Energie- und qualitätsbewußte Einplanung von periodischen Prozessen in eingebetteten Echtzeitsystemen

Dissertation

von

Markus Ramsauer

zur Erlangung des Doktorgrades der Naturwissenschaften
an der Fakultät für Mathematik und Informatik
der Universität Passau

Gutachter:

Prof. Dr.-Ing. Werner Grass
Prof. Dr. Peter Marwedel

23. Januar 2005

Danksagung

Besonderer Dank gilt meinem Doktorvater Prof. Dr.-Ing. Werner Grass, der es mir ermöglichte diese Dissertation zu erstellen und mir stets mit seinem Rat zur Seite stand. Prof Dr. Peter Marwedel danke ich für die Erstellung des Zweitgutachtens. Dr. Bernhard Sick, Thomas Schwarzfischer und alle anderen Kollegen am Lehrstuhl für Rechnerstrukturen haben durch wertvolle Diskussionen und Anregungen zur Abrundung dieser Arbeit beigetragen.

Meinen Eltern Rosa und Ludwig Ramsauer danke ich dafür, daß sie mir erlaubten meinen Weg zu gehen und mich dabei immer unterstützten. Die wichtigste Stütze bei der Anfertigung dieser Arbeit war Nadja, die immer an mich glaubte und mir beistand.

Markus Ramsauer

Zusammenfassung

Mobile Geräte dienen immer häufiger zur Ausführung von Echtzeitanwendungen, sie bieten immer mehr Rechenleistung und sie werden kleiner und leichter. Hohe Rechenleistung erfordert jedoch sehr viel Energie, was im Gegensatz zu den geringen Akkukapazitäten, die aus der Forderung nach kleinen und leichten Geräten resultieren, steht. Bei der Echtzeiteinplanung von Rechenprozessen gewinnt daher der Energieverbrauch der Geräte neben der rechtzeitigen Beendigung von Anwendungen zunehmend an Bedeutung, weil sie möglichst lange unabhängig vom Stromnetz betrieben werden sollen. Andererseits werden auf diesen Geräten rechenintensive Anwendungen ausgeführt, bei denen es wünschenswert ist, die maximale mit der verfügbaren Rechenleistung erzielbare Qualität zu erhalten.

In dieser Arbeit wird ein Systemmodell vorgestellt, das den Design-to-time-Ansatz mit den Möglichkeiten der dynamischen Leistungsanpassung (Rechenleistung und verbrauchte elektrische Leistung) moderner Prozessoren vereinigt. Der Design-to-time-Ansatz ermöglicht Energieeinsparungen oder Qualitätssteigerungen durch die dynamische Auswahl alternativer Implementierungen, welche dieselbe Aufgabe mit unterschiedlicher Ausführungsdauer und Qualität bzw. Energieverbrauch erfüllen. Das Systemmodell umfaßt unter anderem periodische Prozesse mit harten Echtzeitbedingungen, Datenabhängigkeiten und alternativen Implementierungen, sowie Prozessoren mit diskreten Leistungsstufen.

Die Einplanung der Prozesse erfolgt in zwei Phasen. In der Offline-Phase wird ein flexibler Schedule berechnet, der für die zur Laufzeit möglichen Kombinationen von verstrichener Zeit und noch einzuplanender Prozeßmenge den jeweils einzuplanenden Prozeß, sowie die zu verwendende Implementierung und gegebenenfalls die einzustellende Leistungsstufe beinhaltet. Dieser flexible Schedule wird während der Online-Phase mit vernachlässigbarem Zeit- und Energieaufwand von einem Scheduler interpretiert. Für die Berechnung der optimalen flexiblen Schedules wurde ein Optimierer entwickelt, der eine Folge von flexiblen Schedules mit monoton steigender Güte (niedriger Energieverbrauch bzw. hohe Qualität) generiert, und damit der Klasse der Anytime-Algorithmen zuzuordnen ist. Eine Variante der Dynamischen Programmierung dient zur Bestimmung global optimaler, flexibler Schedules, die beispielsweise als Basis für Benchmarks dienen. Eine auf Simulated Annealing basierende Variante des Optimierers ermöglicht ein schnelleres Auffinden guter, flexibler Schedules für umfangreichere Anwendungen.

Inhaltsverzeichnis

Symbolverzeichnis ix

1 Einführung **1**
1.1 Beispielanwendung 4
1.2 Zweiphasige Einplanung 8
1.2.1 Optimierung 11
1.2.2 Scheduling 12
1.3 Inhalt und Aufbau der Arbeit 13

2 Stand der Technik **15**
2.1 Scheduling 15
2.1.1 Qualitätsbewußtes Scheduling 16
2.1.2 Energiebewußtes Scheduling 19
2.2 Optimierung 20
2.2.1 Dynamische Programmierung 21
2.2.2 Simulated Annealing 22
2.3 Einordnung dieser Arbeit 22

3 Modellierung **25**
3.1 Anwendungs- und Hardwaremodell 26
3.1.1 Prozessoren 26
3.1.2 Methoden 27
3.1.3 Anwendung und Prozesse 28
3.1.4 Instanzierung 29
3.1.5 Berechnung der effektiven Bereitzeiten und Fristen 30
3.1.6 Bestimmung der Präzedenzrelation 30
3.2 Formales Modell der Beispielanwendung 32
3.3 Schedule 32
3.4 Zielfunktionen 33
3.4.1 Energiebewußtes Scheduling 34
3.4.2 Qualitätsbewußtes Scheduling 35
3.4.3 Mehrzieloptimierung 36

3.4.4 Beschränkung auf pareto-optimale Methoden 36
3.5 Bemerkungen zur Modellierung 39

4 Methodik und Durchführung 41
4.1 Modellierung für Dynamische Programmierung 41
4.1.1 Rekursive Problembeschreibung 42
4.1.2 Komplexitätsanalyse bei Dynamischer Programmierung . 42
4.2 Anforderungsgetriebene Dynam. Programmierung 44
4.2.1 Optimierungseinsparungen 44
4.2.2 Umsetzung der anforderungsgetriebenen Dynamischen Programmierung 46
4.3 Optimierungsalgorithmen 56
4.3.1 Anforderungsgetriebene Dynamische Programmierung . . 56
4.3.2 Simuliertes Ausglühen 57
4.3.3 Abschneiden von Teil-Bedingungsgraphen 58
4.3.4 Vergröberung des Zeitrasters 59
4.4 Nachbearbeitung 59
4.4.1 Bedingungsgraph Kompaktifizierung 59
4.5 Dynamisches Scheduling 65
4.6 Bemerkungen zur Methodik und Durchführung 67

5 Strukturelle Umsetzung der Optimierer 69
5.1 Lotsen 71
5.1.1 Sortierte Lotsen (EDF, ERF, RM) 72
5.1.2 Zufallslotse (MC) 75
5.1.3 Dynamische Scheduling-Algorithmen als Lotsen 75
5.2 Knotenspeicher 77
5.2.1 Aufbau des Knotenspeichers 77
5.2.2 Konsistenzsicherung im Knotenspeicher 78
5.2.3 Monitoring 84
5.3 Optimierer 85
5.3.1 Anforderungsgetriebene Dynamische Programmierung . . 85
5.3.2 Simulated Annealing 86
5.4 Bemerkungen zur strukturellen Umsetzung 92

6 Untersuchungen 95
6.1 Versuchsdurchführung 95
6.2 Optimierungsdauer 99
6.2.1 Grapheigenschaften 99
6.2.2 Parameter des Optimierers 105
6.3 Qualität 114

6.3.1 Anzahl der Methoden je Instanz 114
6.3.2 Anzahl der Ausführungszeiten je Methode 115
6.3.3 Minimale worst-case Auslastung 116
6.3.4 Verhältnis durchschnittliche Ausführungsdauer zu worst-case Ausführungsdauer 117
6.3.5 Zeitraster 118
6.4 Energieverbrauch 119
6.4.1 Anzahl der Methoden je Instanz 119
6.4.2 Anzahl der Ausführungszeiten je Methode 120
6.4.3 Minimale worst-case Auslastung 121
6.4.4 Verhältnis durchschnittliche Ausführungsdauer zu worst-case Ausführungsdauer 122
6.4.5 Anzahl der Leistungsstufen 123
6.5 Anytime-Profile 124
6.5.1 Anforderungsgetriebene Dynamische Programmierung . . 125
6.5.2 Simulated Annealing 127
6.6 Vergleich mit anderen Schedulern 132
6.7 Bewertung der Untersuchungen 133

7 Zusammenfassung 135

A Fallstudie RoboCup 139
A.1 Allgemeiner Ablauf 139
A.1.1 Modellbildung 140
A.1.2 Anwendungsoptimierung und Modellverfeinerung 144
A.2 RoboCup-Anwendung 144
A.2.1 Modellierung 145
A.2.2 Initiale Messung der Methodenlaufzeiten 147
A.2.3 Optimierung 147
A.2.4 Ausführung der Zielanwendung 149
A.2.5 Laufzeitmessung während der Anwendungsausführung . . 152

B Erweiterung auf Mehrprozessorsysteme 155

C Energieverbrauch von Prozessoren 159

D Schrankenbestimmung 161
D.1 Qualitätsabschätzung 161
D.2 Energieabschätzung 163

E Integration in das Framework PASCHA **165**
E.1 Graphischer Editor 165
E.2 Simulator 168
E.3 Visualisierung der Optimierungsphase 170
E.4 Visualisierung der Simulation 170
E.5 Parameter der Schedulingalgorithmen 174

