anpassungen des Betriebssystemcodes. Dies wiederum hat zur Konsequenz, dass der entsprechende Quellcode entweder vom Hersteller offengelegt sein muss oder die Anpassungen durch den Hersteller durchgeführt werden müssen.

In Abschnitt 2.4.1 wurden für gängige Betriebssysteme drei Task-Zustände beschrieben, von denen ausschließlich der Zustand *Running* die Ausführung einer Task auf einer CPU oder einem CPU-Kern definiert. Von den entsprechenden Zustandsübergängen sind somit für die Modellierung nur die in Tabelle 5.2 genannten relevant, da diese die Fortsetzung respektive Unterbrechung einer Task-Ausführung definieren. Eine instrumentierte Version des Betriebssystems muss deshalb für jeden dieser Zustandsübergänge einer Task oder eines Interrupts $t \in T$

Ereignis	**Parameter**		**Beschreibung**
$l_{\uparrow,t}$	CPU-Kern	Zeitstempel	Zustandswechsel einer Task *t* von einem beliebigen Zustand nach *Running*; Eintritt in eine Interrupt Service Routine *(ISR)*
$l_{\downarrow,t}$	CPU-Kern	Zeitstempel	Zustandswechsel einer Task *t* von *Running* in einen beliebigen Zustand; Austritt aus einer Interrupt Service Routine *(ISR)*

Tabelle 5.2: Minimalumfang der für die Rekonstruktion der Task-Knoten eines Task-Graphen erforderlichen Instrumentierung

einen Log-Eintrag $l_{\uparrow,t}$, $l_{\downarrow,t}$ mit dem aktuellen Zeitstempel und der entsprechenden Task-ID respektive Interrupt-Nummer erstellen.

Zur Rekonstruktion der in einem System zur Laufzeit stattfindenden Task-Interaktionen mit der in Abschnitt 5.4.1 definierten Produzenten-Konsumenten-Semantik sind darüber hinausgehende Laufzeitinformationen nötig. Diese lassen sich ebenfalls mittels einer Betriebssysteminstrumentierung aufzeichnen, da die entsprechenden Interaktionen stets per Syscall angestoßen werden. Unsynchronisierte Lese- und Schreibzugriffe auf gemeinsame Daten werden hingegen gemäß Abschnitt 5.4.1 im Folgenden nicht berücksichtigt. Im Rahmen dieser Arbeit seien für Task-Interaktionen mit einer Produzenten-Konsumenten-Semantik in Anlehnung an [158] die in Abbildung 5.11 dargestellten Alternativen wie folgt definiert:

Definition 14 (Direkte Task-Interaktion) *Eine direkte Task-Interaktion zeichnet sich dadurch aus, dass der Produzent als Ziel der Interaktion explizit deren Konsumenten nennt. Der Konsum der Interaktion muss wiederum durch den Konsumenten explizit angestoßen werden.*

Definition 15 (Indirekte Task-Interaktion) *Eine indirekte Task-Interaktion zeichnet sich dadurch aus, dass der Produzent als Ziel der Interaktion ein Medium nennt. Der Konsum der Interaktion aus einem Medium muss wiederum durch den Konsumenten explizit angestoßen werden.*

Sowohl das Medium der indirekten Interaktion als auch der konsumentenseitig zum Empfang von Interaktionen vorgesehene Speicher bei direkten Interaktionen ist dabei als Puffer mit einer definierten Kapazität realisiert. Eine spezifische Semantik, wie beispielsweise *FIFO*, wird für den Konsum von Interaktionen aus einem Puffer generell nicht unterstellt.

Definition 16 (Sendeoperation) *Eine Sendeoperation wird stets durch den Produzenten einer Interaktion angestoßen und produziert genau eine Interaktion. Eine Sendeoperation nennt im Fall einer direkten Interaktion als Ziel einen Konsumenten und im Fall einer indirekten Interaktion ein Interaktionsmedium. Ist eine Sendeoperation nicht unmittelbar erfolgreich, so handelt es sich bei einer nichtblockierenden Operation um einen Fehlschlag. Bei blockierenden Sendeoperationen kann der Erfolg hingegen noch innerhalb eines definierten Zeitintervalls* (Timeout) *eintreten, so dass die Operation erst nach Ablauf dieses Intervalls als fehlgeschlagen gewertet wird.*

Als Grund für den Fehlschlag einer Sendeoperation ist das Szenario definiert, dass die Interaktion nicht beim Konsumenten (direkte Interaktion) respektive Medium (indirekte Interaktion) eingereiht werden kann. Dieser Fall kann eintreten, wenn die Kapazität des Konsumenten respektive des Mediums erschöpft ist, da vorherige Sendeoperationen nicht konsumiert wurden.

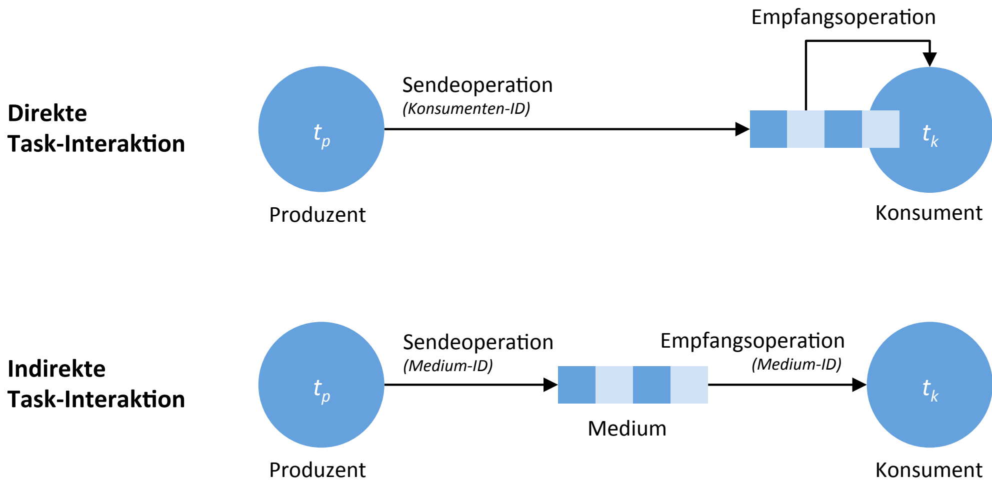


Abbildung 5.11: Arten der Interaktion einer Produzenten-Task t_p mit einer Konsumenten-Task t_k

Definition 17 (Empfangsoperation) *Eine Empfangsoperation wird stets durch den Konsumenten einer Interaktion angestoßen und konsumiert genau eine Interaktion. Eine Empfangsoperation hat im Fall einer direkten Interaktion keine Argumente und nennt im Fall einer indirekten Interaktion ein Interaktionsmedium. Ist eine Empfangsoperation nicht unmittelbar erfolgreich, so handelt es sich bei einer nichtblockierenden Operation um einen Fehlschlag. Bei blockierenden Empfangsoperationen kann der Erfolg hingegen noch innerhalb eines definierten Zeitintervalls (Timeout) eintreten, so dass die Operation erst nach Ablauf dieses Intervalls als fehlgeschlagen gewertet wird.*

Die Empfangsoperation einer direkten Interaktion schlägt genau dann fehl, wenn die zu empfangende Interaktion nicht durch einen Konsumenten zur Verfügung gestellt wurde. Der Fehlschlag der Empfangsoperation einer indirekten Interaktion ist wiederum dadurch begründet, dass im angegebenen Medium keine konsumierbare Interaktion vorliegt.

Basierend auf diesen Definitionen wird von einer Instrumentierung direkter und indirekter Interaktionen zunächst der in Tabelle 5.3 definierte Minimalumfang an aufzuzeichnenden Daten gefordert. Dieser wird im Rahmen der Modellausprägung für die Rekonstruktion der Task-Interaktionen vorausgesetzt. Eine Interaktion ist zudem stets mit dem Austausch von Daten verbunden. Da diese entsprechend Abschnitt 5.4.1 einen relevanten Aspekt des Overheads von Inter-Core-Interaktionen darstellen, sollten im Rahmen der Instrumentierung zudem Informationen über das bei einer direkten oder indirekten Interaktion transferierte Datenvolumen aufgezeichnet werden. Weiterhin müssen fehlgeschlagene Sende- oder Empfangsoperationen in der Aufzeichnung entweder entsprechend markiert werden oder dürfen nicht aufgezeichnet werden. Dies gilt auch für Operationen, die erst mit Ablauf eines Timeouts fehlschlagen.

Art	Direkte Interaktion		Indirekte Interaktion	
Typ	Sendeoperation	Empfangsoperation	Sendeoperation	Empfangsoperation
Daten	Produzenten-ID Konsumenten-ID (Datenvolumen)	Konsumenten-ID (Datenvolumen)	Produzenten-ID Medium-ID (Datenvolumen)	Konsumenten-ID Medium-ID (Datenvolumen)

Tabelle 5.3: Minimalumfang der bei Task-Interaktionen aufzuzeichnenden Daten

5.4.4 Modellausprägung

Wurde eine Laufzeitaufzeichnung einer Firmware erstellt, so kann aus dieser ein Task-Graph generiert werden, der die dynamische Systemcharakteristik mit Fokus auf die Exploration und Evaluation effizienter Mapping-Alternativen abbildet. Die nachfolgend beschriebene Vorgehensweise ist dabei auch zur Modellierung einer bereits auf mehrere Kerne verteilten Firmware geeignet, um gegebenenfalls effizientere Alternativen des Mappings zu finden.

Modellierung der Tasks

In einem ersten Schritt wird auf Basis der Laufzeitaufzeichnung für jede Task, die im Laufe des Profilings mindestens einmal den Zustand *Running* hatte, ein Knoten im Graphen instanziiert. Zudem wird jede innerhalb des Aufzeichnungsintervalls mindestens einmal ausgeführte Interrupt Service Routine als Knoten modelliert. Als Knotengewicht wird dabei der Lastanteil $f_{l_t}(t)$ der jeweiligen Task respektive des jeweiligen Interrupts gewählt. Die Rekonstruktion dieser Lastanteile basiert ausschließlich auf den in Tabelle 5.2 definierten Log-Einträge. Das dabei angewandte Vorgehen für eine Task oder einen Interrupt t gestaltet sich entsprechend dem in Algorithmus 5.1 dargestellten Pseudocode wie folgt: Generell werden bei der Rekonstruktion die Log-Ereignisse L der Aufzeichnung chronologisch nach den für t relevanten Einträgen $l_{\uparrow,t}$, $l_{\downarrow,t}$ durchmustert. Wurde ein Ereignis $l_{\uparrow,t}$ gefunden, das somit den Beginn der Ausführung einer Task respektive eines Interrupts t markiert, so wird dessen Zeitstempel $\phi(l_{\uparrow,t})$ temporär gesichert. Die chronologische Durchmusterung von L wird nun fortgesetzt, bis ein Eintrag $l_{\downarrow,t}$ gefunden wird, der somit die Ausführung von t zum Zeitpunkt $\phi(l_{\downarrow,t})$ beendet. Die entsprechende Zeitdifferenz $\phi(l_{\downarrow,t}) - \phi(l_{\uparrow,t})$ entspricht nun der Ausführungsdauer von t bei dieser konkreten Aktivierung. Auf diese Weise wird sichergestellt, dass Ausführungszeiten des Betriebssystem-Kernels zwischen einem Ereignis $\phi(l_{\downarrow,t_i})$ einer Task t_i und einem unmittelbar nachfolgenden Ereignis $\phi(l_{\uparrow,t_j})$ einer Task t_j nicht den Lastanteilen der Tasks t_i, t_j zugerechnet werden. Die Zeitdauer $\phi(l_{\downarrow,t}) - \phi(l_{\uparrow,t})$ wird nun zu der mit 0 initialisierten Gesamtlaufzeit f_{l_t} der entsprechenden Task t addiert. Nun wird in L die Suche nach weiteren, für t relevanten Ereignissen $\phi(l_{\uparrow,t}), \phi(l_{\downarrow,t})$ fortgesetzt und die zuvor genannten Schritte gegebenenfalls wiederholt. Wurde L vollständig durchmustert, wird der kumulierte Wert f_{l_t} schließlich durch die Zeitdifferenz zwischen der ersten Aktivierung und der letzten Deaktivierung einer Task oder eines Interrupts im Zeitraum der Aufzeichnung dividiert. Auf diese Weise ist der Lastanteil einer Task oder eines Interrupts stets bezüglich der im Aufzeichnungsintervall maximal verfügbaren Rechenzeit eines CPU-Kerns normiert. Zu beachten ist, dass die bei der Berechnung des Laufzeitanteils erreichte Genauigkeit direkt mit der Genauigkeit der Zeitstempel des Profilings

Eingabe: Array L chronologischer Log-Ereignisse mit Zeitstempeln $\phi\colon L \to \mathbb{R}$; $t \in T$.
Ausgabe: Lastanteil $f_{l_t}(t)$.
begin

$x_1 \leftarrow \min_{t' \in T} \left(\phi \left(l_{\uparrow,t'} \right) \right)$;
$x_2 \leftarrow \max_{t' \in T} \left(\phi \left(l_{\downarrow,t'} \right) \right)$;
$f_{l_t} \leftarrow 0$; $r \leftarrow$ false;
$x_3 \leftarrow 0$;
foreach $l \in L$ **do**
  if $l = l_{\uparrow,t}$ **then**
    $x_3 \leftarrow \phi(l)$; $r \leftarrow$ true;
  if $l = l_{\downarrow,t}$ **and** r = true **then**
    $f_{l_t} \leftarrow f_{l_t} + \phi(l) - x_3$;
    $x_3 \leftarrow 0$; $r \leftarrow$ false;
$f_{l_t} \leftarrow \frac{f_{l_t}}{x_2 - x_1}$;

Algorithmus 5.1: Berechnung des Lastanteils $f_{l_t}(t)$ einer Task $t \in T$

korreliert. Darüber hinaus wird für jeden Knoten des Graphen die Nummer des CPU-Kerns gesichert, auf dem die entsprechende Task oder der Interrupt während des Profilings ausgeführt wurde. Diese Informationen sind mittels der zu jedem Log-Eintrag gesicherten Nummer des CPU-Kerns, in dessen Kontext das Ereignis stattfand, rekonstruierbar.

Modellierung der Interaktionen

In einem weiteren Schritt wird mittels einer Analyse des Laufzeit-Logs die Menge I der im System während der Aufzeichnung aufgetretenen Task-Interaktionen mit Produzenten-Konsumenten-Semantik rekonstruiert. Dabei sind pro Interaktion $i \in I$ folgende Informationen zu extrahieren, deren Gesamtheit i in eindeutiger Weise definiert: Der Interaktionsmechanismus (z.B. Nachricht, Ereignis, Semaphor), die Medium-ID und optional das Datenvolumen.

Zu jeder Interaktion $i \in I$ gilt es nun, deren Häufigkeiten $f_h\colon I \times T \times T \to \mathbb{R}$ zwischen produzierenden Tasks $t_p \in T$ und konsumierenden Tasks $t_k \in T$ im Aufzeichnungsintervall zu bestimmen. Dabei werden ausschließlich Laufzeitdaten auf Basis der in Abschnitt 5.4.3 definierten Instrumentierung vorausgesetzt. Die Häufigkeiten $f_h(i, t_p, t_k)$ bilden schließlich die Grundlage für die Modellierung der gewichteten Interaktionskanten E_i des Task-Graphen. Eine Validierung der nachfolgend beschriebenen Vorgehensweise erfolgt in Abschnitt 5.4.5.

Zunächst sei eine direkte Task-Interaktion $i \in I$ einer Produzenten-Task $t_p \in T$ mit einer Konsumenten-Task $t_k \in T$ betrachtet. Es sei die Menge $S_i(t_p, t_k)$ der erfolgreichen Sendeoperationen von i durch t_p an t_k definiert. Damit sei die Häufigkeit $f_h(i, t_p, t_k)$ einer Interaktion i zwischen den Tasks t_p und t_k wie folgt bestimmt:

$$f_h\left(i, t_p, t_k\right) := \left| S_i\left(t_p, t_k\right) \right| \tag{5.4}$$

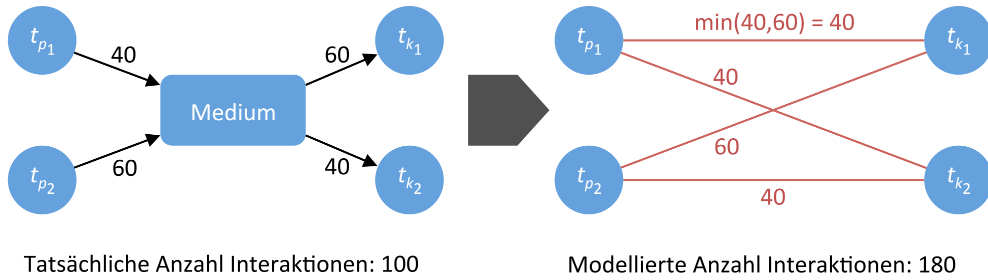


Abbildung 5.12: Modellierung indirekter Interaktionen nach Formel 5.5

Somit wird jede erfolgreiche Sendeoperation einer direkten Interaktion als Interaktion zwischen dem Produzenten t_p und dem Konsumenten t_k gewertet.

Anders gestaltet sich die Modellierung indirekter Task-Interaktionen. Die Herausforderung der Modellierung besteht hier darin, unter den eingeschränkten Informationen einer Minimalinstrumentierung einen bipartiten Graphen, bestehend aus Tasks und Interaktionsmedien, in Produzenten-Konsumenten-Relationen zu transformieren (vgl. Abbildung 5.12). Dieser Vorgang ist trivial, wenn es zu einem Medium genau einen Produzenten und einen Konsumenten gibt. Problematisch ist hingegen die Nachverfolgung der Interaktionen bei $n > 1$ Produzenten und $m > 1$ Konsumenten, die auf ein gemeinsames Medium zugreifen. Zwar liefert die Minimalinstrumentierung die Information, dass zu einem spezifischen Zeitpunkt eine Interaktion in das Medium eingereiht und zu einem späteren Zeitpunkt aus diesem entnommen wird. Auf Basis dieser Informationen lässt sich allerdings der Weg einer Interaktion mit den in Abschnitt 5.4.3 definierten Eigenschaften von ihrem Produzenten zu ihrem Konsumenten nicht nachverfolgen. Zusätzlich müsste der Puffer des Mediums strikt nach dem FIFO-Prinzip arbeiten und zu Beginn der Aufzeichnung leer sein. Diese Eigenschaften können allerdings in der Realität nicht gefordert werden und wurden somit auch nicht als Voraussetzung für die Rekonstruktion definiert. Deshalb wird zur Rekonstruktion indirekter Interaktionen eine Approximation gewählt. Dabei sei $S_i(t_p, m_i)$ die Menge der erfolgreichen Sendeoperationen einer indirekten Interaktion $i \in I$ durch eine Produzenten-Task $t_p \in T$ in das Medium m_i der Interaktion i. Analog dazu sei $R_i(t_k, m_i)$ die Menge der erfolgreichen Empfangsoperationen dieser Interaktion i aus dem Medium m_i durch eine Konsumenten-Task $t_k \in T$. Die Häufigkeit $f_h(i, t_p, t_k)$ einer indirekten Interaktion i zwischen t_p und t_k bestimme sich nun als

$$f_h\left(i, t_p, t_k\right) := \min\left(\left|S_i\left(t_p, m_i\right)\right|, \left|R_i\left(t_k, m_i\right)\right|\right). \tag{5.5}$$

Dieser Wert stellt eine sichere obere Schranke hinsichtlich der Häufigkeit der indirekten Interaktion i zwischen t_p und t_k dar, da diese über m_i nicht mehr Objekte austauschen können als das Minimum aus der Häufigkeit an $S_i(t_p, m_i)$- und $R_i(t_k, m_i)$-Operationen. Allerdings kann diese obere Schranke eine deutliche Überschätzung der tatsächlichen Häufigkeit an Interaktionen liefern. Abbildung 5.12 stellt diesen Zusammenhang dar: Während insgesamt 100 Interaktionen stattgefunden haben, werden auf diese Weise in Summe 180 Interaktionen modelliert. Deshalb

sei als Alternative eine probabilistische Approximation $f'_h: I \times T \times T \to \mathbb{R}$ der Häufigkeit einer indirekten Interaktion i zwischen t_p und t_k über das Medium m_i definiert:

$$f_{h'}\left(i, t_p, t_k\right) := \frac{\min\left(\sum_{t\in T}\left|S_i\left(t, m_i\right)\right|, \sum_{t\in T}\left|R_i\left(t, m_i\right)\right|\right)}{\sum_{t_1\in T}\sum_{t_2\in T} f_h\left(i, t_1, t_2\right)} \cdot f_h\left(i, t_p, t_k\right) \tag{5.6}$$

Dabei wird jede obere Schranke $f_h(i, t_p, t_k)$ mit einem Normierungsfaktor verrechnet und damit in probabilistischer Weise approximiert. Dieser Faktor berechnet sich als Quotient aus der tatsächlichen Häufigkeit der Interaktion i, die im System über das Medium m_i abgewickelt wurde und der Häufigkeit von i über m_i, die sich kumuliert über alle Task-Paare aus T mittels der oberen Schranke f_h ergibt.

In einem weiteren Schritt der Modellierung werden nun die Interaktionskanten E_i des Task-Graphen generiert. Für eine übersichtlichere Darstellung wird dabei pro Task-Paar t_1, t_2 maximal eine Interaktionskante $e = (t_1, t_2)$ hinzugefügt. Diese speichert dann mittels der in Abschnitt 5.3.3 beschriebenen Erweiterung des *GraphML*-Schemas alle Interaktionen der folgendermaßen definierten Menge $I_e \subseteq I$:

$$I_e := \left\{i \in I \,\middle|\, f_{h/h'}\left(i, t_1, t_2\right) > 0\right\} \quad \text{mit } e = \left(t_1, t_2\right) \tag{5.7}$$

Folgende Daten werden dabei pro Interaktion $i \in I_e$ gesichert: Der Interaktionsmechanismus, die Medium-ID, das optionale Datenvolumen und die für den Aufzeichnungszeitraum entsprechend einer der zuvor definierten Metriken f_h respektive $f_{h'}$ rekonstruierte Häufigkeit. Die Darstellung einer Interaktionskante $e \in E_i$ im *GraphML*-Format zeigt Abbildung 5.13.

Die Evaluation von Mapping-Alternativen erfordert schließlich die Reduktion jeder Interaktionskante $e \in E_i$ auf ein skalares Kantengewicht $f_g: E_i \to \mathbb{R}$. Im einfachsten Fall berechne sich dieses für $e \in E_i$ als die kumulierte Häufigkeit f_h respektive $f_{h'}$ aller Interaktionen $i \in I_e$:

$$f_g\left(e\right) := \sum_{i\in I_e} f_{h/h'}\left(i, t_1, t_2\right) \quad \text{mit } e = \left(t_1, t_2\right) \tag{5.8}$$

Dies hat jedoch zur Folge, dass alle Interaktionen hinsichtlich ihres potentiellen Overheads im Fall einer Abwicklung über Kerngrenzen als äquivalent angesehen werden. Es kann jedoch unterstellt werden, dass dies vor allem bei Interaktionen, die mit dem Austausch eines signifikanten Datenvolumens einhergehen, eine Abstraktion darstellt, da der Overhead mit dem über Kerngrenzen ausgetauschten Datenvolumen wächst. Aufgrund der divergierenden Implementierungen verschiedener Interaktionsmechanismen, wie beispielsweise Nachrichten und Ereignisse, ist darüber hinaus zu vermuten, dass sich auch diese hinsichtlich des potentiellen Overheads unterscheiden. Aus diesem Grund ist in der *EEEPA*-Toolchain als Alternative zur einfachen kumulativen Verrechnung (vgl. Formel 5.8) die Möglichkeit zur gewichteten kumulativen Verrechnung vorgesehen, bei der für jeden Interaktionstyp die Häufigkeit der Interaktionen mit einem spezifischen Gewicht verrechnet wird. Dieses Vorgehen ist zulässig, da die Gewichte von Interaktionskanten ohnehin nur einer relativen und nicht einer absoluten Bewertung der Inter-Core-Interaktionen eines konkreten Mappings dienen. Wie zuvor erwähnt, definiert I die Menge der Interaktionen, die auf Grundlage des Logs modelliert wurden. Weiterhin sei Ψ

```
<edge id="e152" source="n35" target="n13">
  <data key="ea0">interaction</data>
  <data key="eai">
    <interactions>
      <message type="VxWorks" id="08E84408_133" count="107" size="133" />
      <message type="VxWorks" id="08F21534_404" count="123" size="404" />
      <message type="VxWorks" id="08F21534_1872" count="109" size="1872" />
      <message type="VxWorks" id="08E2DD5C_133" count="2157" size="133" />
      <message type="VxWorks" id="08E2DD5C_144" count="2" size="144" />
      <semaphore type="VxWorks" id="08F20D54" count="5" />
      <semaphore type="VxWorks" id="01944B7C" count="67" />
      <semaphore type="VxWorks" id="08F20CDC" count="5" />
    </interactions>
  </data>
</edge>
```

Abbildung 5.13: Darstellung einer Interaktionskante im *GraphML*-Format

die Menge aller in einem spezifischen System definierten Interaktionstypen, wobei Ψ bei Interaktionstypen mit signifikantem Datenvolumen ein separates Element für jedes beim Profiling aufgezeichnete Datenvolumen dieser Interaktion enthält:

$$\Psi = \left\{\text{Event}, \text{Semaphor}, \text{Message_1B}, \dots, \text{Message_768B}, \dots, \text{Message_4096B}, \dots\right\} \quad (5.9)$$

Die Funktion $f_\Psi \colon I \to \Psi$ ordne nun im Rahmen einer Klassifikation jeder Interaktion $i \in I$ das entsprechende Element aus Ψ zu. Weiterhin bilde eine sogenannte *Benchmark-Funktion* $f_b \colon \Psi \to \mathbb{R}$ jedes Element aus Ψ auf ein reellwertiges Gewicht ab. Das Kantengewicht $f_g \colon E_i \to \mathbb{R}$ einer Interaktionskante $e \in E_i$ ergebe sich nun bei der gewichteten kumulativen Verrechnung der Häufigkeiten f_h respektive $f_{h'}$ wie folgt:

$$f_g(e) \coloneqq \sum_{i \in I_e} f_b\left(f_\Psi(i)\right) \cdot f_{h/h'}(i, t_1, t_2) \quad \text{mit } e = (t_1, t_2) \quad (5.10)$$

Nun gilt es, die Funktion $f_b \colon \Psi \to \mathbb{R}$ zu definieren. Diese muss für einen spezifischen Interaktionstyp aus Ψ einen Wert liefern, der den potentiellen Overhead dieser Interaktion bei einer Abwicklung über Kerngrenzen in Relation zu den übrigen Elementen in Ψ beschreibt. Wie in Abschnitt 5.4.1 erläutert, haben Inter-Core-Interaktionen tendenziell eine höhere Latenz und führen zu einer höheren Systemauslastung. Da beide dieser Arten von Overhead für eine Parallelisierung in der Automatisierungstechnik relevant sind, definiert die *EEEPA*-Toolchain zwei alternative Verfahren, um f_b zu definieren:

- Das Latenz-Benchmarking ermittelt die Differenz der Latenzen, welche die entsprechende Interaktion aufweist, wenn die Produzenten- und Konsumenten-Task auf verschiedenen und auf dem gleichen Kern ausgeführt werden.
- Beim Last-Benchmarking wird die Differenz der Systemlasten ermittelt, die in einem System entstehen, wenn die entsprechende Interaktion zwischen Tasks auf verschiedenen und auf dem gleichen Kern stattfindet. Dabei empfiehlt sich eine Erfassung der vollständigen Systemlast als Komplement der Last der Idle-Tasks der einzelnen Kerne.

Das Benchmarking ist für eine spezifische Zielplattform der Parallelisierung in Form einer Hardware und eines Betriebssystems einmalig durchzuführen. Die dabei gewonnenen Ergebnisse können dann immer für die Definition von f_b für eine modellbasierte Evaluation von Task-Mappings auf eben dieser Plattform verwendet werden. Eine Implementierung dieser Benchmarks für das Betriebssystem *VxWorks* erfolgte in [72].

Um den Aufwand des Benchmarking zu reduzieren, werden bei Interaktionen mit Datenvolumina $v \in V$ in der Regel nur für ausgewählte Volumina v' Benchmark-Werte x_v ermittelt. Da im System in der Regel jedoch beliebige Volumina innerhalb definierter Grenzen aufgezeichnet werden, muss f_b auf allen ganzzahligen Werten innerhalb dieser Grenzen definiert sein. Für jeden Interaktionstyp mit Datenvolumen sind deshalb für eine Funktion $\Phi(v; a_0, \ldots, a_n)$ die Parameter $a_0, \ldots, a_n$ so zu bestimmen, dass diese ein beliebiges Datenvolumen $v \in V$ innerhalb der vom System definierten Grenzen so auf eine reellwertige Zahl abbildet, dass für alle zuvor gemessenen Stützstellen (v', x_v) gilt: $\Phi(v'; a_0, \ldots, a_n) = x_v$. Somit definiert $\Phi(v; a_0, \ldots, a_n)$ für einen spezifischen Interaktionsmechanismus den Wert von f_b für eine Interaktion mit einem beliebigen Datenvolumen v innerhalb der definierten Grenzen. Dieses Verfahren wird als *Interpolation* bezeichnet, die Vor- und Nachteile entsprechender Methoden diskutiert [56].

Abschließend gilt es, die Bewertung eines Mappings $f_m \in C^T$ für einen Task-Graphen mit Tasks T auf eine Menge von CPU-Kernen C hinsichtlich der Inter-Core-Interaktionen zu spezifizieren. Zu diesem Zweck sei die Menge $E_i' \subseteq E_i$ der Interaktionskanten definiert, deren inzidente Tasks $t_j, t_k \in T$ unter f_m unterschiedlichen CPU-Kernen zugewiesen sind:

$$E_i' := \left\{ \left(t_j, t_k\right) \in E_i \,\middle|\, f_m\left(t_j\right) \neq f_m\left(t_k\right) \right\} \tag{5.11}$$

Die Bewertungsmetrik $f_i \colon C^T \to [0,1]$ der Inter-Core-Interaktionen eines parametrierten Task-Graphen berechne sich schließlich auf Basis der Kantengewichte $f_g \colon E_i \to \mathbb{R}$:

$$f_i\left(f_m\right) := \frac{\sum\limits_{e' \in E_i'} f_g\left(e'\right)}{\sum\limits_{e \in E_i} f_g\left(e\right)} \tag{5.12}$$

Auf diese Weise erfolgt die Bewertung der Inter-Core-Interaktionen eines spezifischen Mappings stets in Relation zu den insgesamt im Modell abgebildeten Interaktionen. Dies ermöglicht den diesbezüglichen Vergleich von Task-Graphen, die unter variierenden Aufzeichnungsintervallen generiert wurden und somit eine davon abhängige Anzahl an Interaktionen aufweisen.

Modellierung der Aufwände

In einem letzten Schritt wird der bislang generierte Task-Graph um die in Abschnitt 5.4.1 definierten Implementierungsaufwände erweitert. Da die Modellierung auf Systemebene bereits von einem Multitasking-System ausgeht, reduzieren sich die zu leistenden Anpassungen in erster Linie auf die Implementierung multilateraler Synchronisationen[9].

[9] Der Begriff der multilateralen Synchronisation wird in diesem Kontext synonym für alle Maßnahmen verwendet, die den sicheren Zugriff auf gemeinsam genutzte Ressourcen durch potentiell echt parallel ausführbare Tasks sicherstellen. Dazu zählt auch die Implementierung der Wiedereintrittsfähigkeit für Routinen, die durch potentiell echt parallel ausführbare Tasks aufgerufen werden (vgl. Abschnitt 2.5.2).

Zu beachten ist, dass Implementierungsaufwände keine dynamische, auf ein konkretes Lastprofil bezogene Systemcharakteristik darstellen, so dass deren Modellierung nicht auf Basis dynamisch gewonnener Daten erfolgt. Stattdessen handelt es sich um eine statische Charakteristik einer spezifischen Systemimplementierung. Allerdings ist es nicht möglich, die Modellierung der Aufwände in automatisierter Weise aus einer statischen Analyse des Systems abzuleiten. Eine solche kann zwar im Code generell unsynchronisierte Speicherzugriffe identifizieren, aber aus den in Abschnitt 2.7 genannten Gründen meist nicht entscheiden, ob diese in potentiellem Konflikt zueinander stehen und somit eine Synchronisation erfordern. Zudem bleibt bei einer statischen Codeanalyse die Programmsemantik unberücksichtigt. Diese ist allerdings von großer Relevanz, um zu entscheiden, ob neben der Synchronisation des Speicherzugriffs gegebenenfalls noch umfangreichere Anpassungen zur Erzielung einer echt parallelen Ausführbarkeit durchzuführen sind. Somit können die erforderlichen Aufwandsabschätzungen nur von Entwicklern geleistet werden, die mit der Systemarchitektur und -struktur sowie mit der Programmsemantik vertraut sind. Entsprechende Schätzverfahren definiert [16].

Somit gilt es, eine geeignete Schnittstelle zwischen Entwicklern und dem *EEEPA*-Tool zu spezifizieren, um diesen eine Modellausprägung unter dem Aspekt der Implementierungsaufwände zu ermöglichen. Die Anforderungen an diese Schnittstelle stellen sich wie folgt dar:

- Die Schnittstelle muss die Formalisierung von Aufwandsabschätzungen unterstützen. Die dabei angewandte Vorgehensweise und die dazu eingesetzten Werkzeuge sollten zur Reduzierung des Einarbeitungsaufwands eng an die im Rahmen der Entwicklungsplanung üblicherweise angewandten Verfahren angelehnt sein.
- Da zur parallelen Ausführung zweier Tasks häufig mehrfache Anpassungen zu leisten sind, darf die Schnittstelle keine Aufwandskanten des Graphen definieren, sondern nur einzelne Aufwände zwischen Tasks, die dann im Rahmen der Modellausprägung in geeigneter Weise zu einer Aufwandskante zusammengefasst werden.
- Die Schnittstelle sollte ein kollaboratives Vorgehen der Entwickler bei der Abschätzung der Aufwände ermöglichen. So kann ein Entwickler beispielsweise nur die Aufwände einzelner Aspekte der Parallelisierung bewerten oder sich auf die Module des Systems beschränken, bei denen er über detailliertes Wissen verfügt und somit potentiell bessere Aufwandsabschätzungen liefert.
- Die Schnittstelle muss in effizienter Weise durch einen automatisierten Parser ausgewertet werden können, der im Rahmen der Modellausprägung die Aufwandskanten des Task-Graphen modelliert.
- Für die formalisierten Aufwandsabschätzungen muss die Möglichkeit zur persistenten Speicherung gegeben sein, um die Ausprägung einer Vielzahl von Modellen durch Aufwandskanten in effizienter Weise durchführen zu können.

Als Konsequenz aus diesen Anforderungen wurde zur Formalisierung von Aufwandsabschätzungen das in Abbildung 5.14 dargestellte tabellarische Format gewählt. Dabei sind ausgehend von nicht multilateral synchronisierten Zugriffen auf gemeinsam genutzte Ressourcen für jedes Task-Paar des Systems die Aufwände abzuschätzen, die für eine echt parallele Ausführung erforderlich sind. Pro Aufwand sind dabei die Namen der beiden Tasks und eine numerische Bewertung des abgeschätzten Aufwands zu definieren. Zu beachten ist, dass vor Beginn der Formalisierung unter allen an der Abschätzung beteiligten Entwicklern eine für die gesamte

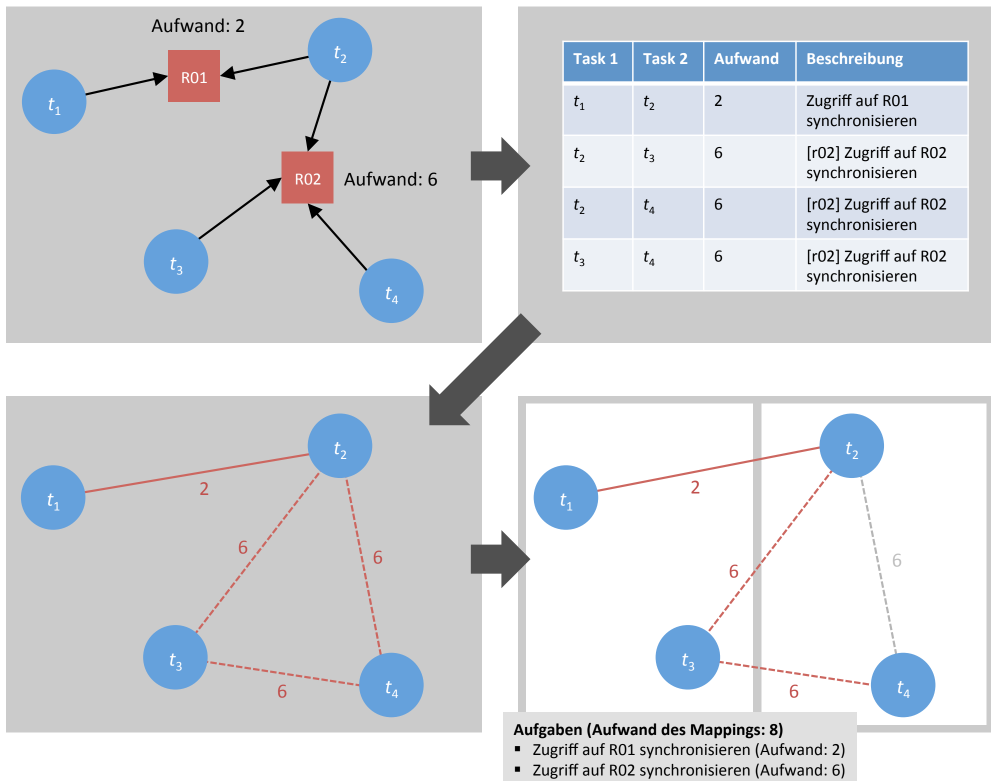


Abbildung 5.14: Der Modellierungsprozess für Aufwandsabschätzungen. Aufwände der multilateralen Synchronisation mit $n > 2$ beteiligten Tasks sind als gestrichelte Aufwandskanten dargestellt. Durch das Mapping werden drei Aufwandskanten geschnitten, von denen allerdings zwei mit dem identischen Präfix [r02] annotiert sind. Somit wird deren Aufwand in Höhe von 6 Einheiten bei der Kumulation nur einmal berücksichtigt, so dass sich ein Gesamtaufwand in Höhe von 8 Einheiten ergibt.

Modellierung eines Systems konsistente Metrik für die Aufwandsbewertung zu vereinbaren ist, um die kollaborative Aufwandsmodellierung zu ermöglichen. Dies kann beispielsweise eine Klassifizierung auf einer vorgegebenen Skala oder eine definierte temporale Einheit sein. Weiterhin ist pro Aufwand eine textuelle Beschreibung zu definieren, um aus einem konkreten Mapping eine Liste der für eine Realisierung zu leistenden Adaptionen generieren zu können.

Den Standardfall bildet die Formalisierung multilateraler Synchronisationen, bei denen ausschließlich durch zwei Tasks $t_1, t_2 \in T$ auf eine gemeinsame Ressource zugegriffen wird. Hier ist ein Aufwand zwischen t_1 und t_2 zu definieren, der im Fall $f_m(t_1) \neq f_m(t_2)$ zu leisten ist. Dieser Fall ist in Abbildung 5.14 für die Ressource R01 dargestellt.

Komplexer gestaltet sich die Situation bei multilateralen Synchronisationen zwischen Tasks $t_1, \ldots, t_n \in T$ mit $n > 2$ beim Zugriff auf eine gemeinsam genutzte Ressource. Sobald die Be-

dingung $\exists t_i, t_j \in \{t_1, \ldots, t_n\}: f_m(t_i) \neq f_m(t_j)$ erfüllt ist, ist für alle Tasks $t_1, \ldots, t_n$ mit einem entsprechenden Gesamtaufwand der explizit synchronisierte Zugriff auf diese Ressource zu implementieren. Dieser Fall ist in Abbildung 5.14 beispielhaft für die Ressource R02 dargestellt. Dabei wurde die Implementierung eines synchronisierten Zugriffs der Tasks t_2, t_3 und t_4 auf R02 mit einem Gesamtaufwand in Höhe von 6 Einheiten abgeschätzt. Da die tabellarische Formalisierung nur Aufwände zwischen Task-Paaren vorsieht, muss eine derartige Konstellation als eine Menge von k Aufwänden zwischen allen an dem entsprechenden Ressourcenzugriff beteiligten Tasks $t_1, \ldots, t_n \in T$ formalisiert werden. k ergibt sich über den Binomialkoeffizienten:

$$k = \binom{n}{2} = \frac{n!}{2! \cdot (n-2)!} = \frac{n \cdot (n-1)}{2} \tag{5.13}$$

Als Gewicht jedes dieser k Aufwände ist dabei stets der entsprechende Gesamtaufwand zu wählen. Zudem ist die textuelle Beschreibung aller k Aufwände um ein identisches eindeutiges Präfix in Form einer ressourcenspezifischen ID zu erweitern. In dem in Abbildung 5.14 dargestellten Beispiel ist dies das Präfix `[r02]`.

Durch den Parser wird nun die Tabelle mit den formalisierten Aufwandsabschätzungen eingelesen und dem Graphen genau dann eine Aufwandskante zwischen zwei Tasks hinzugefügt, wenn die Tabelle mindestens einen Aufwand zwischen diesen beiden Tasks definiert. Sind mehrere Aufwände zwischen diesen Tasks modelliert, so werden diese in der entsprechenden Kante als Liste gespeichert. Das Gewicht einer Aufwandskante entspricht stets dem kumulierten Gewicht aller für diese Kante definierten Aufwände.

Die Bewertung eines Mappings f_m hinsichtlich seines Implementierungsaufwands berechne sich schließlich als das kumulierte Gewicht aller Aufwandskanten, deren inzidente Tasks $t_i, t_j \in T$ unter f_m unterschiedlichen CPU-Kernen zugeordnet sind: $f_m(t_i) \neq f_m(t_j)$. Die Aufwände aller übrigen Aufwandskanten sind hingegen nicht zu leisten, da die jeweils inzidenten Tasks auch unter dem Mapping auf dem gleichen CPU-Kern ausgeführt werden und somit wie bisher implizit synchronisiert ausgeführt werden können. Mit einer identischen ressourcenspezifischen ID annotierte Aufwände werden bei der Kumulation der Aufwände allerdings nur einmal berücksichtigt, da das Gewicht eines einzelnen Aufwands bereits den zu leistenden Gesamtaufwand für die Implementierung einer multilateralen Synchronisation der entsprechenden Ressource berücksichtigt. Dies wird in Abbildung 5.14 deutlich.

5.4.5 Validierung

Dass ein ausgeprägter und parametrierter Task-Graph als Ergebnis einer analytischen Untersuchung die zu Beginn definierten Bewertungskriterien liefern kann, wurde bereits im Kontext der Spezifikation des Basismodells und der Modellausprägung gezeigt. Allerdings ist als generelle Ursache einer potentiell beeinträchtigten Validität des Modells das dynamische Profiling zu nennen, welches durch seinen Overhead das Laufzeitverhalten eines Systems und somit auch dessen in automatisierter Weise generierte Modelle beeinflusst. Weitere mögliche Einflüsse im Kontext der Systeminstrumentierung, der Modellausprägung und der Modellparametrierung werden nachfolgend diskutiert.

Validierung der Instrumentierung und Ausprägung

Bei der Modellierung auf Systemebene ist zunächst die Ausprägung der Task-Knoten, der Interaktionskanten und der Aufwandskanten hinsichtlich möglicher Defizite zu prüfen. Da mit einer Instrumentierung der Ereignisse $l_{\uparrow,t}$ und $l_{\downarrow,t}$ sowohl der Anfang als auch das Ende jeder Ausführung einer Task oder eines Interrupts t erfasst und zugeordnet werden kann, kann die Modellausprägung die korrekten Lastanteile der Tasks und Interrupts einer konkreten Ausführung der Firmware unter Voraussetzung ausreichend exakter Zeitstempel rekonstruieren. Diese Lastanteile sind allerdings nur dann korrekt, wenn auch die Idle-Task im Aufzeichnungsintervall zur Ausführung gekommen ist. Nur dann kann unterstellt werden, dass während des Profilings allen Tasks des Systems durch den Betriebssystem-Scheduler die beanspruchte Rechenkapazität zugeteilt wurde. Andernfalls müsste davon ausgegangen werden, dass bei niederprioren Tasks als Folge einer Verdrängung ein zu geringer Lastanteil modelliert werden würde.