F Klassendiagramme **179**
F.1 Optimierer 179
F.2 Lotsen 180
F.3 Knotenspeicher 181
F.4 Systemmodell 184
F.5 Parameter der Algorithmen 185

Literaturverzeichnis **187**

Abbildungsverzeichnis

1.1 Leistungsbedarf, Versorgungsspannung und Taktfrequenz eines Intel® Pentium® M Prozessors (nach [Int04]) . . . 2
1.2 PDA mit integriertem GPS-Empfänger und dynamischer Leistungsanpassung . . . 3
1.3 Energiebewußte Video-Anwendung auf einem Mobiltelefon . . . 4
1.4 Auslastung im schlechtesten Fall und bei voller Rechenleistung . . . 6
1.5 Auslastung im schlechtesten Fall und bei variabler Rechenleistung 6
1.6 Auslastung bei voller Rechenleistung mit Berücksichtigung des dynamischen Leerlaufs . . . 7
1.7 Flexibler Schedule mit dynamischer Methodenauswahl . . . 8
1.8 Optimaler, flexibler Schedule . . . 9
1.9 Ablauf der Optimierung und Einplanung im Überblick . . . 10
1.10 Kontrolle des Prozessors bei verschiedenen Anwendungen . . . 12

3.1 Vergleichbare und unvergleichbare Verteilungsfunktionen . . . 38

4.1 Rekursive Zusammensetzung einer optimalen Lösung . . . 42
4.2 Komplexität und Auswertungsreihenfolge bei Dynamischer Programmierung . . . 43
4.3 Einsparungen durch anforderungsgetriebene Dynamische Programmierung (schematisch) . . . 45
4.4 Referenzierung von Teillösungen (schematisch) . . . 47
4.5 Dynamische Erzeugung angeforderter Zellen (schematisch) . . . 48
4.6 Einige Typen von Leistungsprofilen . . . 49
4.7 Unvergleichbare Leistungsprofile . . . 49
4.8 Aufbau eines Bedingungsgraph-Knotens . . . 50
4.9 Ausschnitt eines Bedingungsgraphen . . . 51
4.10 Bedingungsgraph Suchraum . . . 54
4.11 Optimaler Bedingungsgraph . . . 62
4.12 Kompakter optimaler Bedingungsgraph . . . 62
4.13 Optimaler Bedingungsgraph (DVD) . . . 63

4.14 Kompakter, optimaler Bedingungsgraph (DVD) 63
4.15 Fast optimaler Bedingungsgraph (DVD, Simulated Annealing) . . 64
4.16 Kompakter, fast optimaler Bedingungsgraph (DVD, Simulated Annealing) 64
4.17 Endlicher Automat für Live-Video Anwendung 66
4.18 Pseudoquelltext des dynamischen Schedulers 67
4.19 Pseudoquelltext der Fehlerbehandlung 68

5.1 Zusammenspiel von Optimierer, Knotenspeicher und Lotse 70
5.2 Mikro- und Makroschritte 70
5.3 Kodierung der Leistungsstufen in Methodenkopien 73
5.4 Pseudoquelltext der sortierten Lotsen 74
5.5 Pseudoquelltext des Zufallslotsen 76
5.6 Aufbau des Knotenspeichers 77
5.7 Qualitätsinversion durch Qualitätsänderung eines Sohnes 79
5.8 Knotenzustände 80
5.9 Änderung mehrerer Knoten und verzögerte Aktualisierung 82
5.10 Pseudoquelltext der zentralen Knotenspeichermethoden 83
5.11 Initialisierung und Iteration über alle möglichen Startkombinationen 87
5.12 Rekursive Optimierung der Knoten 88
5.13 Jeder zulässige Knoten wird akzeptiert. 89
5.14 Pseudoquelltext des Simulated Annealing-Algorithmus 90
5.15 Pseudoquelltext der Temperaturanpassung 91
5.16 Pseudoquelltext der Auswahl des Opferknotens 91
5.17 Pseudoquelltext für Akzeptanz von Knoten 92

6.1 Ablauf einer Untersuchung für Modelleigenschaften 97
6.2 Ablauf einer Untersuchung für Parameter des Optimierers 98
6.3 Legende der Graphen 98
6.4 Anzahl der Instanzen → Optimierungsdauer in Millisekunden . . 100
6.5 Anzahl der Zusammenhangskomponenten → Optimierungsdauer . 101
6.6 Anzahl der Methoden → Optimierungsdauer 102
6.7 Anzahl der Ausführungszeiten je Methode → Optimierungsdauer 103
6.8 Minimale worst-case Auslastung → Optimierungsdauer 104
6.9 Durchschnittliche Ausführungsdauer/worst-case Ausführungsdauer → Optimierungsdauer 105
6.10 Anzahl der Leisungsstufen → Optimierungsdauer 106
6.11 ADP: Lotse → Optimierungsdauer (Kodierung siehe Tabelle 6.1) . 107
6.12 SA: Initialisierungslotse → Optimierungsdauer (Kodierung siehe Tabelle 6.1) 107
6.13 ADP: Knotenspeichergröße → Optimierungsdauer 109

6.14 SA: Knotenspeichergröße → Optimierungsdauer 109
6.15 Aktivierung der Zeit-Abbruchsbedingung → Optimierungsdauer . 110
6.16 Aktivierung der Zeit-Abbruchsbedingung → Optimierungsdauer . 111
6.17 Aktivierung der Qualität-Abbruchsbedingung → Optimierungsdauer 112
6.18 Aktivierung der Qualität-Abbruchsbedingung → Optimierungsdauer 112
6.19 Feinheit des Zeitrasters → Optimierungsdauer 113
6.20 Feinheit des Zeitrasters → Optimierungsdauer 114
6.21 Anzahl der Methoden je Instanz → Qualität 115
6.22 Anzahl der Ausführungszeiten je Methode → Qualität 116
6.23 Minimale worst-case Auslastung → Qualität 117
6.24 Durchschnittliche Ausführungszeit/worst-case Ausführungszeit → Qualität 118
6.25 Feinheit des Zeitrasters → Qualität 119
6.26 Anzahl der Methoden je Instanz → Energie 120
6.27 Anzahl der Ausführungszeiten je Methode → Energie 121
6.28 Minimale worst-case Auslastung → Energie 122
6.29 Durchschnittliche Ausführungszeit/worst-case Ausführungszeit → Energie 123
6.30 Anzahl Leistungsstufen → Energie 124
6.31 Qualitätsbruchteil → Optimierungsdauer in Millisekunden 125
6.32 Qualitätsbruchteil → erzeugte Knoten 126
6.33 Qualitätsbruchteil → durchgeführte Aktualisierungen 127
6.34 Qualitätsbruchteil → Optimierungsdauer 128
6.35 Qualitätsbruchteil → Optimierungsdauer in Millisekunden 128
6.36 Qualitätsbruchteil → erzeugte Knoten 129
6.37 Qualitätsbruchteil → durchgeführte Aktualisierungen 130
6.38 Qualitätsbruchteil → Optimierungsdauer 131
6.39 Energieeinsparungen durch erweitertes Modell 132
6.40 Energieverbrauch bei unterschiedlich vielen Leistungsmodi 133
6.41 Energieverbrauch verschiedener Scheduling-Verfahren 134

A.1 Datenflußdiagramm des Werkzeuges für die Anwendungsoptimierung 141
A.2 Konfiguration der Bestandteile einer Anwendung 143
A.3 RoboCup-Roboter im Spielfeld 144
A.4 Datenflußmodell der RoboCup-Anwendung 146
A.5 Initialer Schedule der RoboCup-Anwendung 150
A.6 Architektur der Echtzeiterweiterung RTAI für Linux 151
A.7 Quelltext zur Erzeugung des RTAI-Prozesses 151

A.8 Main-Methode der RoboCup-Anwendung 152

B.1 Beispielanwendung für Mehrprozessorscheduling 157
B.2 Bedingungsgraph für Mehrprozessorscheduling 158

D.1 ILP für obere Qualitätsschranke 162
D.2 ILP für untere Energieschranke 164

E.1 Oberfläche des PASCHA-Editors 166
E.2 Parametereinstellungen im PASCHA-Editor 167
E.3 Ausführungszeitverteilung und Qualität einer Methode 168
E.4 Oberfläche des PASCHA-Simulators 168
E.5 Visualisiierung der Optimierungsphase 169
E.6 Visualisierung eines Simulationszeitpunktes in der Graphansicht . 171
E.7 Visualisierung des zeitlichen Verlaufes 172
E.8 Konfiguration des Optimierers (Lotse und Suchalgorithmus) . . . 173

F.1 Klassendiagramm der Optimierer 180
F.2 Klassendiagramm der Lotsen 181
F.3 Klassendiagramm des Knotenspeichers 183
F.4 Klassendiagramm des Anwendungsmodells 184
F.5 Klassendiagramm der Parameter der Optimierer 186