Die Analyse der Ausprägung mit Interaktionskanten zeigt weiterhin, dass bei direkten Interaktionen alle für eine Rekonstruktion erforderlichen Informationen im Rahmen eines Profilings aufgezeichnet werden. So werden zu jeder erfolgreichen und somit tatsächlich durchgeführten Sendeoperation mit der Konsumenten-Task, der Produzenten-Task und dem Datenvolumen alle zur Modellierung der Interaktion erforderlichen Daten erfasst. Die bei der Modellierung direkter Interaktionen gemäß Formel 5.4 vorgenommene Beschränkung auf erfolgreiche Sendeoperationen kann insofern als valide gelten, als in einem korrekt funktionierenden System der Konsum aller produzierten Interaktionen innerhalb eines entsprechenden Zeitintervalls unterstellt werden kann. Ebenfalls korrekt ist die Modellierung indirekter Interaktionen, sofern jedes Interaktionsmedium des Systems nur einen Produzenten und einen Konsumenten hat. Hier wird im Modell die exakte Häufigkeit der über das Medium ausgetauschten Interaktionen abgebildet. Eingeschränkt ist hingegen die Präzision der Modellierung indirekter Interaktionen bei einer n-zu-m-Beziehung zwischen Produzenten und Konsumenten, da diese auf Basis von begrenztem Wissen über das dynamische Systemverhalten in probabilistischer Weise erfolgt. Die Modellierung bildet zwar die korrekte Häufigkeit der über ein Medium abgewickelten Interaktionen ab, kann diese aber Task-Paaren nur in approximativer Weise zuordnen. Eine präzisere Rekonstruktion würde hier jedoch eine signifikant umfangreichere Systeminstrumentierung mit den zuvor diskutierten Nachteilen erfordern.

In den vorherigen Abschnitten wurde für direkte und indirekte Task-Interaktionen eine generische Instrumentierung und Modellausprägung spezifiziert und validiert. Die Implementierung des *EEEPA*-Tools realisiert schließlich eine Rekonstruktion von Interaktionen unter dem Echtzeitbetriebssystem *VxWorks* für die folgenden Interaktionsmechanismen [186]:

- Als Mittel der direkten Interaktion existieren 32 binäre *Events*, die zwischen Tasks signalisiert werden können. Dazu sind zwei Syscalls definiert: `EventSend` zur Signalisierung einer Event-Maske an eine andere Task und `EventReceive` zum Warten auf eine spezifische Event-Maske.
- *Messages* sind ein Mittel der indirekten Task-Interaktion, bei denen eine Nachricht mit definiertem Datenvolumen mittels des Syscalls `MsgQSend` in eine Message-Queue eingereiht wird. Per `MsgQReceive` wiederum kann eine Nachricht aus einer Message-Queue empfangen werden. Die Message-Queue ist dabei nicht zwingend nach dem FIFO-Prinzip aufgebaut, da als dringend markierte Nachrichten möglich sind, die beim Versand direkt an den Beginn der Message-Queue geschoben werden.

- *Semaphore* werden zur unilateralen und multilateralen Synchronisation genutzt und stellen einen weiteren indirekten Interaktionsmechanismus dar. *VxWorks* definiert zu diesem Zweck binäre und zählende Semaphore, so dass diese ein Interaktionsmedium mit entsprechend definierter Puffergröße darstellen. Ein Semaphor wird durch den Syscall `SemTake` geholt und mittels `SemGive` freigegeben. Wartende Tasks werden entweder nach dem FIFO-Prinzip oder entsprechend ihrer Priorität bedient.

Problematisch ist hierbei, dass diese Interaktionsmechanismen in einigen Fällen die zuvor für Interaktionen definierten Eigenschaften verletzen. So besitzt der Event-Mechanismus bei der Empfänger-Task nicht den in Abschnitt 5.4.3 geforderten Puffer, sondern eine binäre Event-Maske. Ein mehrfaches `EventSend` eines identischen Events bleibt somit ohne Auswirkung oder Rückmeldung eines Fehlschlags und wird durch ein einzelnes `EventReceive` konsumiert. Zum Profiling wird in der *EEEPA*-Implementierung zudem der in Abschnitt 5.3.2 beschriebene *Wind River System Viewer* genutzt, da dieser unter anderem das Logging der Task-Zustandswechsel, der Ein- und Austritte in Interrupt Service Routinen und der Sende- und Empfangsoperationen von Task-Interaktionen mittels Events, Messages und Semaphoren unterstützt. Allerdings verletzt auch dieser Profiler eine Reihe der bei der Spezifikation der Instrumentierung geforderten Eigenschaften, da bei Sende- und Empfangsoperationen generell keine Informationen über den Erfolg oder Fehlschlag der entsprechenden Operation aufgezeichnet werden. Als Konsequenz ist die Korrektheit der Modellierung von Task-Graphen mittels der bei einem Profiling von *VxWorks* mit dem *Wind River System Viewer* gewonnenen Daten eingeschränkt. Um die Validität dieser Modelle zu erhöhen, müssten allerdings betriebssystemspezifische Erweiterungen des Profilings ebenso durchgeführt werden, wie eine Adaption des Profilers an die in Abschnitt 5.4.3 geforderte Spezifikation. Derartige Adaptionen sind allerdings weder Gegenstand dieser Arbeit noch wären diese im konkreten Fall von *VxWorks* möglich, da zu diesem Zweck eine Anpassung des Betriebssystemcodes erfolgen müsste.

Abschließend kann die zur Modellierung der Aufwände spezifizierte Formalisierung als valide angesehen werden, da für multilaterale Synchronisationen zwischen Tasks $t_1, \ldots, t_n \in T$ mit $n \geq 2$ die entsprechend erforderlichen Aufwände berücksichtigt werden. Die Modellausprägung ist wiederum trivial, da hierbei lediglich eine Konvertierung tabellarisch formalisierter Aufwände in kumulierte Aufwandskanten geleistet wird.

Validierung der Parametrierung

Zu beachten ist, dass bislang ausschließlich erörtert wurde, inwiefern die ausgeprägten Modelle auf Taskebene eine konkrete dynamische Ausführung einer Firmware auf einer Hardware mit einer spezifischen Anzahl an CPU-Kernen in valider Weise abbilden. Allerdings wurde nicht diskutiert, inwiefern die dabei generierten Task-Graphen auch nach einer Parametrierung durch ein Mapping f_m noch als valide Modelle einer dementsprechend realisierten Firmware-Parallelisierung gelten können. So wird hier durch das parametrierte Modell eine Prädiktion der Firmware-Charakteristik bei einer Ausführung unter einem anderen dynamischen Schedule auf einer gegebenenfalls anderen Hardware mit einer abweichenden Anzahl an CPU-Kernen gefordert. Als Konsequenz sind folgende Einflussfaktoren zu berücksichtigen:

- Unterscheidet sich die Hardware des Profilings von derjenigen, für die das Systemverhalten zu prädizieren ist, können sich die Relationen zwischen den Lastanteilen der Tasks verschieben, da sich aufgrund anderer Speicher- und Prozessorarchitekturen die Ausführungszeiten von Instruktionen unterscheiden können. Aus diesem Grund empfiehlt es sich, das Profiling bereits unter Ausführung der jeweiligen Firmware auf der entsprechenden Zielplattform durchzuführen.
- Den Laufzeiten von Tasks werden bei der Modellierung auch Phasen eines gegebenenfalls aktiven Wartens bei Synchronisationen zugeschlagen, deren Dauer aber vom jeweiligen dynamischen Schedule abhängig ist. Die aus einer parallelen Ausführung unter einem anderen dynamischen Schedule resultierenden Einflüsse auf die Lastanteile der Tasks können somit im Rahmen der Parametrierung nicht berücksichtigt werden.
- Die Produzenten-Konsumenten-Relationen indirekter Interaktionen können aus den zuvor genannten Gründen bei der Modellausprägung nur approximativ rekonstruiert werden. Allerdings ist die exakte Zuordnung von Produzenten und Konsumenten ohnehin spezifisch für den jeweiligen dynamischen Schedule. Dies gilt auch für die Produzenten-Konsumenten-Relationen indirekter Interaktionen auf der Zielplattform, so dass diese ebenfalls nur approximativ prädizierbar sind.
- Task-Synchronisationen, die für eine sichere echt parallele Ausführung eines partitionierten Schedulings noch zu implementieren sind, beeinflussen die Lastanteile der Tasks und generieren Interaktionen, die im ausgeprägten Modell noch nicht berücksichtigt sind.

Diese Aspekte führen in Summe dazu, dass die bei der Modellparametrierung unterstellte Neutralität der Task-Lastanteile und der Task-Interaktionen bezüglich einer Parametrierung des Graphen eine Abstraktion der Realität darstellt. Eine exaktere Prädiktion würde allerdings die inhärent nicht mögliche Prognose des dynamischen Schedules auf der Zielplattform und eine signifikant größere Modellierungstiefe erfordern. Zumindest der zweite Aspekt wäre zwar realisierbar, würde aber eine umfassendere Systeminstrumentierung bedingen, die entsprechend der Begründung in Abschnitt 5.4.3 abgelehnt wird. Unter dieser Prämisse können die erzielten Ergebnisse jedoch als für ihren Zweck ausreichend akkurat gelten, so dass die zuvor beschriebene Methode der Modellierung als valide bewertet werden kann.

5.5 Modellierung auf Taskebene

5.5.1 Bewertungskriterien

Gemäß Definition 7 besteht das Ziel einer Modellierung auf Taskebene[10] in der Exploration und Evaluation statischer Dekompositionen einer Task.

Definition 18 (Task-Dekomposition) *Als Task-Dekomposition wird der Prozess bezeichnet, bei dem eine Task $t \in T$ in eine Menge von Tasks T_d überführt wird, die in vollständig nebenläufiger oder synchronisierter Weise die Funktionalität von t bereitstellen. Eine Dekomposition er-*

[10] Im Gegensatz zur Modellierung auf Systemebene wird im Folgenden der Begriff *Task* nicht stellvertretend für Interrupts verwendet. Da Interrupts in realen Systemen zur Wahrung der Systemreaktivität sehr kurze Ausführungszeiten haben, ist deren Parallelisierung in der Regel nicht erforderlich. Somit ist die nachfolgend beschriebene Methode auf die Exploration und Evaluation von Alternativen der Task-Dekomposition optimiert.

folgt zum Entwurfszeitpunkt spezifisch für eine Menge von CPU-Kernen C und erfüllt die Bedingung $|T_d| \leq |C|$. Neben der Task-Menge T_d definiert eine Dekomposition deren Mapping $f_m\colon T_d \rightarrow C$ und somit ein partitioniertes Scheduling der resultierenden Tasks T_d. Dabei gelte für Tasks $t_{d_1}, t_{d_2} \in T_d$ stets

$$f_m\left(t_{d_1}\right) = f_m\left(t_{d_2}\right) \Longleftrightarrow t_{d_1} = t_{d_2}. \tag{5.14}$$

Somit ist das im Rahmen einer Dekomposition definierte Task-Mapping f_m injektiv.

Auch die Exploration geeigneter Task-Dekompositionen erfordert zunächst die Definition von Kriterien, unter denen potentielle Lösungen evaluiert werden. Um dies algorithmisch leisten zu können, ist auch hier deren Quantifizierung in Form geeigneter Metriken erforderlich.

Speedup

Gemäß Abschnitt 3.1.2 zielt eine Task-Dekomposition in erster Linie auf die Parallelisierung zyklischer Tasks ab, deren Ausführungszeit pro Zyklus in signifikanter Weise die Performanz des kompletten Systems beeinflusst. Aus diesem Grund wird bei der Dekomposition einer Task $t \in T$ in eine Menge von Tasks T_d die Performanz nicht dadurch erhöht, dass die Lastanteile der resultierenden Tasks wie bei einer Parallelisierung auf Systemebene optimal balanciert werden. Stattdessen besteht das erste Zielkriterium darin, durch die Task-Dekomposition die Ausführungsdauer $f_d(T_d)$ der Tasks T_d pro Zyklus zu reduzieren. Aus den Startzeitpunkten $f_s(t_d)$ und den Endzeitpunkten $f_e(t_d)$ der Ausführung der Tasks $t_d \in T_d$ ergibt sich dabei:

$$f_d\left(T_d\right) \coloneqq \max_{t_d \in T_d}\left(f_e\left(t_d\right)\right) - \min_{t_d \in T_d}\left(f_s\left(t_d\right)\right) \tag{5.15}$$

Dem steht die Ausführungsdauer $f_d(t)$ der ursprünglichen Task t gegenüber:

$$f_d\left(t\right) \coloneqq f_e\left(t\right) - f_s\left(t\right) \tag{5.16}$$

Daraus ergibt sich schließlich als Metrik zur Bewertung einer Task-Dekomposition der sogenannte *Speedup* $f_u(t, T_d)$ als Quotient aus der Ausführungsdauer der ursprünglichen Task t und der Ausführungsdauer der aus der Task-Dekomposition hervorgegangenen Tasks T_d:

$$f_u\left(t, T_d\right) \coloneqq \frac{f_d\left(t\right)}{f_d\left(T_d\right)} = \frac{f_e\left(t\right) - f_s\left(t\right)}{\max\limits_{t_d \in T_d}\left(f_e\left(t_d\right)\right) - \min\limits_{t_d \in T_d}\left(f_s\left(t_d\right)\right)} \tag{5.17}$$

Inter-Core-Interaktionen

Der potentielle Overhead von Inter-Core-Interaktionen wurde bereits im Rahmen der Modellierung auf Systemebene diskutiert. Zwar muss die Ausführung der Tasks T_d in geeigneter Weise synchronisiert werden, allerdings gilt es dabei, den Anteil der über Kerngrenzen abgewickelten Interaktionen aus den in Abschnitt 5.4.1 genannten Gründen zu reduzieren.

Implementierungsaufwand

Auch die Implementierung einer Task-Dekomposition ist mit Aufwänden verbunden. Eine effiziente Parallelisierung erfordert somit auch hier, dass den zuvor definierten Kriterien der Laufzeitperformanz die jeweiligen Implementierungsaufwände gegenübergestellt werden.

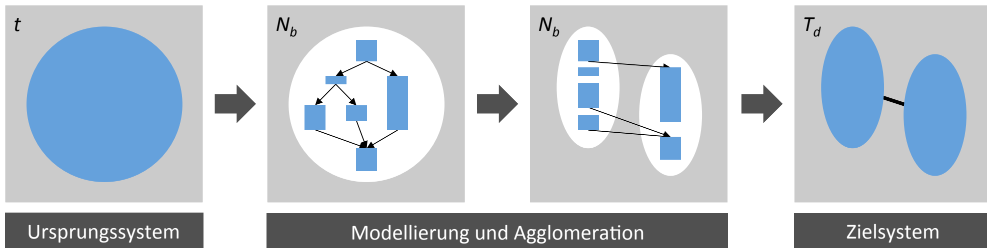


Abbildung 5.15: Stufen der Granularität bei der Parallelisierung auf Taskebene: Das Ursprungssystem in Form einer Task t wird als eine Menge von Codeblöcken N_b modelliert, die schließlich zu einer Menge von Tasks T_d agglomeriert werden.

5.5.2 Basismodell

Das Codeblock-Konzept

Die in Abschnitt 5.4 beschriebene Vorgehensweise zur Modellierung auf Systemebene hat als Vorbedingung eine Multitasking-Software. Tasks stellen hier die kleinste, einem CPU-Kern einzeln zuordenbare Einheit dar; eine darüber hinausgehende Granularisierung des Systems findet nicht statt. Somit ist die Granularität des Ursprungssystems identisch mit der des Modells und der des parallelisierten Zielsystems.

Davon abweichend gestaltet sich die Situation bei der Parallelisierung auf Taskebene. Da hier eine Task-Dekomposition durchgeführt werden soll, ist das Ziel-System T_d gegenüber dem monolithischen ursprünglichen System $t \in T$ feiner granularisiert. Dieser Zusammenhang ist in Abbildung 5.15 dargestellt. Eine weitere, davon unabhängige Ebene der Granularisierung in Form einer Menge von Objekten N_b ist bei der Modellierung der ursprünglichen Task t zu wählen. Dieser Schritt ist entscheidend, da ein Objekt $n_b \in N_b$ die kleinste im Rahmen der Dekomposition einem CPU-Kern einzeln zuordenbare Einheit definiere. Die Implementierung einer konkreten Dekomposition agglomeriert schließlich die Objekte N_b wieder zu Tasks T_d und generiert auf diese Weise ein Zielsystem mit einer gegenüber dem Modell geringeren Granularität. Der Entwurfsraum möglicher Dekompositionen einer Task t wäre dabei am größten, wenn jede Anweisung des entsprechenden Codes ein separates Objekt $n_b \in N_b$ bilden würde. Eine Modellierung in dieser Granularität mittels Profiling und Expertenwissen wäre allerdings aufgrund des hohen Profiling-Overheads und des Umfangs der erforderlichen Modellierungsaufwände in der Praxis nicht realisierbar. Dies erfordert eine Modellgranularität, die einen guten Kompromiss aus einer Einschränkung des Entwurfsraums und der Praxistauglichkeit der Modellierung darstellt. Deshalb seien zur Modellierung auf Taskebene sogenannte *Codeblöcke* N_b definiert:

Definition 19 (Codeblock) *Ein* Codeblock $n_b \in N_b$ *stellt das feingranularste Objekt einer Modellierung auf Taskebene dar. Er umfasst eine sequentielle Folge von Anweisungen der ursprünglichen Task t, die sich in jeder möglichen resultierenden Dekomposition T_d in einer Task $t_d \in T_d$ in eben dieser Reihenfolge wiederfinden lässt. Die Ausführung eines Codeblocks besitzt in der Anweisungsfolge stets einen eindeutigen Eintrittspunkt und einen eindeutigen Austrittspunkt.*

Zu beachten ist, dass die Forderung nach einem eindeutigen Eintrittspunkt und einem eindeutigen Austrittspunkt genau dann erfüllt ist, wenn sich ein Codeblock um zwei Anweisungen ergänzen lässt, von denen die eine in jeder möglichen Ausführung der Anweisungsfolge stets als erste und die andere stets als letzte Anweisung ausgeführt wird.

Abzugrenzen sind Codeblöcke von den aus dem Compilerbau bekannten *Basisblöcken*. Im Assemblercode eines Programms lässt sich ein Basisblock wie folgt definieren [70, 80]:

- Ein Basisblock beginnt mit dem Beginn eines Programms, dem Beginn einer Prozedur, einer als Sprungmarke im Programm auftretenden Anweisung oder einer direkt auf eine Sprung- oder Verzweigungsanweisung folgenden Anweisung.
- Ein Basisblock endet mit dem Ende eines Programms, dem Ende einer Prozedur oder mit einer Sprung- oder Verzweigungsanweisung. Somit endet ein Basisblock mit der letzten Anweisung vor Beginn eines anderen Basisblocks.

Somit besitzt auch ein Basisblock einen eindeutigen Eintrittspunkt und einen eindeutigen Austrittspunkt. Allerdings umfasst ein Basisblock eine Anweisungsfolge maximaler Länge, die in jeder möglichen Ausführung entweder in der angegebenen Reihenfolge oder gar nicht ausgeführt wird. Als Konsequenz sind die Basisblöcke einer Anweisungsfolge entsprechend den zuvor genannten Eigenschaften stets eindeutig determiniert. Ein Codeblock muss hingegen nicht die maximale Anweisungsfolge bis zur nächsten Änderung des Kontrollflusses umfassen und kann auch komplexe Programmstrukturen wie Schleifen, Verzweigungen und Funktionsaufrufe beinhalten. Der Anfang und das Ende eines Codeblocks ist somit weitestgehend indeterminiert, so dass sich für die Partitionierung einer Anweisungsfolge in Codeblöcke Freiheitsgrade ergeben.

Definition 20 (Codeblock-Partitionierung) *Die Partitionierung einer Folge von Anweisungen in Codeblöcke erfolgt in manueller Weise durch einen Entwickler unter Berücksichtigung der Bedingungen, die sich aus den Eigenschaften eines Codeblocks gemäß Definition 19 ergeben. Dabei ist sicherzustellen, dass bei jeder möglichen Ausführung des partitionierten Codes ein Codeblock erst dann über seinen eindeutigen Eintrittspunkt betreten wird, wenn der zuvor betretene Codeblock über seinen eindeutigen Austrittspunkt verlassen wurde. Der Vorgang der Partitionierung ist explizit unabhängig von gegebenenfalls einzuhaltenden Abhängigkeiten zwischen Anweisungen und liefert somit keine Aussage über die Parallelisierbarkeit einer Anweisungsfolge.*

Um diese Anforderungen bei der Codeblock-Partitionierung eines in einer Hochsprache implementierten Programms zu erfüllen, sind die nachfolgend genannten Bedingungen zu beachten. Entsprechende Beispiele sind dabei in Abbildung 5.16 annotiert und nachfolgend referenziert.

- Die instrumentierte Anweisungsfolge ist maximal auf den vollständigen Rumpf einer einzelnen Funktion respektive Prozedur beschränkt und umfasst somit keine weiteren Funktions- respektive Prozedurimplementierungen.
- Bedingte und unbedingte Sprünge sind innerhalb einer durch einen Codeblock gekapselten Anweisungsfolge generell nicht zulässig. Dies betrifft beispielsweise die `goto`-Anweisung ebenso wie die `break`- oder `continue`-Anweisung, die zu einem bedingten vorzeitigen Abbruch einer Schleife respektive Schleifeniteration führen.
- Funktionsaufrufe müssen innerhalb eines Codeblocks gekapselt sein (Referenz 1). Ein Codeblock kann dabei mehrere Funktionsaufrufe umfassen, die selbst wiederum geschachtelt sein können.

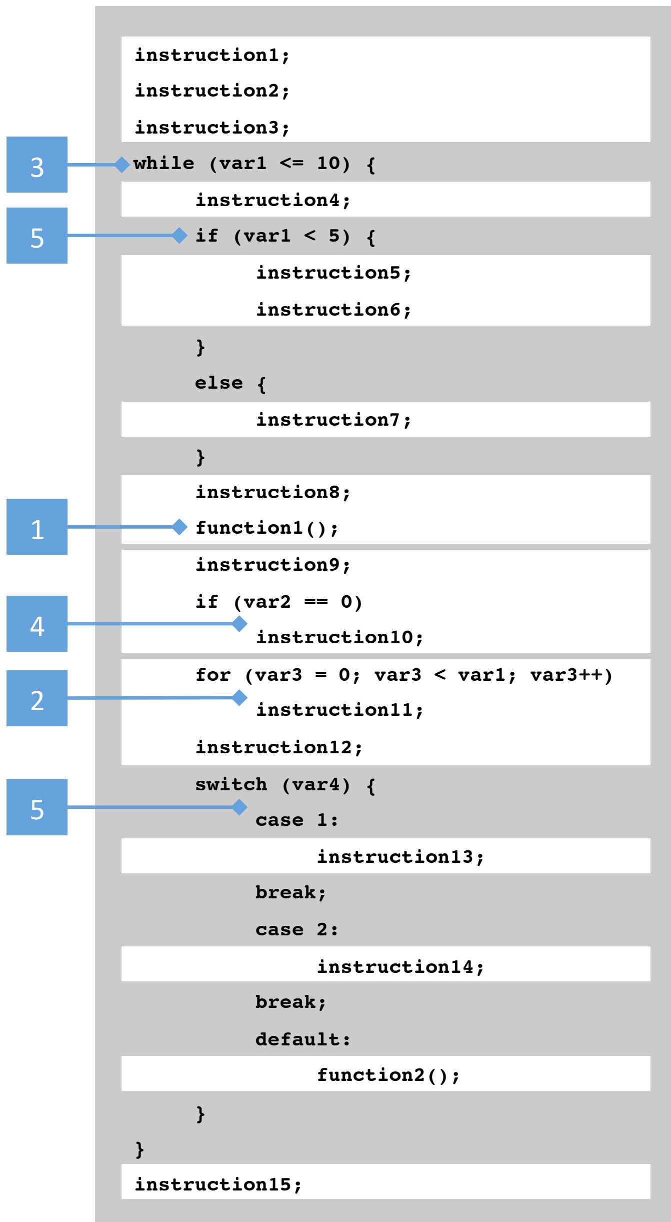


Abbildung 5.16: Beispielhafte Partitionierung einer Anweisungsfolge in Codeblöcke, die als weiße Rechtecke visualisiert sind. Die Referenzen beziehen sich auf die im Text genannten Bedingungen bei der Annotation von Codeblöcken.

- Schleifen sind innerhalb eines Codeblocks erlaubt, wenn sowohl der Schleifenrumpf als auch die Schleifenkondition vollständig im Codeblock gekapselt sind (Referenz 2). Ist eine Schleife nicht vollständig durch einen Codeblock gekapselt, so muss der Schleifenrumpf vollständig durch mindestens einen Codeblock gekapselt werden. Die Schleifenkondition ist dann nicht Bestandteil eines Codeblocks (Referenz 3).
- Verzweigungen sowie mehrfache Verzweigungen (*Switch*-Anweisungen) sind innerhalb eines Codeblocks erlaubt, wenn alle Alternativen vollständig im Codeblock gekapselt sind (Referenz 4). Ist eine Verzweigung nicht vollständig durch einen Codeblock gekapselt, so muss jede der Alternativen jeweils vollständig durch mindestens einen Codeblock gekapselt sein. Die Konditionen und Einsprungspunkte der Verzweigung sind nicht Bestandteil eines Codeblocks (Referenz 5).
- Mit Ausnahme der zuvor genannten Anweisungen muss bei der Definition der Codeblöcke jede Anweisung der entsprechenden Anweisungsfolge in genau einem Codeblock enthalten sein. Somit definieren Codeblöcke eine Partitionierung der jeweils zugrunde liegenden Anweisungsfolge.

Generell sollte die Partitionierung einer Anweisungsfolge in Codeblöcke nur in Betracht gezogen werden, wenn diese Aufspaltung auch in der Realität umgesetzt werden würde. Wäre dies aus strukturellen Gründen nicht möglich oder nicht wünschenswert, da die diesbezügliche Programmadaption einen signifikanten Implementierungsaufwand verursachen oder nur sehr schwer wartbaren Code generieren würde, sollte hier auf eine Partitionierung verzichtet werden. Zu beachten ist allerdings, dass dies bereits eine Einschränkung des unter den Kriterien der Performanz und des Implementierungsaufwands orthogonal zu explorierenden Entwurfsraums darstellt. Hingegen sollte die Partitionierung einer Anweisungsfolge in Codeblöcke auch dann erwogen werden, wenn die resultierenden Codeblöcke aufgrund von Abhängigkeiten zwingend sequentiell ausgeführt werden müssen. So zeigt Abbildung 5.17 an einem Beispiel, dass durch die Partitionierung abhängiger Anweisungen die parallele Ausführungsdauer reduziert werden kann. Eine effiziente Strategie, um eine geeignete Granularisierung der Modellierung zu finden, gestaltet sich generell wie folgt: Zunächst wird der Code mit einer relativ geringen Codeblock-Granularität annotiert und mittels der Toolchain evaluiert, welcher Performanzgewinn auf Basis der daraus resultierenden Modelle erzielt werden kann. Ist dieser Leistungszuwachs noch nicht zufriedenstellend, kann sukzessive die Granularität erhöht werden.

Der Codeblock-Graph

Zur Modellierung auf Taskebene und somit zur Bewertung verschiedener Alternativen der Task-Dekomposition unter den zuvor definierten Kriterien wird der sogenannte *Codeblock-Graph* $G = (N_b, E_c, E_i, E_e)$ vorgeschlagen, Abbildung 5.18 zeigt eine Skizze. Ein Codeblock-Graph ist zwingend azyklisch und definiert die folgenden Elemente:

- Gewichtete *Codeblock-Knoten* N_b repräsentieren die Codeblöcke des Programmcodes, welcher der modellierten Task zugrunde liegt. Die Partitionierung einer Task in Codeblöcke erfolgt dabei gemäß den zuvor definierten Bedingungen. Das Gewicht eines Codeblock-Knotens entspricht der Ausführungszeit der durch den Codeblock gekapselten Anweisungen.

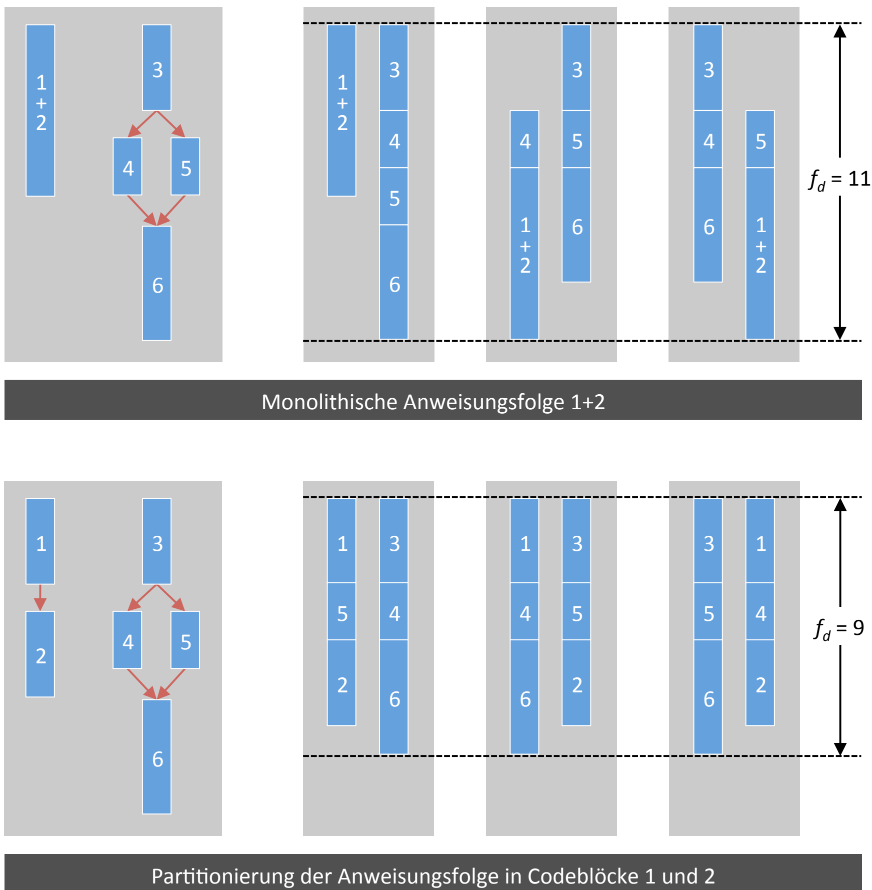


Abbildung 5.17: Konstruktion eines Beispiels, bei dem sich infolge der Aufspaltung einer Anweisungsfolge in voneinander abhängige Codeblöcke die Ausführungsdauer $f_d\ (T_d)$ einer Codeblock-Sequenz von 11 auf 9 reduzieren lässt

- Gerichtete, ungewichtete *Kontrollflusskanten* E_c zeigen in einem ausgeprägten Codeblock-Graphen die Ausführungsreihenfolge der Codeblöcke in der modellierten Task und in einem parametrierten Codeblock-Graphen die Reihenfolge der Codeblock-Ausführung unter einer spezifischen Task-Dekomposition an.
- Gerichtete, gewichtete *Interaktionskanten* E_i repräsentieren Interaktionen in Form gerichteter Datenflüsse zwischen den jeweils inzidenten Codeblöcken. Auf diese Weise werden sowohl der Umfang des Datenaustauschs als auch Datenabhängigkeiten zwischen Codeblöcken im Modell abgebildet.

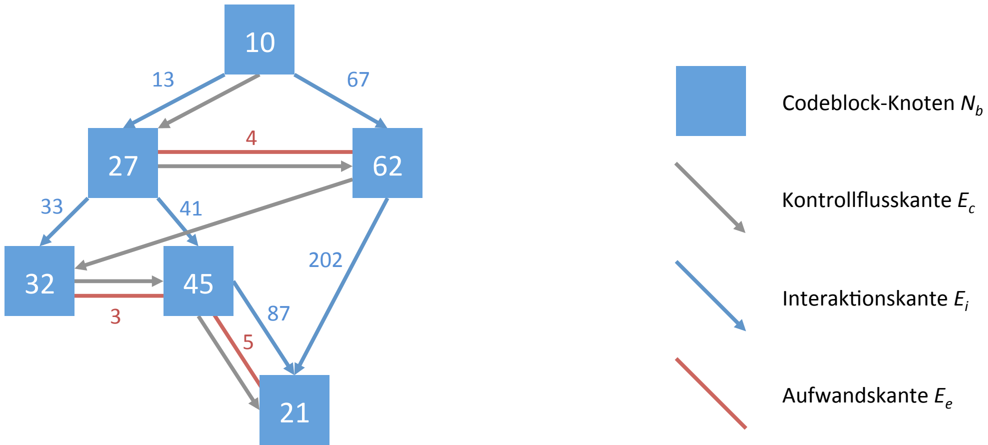


Abbildung 5.18: Visualisierung eines Codeblock-Graphen

- Ungerichtete, gewichtete *Aufwandskanten* E_e modellieren die erforderlichen Implementierungsaufwände, um die sichere echt parallele Ausführung der jeweils inzidenten Codeblöcke im Rahmen einer Task-Dekomposition zu ermöglichen.

In Abschnitt 5.1.2 wurden in allgemeiner Weise die Schritte der Ausprägung und Parametrierung eines Modells im Kontext der hier entwickelten Methode beschrieben; Abbildung 5.19 stellt diese nun spezifisch für den Codeblock-Graphen dar. Im Rahmen der Modellausprägung werden dabei Instanzen von Codeblock-Knoten, Kontrollflusskanten, Interaktionskanten und Aufwandskanten generiert. Sei nun $P(N_b)$ eine Partition der Menge N_b der Codeblock-Knoten mit $|P(N_b)| \leq |C|$. Weiterhin sei $R_i = P_i \times P_i$ eine totale Ordnung auf einer Menge $P_i \in P(N_b)$. Die Parametrierung eines Codeblock-Graphen besteht nun in der Definition eines sogenannten *Schedules* S:

$$S := \left(P\left(N_b\right), R_1, \ldots, R_{|P(N_b)|} \right) \tag{5.18}$$

Dabei muss gelten, dass alle totalen Ordnungen R_i zu den durch die Interaktionskanten E_i vorgegebenen Abhängigkeiten der Ausführungsreihenfolge von Codeblöcken kompatibel sind. Jede totale Ordnung R_i wird schließlich im parametrierten Codeblock-Graphen G durch eine Menge $E_{c_i} \subseteq E_c$ von Kontrollflusskanten realisiert, wobei zwischen je zwei in der totalen Ordnung direkt aufeinanderfolgenden Knoten eine Kontrollflusskante definiert wird. Eine Ausprägung und Parametrierung eines Codeblock-Graphen muss dabei der gemäß der Spezifikation des Basismodells geforderten Zyklenfreiheit bezüglich der Vereinigungsmenge aus Kontrollfluss- und Datenflusskanten genügen. Somit gibt es im ausgeprägten und parametrierten Codeblock-Graphen mindestens einen Codeblock-Knoten $n_b \in N_b$, zu dem im Graphen keine eingehende Kontrollflusskante und keine eingehenden Interaktionskanten definiert sind. Dies hat unter anderem zur Konsequenz, dass Zyklen in der modellierten Anweisungsfolge im Graphen abgerollt werden müssen; nähere Informationen hierzu liefert Abschnitt 5.5.4.

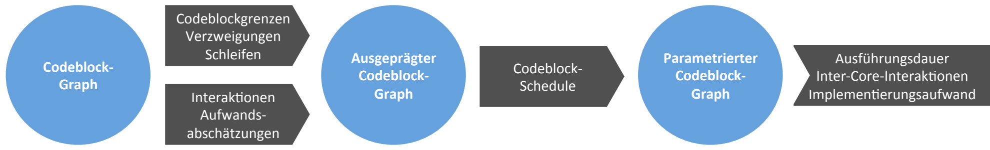


Abbildung 5.19: Prozess der Modellierung und Evaluation auf Taskebene

Die aus einer Dekomposition einer Task t resultierende Menge von Tasks T_d wird schließlich durch eine Agglomeration der Codeblöcke eines durch einen Schedule S parametrierten Codeblock-Graphen gebildet. Eine Codeblock-Agglomeration sei dabei als eine Funktion $f_a : N_b \to T_d$ beschrieben, für die gilt:

$$n_{b_1}, n_{b_2} \in P_i \iff f_a\left(n_{b_1}\right) = f_a\left(n_{b_2}\right) = t_{d_i} \tag{5.19}$$

Zwei Codeblöcke, die Bestandteil der gleichen Menge P_i der Partition $P(N_b)$ eines Schedules S sind, unterliegen somit einer Agglomeration zu einer Task $t_{d_i} \in T_d$. Bei der Implementierung der Task-Dekomposition muss dabei sichergestellt werden, dass die Ausführung der Codeblöcke durch die Tasks $t_d \in T_d$ genau in der durch die Relationen $R_1, \ldots, R_{|P(N_b)|}$ des Schedules S definierten Reihenfolge stattfindet.

Schließlich gilt es, die Bewertung eines ausgeprägten und parametrierten Task-Graphen unter den in Abschnitt 5.5.1 definierten Kriterien zu spezifizieren. Zu diesem Zweck sei zunächst mit $\text{pred}(n_b, S)$ für einen Knoten $n_b \in N_b$ unter einem Schedule S die Menge aller direkten Vorgängerknoten definiert:

$$\text{pred}\left(n_b, S\right) := \left\{ n'_b \in N_b \,\middle|\, \left(n'_b, n_b\right) \in \left(E_i \cup E_c\right) \right\} \tag{5.20}$$

Wie beschrieben, ergeben sich dabei die Kontrollflusskanten E_c direkt aus dem Schedule S. Sei darüber hinaus $r(n_b)$ die für einen Knoten n_b im Rahmen der Ausprägung definierte Ausführungszeit und $f_s(n_b, S)$ der Startzeitpunkt der Ausführung eines Codeblocks $n_b \in N_b$ unter einem Schedule S. Dann gilt für dessen Endzeitpunkt $f_e(n_b, S)$:

$$f_e\left(n_b, S\right) := f_s\left(n_b, S\right) + r\left(n_b\right) \tag{5.21}$$

Der Startzeitpunkt $f_s(n_b, S)$ eines Codeblocks unter einem Schedule S berechne sich nun in rekursiver Weise:

$$f_s\left(n_b, S\right) := \begin{cases} 0, & \text{falls pred}\left(n_b, S\right) = \varnothing \\ \max\limits_{n'_b \in \text{pred}(n_b, S)} f_e\left(n'_b, S\right), & \text{falls pred}\left(n_b, S\right) \neq \varnothing \end{cases} \tag{5.22}$$

Diese Formel berechnet die Länge des sogenannten *kritischen Pfads* $(n_{b_1}, n_{b_2}, \ldots, n_{b_k})$ zu einem Knoten $n_{b_{k+1}} \in N_b$ durch einen gerichteten Graphen. Der kritische Pfad ist dabei definiert als der Pfad von einem Knoten n_{b_1} zu einem Knoten n_{b_k}, bei dem die Summe der Knotengewichte entlang des Pfades maximal ist. Da jeder ausgeprägte und parametrierte Codeblock-Graph per

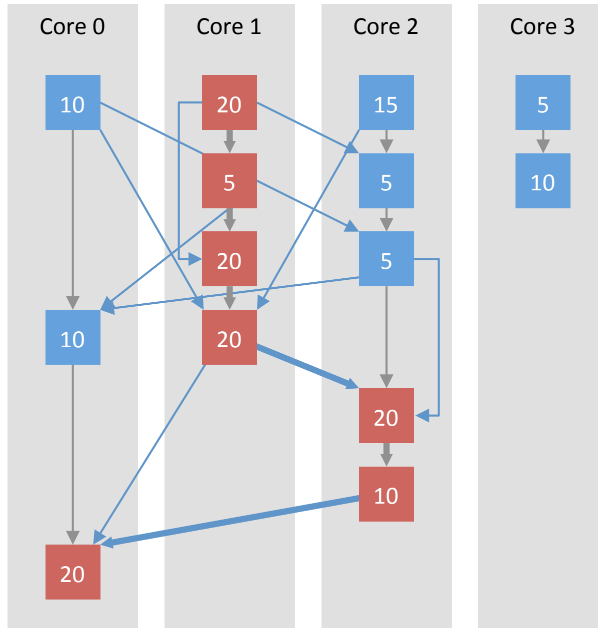


Abbildung 5.20: Darstellung des kritischen Pfades durch einen ausgeprägten Codeblock-Graphen, parametriert durch einen Schedule S für 4 CPU-Kerne. Codeblock-Knoten sind mit Gewichten in Form von Ausführungszeiten annotiert. Knoten entlang des kritischen Pfades sind rot markiert.

Definition azyklisch ist und es somit mindestens einen Knoten $n_b \in N_b$ mit $\text{pred}(n_b, S) = \varnothing$ gibt, ist der kritische Pfad für jeden Knoten $n_b \in N_b$ definiert. Den kritischen Pfad eines ausgeprägten und mit einem Schedule S parametrierten Graphen zeigt Abbildung 5.20. Somit berechnen sich alle Startzeitpunkte $f_s(n_b, S)$ relativ zum Startzeitpunkt 0, zu dem die Ausführung aller Codeblock-Knoten ohne eingehende Kontroll- und Interaktionskanten beginnt.

Zu beachten ist, dass der Startzeitpunkt $f_s(n_b, S)$ eines Knotens $n_b \in N_b$ hier unter einem Schedule S berechnet wird, der im Rahmen der Modellparametrierung definiert wird. Verwendet man hingegen in Formel 5.22 eine Vorgängerfunktion

$$\text{pred'}(n_b) := \left\{ n_b' \in N_b \,\middle|\, \left(n_b', n_b\right) \in E_i \right\}, \tag{5.23}$$

so kann man die Startzeitpunkte $f_s(n_b)$ der Codeblöcke unabhängig von einem Schedule berechnen. In diesem Fall werden für die Berechnung des kritischen Pfades nur Abhängigkeiten in Form von Interaktionskanten berücksichtigt und man erhält als Resultat die Startzeitpunkte der Codeblöcke, falls keine Ressourcenbeschränkung durch eine maximal verfügbare Anzahl an CPU-Kernen besteht. Abbildung 5.21 stellt diesen Zusammenhang dar.

Das wesentliche Ziel einer Modellierung auf Taskebene besteht allerdings darin, Alternativen der Task-Dekomposition für eine vorgegebene Anzahl an CPU-Kernen zu finden und zu bewerten. Unter zusätzlicher Berücksichtigung von Synchronisationen zwischen den Tasks $t_d \in T_d$

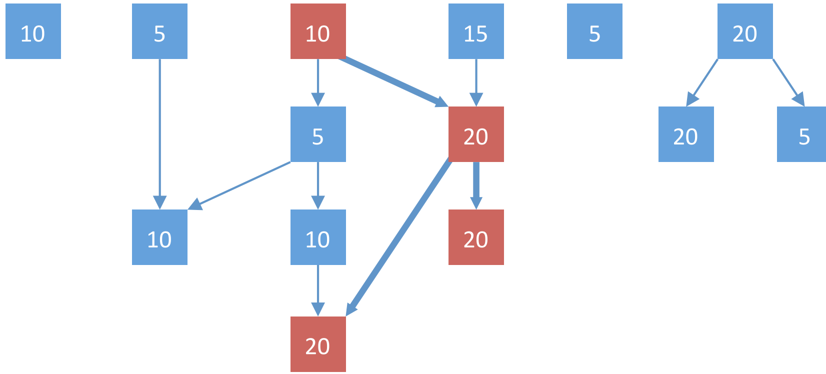


Abbildung 5.21: Darstellung der kritischen Pfade durch einen äquivalent zu Abbildung 5.20 ausgeprägten Codeblock-Graphen ohne Parametrierung durch einen Schedule S. Codeblock-Knoten sind mit Gewichten in Form von Ausführungszeiten annotiert. Knoten entlang des kritischen Pfades sind rot markiert.

aufgrund von Datenabhängigkeiten gilt dann für die Startzeitpunkte $f_s(t_d)$ und die Endzeitpunkte $f_e(t_d)$ der resultierenden Tasks:

$$f_s(t_d) \coloneqq \min_{n_b \in N_b | f_a(n_b) = t_d} \left(f_s(n_b, S) \right) \tag{5.24}$$

$$f_e(t_d) \coloneqq \max_{n_b \in N_b | f_a(n_b) = t_d} \left(f_e(n_b, S) \right) \tag{5.25}$$

Weiterhin gilt:

$$\exists n'_b \in N_b : \text{pred}\left(n'_b, S\right) = \varnothing \overset{(5.22)}{\Longrightarrow} f_s\left(n'_b, S\right) = 0 \Longrightarrow \min_{n_b \in N_b} \left(f_s(n_b, S) \right) = 0 \tag{5.26}$$

Damit ergibt sich für die zur Berechnung des Speedups f_u (vgl. Formel 5.17) erforderliche Ausführungsdauer $f_d(T_d)$ einer Task-Dekomposition eines gemäß einem Schedule S parametrierten Task-Graphen folgender Zusammenhang:

$$\begin{aligned} f_d(T_d) &\coloneqq \max_{t_{d_1} \in T_d} \left(f_e\left(t_{d_1}\right) \right) - \min_{t_{d_2} \in T_d} \left(f_s\left(t_{d_2}\right) \right) \\ &= \max_{t_{d_1} \in T_d} \left(\max_{n_b \in N_b | f_a(n_b) = t_{d_1}} \left(f_e(n_b, S) \right) \right) - \min_{t_{d_2} \in T_d} \left(\min_{n_b \in N_b | f_a(n_b) = t_{d_2}} \left(f_s(n_b, S) \right) \right) \\ &= \max_{n_b \in N_b} \left(f_e(n_b, S) \right) - \min_{n_b \in N_b} \left(f_s(n_b, S) \right) \\ &\overset{(5.26)}{=} \max_{n_b \in N_b} \left(f_e(n_b, S) \right) \end{aligned} \tag{5.27}$$

Die Ausführungszeit einer Task-Dekomposition ist somit als spätester Endzeitpunkt $f_e(n_b, S)$ eines Knotens $n_b \in N_b$ in einem durch einen Schedule S ausgeprägten Codeblock-Graphen berechenbar.

Die Bewertung eines Schedules S hinsichtlich der Inter-Core-Interaktionen und Implementierungsaufwände ergibt sich jeweils in Abhängigkeit von den Gewichten der Interaktions- respektive Aufwandskanten, bei denen den inzidenten Codeblock-Knoten unterschiedliche Tasks $t_d \in T_d$ und somit mittels $f_m\colon T_d \to C$ unterschiedliche CPU-Kerne zugewiesen sind. Eine detaillierte Beschreibung der dabei angewandten Logik würde allerdings zu weit vorausgreifen, so dass diesbezüglich auf Abschnitt 5.5.4 verwiesen sei.

5.5.3 Systeminstrumentierung

Für die Ausprägung eines Codeblock-Graphen sieht die im Rahmen dieser Arbeit entwickelte Methode die Instanziierung von Codeblock-Knoten und Kontrollflusskanten auf Basis von Laufzeitdaten des jeweiligen Systems vor. Interaktionskanten und Aufwandskanten werden hingegen in semiautomatischer Weise unter Einbeziehung entsprechenden Expertenwissens durch mit dem System vertraute Entwickler modelliert.

Die für ein Profiling erforderliche Systeminstrumentierung unterscheidet sich allerdings in einigen wesentlichen Aspekten von der zur Ausprägung von Task-Graphen. Im letzteren Fall hatten alle für die Modellierung relevanten Systemereignisse die Form von Syscalls des Betriebssystems, so dass die entsprechenden Laufzeitdaten durch Instrumentierungen des Betriebssystemcodes aufgezeichnet werden konnten. Anders gestaltet sich die Modellierung auf Taskebene: So sind zur Rekonstruktion der Codeblock-Knoten die exakten Zeitpunkte aufzuzeichnen, zu denen die Ausführung eines Codeblocks beginnt und endet. Da diese Ereignisse nicht mit einem Syscall verbunden sind, ist statt einer zentralen Stelle im Betriebssystem der Code der zu modellierenden Task selbst zu instrumentieren. Sofern dieser, wie in Abschnitt 3.1.2 gefordert, einen überschaubaren Umfang aufweist, ist der im Rahmen einer Instrumentierung zu leistende Aufwand jedoch in der Regel tragbar. Die Instrumentierung erfolgt dabei mittels einer Erweiterung des eigentlichen Quellcodes um zusätzliche Anweisungen, deren Ausführung durch das entsprechend eingesetzte Profiling-Tool aufgezeichnet wird. Im Fall des *Wind River System Viewer* handelt es sich hierbei um die Routine `wvEvent()`, der neben einer Event-ID ein frei definierbarer Speicherbereich mit zusätzlichen Informationen als Parameter übergeben werden kann [188]. Um die Lesbarkeit des eigentlichen Task-Codes durch die Instrumentierung nicht zu stark zu beeinträchtigen, empfiehlt sich die Definition entsprechender *Makros*, die schließlich erst der *C-Präprozessor* durch die eigentlichen Event-Routinen substituiert. Dies hat darüber hinaus den Vorteil, dass bei einem Wechsel des Profiling-Tools nur die entsprechenden Makros zu adaptieren sind. Die für die Modellierung einer Task als Codeblock-Graph mindestens erforderliche Instrumentierung stellt Tabelle 5.4 dar. Deren erste Spalte nennt dabei die Namen der entsprechenden Makros, während die zweite und dritte Spalte die im jeweiligen Fall zu sichernden Parameter auflisten. Zudem sind für eine Rekonstruktion die bereits in Abschnitt 5.4.3 definierten Ereignisse $l_{\uparrow,t}$ und $l_{\downarrow,t}$ zur Aufzeichnung von Zustandswechseln erforderlich und somit ebenfalls in der Tabelle aufgeführt. Da diese Ereignisse entsprechend Abschnitt 5.4.3 mittels einer Betriebssysteminstrumentierung aufgezeichnet werden können, ist für diese kein Makro definiert.

Makro/Ereignis	Parameter		Beschreibung
LOGSTART	-	-	Beginn der Modellierung
LOGEND	-	-	Ende der Modellierung
CODEBLOCKSTART	Bezeichner	Zeitstempel	Beginn eines Codeblocks
CODEBLOCKEND	Bezeichner	Zeitstempel	Ende eines Codeblocks
LOOPSTART	Bezeichner	Zählvariable	Beginn des Rumpfes einer *For*- oder *While*-Schleife
LOOPTAILSTART	Bezeichner	Zählvariable	Beginn des Rumpfes einer *Do-While*-Schleife
LOOPEND	Bezeichner	-	Ende des Rumpfes einer *For-*, *While*- oder *Do-While*-Schleife
BRANCHSTART	Bezeichner	-	Beginn einer Alternative einer *n*-fachen Verzweigung
BRANCHEND	Bezeichner	-	Ende einer Alternative einer *n*-fachen Verzweigung
$I_{\uparrow,t}$	CPU-Kern	Zeitstempel	Zustandswechsel einer Task *t* zwischen einem beliebigen Zustand und *Running*; Betreten einer Interrupt Service Routine
$I_{\downarrow,t}$	CPU-Kern	Zeitstempel	Zustandswechsel einer Task *t* zwischen *Running* und einem beliebigen Zustand; Verlassen einer Interrupt Service Routine

Tabelle 5.4: Minimalumfang der für die Modellierung eines Codeblock-Graphen erforderlichen Instrumentierung

Die nachfolgend beschriebenen Schritte der Instrumentierung einer Task lassen sich in Abbildung 5.22 an einem Beispiel nachvollziehen. Dabei wird zunächst der Beginn und das Ende des zur Modellierung ausgewählten Codes mit einem LOGSTART- respektive einem LOGEND-Makro definiert. Nun erfolgt eine Partitionierung des Quellcodes unter Berücksichtigung der im vorherigen Abschnitt beschriebenen Bedingungen und Empfehlungen. Jeder Codeblock wird dabei durch ein CODEBLOCKSTART- und ein CODEBLOCKEND-Makro markiert. Beiden ist als Parameter ein identischer, eindeutiger Codeblock-Bezeichner zur späteren Zuordnung des Codeblock-Knotens im resultierenden Graphen zu übergeben. Da eine Kaskadierung von Codeblöcken nicht erlaubt ist, könnte beim CODEBLOCKEND-Makro prinzipiell auf den Bezeichner verzichtet werden. Die Erfahrung bei der Instrumentierung von Programmen hat jedoch gezeigt, dass gerade bei komplexeren Programmen mit langen Codeblöcken die Übersichtlichkeit und Adaptierbarkeit signifikant erhöht wird, wenn direkt ersichtlich ist, welchem Codeblock ein CODEBLOCKEND-Makro zuzuordnen ist.

Wie in Abschnitt 5.5.2 definiert, kann eine Schleife im Programm entweder vollständig durch einen Codeblock gekapselt sein oder für den Schleifenrumpf ist mindestens ein Codeblock zu definieren. Im letzteren Fall ist eine weitere Codeinstrumentierung in Form einer Kapselung des kompletten Schleifenrumpfes entsprechend dem Typ der Schleife durch LOOPSTART-/ LOOPTAILSTART- und LOOPEND-Makros vorzunehmen. Zu beachten ist, dass beide Makros

Abbildung 5.22: Darstellung einer Codeinstrumentierung am Beispiel der in Abbildung 5.16 definierten Codeblock-Partitionierung

entsprechend der Darstellung in Abbildung 5.22 innerhalb des Schleifenrumpfes einzufügen sind, damit sie bei jedem Schleifendurchlauf aufgerufen werden. Aus diesem Grund ist sicherzustellen, dass bei einem bedingten vorzeitigen Verlassen einer Iteration, beispielsweise mittels einer `break`- oder `continue`-Anweisung, zunächst das `LOOPEND`-Makro aufgerufen wird. Ist die Schleife über eine Zählvariable gesteuert, so kann diese als zweiter Parameter übergeben werden. Andernfalls ist eine andere, zur eindeutigen Identifikation der einzelnen Iterationen geeignete Variable zu definieren und beim Aufruf des `LOOPSTART`-/`LOOPTAILSTART`-Makros zu übergeben. Die Kaskadierung der Schleifeninstrumentierung ist explizit zulässig, um mehrfach geschachtelte Schleifen korrekt instrumentieren zu können.

Weiterhin sieht das Codeblock-Konzept entweder die vollständige Kapselung von Verzweigungen oder die Definition von mindestens einem Codeblock pro Alternative der Verzweigung

vor. Im letzteren Fall ist zusätzlich jede Alternative einer derartigen Verzweigung entsprechend Abbildung 5.22 vollständig durch ein `BRANCHSTART`- und `BRANCHEND`-Makro zu kapseln. Zu beachten ist, dass die Makros aller von der gleichen Kondition abhängigen Alternativen mit einem identischen Bezeichner parametriert werden. Auch eine Kaskadierung dieser Makros ist zulässig, um mehrfach geschachtelte Verzweigungen zu instrumentieren. Weiterhin ist die Instrumentierung von Schleifen innerhalb von Verzweigungen und die Instrumentierung von Verzweigungen innerhalb von Schleifen bis zu einer beliebigen Kaskadierungstiefe erlaubt.

5.5.4 Modellausprägung

Auf Basis eines Profilings einer Firmware mit einer instrumentierten Task kann nun ein Codeblock-Graph ausgeprägt werden, wobei jedoch vorausgesetzt wird, dass zur Laufzeit keine Migration der instrumentierten Task auf einen anderen CPU-Kern stattfand. Ein textbasiertes Log im *CSV*-Format ist in der derzeitigen Implementierung der *EEEPA*-Toolchain für diesen Vorgang ausreichend, sofern es die in Tabelle 5.4 definierten Daten beinhaltet. Der wesentliche Unterschied gegenüber der Modellierung auf Systemebene besteht darin, dass aus einem *CSV*-Log pro Ausführungszyklus der instrumentierten Task und somit pro Paar von `LOGSTART`- und `LOGEND`-Events ein separater Graph G_i generiert wird, der die Laufzeitdynamik der instrumentierten Task in diesem Zyklus repräsentiert.

Modellierung der Codeblöcke

Aus den Laufzeit-Logs werden nun in einem ersten Schritt alle für die Modellierung der Codeblöcke relevanten Daten extrahiert. Dabei wird für jedes Paar von `CODEBLOCKSTART`- und `CODEBLOCKEND`-Events ein neuer Codeblock-Knoten instanziiert, dessen Name dem entsprechenden Parameter der Logging-Events entspricht. Als Knotengewicht wird gemäß der Modellspezifikation die Ausführungszeit des dem Codeblock zugrunde liegenden Codes gewählt, die sich mittels der Zeitstempel der jeweiligen `CODEBLOCKSTART`- und `CODEBLOCKEND`-Events berechnen lässt. Wurde die Ausführung eines Codeblocks durch einen Task-Wechsel oder einen Interrupt unterbrochen, so lässt sich dies mittels der in Tabelle 5.4 definierten Logging-Events $l_{\uparrow,t}$ und $l_{\downarrow,t}$ rekonstruieren und bei der Berechnung berücksichtigen. Generell sind nur die Codeblöcke Bestandteil eines Codeblock-Graphen, die zwischen den entsprechenden `LOGSTART`- und `LOGEND`-Events durchlaufen wurden. So wird beispielsweise im Fall von Konditionen, bei denen für die bedingt auszuführenden Anweisungsfolgen separate Codeblöcke definiert sind, je nach Auswertung der Kondition bei einer konkreten Ausführung nur eine Teilmenge der annotierten Codeblöcke durchlaufen, so dass auch nur diese im entsprechenden Graphen G_i modelliert wird.

Eine Besonderheit bei der Rekonstruktion stellt die Forderung dar, dass ein Codeblock-Graph gemäß der Definition des Basismodells azyklisch ist. In diesem Zusammenhang ist die Eigenschaft, dass jeder beliebige Kontrollflusspfad durch den Graphen einen Codeblock maximal einmal beinhaltet, eine notwendige, wenn auch nicht hinreichende Bedingung. Diese Invariante kann jedoch zunächst nicht vorausgesetzt werden, da Codeblöcke innerhalb einer oder mehrerer geschachtelter Schleifen definiert sein können und somit bei jeder Schleifeniteration ausgeführt werden. Um trotzdem einen azyklischen Graphen zu erhalten, muss für jeden in der Programmausführung mehrfach durchlaufenen Codeblock pro Ausführung eine eige-

ne Instanz im Codeblock-Graphen generiert werden. Dabei müssen die Knotenbezeichner im Graphen zum Zweck der späteren Zuordnung immer eindeutig sein. Da Sprungbefehle im instrumentierten Code per Definition ebenso ausgeschlossen sind wie mit Codeblöcken instrumentierte Funktionen, die aus dem instrumentierten Code heraus potentiell mehrfach aufgerufen werden können, stellen Schleifen die einzige Ursache für die multiple Ausführung einer als Codeblock definierten Anweisungsfolge dar. Wird rekonstruiert, dass ein Codeblock Bestandteil einer oder gegebenenfalls mehrerer Schleifen ist, so wird deshalb für jede Instanz des Codeblocks der Name um ein sogenanntes *Iterator-Suffix* erweitert. Dieses beginnt stets mit der Zeichenfolge `it`, gefolgt von einer durch Punkte separierten Liste numerischer Werte. Jeder dieser Werte repräsentiert den Iterationsindex einer der umgebenden Schleifen und identifiziert somit innerhalb einer solchen eine Iteration in eindeutiger Weise. So bezeichnet beispielsweise `CodeblockX#it2.4` die Ausführung eines Codeblocks `CodeblockX` in der 4. Iteration einer inneren und der 2. Iteration einer äußeren Schleife. Die Rekonstruktion der Werte erfolgt dabei aus den Argumenten aller `LOOPSTART`-/`LOOPTAILSTART`-Events, die im chronologischen Log zum Zeitpunkt des `CODEBLOCKSTART`-Events des entsprechenden Codeblocks noch nicht durch ein `LOOPEND`-Event geschlossen wurden. Als Konsequenz ist bei der Auswertung eines parametrierten Graphen eine eindeutige Zuordnung von Codeblöcken zu einzelnen Schleifeniterationen möglich.

Abschließend gilt es noch, Vorbereitungen zur semiautomatischen Modellierung der Interaktionen im Graphen zu treffen. Eine Kontrollabhängigkeit liegt generell genau dann vor, wenn die Ausführung einer Anweisung vom Ergebnis der Auswertung einer anderen Anweisung abhängig ist [159]. Dieser Zusammenhang gilt analog auch für Anweisungsfolgen in Form von Codeblöcken. So ist beispielsweise die Ausführung aller Codeblöcke einer Verzweigungsalternative von der Auswertung der Verzweigungskondition abhängig. Das gilt ebenso für *While-* und *For*-Schleifen, bei denen die Ausführung des Schleifenrumpfes erst nach der Auswertung der Schleifenkondition möglich ist. Lediglich bei *Do-While*-Schleifen wird die erste Iteration unabhängig von der Kondition in jedem Fall durchlaufen, alle weiteren Iterationen sind dann wieder kontrollabhängig zur Kondition. In einem letzten Schritt wird deshalb analysiert, in welche Schleifen und Verzweigungen ein Codeblock im Programm eingebettet ist, da dieser Codeblock dann zu deren Konditionen kontrollabhängig ist. Ein Codeblock ist dabei Bestandteil aller Schleifenrümpfe und Verzweigungsalternativen, deren Eintritt im chronologischen Log vor dem Zeitpunkt des jeweiligen `CODEBLOCKSTART`-Events per `LOOPSTART`, `LOOPTAILSTART` oder `BRANCHSTART` signalisiert und noch nicht per `LOOPEND` respektive `BRANCHEND` geschlossen wurde. Die Bezeichner aller Schleifen und Verzweigungen werden dabei zunächst analog zu den Codeblöcken um ein entsprechendes Iterator-Suffix erweitert, das den aktuellen Index aller Schleifen beschreibt, in welche die entsprechende Schleifen- oder Verzweigungskondition eingebettet ist. Diese erweiterten Bezeichner werden schließlich entsprechend Abbildung 5.23 im rekonstruierten Graphen als zusätzliche Informationen zu den Codeblöcken gespeichert. Somit ist die Ausführung aller Codeblöcke, die mit dem gleichen erweiterten Schleifen- oder Verzweigungsbezeichner annotiert sind, von der entsprechenden Schleife oder Verzweigung kontrollabhängig. Die Annotation erfolgt dabei für alle Schleifen und Verzweigungen mit Ausnahme von *Do-While*-Schleifen in identischer Weise. Bei letzteren hingegen werden die Codeblöcke der ersten Schleifeniteration noch nicht annotiert, sondern erst die aller weiteren Iterationen, da die erste Iteration einer *Do-While*-Schleife unabhängig von der Kondition immer ausgeführt wird.

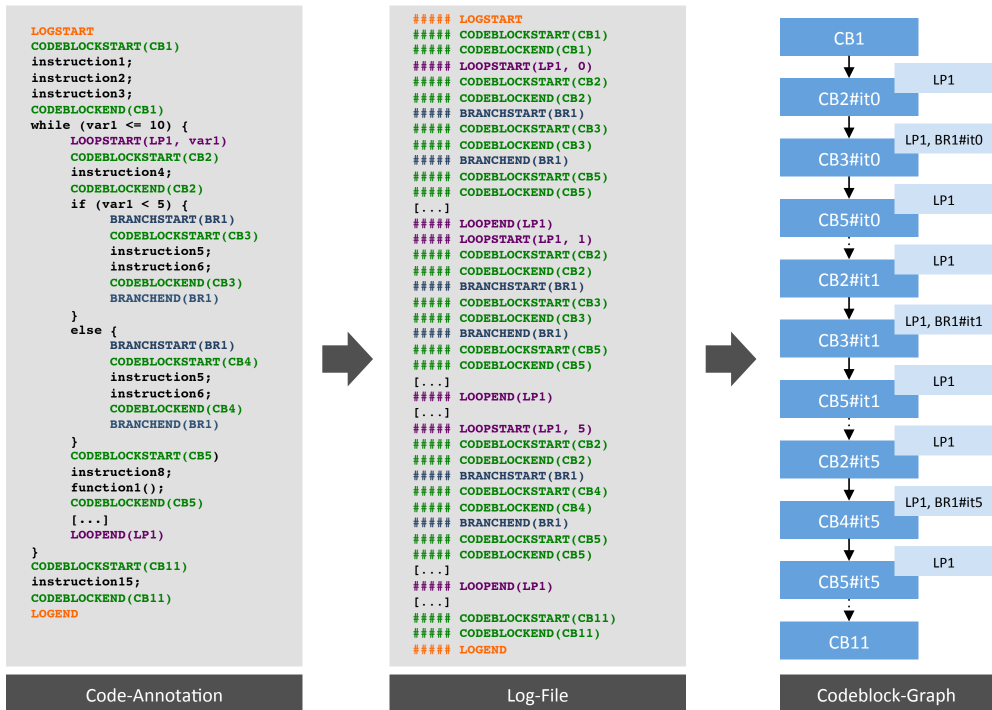


Abbildung 5.23: Vorgehensweise bei der Modellierung von Schleifen und Verzweigungen

Modellierung der Kontrollflüsse

Kontrollflusskanten zeigen im Rahmen der Modellausprägung die derzeit implementierte Ausführungsreihenfolge der Codeblöcke an. Somit wird genau dann eine Kontrollflusskante zwischen zwei Codeblock-Knoten im Graphen modelliert, wenn die aufgezeichneten Daten auf eine direkt aufeinanderfolgende Ausführung der Codeblöcke schließen lassen.

Modellierung der Interaktionen

Da ein Codeblock-Graph eine einzelne Task $t \in T$ einer Multitasking-Firmware modelliert, beschränken sich die Interaktionen zwischen Codeblöcken auf Lese- und Schreibzugriffe der in den Codeblöcken gekapselten Anweisungen auf lokale und globale Variablen. Haben derartige Datenflüsse eine Produzenten-Konsumenten-Semantik, so sind sie im Fall einer Dekomposition von t in eine Menge von Tasks T_d mittels Betriebssystemmechanismen als Inter-Task-Interaktionen zu implementieren, wenn die beteiligten Codeblöcke unterschiedlichen Tasks $t_d \in T_d$ zugeordnet sind. Dies kann zugleich die in Abschnitt 5.4.1 beschriebenen Arten des Overheads von Inter-Core-Interaktionen verursachen, da alle Tasks T_d gemäß Definition 5.14 unterschiedlichen CPU-Kernen zugeordnet werden. Weiterhin repräsentieren diese Interaktionen Datenabhängigkeiten zwischen Codeblöcken, zu denen ein gültiger Schedule kompatibel sein muss.

Diese Aspekte motivieren schließlich die Modellierung von Datenflüssen mit Produzenten-Konsumenten-Semantik im Codeblock-Graphen, wobei jedoch nur die probleminhärenten Datenflüsse abgebildet werden sollten, die auch durch eine Optimierung der konkreten Implementierung nicht eliminiert werden können [122].

Da Datenflüsse in Form von lesenden und schreibenden Speicherzugriffen zwischen einzelnen Anweisungen einer Task nicht mit Syscalls verbunden sind, ist deren Profiling mittels einer Instrumentierung des Betriebssystems nicht möglich, so dass für die Rekonstruktion eine umfassende Instrumentierung des Quellcodes erforderlich wäre. Ohnehin wäre aber eine auf der konkreten Implementierung einer Task basierende Bottom-Up-Analyse nur von begrenztem Nutzen, da alle dabei modellierten Datenflüsse zunächst in manueller Weise hinsichtlich ihrer Probleminhärenz bewertet werden müssten. Als Konsequenz erfolgt die Ausprägung eines Codeblock-Graphen mit Interaktionen in semiautomatischer Weise gemäß einem sich stärker auf die Spezifikation als auf die Implementierung fokussierenden Top-Down-Ansatz. So werden gerichtete, probleminhärente Datenflüsse zwischen Codeblöcken durch mit der Firmware vertraute Entwickler manuell formalisiert und unter Einbeziehung automatisch extrahierter Profiling-Informationen im Codeblock-Graphen in geeigneter Weise modelliert. Wie zuvor beschrieben, liefert die Modellierung auf Taskebene pro Ausführungszyklus des instrumentierten Codes einen separaten Graphen G_i. Bedingt ausgeführte Schreib- und Leseoperationen können allerdings dazu führen, dass die Interaktionen zwischen Codeblöcken zyklenspezifisch sind, so dass diese prinzipiell für jeden Graphen G_i separat modelliert werden müssten. Allerdings besteht das Ziel der hier beschriebenen Methode darin, auf Basis des Firmware-Verhaltens unter ausgewählten Lastprofilen eine statische Task-Dekomposition zum Entwicklungszeitpunkt zu definieren, die somit zu allen unter diesen Lastprofilen auftretenden Abhängigkeiten kompatibel sein muss. Als Konsequenz muss jeder potentielle Schedule S ohnehin dahingehend evaluiert werden, ob er zu der Vereinigungsmenge aller in den modellierten Graphen $\mathcal{G}$ definierten Interaktionskanten kompatibel ist (vgl. Abschnitt 5.6.3), so dass auf eine für einen konkreten Ausführungszyklus des Codes spezifische Modellierung der Abhängigkeiten verzichtet werden kann. Soll darüber hinaus sichergestellt werden, dass die Dekomposition auch für beim Profiling nicht berücksichtigte Lastprofile Gültigkeit besitzt, so müssen sogar alle in der Implementierung definierten Datenflüsse modelliert werden, sofern diese probleminhärent sind. In beiden Fällen ist allerdings zu beachten, dass ein Verzicht auf eine zyklenspezifische Modellierung der Datenflüsse zwar die Modellierungsaufwände reduziert, aber zugleich die Abstraktionsebene der Modellierung erhöht. Die Konsequenzen diskutiert Abschnitt 5.5.5.

Um die Korrektheit der explorierten Schedules zu gewährleisten, müssen alle bei einer Dekomposition zu berücksichtigenden Abhängigkeiten zwischen Codeblöcken modelliert werden. Zu diesen zählen neben den Daten- auch die Kontroll- und Namensabhängigkeiten [74]. Da letztere aufgrund ihrer fehlenden Probleminhärenz allerdings unberücksichtigt bleiben können, steht noch die geeignete Modellierung von Kontrollabhängigkeiten aus. Liegt der durch eine Verzweigungs- oder Schleifenkondition ausgewertete Wert noch nicht zu Beginn des modellierten Codes vor, so wird dieser durch einen Codeblock $n_{b_1} \in N_b$ des Graphen berechnet und als Konsequenz ist die Kondition zu diesem Codeblock datenabhängig. Ist nun die komplette Verzweigung respektive die komplette Schleife inklusive der Kondition durch einen Codeblock $n_{b_2} \in N_b$ gekapselt, so handelt es sich hierbei um eine einfache Datenabhängigkeit zwischen den Codeblöcken n_{b_1} und n_{b_2}. Sind hingegen die Verzweigungsalternativen respektive der Schleifenrumpf in von der Kondition kontrollabhängige Codeblöcke $N'_b \subseteq N_b \backslash \{n_{b_1}\}$ parti-

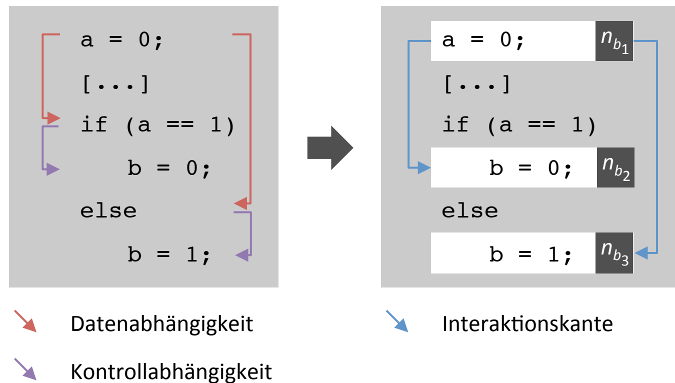
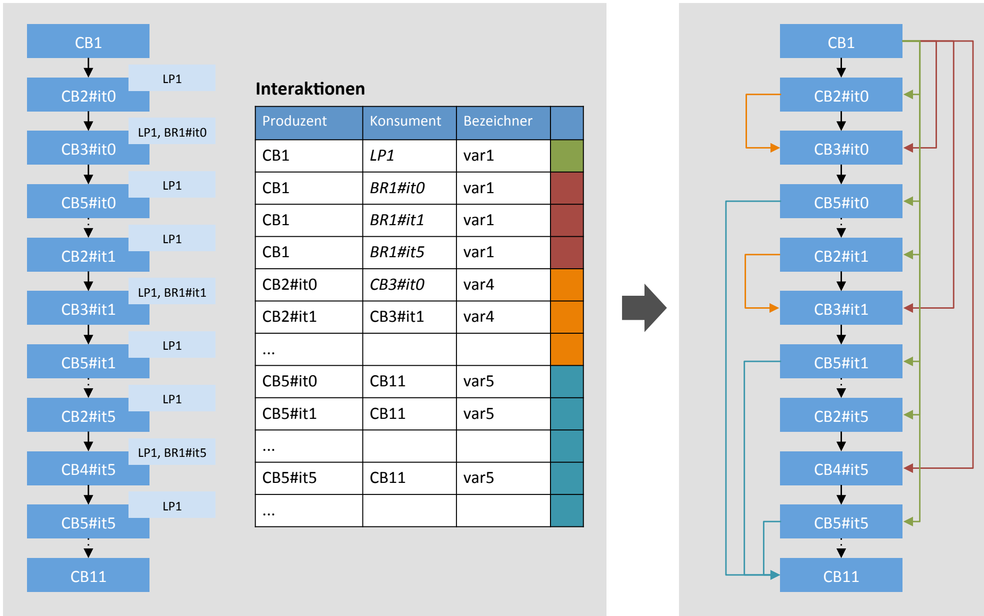


Abbildung 5.24: Transformation von Kontrollabhängigkeiten und Datenabhängigkeiten: Die Datenabhängigkeit der Kondition von der Anweisung a = 0 und die Kontrollabhängigkeiten der bedingten Anweisungen von der Kondition werden als gerichtete Interaktionen vom Codeblock n_{b_1} zu den Codeblöcken n_{b_2} und n_{b_3} modelliert.

tioniert, so ist die Kondition selbst entsprechend Abschnitt 5.5.2 nicht Bestandteil des Graphen und die Abhängigkeiten müssen in geeigneter Weise modelliert werden. Zu diesem Zweck wird im Graphen eine gerichtete Interaktionskante zwischen dem Codeblock n_{b_1} und jedem zu der Kondition kontrollabhängigen Codeblock $n_{b_i} \in N'_b$ modelliert. Dabei wird die Transitivitätseigenschaft der Abhängigkeit genutzt: Diese lässt aus der Datenabhängigkeit einer Kondition von einem Codeblock n_{b_1} und der Kontrollabhängigkeit der Codeblöcke n_{b_i} von der Kondition auf gerichtete Interaktionen vom Codeblock n_{b_1} zu den Codeblöcken n_{b_i} schließen. Abbildung 5.24 skizziert diesen Vorgang.

Aus den in Abschnitt 5.4.4 genannten Gründen wird als Entwicklerschnittstelle zur Formalisierung von Codeblock-Interaktionen erneut ein tabellarisches Format gewählt. Dabei werden in der Tabelle keine Interaktionskanten formalisiert, sondern jede Zeile definiert den Produzenten, den Konsumenten und den Bezeichner einer Interaktion. Der Bezeichner muss die mittels der Interaktion übertragenen Daten innerhalb des Graphen eindeutig identifizieren, so dass sich zu diesem Zweck beispielsweise der Name der entsprechend zum Datenaustausch genutzten Variable anbietet. Da der aus den Laufzeitdaten extrahierte Codeblock-Graph mittels entsprechender Suffixe entlang des aufgezeichneten Kontrollflusses vollständig abgerollt wurde, ist eine korrekte Formalisierung der gerichteten Datenflüsse stets frei von Zyklen, da eine Interaktion mit einer Produzenten-Konsumenten-Semantik stets entlang des Kontrollflusses eines Graphen ausgerichtet ist. Bei der Modellierung der Interaktionen wird schließlich folgendes Vorgehen angewandt, das in Abbildung 5.25 dargestellt ist:

- Sei als Produzent einer Interaktion der Bezeichner eines Codeblocks n_{b_p} und als Konsument der Bezeichner eines Codeblocks n_{b_k} formalisiert. Dann wird eine einzelne gerichtete Interaktion zwischen n_{b_p} und n_{b_k} modelliert.



Produzent	Konsument	Bezeichner
CB1	*LP1*	var1
CB1	*BR1#it0*	var1
CB1	*BR1#it1*	var1
CB1	*BR1#it5*	var1
CB2#it0	*CB3#it0*	var4
CB2#it1	CB3#it1	var4
...		
CB5#it0	CB11	var5
CB5#it1	CB11	var5
...		
CB5#it5	CB11	var5
...		

Abbildung 5.25: Formalisierung und Modellierung von Interaktionskanten, dargestellt an dem bereits in den Abbildungen 5.16, 5.22 und 5.23 gewählten Beispiel

- Sei als Produzent einer Interaktion der Bezeichner eines Codeblocks n_{b_p} und als Konsument der Bezeichner einer Verzweigung oder einer Schleife formalisiert. Weiterhin sei $N_{b_k} \subseteq N_b$ die Menge der Codeblöcke, die mit dem entsprechenden Verzweigungs- oder Schleifenbezeichner annotiert wurden. Dann wird je eine gerichtete Interaktion zwischen n_{b_p} und jedem Codeblock $n_{b_k} \in N_{b_k}$ modelliert. Dies ergibt sich aus der zuvor beschriebenen Transitivität der Abhängigkeit und der zuvor beschriebenen Annotation eines Codeblocks mit den Bezeichnern aller Verzweigungen und Schleifen, zu denen er kontrollabhängig ist.

Zu berücksichtigen sind bei der Formalisierung die Erweiterungen der Bezeichner um Iterator-Suffixe, die eine feingranulare Formalisierung der Interaktionen ermöglichen. Für Graphen geringer Komplexität kann diese Art der Formalisierung als ausreichend gelten. Weist der Graph jedoch zahlreiche geschachtelte Schleifen auf, kann eine mächtigere Syntax sinnvoll sein, um die Formalisierung kompakt zu halten und trotzdem komplexe Interaktionsstrukturen modellieren zu können. Hier empfiehlt sich beispielsweise die Verwendung regulärer Ausdrücke [57] zur Definition der Produzenten- und Konsumentenbezeichner.

Sei nun I die Menge aller Interaktionen, die in der zuvor beschriebenen Weise aus den Formalisierungen rekonstruiert wurden. Weiterhin seien die Abbildungen $f_q: I \rightarrow N_b$ und $f_z: I \rightarrow N_b$ definiert, die einer Interaktion den Quell- respektive Ziel-Codeblock zuordnen. Im folgenden Schritt der Modellierung werden nun die Interaktionskanten E_i des Codeblock-Graphen generiert, wobei pro Paar von Codeblock-Knoten n_{b_p}, n_{b_k} maximal eine Interaktionskante

$e = (n_{b_p}, n_{b_k})$ hinzugefügt wird. Wie auch beim Task-Graphen speichert diese Kante dann mittels der in Abschnitt 5.3.3 beschriebenen Erweiterung des *GraphML*-Schemas eine vollständige Auflistung aller folgendermaßen definierten Interaktionen $I_e \subseteq I$:

$$I_e := \left\{ i \in I \middle| f_q(i) = n_{b_p} \wedge f_z(i) = n_{b_k} \right\} \quad \text{mit } e = \left(n_{b_p}, n_{b_k} \right) \tag{5.28}$$

Die Spezifikation des Codeblock-Graphen erfordert zudem die Reduktion jeder Interaktionskante $e \in E_i$ auf ein skalares Kantengewicht $f_g\colon E_i \to \mathbb{N}$, welches wie folgt definiert sei:

$$f_g(e) := |I_e| \tag{5.29}$$

Die Menge der Inter-Core-Interaktionen $I_{T_d} \subseteq I$ eines Codeblock-Graphen unter einer Task-Dekomposition T_d definiere sich nun unter Berücksichtigung der Codeblock-Agglomeration $f_a\colon N_b \to T_d$ (vgl. Formel 5.19) wie folgt:

$$I_{T_d} := \left\{ i \in I \middle| f_a\left(f_q(i)\right) \neq f_a\left(f_z(i)\right) \right\} \tag{5.30}$$

Somit umfasst I_{T_d} alle Interaktionen des Graphen, die unter einer Task-Dekomposition T_d zwischen unterschiedlichen Tasks $t_{d_1}, t_{d_2} \in T_d$ und somit über die Grenzen der CPU-Kerne hinweg abgewickelt werden. Allerdings kann für einen durch einen Codeblock n_b produzierten Wert, der mittels einer Task-Interaktion an eine Task t_d übertragen wurde, eine lokale Speicherung durch t_d und erst bei der Änderung des Wertes durch einen weiteren Codeblock eine erneute Übertragung unterstellt werden. Dies berücksichtigt die Menge der effektiven Inter-Core-Interaktionen $\tilde{I}_{T_d} \subseteq I_{T_d}$ einer Task-Dekomposition T_d. Es sei in diesem Zusammenhang die Funktion $f_n(i)$ definiert, die zu einer Interaktion $i \in I$ den Bezeichner liefert. Damit definiere sich $\tilde{I}_{T_d} \subseteq I_{T_d}$ wie folgt:

$$\begin{aligned} i_1, i_2 \in \tilde{I}_{T_d} \iff & i_1, i_2 \in I_{T_d} \wedge \\ & \left(f_q(i_1) \neq f_q(i_2) \vee f_a\left(f_z(i_1)\right) \neq f_a\left(f_z(i_2)\right) \vee f_n(i_1) \neq f_n(i_2) \right) \end{aligned} \tag{5.31}$$

Somit werden mehrfache Interaktionen mit gleichem Bezeichner zwischen einem Codeblock $n_{b_p} \in N_b$ und den Codeblöcken $N_{b_k} \subseteq N_b$ einer Task $t_d \in T_d$ in der Menge $\tilde{I}_{T_d}$ nur einmal berücksichtigt. Auf Basis der Menge der effektiven Inter-Core-Interaktionen $\tilde{I}_{T_d}$ berechne sich schließlich die geforderte relative Bewertung $f_i\colon T_d \to [0,1]$ der Inter-Core-Interaktionen einer Task-Dekomposition T_d wie folgt:

$$f_i(T_d) := \frac{\left|\tilde{I}_{T_d}\right|}{|I|} \tag{5.32}$$

Modellierung der Aufwände

Aufwände zur Implementierung einer Task-Dekomposition entstehen im Wesentlichen durch die Restrukturierung der Codeblöcke zu Tasks T_d und die Implementierung der für deren synchronisierte Ausführung erforderlichen Mechanismen. Restrukturierungsaufwände werden allerdings im Folgenden nicht berücksichtigt, da diese als fixe Aufwände betrachtet werden, die

in einem von der jeweiligen Dekomposition T_d nahezu unabhängigen Umfang in jedem Fall zu leisten sind. Für die infolge von Synchronisationen anfallenden Aufwände empfiehlt sich hingegen eine differenziertere Betrachtung.

Eine unilaterale Synchronisation der Ausführung zweier Codeblöcke ist nach Abschnitt 2.5.2 genau dann erforderlich, wenn eine spezifische Ausführungsreihenfolge aufgrund kausaler Abhängigkeiten, beispielsweise infolge von Datenflüssen, eingehalten werden muss. Während dies bei einer sequentiellen Ausführung meist in impliziter Weise durch die Anordnung der Anweisungen im Code gewährleistet wird, ist die unilaterale Synchronisation bei einer Task-Dekomposition in expliziter Weise mittels entsprechender Mechanismen der Task-Interaktion zu realisieren. Signifikante Aufwände entstehen zudem, wenn die zuverlässige echt parallele Ausführbarkeit der resultierenden Tasks T_d hinsichtlich des Zugriffs auf gemeinsam genutzte Ressourcen und Routinen sichergestellt werden muss. Für diesen Zweck ist die multilaterale Synchronisation des Ressourcenzugriffs respektive die Implementierung wiedereintrittsfähiger Routinen erforderlich[11]. Dies ist in der Regel für eine Ressource oder Routine nicht erfolgt, wenn auf diese bislang durch die zu modellierende Task exklusiv zugegriffen wurde. In diesem Fall waren derartige Maßnahmen nicht erforderlich, da der wechselseitige Ausschluss in impliziter Weise durch die sequentielle Ausführung des entsprechenden Codes gewährleistet wurde.

Die semiautomatische Modellierung der zuvor genannten Aufwände basiert wie auch im Fall von Task-Graphen auf einer durch Entwickler in Tabellen vorgenommenen Formalisierung. Dabei wird entsprechend dem in Abschnitt 5.4.4 spezifizierten Vorgehen die Modellierung der Aufwände sowohl für unilaterale als auch für multilaterale Synchronisationen unterstützt.

Abschließend gilt es, die Bewertung eines durch einen Schedule S parametrierten Codeblock-Graphen hinsichtlich der Implementierungsaufwände zu spezifizieren. Die Ausführung zweier Codeblöcke $n_{b_1}, n_{b_2} \in N_b$ muss genau dann in entsprechender Weise synchronisiert werden, wenn unter einem Schedule S die Bedingung $f_a(n_{b_1}) \neq f_a(n_{b_2})$ gilt, da n_{b_1} und n_{b_2} in diesem Fall Bestandteil unterschiedlicher Tasks $t_{d_1}, t_{d_2} \in T_d$ sind. Im Fall $f_a(n_{b_1}) = f_a(n_{b_2}) = t_d$ ist eine Synchronisation zwischen n_{b_1} und n_{b_2} hingegen nicht erforderlich, da diese durch die sequentielle Anordnung der Codeblöcke innerhalb der Task t_d sichergestellt wird. Somit berechnet sich die Bewertung einer Task-Dekomposition bezüglich des Implementierungsaufwands generell als das kumulierte Gewicht aller Aufwandskanten, bei denen die inzidenten Knoten unter S unterschiedlichen Tasks und somit CPU-Kernen zugewiesen sind. Als Ausnahme werden Aufwände für multilaterale Synchronisationen, deren Beschreibung um ein identisches Präfix erweitert wurde, analog zum Vorgehen beim Task-Graphen nur einmalig gewertet.

5.5.5 Validierung

Validierung des Basismodells

Im Kontext der Spezifikation des Basismodells und der Modellausprägung wurde gezeigt, dass ein ausgeprägter und parametrierter Codeblock-Graph als Ergebnis einer analytischen Auswertung die zu Beginn definierten Metriken liefern kann. Gemäß der Spezifikation des Codeblock-Konzepts in Abschnitt 5.5.2 sind dabei allerdings Anweisungen in Form von Verzweigungs- und Schleifenkonditionen nicht Bestandteil der Codeblöcke. Diese bleiben somit bei der Be-

[11] Beide Maßnahmen werden im Folgenden analog zu Abschnitt 5.4.4 vereinfachend unter dem Begriff der multilateralen Synchronisation zusammengefasst.

rechnung des Speedups unberücksichtigt und beeinträchtigen auf diese Weise die Validität des Modells, auf dem die Analyse basiert. Dies ist jedoch vernachlässigbar, sofern die Auswertung der entsprechenden Konditionen in Relation zu den übrigen Anweisungen des modellierten Codes nur eine kurze Zeitdauer beansprucht. Eine weitere Beeinträchtigung der Modellvalidität hat ihre Ursache in dem zur Modellierung durchgeführten dynamischen Profiling, welches das Laufzeitverhalten eines Systems und somit dessen in automatisierter Weise generierte Modelle beeinflusst. Darüber hinausgehende Einschränkungen der Validität der Modellierung können nur die Konsequenz einer mangelnden Korrektheit oder einer Unvollständigkeit der Systeminstrumentierung, Modellausprägung oder Modellparametrierung sein.

Validierung der Instrumentierung und Ausprägung

Zunächst gilt es, die Instrumentierung sowie die Ausprägung der Codeblock-Knoten, Interaktionskanten und Aufwandskanten hinsichtlich möglicher Defizite zu prüfen. Da mit der in Tabelle 5.4 definierten Instrumentierung zu jedem in einem konkreten Lastprofil durchlaufenen Codeblock die korrekte, um Unterbrechungen bereinigte Ausführungsdauer berechnet werden kann, ist die Ausprägung der Lastanteile der Tasks und Interrupts unter Voraussetzung ausreichend exakter Zeitstempel korrekt. Weiterhin wird mittels entsprechender Suffixe die Zyklenfreiheit sichergestellt und es werden unter Voraussetzung einer spezifikationsgemäßen Instrumentierung zu jedem Codeblock die Bezeichner aller Verzweigungen und Schleifen modelliert, welche die bedingte Ausführung dieses Codeblocks beeinflussen. Die Ausprägung der Kontrollflusskanten kann ebenfalls als valide gelten, da sich diese in eindeutiger Weise aus der Ausführungsreihenfolge der Codeblöcke im Log ergibt.