Symbolverzeichnis

Prozesse	
P	Menge aller periodischen Prozesse
P_i	i-ter Prozeß (Menge aller Instanzen des i-ten Prozesses)
$P_{i,j}$	j-te Instanz des Prozesses P_i
p_i	Periodendauer des Prozesses P_i
b_i	Bereitzeit des Prozesses P_i relativ zum Periodenbeginn
f_i	Frist des Prozesses P_i relativ zum Periodenbeginn
p	Hyperperiode aller Prozesse aus P
I	Menge aller Instanzen in einer Hyperperiode
I_i	einfach indizierte Instanz i
$b_{i,j}$	Bereitzeit der j-ten Instanz von Prozeß i
$f_{i,j}$	Frist der j-ten Instanz von Prozeß i
$\hat{b}_{i,j}$	effektive Bereitzeit der j-ten Instanz von Prozeß i
$\hat{f}_{i,j}$	effektive Frist der j-ten Instanz von Prozeß i
Methoden	
M	Menge aller Methoden aller Prozesse
M_i	Menge aller Methoden für Prozeß P_i
$M_{i,j}$	j-te Methode für Prozeß P_i
$A_{i,j}$	Wahrscheinlichkeitsverteilung der benötigten Instruktionseinheiten der Methode $M_{i,j}$
$q_{i,j}$	Qualität der Methode i
$K_{i,j}$	Menge der mit Methode $M_{i,j}$ kompatiblen Prozessortypen
e_i	(erwarteter) Energieverbrauch der Methode i
a_i	erwartete Instruktionseinheiten von Methode i
Prozessor	
Z	Menge aller Prozessoren
Z_i	Prozessortyp
$L_{i,j}$	j-te Leistungsstufe des Prozessortyps Z_i
$r(i,j)$	Anzahl der pro Millisekunde in Modus $L_{i,j}$ bearbeiteten Instruktionseinheiten
$e_b(i,j)$	Energieverbrauch je Millisekunde rechnend in Modus $L_{i,j}$
$e_s(i,j)$	Energieverbrauch je Millisekunde schlafend in Modus $L_{i,j}$
$e_w(i,j,k)$	Energieverbrauch für Übergang von Modus $L_{i,j}$ zu Modus $L_{i,k}$
$t_w(i,j,k)$	Zeitverbrauch für Übergang von Modus $L_{i,j}$ zu Modus $L_{i,k}$

Kapitel 1

Einführung

Eingebettete Systeme[1] werden in immer mehr Bereichen des alltäglichen Lebens eingesetzt. Moderne Fahrzeuge beherbergen eine Vielzahl von eingebetteten Prozessoren, und mobile Geräte für Endverbraucher verfügen über Rechenleistungen, die vor wenigen Jahren nur in Personalcomputern zu finden waren. Der Preis für diese gesteigerte Rechenleistung ist ein relativ hoher Energieverbrauch dieser Geräte, auch wenn die verfügbare Rechenleistung wegen fehlender Aufträge nur in kurzen Zeitintervallen genutzt wird. Batterien und Akkus müssen daher immer mehr Kapazität besitzen, um bei gleichem Rechenleistungsbedarf in Standardsituationen die Geräte für die gleiche Zeit zu versorgen.

> „*The processor speed in handsets is on the same curve as those for the PCs of a decade earlier, albeit with enormous focus on energy efficiency. [...] The processing of media data in these systems presets many challenges. These functions must be timely - even real-time - but must also be energy and bandwidth efficient.* “ (Gupta [Gup04])

Eine Lösung für dieses Problem stellt energiebewußtes Scheduling dar (z.B. [Ram04]). Energiebewußtes Scheduling nutzt die Eigenschaft moderner Prozessoren aus, in verschiedenen Leistungsstufen arbeiten zu können. Die verbrauchte Leistung sinkt hierbei in etwa kubisch im Verhältnis zur Reduzierung der Rechenleistung[2]. Der Grund dafür sind die Proportionalität $P \propto f$ des Leistungsverbrauchs P von C-MOS Prozessoren zu ihrer Taktfrequenz f und der Propor-

[1]Eingebettete Systeme (engl.: embedded systems): „*Embedded systems can be defined as information processing systems embedded into enclosing products such as cars, telecommunication or fabrication equipment. Such systems come with large number of common characteristics, including real-time constraints, and dependability as well as efficiency requirements. Embedded system technology is essential for providing ubiquitous information, one of the key goals of modern information technology (IT).*“ (Marwedel [Mar03])

[2]Für eine gegebene Architektur steigt die Rechenleistung indirekt proportional zur Taktfrequenz.

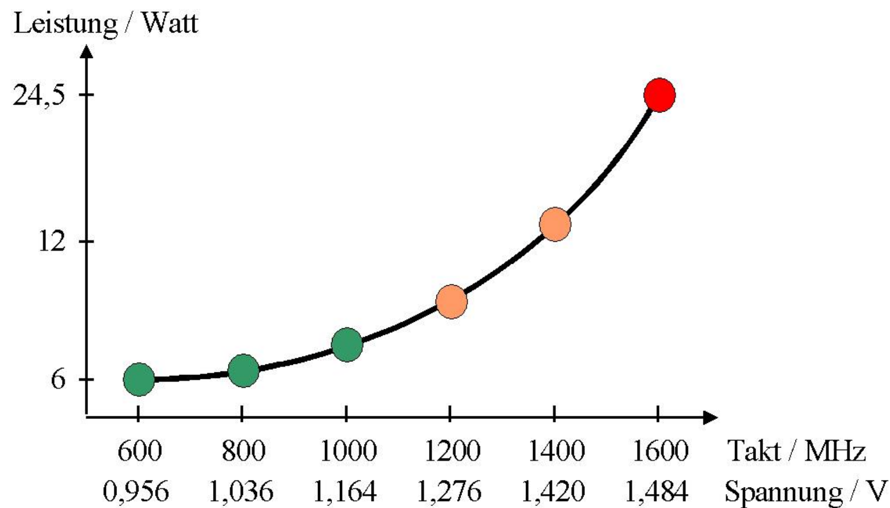

Abbildung 1.1: Leistungsbedarf, Versorgungsspannung und Taktfrequenz eines Intel® Pentium® M Prozessors (nach [Int04])

tionalität $P \propto V^2$ des Leistungsverbrauchs zum Quadrat der anliegenden Versorgungsspannung V. Da diese proportional zur Taktfrequenz ($V \propto f$) gesenkt werden kann, ergibt sich $P \propto V^2 \cdot f \propto f^3$. Zusammen mit der indirekt proportional zur Taktfrequenz steigenden Ausführungsdauer für eine Anwendung reduziert eine Halbierung der Taktfrequenz und der Versorgungsspannung damit den Energieverbrauch auf ein Viertel. Abbildung 1.1 zeigt den Zusammenhang zwischen Versorgungsspannung, Taktfrequenz und Leistungsverbrauch für einen Intel® Pentium® M Prozessor. Im Ergebnis kann man die Nutzdauer bedarfsgerecht verlängern, ohne auf die Spitzenleistung ganz verzichten zu müssen.

Ein schwierigere Situation tritt bei Anwendungen mit (harten) Echtzeitbedingungen und Qualitätsanforderungen [Ram03] auf, da dann die Leistungsfähigkeit des Systems auch im schlechtesten Fall[3] ausreichen muß, um alle Prozesse rechtzeitig abzuarbeiten. Da die Systeme auf diesen Rechenleistungsbedarf ausgelegt werden müssen, der aber nur sehr selten auftritt, sind die finanziellen Kosten unnötig hoch. Auch qualitätsbewußtes Scheduling beschäftigt sich mit der besseren Ausnutzung der verfügbaren Rechenleistung. Um dies zu ermöglichen, können für die Umsetzung eines Prozesses verschiedene Implementierungen angegeben werden, die sich hinsichtlich ihres Zeitaufwandes und ihrer Ergebnisqualität unterscheiden. Ein qualitätsbewußter Scheduler wählt dann zur Laufzeit die am besten geeignete Implementierung aus, um die gelieferte Qualität zu erhöhen oder um die Zeitbedingungen auch im schlimmsten Fall - mit eventuell reduzierter Qualität - einzuhalten. Durch diese Wahlmöglichkeit lassen sich die Anforde-

[3]engl.: worst-case execution time

Navigationsanwendung (energiekritisch) | Videoanwendung (qualitätskritisch)

Abbildung 1.2: PDA mit integriertem GPS-Empfänger und dynamischer Leistungsanpassung

rungen an das System reduzieren, da eine günstigere Hardware-Plattform, die im Normalfall optimale und im schlimmsten Fall immer noch ausreichende Qualität liefert, gewählt werden kann.