Die Spezifikation der Modellierung von Interaktionen ermöglicht es schließlich, beliebige gerichtete Datenflüsse zwischen zwei Codeblöcken oder einem Codeblock und einem Konditionsbezeichner, der wiederum eine Menge kontrollabhängiger Codeblöcke repräsentiert, zu formalisieren. Im ersten Fall ist die Modellausprägung trivial, da hierbei lediglich eine Konvertierung tabellarisch formalisierter Aufwände in kumulierte Aufwandskanten geleistet wird. Die Korrektheit der Modellierung im zweiten Fall ergibt sich gemäß Abschnitt 5.5.4 aus der Transitivität der Abhängigkeitsrelation. Als Option zur Reduzierung des Modellierungsaufwands wurde der Verzicht auf eine zyklenspezifische Formalisierung der Datenflüsse und die Modellierung aller unter den Lastprofilen oder gemäß der Spezifikation möglichen Datenflüsse genannt. Diese Abstraktion führt potentiell zu einer Überschätzung der tatsächlich in einem Zyklus stattfindenden Datenflüsse und reduziert somit die Genauigkeit der Modellierung in einem Umfang, der generisch nicht abgeschätzt werden kann. Stattdessen muss in Abhängigkeit von der zyklenspezifischen Dynamik der jeweiligen Firmware und der Anzahl der modellierten Graphen in einem konkreten Anwendungsfall entschieden werden, ob der reduzierte Modellierungsaufwand diese Einschränkung rechtfertigt.

Das spezifizierte Vorgehen zur Modellierung von Aufwänden ermöglicht schließlich die Formalisierung von Aufwänden für die Implementierung unilateraler und multilateraler Synchronisationen. Analog zum Task-Graphen ist auch hier die Modellausprägung trivial, da tabellarisch formalisierte Aufwände in kumulierte Aufwandskanten konvertiert werden.

Validierung der Parametrierung

In einem letzten Schritt gilt es zu evaluieren, inwiefern ein durch einen Schedule S parametrierter Codeblock-Graph ein valides Modell einer entsprechend der Parametrierung realisierten Dekomposition einer Task t darstellt. Analog zur Validierung auf Systemebene in Abschnitt 5.4.5 sind auch hier als probleminhärente Ursachen einer potentiell beeinträchtigten Prädiktionsgüte des parametrierten Modells unter anderem das dynamische Scheduling und eine Ausführung der parallelisierten Firmware auf einer abweichenden Hardware-Plattform zu nennen. Diese Aspekte sind für eine Auswahl der folgenden potentiellen Einflussfaktoren verantwortlich:

- Unterscheidet sich die Hardware, auf der das Profiling erfolgte, von der Hardware, für welche die Eigenschaften einer Task-Dekomposition prädiziert werden, können sich die Relationen zwischen den Codeblock-Laufzeiten analog zur Modellierung auf Systemebene verschieben. Aus diesem Grund empfiehlt es sich, auch bei einer Modellierung auf Taskebene das Profiling bereits unter einer Ausführung der jeweiligen Firmware auf der Zielplattform durchzuführen.
- Ebenfalls analog zur Modellierung auf Systemebene werden bei der Rekonstruktion der Codeblöcke vom dynamischen Schedule abhängige Phasen eines möglichen aktiven Wartens bei Synchronisationen deren Ausführungsdauer zugerechnet. Im Rahmen der Parametrierung können die diesbezüglichen Einflüsse eines abweichenden dynamischen Schedules auf der Zielplattform allerdings nicht berücksichtigt werden.
- Im parametrierten Modell wird unterstellt, dass die Ausführung einer Task $t_d \in T_d$ ausschließlich infolge der unilateralen Synchronisation mit anderen Tasks $t'_d \in T_d \backslash \{t_d\}$ suspendiert wird. Dies kann aber in einem realen Multitasking-System nur bedingt vorausgesetzt werden. Stattdessen können Phasen des passiven Wartens bei multilateralen Synchronisationen mit anderen Tasks $\{T \cup T_d\} \backslash \{t, t_d\}$ ebenso eintreten wie durch den Betriebssystem-Scheduler veranlasste Suspendierungen zugunsten anderer Tasks $T \backslash \{t\}$ mit höherer Priorität. Wird aufgrund eines derartigen Ereignisses die Ausführung einer Task $t_d \in T_d$ zur Laufzeit durch den dynamischen Scheduler unterbrochen, so kann dies sowohl die Startzeitpunkte $f_s(t'_d)$ als auch die Endzeitpunkte $f_e(t'_d)$ anderer Tasks $t'_d \in T'_d$ mit $T'_d \subseteq T_d \backslash \{t_d\}$ verändern und somit die Ausführungsdauer $f_d(T_d)$ beeinflussen.
- Im Rahmen der Modellparametrierung wird nicht berücksichtigt, mittels welcher Interaktionsmechanismen die unilateralen Synchronisationen in der späteren Implementierung der Dekomposition realisiert werden. Deren Kenntnis wäre jedoch neben den dann noch zu formalisierenden Datenvolumina erforderlich, um zum Zweck einer höheren Prädiktionsgüte eine genauere Interaktionsmodellierung mittels einer gewichteten kumulativen Aggregation entsprechend Formel 5.10 vornehmen zu können. Das Wissen über die gewählten Mechanismen würde darüber hinaus die Vorbedingung darstellen, um Latenzen der unilateralen Synchronisationen bei der Berechnung der Ausführungsdauer $f_d(T_d)$ berücksichtigen zu können.
- Die Implementierung multilateraler Synchronisationen für den Zugriff der aus der Dekomposition resultierenden Tasks T_d auf gemeinsame Betriebsmittel generiert weitere, im Rahmen der Modellparametrierung nicht berücksichtigte Interaktionen.

Diese Aspekte führen in Summe dazu, dass die Bewertung eines parametrierten Modells auf Taskebene lediglich eine Approximation der Werte liefern kann, die eine tatsächlich implementierte Dekomposition aufweist. Als Ursache kann die inhärent nicht mögliche Prognose des dynamischen Schedules und die fehlende Berücksichtigung von im Rahmen der Dekomposition noch zu implementierenden Interaktionen gelten. Um zumindest letztere Einflüsse bei der Modellparametrierung berücksichtigen zu können, müsste ein signifikant höherer Aufwand in die Modellierung investiert werden. Somit können die zuvor beschriebenen Methoden der Modellierung insofern als valide angesehen werden, als sie unter der Prämisse eines anwendungsorientierten und somit pragmatischen Vorgehens ausreichend genaue Ergebnisse liefern.

5.6 Entwurfsraumexploration mittels genetischer Algorithmen

5.6.1 Modellübergreifende Vorgehensweise

Für die Exploration des Entwurfsraums paralleler Firmware-Architekturen auf System- und Taskebene hinsichtlich Pareto-optimaler Lösungen empfiehlt sich aus den in Abschnitt 5.1.1 genannten Gründen der Einsatz genetischer Algorithmen. Für die im Rahmen dieser Arbeit entwickelte Methode gilt es nun, die Ausgestaltung eines solchen Algorithmus zu spezifizieren. Dabei wird die Konformität des Konzepts mit der in Abschnitt 5.3.4 beschriebenen *PISA*-Schnittstellenspezifikation gefordert. Dies wiederum hat zur Folge, dass die problemunabhängigen Aspekte des genetischen Algorithmus in Form des Selektors nicht spezifiziert und implementiert werden müssen, sondern als ausführbare Dateien von der entsprechenden Projekt-Homepage [83] geladen werden können. Somit ist nur der Variator des genetischen Algorithmus zu implementieren, dessen Kontroll- und Datenfluss in Abbildung 5.26 dargestellt ist. Die Spezifikation des Variators umfasst folgende problemspezifische Aspekte:

- Vorbereitung der Graph-Modelle $\mathcal{G}$ für die Exploration
- Repräsentation der Genotypen
- Generierung der Initialpopulation
- Operatoren der Rekombination und Mutation
- Ableitung der Phänotypen aus den Genotypen
- Fitnessbewertung der Individuen unter Beachtung von Nebenbedingungen
- Terminierung des Algorithmus und Ausgabe der Lösungen

Die Ausgangssituation des genetischen Algorithmus ist entweder eine Menge von Task- oder von Codeblock-Graphen $\mathcal{G}$, die mittels nachfolgender Maßnahmen zunächst für eine Entwurfsraumexploration vorzubereiten sind:

- Im Fall von Task-Graphen müssen alle Idle-Tasks aus dem Graphen entfernt werden, da diese keine Systemlast darstellen und somit beim Mapping nicht berücksichtigt werden. Im resultierenden Mapping ergeben sich deren Lasten schließlich wieder in impliziter Weise, indem die durch das SMP-Betriebssystem für jeden CPU-Kern instanziierte Idle-Task die jeweils verbleibende Rechenzeit des Kerns konsumiert.

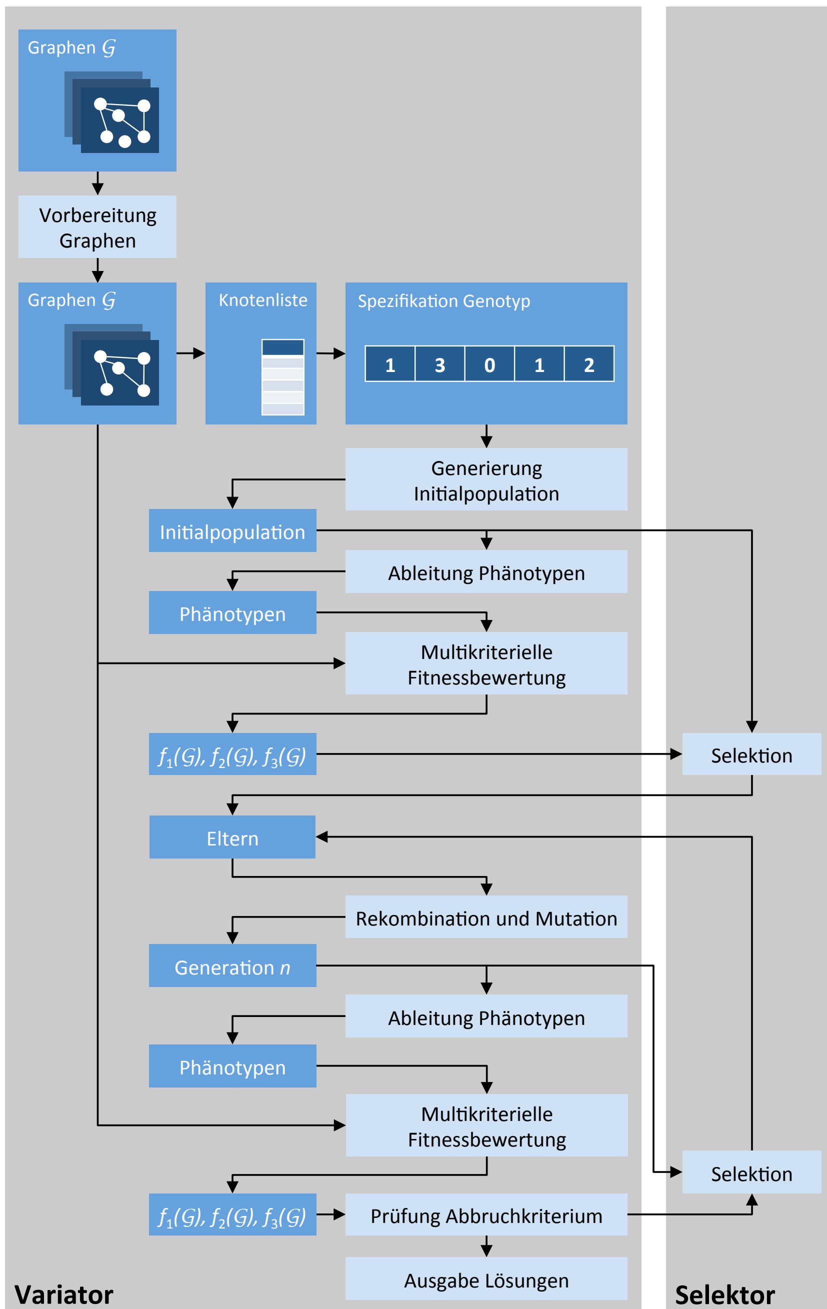


Abbildung 5.26: Detaillierte Darstellung des Kontroll- und Datenflusses beim genetischen Algorithmus (inspiriert durch die Darstellung der *PISA*-Spezifikation in [21])

- Darüber hinaus müssen im Task-Graphen alle Aufwandskanten entfernt werden, bei denen für die inzidenten Tasks beim Profiling bereits eine Ausführung auf unterschiedlichen CPU-Kernen festgestellt wurde. In diesem Fall wird unterstellt, dass die entsprechenden Aufwände zur Sicherstellung der echt parallelen Ausführung schon geleistet wurden und nicht mehr berücksichtigt werden müssen.
- Im Codeblock-Graphen werden alle Kontrollflusskanten entfernt, da diese für das Scheduling irrelevant sind und gemäß dem resultierenden Schedule neu generiert werden.

Im Rahmen der genetischen Optimierung gilt es nun, geeignete Graph-Parametrierungen in automatisierter Weise durch Mittel der genetischen Reproduktion zu generieren. Somit bilden Task-Mappings f_m und Codeblock-Schedules S für Task- respektive Codeblock-Graphen die Phänotypen des hier spezifizierten genetischen Algorithmus. Als Konsequenz muss ein Phänotyp alle Informationen abbilden, die zur eindeutigen Definition eines Mappings respektive Schedules für eine Menge von Graphen $\mathcal{G}$ erforderlich sind. Dies ist notwendig, da eine Task-Verteilung respektive Task-Dekomposition stets bezüglich allen als Graphen $\mathcal{G}$ modellierten Lastprofilen evaluiert werden soll. Gegenstand jedes genetischen Algorithmus sind allerdings nicht die Phänotypen, sondern deren Repräsentation in Form entsprechend kodierter Genotypen. Somit muss der Genotyp bei der Optimierung auf Systemebene das Mapping für die Vereinigungsmenge aller Tasks der Graphen $G_i \in \mathcal{G}$ und bei der Optimierung auf Taskebene den Schedule für die Vereinigungsmenge aller Codeblöcke der Graphen $G_i \in \mathcal{G}$ definieren. Bei einem genetischen Algorithmus kann der Raum der Genotypen allerdings generell folgende Arten von Individuen enthalten [62]:

- *Illegale Individuen*, die sich nicht auf einen Phänotyp abbilden lassen.
- *Ungeeignete Individuen*, deren Phänotyp keine valide Lösung des Problems darstellt.
- *Geeignete Individuen*, die sich auf einen validen Phänotyp abbilden lassen.

Mittels eines geeigneten Genotyps lässt sich in der Regel verhindern, dass illegale Individuen repräsentiert werden. Die Randbedingungen, die eine Lösung hingegen zu einer ungeeigneten Lösung machen, sind häufig komplex und nur auf dem Phänotyp auswertbar, so dass sich ungeeignete Lösungen durch die Wahl der Repräsentation nur selten verhindern lassen. Wird eine Lösung im Rahmen der Bewertung allerdings als ungeeignet klassifiziert, gibt es direkte und indirekte Strategien zum weiteren Vorgehen [47]. Bei der indirekten Strategie wird die Eignung eines Individuums mittels der Fitnessfunktion abgebildet, indem die Fitness ungeeigneter Lösungen mit einer Strafe *(Penalty)* belegt wird. Somit strebt die Population aufgrund des Evolutionsdrucks automatisch in den Bereich geeigneter Lösungen. Bei der direkten Strategie hingegen wird sichergestellt, dass die Population stets ausschließlich geeignete Lösungen enthält, indem ungeeignete Individuen regelmäßig eliminiert oder repariert werden.

Da die Genotyp-Repräsentation von Task-Mappings und Codeblock-Schedules jeweils in spezifischer Weise ausgestaltet ist, werden diese in den Abschnitten 5.6.2 bzw. 5.6.3 für die jeweilige Modellierungsebene beschrieben und in diesem Kontext auch hinsichtlich ihrer Eignung bewertet. Dabei empfiehlt sich generell eine Evaluation hinsichtlich folgender Kriterien [62]:

- *Redundanzfreiheit und Vollständigkeit*: Es existiert eine bijektive Abbildung von der Menge der Genotypen in die Menge der Phänotypen.
- *Legalität*: Jede Genotyp-Permutation entspricht einem Phänotyp.

- *Lamarcksche Eigenschaft*: Die Semantik eines Allels hängt nicht von anderen Genen ab.
- *Kausalität*: Kleine Änderungen des Genotyps im Rahmen der Mutation bedingen kleine Änderungen des Phänotyps.

Zu Beginn des genetischen Algorithmus muss zunächst eine geeignete Initialpopulation generiert werden. Da kein Vorwissen über die Lage der Pareto-Optima im Entwurfsraum vorausgesetzt werden kann, sollte die Initialpopulation möglichst diversitäre Elemente umfassen, wodurch das Risiko einer frühen Konvergenz an lokalen Pareto-Optima reduziert wird. Um eine ausreichende Diversität zu erzielen, wurde deshalb auch hier die üblicherweise angewandte randomisierte Generierung der initialen Individuen gewählt.

Die auf die Genotypen der Population angewandten Operatoren der Rekombination und Mutation unterscheiden sich wiederum bei der Task-Verteilung und Task-Dekomposition und sind deshalb in den Abschnitten 5.6.2 und 5.6.3 in der jeweils spezifischen Form beschrieben. Zuvor wurde erwähnt, dass sowohl die Phänotypen als auch die Genotypen unter Berücksichtigung der Ausprägung aller Graphen $G_i \in \mathcal{G}$ zu definieren sind. Dies wird auch für den Prozess der Rekonstruktion der Phänotypen aus den Genotypen gefordert. Das für die jeweilige Modellierungsebene spezifische Vorgehen definieren ebenfalls die Abschnitte 5.6.2 und 5.6.3.

Im Anschluss an die Ableitung der Phänotypen aus den Genotypen muss deren Fitness bewertet werden, da diese die Entscheidungsgrundlage des Selektors bildet. Dabei werden die problemübergreifenden Phänotypen unter allen Graphen $\mathcal{G}$ hinsichtlich der entsprechenden Metriken evaluiert. Die *PISA*-Spezifikation definiert dabei den für alle Variatoren und Selektoren gültigen Konsens, dass Bewertungskriterien im Rahmen der Optimierung zu minimieren sind [21]. Während die in Abschnitt 5.4.1 definierten Metriken der Lastverteilung diese Forderung erfüllen, muss das in Abschnitt 5.5.1 definierte Kriterium des Speedups bei der Bewertung eines Individuums in reziproker Form einfließen, um dieser Bedingung zu genügen. Die Kriterien der Inter-Core-Interaktionen und des Implementierungsaufwands entsprechen gemäß ihrer Definition wiederum diesem Konsens. Bei der Auswertung der Fitness der Individuen erfolgt zunächst die Bewertung jedes einzelnen Graphen $G_i \in \mathcal{G}$ unter den Kriterien f_1 (Lastverteilung respektive Speedup) und f_2 (Inter-Core-Interaktionen). Das jeweilige Vorgehen ist in Abschnitt 5.6.2 respektive 5.6.3 beschrieben. Allgemein gilt es dabei pro Kriterium f_j, diese Werte $f_j(G_i)$ zu einem Wert $f_j(\mathcal{G})$ zu aggregieren, der die Verteilung oder Dekomposition aller zu Beginn berücksichtigten Graphen $\mathcal{G}$ bewertet. Zu diesem Zweck seien zwei Alternativen in Form des arithmetischen Mittels (Formel 5.33) und des Maximums (Formel 5.34) definiert:

$$f_j(\mathcal{G}) := \frac{\sum_{G_i \in \mathcal{G}} f_j(G_i)}{|\mathcal{G}|} \quad \text{für } j = 1, 2 \tag{5.33}$$

$$f_j(\mathcal{G}) := \max_{G_i \in \mathcal{G}} \left(f_j(G_i) \right) \quad \text{für } j = 1, 2 \tag{5.34}$$

Unter Anwendung welcher dieser Alternativen die Aggregation erfolgt, hängt von der jeweiligen Präferenz und Zielstellung ab. So kann es in manchen Fällen gewünscht sein, ein auf allen Lastprofilen im Durchschnitt gutes Resultat zu erzielen, während in anderen Fällen das Ergebnis im schlechtesten Fall oberste Priorität besitzt.

Bei der Bewertung eines Mappings oder Schedules hinsichtlich der Implementierungsaufwände f_3 werden die Aufwände hingegen nicht separat pro Graph $G_i \in \mathcal{G}$ ermittelt, sondern direkt auf Basis von $\mathcal{G}$. Der Grund dafür ist die Tatsache, dass Implementierungsaufwände nicht

spezifisch für jedes Lastprofil entstehen, sondern ein Parallelisierungsaufwand genau dann zu leisten ist, wenn er unter mindestens einem der berücksichtigten Lastprofile erforderlich ist. Somit wird die Bewertung f_3 auf Basis einer Vereinigung[12] aller Graphen $G_i \in \mathcal{G}$ entsprechend der in Abschnitt 5.4.4 respektive 5.5.4 definierten Weise ermittelt:

$$f_3(\mathcal{G}) \coloneqq f_3\left(\bigcup_{G_i \in \mathcal{G}} G_i\right) \tag{5.35}$$

Dabei ist der Sonderfall eines Aufwands, der zwischen zwei Task- oder Codeblock-Knoten n_1 und n_2 formalisiert wurde, zu berücksichtigen. Dabei gelte, dass die Elemente n_1 und n_2 unter keinem Lastprofil gemeinsam rekonstruiert wurden und es somit keinen Graphen $G_i \in \mathcal{G}$ gibt, der beide Knoten enthält. Als Konsequenz wurde dieser Aufwand in keinem Graphen $G_i \in \mathcal{G}$ modelliert und ist deshalb auch nicht Bestandteil der Vereinigung der Graphen $\mathcal{G}$. Da der Code der Knoten n_1 und n_2 allerdings unter den modellierten Lastprofilen auch nie parallel ausgeführt wird, ist dieser Aufwand im Rahmen einer Parallelisierung für die modellierten Fälle auch nicht zu leisten und die zuvor definierte Bewertung $f_3(\mathcal{G})$ ist somit valide.

Ein weiterer, vom Optimierungsproblem unabhängiger Aspekt der genetischen Optimierung ist die Wahl des Abbruchkriteriums für den Algorithmus. Im einfachen Fall handelt es sich hierbei um eine feste Schranke wie das Erreichen einer definierten Laufzeit des Algorithmus oder die Generierung einer definierten Anzahl n an Generationen. Diese Kriterien führen allerdings unabhängig vom aktuellen Zustand der Population zur Terminierung und Ausgabe der bis dahin gefundenen Pareto-Optima. Es kann jedoch der Fall eintreten, dass der Algorithmus unmittelbar vor dem Abbruch ein neues Genotyp-Schema [79] exploriert hat, welches Potential für neue Pareto-Optima besitzt. Aus diesem Grund wird in Ergänzung zu den zuvor genannten einfachen Schranken als intelligenteres Terminierungskriterium der Zustand definiert, dass innerhalb der letzten n_p Generationen kein neues Pareto-Optimum gefunden wurde. Dies kann als Indiz dafür gelten, dass die Population an einer Front entweder globaler oder lokaler Pareto-Optima des Suchraums konvergiert ist oder aber dass die genetische Diversität der Population zu stark degeneriert ist, um neue Lösungen zu generieren. In jedem Fall motiviert dieser Zustand einen Abbruch des Algorithmus, um keine weiteren Rechenressourcen zu verschwenden. Alternativ zu diesem Terminierungskriterium wird seit einigen Jahren intensiv im Bereich sogenannter *Online Stopping Criteria (OSC)* für multikriterielle genetische Algorithmen geforscht, bei denen die Terminierung auf Basis einer Überwachung komplexerer Fortschrittsindikatoren zur Laufzeit erfolgt. Entsprechende Ansätze werden in [178] vorgestellt, sind jedoch nicht Gegenstand der vorliegenden Arbeit.

Die nach der Terminierung des Algorithmus erforderliche Entscheidung über eine eventuelle Wiederholung der Exploration hängt nun von der Güte der gefundenen Lösungen ab. Generell zeichnet sich ein gutes Resultat einer multikriteriellen Optimierung dadurch aus, dass die explorierten Pareto-Optima eine möglichst geringe Distanz zur globalen Pareto-Front des Suchraums aufweisen, eine gute und somit üblicherweise gleichmäßige Verteilung besitzen und bezüglich jedes Kriteriums einen möglichst breiten Wertebereich abdecken [193]. Aufgrund der probleminhärenten Unkenntnis der globalen Pareto-Front ist eine Bewertung der explo-

[12] Die Vereinigung zweier allgemeiner Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ ist als $G_1 \cup G_2 \coloneqq (V_1 \cup V_2, E_1 \cup E_2)$ definiert. Analog dazu definiere sich auch die Vereinigung von Task- oder Codeblock-Graphen in entsprechend spezifischer Weise.

rierten Lösungen hinsichtlich dieser Kriterien allerdings nicht oder nur eingeschränkt möglich. Somit muss nach der Terminierung des Algorithmus stets spezifisch für einen Anwendungsfall bewertet werden, ob die gefundenen Lösungen eine für diesen ausreichende Güte besitzen.

In einem letzten Schritt erfolgt die Ausgabe der explorierten Pareto-Optima mittels einer Speicherung als entsprechend den Genotypen parametrierte Graphen. Dabei werden nicht nur die Individuen der aktuellen Generation, sondern auch die eines gegebenenfalls mitgeführten Archivs berücksichtigt. Ein Individuum ist allerdings nur dann Bestandteil der Lösungsmenge, wenn es hinsichtlich allen anderen ausgegebenen Lösungen der Anforderung der globalen Pareto-Optimalität (vgl. Abschnitt 2.8) genügt. Des Weiteren wird sichergestellt, dass die Lösungsmenge bezüglich der Phänotypen frei von Duplikaten ist.

5.6.2 Entwurfsraumexploration auf Systemebene

Repräsentation des Genotyps

Die Aufgabe eines Genotyps bei der Exploration auf Systemebene besteht darin, die Parametrierung eines Task-Graphen durch ein Mapping $f_m\colon T \to C$ in geeigneter Weise abzubilden. Zur Kodierung der Indizes der CPU-Kerne in den Genen des Genotyps bietet sich generell die *binäre Kodierung* oder die *Wertekodierung* an [160]. Häufig wird dabei als Implikation des *Schema-Theorems* von Holland [79] die nur zwei Allele nutzende binäre Kodierung bevorzugt [11], wobei es auch dieser Theorie widersprechende Meinungen gibt, die eine Repräsentation mit einer größeren Zahl an Allelen präferieren [9]. Im vorliegenden Fall ist die binäre Kodierung eines Task-Mappings allerdings ungeeignet, da ein binär kodierter Genotyp der Länge $m \cdot |T|$ illegale Core-Zuordnung generieren kann, wenn $|C| = 2^m - 1$ gilt. Aus diesem Grund wurde die Wertekodierung gewählt, um die Zuordnung von Tasks zu CPU-Kernen zu kodieren. Somit definiert der Genotyp auf Systemebene in der Regel $|T|$ Gene zu je $|C|$ Allelen, so dass für jede Task $t \in T$ ein Gen existiert, welches mittels seines den Index eines CPU-Kerns $c \in C$ repräsentierenden Werts deren Mapping festlegt. Generell stehen dem Anwender folgende Mittel zur Verfügung, um im Voraus Anforderungen an die explorierten Lösungen zu formulieren:

- Es können Task-Gruppen $T' \subseteq T$ definiert werden, deren Elemente in jedem durch den Algorithmus explorierten Mapping einem identischen CPU-Kern zuzuordnen sind. Die Motivation zur Definition dieser Gruppen können beispielsweise Tasks darstellen, deren parallele Ausführung durch einen Entwickler als nicht möglich oder als nicht wünschenswert eingeschätzt wird. Im Genotyp bildet sich eine entsprechende Einschränkung insofern ab, als pro Task-Gruppe lediglich ein Gen definiert wird, dessen Wert die Core-Zuordnung der kompletten Gruppe bestimmt.
- Weiterhin können Tasks bereits konfigurativ einem spezifischen CPU-Kern zugewiesen werden, so dass durch den genetischen Algorithmus nur noch das Mapping der übrigen Tasks exploriert wird. Soll ein solches konfigurativ definiertes Mapping für eine der zuvor genannten Task-Gruppen gelten, ist die Definition der Affinität für einen beliebigen Repräsentanten der Gruppe ausreichend. Für Tasks, deren Mapping auf diese Weise bereits im Voraus festgelegt wurde, definiert der Genotyp kein Gen.
- Um Tasks explizit unterschiedlichen Kernen zuzuweisen, können Task-Gruppen $T'' \subseteq T$ mit $|T''| \leq |C|$ definiert werden, für die gefordert wird: $t_i, t_j \in T'' \implies f_m(t_i) \neq f_m(t_j)$.

Diese Invariante wird jedoch nicht im Genotyp abgebildet, sondern findet erst im Rahmen der noch zu definierenden Fitnessbewertung Berücksichtigung.

Als *Epistase* wird generell der Effekt bezeichnet, dass der Beitrag, den ein Gen i aus einer Menge von N Genen zur Fitness eines Individuums liefert, von dem Wert K_i anderer Gene abhängt. Zur Beschreibung dieses Effekts definierte Kauffman [93] die sogenannten *NK-Landschaften* für skalare Optimierungen, die sich schließlich für multikriterielle Optimierungen zu sogenannten *MNK-Landschaften* erweitern lassen. Mittels MNK-Landschaften lässt sich beschreiben, dass bei einer M-dimensionalen Optimierung die Art der Interaktion zwischen den Genen eines Genotyps x spezifisch für jedes Kriterium $f_i(x)$ mit $1 \leq i \leq M$ ist [6]:

$$f_i(x) = \frac{1}{N} \sum_{j=1}^{N} f_{i,j} \left(x_j, z_1^{(i,j)}, z_2^{(i,j)}, \ldots, z_{K_i}^{(i,j)} \right) \tag{5.36}$$

Dabei sei x_j das j-te Gen eines Genotyps x der Länge N. Zudem seien $z_1^{(i,j)}, z_2^{(i,j)}, \ldots, z_{K_i}^{(i,j)}$ die K_i mit x_j epistatisch interagierenden Gene, von denen der Beitrag $f_{i,j}$ eines Gens x_j zur i-ten Dimension $f_i(x)$ der Fitness abhängt. Für den zuvor definierten Genotyp zur Repräsentation eines Task-Mappings gilt dabei:

- Für das Kriterium der Lastverteilung hängt der Beitrag eines Gens zur entsprechenden Fitnessbewertung $f_1(x)$ von $K_1 = N - 1$ Genen ab. Dies ergibt sich aus der Tatsache, dass sich sowohl die Metrik der Spannweite als auch die der Maximallast (vgl. Formeln 5.2 und 5.3) aus den durchschnittlichen Auslastungen der CPU-Kerne C ergibt, deren Berechnung wiederum die Lastanteile aller Tasks T berücksichtigt (vgl. Formel 5.1). Somit ist der Einfluss einer Core-Zuordnung einer spezifischen Task $t \in T$ auf die Lastverteilung eines Mappings stets von den Zuordnungen aller übrigen $N - 1$ Tasks abhängig.
- Hinsichtlich der Inter-Core-Interaktionen ist der Beitrag der Core-Zuordnung einer Task $t \in T$ zur diesbezüglichen Bewertung $f_2(x)$ eines Task-Mappings vom Mapping aller $K_2 \leq N - 1$ Tasks $\hat{T} \subset T$ abhängig, zu denen der Graph von t aus eine Interaktionskante definiert. Dies gilt, da erst die zusätzliche Berücksichtigung des Mappings jeder Task $\hat{t} \in \hat{T}$ definiert, ob es sich hierbei jeweils um eine Inter-Core-Interaktion handelt.
- Bezüglich des Aufwands $f_3(x)$ besteht analog dazu die epistatische Interaktion zwischen einer Task und allen $K_3 \leq N - 1$ Tasks, zu denen der Graph eine Aufwandskante definiert.

Somit kann die Gen-Epistase je nach Graph gegebenenfalls hinsichtlich aller Bewertungskriterien maximal sein. Dies stellt für den genetischen Algorithmus insofern eine Herausforderung dar, als diese epistatischen Interaktionen zwischen den Genen die Exploration eines Genotyps mit einer guten Fitness erschweren. So kann eine Mutation eines Gens dazu führen, dass sich der Beitrag dieses Gens zur Fitness des Phänotyps verbessert, während sich als Folge der epistatischen Interaktionen zugleich der Fitnessbeitrag anderer Gene verschlechtert. Dies macht es schwerer, den Fitnessbeitrag aller Gene gleichzeitig zu optimieren [6] und stellt neben den in Konkurrenz stehenden Bewertungskriterien einen weiteren Konflikt dar, mit dem der genetische Algorithmus umgehen muss.

Die in Abschnitt 5.6.1 geforderte Redundanzfreiheit wird von der zuvor definierten Genotyp-Repräsentation erfüllt, da zwischen den Genotypen und Mappings eine 1-zu-1-Beziehung besteht. Des Weiteren stellen alle Genotypen eine legale Lösung für die definierte Anzahl an CPU-

Kernen dar und zu jedem möglichen Mapping lässt sich der Genotyp definieren. Zudem gilt die Lamarcksche Eigenschaft, da sich die Zuordnung einer Task zu einem CPU-Kern unabhängig von den Werten der übrigen Gene ableiten lässt. Schließlich wird auch die Kausalitätsforderung erfüllt, da die Änderung eines Gens lediglich das Mapping einer einzelnen Task beeinflusst.

Genetische Operatoren

Als Rekombinationsoperator auf jeweils zwei zufällig ausgewählten Genotypen wird ein Ein-Punkt-Crossover mit einer Wahrscheinlichkeit p_c eingesetzt, wobei die Position des Crossovers zufällig bestimmt wird. Dabei zeigt sich ein weiterer Vorteil der Wertekodierung: Die aus dem Crossover resultierenden Genotypen definieren nur Core-Indizes, die das jeweilige Gen in einem der beiden Elternindividuen hatte. Dies gilt, da die Crossover-Position im Genotyp stets zwischen zwei Genen und somit im Fall der Wertekodierung zwischen zwei Task-Zuordnungen liegt. Hätte man eine binäre Kodierung gewählt, könnte ein Crossover zwischen zwei Bits erfolgen und somit im resultierenden Genotyp für eine Task einen Core-Index generieren, der dieser Task in keinem der beiden Elternindividuen zugewiesen war.

Weiterhin ist für einen Genotyp eine Mutation mit der Wahrscheinlichkeit p_m pro Gen implementiert. Im Fall der Mutation eines Gens findet eine Substitution des aktuellen Core-Index durch einen zufällig gewählten Core-Index $0 \leq j < |C|$ statt. Da bei der Evaluation eines Task-Mappings hinsichtlich der Inter-Core-Interaktionen und Aufwände die Distanz der Core-Indizes irrelevant ist, muss hier kein Mutationsradius berücksichtigt werden. Würde man allerdings die Modellierung dahingehend erweitern, dass beispielsweise eine Interaktion zwischen CPU-Kernen c_1 und c_2 aufgrund eines gemeinsamen L2-Cache einen geringeren Overhead verursacht als zwischen Kernen c_2 und c_3, wäre auch der Mutationsradius relevant.

Ableitung und Bewertung des Phänotyps

Die Ableitung eines Task-Mappings aus dem zuvor definierten Genotyp gestaltet sich trivial, indem für jede Task $t \in T$ ein Mapping auf den CPU-Kern definiert wird, dessen Index an entsprechender Stelle im Genotyp kodiert ist. Nun werden alle berücksichtigten Graphen $\mathcal{G}$ mittels dieses Mappings parametriert, wobei Zuordnungen für Tasks, die nicht Bestandteil eines Graphen $G_i \in \mathcal{G}$ sind, ignoriert werden. Die Bewertung dieser parametrierten Graphen erfolgt schließlich mittels der in Abschnitt 5.4.1 beschriebenen Metriken.

Allerdings kann der Raum der Genotypen trotz der Wahl einer geeigneten Kodierung ungeeignete Individuen enthalten. Im Fall des Task-Mappings bestimmt sich die Eignung einer Lösung unter anderem mittels der Einschränkung, dass für die Auslastung f_{l_c} keines der CPU-Kerne $c \in C$ unter einem Mapping f_m die Eigenschaft $f_{l_c}(f_m, c) > 100\,\%$ zutrifft. Um diese harte Randbedingung abzubilden, wird ein Task-Mapping, das eine Überlast auf einem CPU-Kern erzeugt, bezüglich seiner Lastverteilung f_1 mit einer Strafe in Höhe des durch den jeweiligen Datentyp maximal darstellbaren Wertes belegt. Dies hat aufgrund der multikriteriellen Optimierung folgende Konsequenzen: Zeichnet sich das Individuum bezüglich der übrigen Kriterien nur durch eine mittelmäßige Fitness aus, führt diese Maximalstrafe meist zur sofortigen Eliminierung durch den Selektor. Dies ist auch wünschenswert, da in diesem Fall davon auszugehen ist, dass das repräsentierte Mapping nicht nur eine ungültige Lastverteilung hat, sondern auch bezüglich der übrigen Kriterien kein Potential besitzt. Im Fall guter Fitnesswerte f_2 oder f_3 wird

die fehlende Eignung des Individuums hingegen in der Regel ignoriert und ein Verbleib in der Population ist zunächst wahrscheinlich. Dies ist ebenfalls gewünscht, damit das Individuum sein genetisches Potential bezüglich f_2 oder f_3 in folgende Generationen weitergeben kann.

Darüber hinaus wurde zuvor mit den Task-Gruppen T'' eine weitere Einschränkung definiert, die es im Rahmen der Fitnessbewertung zu berücksichtigen gilt. So gilt jedes Mapping, das zwei Tasks t_i, t_j einer Gruppe T'' dem gleichen CPU-Kern zuordnet, als ungeeignet. Eine derartiges Individuum wird hinsichtlich aller Bewertungskriterien mit der jeweils maximalen Strafe belegt und somit in der Regel sofort durch den Selektor aus der Population eliminiert.

5.6.3 Entwurfsraumexploration auf Taskebene

Repräsentation des Genotyps

Bei der Modellierung auf Taskebene besteht die Aufgabe des Genotyps darin, die Parametrierung eines Codeblock-Graphen durch einen Schedule S abzubilden. Somit sind neben einer Partition $P(N_b)$ auch die Ausführungsreihenfolgen der Codeblöcke jedes CPU-Kerns als totale Ordnungen R_i zu repräsentieren (vgl. Formel 5.18). Da eine Task $t_d \in T_d$ eine Agglomeration von Codeblöcken $n_b \in N_b$ darstellt, gilt somit für das Mapping $f_m \colon N_b \to C$[13]:

$$f_m\left(n_b\right) := f_m\left(f_a\left(n_b\right)\right) = f_m\left(t_d\right) \tag{5.37}$$

Daraus ergibt sich folgende Eigenschaft für eine Partition $P_i \in P(N_b)$:

$$\begin{aligned} n_{b_1}, n_{b_2} \in P_i &\overset{(5.19)}{\iff} f_a\left(n_{b_1}\right) = f_a\left(n_{b_2}\right) \\ &\overset{(5.14)}{\iff} f_m\left(f_a\left(n_{b_1}\right)\right) = f_m\left(f_a\left(n_{b_2}\right)\right) \\ &\overset{(5.37)}{\iff} f_m\left(n_{b_1}\right) = f_m\left(n_{b_2}\right) \end{aligned} \tag{5.38}$$

Als Konsequenz kann die Partition $P(N_b)$ im Genotyp über ein Mapping $f_m \colon N_b \to C$ abgebildet werden, indem zwei Codeblöcke n_{b_1}, n_{b_2} genau dann der gleichen Partition P_i zugeordnet sind, wenn mittels eines Mappings f_m die Allokation zu einem identischen CPU-Kern definiert ist. Sinnen [159] definiert für die Genotyp-Repräsentation eines Schedules, bestehend aus einer Prozessorallokation und einer Ausführungsreihenfolge, folgende Alternativen:

- *Indirekte Repräsentation:* Bei der indirekten Repräsentation wird der Schedule durch den Genotyp nur partiell festgelegt, indem dieser entweder eine Allokation oder eine Ausführungsreihenfolge respektive Prioritätenordnung, aus der sich eine Ausführungsreihenfolge ableiten lässt, definiert. Um einen Schedule jedoch vollständig zu determinieren, muss bei der Ableitung des Phänotyps entweder zu einer Prozessorallokation noch die Ausführungsreihenfolge oder zu einer Ausführungsreihenfolge noch die Prozessorallokation bestimmt werden. Da beide dieser Probleme NP-schwer sind, empfiehlt sich der Einsatz einer Heuristik, wie beispielsweise ein *List Scheduling*, um den Schedule abzuleiten. Eine solche kann jedoch nicht garantieren, dass der optimale Schedule gefunden wird, so dass im Fall der indirekten Repräsentation der optimale Schedule nicht zwingend im Suchraum des genetischen Algorithmus liegt [159].

[13] Im Folgenden sei f_m sowohl als $f_m \colon T_d \to C$ als auch als $f_m \colon N_b \to C$ definiert und ordne je nach Kontext den Tasks T_d respektive den Codeblöcken N_b den jeweiligen CPU-Kern $c \in C$ zu.

- *Direkte Repräsentation:* Bei der direkten Repräsentation besteht das Ziel hingegen darin, auf Basis einer Repräsentation keine weiteren heuristischen Entscheidungen mehr treffen zu müssen. Aus diesem Grund kann eine direkte Repräsentation garantieren, dass der optimale Schedule Bestandteil des Suchraums ist [159]. Als Konsequenz muss dann allerdings durch den Genotyp sowohl eine Allokation als auch eine Ausführungsreihenfolge oder eine Priorität definiert werden, auf Basis derer die Reihenfolge in deterministischer Weise abgeleitet wird. Dies hat jedoch einen deutlich komplexeren Genotyp sowie entsprechend komplexere Reproduktionsoperatoren zur Folge.

Der Einsatz genetischer Algorithmen ist entsprechend Abschnitt 5.1.1 durch die Tatsache motiviert, dass durch diese eine Lösung stets problemübergreifend generiert und problemabhängig für eine Vielzahl unterschiedlich ausgeprägter Modelle evaluiert werden kann. Die bei einer indirekten Lösungsrepräsentation erforderliche Scheduling-Heuristik kann allerdings nicht problemübergreifend angewandt werden, da diese die spezifischen Abhängigkeiten und Codeblock-Laufzeiten eines Graphen berücksichtigen muss. Als Konsequenz wurde für den Genotyp des Codeblock-Schedulings die direkte Repräsentation gewählt. Zur direkten Repräsentation ist dabei ein aus zwei *Chromosomen* bestehender Genotyp nötig. Das erste Chromosom kodiert analog zu der in Abschnitt 5.6.2 für den Task-Graphen beschriebenen Weise das Mapping der Codeblöcke auf die verfügbaren CPU-Kerne, so dass die Gene aus den zuvor genannten Gründen erneut wertekodiert sind. Dabei können ebenso wie beim Task-Graphen Einschränkungen hinsichtlich des Mappings von Codeblöcken auf CPU-Kerne definiert werden, die dann äquivalent zu der in Abschnitt 5.6.2 beschriebenen Weise bei der Repräsentation des Genotyps berücksichtigt werden. Das zweite Chromosom wiederum definiert eine totale Ordnung der Codeblöcke in Form einer sogenannten *Permutationskodierung* [160]. Zur Kodierung wird dabei zunächst jedem Codeblock $n_b \in N_b$ ein ganzzahliger Index $j \in \{0, \ldots, |N_b| - 1\}$ zugewiesen. Nun repräsentiert der Wert jedes Gens des zweiten Chromosoms einen Codeblock über seinen Index j und die Reihenfolge der Gene im Chromosom definiert eine totale Codeblock-Ordnung. Die durch das zweite Chromosom definierte totale Ordnung könnte nun als Ausführungsreihenfolge der Codeblöcke interpretiert werden. Die Häufigkeit, mit der dann im Rahmen der Reproduktion Ausführungsreihenfolgen generiert werden würden, die sich aufgrund von Konflikten mit den Datenflüssen der Graphen $\mathcal{G}$ nicht realisieren lassen, ist allerdings potentiell hoch. Derartige ungeeignete Genotypen müssen regelmäßig eliminiert oder im Rahmen der Fitnessbewertung bestraft werden. Aus diesem Grund wird die totale Ordnung des zweiten Chromosoms als Prioritätenordnung interpretiert, auf Basis derer unter Berücksichtigung der in den Graphen $\mathcal{G}$ definierten Abhängigkeiten die Ordnungen R_i abgeleitet werden.