Abbildung 1.2 zeigt einen *Personal Digital Assistant* (PDA), der aufgrund des integrierten GPS-Empfängers[4] zur Navigation auf Wander- oder Fahrradtouren benutzt werden kann. Eine weitere Anwendungsmöglichkeit des Gerätes ist die Wiedergabe von Videodateien. Der PDA verfügt über einen Intel PXA 255 Prozessor mit drei Leistungsstufen (max. 300 MHz), 64 MB Speicher und wiegt bei den Abmessungen von 112 x 70 x 16 Millimetern nur 152 Gramm. Der integrierte Akku besitzt eine Kapazität von 5 Wattstunden. Damit kann der PDA mit automatischer Anpassung der Rechenleistung durch das Betriebssystem und mit aktiviertem GPS-Empfänger zirka $4\frac{1}{2}$ Stunden betrieben werden. Während bei der Navigationsanwendung kleine Verzögerungen der Positionsbestimmung aufgrund der dynamischen Leistungsanpassung akzeptabel sind, führt die Leistungsanpassung beispielsweise bei der Videowiedergabe mit den deutlich strengeren Echtzeitanforderungen eventuell zu Rucklern, die die empfundene Wiedergabequalität stark mindern. Hier wäre eine Lösung, die die Leistungsanpassung der Anwendung überläßt, besser. Insbesondere ist es wünschenswert, die Wiedergabequalität

[4]GPS: global positioning system

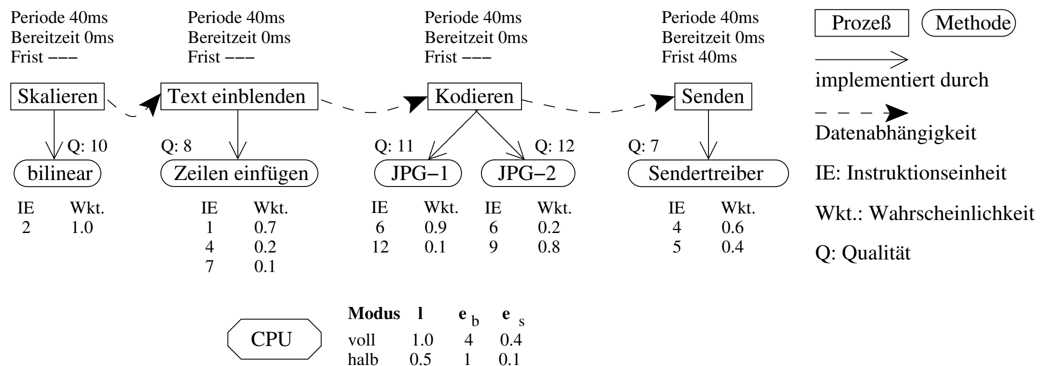

Abbildung 1.3: Energiebewußte Video-Anwendung auf einem Mobiltelefon

zu maximieren, während zugleich der Energieverbrauch so gering gehalten wird, daß das komplette Video betrachtet werden kann. Die Strategie, die das Nebenwissen der Anwendung über die internen Prozesse ausnutzt, liegt auch dem in dieser Arbeit verwendeten Scheduler zu Grunde.

Die beiden Scheduling-Ziele, energiebewußte und qualitätsbewußte Einplanung, basieren auf ähnlichen Voraussetzungen. Es sind mehrere Prozesse unter Echtzeitbedingungen abzuarbeiten. Oftmals sind Prozesse von anderen Prozessen datenabhängig, und sie können auf verschiedene Weisen ausgeführt werden (mehrere Leistungsstufen oder Implementierungen). Zusätzlich ist es wünschenswert, daß Schwankungen der Ausführungszeit eines Prozesses, die z.B. von der Art der Eingabedaten abhängen, für nachfolgende Prozesse genutzt werden, um die Qualität zu verbessern oder durch die Wahl einer niedrigeren Leistungsstufe den Energieverbrauch zu verringern.

1.1 Beispielanwendung

Abbildung 1.3 zeigt das Modell einer einfachen Anwendung für ein Mobiltelefon. Der Benutzer will ein Video aufzeichnen und es in Echtzeit an einen Freund senden. Es sollen 25 Bilder pro Sekunde übertragen werden, und während der Aufzeichnung kann der Benutzer auf Knopfdruck z.B. Datum, Uhrzeit und eine vorher eingegebene Textzeile einblenden. Da der Akku des Mobiltelefons nur einen beschränkten Energievorrat speichern kann, ist das Ziel der Prozeßeinplanung ein minimaler durchschnittlicher Energieverbrauch, um das Gerät möglichst lange ohne neues Laden des Akkus betreiben zu können.

Die Anwendung besteht aus vier Prozessen, deren Reihenfolge durch die Datenabhängigkeiten vorgegeben ist. Der erste Prozeß (*Skalieren*) hat die Aufgabe, die für die Übertragung zu hohe Auflösung der Bilder zu reduzieren. Anschlie-

Modus	rechnend	schlafend
voll	4.0	0.4
halb	1.0	0.1

Tabelle 1.1: Energieverbrauch der Prozessormodi

ßend fügt der zweite Prozeß (*Text einfügen*) - abhängig von der vom Benutzer gedrückten Taste - das Datum, die Uhrzeit und die vorher eingegebene Textzeile ein. Der Prozeß *Kodieren* wandelt das entstandene Bild in das JPG-Format um, um die zu übertragende Datenmenge zu verringern, bevor es schließlich vom Prozeß *Senden* übertragen wird. Da zunächst keine relativen Verzögerungen der Prozesse vorgegeben sind, haben alle die Bereitzeit 0. Aus der Anforderung, 25 Bilder pro Sekunde zu übertragen, ergibt sich für alle Prozesse die Periodendauer 40 Millisekunden und für den *Senden*-Prozeß eine harte Frist von 40 Millisekunden.

Für die Prozesse *Skalieren*, *Text einblenden* und *Senden* existiert jeweils nur eine Implementierung (Methode), während für den *Kodieren*-Prozeß die zwei Methoden *JPG-1* und *JPG-2*, die sich hinsichtlich ihrer durchschnittlichen und ihrer worst-case Ausführungsdauer unterscheiden, zur Verfügung stehen. Alle Methoden mit Ausnahme der *bilinear*en Skalierung benötigen bei ihrer Ausführung eine nicht-deterministische Anzahl von Instruktionseinheiten.[5] Für diese Schwankungen gibt es verschiedene Ursachen. Während bei der Methode *Zeilen einfügen* die Anzahl der Instruktionseinheiten davon abhängt, welche Taste der Benutzer drückt, d.h. wieviele Zeilen eingefügt werden müssen, ist sie bei der JPG-Komprimierung vom Inhalt des Bildes abhängig. Die Datenmenge des komprimierten Bildes bestimmt die Anzahl der Instruktionseinheiten, die die Methode *Sendertreiber* zur Übertragung benötigt.

Das Mobiltelefon, auf dem die Video-Anwendung ausgeführt werden soll, besitzt einen Prozessor, der mit zwei Taktraten betrieben werden kann (Tabelle 1.1). Mit der ersten Taktrate (*voll*) kann der Prozessor definitionsgemäß eine Instruktionseinheit pro Millisekunde verarbeiten und verbraucht dabei vier Energieeinheiten, wenn er Berechnungen durchführt, und 0.4 Energieeinheiten, wenn er im Leerlauf ist. Bei der zweiten Taktrate werden 0.5 Instruktionseinheiten pro Millisekunde mit einem Energieverbrauch von einer Einheit im Rechenbetrieb verarbeitet, und es werden 0.1 Einheiten je Millisekunde Leerlauf verbraucht.

Um den möglichst langen Betrieb des Mobiltelefons ohne Stromnetzanschluß zu ermöglichen, ist die Zielfunktion die Minimierung der pro Zeiteinheit verbrauchten Energie. Der gesuchte flexible Schedule muß also alle Prozesse so ein-

[5] Instruktionseinheit: Anzahl der pro Millisekunde in der höchsten Leistungsstufe ausgeführten Instruktionen

planen, daß die gewählten Methoden immer rechtzeitig vor den gegebenen Fristen beendet sind, aber der Prozessor möglichst lange in der niedrigeren Leistungsstufe betrieben werden kann.

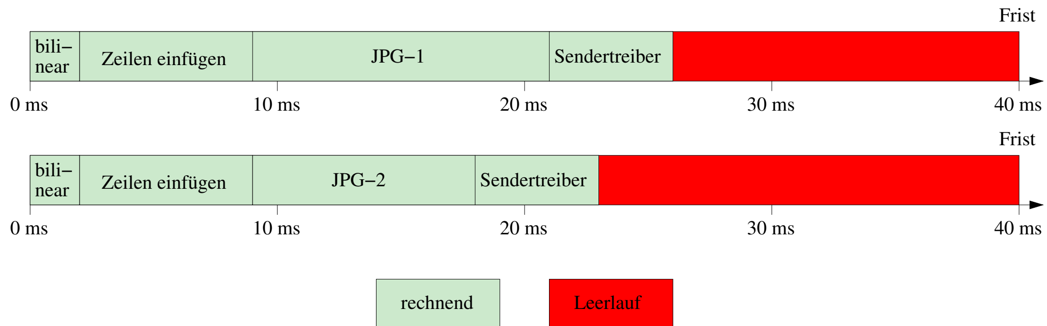

Abbildung 1.4: Auslastung im schlechtesten Fall und bei voller Rechenleistung

Abbildung 1.4 zeigt die Auslastung des Prozessors bei voller Rechenleistung, wenn alle Methoden ihre maximale Ausführungsdauer benötigen. Die Verwendung der Methode *JPG-1* führt zu einer maximalen Gesamtausführungszeit von 26 Millisekunden, während bei Verwendung von *JPG-2* nur 23 Millisekunden benötigt werden, d.h. es ist ein geringerer Energieverbrauch zu erwarten. Die statische Analyse des schlimmsten Falles legt also den Einsatz von Methode *JPG-2* nahe.