Für den bei N Codeblöcken durch zwei Chromosomen mit jeweils N Genen definierten Genotyp zur Repräsentation eines Schedules gilt hinsichtlich der Gen-Epistase:

- Entsprechend Formel 5.17 berechnet sich der Speedup einer Task-Dekomposition auf Basis deren Ausführungsdauer, die wiederum als Endzeitpunkt der am spätesten endenden Ausführung eines Codeblocks (vgl. Formel 5.27) definiert ist. Ob ein Codeblock den spätesten Endzeitpunkt aller Codeblöcke hat, wird jedoch weder allein durch dessen Mapping noch durch dessen Priorität definiert, sondern stets auch durch das Mapping und die Priorität aller übrigen Codeblöcke. Somit hängt der Fitnessbeitrag eines Gens zum Kriterium $f_1(x)$ stets auch von den Werten der $K_1 = 2 \cdot N - 1$ übrigen Gene ab.

- Hinsichtlich der Inter-Core-Interaktionen $f_2(x)$ und des Aufwands $f_3(x)$ besteht analog zum Task-Graphen erneut eine Abhängigkeit zwischen dem Mapping eines Codeblocks und den $K_3 \leq N - 1$ Genen, die das Mapping der Codeblöcke bestimmen, zu denen der Graph eine Interaktions- beziehungsweise Aufwandskante definiert. Die Gene des zweiten Chromosoms sind hingegen irrelevant für die Kriterien $f_2(x)$ und $f_3(x)$, da die Ausführungsreihenfolge der Codeblöcke für die Bewertung der Inter-Core-Interaktionen und des Implementierungsaufwands nicht maßgeblich ist.

Somit kann bei der Optimierung auf Taskebene die epistatische Gen-Interaktion je nach Graph und Bewertungskriterium alle Gene des Genotyps respektive des ersten Chromosoms umfassen, was die bereits in Abschnitt 5.6.2 dargestellten Konsequenzen für den Algorithmus hat.

Da bei der zuvor definierten Genotyp-Repräsentation mehrere Prioritätenordnungen auf eine identische Ausführungsreihenfolge abgebildet werden können, gilt darüber hinaus die in Abschnitt 5.6.1 geforderte Redundanzfreiheit nicht. Allerdings genügt die Repräsentation der Forderung nach Legalität und Vollständigkeit, da jeder Genotyp in einen geeigneten oder ungeeigneten Schedule transformiert werden kann und sich jeder beliebige Schedule als Kombination eines Codeblock-Mappings und einer Prioritätenordnung abbilden lässt. Die Lamarcksche Eigenschaft wird wiederum nur eingeschränkt erfüllt, da der Phänotyp mittels zweier Chromosomen determiniert wird. Als Konsequenz ist beispielsweise die Semantik einer Codeblock-Priorität von dessen Mapping abhängig, so dass ein Codeblock mit identischer Priorität in Abhängigkeit von seinem Mapping entweder den Abschluss der Task auf Core $c_i \in C$ oder den Anfang der Task auf Core $c_j \in C$ mit $c_i \neq c_j$ bildet. Folglich ist auch die Kausalität der Genotypen beim Codeblock-Scheduling geringer ausgeprägt als beim Task-Mapping, da zum Beispiel ein anderes Mapping eines einzelnen Codeblocks bei einer unveränderten Prioritätenordnung zu einem grundlegend anderen Schedule führen kann.

Genetische Operatoren

Da der Genotyp des Codeblock-Schedulings aus zwei Chromosomen unterschiedlicher Kodierung besteht, ist für jedes Chromosom eine spezifische Vorgehensweise bei der Reproduktion erforderlich. Die Kodierung des ersten Chromosoms ist identisch mit dem Genotyp des Task-Mappings, so dass auch der Crossover-Operator gemäß der Beschreibung in Abschnitt 5.6.2 definiert ist. Im Fall der Permutationskodierung hingegen ist der Crossover-Operator zwar ebenfalls ein Ein-Punkt-Crossover, allerdings kann ein einfacher Crossover dazu führen, dass das resultierende Chromosom keine Permutation mehr abbildet, sondern Codeblock-Indizes mehrfach enthält. Aus diesem Grund wird der Permutations-Crossover nach [139] angewandt, indem zunächst alle Gene des ersten Elternchromosoms bis zur Crossover-Position kopiert werden. Danach werden, wie in Abbildung 5.27 dargestellt, in der Reihenfolge der Permutation nacheinander alle Gene des zweiten Elternchromosoms angehängt, die nicht Bestandteil der Genfolge waren, die vom ersten Elternchromosom kopiert wurde. Auf diese Weise wird sichergestellt, dass beide resultierenden Chromosomen wieder eine Permutation darstellen.

Der Mutationsoperator für das wertekodierte erste Chromosom ist wiederum in identischer Weise implementiert wie beim Task-Mapping. Um die Permutationseigenschaft zu wahren, darf beim zweiten Chromosom hingegen für ein Gen nicht zufällig ein anderer Wert gesetzt werden. Stattdessen werden entsprechend Abbildung 5.27 die Werte zweier zufällig ausgewählter Gene im Chromosom miteinander vertauscht [139]. Die Mutation ist dabei wie folgt implementiert:

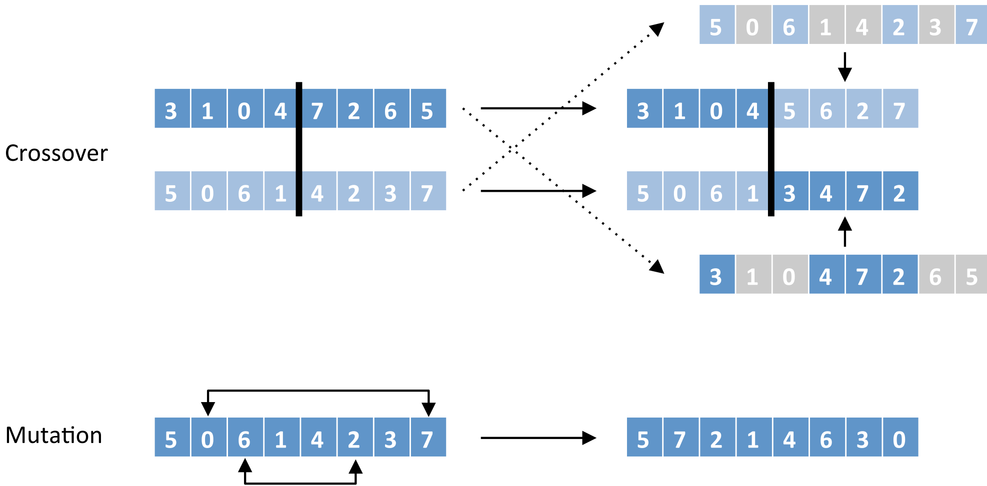


Abbildung 5.27: Crossover und Mutation bei Permutationskodierung (nach [159])

Jeder Genindex wird mit der Wahrscheinlichkeit p_m in eine Indexliste aufgenommen, die anschließend zufällig permutiert wird. Nun werden im Chromosom die Werte von jeweils zwei in der permutierten Auswahl aufeinanderfolgenden Indizes vertauscht. So ist sichergestellt, dass pro Gen maximal ein Tausch stattfindet und die Permutationseigenschaft gewahrt bleibt.

Ableitung und Bewertung des Phänotyps

Durch die Wahl der direkten Repräsentation gibt es bei der Ableitung des Codeblock-Schedules aus den per Genotyp vorgegebenen Codeblock-Mappings und -Prioritäten nur noch wenige Freiheitsgrade. Diese sind allerdings erforderlich, um die Häufigkeit der evolutionären Generierung ungeeigneter Ausführungsreihenfolgen, die sich aufgrund von Abhängigkeiten nicht realisieren lassen, zu reduzieren. Da der abgeleitete Schedule für alle im Rahmen der genetischen Optimierung berücksichtigten Graphen $\mathcal{G}$ identisch und valide sein soll, muss eine problemübergreifende Festlegung der Ausführungsreihenfolge die Vereinigungsmenge aller in den Graphen $\mathcal{G}$ mittels Interaktionskanten definierten Abhängigkeiten berücksichtigen. Zu diesem Zweck sei zunächst die Vereinigungsmenge N'_b der Codeblöcke aller Graphen $\mathcal{G}$ definiert; $N_b(G)$ bezeichne dabei die Menge der Codeblöcke eines Graphen G:

$$N'_b \coloneqq \bigcup_{G \in \mathcal{G}} N_b(G) \tag{5.39}$$

Weiterhin sei eine Relation $R_E \subseteq N'_b \times N'_b$ definiert; $E_i(G)$ bezeichne dabei die Menge der Interaktionskanten eines Graphen G:

$$\left(n_{b_1}, n_{b_2}\right) \in R_E \Longleftrightarrow \exists G \in \mathcal{G}: \left(n_{b_1}, n_{b_2}\right) \in E_i(G) \tag{5.40}$$

Eingabe: Vereinigungsmenge N'_b der Codeblöcke aller Graphen $\mathcal{G}$ (Formel 5.39).
Ausgabe: Topologische Sortierung L_p der Codeblöcke N'_b.
begin
  $L_u \leftarrow N'_b$; $L_p \leftarrow \varnothing$;
  Sortiere L_u nach absteigender Priorität entsprechend dem zweiten Chromosom;
  while ($L_u \neq \varnothing$) **do**
    for $i \leftarrow 0$ **to** $|L_u| - 1$ **do**
      $n_b \leftarrow L_u[i]$;
      if $\text{pred}_{g/l}(n_b) \subseteq L_p$ **then**
        break;
    Entferne n_b aus L_u und füge n_b an das Ende der Liste L_p an;

Algorithmus 5.2: Topologische Sortierung der Codeblöcke N'_b auf Basis der im zweiten Chromosom definierten Prioritätenordnung unter Berücksichtigung globaler oder lokaler Datenabhängigkeiten pred_g respektive pred_l

Es besteht also genau dann eine Relation R_E zwischen zwei Codeblöcken $n_{b_1}, n_{b_2} \in N'_b$, wenn es mindestens einen Graphen $G \in \mathcal{G}$ gibt, der eine Interaktionskante $e_i = (n_{b_1}, n_{b_2})$ zwischen diesen beiden Codeblöcken definiert.

Mittels einer Funktion $\text{pred}_g\colon N'_b \to \mathcal{P}(N'_b)$ sei zudem die Menge aller Codeblöcke definiert, von denen ein Codeblock $n_b \in N'_b$ direkt oder indirekt datenabhängig ist; R_E^+ bezeichnet dabei den transitiven Abschluss der Relation R_E:

$$\text{pred}_g(n_b) \coloneqq \left\{ n'_b \in N'_b \,\middle|\, (n'_b, n_b) \in R_E^+ \right\} \tag{5.41}$$

Alternativ dazu sei durch eine Funktion $\text{pred}_l\colon N'_b \to \mathcal{P}(N'_b)$ die Menge aller Codeblöcke definiert, von denen ein Codeblock $n_b \in N'_b$ direkt oder indirekt datenabhängig ist und die durch das im Genotyp kodierte Mapping dem gleichen CPU-Kern wie n_b zugeordnet sind:

$$\text{pred}_l(n_b) \coloneqq \left\{ n'_b \in N'_b \,\middle|\, (n'_b, n_b) \in R_E^+ \wedge f_m(n'_b) = f_m(n_b) \right\} \tag{5.42}$$

Im Rahmen der Generierung eines Schedules auf Basis eines Genotyps gilt es nun zunächst, aus der im zweiten Chromosom des Genotyps kodierten Prioritätenordnung der Codeblöcke eine zu den in den Graphen $\mathcal{G}$ definierten Abhängigkeiten kompatible Codeblock-Ordnung abzuleiten. Dabei handelt es sich um das Problem, eine topologische Sortierung der Codeblöcke hinsichtlich der Interaktionskanten der Graphen $\mathcal{G}$ unter einer möglichst geringen Abweichung von der Prioritätenordnung vorzunehmen (vgl. Algorithmus 5.2). Eine solche Sortierung existiert stets, da der Graph per Definition bezüglich aller Kanten und somit insbesondere bezüglich der Interaktionskanten azyklisch ist. Der Algorithmus generiert dabei zunächst eine Liste L_u ungeplanter Codeblöcke N'_b, die in absteigender Weise entsprechend der im zweiten Chromosom definierten Priorität sortiert wird. Weiterhin wird eine zunächst leere Liste L_p geplanter Codeblöcke instanziiert, die zum Schluss der topologischen Sortierung der Codeblöcke entspricht. Stets beginnend bei dem Codeblock mit der höchsten Priorität wird die Liste L_u nun

iterativ nach einem Codeblock n_b durchsucht, der keine lokalen oder globalen Abhängigkeiten $\text{pred}_l(n_b)$ respektive $\text{pred}_g(n_b)$ zu Codeblöcken hat, die noch nicht in der Liste L_p der bereits geplanten Codeblöcke enthalten sind. Die Ausführung von n_b ist dann nicht von einem noch ungeplanten Codeblock abhängig und kann somit mittels einer Verschiebung des Codeblocks n_b von L_u nach L_p geplant werden. Die iterative Suche wird nun so lange fortgesetzt, bis die Migration aller Codeblöcke von L_u nach L_p abgeschlossen ist. Auf diese Weise wird gewährleistet, dass jeder Codeblock zum frühestmöglichen Zeitpunkt in die resultierende Prioritätenordnung L_p übernommen wird, so dass die ursprüngliche Ordnung L_u nur soweit modifiziert wird, wie es aufgrund lokaler respektive globaler Abhängigkeiten zwingend erforderlich ist. Dabei werden bei einer topologischen Sortierung unter Berücksichtigung der Mengen $\text{pred}_g(n_b)$ die globalen direkten und indirekten Abhängigkeiten eines Codeblocks n_b berücksichtigt, während die Mengen $\text{pred}_l(n_b)$ nur die lokalen direkten und indirekten Abhängigkeiten eines Codeblocks zu Codeblöcken des gleichen CPU-Kerns miteinbeziehen.

Da die Funktion pred_l gegenüber pred_g eine Einschränkung hinsichtlich der Core-Zuordnung definiert, gilt für jeden Codeblock $n_b \in N'_b$ die Eigenschaft $\text{pred}_l(n_b) \subseteq \text{pred}_g(n_b)$. Somit wird bei einer topologischen Sortierung unter pred_l im Vergleich zu pred_g stets nur eine Teilmenge der Abhängigkeiten berücksichtigt. Dies wiederum führt bei einer Sortierung unter pred_l zu weniger oder maximal gleich vielen korrigierenden Eingriffen in die im zweiten Chromosom kodierte Prioritätenordnung wie bei einer Sortierung unter pred_g. In Abbildung 5.28 wird dies unter anderem am Beispiel von Core 0 deutlich. Durch den Genotyp wird für die Codeblöcke mit den IDs 1, 9 und 12 die Prioritätenordnung $12, \ldots, 1, \ldots, 9$ vorgegeben. Im Fall globaler Abhängigkeiten ist dem Codeblock 12 allerdings der am niedrigsten priorisierte Codeblock 10 vorzuziehen, da er von diesem im Modell datenabhängig ist. Somit rückt der Codeblock 12 an das Ende der topologischen Sortierung, so dass für Core 0 die Ausführungsreihenfolge $1, 9, 12$ resultiert. Werden hingegen nur lokale Abhängigkeiten berücksichtigt, ist der Codeblock 12 nur vom Codeblock 1 indirekt über die Codeblöcke 5 und 10 oder Codeblock 6 datenabhängig. Somit wird nur der Codeblock 1 vorgezogen und dann entsprechend der Prioritätenordnung der Codeblock 12 vor dem Codeblock 9 eingeplant, woraus sich für Core 0 die Reihenfolge $1, 12, 9$ ergibt, die der kodierten Prioritätenordnung deutlich ähnlicher ist.

Schließlich erfordert die vollständige Definition des Schedules, die totalen Ordnungen R_i aus der zuvor generierten topologischen Sortierung abzuleiten. Dies erfolgt, indem jeweils nur die einem CPU-Kern $c_i \in C$ zugeordneten Codeblöcke entsprechend ihrer Sortierung in eine totale Ordnung R_i übernommen werden. Der Prozess der Ableitung des Schedules ist in Abbildung 5.28 jeweils für eine topologische Sortierung unter globalen Abhängigkeiten pred_g und unter lokalen Abhängigkeiten pred_l dargestellt. Abbildung 5.29 zeigt zudem die entsprechend dieser Schedules resultierende Parametrierung eines Graphen mit spezifisch ausgeprägten Codeblock-Laufzeiten als Gantt-Diagramm. Generell werden bei der Parametrierung eines Graphen $G_i \in \mathcal{G}$ durch einen Schedule Codeblöcke, die nicht Bestandteil von G_i sind, ignoriert.

In einem letzten Schritt erfolgt die Bewertung der durch den abgeleiteten Schedule parametrierten Graphen $\mathcal{G}$ unter den in Abschnitt 5.5.1 definierten Metriken. Auch hier ist ein angemessener Umgang mit ungeeigneten Lösungen gefordert: Bei der Berücksichtigung ausschließlich lokaler Abhängigkeiten können entsprechend dem in Abbildung 5.30 dargestellten Beispiel ungültige Schedules mit Zyklen aus Datenfluss- und Kontrollflusskanten entstehen, die somit die geforderte Azyklik des Modells des Codeblock-Graphen verletzen: Die Codeblöcke 2 und 6 sind unterschiedlichen Kernen zugeordnet, so dass deren Datenabhängigkeit im Fall

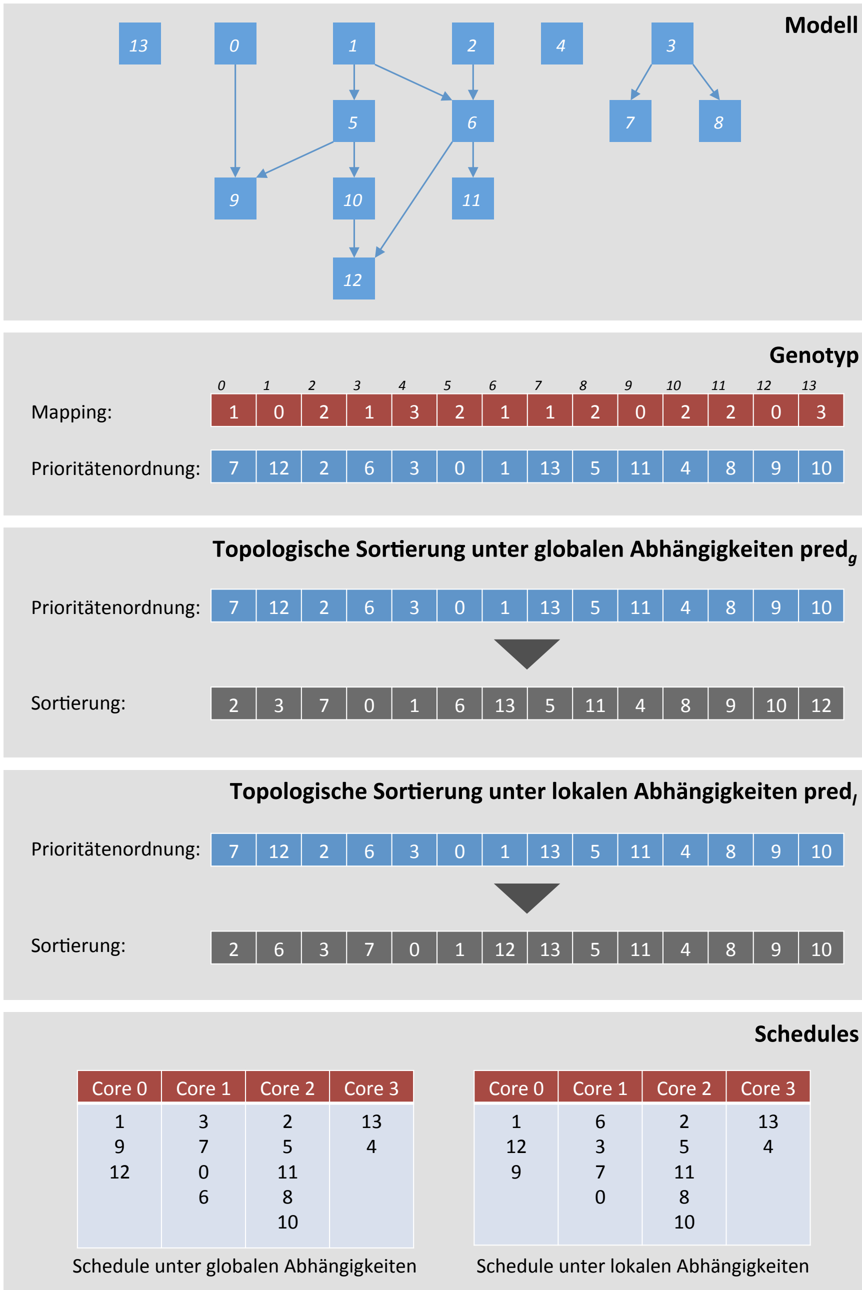


Abbildung 5.28: Alternativen der Ableitung der Schedules von Codeblöcken aus einem Genotyp unter Berücksichtigung der Datenflüsse eines ausgeprägten Modells

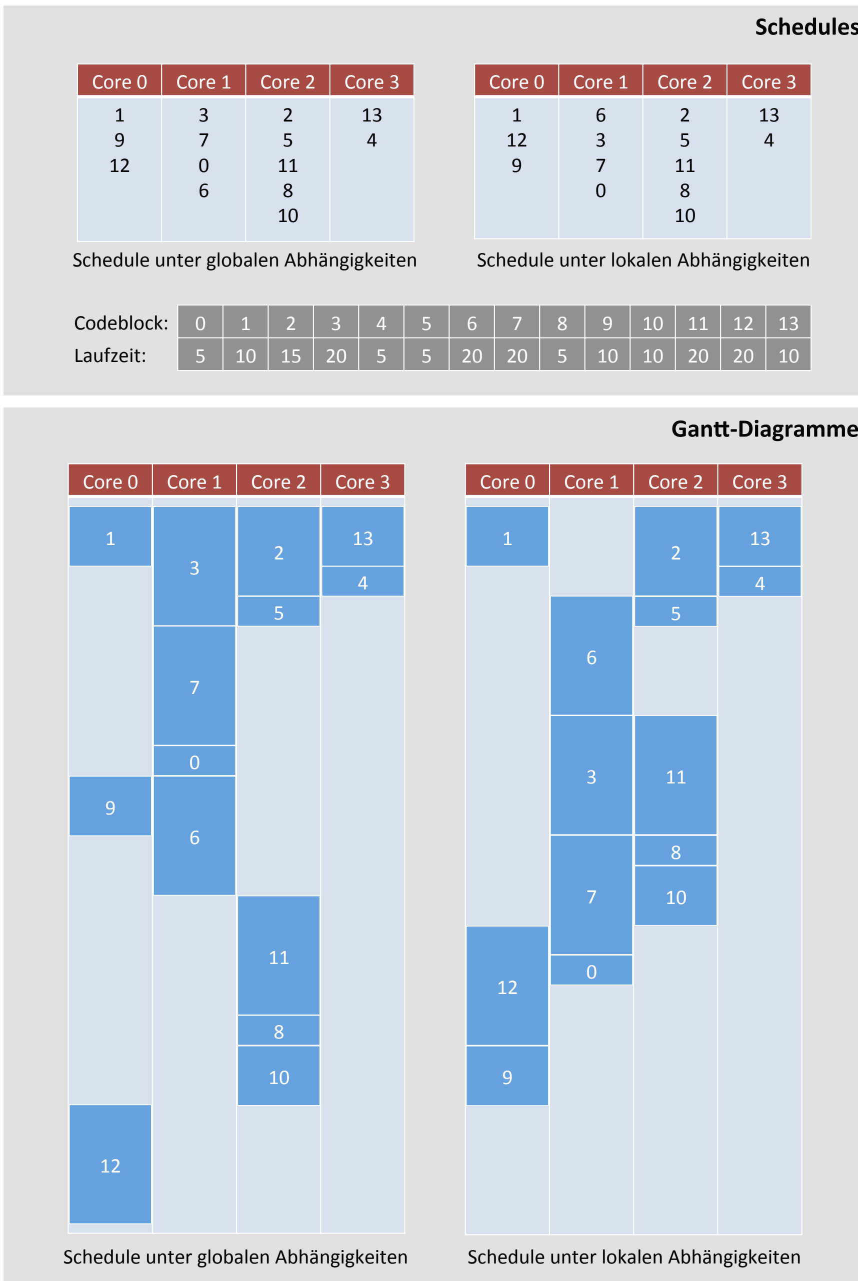


Abbildung 5.29: Gantt-Diagramme des in Abbildung 5.28 dargestellten Modells bei einer den abgeleiteten Schedules entsprechenden Parametrierung

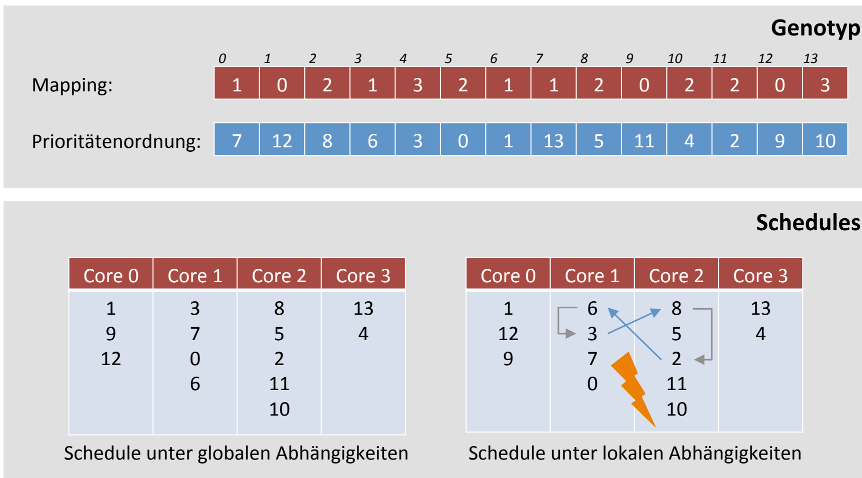


Abbildung 5.30: Konflikt in der Ausführungsreihenfolge bei einer Ableitung des Schedules unter lokalen Abhängigkeiten

lokaler Abhängigkeiten nicht berücksichtigt wird. Das Gleiche gilt für die Codeblöcke 3 und 8. Somit werden alle vier Codeblöcke entsprechend ihrer Priorität geplant, woraus sich allerdings der in der Abbildung dargestellte Konflikt ergibt, der zu einem ungültigen Schedule führt. Bei einer Korrektur der Prioritätenordnung unter Berücksichtigung globaler Abhängigkeiten kann dies hingegen nicht passieren, da ein Codeblock erst nach allen Codeblöcken, zu denen er datenabhängig ist, eingeplant wird. Zusammenfassend gilt somit:

- Die Ableitung eines Schedules unter Berücksichtigung globaler direkter und indirekter Abhängigkeiten pred_g nimmt unter keinen Umständen weniger Korrekturen in der durch den Genotyp definierte Prioritätenordnung vor als eine Ableitung unter lokalen Abhängigkeiten pred_l , erzeugt aber stets einen gültigen Schedule.
- Die Ableitung eines Schedules unter Berücksichtigung lokaler direkten und indirekter Abhängigkeiten pred_l nimmt unter keinen Umständen mehr Korrekturen in der durch den Genotyp definierte Prioritätenordnung vor als eine Ableitung unter globalen Abhängigkeiten pred_g , kann allerdings einen ungültigen Schedule erzeugen.

Um die Randbedingung der Azyklik in der Evolution abzubilden, werden Individuen der Population, die einen Zyklus aufweisen, analog zum Task-Mapping mit der im jeweiligen Datentyp maximal möglichen Strafe belegt. Als Konsequenz wird ein zyklischer Schedule, der auch bezüglich der übrigen Kriterien eine suboptimale Fitness aufweist, mit hoher Wahrscheinlichkeit durch den Selektor eliminiert. Im Fall einer guten Bewertung bezüglich den Metriken f_2 oder f_3 besteht hingegen die Chance, dass auch eine ungeeignete Lösung in der Population verbleibt. Dies ist auch gewünscht, da durch eine Mutation im Mapping oder in der Prioritätenordnung die Möglichkeit zur Entwicklung in einen azyklischen Schedule besteht.

5.7 Ableitung der Implementierung

Wurde eine Pareto-optimale Alternative der Firmware-Parallelisierung auf System- oder Taskebene gewählt, ist diese in eine entsprechende Implementierung zu überführen. Diesen Prozess beschreiben die nachfolgenden Abschnitte zunächst separat für die System- und die Taskebene und definieren im Anschluss daran ein ganzheitliches Vorgehen zur kombinierten Anwendung der zuvor beschriebenen Methode.

5.7.1 Parallelisierung auf Systemebene

Da auf Systemebene entsprechend Definition 6 bereits ein Multitasking-System modelliert wird, sind umfangreiche Restrukturierungen des Codes in der Regel nicht erforderlich. Stattdessen besteht der erste Schritt einer Realisierung in der Definition der statischen Task-Affinitäten entsprechend dem zur Implementierung ausgewählten Mapping. Für *VxWorks* beispielsweise erfolgt dies mittels der Systemroutine `taskCpuAffinitySet()` [187]. Weiterhin müssen die Interrupts des Systems den CPU-Kernen entsprechend dem definierten Mapping zugewiesen werden. Im Fall von *VxWorks* wird diese Zuordnung mittels einer Konfiguration des Board Support Package (BSP) realisiert. Bei jedem Boot-Vorgang wird nun das Mapping der Interrupts aus dem BSP ausgelesen und im Interrupt-Controller (I/O-APIC) ein Routing an den entsprechenden Core konfiguriert [187]. In einem letzten Schritt sind nun die durch das Modell prädizierten Aufwände zur Implementierung multilateraler Synchronisationen beim Zugriff auf gemeinsame Ressourcen zu leisten, um eine zuverlässige Ausführung des parallelen Task-Systems zu gewährleisten.

5.7.2 Parallelisierung auf Taskebene

Die Implementierung einer Parallelisierung auf Taskebene lässt sich in eine Reihe von Schritten gliedern, die in der vorgegebenen Reihenfolge durchzuführen sind:

1. Zunächst muss eine *Refaktorierung (Refactoring)* [55] der Variablennutzung alle Abhängigkeiten eliminieren, die einer Restrukturierung des Codes entsprechend dem durch den parametrierten Codeblock-Graphen definierten Schedule S im Weg stehen. Da alle dem Problem inhärenten Abhängigkeiten bei der Definition des Schedules berücksichtigt wurden, handelt es sich hierbei ausschließlich um solche, die sich mittels einer Refaktorierung auflösen lassen.
2. Es müssen genau die multilateralen Synchronisationen bei der Nutzung gemeinsamer Ressourcen implementiert werden, die im Rahmen der Aufwandsbewertung einer Task-Dekomposition als erforderlich identifiziert wurden.
3. Nun müssen die Tasks $t_d \in T_d$ instanziiert und per Affinität statisch an den jeweiligen Kern gebunden werden, wobei deren Priorität in Anlehnung an die der ursprünglichen Task t gewählt werden sollte. Des Weiteren ist pro Task $t_d \in T_d$ eine eigene Funktion zu definieren, welche deren Eintrittspunkt darstellt. Nun kann gemäß dem im Graphen definierten Mapping eine Migration der Codeblöcke N_b der Task t in die zuvor definierten Funktionen erfolgen, wobei im Fall parallelisierter Schleifen mehrfache Kopien der entsprechenden Codeblöcke anzulegen sind. Die Reihenfolge der Codeblöcke wird dabei

durch die im parametrierten Graphen definierten Kontrollflusskanten vorgegeben. Besondere Aufmerksamkeit ist generell den Codeblöcken zu widmen, die in keinem der modellierten Codeblock-Graphen enthalten und somit im Schedule S nicht berücksichtigt sind. Dies lässt mehrere Schlussfolgerungen zu: Werden die entsprechenden Anweisungen nur in den berücksichtigten Lastprofilen nicht durchlaufen, sind aber generell relevant, so müssen diese trotzdem migriert werden, um die Äquivalenz der Funktionalität zwischen der ursprünglichen Task t und den Tasks T_d sicherzustellen. Die Zuordnung dieser Codeblöcke zu den Tasks T_d kann dann nach dem Ermessen des Entwicklers erfolgen. Dies beeinflusst zwar die Charakteristik der Task-Dekomposition, allerdings nicht unter den zur Optimierung ausgewählten Lastprofilen. Es besteht allerdings auch die Möglichkeit, dass die Codeblöcke unter keinen in der Realität relevanten Systemumgebungen durchlaufen werden oder dass es sich um generell nicht erreichbaren Code *(Dead Code)* [123] handelt. Kann diese Vermutung nach kritischer Prüfung aufrechterhalten werden, ist eine Migration dieser Codeblöcke nicht erforderlich und der Umfang der resultierenden Tasks kann um nicht mehr benötigten Code reduziert werden.

4. Es ist darauf zu achten, dass bei der Restrukturierung des Codes alle Kontrollabhängigkeiten beibehalten werden, indem zu einer Schleifen- oder Verzweigungskondition kontrollabhängige Codeblöcke nach wie vor in Abhängigkeit von der Auswertung dieser Kondition ausgeführt werden. Die im Graphen zu den Codeblöcken annotierten Verzweigungs- und Schleifenbezeichner unterstützen diesen Prozess insofern, als sie explizit auf die Kontrollabhängigkeiten eines Codeblocks hinweisen. Im Rahmen der Restrukturierung ist dabei gegebenenfalls eine umfangreiche Refaktorierung der Konditionen von Schleifen und Verzweigungen erforderlich, wie Abbildung 5.31 exemplarisch zeigt.
5. In einem weiteren Schritt ist die unilaterale Synchronisation der resultierenden Tasks T_d bei Datenabhängigkeiten zu implementieren. Entsprechende Mittel wie beispielsweise Semaphore, Ereignisse oder Nachrichten werden durch das Betriebssystem bereitgestellt.
6. Sei T_1 das ursprüngliche Task-System, T_2 das aus der Dekomposition einer Task $t \in T_1$ resultierende Task-System und $T_d \subseteq T_2$ die aus der Dekomposition von t resultierende Task-Menge. Dann ist mittels entsprechender Maßnahmen die echt parallele Ausführbarkeit für alle Task-Paare (t_i, t_j) mit $t_i \in T_1, T_2$ und $t_j \in T_d$ sicherzustellen, für die gilt: $f_m(t_i) = f_m(t) \neq f_m(t_j)$. In diesem Fall wurden t_i und t vor der Dekomposition auf dem gleichen CPU-Kern und somit quasiparallel ausgeführt, während t_i und t_j danach unterschiedlichen CPU-Kernen zugeordnet sind und somit potentiell echt parallel laufen.

5.7.3 Kombinierte Anwendung der Methode

Nachdem zuvor die Realisierung einer parallelen Implementierung sowohl auf System- als auch auf Taskebene umfassend beschrieben wurde, gilt es nun, eine kombinierte Vorgehensweise zur ganzheitlichen Parallelisierung einer Multitasking-Firmware mittels der zuvor entwickelten Methode zu definieren. Das entsprechende mehrstufige Vorgehen stellt Abbildung 5.32 dar.

Zunächst sollte eine Modellierung des kompletten Systems als Task-Graph durchgeführt werden, ohne dabei bereits Alternativen der Task-Verteilung zu evaluieren. Stattdessen kann eine detaillierte Analyse der generierten Modelle in einem ersten Schritt der Optimierung des bestehenden Systems dienen. Dabei ist der Fokus auf folgende Faktoren zu legen:

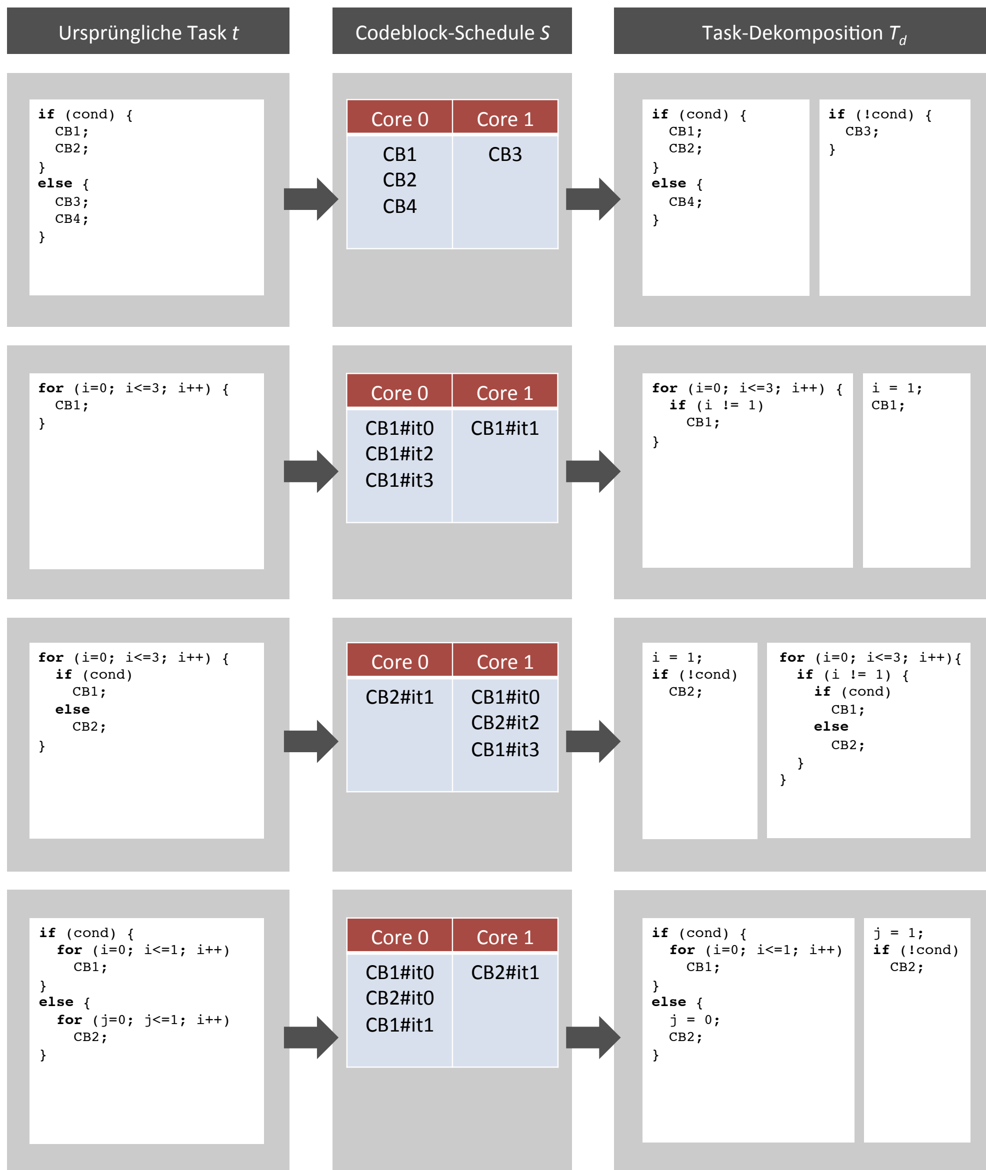


Abbildung 5.31: Beispiele für die Refaktorierung von Konditionen von Schleifen, Verzweigungen und Schachtelungen derselben

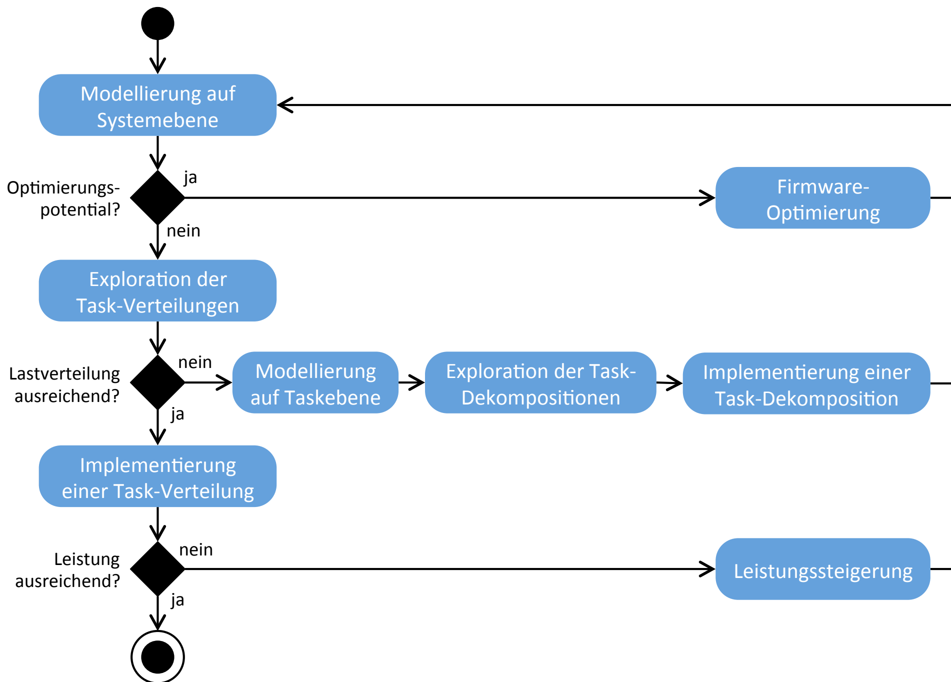


Abbildung 5.32: UML-Aktivitätsdiagramm eines ganzheitlichen Vorgehens bei der Anwendung der in der vorliegenden Arbeit entwickelten Methode

- Ist der Lastanteil einzelner Tasks signifikant höher als dies der jeweilige Anwendungsfall vermuten lässt, so kann dies einen Hinweis auf Phasen des aktiven Wartens, unnötige Berechnungen oder ein *False Sharing* [23] von Cache-Lines liefern. In diesem Fall sollte der entsprechende Code kritisch geprüft und gegebenenfalls überarbeitet werden.
- Fallen bestimmte Task-Gruppen durch einen hohen Grad an Interaktionen auf, so ist dies ein Hinweis auf zahlreiche Abhängigkeiten und Synchronisationen in der Ausführung dieser Tasks. In diesem Fall sollte eine Adaption der Firmware-Architektur mit dem Ziel erwogen werden, diese Tasks weitestgehend unabhängig voneinander auszuführen, indem Synchronisationen mit den übrigen Tasks nur zu ausgewählten Zeitpunkten stattfinden und Zugriffe auf nicht exklusiv genutzte Ressourcen reduziert werden.
- Wird eine Interaktion, wie beispielsweise ein spezifisches Semaphor, durch eine Vielzahl von Tasks genutzt, so kann dies ein Hinweis auf eine zu hohe *Lock*-Granularität im System sein, die wiederum oft einen Flaschenhals für die Systemperformanz bildet. Untermauert eine entsprechende Inspektion des Quellcodes diese Vermutung, so sollte diese multilaterale Synchronisation durch mehrere Locks geringerer Granularität ersetzt werden, um Wartezeiten im System zu reduzieren. Zu vermeiden ist allerdings auch eine zu geringe Lock-Granularität, um den Synchronisations-Overhead und das Risiko von Deadlocks zu reduzieren [64].

Wurde das System entsprechend den jeweiligen Möglichkeiten optimiert, sind erneute Systemanalysen erforderlich, um das weitere Vorgehen zu bestimmen. Zunächst ist eine weitere Modellierung auf Systemebene und eine anschließende Exploration der Pareto-Optima für die durch die Zielarchitektur vorgegebene Anzahl an CPU-Kernen durchzuführen. Zeichnet sich hier selbst das Task-Mapping mit der besten Lastverteilung durch diesbezüglich schlechte Fitnesswerte aus, sollte unter Berücksichtigung der in Abschnitt 3.1.2 definierten Kriterien eine Task im System identifiziert werden, die aufgrund ihres Lastanteils für eine Dekomposition besonders geeignet ist. Für diese ist nun eine Modellierung auf Taskebene in Verbindung mit einer Exploration der Pareto-optimalen Dekompositionen durchzuführen. Abschließend ist eines dieser Pareto-Optima auszuwählen und als Task-Menge T_d zu implementieren, wobei jedem CPU-Kern maximal eine Task $t_d \in T_d$ zugeordnet wird.

Schließlich gilt es, das nun vorliegende Task-System in geeigneter Weise auf die CPU-Kerne der Zielarchitektur zu verteilen. Wurde eine Task-Dekomposition durchgeführt, so hat diese die Systemcharakteristik hinsichtlich der Lastanteile und Interaktionen von Tasks signifikant beeinflusst, so dass zunächst eine erneute Modellierung auf Systemebene durchzuführen ist. Bei der Exploration muss nun verhindert werden, dass mehrere Tasks t_d einer Dekomposition T_d dem gleichen CPU-Kern zugeordnet werden. Diese Bedingung ist entsprechend der Beschreibung in Abschnitt 5.6.2 vor der automatisierten Exploration des Entwurfsraums durch den genetischen Algorithmus zu formalisieren. Die Pareto-Optima der Task-Verteilung des neuen Systems sollten nun aufgrund der zu erwartenden reduzierten durchschnittlichen Task-Granularität Lösungen mit einer zufriedenstellenden Lastverteilung beinhalten, von denen nun eine implementiert werden kann. Ist die beste erzielte Lastverteilung hingegen erneut suboptimal, sollte die Dekomposition weiterer Tasks in Betracht gezogen werden.

In Abschnitt 3.1.1 wurde als Nutzungsszenario einer Parallelisierung unter anderem die quantitative Leistungssteigerung definiert. Erfolgt nun eine Anpassung der Systemkonfiguration auf eine der dabei beschriebenen Arten, ist zwingend eine erneute Modellierung vorzunehmen, da die Systemcharakteristik durch eine derartige Änderung stark beeinflusst wird. Nun gilt es, für dieses Modell des geänderten Systems für eine unveränderte Anzahl an CPU-Kernen erneut die Pareto-Optima der Task-Verteilung zu bestimmen. Da bereits bei der Implementierung des ersten Task-Mappings gegebenenfalls zahlreiche Aufwände für multilaterale Synchronisationen geleistet wurden und diese nun vor der Optimierung aus dem Modell entfernt werden (vgl. Abschnitt 5.6.1), sind diesmal deutlich geringere Aufwandsbewertungen der Lösungen zu erwarten. Dabei können sich allerdings zuvor geleistete Aufwände zur Sicherstellung der echt parallelen Ausführbarkeit als überflüssig erweisen, wenn die nun echt parallel ausführbaren Tasks durch das resultierende Mapping wieder dem gleichen CPU-Kern zugeordnet werden.

Kapitel 6

Fallstudie

In den vorhergehenden Kapiteln wurde eine Methode zur modellbasierten Exploration und Evaluation paralleler Firmware-Architekturen beschrieben, deren Konzeption durch die im Bereich der Automatisierungstechnik gegebenen Rahmenbedingungen motiviert ist. Eine Fallstudie soll nun am Beispiel typischer Problemstellungen die jeweilige Vorgehensweise beim Einsatz der Methode darstellen und evaluieren.

6.1 Gegenstand der Fallstudie

Diese Fallstudie hat die Anwendung der in der vorliegenden Arbeit entwickelten Methode zur Firmware-Parallelisierung in Form der in Abschnitt 5.3 beschriebenen *EEEPA*-Implementierung auf eine konkrete Automatisierungs-Firmware der Firma *Bosch Rexroth* zum Gegenstand[14]. Bei der Firmware der CNC-Plattform *Rexroth IndraMotion MTX* handelt es sich um ein Multitasking-System auf Basis des Echtzeitbetriebssystems *VxWorks 6.7* der Firma *Wind River*. Daher wurde zum Profiling im Rahmen der Fallstudie der in Abschnitt 5.3.2 beschriebene *Wind River System Viewer 3.2* genutzt. Die Anzahl der von der Firmware instanziierten Tasks ist dabei von der jeweiligen Konfiguration der Steuerung, wie beispielsweise der Anzahl der gemeinsam interpolierenden Achsverbünde und der Anzahl der SPS-Anwenderprogramme, abhängig. In den im Rahmen der Fallstudie betrachteten Lastprofilen in Form typischer Einsatzszenarien der Firmware (vgl. Tabelle 6.2) waren dies zwischen 63 und 78 Tasks, die sich den in Abschnitt 2.3.2 für eine numerische Steuerung definierten Funktionsbereichen zuordnen lassen. In der vorliegenden Version der Firmware war bereits eine Parallelisierung für zwei CPU-Kerne realisiert.

Die Hardware-Plattform, für die im Rahmen dieser Fallstudie parallele Firmware-Designs evaluiert werden sollen, ist die in Abbildung 6.1 dargestellte Automatisierungssteuerung *IndraControl L85* von *Rexroth*, die über einen Dual-Core-Prozessor vom Typ *Intel Core 2 Duo* und 1 Gigabyte *SDRAM* verfügt. Bei dieser Fallstudie wurde der in Abschnitt 5.4.5 begründeten Empfehlung, das Profiling bereits auf der Zielplattform der Firmware durchzuführen, gefolgt. Zu diesem Zweck standen 188 Megabyte des Arbeitsspeichers zur Verfügung.

[14] Ohne die Verständlichkeit oder die Aussagekraft der nachfolgenden Beschreibungen zu beeinträchtigen, wird im Rahmen der Fallstudie auf die Beschreibung implementierungstechnischer Details verzichtet, um die Eigentumsrechte von *Bosch Rexroth* am Quellcode der Firmware zu wahren.

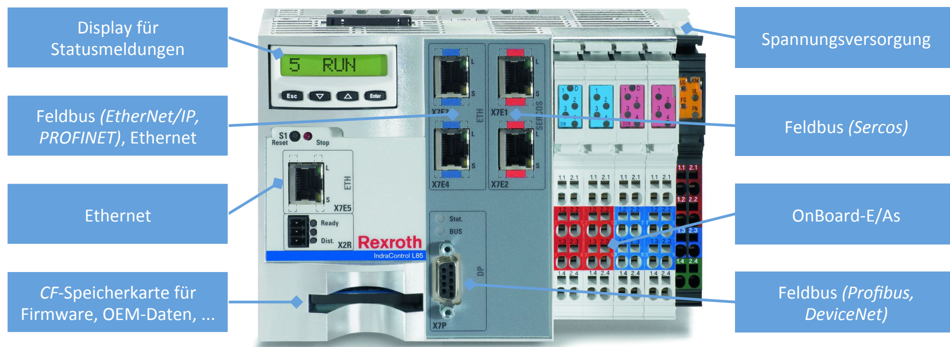


Abbildung 6.1: Automatisierungssteuerung *Rexroth IndraControl L85 (Bild: © Bosch Rexroth AG 2014)*

6.2 Einsatz der *EEEPA*-Toolchain

6.2.1 Parametrierung und Konfiguration

Im Rahmen der Methodenbeschreibung in Kapitel 5 wurden bezüglich zahlreicher Aspekte stets alternative Möglichkeiten der Konfiguration beschrieben, die ein auf den jeweiligen Anwendungsfall zugeschnittenes Vorgehen mittels einer entsprechenden Parametrierung der Toolchain ermöglichen. Die in dieser Fallstudie gewählten Parameter zeigt Tabelle 6.1. So wurde bei der Gewichtung der Interaktionskanten die kumulative Methode gewählt, da eine Analyse der auf Basis der Profiling-Daten generierten Firmware-Modelle gezeigt hatte, dass die Firmware hier keine Messages mit einem *Payload* von mehr als 4125 Byte nutzte. Auf der Hardware-Plattform *IndraControl L85* durchgeführte Benchmarks hatten jedoch bei diesem maximalen Message-Payload zwischen den modellierten Interaktionsmechanismen keine signifikanten Unterschiede bezüglich des Overheads einer Abwicklung über Kerngrenzen ergeben. Aus diesem Grund wurde auf eine interaktionsspezifische Gewichtung verzichtet.

Zur Aggregation der Metriken mehrerer Graphen wurde das in Formel 5.34 definierte Maximum gewählt, da bei der Evaluation paralleler Alternativen der vorliegenden Automatisierungs-Firmware die minimale Leistung, die unter den betrachteten Lastprofilen erzielt werden würde, im Fokus stand. Eine Aggregation auf Basis des arithmetischen Mittels (Formel 5.33) könnte hingegen infolge von Kompensationseffekten zwischen den Bewertungen der Lastprofile parallele Alternativen vorschlagen, die unter manchen der ausgewählten Lastprofile eine für den jeweiligen Anwendungsfall respektive Kunden inakzeptable Performanz aufweisen würden.

Bei der genetischen Optimierung wurde von der Möglichkeit der *PISA*-Schnittstelle Gebrauch gemacht, die Größe der Initialpopulation mit 100 Individuen unabhängig von der späteren Populationsgröße zu wählen, um die Exploration des Entwurfsraums mit möglichst diversitären Individuen zu starten. Der Wahl der Crossover- und Mutationswahrscheinlichkeit gingen zunächst umfangreiche Analysen zahlreicher Optimierungen mit diversen Parametrierungen voraus. Zudem wurde die Arbeit von Schaffer et al. [155] berücksichtigt, in der eine Crossover-Wahrscheinlichkeit zwischen 0,75 und 0,95 sowie eine Mutationswahrscheinlichkeit zwischen 0,005 und 0,01 als besonders geeignet bewertet wurde.

	Parameter	Wert	Referenz
†	Direkte Interaktionen	Events	Abschnitt 5.4.5
†	Indirekte Interaktionen	Semaphore, Messages	Abschnitt 5.4.5
†	Gewicht indirekter Interaktionen	normalisiert	Formel 5.6
†	Metrik der Lastverteilung	Spannweite	Formel 5.2
†	Gewichtung Interaktionskanten	kumulativ	Formel 5.8
†	Parallelisierungsaufwände	nicht berücksichtigt	Abschnitt 5.4.1/5.5.1
†, ‡	Aggregation der Graph-Metriken	Maximum	Formel 5.34
‡	Codeblock-Abhängigkeiten	global	Abschnitt 5.6.3
†, ‡	Größe der Initialpopulation	100 Individuen	Abschnitt 5.3.4
†, ‡	Anzahl der selektierten Eltern	20 Individuen	Abschnitt 5.3.4
†, ‡	Anzahl der Nachkommen	20 Individuen	Abschnitt 5.3.4
†, ‡	Crossover-Wahrscheinlichkeit	0,75	Abschnitt 5.6.2/5.6.3
†, ‡	Mutationswahrscheinlichkeit	0,01	Abschnitt 5.6.2/5.6.3
†, ‡	Selektor	*SPEA2*	Abschnitt 6.2.1
†, ‡	Terminierende Generationenzahl	10 000 (†) / 1000 (‡)	Abschnitt 5.6.1

Tabelle 6.1: Parametrierung der Toolchain im Rahmen der Fallstudie bei der Modellierung auf Systemebene (†) und auf Taskebene (‡)

Als Selektor für die genetische Optimierung wurde der Algorithmus *SPEA2* [195] gewählt, der eine Weiterentwicklung des *SPEA (Strength Pareto Evolutionary Algorithm)* [196] darstellt. Zur Selektion setzt dieser binäre Turniere ein, bei denen wiederholt je zwei Individuen zufällig ausgewählt werden und das jeweils geeignetere in die Menge der Elternindividuen übernommen wird, bis deren zuvor definierter Umfang erreicht ist [194]. Die Eignung der Individuen X wird dabei mittels folgender Faktoren bewertet:

- Die Relation der multikriteriellen Fitness eines Individuums $x \in X$ zu der Fitness der übrigen Individuen $X \setminus \{x\}$ wird bei der Selektion in Form von zwei Faktoren berücksichtigt: Die Anzahl der Individuen $X_d \subset X$, von denen x dominiert wird *(Dominance Rank)* und die Anzahl der Individuen $X'_d \subset X$, die von x dominiert werden *(Dominance Count)*.
- Um eine ausreichende Diversität der durch den Algorithmus explorierten Pareto-Optima sicherzustellen, wird darüber hinaus bei der Selektion die Individuendichte im Umfeld eines Individuums berücksichtigt. Dazu wird die Fitness eines Individuums um den reziproken Wert eines mittels eines *k-NN*-Algorithmus *(k-Nearest-Neighbor)* berechneten Distanzwerts reduziert.

Ein weiteres wesentliches Merkmal des *SPEA2* stellt die Verwaltung eines Individuenarchivs konstanter Größe dar. Dieses wird während der kompletten Laufzeit des Algorithmus als externer Individuenspeicher mitgeführt, wobei Individuen dieses Archivs auch als Eltern der nächsten Generation ausgewählt werden können. Das Archiv selbst wird in jeder Generation

	Maschine	Bearbeitung	Interpolationstakt	SPS-Zykluszeit
L_1	5-Achs-Fräsmaschine	Fräsen	2 ms	3 ms
L_2	5-Achs-Fräsmaschine	Fräsen	1 ms	3 ms
L_3	Rundtaktmaschine	u.a. Bohren, Fräsen	2 ms	4 ms
L_4	Rundtaktmaschine	u.a. Bohren, Fräsen	2 ms	8 ms
L_5	Rundtaktmaschine	u.a. Bohren, Fräsen	4 ms	8 ms
L_6	Laser-/Stanzmaschine	Laserschneiden	1 ms	4 ms
L_7	Laser-/Stanzmaschine	Blechstanzen	1 ms	4 ms
L_8	Laser-/Stanzmaschine	Blechstanzen	2 ms	4 ms

Tabelle 6.2: Lastprofile der Parallelisierung auf Systemebene

aktualisiert, indem nach jeder Reproduktion zunächst alle nichtdominierten Individuen aus der Vereinigungsmenge des bisherigen Archivs und der neuen Generation übernommen werden. Dies kann in folgenden Konstellationen resultieren: Ist die maximale Größe des Archivs noch nicht erreicht, werden zusätzlich auch dominierte Individuen in das Archiv übernommen, wobei eine Priorisierung entsprechend der zuvor definierten Werte des *Dominance Rank* und *Dominance Count* erfolgt. Ist die maximale Größe des Archivs hingegen überschritten, wird iterativ das Individuum mit der geringsten *k-NN*-Bewertung aus dem Archiv entfernt, bis die erforderliche Archivgröße erreicht ist. Auf diese Weise wird die Diversität des Archivs erhöht.

Zur Terminierung des genetischen Algorithmus wurde in dieser Fallstudie eine absolute Grenze von 10 000 Generationen auf Systemebene und 1000 Generationen auf Taskebene gewählt, um eine bessere Vergleichbarkeit der jeweils erzielten Ergebnisse zu ermöglichen.

6.2.2 Modellierung und Optimierung auf Systemebene

Bei der Exploration und Evaluation von Parallelisierungsalternativen auf Systemebene wurden Firmware-Modelle für acht repräsentative Lastprofile mit der in Tabelle 6.1 dargestellten Parametrierung generiert; eine Übersicht liefert Tabelle 6.2. Diese Lastprofile umfassen drei Werkzeugmaschinen mit jeweils spezifischen Bearbeitungsformen und typischen Konfigurationen des Interpolationstakts und der SPS-Zykluszeit:

- *Fräsmaschinen* zur Bearbeitung von Freiformflächen verarbeiten gemäß Abschnitt 3.1.2 meist komplexe NC-Programme und besitzen als Konsequenz eine hohe Auslastung im Bereich der Satzvorbereitung. Darüber hinaus führen berechnungsintensive Achstransformationen und der für die Erzielung einer hohen Oberflächengüte erforderliche geringe Interpolationstakt zu einer hohen Systemlast im Interpolator.
- *Rundtaktmaschinen* steuern eine Vielzahl an Achsen, die sich auf die einzelnen Bearbeitungsstationen der Maschine verteilen. Da in jedem Takt eine Sollwertvorgabe pro Achse zu berechnen ist, besitzt die Steuerung einer derartigen Maschine eine hohe Last seitens des Interpolators. Eine hohe Auslastung weist darüber hinaus in der Regel die integrierte SPS auf, da diese die Anpasssteuerung für die parallelen Bearbeitungsvorgänge aller Stationen realisiert.

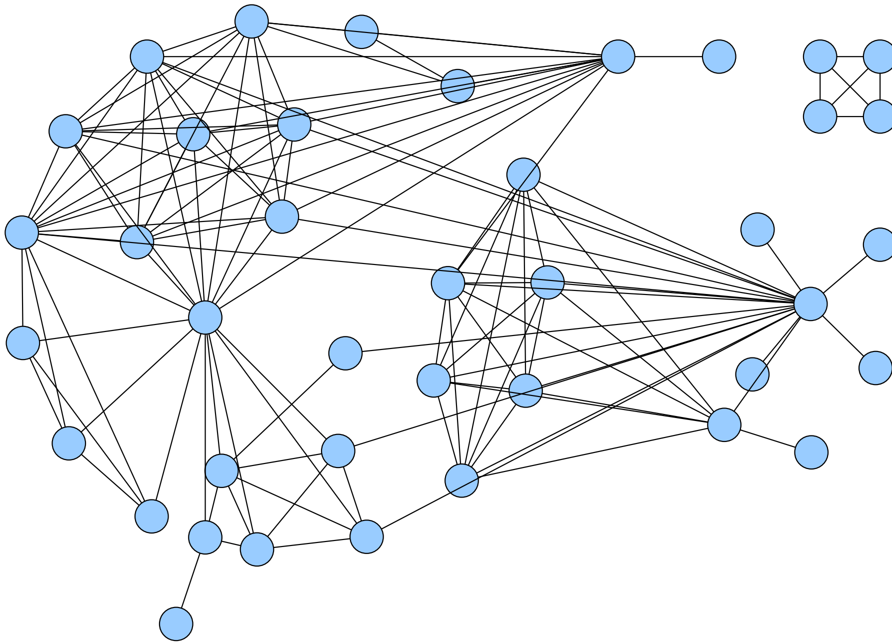


Abbildung 6.2: Qualitative Visualisierung des Task-Graphen des Lastprofils L_3 durch die Software *yEd* [191]. Zugunsten einer besseren Übersichtlichkeit werden Interaktionskanten mit weniger als 1000 Interaktionen und Task-Knoten, deren inzidente Interaktionskanten weniger als 1000 Interaktionen aufweisen, nicht dargestellt.

- *Laser-/Stanzmaschinen* müssen meist mit einem sehr geringen Interpolationstakt betrieben werden, um trotz der geforderten schnellen Verfahrbewegungen eine hohe Konturtreue realisieren zu können. Diese hochfrequente Ausführung generiert sowohl im Interpolator als auch in den Kommunikations-Tasks, welche die Sollwertvorgaben an den Feldbus-Controller übergeben, eine hohe Systemlast.

Somit hat die vorliegende Fallstudie besonders anspruchsvolle Anwendungsfälle einer CNC zum Gegenstand, da bei diesen aufgrund der entsprechend hohen Performanzanforderungen eine Parallelisierung für Multicore-Prozessoren den größten Nutzen verspricht.

Die auf der Plattform *IndraControl L85* für das Profiling verfügbaren 188 Megabyte *SDRAM* ermöglichten je nach Lastprofil Aufzeichnungen mit einer Dauer zwischen 11,59 s (L_2) und 23,81 s (L_6), wobei jeweils zwischen 5,15 Millionen (L_8) und 11,71 Millionen (L_1) Firmware-Ereignisse aufgezeichnet wurden. Bei den betrachteten Maschinen lag dabei die zur Abdeckung des typischen Firmware-Verhaltens maßgebliche maximale Periodendauer zyklisch durchgeführter Bearbeitungen meist im Bereich weniger Sekunden und war damit deutlich kleiner als die Aufzeichnungsdauer. Zu jeder der unter den zuvor dargestellten Lastprofilen gewonnenen Laufzeitaufzeichnungen wurde nun durch die *EEEPA*-Toolchain ein separates Modell in Form eines Task-Graphen extrahiert. Abbildung 6.2 stellt exemplarisch den Graphen des Szenarios L_3

Lastprofil	Last Core 0	Last Core 1	Lastdifferenz	Inter-Core-Interaktionen
L_1	59,54 %	98,32 %	38,78 %	29,23 %
L_2	67,91 %	96,36 %	28,45 %	28,94 %
L_3	95,25 %	89,15 %	6,10 %	30,29 %
L_4	90,96 %	72,81 %	18,15 %	28,20 %
L_5	73,59 %	73,56 %	0,03 %	28,89 %
L_6	51,44 %	88,97 %	37,53 %	32,07 %
L_7	56,10 %	90,24 %	34,14 %	31,79 %
L_8	47,96 %	87,58 %	39,62 %	31,84 %

Tabelle 6.3: Bewertung der derzeitigen Firmware-Parallelisierung der *IndraMotion MTX*. Das für die Gesamtbewertung des parallelen Designs relevante Maximum jedes Kriteriums ist hervorgehoben.

in qualitativer Form und somit ohne eine visuelle Differenzierung bezüglich der Kanten- und Kontengewichte dar. Dieses aus einer Laufzeitaufzeichnung von 21,54 s Dauer extrahierte Modell definiert 78 Tasks und Interrupts sowie 814 980 Interaktionen an 1404 Interaktionskanten. Damit wird deutlich, dass der Versuch einer manuellen Exploration effizienter Task-Verteilungen infolge der Firmware-Komplexität zum Scheitern verurteilt ist und stattdessen geeignete Problemlöser eingesetzt werden müssen.

Ein erstes Anwendungsszenario der *EEEPA*-Toolchain ist die Bewertung einer spezifischen Firmware-Parallelisierung hinsichtlich der in Abschnitt 5.4.1 definierten Kriterien. Dies gilt insbesondere für das bereits implementierte parallele Firmware-Design der hier modellierten Firmware-Version der *IndraMotion MTX*. Eine Leistungsbewertung der hier vorliegenden Task-Verteilung ist dabei von besonderem Interesse, um das Potential und den daraus resultierenden Handlungsbedarf weiterführender Parallelisierungen evaluieren zu können. Die für die Lastprofile L_1 bis L_8 ermittelten Bewertungen stellt Tabelle 6.3 dar, während Abbildung 6.3 diese in die Bewertungsmatrix aus der Lastverteilung und dem Anteil der Inter-Core-Interaktionen an den gesamten Task-Interaktionen einordnet. Zunächst wird deutlich, dass die Auslastung der CPU-Kerne in allen Fällen im Mittel stets unter 100 % liegt, so dass die modellierten Lastanteile als ausreichend korrekt angenommen werden können. Eine geringere Auslastung würde allerdings die Validität der Modelle entsprechend Abschnitt 5.4.5 noch erhöhen, da bei einer mittleren Core-Auslastung von bis zu 98,3 % (L_1) sporadische Überlastsituationen nicht zwingend ausgeschlossen werden können. Die Resultate zeigen darüber hinaus, dass sich bei der derzeitigen Parallelisierung die Lastprofile bezüglich des Anteils der Inter-Core-Interaktionen nur geringfügig unterscheiden, da die Werte stets innerhalb des Intervalls von 28,2 % bis 32,1 % liegen. Zugleich sind jedoch bei der Lastverteilung signifikante Unterschiede zu beobachten. Während die CPU-Kerne im Lastprofil L_5 eine nahezu identische Auslastung aufweisen, haben die Core-Auslastungen im Lastprofil L_8 eine Spannweite von 39,6 %. Dies unterstreicht den in Abschnitt 3.1.2 beschriebenen Einfluss des Maschinentyps und der Maschinenkonfiguration auf das Lastprofil einer Firmware und somit auch auf die Performanz einer spezifischen Firmware-Parallelisierung. Weiterhin wird deutlich, dass das hier gewählte parallele Design zwar zu

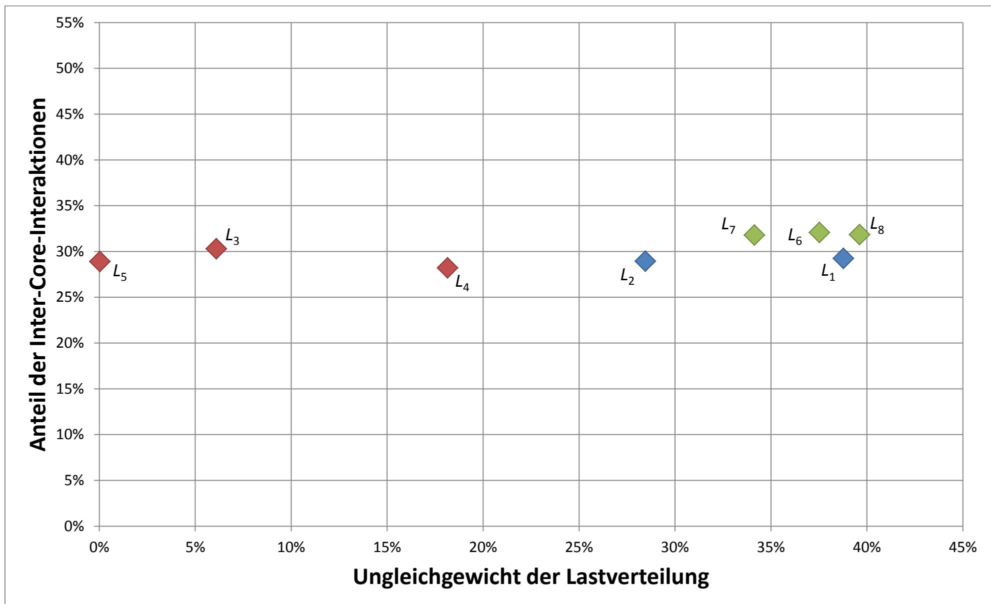


Abbildung 6.3: Einordnung der Parallelisierung, die in der modellierten Firmware-Version der *IndraMotion MTX* implementiert ist

durchaus akzeptablen Resultaten führt, zugleich aber mittels einer automatisierten Exploration des Entwurfsraums die Existenz möglicherweise besserer Alternativen evaluiert werden sollte.

Die mittels des Maximums aggregierten Resultate einer gemeinsamen Optimierung der Task-Verteilung für die Lastprofile L_1 bis L_8 zeigt Abbildung 6.4 in Form der explorierten Pareto-Optima. Zugleich liefert die Tabelle 6.4 eine detaillierte, nach Lastprofilen differenzierte Auflistung der prognostizierten Resultate für drei exemplarische Task-Verteilungen M_1, M_2 und M_3. Zum Vergleich ist die ebenfalls mittels des Maximums aggregierte Bewertung der aktuellen Parallelisierung der *IndraMotion MTX* in Abbildung 6.4 als Status Quo dargestellt, wobei deutlich wird, dass diese nicht Pareto-optimal ist. So wäre eine Reduzierung des maximalen Anteils der Inter-Core-Interaktionen gegenüber dem Status Quo um absolut 15,4 % auf 16,7 % bei einer zugleich um den Wert von 9,4 % geringeren Lastdifferenz von maximal 30,2 % möglich (M_3). Alternativ könnte die Lastdifferenz auf ein Minimum von 12,4 % reduziert werden, falls eine deutliche Zunahme der Inter-Core-Interaktionen auf 49,5 % akzeptiert werden würde (M_1). Zur Realisierung empfohlen ist hingegen die Verteilung M_2, da ausgehend von dieser eine geringe Verbesserung bei einer Metrik mit einer signifikanten Verschlechterung bei der anderen Metrik erkauft werden müsste. Diese Task-Verteilung würde sich gegenüber dem Status Quo durch eine um den Wert von 24 % geringere Lastdifferenz und einen um den Wert von 9,8 % geringeren Anteil an Inter-Core-Interaktionen auszeichnen. Die detaillierte Auflistung der Werte in Tabelle 6.4 zeigt, dass die Task-Verteilung M_2 den Anteil der Inter-Core-Interaktionen in allen Lastprofilen um einen einstelligen Prozentwert reduzieren würde, wobei die Laser-/Stanz-

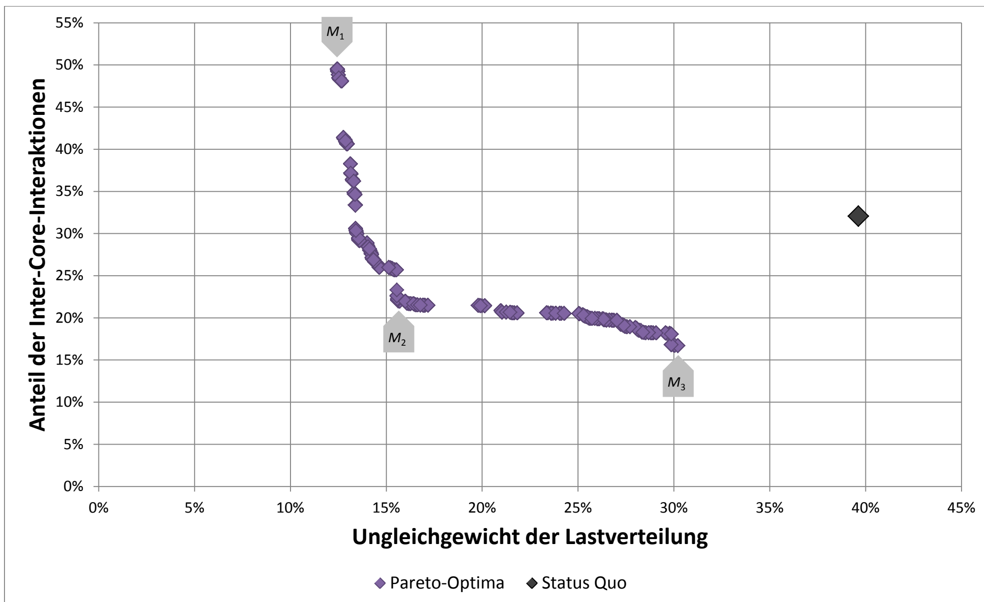


Abbildung 6.4: Einordnung der derzeitigen Parallelisierung der *IndraMotion MTX* im Vergleich zu den möglichen Pareto-Optima im Fall einer gemeinsamen Optimierung für die Lastprofile L_1 bis L_8

maschine (L_6 bis L_8) am stärksten profitieren würde. Dies würde auch für die Lastdifferenz mit einer Verbesserung gegenüber dem Status Quo in Höhe von bis zu 36,8 % gelten. Im Fall der Rundtaktmaschine (L_3 bis L_5) würde M_2 zwar den Anteil der Inter-Core-Interaktionen in allen Lastprofilen reduzieren, dabei aber in den Fällen L_3 und L_5 eine um den Wert von 6,7 % respektive 15,6 % größere Lastdifferenz aufweisen. Dies könnte vor allem im Lastprofil L_3 kritisch sein, da hier auf dem ersten CPU-Kern trotz einer Lastdifferenz von lediglich 12,8 % eine Auslastung von 99,7 % erzielt werden würde. Dass die Optimierung der Task-Verteilung eine derartige Lösung wählte, ist darauf zurückzuführen, dass die Metrik der Spannweite nur die Lastdifferenz, nicht aber die absolute Auslastung der CPU-Kerne berücksichtigt. Als Alternative zur Spannweite wurde in Abschnitt 5.4.1 die Maximallast definiert. In Kombination mit einer Graph-Aggregation mittels des Maximums würde eine derart parametrierte Optimierung darauf abzielen, die unter allen Lastprofilen maximal auftretende Auslastung eines CPU-Kerns zu reduzieren, während die Lastverteilung in den übrigen Fällen unberücksichtigt bliebe. Eine derartige Priorisierung kann durchaus begründet sein, sollte aber mit Bedacht gewählt werden.

Die in Tabelle 6.4 detailliert dargestellten Bewertungen der einzelnen Lastprofile bei der Task-Verteilung M_2 zeigen, dass bei einer zum Entwurfszeitpunkt definierten Task-Allokation die in Abschnitt 3.1.2 genannten Vorteile mit Kompromissen erkauft werden müssten. So wurde durch den genetischen Algorithmus die im schlechtesten Fall erzielte Performanz einer Task-Verteilung in den Lastprofilen L_1 bis L_8 optimiert und zugleich in Kauf genommen, dass unter keinem der Lastprofile das mögliche Optimum erzielt werden würde. Dies lässt sich verhindern, indem die Task-Verteilung nicht statisch zum Entwurfszeitpunkt sondern stets spezifisch für jeden

Lastprofil	Last Core 0	Last Core 1	Lastdifferenz	Inter-Core-Interaktionen
Bewertung des Pareto-Optimums mit der besten Lastverteilung (M_1)				
L_1	86,35 %	75,99 %	10,36 % (-28,42)	31,97 % (+2,74)
L_2	89,05 %	76,61 %	12,44 % (-16,01)	34,09 % (+5,15)
L_3	98,49 %	88,07 %	10,42 % (+4,32)	47,52 % (+17,23)
L_4	77,55 %	90,00 %	12,45 % (-5,70)	48,05 % (+19,85)
L_5	81,99 %	69,55 %	12,45 % (+12,42)	49,51 % (+20,62)
L_6	75,32 %	70,11 %	5,2 % (-32,33)	32,61 % (+0,54)
L_7	77,53 %	74,27 %	3,25 % (-30,89)	34,21 % (+2,42)
L_8	75,59 %	65,79 %	9,8 % (-29,82)	33,91 % (+2,07)
Bewertung des zur Implementierung empfohlenen Pareto-Optimums (M_2)				
L_1	88,49 %	73,85 %	14,64 % (-24,14)	21,58 % (-7,65)
L_2	85,00 %	80,65 %	4,34 % (-24,11)	22,11 % (-6,83)
L_3	99,66 %	86,89 %	12,77 % (+6,67)	22,19 % (-8,10)
L_4	75,99 %	91,56 %	15,56 % (-2,59)	22,25 % (-5,95)
L_5	83,56 %	67,98 %	15,58 % (+15,55)	20,72 % (-8,17)
L_6	73,09 %	72,34 %	0,75 % (-36,78)	22,14 % (-9,93)
L_7	75,63 %	76,17 %	0,55 % (-33,59)	22,11 % (-9,68)
L_8	74,07 %	67,32 %	6,75 % (-32,87)	22,10 % (-9,74)
Bewertung des Pareto-Optimums mit den wenigsten Inter-Core-Interaktionen (M_3)				
L_1	97,93 %	67,72 %	30,21 % (-8,57)	16,70 % (-12,53)
L_2	94,59 %	67,75 %	26,84 % (-1,61)	16,40 % (-12,54)
L_3	94,43 %	92,13 %	2,30 % (-3,80)	10,64 % (-19,65)
L_4	68,68 %	98,86 %	30,18 % (+12,03)	10,19 % (-18,01)
L_5	76,76 %	74,78 %	1,98 % (+1,95)	7,67 % (-21,22)
L_6	72,74 %	72,69 %	0,05 % (-37,48)	14,87 % (-17,20)
L_7	75,15 %	76,65 %	1,49 % (-32,65)	15,59 % (-16,20)
L_8	73,53 %	67,86 %	5,67 % (-33,95)	15,65 % (-16,19)

Tabelle 6.4: Resultate der Pareto-Optima M_1, M_2 und M_3 in den Lastprofilen L_1 bis L_8. Die jeweiligen, bei der Graph-Aggregation maßgeblichen Maxima sind hervorgehoben. Zudem ist zu jedem Wert die absolute Abweichung von dem im jeweiligen Lastprofil bei der derzeit realisierten Task-Verteilung erzielten Wert notiert.

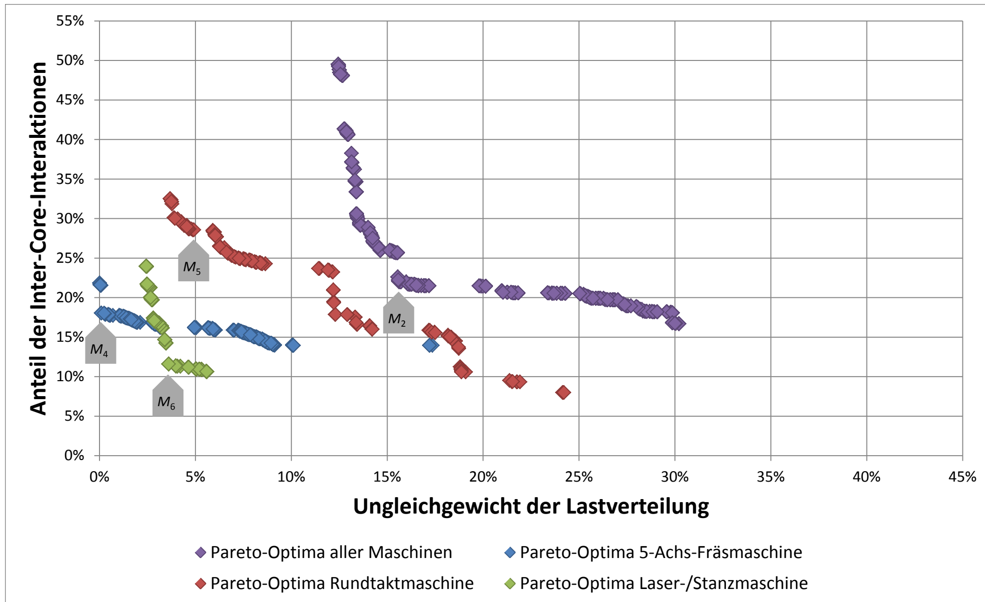


Abbildung 6.5: Pareto-Optima einer spezifisch für den jeweiligen Maschinentyp optimierten Task-Verteilung in Relation zu den Pareto-Optima einer für alle Lastprofile L_1 bis L_8 gemeinsam optimierten Task-Verteilung

Maschinentyp oder gar für jede Maschinenkonfiguration definiert wird. Die in der vorliegenden Arbeit beschriebene Methode bietet die Möglichkeit, das quantitative Potential eines derartigen Vorgehens für eine konkrete Firmware zu bewerten, wobei in dieser Fallstudie eine Beschränkung auf die maschinenspezifische Optimierung erfolgt. Zu diesem Zweck wurde für jeden der in Tabelle 6.2 dargestellten Maschinentypen eine spezifische Optimierung unter Berücksichtigung der jeweiligen Firmware-Modelle durchgeführt. Das bedeutet, dass beispielsweise im Fall der Rundtaktmaschine bei der Entwurfsraumexploration eine spezifische Task-Verteilung nur bezüglich der im schlechtesten Fall erzielten Performanz in den Lastprofilen L_3 bis L_5 bewertet wurde. Die für die drei Maschinentypen jeweils erzielten Pareto-Fronten stellt Abbildung 6.5 der zuvor beschriebenen Pareto-Front einer für alle Maschinen gemeinsamen Optimierung gegenüber. Die Ergebnisse bestätigen zunächst die Vermutung, dass die Pareto-Optima einer für alle Maschinen identischen Task-Verteilung durch die einer maschinenspezifischen Verteilung dominiert werden. So gibt es zu jedem Pareto-Optimum der maschinenübergreifenden Optimierung bei der maschinenspezifischen Optimierung pro Maschinentyp mindestens ein Pareto-Optimum, welches ersteres dominiert. Für eine detaillierte Analyse der Resultate wird nun erneut für jeden Maschinentyp ein zur Implementierung empfohlenes Pareto-Optimum M_4, M_5 und M_6 ausgewählt. Diese gesondert für jeden Maschinentyp durchgeführte Wahl des Pareto-Optimums ist insofern von Vorteil, als hierbei maschinenspezifische Präferenzen einfließen können. So wird für die Rundtaktmaschine im Wissen um die hohe Systemlast im Profil L_3 eine gleichmäßige Lastverteilung gegenüber einer Reduzierung der Inter-Core-Interaktionen

Lastprofil	Last Core 0	Last Core 1	Lastdifferenz	Inter-Core-Interaktionen
Bewertung des zur Implementierung empfohlenen Pareto-Optimums (M_4)				
L_1	81,20 %	81,13 %	0,07 % (-14,57)	17,32 % (-4,26)
L_2	82,78 %	82,78 %	0,00 % (-4,34)	18,04 % (-4,07)
Bewertung des zur Implementierung empfohlenen Pareto-Optimums (M_5)				
L_3	95,36 %	91,20 %	4,16 % (-8,61)	28,57 % (+6,38)
L_4	81,32 %	86,23 %	4,92 % (-10,64)	27,32 % (+5,07)
L_5	77,67 %	73,87 %	3,80 % (-11,78)	26,46 % (+5,74)
Bewertung des zur Implementierung empfohlenen Pareto-Optimums (M_6)				
L_6	74,49 %	70,95 %	3,54 % (+2,79)	11,03 % (-11,11)
L_7	77,70 %	74,10 %	3,61 % (+3,06)	11,61 % (-10,50)
L_8	68,89 %	72,50 %	3,61 % (-3,14)	11,56 % (-10,54)

Tabelle 6.5: Resultate der Pareto-Optima M_4, M_5 und M_6 in den jeweiligen Lastprofilen. Dabei ist jeweils die absolute Abweichung von dem Wert dargestellt, der im jeweiligen Lastprofil bei einer Task-Verteilung M_2 (vgl. Tabelle 6.4) erzielt werden würde.

präferiert, so dass die Lösung M_5 nahezu am linksseitigen Ende der Pareto-Front liegt. Im Fall der beiden anderen Maschinen basiert die Wahl der empfohlenen Lösungen hingegen wie auch beim Pareto-Optimum M_2 auf dem Steigungsverhalten der Pareto-Front. Die Bewertungen der einzelnen Lastprofile in den jeweiligen Pareto-Optima M_4, M_5 und M_6 stellt die Tabelle 6.5 denen gegenüber, die im jeweiligen Lastprofil unter M_2 erzielt werden würden. Dabei wird deutlich, dass vor allem die Fräsmaschine (L_1, L_2) von einer maschinenspezifischen Task-Verteilung M_4 profitieren würde. Gegenüber der Task-Verteilung M_2 könnte hier der Anteil der Inter-Core-Interaktionen in beiden Lastprofilen um einen Wert von ungefähr 4 % bei einer zugleich nahezu optimal balancierten Auslastung der CPU-Kerne reduziert werden. Im Fall der Rundtaktmaschine (L_3 - L_5) ließe sich die Lastdifferenz auf Kosten der Inter-Core-Interaktionen ebenfalls deutlich reduzieren, so dass auf diese Weise in dem unter der Task-Verteilung M_2 kritischen Lastprofil L_3 die maximale Auslastung eines CPU-Kerns auf 95,4 % gesenkt werden könnte. Bei der Laser-/Stanzmaschine (L_6 - L_8) wiederum könnte im Fall einer Realisierung der Task-Verteilung M_6 der Anteil der Inter-Core-Interaktionen um einen Wert zwischen 10 % und 11 % reduziert werden, ohne zugleich mehr als 3,6 % Lastdifferenz zwischen den CPU-Kernen zu verursachen. Dieser Wert wäre zwar etwas höher als im Fall von M_2, allerdings aufgrund der relativ geringen Systemlast durchaus tolerierbar.

Insgesamt zeigt somit die Methode für die Firmware der *IndraMotion MTX* ein zusätzliches Performanzpotential einer maschinenspezifischen gegenüber einer zum Entwurfszeitpunkt definierten Task-Verteilung. Die quantitative Verbesserung ist dabei abhängig von der jeweiligen Wahl der Pareto-Optima. So wäre im Fall der Lösungen M_4 bis M_6 meist eine Reduzierung der Werte im einstelligen und unteren zweistelligen Bereich möglich. Dem stehen allerdings die zusätzlichen Implementierungs- und Testaufwände einer maschinenspezifischen Adaption der Task-Verteilung gegenüber, so dass eine solche Entscheidung sorgfältig geprüft werden sollte.

6.2.3 Modellierung und Optimierung auf Taskebene

Neben der Verteilung der bestehenden Tasks wurde zudem die Task-Dekomposition als geeignete Strategie der Parallelisierung identifiziert. So zeigte eine detaillierte Analyse der zuvor generierten Firmware-Modelle, dass der die Interpolation realisierende monolithische Funktionsbereich der NC-Firmware mit einem Lastanteil zwischen 8,93 % (L_8) und 45,85 % (L_4) häufig den mit Abstand größten Anteil der Rechenkapazität beansprucht. Zudem hat der Interpolator als zentrale Komponente des zyklischen Firmware-Teils signifikanten Einfluss auf die quantitativen Leistungsdaten des Systems und weist hinsichtlich seines Quellcodes einen überschaubaren Umfang auf. Diese Faktoren begründen gemäß Abschnitt 3.1.2 die besondere Eignung des diesem Funktionsbereich zugrunde liegenden Codes für eine Dekomposition. Die vorliegende Fallstudie beschränkte sich dabei auf die Modellierung des Interpolators unter dem zuvor definierten Lastprofil L_1. Es gilt jedoch zu beachten, dass für eine tatsächliche Entscheidungsfindung hinsichtlich der statischen Dekomposition des Interpolators auch die übrigen, in Tabelle 6.2 dargestellten Lastprofile bei der Optimierung zu berücksichtigen wären, um eine für alle diese Anwendungsfälle geeignete Implementierung zu explorieren.