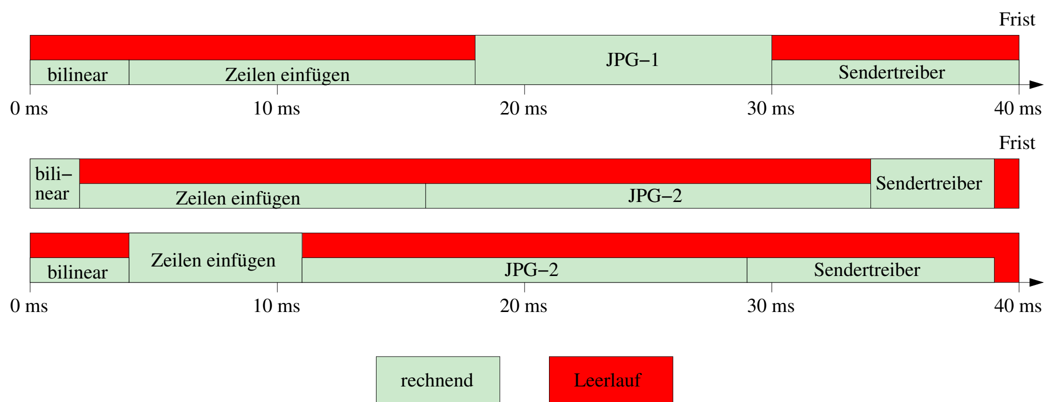

Abbildung 1.5: Auslastung im schlechtesten Fall und bei variabler Rechenleistung

Auch bei der zusätzlichen Verwendung des Leistungsmodus *halb* bleibt dieses Verhältnis erhalten. Abbildung 1.5 zeigt den maximal möglichen Einsatz des niedrigeren Leistungsmodus. Bei der Ausführung von *JPG-1* mit voller Leistung können alle anderen Methoden mit halber Leistung ausgeführt werden. Wird *JPG-2* zur Kodierung des Bildes verwendet, so muß entweder die Methode *Zeilen*

einfügen oder es müssen die Methoden *bilinear* und *Sendertreiber* mit voller Leistung ausgeführt werden. Mit *JPG-1* rechnet der Prozessor für 28 Millisekunden mit halber und für 12 Millisekunden mit voller Leistung. Bei *JPG-2* rechnet der Prozessor nur sieben Millisekunden mit voller Leistung und befindet sich sogar noch für eine Millisekunde im Leerlauf.

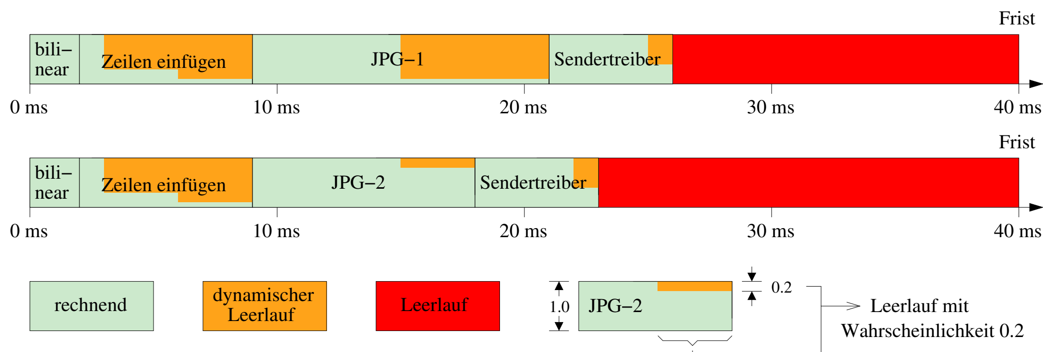

Abbildung 1.6: Auslastung bei voller Rechenleistung mit Berücksichtigung des dynamischen Leerlaufs

Ein anderes Ergebnis ergibt eine statische Analyse, die bereits die Ausführungszeitschwankungen der Methoden berücksichtigt. Abbildung 1.6 zeigt die zu erwartende Auslastung des Prozessors bei voller Rechenleistung. Die durch die Schwankungen hervorgerufenen Leerlaufzeiten (dynamischer Leerlauf) ergeben zusammen mit den worst-case Leerlaufzeiten für die Verwendung von *JPG-1* eine niedrigere zu erwartende Auslastung als für *JPG-2*. Bei einer statischen Zuweisung der verwendeten Rechenleistung zu den verwendeten Methoden kann diese dynamische Leerlaufzeit nicht berücksichtigt werden. Daher wird in dieser Arbeit die für eine Methode verwendete Rechenleistung abhängig von den Ausführungsdauern der bereits beendeten Methoden dynamisch während der Anwendungsausführung festgelegt.

Abbildung 1.7 zeigt einen optimalen flexiblen Schedule für die Beispielanwendung. Die für die Kodierung des Bildes verwendete Methode und Leistungsstufe wird abhängig vom Endzeitpunkt der Methode *Zeilen einfügen* gesetzt. Auf diese Weise können alle Methoden mit halber Leistung ausgeführt werden, wenn *Zeilen einfügen* nur zwei oder acht Millisekunden benötigt. Nur in zehn Prozent aller Fälle, d.h. wenn *Zeilen einfügen* 14 Millisekunden benötigt, wird *JPG-1* mit voller Leistung ausgeführt.

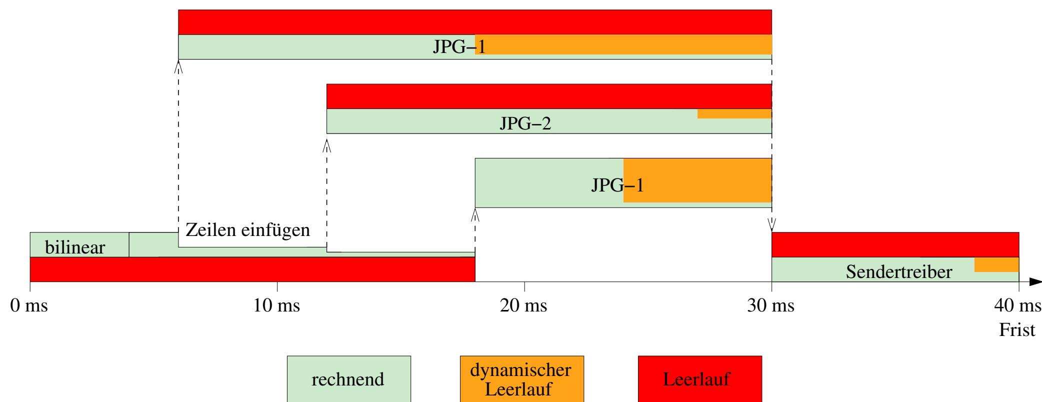

Abbildung 1.7: Flexibler Schedule mit dynamischer Methodenauswahl

1.2 Zweiphasige Einplanung

In dieser Arbeit wird ein gemeinsamer Ansatz für die beiden Aufgaben Energieminimierung und Qualitätsmaximierung vorgestellt. Aufgrund der Komplexität des gewählten Anwendungsmodells ist der Einsatz eines dynamischen Schedulers, der auch die Optimierung der Einplanung zur Laufzeit vornimmt, nicht sinnvoll. Diese Optimierung auf der Zielplattform hätte zwei schwerwiegende Nachteile: Erstens ist die dafür benötigte Zeit relativ hoch und zweitens würde die Optimierung selbst einen Teil der knappen Energie verbrauchen, die dann nicht für die Anwendung genutzt werden könnte.

Aus diesen Gründen wurde im Rahmen dieser Arbeit ein zweiteiliges Verfahren entwickelt. In der ersten Phase wird die Zielfunktion für alle zur Laufzeit auftretenden, wichtigen Situationen optimiert. Dieser Vorgang kann auf einem leistungsfähigen Arbeitsplatzrechner erfolgen und liefert einen mit Nebenbedingungen annotierten Graphen als Ausgabe. Jeder Knoten des Graphen spezifiziert den optimalen Prozeß, die optimale Methode und die optimale Leistungsstufe, die als nächstes zu wählen sind, wenn der Scheduler den Knoten erreicht hat und Parameter des Anwendungsprozesses bestimmte Bedingungen erfüllen. Abbildung 1.8 zeigt einen Graphen, der einen optimalen flexiblen Schedule für die Beispielanwendung aus Abbildung 1.3 darstellt.

Der Graph wird in der zweiten Phase durch den dynamischen Scheduler interpretiert. Er versetzt den Prozessor in die angegebene Leistungsstufe und teilt der auszuführenden Methode den Prozessor zu. Sobald die Methode ihre Berechnungen beendet hat, verzweigt der Scheduler zu dem Sohn, der für die von der Methode tatsächlich verbrauchte Zeit vorgesehen ist. Dieser Vorgang wird bis zu den Blättern des Graphen wiederholt. Sobald der Scheduler die in einem Blatt spezifizierte Methode ausgeführt hat, beginnt er wieder an der Wurzel des Gra-

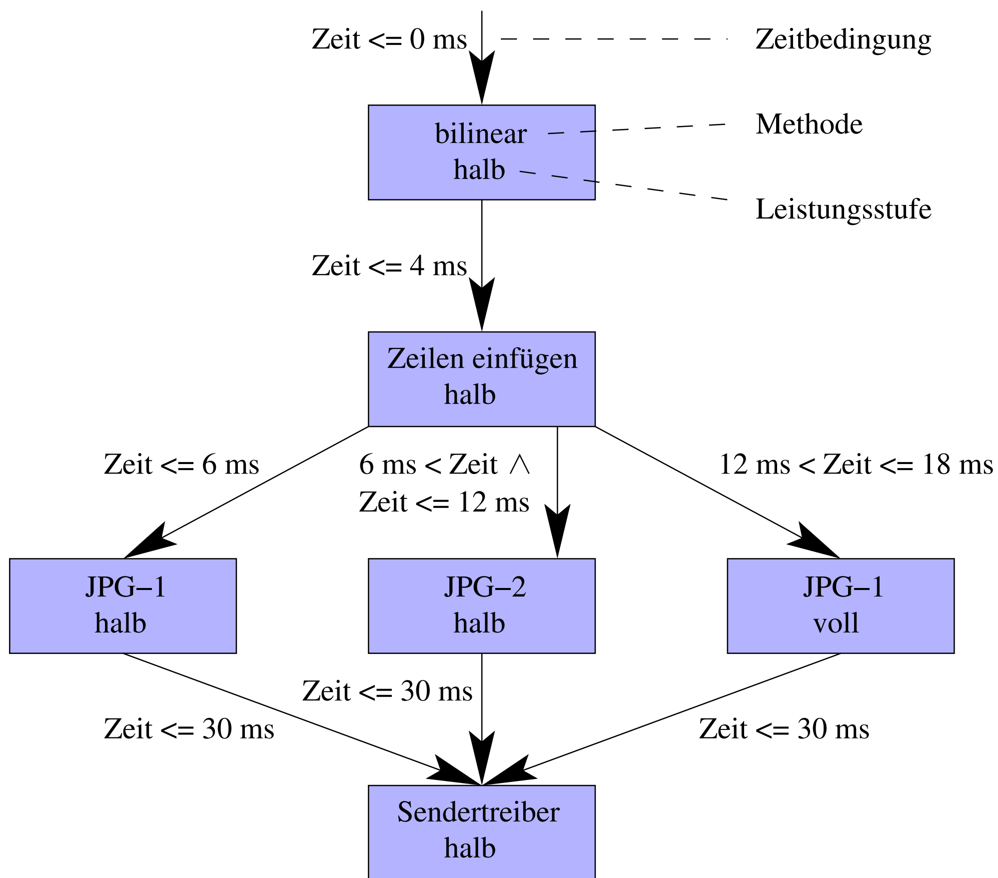

Abbildung 1.8: Optimaler, flexibler Schedule

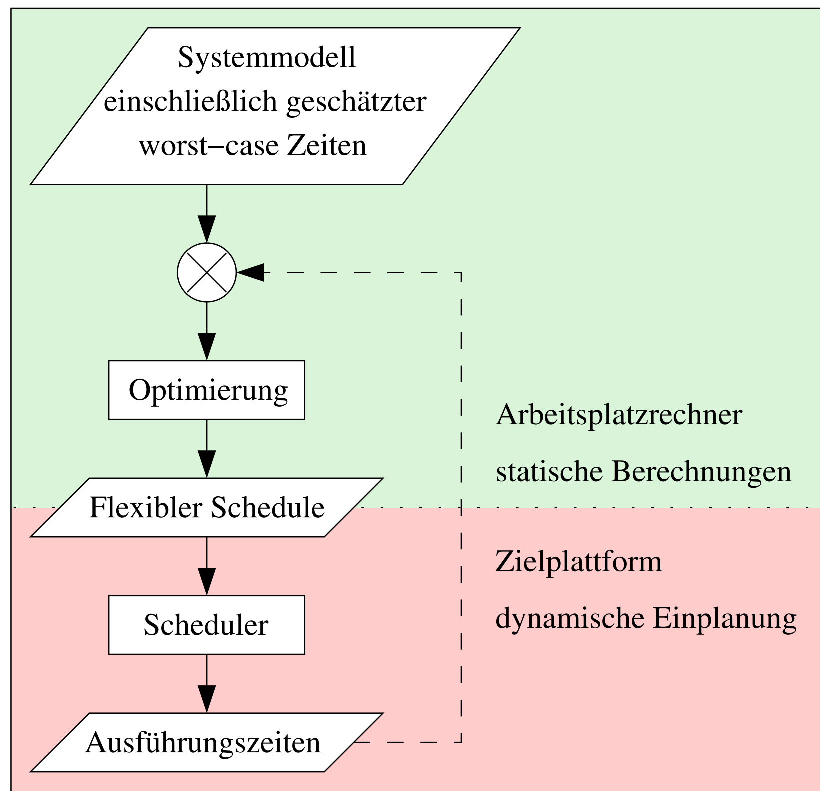

Abbildung 1.9: Ablauf der Optimierung und Einplanung im Überblick

phen. Die zweiphasige Vorgehensweise verringert den während der Laufzeit nötigen Rechenaufwand zur optimalen Einplanung der Prozesse auf ein Maß, das vernachlässigt oder während der Optimierung berücksichtigt werden kann.

Abbildung 1.9 zeigt einen Überblick des Gesamtvorganges. Der Datenaustausch erfolgt über XML-Dateien, wodurch die verschiedenen Phasen auf unterschiedlichen Plattformen ausgeführt werden können. Aufgrund der relativ geringen Rechenleistung der Zielplattform ist es wünschenswert, den Optimierer auf einem leistungsfähigen Rechner auszuführen, während die Messung der Laufzeiten natürlich nur auf der Zielplattform erfolgen kann. Der Anwendungsentwickler erstellt eine XML-Datei, die das Design-to-time-Systemmodell der Anwendung enthält. Dabei steht es ihm frei, einen Text-Editor oder den graphischen Editor des PASCHA-Systems[6] zu verwenden. Das Modell dient zusammen mit bereits existierenden gemessenen oder geschätzten Ausführungszeitwerten[7] als

[6]PASCHA (**PA**ssau **SCH**eduling **A**nalyzer: Ein Rahmenwerk zur Entwicklung und Untersuchung von Scheduling-Algorithmen; siehe Anhang E

[7]Falls Fristüberschreitungen bei der Ausführung der Anwendung zu irreparablen Schäden

Eingabe für den Optimierer, der daraus einen flexiblen Schedule (Entscheidungsgraph) erstellt. Der dynamische Scheduler plant anhand dieses flexiblen Schedules die Anwendung ein, mißt die Ausführungszeiten der ausgeführten Methoden und schreibt sie in eine XML-Datei. Während der Ausführung sind noch Fristüberschreitungen möglich, wenn der Optimierer auf Standard-Laufzeitwerten gearbeitet hatte. Nachdem die Anwendung ausreichend lange ausgeführt wurde, um genügend Ausführungszeitdaten zu sammeln, startet der Benutzer den Optimierer erneut. In diesem zweiten Durchlauf verwendet der Optimierer das Systemmodell und die neuen Ausführungszeitdaten, sodaß nach dieser Optimierung ein auf das Anwendungsgebiet zugeschnittener, flexibler Schedule vorliegt. Jede Ausführung der Anwendung kann wieder genutzt werden, um zusätzliche Ausführungszeitinformationen zu gewinnen, oder diese neu zu messen, falls etwas an der Implementierung der Anwendung geändert wurde.

1.2.1 Optimierung

Die Optimierung der Zielfunktion wird durch eine neu entwickelte Variante der Dynamischen Programmierung bewerkstelligt. Dieses Verfahren vereint die Vorteile von Dynamischer Programmierung mit denen von Anytime-Algorithmen. Es garantiert das (effiziente) Auffinden eines Optimums und erzeugt während der Berechnung eine Folge von Lösungen steigender Güte, die jederzeit zugreifbar sind. Ein weiterer Vorteil gegenüber der klassischen Dynamischen Programmierung ist die Möglichkeit, den Speicherbedarf des Algorithmus zu steuern. Im Gegensatz zur klassischen Implementierung werden die Teillösungen nicht in einer Tabelle gespeichert, deren Einträge von unten nach oben optimiert werden, sondern es werden nur Teillösungen erzeugt und gespeichert, die tatsächlich für die Lösung und Optimierung des Gesamtproblems nötig sind. Die Teillösungen werden in den Knoten eines gerichteten, azyklischen Graphen gespeichert. Die Knoten ersetzen die Zellen der Tabelle und die Kanten repräsentieren die rekursive Referenzierung der Zellen in der Tabelle. Wenn der zugewiesene Speicher mit Knoten gefüllt ist, werden diejenigen Knoten gelöscht, die nicht in der aktuellen Lösung enthalten sind. Falls ein gelöschter Knoten erneut benötigt wird, so muß er zwar neu erzeugt und optimiert werden, jedoch kann hier angenommen werden, daß dieser Fall aufgrund der Suchreihenfolge relativ selten auftritt.[8] Erhält

führen können, müssen die Standardwerte durch Schätzwerte ersetzt werden, die in jedem Fall größer als die längste Ausführungszeit der implementierten Methoden sind.

[8]Es kann angenommen werden, daß der Algorithmus analog zum lokalen Hauptspeicherzugriff großer Programme in der Regel einen Knoten nur über einen gewissen Zeitraum immer wieder benötigt, während er vor und nach diesem Zeitraum nicht angefragt wird. Dieses Zugriffsverhalten erlaubt die Löschung alter Knoten, und damit eine Reduzierung des Speicherbedarfs, ohne große Effizienzeinbußen befürchten zu müssen.

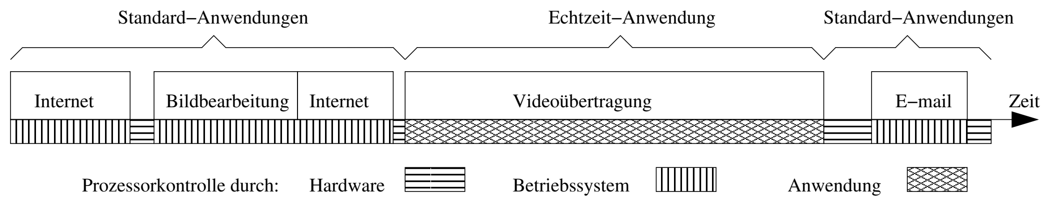

Abbildung 1.10: Kontrolle des Prozessors bei verschiedenen Anwendungen

diese sogenannte anforderungsgetriebene Dynamische Programmierung ausreichend Speicher für alle Knoten, so muß wie im Falle der klassischen Dynamischen Programmierung jeder Knoten nur einmal erzeugt und optimiert werden, jedoch bleibt der Vorteil, daß immer Zwischenlösungen abrufbar sind, bestehen. Ein Schwachpunkt der anforderungsgetriebenen Dynamischen Programmierung ist die zwar relativ gleichmäßige aber langsame Annäherung der Lösungsfolge an eine optimale Lösung. Bei einigen Anwendungen ist es wichtig, gute Lösungen in relativ kurzer Zeit zu finden, wohingegen die Garantie, eine optimale Lösung zu finden, von untergeordneter Bedeutung ist. Für solche Anwendungen wurde ein auf Simulated Annealing basierender Optimierer entwickelt, der ebenfalls gerichtete azyklische Graphen für die Repräsentation von Lösungen verwendet.