Unterstützt durch einen mit dem entsprechenden Code vertrauten Firmware-Entwickler wurden zunächst auf die in Abschnitt 5.5.3 beschriebene Weise insgesamt 122 Codeannotationen eingefügt, mittels derer 37 Codeblöcke, 22 Verzweigungsalternativen und ein Schleifenrumpf definiert wurden. Im Anschluss daran wurde auf der Steuerungs-Hardware *IndraControl L85* eine Laufzeitaufzeichnung unter dem Lastprofil L_1 mit einer Dauer von 20 ms erstellt, aus der sich infolge des zyklischen Charakters des entsprechenden NC-Programms ausreichend repräsentative Modelle extrahieren ließen. Aufgrund des in L_1 konfigurierten Interpolationstakts von 2 ms fanden in diesem Aufzeichnungsintervall 10 Ausführungszyklen des Interpolators statt, so dass aus der Aufzeichnung auch diese Anzahl an Codeblock-Graphen extrahiert werden konnte. Eine detaillierte Analyse der Graphen zeigt, dass diese jeweils die identische Zahl von 30 Codeblöcken umfassen, von denen je Graph 7 Codeblöcke mit je zwei Instanzen vertreten sind, da diese Bestandteil des Schleifenrumpfes sind. Somit wurden unter diesem Lastprofil 23 der insgesamt 37 im Quellcode definierten Codeblöcke durchlaufen. Alle extrahierten Graphen enthalten dabei die gleichen Codeblöcke in identischer Ausführungsreihenfolge, allerdings mit spezifischen Ausführungszeiten. Dies ist unter anderem eine Konsequenz von bedingten Ausführungen innerhalb der Codeblöcke, deren Konditionen in den einzelnen Interpolationszyklen unterschiedlich ausgewertet wurden.

Gemäß Abschnitt 5.5.4 erfolgt eine Modellierung auf Taskebene stets in semiautomatischer Weise, so dass Daten- und Kontrollabhängigkeiten durch einen erfahrenen Entwickler formalisiert werden mussten. Da das Ziel dieser Fallstudie darin besteht, statische Dekompositionen des Interpolators zum Entwicklungszeitpunkt zu evaluieren, fand dabei die in Abschnitt 5.5.4 beschriebene Variante, alle gemäß der Programmspezifikation unter einem beliebigen Lastprofil möglichen Abhängigkeiten zu modellieren, Verwendung. Dabei handelt es sich um insgesamt 213 Datenabhängigkeiten zwischen Codeblöcken und 23 Kontrollabhängigkeiten zwischen Codeblöcken und Konditionen. Bei der Ergänzung der zuvor extrahierten Graphen um die Interaktionen wurden schließlich jeweils 97 dieser Abhängigkeiten modelliert, während die übrigen Abhängigkeiten im konkreten Lastprofil L_1 nicht durchlaufene und somit in den Graphen auch nicht abgebildete Codeblöcke als Quelle oder Ziel des Datenflusses haben. Abbildung 6.6 stellt

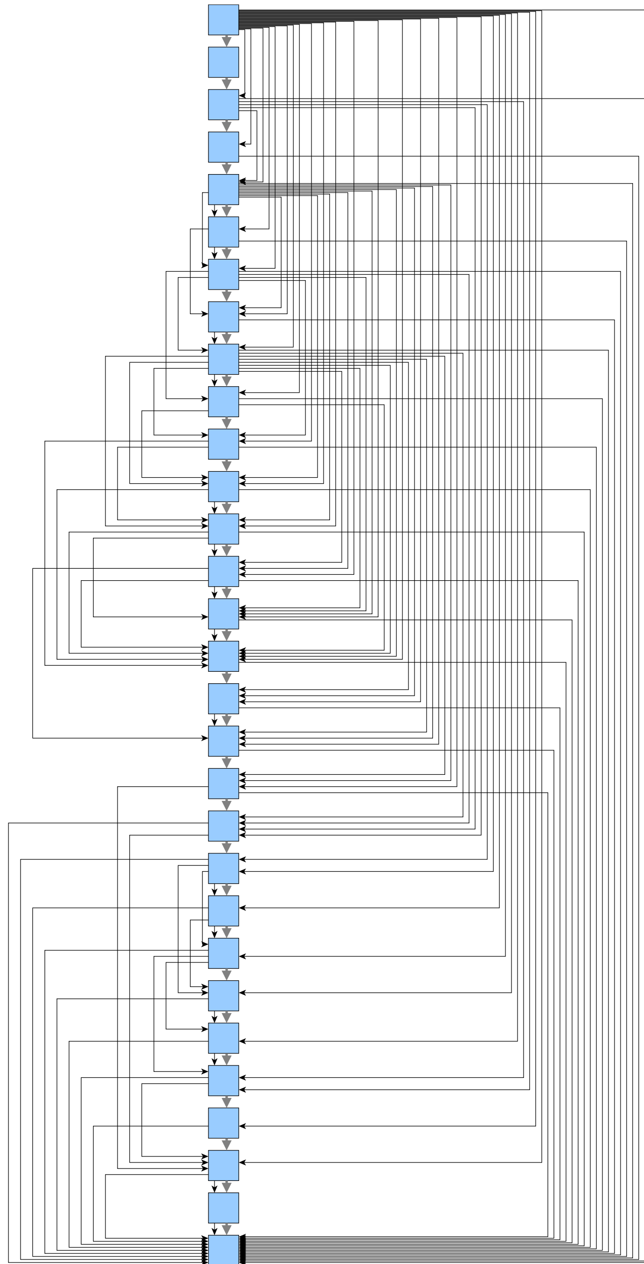


Abbildung 6.6: Qualitative Visualisierung des Interpolators in seiner derzeitigen sequentiellen Implementierung unter dem Lastprofil L_1 durch die Software *yEd* [191]. Kontrollflüsse sind als dicke graue und Datenflüsse als dünne schwarze Pfeile dargestellt.

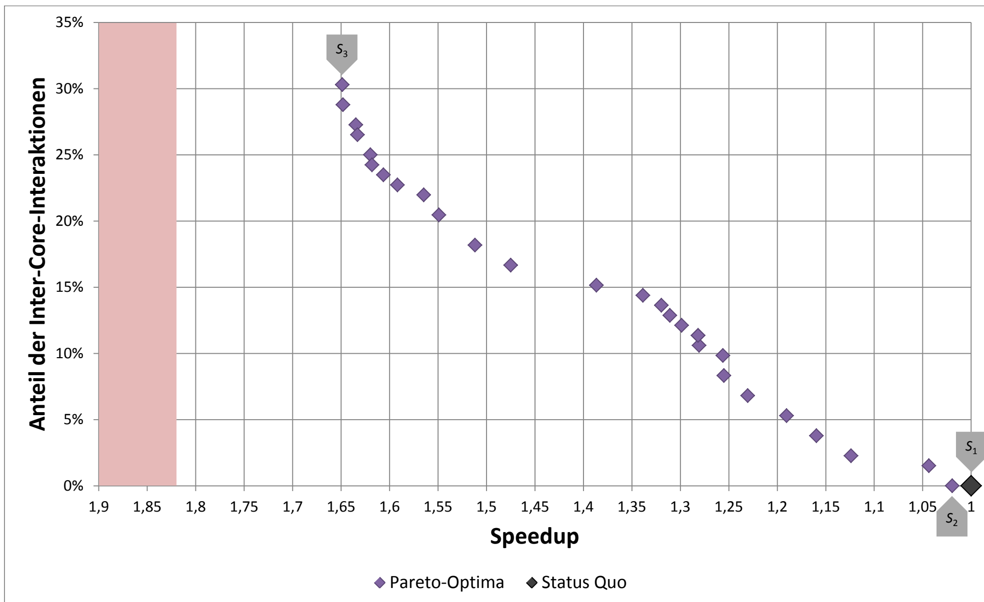


Abbildung 6.7: Darstellung der möglichen Pareto-Optima einer Dekomposition des Interpolators der *IndraMotion MTX* für zwei CPU-Kerne in Relation zur sequentiellen Implementierung S_1 im Lastprofil L_1. Der aufgrund der kritischen Pfade der Graphen generell nicht erreichbare Speedup-Bereich ist als rote Fläche dargestellt.

die qualitative Charakteristik des Interpolators in seiner derzeitigen sequentiellen Implementierung unter dem Lastprofil L_1 dar.

Im Rahmen dieser Fallstudie gilt es nun, geeignete Alternativen der Task-Dekomposition für die zwei CPU-Kerne der Steuerungsplattform *IndraControl L85* zu identifizieren. Zu diesem Zweck fand auf Basis der zuvor generierten Codeblock-Graphen durch die *EEEPA*-Toolchain eine automatisierte Exploration des Entwurfsraums mit der in Tabelle 6.1 dargestellten Parametrierung statt. Die Bewertung der Schedules erfolgte dabei stets auf Grundlage der Codeblock-Graphen, die zuvor unter dem Lastprofil L_1 der 5-Achs-Fräsmaschine generiert wurden. Die Resultate wurden dabei jeweils erneut mittels des Maximums in Form der unter den berücksichtigten Laufzeitmodellen des Interpolators im schlechtesten Fall erzielten Bewertung aggregiert. Die Einordnung der durch den genetischen Algorithmus explorierten Pareto-Optima in die Bewertungsmatrix aus dem erzielten Speedup und dem Anteil der Inter-Core-Interaktionen visualisiert die Abbildung 6.7 und stellt diese der derzeitigen Implementierung gegenüber. Der Schedule S_1 bildet dabei den sequentiellen Status Quo, bei dem somit innerhalb des Interpolators kein Datenaustausch zwischen den CPU-Kernen stattfindet. Dies gilt ebenso für den Schedule S_2, bei dem ein Codeblock ohne Interaktionen mit den übrigen Codeblöcken des Interpolators auf den zweiten CPU-Kern verlagert wird, was in einem Speedup-Faktor von mindestens 1,02 resultieren würde. Der größte Speedup ließe sich hingegen beim Schedule S_3 erzielen, unter dem bei den berücksichtigten Interpolatormodellen ein Faktor von mindestens 1,65

erreicht werden würde. Dieser Schedule ist damit der kürzeste, der im Rahmen dieser Optimierung für eine CPU mit zwei Kernen exploriert wurde. Die dabei realisierte Ausführung der Codeblöcke visualisiert Abbildung 6.8. Der Nachteil des Schedules S_3 besteht allerdings darin, dass bis zu 30,3 % der Interaktionen zwischen den Codeblöcken über die Grenzen der CPU-Kerne hinweg stattfinden würden, was auch Abbildung 6.8 verdeutlicht. So gibt es nahezu keinen Codeblock auf dem ersten CPU-Kern, der ohne eine unilaterale Synchronisation mit einem Codeblock des zweiten Kerns ausgeführt werden kann. Generell gilt es zu beachten, dass aus dem Speedup-Faktor des Interpolators allein noch keine Rückschlüsse auf eine mögliche Reduzierung des Interpolationstakts in dem konkreten Lastprofil abgeleitet werden können. So ist stets auch die übrige Auslastung des Systems zu berücksichtigen, da diese die für eine konfigurative Performanzsteigerung zur Verfügung stehende Idle-Zeit im System determiniert.

Unabhängig von der konkreten Zielplattform *IndraControl L85* stellt sich zudem die Frage, welcher Speedup für den Interpolator auf einer CPU mit mehr als 2 Kernen erzielt werden könnte. Aus diesem Grund wurden für die bereits zuvor gewählten Modelle des Lastprofils L_1 potentielle Task-Dekompositionen für Architekturen mit 4 und 8 CPU-Kernen exploriert. In keinem der beiden Fälle wurde dabei ein Schedule gefunden, dessen mindestens erzielter Speedup über dem Faktor 1,82 läge. Dies legt den Verdacht nahe, dass dieser Faktor durch den längsten kritischen Pfad der berücksichtigten Graphen bestimmt wird, was sich durch eine entsprechende Analyse bestätigen ließ. Diese obere Schranke ist in Abbildung 6.7 als schraffierter Bereich dargestellt und verdeutlicht so, dass sich unter dem Lastprofil L_1 auch beim Einsatz einer CPU mit einer beliebig hohen Anzahl an CPU-Kernen keine für alle berücksichtigten Graphen gültige untere Grenze für den Speedup erzielen ließe, die höher wäre. Beachtenswert ist dabei, dass für den Interpolator im Lastprofil L_1 bereits beim Einsatz von vier CPU-Kernen der maximale Speedup-Faktor von 1,82 erreicht werden könnte und somit ein Einsatz weiterer Kerne keinen Vorteil brächte. Die Realisierung eines solchen Schedules auf einer Quad-Core-CPU würde allerdings bereits bis zu 40,9 % der Kommunikation innerhalb des Interpolators über Kerngrenzen hinweg abwickeln. Als Konsequenz ist hier zumindest für das Lastprofil L_1 eine Dekomposition für vier CPU-Kerne nur dann empfehlenswert, wenn die Erzielung eines möglichst hohen Speedups in besonderer Weise im Fokus der Dekomposition steht.

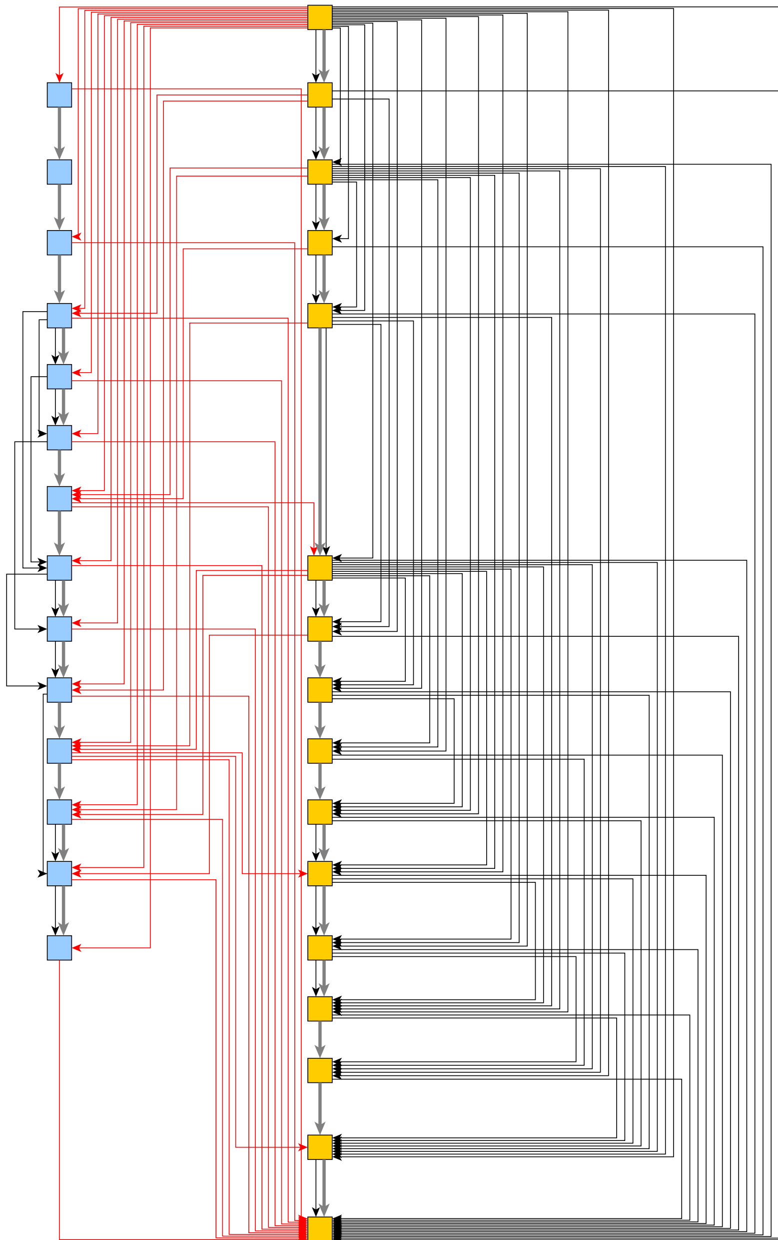


Abbildung 6.8: Qualitative Visualisierung des Interpolators unter dem Lastprofil L_1 im Schedule S_3. Codeblöcke auf Core 0 sind blau, Codeblöcke auf Core 1 gelb dargestellt. Kontrollflüsse sind als dicke graue, Datenflüsse innerhalb eines Cores als dünne schwarze und Datenflüsse zwischen Cores als dünne rote Pfeile dargestellt.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Beitrag zum Stand der Forschung und Technik

Die Zielstellung der vorliegenden Arbeit bestand darin, Konzepte, Strategien und Methoden, die einen effizienten Einsatz von Multicore-Prozessoren in der Steuerungstechnik ermöglichen, zu spezifizieren und zu bewerten. In diesem Kontext wurden die nachfolgenden Beiträge zum derzeitigen Stand der Forschung und Technik geleistet:

Evaluation einer Systemkonsolidierung in der Steuerungstechnik: Basierend auf den von Popek und Goldberg [140] allgemein für eine Virtualisierung geforderten Eigenschaften wurden für die Domäne der Steuerungstechnik die Anforderungen an eine integrierte Ausführung formuliert. Mit je einem virtualisierten und einem hybriden Konsolidierungsansatz wurden ferner typische Repräsentanten kommerzieller Konsolidierungslösungen für die Domäne eingebetteter Systeme vorgestellt. Diese wurden schließlich bezüglich des im jeweils integrierten Echtzeitsystem erzielten temporalen Verhaltens sowie der potentiellen Zuverlässigkeit und Sicherheit der Konsolidierung bewertet.

Spezifikation einer Methode zur Firmware-Parallelisierung: Es wurde eine anwendungsorientierte Methode spezifiziert, um für eine existierende Steuerungs-Firmware die hinsichtlich relevanter Kriterien Pareto-optimalen Alternativen einer Parallelisierung für homogene Multicore-Prozessoren zu explorieren. Den im Kontext der Steuerungstechnik als relevant klassifizierten Verfahren zur Erzielung einer Parallelität auf Taskebene in Form eines partitionierten Schedulings und einer Task-Dekomposition wird dabei mit einer jeweils spezifischen Vorgehensweise begegnet. In beiden Fällen werden zunächst geeignete Modelle der Firmware generiert, indem unter repräsentativen Lastprofilen durchgeführte dynamische Codeanalysen mit formalisiertem Expertenwissen kombiniert werden. Die dynamischen Analysen basieren auf einem Profiling der zuvor in manueller Weise instrumentierten Firmware während der Steuerung realer Maschinen. Die Exploration des Entwurfsraums paralleler Firmware-Architekturen erfolgt schließlich mittels genetischer Algorithmen und kann multiple Firmware-Modelle aggregieren, um die Pareto-Optimalität und Validität der Ergebnisse bezüglich mehrerer Lastprofile zu gewährleisten. Die Methodenspezifikation schloss mit der Empfehlung einer Vorgehensweise, um die Implementierung einer existierenden Firmware in der Weise zu adaptieren, wie es ein im Rahmen der zuvor beschriebenen Methode exploriertes Modell einer parallelen Firmware definiert. Neben einer jeweils separaten Handlungsempfehlung für ein partitioniertes Scheduling und eine Task-Dekomposition wurde in diesem Zusammenhang auch ein kombiniertes Vorgehen für eine Parallelisierung auf System- und Taskebene beschrieben.

Spezifikation von Firmware-Modellen: Um eine analytische Bewertung von Alternativen des partitionierten Schedulings und der Task-Dekomposition bezüglich der für eine Parallelisierung relevanten Kriterien zu ermöglichen, wurden auf Graphen basierende Firmware-Modelle und auf diese anwendbare Metriken spezifiziert.

Spezifikation einer Betriebssystem- und Codeinstrumentierung: Für ein Betriebssystem sowie eine imperative Implementierung erfolgte die Spezifikation einer Minimalinstrumentierung, auf Basis derer im Rahmen eines Profilings alle Laufzeitdaten, die für eine nachgelagerte Modellierung hinreichend sind, aufgezeichnet werden können. In diesem Zusammenhang wurde auch eine Vorgehensweise spezifiziert, um auf Basis dieser Daten partielle Firmware-Modelle in automatisierter Weise zu rekonstruieren.

Spezifikation einer Formalisierung und Modellierung von Kontroll- und Datenabhängigkeiten: Zur Formalisierung und Modellierung von Expertenwissen im Kontext einer Exploration alternativer Task-Dekompositionen wurde ein geeignetes Verfahren spezifiziert. Dieses ermöglicht die Definition probleminhärenter Kontroll- und Datenabhängigkeiten zwischen Anweisungsfolgen einer sequentiellen imperativen Implementierung, mittels derer die partiellen Firmware-Modelle schließlich erweitert werden.

Spezifikation einer Formalisierung und Modellierung von Implementierungsaufwänden: Die Zielstellung eines effizienten Vorgehens bei der Parallelisierung impliziert eine Berücksichtigung der im Rahmen einer Firmware-Adaption zu leistenden Implementierungsaufwände. Als Konsequenz erfolgte die Spezifikation eines Verfahrens zur Formalisierung und Modellierung entsprechender Aufwandsabschätzungen durch einen Experten.

Spezifikation der multikriteriellen Optimierung einer Aggregation multipler Modelle: Es wurde eine Methode zur Exploration der Pareto-Optima im Entwurfsraum eines partitionierten Schedulings und einer Task-Dekomposition spezifiziert. Diese basiert auf einer durch genetische Algorithmen geleisteten multikriteriellen Optimierung der zuvor generierten Firmware-Modelle hinsichtlich der bei einer Parallelisierung relevanten Kriterien. In diesem Kontext wurden die problemspezifischen Teile der genetischen Algorithmen unter Wahrung der Konformität mit der *PISA*-Schnittstellenspezifikation [21] definiert. Hervorzuheben sind dabei die eingesetzten Verfahren zur aggregierten Optimierung multipler Firmware-Modelle. Zudem wurden Maßnahmen definiert, um die Validität der explorierten Task-Dekompositionen hinsichtlich der Daten- und Kontrollabhängigkeiten multipler Firmware-Modelle zu gewährleisten.

Beschreibung einer prototypischen Implementierung: Mit der *EEEPA*-Toolchain wurde eine prototypische Implementierung der in der vorliegenden Arbeit spezifizierten Methode beschrieben. Durch die Nutzung definierter oder einfach adaptierbarer Schnittstellen konnte dabei eine mit geringem Aufwand realisierbare Einbindung in bestehende Entwicklungsprozesse ermöglicht werden. In diesem Zusammenhang wurde zudem eine Erweiterung des *GraphML*-Dateiformats definiert, welche eine detaillierte Speicherung von Firmware-Modellen ermöglicht und dabei von gängigen Programmen der Graph-Visualisierung interpretiert werden kann.

Dem Autor sind keine Arbeiten bekannt, die eine Evaluation von Konsolidierungsansätzen hinsichtlich eines Einsatzes in automatisierungstechnischen Steuerungen zum Inhalt haben. Nach bestem Wissen des Autors sind zudem die zuvor genannten Beiträge zur Firmware-Parallelisierung für Multicore-Prozessoren insbesondere in der hier geleisteten Kombination zu einer im Bereich der Steuerungstechnik und darüber hinaus anwendbaren Methode nicht Gegenstand bereits existierender Arbeiten auf diesem Gebiet.

7.2 Bewertung der Ergebnisse

Eine wesentliche Herausforderung bei der Evaluation der Systemkonsolidierung bestand darin, dass es sich bei den gewählten Repräsentanten der beiden Konsolidierungsansätze um nicht quelloffene Lösungen handelt. Dies begründet sich damit, dass quelloffene Lösungen insbesondere im Bereich der Software-Plattform automatisierungstechnischer Steuerungen traditionell kaum verbreitet sind [142], wenngleich seit einigen Jahren auch in der industriellen Steuerungstechnik Bestrebungen zum vermehrten Einsatz von *Open-Source*-Software erkennbar sind [29, 136]. Die fehlende Offenheit der Konsolidierungsansätze führte jedoch dazu, dass bei der Interpretation der im Rahmen der Evaluation erzielten Ergebnisse eine Ursachenforschung nur innerhalb gewisser Grenzen geleistet werden konnte. Die Evaluation auf einer Multicore-Hardware zeigte sowohl beim hybriden als auch beim virtualisierten Konsolidierungsansatz unter spezifischen Lastkonstellationen eine signifikante Reduzierung des Zeitdeterminismus des jeweils integrierten Echtzeitbetriebssystems. Während zudem die Reaktivität der integrierten Systeme beim virtualisierten Ansatz ebenfalls stark beeinträchtigt wird, ist hier beim hybriden Ansatz nahezu kein Einfluss feststellbar. Dies lässt sich auf die nichtvirtualisierte Ausführung von Echtzeitsystemen bei diesem Konsolidierungsansatz zurückführen. Diese fehlende Deprivilegierung verhindert jedoch nach aktuellem Stand der Technik eine formale Verifikation der Zuverlässigkeit und Sicherheit einer derartigen Konsolidierung, welche unter anderem zum Nachweis einer sicheren Isolation der integrierten Systeme erforderlich wäre. Unter Berücksichtigung aller Ergebnisse kann eine virtualisiert oder hybrid realisierte Systemkonsolidierung auf aktuellen Multicore-Architekturen schließlich nur für Systeme mit weichen Echtzeitanforderungen empfohlen werden. Werden an eine in integrierter Weise ausgeführte Applikation hingegen harte Echtzeitanforderungen gestellt, so muss durch die Konsolidierungsansätze mindestens die Einhaltung definierter oberer Zeitschranken garantiert werden, damit diese beim Systementwurf berücksichtigt werden können.

Die Anwendbarkeit der in der vorliegenden Arbeit spezifizierten Methode zur Firmware-Parallelisierung konnte auf Basis der prototypischen *EEEPA*-Implementierung im Rahmen einer Fallstudie, die eine reale Steuerungs-Firmware zum Gegenstand hatte, nachgewiesen werden. Mittels eines Profilings bei der Steuerung repräsentativer Maschinentypen mit jeweils variierenden Konfigurationen wurden dabei zunächst hinsichtlich aller Lastprofile Pareto-optimale Alternativen eines partitionierten Schedulings dieser Firmware exploriert und evaluiert. Zudem konnte der Umfang quantifiziert werden, in dem eine maschinenspezifisch optimierte Task-Verteilung Vorteile gegenüber einer für alle Lastprofile gemeinsam durchgeführten Optimierung bieten würde. Für den Interpolator der Firmware wurden schließlich auf Basis eines ausgewählten Lastprofils Pareto-optimale Alternativen der Task-Dekomposition exploriert, für deren Implementierung ein signifikanter Speedup prognostiziert werden konnte. Zudem wurde für diesen Anwendungsfall die Anzahl der CPU-Kerne ermittelt, die zum Erreichen des maximalen, durch den kritischen Pfad des Interpolators determinierten Speedups nötig wäre.

Im Rahmen der Spezifikation wurden auch die Grenzen der in dieser Arbeit entwickelten Parallelisierungsmethode diskutiert. Hervorzuheben ist hierbei die Beschränkung auf eine Minimalinstrumentierung der Firmware, um den durch das Profiling generierten Overhead und den Aufwand der Instrumentierung in einem akzeptablen Rahmen zu halten. Dies wiederum erfordert bei der automatisierten Modellrekonstruktion die Anwendung approximativer Verfahren. Ferner lassen sich auch die zu erwartenden Einflüsse einer tatsächlich umgesetzten

Parallelisierung nur in begrenztem Umfang in den Modellen abbilden. So können beispielsweise die im Rahmen der Implementierung eines partitionierten Schedulings oder einer Task-Dekomposition noch zu leistenden Adaptionen am Quellcode hier nicht berücksichtigt werden. Zusammenfassend liefert die in der vorliegenden Arbeit entwickelte Methode somit einen anwendungsorientierten Ansatz zur modellbasierten Exploration effizienter Firmware-Parallelisierungen. Eine umfassendere Instrumentierung stellt eine Maßnahme dar, um hinsichtlich ausgewählter Aspekte eine höhere Modellierungstiefe zu erreichen. Als Konsequenz könnte der Abstraktionsgrad der Modellierung reduziert werden, wodurch sich eine potentiell höhere Prädiktionsgüte erzielen ließe. Dieser Vorteil müsste allerdings mit einem signifikant höheren Instrumentierungsaufwand sowie Laufzeit-Overhead erkauft werden, wobei letzterer wiederum die Systemcharakteristik beeinflussen und so die Modellvalidität beeinträchtigen könnte.

7.3 Ausblick

Die in der vorliegenden Arbeit entwickelte Methode spezifiziert vielfältige Konzepte, um den im Rahmen einer Firmware-Parallelisierung entstehenden Herausforderungen in umfassender Weise zu begegnen. Gleichwohl existieren in diesem Kontext noch weitere Themenstellungen, welche das Potential für zukünftige Betrachtungen bergen.

Da sich die Charakteristik einer Steuerungs-Firmware nur im Rahmen einer Interaktion mit einer realen Maschine vollständig manifestiert, wurde in der vorliegenden Arbeit eine statische oder eine auf einer emulierten Ausführung basierende dynamische Analyse als Alternative zu einem Profiling verworfen. Zugleich stellt jedoch jeglicher Datenaustausch zwischen Tasks oder Anweisungen, der nicht auf dem Aufruf von Betriebssystem-Syscalls basiert, eine Herausforderung für die einem Profiling vorausgehende Instrumentierung dar. So müsste der Firmware-Code derart instrumentiert werden, dass alle Lese- und Schreibzugriffe auf gemeinsame Speicherbereiche als Ereignis aufgezeichnet würden. Dies hätte allerdings einen sehr hohen Aufwand und eine große Unsicherheit zur Konsequenz, weil alle entsprechenden Stellen zuverlässig identifiziert und adaptiert werden müssten. Aus diesem Grund berücksichtigt die in der vorliegenden Arbeit spezifizierte Methode bei der automatisierten Modellierung auf Systemebene derartige unsynchronisierte Lese- und Schreibzugriffe nicht. Auf Taskebene wiederum wird zwar aufgrund der Beschränkung auf probleminhärente Abhängigkeiten ohnehin nur eine semiautomatische Modellierung der Datenflüsse geleistet. Allerdings ließe sich dieser Prozess potentiell vereinfachen, wenn die hinsichtlich ihrer Probleminhärenz zu bewertenden Abhängigkeiten statt in manueller in automatisierter Weise rekonstruiert würden. Als Gegenstand zukünftiger Arbeiten empfiehlt sich somit die Entwicklung geeigneter Verfahren, die unter den zuvor genannten Rahmenbedingungen mit einem möglichst geringen manuellen Modellierungs- und Instrumentierungsaufwand unsynchronisierte Datenflüsse im Rahmen eines Profilings aufzeichenbar machen. Dies könnte beispielsweise eine auf einer statischen Codeanalyse basierende automatisierte Firmware-Instrumentierung leisten. Verwandte Arbeiten mit einer dahingehenden Vorgehensweise nennt Abschnitt 5.2.

Weiteres Potential besitzt die spezifizierte Methode trotz der Beschränkung auf homogene Multicore-Architekturen hinsichtlich der Einbeziehung eines detaillierteren Hardware-Modells in die Entwurfsraumexploration. Bislang wurden Inter-Core-Interaktionen zwischen allen CPU-Kernen stets in identischer Weise bewertet. Diese Annahme kann für die Mehrzahl der derzeit

verbreiteten Multicore-Prozessoren insofern als valide gelten, als deren Caches meist entweder für jeden Kern exklusiv oder für alle Kerne gemeinsam zur Verfügung stehen. Insbesondere für zukünftige Multicore-Prozessoren sind allerdings Architekturen angedacht, die eine Kombination exklusiver und gemeinsamer Caches aufweisen. Ein entsprechendes hypothetisches Beispiel stellt eine CPU mit 16 Kernen dar, von denen jeder einen eigenen L1-Cache besitzt und jeweils vier Kerne gemeinsam einen von vier L2-Caches nutzen [14]. Somit gäbe es Ebenen in der Speicherhierarchie, deren Caches jeweils nur durch eine Gruppe von CPU-Kernen gemeinsam genutzt würden. CPU-Kerne, die nicht Bestandteil der gleichen Gruppe wären, könnten Daten hingegen nur über einen Cache höherer Ebene oder über den Hauptspeicher austauschen, was mit einem entsprechend größeren Overhead verbunden wäre. Mittels eines entsprechenden Hardware-Modells könnten derartige Szenarien bei der Bewertung einer Parallelisierung hinsichtlich des Anteils der Inter-Core-Interaktionen berücksichtigt werden. Dabei müsste die Bewertung einer Inter-Core-Interaktion zwischen verschiedenen CPU-Kernen zugeordneten Tasks berücksichtigen, auf welcher Ebene der Speicherhierarchie diese Kerne über einen gemeinsamen Cache verfügen.

Eine operationale Validierung der spezifizierten Parallelisierungsmethode ist aus den in Abschnitt 5.1.2 genannten Gründen nicht Gegenstand der vorliegenden Arbeit. Eine noch offene Themenstellung stellt somit die Definition einer generischen Vorgehensweise dar, um eine solche in spezifischen Anwendungsfällen leisten zu können. So könnte eine operationale Validierung entsprechend Abschnitt 5.1.2 beispielsweise mittels eines Vergleichs der modellbasierten Prädiktionen und den Eigenschaften eines entsprechend der Parametrierung implementierten realen Systems erfolgen. Ein korrektes Vorgehen bei der Validierung der in der vorliegenden Arbeit spezifizierten Methode erfordert jedoch zwingend, dass zur Rekonstruktion der Merkmale des parallelisierten Realsystems keine Mechanismen eben dieser Methode eingesetzt werden. Stattdessen muss die Beobachtung der zu vergleichenden Eigenschaften im parallelisierten System mittels einer vollständig anderen Vorgehensweise erfolgen, die jedoch während der Steuerung einer realen Maschine anwendbar sein muss.

In der vorliegenden Arbeit wurde ein zum Entwicklungszeitpunkt definiertes partitioniertes Scheduling als hinsichtlich der zu leistenden Implementierungsaufwände optimale Strategie zur parallelen Task-Ausführung auf einer Multicore-CPU bewertet. Zugleich ist aber zu erwarten, dass die dabei erreichbare Performanz in der Regel einen Kompromiss darstellt, da eine für das jeweilige Lastprofil optimierte Task-Verteilung üblicherweise bessere Ergebnisse erzielen würde. Dies bestätigte die in Kapitel 6 beschriebene Fallstudie mindestens für den Fall einer maschinenspezifischen Optimierung. Als Alternative zu dieser Vorgehensweise könnte im Rahmen der Entwicklung auf Basis der abgeschätzten Parallelisierungsaufwände auch eine Partition der Task-Menge generiert werden, die sich durch folgende Eigenschaften auszeichnet: Sind zwei Tasks in unterschiedlichen Mengenelementen der Partition enthalten, so wird deren echt parallele Ausführbarkeit in entsprechender Weise sichergestellt und getestet. Sind zwei Tasks hingegen im gleichen Mengenelement der Partition enthalten, so wird deren echt parallele Ausführung unter allen Umständen als ausgeschlossen angenommen und muss somit nicht gewährleistet werden. In Kapitel 5.6.2 wurde bereits definiert, in welcher Weise eine Entwurfsraumexploration unter der Vorgabe von Task-Gruppen, deren Elemente stets dem gleichen CPU-Kern zuzuordnen sind, erfolgen kann. Würden nun die zuvor genannten Mengenelemente der Partition als Task-Gruppen der Exploration definiert, so wäre für alle durch den genetischen Algorithmus explorierten Varianten eines partitionierten Schedulings eine sichere parallele Ausführung gewährleistet.

Als Konsequenz ließe sich nun eine Verlagerung der in der vorliegenden Arbeit spezifizierten Parallelisierungsmethode von der Firmware-Entwicklung in die Maschineninbetriebnahme realisieren [32, 34]. Im Rahmen einer Kalibrierungsphase der Maschinensteuerung müssten dabei zunächst die erforderlichen Laufzeitdaten aufgezeichnet werden. Die Entwurfsraumexploration würde schließlich unter Berücksichtigung der zuvor definierten Task-Gruppen erfolgen und keine Implementierungsaufwände mehr bewerten. Zu beachten ist, dass der Entwurfsraum als Folge der Gruppenbildung eingeschränkt wäre und somit nur ausgewählte Varianten der Task-Verteilung umfassen würde. Dem Inbetriebnahmeingenieur würden schließlich die explorierten Pareto-optimalen Alternativen der Task-Verteilung präsentiert, so dass eine geeignete Variante ausgewählt und als zukünftige Konfiguration der Steuerung für dieses konkrete Lastprofil übernommen werden könnte. Auf diese Weise ließe sich potentiell eine für das jeweilige Lastprofil adäquatere Parallelisierung realisieren, wobei die optimale Task-Verteilung gegebenenfalls nicht Bestandteil des zuvor eingeschränkten Entwurfsraums wäre. Zugleich wären allerdings die a priori zu investierenden Aufwände zur Sicherstellung der parallelen Ausführbarkeit der Task-Gruppen nicht unbeträchtlich, so dass bei einer derartigen Strategie die als wesentliches Zielkriterium der vorliegenden Arbeit formulierte Effizienz bei der Nutzung von Multicore-Prozessoren infrage zu stellen wäre.

Literaturverzeichnis

[1] Abramson, Darren, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu und John Wiegert: *Intel Virtualization Technology for Directed I/O*. Intel Technology Journal, 10(3):179–192, August 2006.

[2] Adämmer, Frank: *Evaluierung echtzeitfähiger Konsolidierungslösungen für Multicore-Architekturen in der Steuerungstechnik*. Diplomarbeit, Fachhochschule Münster, Oktober 2009.

[3] Advanced Micro Devices, Inc. (AMD): *AMD Athlon 64 X2 Dual-Core Processor Product Data Sheet, Revision 3.10*, Januar 2007.

[4] Advanced Micro Devices, Inc. (AMD): *AMD64 Architecture Programmer's Manual Volume 2: System Programming, Revision 3.18*, Mai 2011.

[5] Agesen, Ole, Alex Garthwaite, Jeffrey Sheldon und Pratap Subrahmanyam: *The Evolution of an x86 Virtual Machine Monitor*. ACM SIGOPS Operating Systems Review, 44(4):3–18, Dezember 2010.

[6] Aguirre, Hernán E. und Kiyoshi Tanaka: *Random Bit Climbers on Multiobjective MNK-Landscapes: Effects of Memory and Population Climbing*. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 88-A(1):334–345, 2005.

[7] Alkassar, Eyad, Mark A. Hillebrand, Wolfgang J. Paul und Elena Petrova: *Automated Verification of a Small Hypervisor*. In: *Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'10, Seiten 40–54, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] Andersen, Lars Ole: *Program Analysis and Specialization for the C Programming Language*. Dissertation, DIKU, Universität Kopenhagen, Kopenhagen, Mai 1994.

[9] Antonisse, Jim: *A New Interpretation of Schema Notation that Overturns the Binary Encoding Constraint*. In: Schaffer, J. D. (Herausgeber): *Proceedings of the Third International Conference on Genetic Algorithms*, Seiten 86–91, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[10] Assenmacher, Walter: *Deskriptive Statistik*. Springer-Verlag, Berlin, Heidelberg, 4. Auflage, 2010.

[11] Bäck, Thomas, Ulrich Hammel und Hans-Paul Schwefel: *Evolutionary Computation: Comments on the History and Current State*. IEEE Transactions on Evolutionary Computation, 1(1):3–17, April 1997.

[12] Baert, Rogier, Erik Brockmeyer, Sven Wuytack und Thomas J. Ashby: *Exploring Parallelizations of Applications for MPSoC Platforms using MPA*. In: *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, Seiten 1148–1153, Leuven, Belgien, April 2009. European Design and Automation Association.

[13] Baker, Theodore P.: *What to Make of Multicore Processors for Reliable Real-Time Systems?* In: *Proceedings of the 15th Ada-Europe international conference on Reliable Software Technologies*, Ada-Europe '10, Seiten 1–18, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] Balasubramonian, Rajeev, Norman P. Jouppi und Naveen Muralimanohar: *Multi-Core Cache Hierarchies*. Nummer 17 in *Synthesis Lectures on Computer Architecture*. Morgan & Claypool Publishers, San Rafael, CA, USA, 2011.

[15] Ball, Thomas: *The Concept of Dynamic Analysis*. In: Nierstrasz, Oscar und Michel Lemoine (Herausgeber): *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '99, Seiten 216–234, London, UK, 1999. Springer-Verlag.

[16] Balzert, Helmut: *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Spektrum Akademischer Verlag, Heidelberg, 3. Auflage, 2009.

[17] Barham, Paul, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt und Andrew Warfield: *Xen and the Art of Virtualization*. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, Seiten 164–177, New York, NY, USA, 2003. ACM.

[18] Bastian, Mathieu, Sebastien Heymann und Mathieu Jacomy: *Gephi: An Open Source Software for Exploring and Manipulating Networks*. In: *Proceedings of the Third International Conference on Weblogs and Social Media*, ICWSM-2009, Menlo Park, CA, USA, 2009. The AAAI Press.

[19] Bastoni, Andrea: *Towards the Integration of Theory and Practice in Multiprocessor Real-Time Scheduling*. Dissertation, Universität Rom, Rom, Mai 2011.

[20] Bengel, Günther, Christian Baun, Marcel Kunze und Karl Stuckey: *Masterkurs Parallele und Verteilte Systeme*. Vieweg+Teubner Verlag, GWV Fachverlage GmbH, Wiesbaden, 1. Auflage, 2008.

[21] Bleuler, Stefan, Marco Laumanns, Lothar Thiele und Eckart Zitzler: *PISA - A Platform and Programming Language Independent Interface for Search Algorithms*. In: Fonseca, Carlos M., Peter J. Fleming, Eckart Zitzler, Kalyanmoy Deb und Lothar Thiele (Herausgeber): *Evolutionary Multi-Criterion Optimization (EMO 2003)*, Band 2632 der Reihe *Lecture Notes in Computer Science*, Seiten 494–508, Berlin, 2003. Springer.

[22] Blumofe, Robert D., Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall und Yuli Zhou: *Cilk: An Efficient Multithreaded Runtime System*. In: *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, Seiten 207–216, New York, NY, USA, 1995. ACM.

[23] Bolosky, William J. und Michael L. Scott: *False Sharing and its Effect on Shared Memory Performance*. In: *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, Seiten 57–71, Berkeley, CA, USA, 1993. USENIX Association.

[24] Bosch Rexroth AG, Lohr am Main: *Rexroth IndraMotion MTX 12VRS Systembeschreibung*, 1. Auflage, Juni 2011.

[25] Bosch Rexroth AG, Lohr am Main: *Rexroth IndraControl L25, L45, L65 und L85 Steuerungen*, 1. Auflage, Januar 2012.

[26] Brandenburg, Björn B., Hennadiy Leontyev und James H. Anderson: *An Overview of Interrupt Accounting Techniques for Multiprocessor Real-Time Systems*. Journal of Systems Architecture - Embedded Systems Design, 57(6):638–654, 2011.

[27] Brandes, Ulrik, Markus Eiglsperger, Ivan Herman, Michael Himsolt und M. Scott Marshall: *GraphML Progress Report, Structural Layer Proposal*. In: Mutzel, Petra, Michael Jünger und Sebastian Leipert (Herausgeber): *Graph Drawing - 9th International Symposium, GD 2001, Wien, Östereich*, Band 2265 der Reihe *Lecture Notes in Computer Science*, Seiten 501–512, Heidelberg, 2001. Springer Verlag.

[28] Brandes, Ulrik, Markus Eiglsperger und Jürgen Lerner: *GraphML Primer*. URL: `http://graphml.graphdrawing.org/primer/graphml-primer.html` (abgerufen am 30.12.2013).

[29] Brecher, Christian, Alexander Verl, Armin Lechler und Michael Servos: *Open Control Systems: State of the Art*. Production Engineering, 4(2-3):247–254, Mai 2010.

[30] Bregenzer, Jürgen: *Automation Firmware Modeling to Explore and Evaluate Efficient Parallelization Alternatives*. In: *Bosch Conference on Systems and Software Engineering (BoCSE)*, Ludwigsburg, November 2011. Robert Bosch GmbH.