1.2.2 Scheduling

Der dynamische Scheduler kann den Betriebssystem-Scheduler auf Systemen ersetzen, auf denen nur eine Anwendung, die aus mehreren Prozessen besteht, permanent ausgeführt wird (z.B. industrielle Anlagen zur Qualitätskontrolle). Auf anderen Systemen, wie z.B. mobilen Multimedia Geräten, kann der Scheduler auch in die verschiedenen Echtzeitanwendungen integriert werden. Während der Ausführung einer solchen Echtzeit-Anwendung übernimmt er die Kontrolle über den Prozessor und sorgt für die optimale Einplanung der anwendungsinternen Prozesse. Sobald die Anwendung beendet wird, erhält der Betriebssystem-Scheduler wieder die Kontrolle über den Prozessor. Durch diese zweite Vorgehensweise wird die Flexibilität des Betriebssystem-Schedulers und damit die des Gerätes beibehalten, während komplexe, rechen- oder energieintensive Anwendungen optimal eingeplant werden.

Abbildung 1.10 zeigt ein Beispiel für den zeitlichen Verlauf der Prozessorkontrolle während der Ausführung verschiedener Anwendungen. Wenn normale Anwendungen wie etwa Internetbrowser oder E-Mail-Programme ausgeführt werden, übernimmt der Betriebsystem-Scheduler die Leistungskontrolle des Prozessors. Er kann beispielsweise verhindern, daß das Gerät in den Schlafmodus wechselt, oder er kann eine zur Auslastung passende Leistungsstufe des Prozes-

sors einstellen. Falls keine Anwendung ausgeführt wird, kann der Betriebssystem-Scheduler die Kontrolle an die Hardware abgeben, die dann z.B. nach einer definierten Dauer ohne Prozessorzugriff in den Schlafmodus wechselt. Während der Ausführung einer Echtzeitanwendung, z.B. einer Videoübertragung, wird die Leistungskontrolle an die Anwendung abgegeben, die die Prozessorleistung mit dem in dieser Arbeit vorgestellten Algorithmus optimal variiert. Da der anwendungsinterne Scheduler die Anforderungen der Anwendung kennt, kann er die Leistungsanpassung so vornehmen, daß die Echtzeitbedingungen eingehalten und trotzdem Energie eingespart werden kann.

1.3 Inhalt und Aufbau der Arbeit

Das übergeordnete Ziel der Arbeit ist es, einen dynamischen Echtzeit-Scheduling-Algorithmus zu entwickeln, der durch geeignete Auswahl von Methoden und - im Falle der Energieminimierung - Leistungsstufen die gegebene Zeit optimal nutzt. Der entwickelte Ansatz beschränkt sich nicht auf die Entwicklung des Scheduling-Algorithmus. Er beinhaltet ein praxisnahes Modell, einen neuartigen Optimierer, der die Stärken von Anytime-Algorithmen mit denen von Simulated Annealing bzw. Dynamischer Programmierung vereint. Die praktische Einsetzbarkeit des Gesamtalgorithmus wird durch eine Fallstudie [RCH04] mit einem Fußball spielenden Roboter gezeigt.

Das nächste Kapitel enthält einen Überblick zu Arbeiten auf verschiedenen Forschungsgebieten, die auch in dieser Arbeit tangiert werden. Anschließend folgt in Kapitel 3 die Vorstellung des verwendeten Modells. Kapitel 4 enthält die Ausführungen zu den beiden Phasen des Gesamtalgorithmus, bevor ihre Umsetzung in Kapitel 5 erläutert wird. Die Ergebnisse, der zu den vorgestellten Algorithmen vorgenommenen Untersuchungen, befinden sich in Kapitel 6. Die Zusammenfassung in Kapitel 7 schließt den Hauptteil der Arbeit ab. Der erste Anhang A stellt eine Fallstudie eines Fußball spielenden Roboters vor. Anhang B erläutert die notwendigen Erweiterungen für heterogene Mehrprozessorsysteme, und Anhang C enthält Details zum Energieverbrauch von CMOS-Prozessoren. In Anhang D befinden sich ILP-Modelle zur Abschätzung der erzielbaren Qualität und Energie mit CPLEX. Abschließend folgen Anhänge mit Details zum Rahmenwerk PASCHA (Anhang E) und der Implementierung (Anhang F).

Kapitel 2

Stand der Technik

Die folgenden Abschnitte geben einen kurzen Überblick zu den in dieser Arbeit tangierten Forschungsbereichen. Der erste Abschnitt befaßt sich mit Arbeiten zu Echtzeit-Scheduling, und im zweiten Abschnitt sind einige Veröffentlichungen zu den hier zugrunde liegenden Optimierungsalgorithmen Dynamische Programmierung und Simulated Annealing aufgeführt.

2.1 Scheduling

Die in dieser Arbeit vorgestellten Anwendungsmodelle und Schedulingverfahren unterscheiden sich grundlegend von *Job-Shop*-Scheduling und den klassischen Echtzeit-Schedulingverfahren *Rate-monotonic*- und *Deadline-monotonic*-Scheduling. Das Ziel beim Job-Shop-Scheduling (z.B. [Ste99]) ist, eine Ausführungsreihenfolge für eine Prozeßmenge zu finden. Dabei müssen alle Nebenbedingungen eingehalten werden, und die Zeit bis zur Beendigung des letzten Prozesses ist zu minimieren. Echtzeit-Schedulingverfahren wie etwa *Rate-monotonic*-Scheduling (z.B. [LL73, Liu00, PL95, KS97]) haben in der Regel nur das Ziel einen gültigen Schedule für die Prozeßmenge zu finden, und sie arbeiten auf einem auf worst-case Ausführungsdauern beschränkten Modell. Der in dieser Arbeit vorgestellte Algorithmus hat das Ziel, die Prozeßmenge unter Berücksichtigung aller Nebenbedingungen einzuplanen und zusätzlich eine Zielfunktion zu optimieren. Im Gegensatz zum Job-Shop-Scheduling ist hier nicht die benötigte Zeit zu minimieren, sondern die verfügbare Zeit vorgegeben und sie soll optimal genutzt werden. Die in dieser Arbeit vorgestellten Algorithmen zielen auf Systeme ab, die bei ausschließlicher Verwendung der jeweils schnellsten Implementierungen nicht in Überlast geraten, bei denen jedoch bei ausschließlicher Verwendung der energiesparenden oder qualitativ besseren Methoden eine Überlastsituation vorliegt. Die zur Einplanung benötigten Ausführungszeiten können durch Messungen

während der Anwendungsausführung [RCH04] ermittelt werden. Falls dies nicht möglich ist, können die Ausführungszeiten durch formale Analyse des Quelltextes [JHE04], z.B. mit einem Werkzeug wie *Symta/P* [SE04], oder mit einem Framework zur Analyse von Systemeigenschaften, wie z.B. *Real-Time Calculus* [CKT03, MCKT04], bestimmt werden.

2.1.1 Qualitätsbewußtes Scheduling

Moderne Computersysteme dienen zur Lösung immer komplexerer Aufgaben, für die es meistens nicht nur eine, sondern eine Vielzahl von Lösungen gibt. Einerseits reicht es oft nicht aus, eine beliebige Lösung zu finden, und andererseits ist es nicht nötig oder aufgrund zeitlicher Nebenbedingungen nicht möglich eine qualitätsoptimale Lösung zu finden. Qualitätsbewußte Scheduling-Algorithmen versuchen daher die zur Verfügung stehende Zeit zu nutzen, um möglichst gute – zumindest aber zufriedenstellende – Lösungen zu finden. Die Definition und Ermittlung der Güte oder Qualität eines Algorithmus ist sehr problemspezifisch und in vielen Bereichen Gegenstand aktueller Forschung [BDTL96, SRS$^+$01]. Für Qualitätsmaße gibt es verschiedene Kategorien [Zil93], von denen eine oder mehrere für einen Algorithmus anwendbar sein können: Genauigkeit (z.B. Entfernung), Spezifität (z.B. Objekttyp), Sicherheit (z.B. tatsächliches Vorhandensein). Auch die Quantisierung der Qualitätsmerkmale ist in den meisten Anwendungsbereichen unerforscht oder nicht zufriedenstellend geklärt. Oftmals ist die erreichbare Qualität unbekannt oder sie kann nicht automatisch abgeschätzt werden, aber es gibt auch Bereiche in denen dies möglich ist, wie z.B. numerische Verfahren zur Approximation von Nullstellen. Diese Arbeit setzt die benötigten Qualitätsmaße als gegeben voraus und plant Algorithmen anhand der ihnen zugewiesenen Qualitäten ein.