[31] Bregenzer, Jürgen: *GraphML-based Exploration and Evaluation of Efficient Parallelization Alternatives for Automation Firmware*. In: Brandes, Ulrik und Sabine Cornelsen (Herausgeber): *Proceedings of the 18th International Symposium on Graph Drawing (GD 2010)*, Band 6502 der Reihe *Lecture Notes in Computer Science*, Seiten 389–390, Berlin, 2011. Springer.

[32] Bregenzer, Jürgen: *Parallelisierung von Automatisierungs-Steuerungen für Multicore-Prozessoren*. In: Bender, Klaus, Walter Schumacher und Alexander Verl (Herausgeber): *SPS/IPC/DRIVES 2011 - Elektrische Automatisierung - Systeme und Komponenten, Fachmesse & Kongress, 22.-24. November 2011, Nürnberg, Tagungsband*, Seiten 243–251, Berlin, Offenbach, 2011. VDE Verlag GmbH.

[33] Bregenzer, Jürgen und Frank Adämmer: *Evaluation of Integration Approaches in common COTS Hypervisors for Use in Industrial Automation Controllers*. In: Engel, Michael, Rüdiger Kapitza, Jörg Nolte, Hans P. Reiser und Olaf Spinczyk (Herausgeber): *Proceedings of the Workshop on Isolation and Integration for Dependable Systems*, IIDS 2010, Paris, Frankreich, 2010.

[34] Bregenzer, Jürgen, Jens Brühl und Reiner Kolla: *Einsatz von Multicore-Prozessoren in numerischen Steuerungen - Eine Übersicht effizienter Nutzungsstrategien*. wt Werkstattstechnik online, 101(5):372–377, Mai 2011.

[35] Bregenzer, Jürgen und Julian Hartmann: *An Approach towards Automation Firmware Modeling for an Exploration and Evaluation of Efficient Parallelization Alternatives*. In: *Proceedings of the 2011 6th International Symposium on Parallel Computing in Electrical Engineering*, PARELEC '11, Seiten 13–18, Washington, DC, USA, 2011. IEEE Computer Society.

[36] Brinkschulte, Uwe und Theo Ungerer: *Mikrocontroller und Mikroprozessoren*. eXamen.press. Springer-Verlag, Berlin, 3. Auflage, 2010.

[37] Brown, Alan W., Jim Conallen und Dave Tropeano: *Introduction: Models, Modeling and Model-Driven Architecture (MDA)*. In: Beydeda, Sami, Matthias Book und Volker Gruhn (Herausgeber): *Model-Driven Software Development*, Seiten 1–16. Springer, Berlin, Heidelberg, 2005.

[38] Canedo, Arquimedes und Mohammad Abdulah Al-Faruque: *Towards Parallel Execution of IEC 61131 Industrial Cyber-Physical Systems Applications*. In: Rosenstiel, Wolfgang und Lothar Thiele (Herausgeber): *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, Seiten 554–557. IEEE Computer Society, März 2012.

[39] Carpenter, John, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson und Sanjoy Baruah: *A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms*. In: Leung, Joseph Y-T. (Herausgeber): *Handbook of Scheduling: Algorithms, Models and Performance Analysis*, Boca Raton, FL, USA, 2004. CRC Press.

[40] Ceng, Jianjiang, Jerónimo Castrillón, Weihua Sheng, Hanno Scharwächter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Tsuyoshi Isshiki und Hiroaki Kunieda: *MAPS: An Integrated Framework for MPSoC Application Parallelization*. In: *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, Seiten 754–759, New York, NY, USA, 2008. ACM.

[41] Choi, Youngsoo, Allan Knies, Geetha Vedaraman und Jeremiah Williamson: *Design and Experience: Using the Intel Itanium 2 Processor Performance Monitoring Unit to Implement Feedback Optimizations*. EPIC2 Workshop, November 2002.

[42] Cmelik, Bob und David Keppel: *Shade: A Fast Instruction-Set Simulator for Execution Profiling*. In: *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '94, Seiten 128–137, New York, NY, USA, 1994. ACM.

[43] Cockx, Johan, Kristof Denolf, Bart Vanhoof und Richard Stahl: *SPRINT: A Tool to Generate Concurrent Transaction-Level Models from Sequential Code*. EURASIP Journal on Advances in Signal Processing, 2007.

[44] Cordes, Daniel, Andreas Heinig, Peter Marwedel und Arindam Mallik: *Automatic Extraction of Pipeline Parallelism for Embedded Software Using Linear Programming*. In: *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11, Seiten 699–706, Washington, DC, USA, 2011. IEEE Computer Society.

[45] Cordes, Daniel und Peter Marwedel: *Multi-objective Aware Extraction of Task-Level Parallelism Using Genetic Algorithms*. In: Rosenstiel, Wolfgang und Lothar Thiele (Herausgeber): *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, Seiten 394–399. IEEE Computer Society, März 2012.

[46] Cordes, Daniel, Peter Marwedel und Arindam Mallik: *Automatic Parallelization of Embedded Software Using Hierarchical Task Graphs and Integer Linear Programming*. In: *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, Seiten 267–276, New York, NY, USA, 2010. ACM.

[47] Craenen, B. G. W., A. E. Eiben und E. Marchiori: *How to Handle Constraints with Evolutionary Algorithms*. In: Chambers, Lance D. (Herausgeber): *Practical Handbook of Genetic Algorithms: Applications*, Seiten 341–361. Chapman & Hall/CRC, 2. Auflage, 2001.

[48] Dagum, Leonardo und Ramesh Menon: *OpenMP: An Industry-Standard API for Shared-Memory Programming*. IEEE Computational Science & Engineering, 5(1):46–55, Januar 1998.

[49] Deb, Kalyanmoy: *Multi-objective Evolutionary Algorithms: Introducing Bias Among Pareto-optimal Solutions*. In: Ghosh, Ashish und Shigeyoshi Tsutsui (Herausgeber): *Advances in Evolutionary Computing: Theory and Applications*, Seiten 263–292. Springer Verlag, Berlin, Heidelberg, 2003.

[50] Domeika, Max: *Software Development for Embedded Multi-Core Systems: A Practical Guide Using Embedded Intel Architecture*. Newnes, Burlington, MA, USA, 2008.

[51] Esmaeilzadeh, Hadi, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam und Doug Burger: *Dark Silicon and the End of Multicore Scaling*. IEEE Micro, 32(3):122–134, Mai/Juni 2012.

[52] ETAS Group: *ASCET Software Produkte*. URL: `http://www.etas.com/de/products/ascet_software_products.php` (abgerufen am 30.12.2013).

[53] Favre-Bulle, Bernard: *Automatisierung komplexer Industrieprozesse: Systeme, Verfahren und Informationsmanagement*. Springer-Verlag, Wien, 2004.

[54] Feiner, Peter, Angela Demke Brown und Ashvin Goel: *Comprehensive Kernel Instrumentation via Dynamic Binary Translation*. In: *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, Seiten 135–146, New York, NY, USA, 2012. ACM.

[55] Fowler, Martin: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman, Inc., Boston, MA, USA, 1999.

[56] Freund, Roland W. und Ronald W. Hoppe: *Stoer/Bulirsch: Numerische Mathematik 1*. Springer-Lehrbuch, Berlin, Heidelberg, 10. Auflage, 2007.

[57] Friedl, Jeffrey E. F.: *Mastering Regular Expressions*. O'Reilly Media, Inc., Sebastopol, CA, USA, 3. Auflage, August 2006.

[58] Ganssle, Jack G.: *The Art of Designing Embedded Systems*. EDN Series for Design Engineers. Newnes, Burlington, MA, USA, 2. Auflage, 2008.

[59] Garcia, Saturnino, Donghwan Jeon, Christopher Louie und Michael Bedford Taylor: *Kremlin: Rethinking and Rebooting gprof for the Multicore Age*. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, Seiten 458–469, New York, NY, USA, 2011. ACM.

[60] Garcia, Saturnino, Donghwan Jeon, Christopher Louie und Michael Bedford Taylor: *The Kremlin Oracle for Sequential Code Parallelization*. IEEE Micro, 32(4):42–53, Juli 2012.

[61] Gebali, Fayez: *Algorithms and Parallel Computing*. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2011.

[62] Gen, Mitsuo und Runwei Cheng: *Genetic Algorithms and Engineering Optimization*. Wiley Series in Engineering Design and Automation. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2000.

[63] Giladi, Ran: *Network Processors: Architecture, Programming, and Implementation*. The Morgan Kaufmann Series in Systems on Silicon. Morgan Kaufmann Publishers, Inc., Burlington, MA, USA, 2008.

[64] Gove, Darryl: *Multicore Application Programming: For Windows, Linux, and Oracle Solaris*. Addison-Wesley Professional, Boston, MA, USA, 2010.

[65] Grama, Ananth, George Karypis, Vipin Kumar und Anshul Gupta: *Introduction to Parallel Computing*. Addison Wesley, 2. Auflage, 2003.

[66] GraphML Project Group: *GraphML Specification*. URL: `http://graphml.graphdrawing.org` (abgerufen am 30.12.2013).

[67] Green Hills Software: *Integrity Multivisor*. URL: `http://www.ghs.com/products/rtos/integrity_virtualization.html` (abgerufen am 30.12.2013).

[68] GRÖTKER, THORSTEN, STAN LIAO, GRANT MARTIN und STUART SWAN: *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[69] GRÖTSCH, EBERHARD E.: *SPS - speicherprogrammierbare Steuerungen als Bausteine verteilter Automatisierung*. Oldenbourg Industrieverlag, München, 5. Auflage, 2004.

[70] GÜTING, RALF H. und MARTIN ERWIG: *Übersetzerbau: Techniken, Werkzeuge, Anwendungen*. Springer-Lehrbuch, Berlin, Heidelberg, 1999.

[71] HARDY, DAMIEN. und ISABELLE PUAUT: *Estimation of Cache Related Migration Delays for Multi-Core Processors with Shared Instruction Caches*. In: *Proceedings of the 17th Real-Time and Network Systems (RTNS 2009)*, Seiten 45–54, Paris, Frankreich, Oktober 2009.

[72] HARTMANN, JULIAN: *Interaktionsorientierte Modellextraktion aus steuerungstechnischer Echtzeit-Firmware am Beispiel einer Mehrachsen-NC-Steuerung mit Mehrkern-System*. Bachelorarbeit, Hochschule Fulda, November 2010.

[73] HELD, JIM, JERRY BAUTISTA und SEAN KOEHL: *From a Few Cores to Many: A Tera-Scale Computing Research Overview*. Intel White Paper, Intel Corporation, 2006.

[74] HENNESSY, JOHN L. und DAVID A. PATTERSON: *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, Inc., Waltham, MA, USA, 5. Auflage, 2011.

[75] HERBER, CHRISTIAN, ANDRE RICHTER, HOLM RAUCHFUSS und ANDREAS HERKERSDORF: *Self-virtualized CAN Controller for Multi-Core Processors in Real-Time Applications*. In: *Proceedings of the 26th International Conference on Architecture of Computing Systems*, ARCS'13, Seiten 244–255, Berlin, Heidelberg, 2013. Springer-Verlag.

[76] HERMAN, IVAN, GUY MELANÇON und M. SCOTT MARSHALL: *Graph Visualization and Navigation in Information Visualization: A Survey*. IEEE Transactions on Visualization and Computer Graphics, 6(1):24–43, Januar 2000.

[77] HERRTWICH, RALF und GÜNTER HOMMEL: *Nebenläufige Programme*. Springer-Lehrbuch, Berlin, Heidelberg, 2. Auflage, 1994.

[78] HIND, MICHAEL und ANTHONY PIOLI: *Which Pointer Analysis Should I Use?* In: *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, Seiten 113–123, New York, NY, USA, 2000. ACM.

[79] HOLLAND, JOHN H.: *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, Cambridge, MA, USA, 1. MIT Press Auflage, 1992.

[80] HYDE, RANDALL: *Write Great Code: Understanding the Machine*. No Starch Press, Inc., San Francisco, CA, USA, 2004.

[81] IBM CORPORATION: *Rational Rhapsody family*. URL: `http://www.ibm.com/software/awdtools/rhapsody/` (abgerufen am 30.12.2013).

[82] INFINEON TECHNOLOGIES AG, München: *TC1797 32-Bit Single-Chip Microcontroller Data Sheet V1.2 2009-09*, September 2009.

[83] INSTITUT TIK, ETH ZÜRICH: *PISA - A Platform and Programming Language Independent Interface for Search Algorithms*. URL: `http://www.tik.ee.ethz.ch/pisa/` (abgerufen am 30.12.2013).

[84] INTEL CORPORATION: *Intel® Advisor XE 2013*. URL: `http://software.intel.com/en-us/intel-advisor-xe/` (abgerufen am 30.12.2013).

[85] INTEL CORPORATION: *Intel Core 2 Duo Processor and Intel Core 2 Extreme Processor on 45-nm Process for Platforms Based on Mobile Intel 965 Express Chipset Family Datasheet*, Januar 2008.

[86] INTEL CORPORATION: *An Introduction to the Intel QuickPath Interconnect*, 1. Auflage, Januar 2009.

[87] INTEL CORPORATION: *Desktop 3rd Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family Datasheet, Volume 1*, Januar 2013.

[88] INTEL CORPORATION: *Intel Virtualization Technology for Directed I/O, Architecture Specification, Revision 2.2*, September 2013.

[89] JEON, DONGHWAN, SATURNINO GARCIA, CHRISTOPHER LOUIE und MICHAEL BEDFORD TAYLOR: *Kismet: Parallel Speedup Estimates for Serial Programs*. In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '11, Seiten 519–536, New York, NY, USA, 2011. ACM.

[90] JOHN, KARL HEINZ und MICHAEL TIEGELKAMP: *SPS-Programmierung mit IEC 61131-3: Konzepte und Programmiersprachen, Anforderungen an Programmiersysteme, Entscheidungshilfen*. Springer-Verlag, Berlin, Heidelberg, 4. Auflage, 2009.

[91] KAISER, ROBERT: *Alternatives for Scheduling Virtual Machines in Real-Time Embedded Systems*. In: *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIES '08, Seiten 5–10, New York, NY, USA, 2008. ACM.

[92] KASTENS, UWE und HANS KLEINE BÜNING: *Modellierung: Grundlagen und formale Methoden*. Carl Hanser Verlag, München, 2. Auflage, 2005.

[93] KAUFFMAN, STUART A.: *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, Inc., New York, NY, USA, 1. Auflage, 1993.

[94] KIENZLE, EBERHARD und JÖRG FRIEDRICH: *Programmierung von Echtzeitsystemen*. Carl Hanser Verlag, München, 2008.

[95] KIM, MINJANG: *Dynamic Program Analysis Algorithms to Assist Parallelization*. Dissertation, School of Computer Science, Georgia Institute of Technology, Dezember 2012.

[96] Kim, Minjang, Hyesoon Kim und Chi-Keung Luk: *SD3: A Scalable Approach to Dynamic Data-Dependence Profiling.* In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, Seiten 535–546, Washington, DC, USA, 2010. IEEE Computer Society.

[97] Kim, Minjang, Pranith Kumar, Hyesoon Kim und Bevin Brett: *Predicting Potential Speedup of Serial Code via Lightweight Profiling and Emulations with Memory Performance Model.* In: *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, Seiten 1318–1329, Washington, DC, USA, 2012. IEEE Computer Society.

[98] Kleidermacher, David und Mike Kleidermacher: *Embedded Systems Security: Practical Methods for Safe and Secure Software and Systems Development.* Newnes, Burlington, MA, USA, 2012.

[99] Klein, Gerwin, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch und Simon Winwood: *seL4: Formal Verification of an OS Kernel.* In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, Seiten 207–220, New York, NY, USA, 2009. ACM.

[100] Krar, Steve und Arthur Gill: *Exploring Advanced Manufacturing Technologies.* Industrial Press, Inc., New York, NY, USA, 2003.

[101] Krumke, Sven Oliver und Hartmut Noltemeier: *Graphentheoretische Konzepte und Algorithmen.* Vieweg+Teubner Verlag, Springer Fachmedien, Wiesbaden, 3. Auflage, 2012.

[102] Landi, William: *Undecidability of Static Analysis.* ACM Letters on Programming Languages and Systems, 1(4):323–337, Dezember 1992.

[103] Larsen, Per, Sven Karlsson und Jan Madsen: *Identifying Inter-task Communication in Shared Memory Programming Models.* In: *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, Seiten 168–182, Berlin, Heidelberg, 2009. Springer-Verlag.

[104] Larsen, Per, Sven Karlsson und Jan Madsen: *Expressing Coarse-Grain Dependencies Among Tasks in Shared Memory Programs.* IEEE Transactions on Industrial Informatics, 7(4):652–660, 2011.

[105] Larus, James und Dennis Gannon: *Multicore Computing and Scientific Discovery.* In: Hey, Tony, Stewart Tansley und Kristin Tolle (Herausgeber): *The Fourth Paradigm - Data-Intensive Scientific Discovery*, Seiten 125–129. Microsoft Research, Redmond, WA, USA, 2009.

[106] Law, Averill M. und David M. Kelton: *Simulation Modeling and Analysis.* McGraw-Hill Higher Education, New York, NY, USA, 3. Auflage, 1999.

[107] Levi, Paul und Ulrich Rembold: *Einführung in die Informatik für Naturwissenschaftler und Ingenieure*. Carl Hanser Verlag, München, Wien, 4. Auflage, 2003.

[108] Li, Peng, Binoy Ravindran und E. Douglas Jensen: *Adaptive Time-Critical Resource Management Using Time/Utility Functions: Past, Present, and Future*. In: *Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - Volume 02*, COMPSAC '04, Seiten 12–13, Washington, DC, USA, 2004. IEEE Computer Society.

[109] Liu, Ai-Hsin und Robert P. Dick: *Automatic Run-Time Extraction of Communication Graphs from Multithreaded Applications*. In: *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '06, Seiten 46–51, New York, NY, USA, Oktober 2006. ACM.

[110] Luk, Chi-Keung, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi und Kim Hazelwood: *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation*. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, Seiten 190–200, New York, NY, USA, 2005. ACM.

[111] LynuxWorks, Inc.: *Virtualization Security Technology: LynxSecure Embedded Hypervisor and Separation Kernel*. URL: `http://www.lynuxworks.com/virtualization/hypervisor.php` (abgerufen am 30.12.2013).

[112] Magnusson, Peter S., Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt und Bengt Werner: *Simics: A Full System Simulation Platform*. Computer, 35(2):50–58, Februar 2002.

[113] Marr, Deborah T., Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, Alan J. Miller und Michael Upton: *Hyper-Threading Technology Architecture and Microarchitecture*. Intel Technology Journal, 6(1):4–15, Februar 2002.

[114] Matassa, Lori M. und Yatin Patil: *Best Practices: Adoption of Symmetric Multiprocessing Using VxWorks and Intel Multi-Core Processors*. Intel Corporation, Februar 2009.

[115] Mathew, Binu: *Very Large Instruction Word Architectures*. In: Oklobdzija, Vojin G. (Herausgeber): *Digital Systems and Publications*, The Computer Engineering Handbook. CRC Press, Boca Raton, FL, USA, 2. Auflage, 2008.

[116] MathWorks Inc.: *Simulink - Simulation and Model-Based Design*. URL: `http://www.mathworks.com/products/simulink/` (abgerufen am 30.12.2013).

[117] Mendelson, Avi, Julius Mandelblat, Simcha Gochman, Anat Shemer, Rajshree Chabukswar, Erik Niemeyer und Arun Kumar: *CMP Implementation in Systems Based on the Core Duo*. Intel Technology Journal, 10(2):99–108, Mai 2006.

[118] Mignolet, Jean-Yves, Rogier Baert, Thomas J. Ashby, Prabhat Avasare, Hye-On Jang und Jae Cheol Son: *MPA: Parallelizing an Application onto a Multicore Platform Made Easy*. IEEE Micro, 29(3):31–39, Mai 2009.

[119] Mignolet, Jean-Yves und Roel Wuyts: *Embedded Multiprocessor Systems-on-Chip Programming*. IEEE Software, 26(3):34–41, Mai 2009.

[120] Monot, Aurélien, Nicolas Navet, Bernard Bavoux und Françoise Simonot-Lion: *Multisource Software on Multicore Automotive ECUs – Combining Runnable Sequencing With Task Scheduling*. Industrial Electronics, IEEE Transactions on, 59(10):3934–3942, 2012.

[121] Moore, Gordon E.: *Cramming More Components onto Integrated Circuits*. Electronics, 38(8):114–117, April 1965.

[122] Moyer, Bryon und Paul Stravers: *Partitioning Programs for Multicore Systems*. In: Moyer, Bryon (Herausgeber): *Real World Multicore Embedded Systems*, Seiten 331 – 384. Newnes, Waltham, MA, USA, 2013.

[123] Muchnick, Steven S.: *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 5. Auflage, 1997.

[124] Müller, Thomas: *Trusted Computing Systeme: Konzepte und Anforderungen*. Xpert.press. Springer-Verlag, Berlin, Heidelberg, 2008.

[125] Neiger, Gil, Amy Santoni, Felix Leung, Dion Rodgers und Rich Uhlig: *Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization*. Intel Technology Journal, 10(3):167–178, August 2006.

[126] Nemati, Farhang und Thomas Nolte: *A Flexible Tool for Evaluating Scheduling, Synchronization and Partitioning Algorithms on Multiprocessors*. In: *Proceedings of 15th IEEE International Conference on Emerging Technologies and Factory Automation*, ETFA 2010, Seiten 1–4. IEEE Computer Society, 2010.

[127] Nethercote, Nicholas und Julian Seward: *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation*. In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, Seiten 89–100, New York, NY, USA, 2007. ACM.

[128] Neto, Pedro, J. Norberto Pires und A. Paulo Moreira: *High-level Programming and Control for Industrial Robotics: Using a Hand-Held Accelerometer-Based Input Device for Gesture and Posture Recognition*. Industrial Robot: An International Journal, 37(2):137–147, 2010.

[129] N.N.: *OSACA - Open System Architecture for Controls within Automation Systems, Final Report OSACA I & II*, 1996.

[130] Norm DIN 66025-1:1983-01: *Programmaufbau für numerisch gesteuerte Arbeitsmaschinen; Allgemeines*, Januar 1983.

[131] Norm DIN 66025-2:1988-09: *Industrielle Automation; Programmaufbau für numerisch gesteuerte Arbeitsmaschinen; Wegbedingungen und Zusatzfunktionen*, September 1988.

[132] Norm DIN EN 61131-3:2003-12 : *Speicherprogrammierbare Steuerungen - Teil 3: Programmiersprachen (IEC 61131-3:2003)*, Dezember 2003.

[133] Norm DIN EN 61508-3 (VDE 0803-3):2011-02: *Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme - Teil 3: Anforderungen an Software*, Februar 2011.

[134] Norm ISO/IEC 7498-1:1994: *Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*, 1994.

[135] Nowotsch, Jan und Michael Paulitsch: *Leveraging Multi-core Computing Architectures in Avionics*. In: *2012 Ninth European Dependable Computing Conference*, Seiten 132–143, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[136] OSADL eG: *OSADL - Open Source Automation Development Lab eG*. URL: `http://www.osadl.org` (abgerufen am 30.12.2013).

[137] Ottoni, Guilherme, Ram Rangan, Adam Stoler und David I. August: *Automatic Thread Extraction with Decoupled Software Pipelining*. In: *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-38, Seiten 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[138] Patterson, David A. und John L. Hennessy: *Computer Organization and Design, Revised Fourth Edition: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 4. Auflage, 2012.

[139] Peeters, Werner: *An Overview of Fuzzy Control Theory*. In: Lowen, Robert und Alain Verschoren (Herausgeber): *Foundations of Generic Optimization. Volume 2: Applications of Fuzzy Control, Genetic Algorithms and Neural Networks*, Band 24 der Reihe *Mathematical Modelling: Theory and Applications*, Seiten 1–138. Springer Publishing Company, Inc., 1. Auflage, 2008.

[140] Popek, Gerald und Robert Goldberg: *Formal Requirements for Virtualizable Third Generation Architectures*. Communications of the ACM, 17(7):412–421, Juli 1974.

[141] Pritschow, Günter: *Einführung in die Steuerungstechnik*, Band 1 der Reihe *Automatisierung in der Produktion*. Carl Hanser Verlag, München, 2006.

[142] Pritschow, Günter, Yusuf Altintas, Francesco Jovane, Yoram Koren, Mamoru Mitsuishi, Shozo Takata, Hendrik van Brussel, Manfred Weck und Kazuo Yamazaki: *Open Controller Architecture – Past, Present and Future*. CIRP Annals - Manufacturing Technology, 50(2):463 – 470, 2001.

[143] Ramalingam, Ganesan: *The Undecidability of Aliasing*. ACM Transactions on Programming Languages and Systems (TOPLAS), 16(5):1467–1471, September 1994.

[144] Randall, Robert Bond: *Vibration-based Condition Monitoring: Industrial, Aerospace and Automotive Applications*. John Wiley & Sons, Inc., West Sussex, UK, 2011.

[145] Rangan, Ram, Neil Vachharajani, Manish Vachharajani und David I. August: *Decoupled Software Pipelining with the Synchronization Array*. In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, Seiten 177–188, Washington, DC, USA, 2004. IEEE Computer Society.

[146] Rauber, Thomas und Gudula Rünger: *Parallele Programmierung*. eXamen.press. Springer-Verlag, Berlin, Heidelberg, 2. Auflage, 2007.

[147] Real-Time Systems GmbH: *Real-Time Systems - Embedded Hypervisor für Multicore Architektur*. URL: `http://www.real-time-systems.com/de/real-time_hypervisor/index.php` (abgerufen am 30.12.2013).

[148] Real-Time Systems GmbH, Ravensburg: *RTS Hypervisor OS Porting Guide*, September 2008.

[149] Real-Time Systems GmbH, Ravensburg: *Real-Time Hypervisor Version 2.2 Product Description & User's Guide*, 2010.

[150] Reichenbach, Frank und Alexander Wold: *Multi-core Technology – Next Evolution Step in Safety Critical Systems for Industrial Applications?* In: *Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, DSD '10, Seiten 339–346, Washington, DC, USA, 2010. IEEE Computer Society.

[151] Robin, John Scott und Cynthia E. Irvine: *Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor*. In: *Proceedings of the 9th Conference on USENIX Security Symposium*, SSYM' 00, Berkeley, CA, USA, 2000. USENIX Association.

[152] Rul, Sean, Hans Vandierendonck und Koen De Bosschere: *A Profile-Based Tool for Finding Pipeline Parallelism in Sequential Programs*. Parallel Computing, 36(9):531–551, September 2010.

[153] Russell, Stuart J. und Peter Norvig: *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Pearson Education, Inc., Upper Saddle River, NJ, USA, 3. Auflage, 2010.

[154] Sargent, Robert. G.: *Verification and Validation of Simulation Models*. In: Mason, Scott J., Raymond R. Hill, Lars Mönch, Oliver Rose, Thomas Jefferson und John W. Fowler (Herausgeber): *Proceedings of the 2008 Winter Simulation Conference*, Seiten 157–169, 2008.

[155] Schaffer, J. David, Richard A. Caruana, Larry J. Eshelman und Rajarshi Das: *A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization*. In: *Proceedings of the 3rd International Conference on Genetic Algorithms*, Seiten 51–60, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers, Inc.

[156] Schutzrecht EP 2341405 A2 (06.07.2011), Robert Bosch GmbH, Pr.: 30.12.2009 DE 102009060891: *Verfahren zum Betrieb einer Maschine*.

[157] Seitz, Matthias: *Speicherprogrammierbare Steuerungen*. Carl Hanser Verlag, München, 2. Auflage, 2008.

[158] Silberschatz, Abraham, Peter Baer Galvin und Greg Gagne: *Operating System Concepts*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 8. Auflage, 2009.

[159] Sinnen, Oliver: *Task Scheduling for Parallel Systems*. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2007.

[160] Sivanandam, S. N. und S. N. Deepa: *Introduction to Genetic Algorithms*. Springer-Verlag, Berlin, Heidelberg, 2008.

[161] Smith, James E. und Ravi Nair: *Virtual Machines: Versatile Platforms for Systems and Processes*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 2005.

[162] Stallings, William: *Operating Systems: Internals and Design Principles*. Pearson Education, Inc., Upper Saddle River, NJ, USA, 7. Auflage, 2012.

[163] Sure, Matthias: *Moderne Controlling-Instrumente: Bewährte Konzepte für das operative und strategische Controlling*. Controlling Competence. Franz Wahlen GmbH, München, 2011.

[164] Sutter, Herb: *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. Dr. Dobb's Journal, 30(3):202–210, 2005.

[165] Sutter, Herb: *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. URL: `http://www.gotw.ca/publications/concurrency-ddj.htm` (abgerufen am 30.12.2013), August 2009.

[166] Sysgo AG: *PikeOS RTOS and Virtualization Concept*. URL: `http://www.sysgo.com/en/products/pikeos-rtos-and-virtualization-concept/` (abgerufen am 30.12.2013).

[167] Szabo, Anita und Norbert Sram: *Functional Programming in Embedded Systems and Soft Computing*. In: *5th International Symposium on Intelligent Systems and Informatics 2007*, SISY 2007, Seiten 61–65, August 2007.

[168] Tanenbaum, Andrew S.: *Modern Operating Systems*. Pearson Education, Inc., Upper Saddle River, NJ, USA, 3. Auflage, 2008.

[169] Thies, William, Vikram Chandrasekhar und Saman Amarasinghe: *A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs*. In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-40, Seiten 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[170] Torchiano, Marco und Maurizio Morisio: *Overlooked Aspects of COTS-Based Development*. IEEE Software, 21(2):88–93, März 2004.

[171] TOURNAVITIS, GEORGIOS und BJÖRN FRANKE: *Semi-Automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism Using Profiling Information*. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, Seiten 377–388, New York, NY, USA, 2010. ACM.

[172] UNGERER, THEO: *Parallelrechner und parallele Programmierung*. Spektrum Hochschultaschenbuch. Spektrum Akademischer Verlag GmbH, Heidelberg, Berlin, 1997.

[173] UNITED STATES DEPARTMENT OF DEFENSE: *Trusted Computer System Evaluation Criteria, DoD 5200.28-STD*, Dezember 1985.

[174] VAJDA, ANDRÁS: *Programming Many-Core Chips*. Springer-Verlag, New York, Dordrecht, Heidelberg, London, 2011.

[175] VAN VELDHUIZEN, DAVID ALLEN: *Multiobjective Evolutionary Algorithms: Classifications, Analyses, and New Innovations*. Dissertation, Air Force Institute of Technology, Wright Patterson AFB, OH, USA, 1999.

[176] VANDIERENDONCK, HANS, SEAN RUL und KOEN DE BOSSCHERE: *The Paralax Infrastructure: Automatic Parallelization With a Helping Hand*. In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, Seiten 389–400, New York, NY, USA, 2010. ACM.

[177] VECTOR FABRICS: *Vector Fabrics Pareon Multicore Optimization*. URL: `http://www.vectorfabrics.com/products/` (abgerufen am 30.12.2013).

[178] WAGNER, TOBIAS, HEIKE TRAUTMANN und LUIS MARTÍ: *A Taxonomy of Online Stopping Criteria for Multi-objective Evolutionary Algorithms*. In: *Proceedings of the 6th International Conference on Evolutionary Multi-Criterion Optimization*, EMO'11, Seiten 16–30, Berlin, Heidelberg, 2011. Springer-Verlag.

[179] WALKINSHAW, NEIL, KIRILL BOGDANOV, MIKE HOLCOMBE und SARAH SALAHUDDIN: *Improving Dynamic Software Analysis by Applying Grammar Inference Principles*. Journal of Software Maintenance and Evolution: Research and Practice - Special Issue on Program Comprehension through Dynamic Analysis (PCODA), 20(4):269–290, Juli 2008.

[180] WECK, MANFRED und CHRISTIAN BRECHER: *Werkzeugmaschinen 3 - Mechatronische Systeme: Vorschubantriebe, Prozessdiagnose*. VDI-Buch. Springer-Verlag, Berlin, Heidelberg, 6. Auflage, 2006.

[181] WECK, MANFRED und CHRISTIAN BRECHER: *Werkzeugmaschinen 4 - Automatisierung von Maschinen und Anlagen*. VDI-Buch. Springer-Verlag, Berlin, Heidelberg, 6. Auflage, 2006.

[182] WEICKER, KARSTEN: *Evolutionäre Algorithmen*. Leitfäden der Informatik. Vieweg+Teubner Verlag, GWV Fachverlage GmbH, Stuttgart, 2. Auflage, 2007.

[183] WEISER, MARK: *Program Slicing*. In: *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, Seiten 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[184] Wills, Craig E.: *Process Synchronization and Interprocess Communication*. In: Tucker, Allen B. (Herausgeber): *Computer Science Handbook*. Chapman & Hall/CRC, Boca Raton, FL, USA, 2. Auflage, 2004.

[185] Wind River Systems, Inc.: *Wind River Hypervisor*. URL: `http://www.windriver.com/products/hypervisor/` (abgerufen am 30.12.2013).

[186] Wind River Systems, Inc., Alameda, CA, USA: *VxWorks Application Programmer's Guide 6.7*, November 2008.

[187] Wind River Systems, Inc., Alameda, CA, USA: *VxWorks Kernel Programmer's Guide 6.7*, November 2008.

[188] Wind River Systems, Inc., Alameda, CA, USA: *Wind River System Viewer User's Guide 3.2*, November 2009.

[189] Wind River Systems, Inc., Alameda, CA, USA: *Wind River Workbench 3.2*, November 2010.

[190] Wörn, Heinz und Uwe Brinkschulte: *Echtzeitsysteme: Grundlagen, Funktionsweisen, Anwendungen*. eXamen.press. Springer-Verlag, Berlin, Heidelberg, 2005.

[191] yWorks GmbH: *yEd Graph Editor*. URL: `http://www.yworks.com/yed/` (abgerufen am 30.12.2013).

[192] Zalfany Urfianto, Mohammad, Tsuyoshi Isshiki, Arif Ullah Khan, Dongju Li und Hiroaki Kunieda: *A Multiprocessor SoC Architecture with Efficient Communication Infrastructure and Advanced Compiler Support for Easy Application Development*. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, E91-A(4):1185–1196, April 2008.

[193] Zitzler, Eckart, Kalyanmoy Deb und Lothar Thiele: *Comparison of Multiobjective Evolutionary Algorithms: Empirical Results*. Evolutionary Computing, 8(2):173–195, Juni 2000.

[194] Zitzler, Eckart, Marco Laumanns und Stefan Bleuler: *A Tutorial on Evolutionary Multiobjective Optimization*. In: Gandibleux, Xavier, Marc Sevaux, Kenneth Sörensen und Vincent T'kindt (Herausgeber): *Metaheuristics for Multiobjective Optimisation*, Band 535 der Reihe *Lecture Notes in Economics and Mathematical Systems*, Seiten 3–38, Berlin, 2004. Springer-Verlag.

[195] Zitzler, Eckart, Marco Laumanns und Lothar Thiele: *SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization*. In: Giannakoglou, K. C. et al. (Herausgeber): *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems*, EUROGEN 2001, Seiten 95–100. International Center for Numerical Methods in Engineering (CIMNE), 2002.

[196] Zitzler, Eckart und Lothar Thiele: *Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach*. IEEE Transactions on Evolutionary Computation, 3(4):257–271, 1999.

Stichwortverzeichnis

Ablaufsteuerung, 11
Allel, 25
Anpasssteuerung, 15
Antriebsregler, 11, 13
Aperiodische Task, 15, 16, 35
ASIC, 1
Aufwandskante, 84, 106
Ausgeprägtes Modell, 63
Automatisierung, 1, 11, 13

Basisblock, 29, 102
Basismodell, 63
Befehlssatzarchitektur, 8, 27
Betriebssystem, 15, 16, 30
    Echtzeit-, 15, 16
    SMP-, 17, 45
    Standard-, 16
Bewegungssteuerung, 13, 14
Binärübersetzung, 29

Cache, 9
    -Kohärenz, 10
    -Konsistenz, 9
Chromosom, 132, 133
CNC, *siehe* Werkzeugmaschinensteuerung
Codeanalyse, *siehe* Software-Analyse
Codeblock, 101
    -Agglomeration, 107
    -Ausführungsdauer, 104, 113, 122
    -Granularität, 104
    -Graph, 60, 64, 104
    -Knoten, 104
    -Partitionierung, 102
    -Schedule, 106, 125
Core, *siehe* CPU-Kern
Core-Affinität, 84
CPU-Kern, 2, 8
Crossover, 26, 130, 133

Direkte Genotyp-Repräsentation, 132
Direkte Task-Interaktion, 86
Drehmaschine, 12, 38
Druckmaschine, 12, 13
Dynamische Software-Analyse, 23, 64, 66–73, 75, 116
Dynamisches Scheduling, 17, 122

Echtzeit, 15, 17, 33
    harte, 15, 16
    weiche, 15
Echtzeitbetriebssystem, 15, 16
EEEPA, 65, 73, 77
Empfangsoperation, 87
Epistase, 129, 132

Feldbus, 13, 15
Fitness, 25
    -bewertung, 25, 78, 123
    -landschaft, 129
FPGA, 1
Fräsmaschine, 38, 148

Genetischer Algorithmus, 24, 62, 71, 123, 146, 162
Genetischer Operator, 25, 123, 130, 133
Genotyp, 25, 123, 125, 128, 131
    Direkte Repräsentation, 132
    Indirekte Repräsentation, 131
Globales Scheduling, 17
GPOS, *siehe* Standardbetriebssystem
Graph, 21, 60, 63
GraphML, 77, 92, 162

Hardware-Virtualisierungsunterstützung, 30, 47
Harte Echtzeitbedingung, 15, 16
Heterogener Multicore-Prozessor, 8
HMI, 13, 43, 57

Homogener Multicore-Prozessor, 8, 60, 164
Hypercall, 30, 46
Hypervisor, 27, 44–46

Indirekte Genotyp-Repräsentation, 131
Indirekte Task-Interaktion, 86
Inter-Core-Interaktion, 82, 100
Inter-Prozessor-Interrupt, 18, 83
Interaktionskante, 84, 105
Interaktionsmedium, 86
Interpolation, 15, 38
Interpolationstakt, 15, 34, 36, 38, 148
IPI, *siehe* Inter-Prozessor-Interrupt

Kontrollflusskante, 105
Kritischer Abschnitt, 20
Kritischer Pfad, 71, 107, 109, 159

Lastverteilung, 81, 126, 150
Logiksteuerung, 14

Manycore-Prozessor, 11
Maschine, 1, 11
  Dreh-, 12, 38
  Druck-, 12, 13
  Fräs-, 38, 148
  Rundtakt-, 34, 38, 148
  Stanz-, 38, 149
  Verpackungs-, 12, 13
  Werkzeug-, 15, 148
Modellierung, 21, 60, 74
  Ausgeprägtes Modell, 63
  Basismodell, 63
  Modellausprägung, 63, 88, 113
  Modellierungsebene, 60
    Systemebene, 60, 80, 128, 140, 148
    Taskebene, 60, 99, 131, 140, 156
  Modellparametrierung, 63, 125
  Modellvalidierung, 21, 96, 120
  Parametriertes Modell, 63
Moore's Law, *siehe* Mooresches Gesetz
Mooresches Gesetz, 2
Multicore-Prozessor, 2, 8, 11, 145
  heterogener, 8
  homogener, 8, 60, 164
Multikriterielle Optimierung, 23, 129

Multilaterale Synchronisation, 20, 82, 93, 94, 98, 120, 140
Mutation, 26, 130, 133

Nebenläufigkeit, 3, 7, 9, 19, 20

Optimierung, 23, 79
  multikriterielle, 23, 129
  skalare, 23, 129

Parallelisierung, 19, 35, 59, 65
Parallelität, 7
  Datenebene, 7, 8, 20, 65
  Instruktionsebene, 2, 7, 8, 20
  Taskebene, 7, 8, 20
  Threadebene, 7
Parametriertes Modell, 63
Paravirtualisierung, 30, 46
Pareto-Dominanz, 24
Pareto-Front, 24, 127, 154
Pareto-optimale Menge, 24
Pareto-Optimierung, *siehe* Multikriterielle Optimierung
Partitioniertes Scheduling, 17, 36, 38, 60, 100, 165
Performanzsteigerung, 3, 33
Periodische Task, 15, 16
Permutationskodierung, 132
Phänotyp, 25, 123, 125, 130, 134
Pipelining, 7, 10
PISA, 75, 78, 80, 123
Privilegierungsstufe, 16, 28, 31
Prozess, 7

Quasiparallelität, 8, 18–20

Reaktivität, 42, 48, 49, 163
Reentrancy, *siehe* Wiedereintrittsfähigkeit
Refaktorierung, 72, 140
Rekombination, *siehe* Crossover
Response Time, 49
Robotersteuerung, 13, 14, 35
RTOS, *siehe* Echtzeitbetriebssystem
Rundtaktmaschine, 34, 38, 148

Safety, 39, 57

Scheduling, 17, 36, 43, 45
dynamisches, 17, 122
globales, 17
partitioniertes, 17, 36, 38, 60, 100, 165
statisches, 17
Schema-Theorem, 127, 128
Security, 35, 39, 57
Sendeoperation, 86
Skalare Optimierung, 23, 129
SMP-Betriebssystem, 17, 45
Software-Analyse, 22, 73
dynamische, 23, 64, 66–73, 75, 116
statische, 22, 66–72, 94
Software-Instrumentierung, 23, 64, 68, 75, 85, 110, 162
Software-Profiling, *siehe* Dynamische Software-Analyse
Speedup, 73, 100, 126, 158
Speicherprogrammierbare Steuerung, 13, 15, 40
Sporadische Task, 15, 16
SPS, *siehe* Speicherprogrammierbare Steuerung
SPS-Zykluszeit, 35, 36, 38, 148
Standardbetriebssystem, 16
Stanzmaschine, 38, 149
Statische Software-Analyse, 22, 66–72, 94
Statisches Scheduling, 17
Steuerung, 1, 3, 11, 14
Ablauf-, 11
Anpass-, 15
Bewegungs-, 13, 14
Roboter-, 13, 14, 35
Speicherprogrammierbare, 13, 15, 40
Verknüpfungs-, 11
Werkzeugmaschinen-, 13
Synchronisation, 7, 17, 18, 20, 100, 122, 143
multilaterale, 20, 82, 93, 94, 98, 120, 140
unilaterale, 20, 82, 98, 120, 141
Systemkonsolidierung, 2, 33, 38, 41, 44, 161
Systemvirtualisierung, 3, 27, 29, 30, 41, 45, 47

Task, 7, 16, 36
-Allokation, *siehe* Task-Mapping
-Dekomposition, 36, 60, 99, 101, 104, 144, 158
-Graph, 60, 64, 84
-Interaktion, 17, 82, 84, 86, 88, 89, 119
direkte, 86
indirekte, 86
-Knoten, 84
-Lastanteil, 81, 84, 89, 97, 99, 121, 143
-Mapping, 36, 38, 61, 81, 84, 125, 131
-Verteilung, *siehe* Task-Mapping
-Zustände, 16
aperiodische, 15, 16, 35
periodische, 15, 16
sporadische, 15, 16
Thread, 7, 20
Trusted Computing Base, 57, 58

Unilaterale Synchronisation, 20, 82, 98, 120, 141

Verknüpfungssteuerung, 11
Verpackungsmaschine, 12, 13
Virtual Machine Monitor, *siehe* Hypervisor
Virtualisierung, 27
Virtuelle Maschine, 27, 29, 45, 47

Weiche Echtzeitbedingung, 15
Werkzeugmaschine, 15, 148
Werkzeugmaschinensteuerung, 13
Wiedereintrittsfähigkeit, 20, 93, 120

Zeitdeterminismus, 42, 48, 53, 163

Der Einsatz von Multicore-Prozessoren in der industriellen Steuerungstechnik birgt sowohl Chancen als auch Risiken. So bietet diese Prozessorarchitektur die Chance zur Konsolidierung derzeit auf dedizierter Hardware ausgeführter heterogener Steuerungssubsysteme unter einer bisher nicht erreichbaren temporalen Isolation. Zugleich werden sich insbesondere in der Steuerungstechnik signifikante Performanzsteigerungen zukünftig nur durch den Einsatz von Multicore-Architekturen erzielen lassen, da die Taktraten von Prozessoren physikalische Grenzen erreicht haben.

Aus diesem Grund entwickelt und bewertet dieses Buch generische Strategien zur effizienten Nutzung von Multicore-Prozessoren in der Steuerungstechnik unter Berücksichtigung der spezifischen Rahmenbedingungen und Anforderungen dieser Domäne.

Würzburg University Press

ISBN 978-3-95826-010-8