Es gibt verschiedene Ansätze für qualitätsbewußtes Scheduling, z.B. *Imprecise Computation* ([HFL96]) oder *Anytime-Scheduling* ([DB88, Hor90, Cam00, Zil93]), die sich hinsichtlich der verwendbaren Algorithmen unterscheiden. Bei Anytime-Scheduling können zwei Arten von Algorithmen zum Einsatz kommen. *Unterbrechbare* Anytime-Algorithmen können jederzeit beendet werden und liefern umso bessere Ergebnisse je später dies geschieht. Im Gegensatz dazu muß bei *Vertrags*-Anytime-Algorithmen schon vor deren Ausführung bekannt sein, wieviel Zeit ihnen zur Verfügung steht. Aufgrund dieser Bedingung sind Vertrags-Algorithmen leichter zu implementieren, weil sie keine Zwischenergebnisse liefern müssen und vor ihrem Berechnungsbeginn für die gegebene Ausführungszeit konfiguriert werden können. Sie haben aber den Nachteil, daß sie kein Ergebnis liefern (müssen), wenn sie vor der zugesicherten Zeit beendet werden. Die Arbeiten von Schwarzfischer [Sch03b, Sch03a, Sch04] befassen sich mit der Zusammenführung von Anytime-Scheduling mit fristgerechtem Scheduling, d.h. einem

einheitlichen Modell für Qualitäts- und Nutzenfunktionen der Prozesse.

Design-to-time-Scheduling wurde vor allem durch die Arbeiten von Garvey und Lesser [Dec96, GL93, GL96, GL95] bekannt und verfolgt die gleiche Zielsetzung wie Anytime-Scheduling. Der Unterschied liegt darin, daß beim Design-to-time-Ansatz Mengen von Lösungsalgorithmen für jeden Prozeß spezifiziert werden. Jede dieser Mengen enthält Algorithmen, die für einen Prozeß unterschiedlich gute Ergebnisse in unterschiedlich langer Zeit liefern, d.h. länger rechnende Algorithmen liefern bessere Ergebnisse. Alle Algorithmen einer Menge müssen nur das gleiche Ein- und Ausgabeformat besitzen und ihre Laufzeit und ihre Ergebnisgüte müssen bekannt sein. Durch die Abschwächung[1] der Anforderungen an die Implementierungen eines Prozesses ergibt sich ein grundlegend anderes Optimierungsproblem. Während bei den beiden Anytime-Modellen eine analytische Optimierung theoretisch denkbar wäre, weil der Schedulingalgorithmus die Laufzeit der Implementierungen frei bestimmen kann, handelt es sich beim dritten Modell um ein kombinatorisches Problem, da der Schedulingalgorithmus eine optimale Kombination der Methoden, die diskrete Laufzeiten besitzen, wählen muß. In dieser Arbeit wird das dritte Modell verwendet.

Garvey und Lesser haben sich sowohl mit deterministischen Methodenausführungszeiten [GL93] als auch mit nicht-deterministischen Methodenausführungszeiten [GL95, GL96] beschäftigt. In ihrem technischen Bericht zum nicht-deterministischen Modell spezifizieren sie Methoden nicht nur durch die Ausführungszeit im schlechtesten Fall, sondern sie geben jeweils eine diskrete Wahrscheinlichkeitsverteilung an. Ihr dynamischer Scheduling-Algorithmus berechnet zunächst einen Schedule für die einzuplanenden Prozesse. Die Ausführung der Prozesse wird überwacht und falls nötig wird der berechnete Schedule angepaßt oder die verbleibenden Prozesse werden neu eingeplant. Ihr Ansatz ist flexibel und kann sporadische, aperiodische und periodische Prozesse behandeln, aber er verbraucht viel Zeit während der Anwendungsausführung für die Überwachung und Neuplanung. Insbesondere, wenn die in der Anwendung angegebenen Fristen knapp bemessen sind, kann dies zu Problemen führen. Eine weitere Schwierigkeit beim Einsatz ihrer Heuristik ist die Vielzahl der einzustellenden Parameter, die die Qualität stark beeinflussen. Um gute Resultate zu erzielen, ist es auch nötig, daß die Methoden während ihrer Ausführung Information über ihren Fortschritt bzw. ihre Restdauer geben können, damit bei zu knapper Zeit auf eine geeignete Notfallmethode gewechselt werden kann. Diese Eigenschaft besitzen nur wenige Implementierungen, und die bis zum Abbruch einer Methode verstrichene Zeit trägt nicht zu einer Verbesserung des Ergebnisses bei, weil Zwischenergeb-

[1]Anstelle der Implementierung eines Algorithmus, der bei längerer Rechendauer bessere Ergebnisse liefert, ist es bei Design-to-time-Scheduling möglich unterschiedliche Algorithmen für unterschiedliche Ausführungsdauern zu verwenden.

nisse in der Regel nicht vorliegen oder nicht aufgegriffen werden können. Nach [GL95] entwickeln Garvey und Lesser den Design-to-time-Ansatz zu Design-to-criteria [WGL97, WGL98] weiter und entwickeln Algorithmen für verteilte Multi-Agenten-Systeme [WL01]. Diese Arbeiten sind überwiegend auf komplexe, verteilte Systeme mit weichen Echtzeitbedingungen und stark variierenden Prozeßmengen ausgerichtet.

Feiler und Walker [FW01] haben einen Scheduling-Algorithmus für Systeme mit knappen Ressourcen entwickelt. Auch sie verwenden ungenutzte CPU-Zeit um die Qualität von Anwendungen zu verbessern. In ihrem Modell können Prozesse aus einem notwendigen und aus einem optionalen Teil, der die Qualität steigert, aber für das System nicht notwendig ist, bestehen. Die erste Version ihres Algorithmus garantiert jedem Prozeß genug Zeit, um seinen notwendigen Teil fertigzustellen, und auch alle eingeplanten optionalen Teile halten ihre Frist ein. Ihre zweite Version besitzt geringere Laufzeitkosten, garantiert aber nicht mehr die Einhaltung von Fristen für optionale Teile. Beide Versionen sind auf unterbrechbare Prozesse ausgelegt, und ihr Modell berücksichtigt keine Unterbrechungskosten. Darum sollte die Zielplattform Unterbrechungen und Kontextwechsel hardwareseitig unterstützen.

Binns [Bin97] stellt einen auf *slack stealing* basierenden Scheduling-Algorithmus für periodische inkrementelle und Design-to-time Prozesse ohne Datenabhängigkeiten vor. Ein statischer Algorithmus sichert jedem Prozeß genug Zeit zu, um ein zumindest ausreichendes Ergebnis zu garantieren. Während der Anwendungsausführung teilt eine *first-come first-granted* Strategie den verbleibenden Prozessen die von den Vorgängern nicht genutzte Zeit zu, um die Güte der Lösung zu steigern.

Charpillet und Boyer [CB97] stellen ein Modell und eine Heuristik für die Einplanung von Design-to-time Prozessen vor. In ihrem Modell bestehen Design-to-time Prozesse jedoch aus einem obligatorischen, einem optionalen und einem Aktions-Teil. Die optionalen Teile sind unterbrechbar und liefern auch bei einem Abbruch noch ein Ergebnis, d.h. die optionalen Teile müssen die Anytime-Eigenschaften erfüllen. Zunächst werden die Ausführungszeiten des obligatorischen Teils und des Aktions-Teils mit einer Adaption des *slack-time server*-Algorithmus [LRT92] garantiert. Anschließend teilt eine Heuristik die verbleibende Zeit den optionalen Teilen zu.

Rusu u.a. [RMM03] stellen Algorithmen für qualitätsmaximierende Prozeßeinplanung mit Fristen und Energiebeschränkungen vor. Ihr Algorithmus behandelt periodische Prozesse ohne Datenabhängigkeiten für die es mehrere Implementierungen gibt (Design-to-time Ansatz), die sich hinsichtlich ihrer Laufzeit, Ergebnisqualität und ihres Energiebedarfs unterscheiden. Sie führen damit die beiden Bereiche qualitäts- und energiebewußtes Scheduling zusammen, wobei ihr Algorithmus zwar versucht die Qualität zu maximieren, jedoch beim Energiever-

brauch nur darauf geachtet wird, eine vorgegebene Schranke nicht zu überschreiten.

2.1.2 Energiebewußtes Scheduling

Ebenso wie beim qualitätsbewußten Scheduling ist auch beim energiebewußten Scheduling eine Zielfunktion zu optimieren, wobei alle Nebenbedingungen eingehalten werden müssen. Die Möglichkeit einer Optimierung wird hierbei aber (meist) nicht durch spezielle Implementierungen der Prozesse geschaffen, sondern durch Besonderheiten des Prozessors. Eine solche Besonderheit ist die Fähigkeit moderner Prozessoren, während des Betriebs in unterschiedlich tiefe Ruhe- oder Schlafmodi zu gehen und in diesen weniger Leistung aufzunehmen. Eine andere Besonderheit ist die Möglichkeit zur dynamischen Anpassung der Taktfrequenz und/oder der Versorgungsspannung des Prozessors. Da sich die Leistungsaufnahme eines CMOS-Prozessors überproportional zur Änderung der Taktfrequenz verringert [BB95, CSB92, RJD98, Yea98], ist es bei ausreichendem Zeitrahmen möglich Energie zu sparen, indem Prozesse mit geringerer Taktfrequenz ausgeführt werden. Dieser Effekt wird in modernen Prozessoren genutzt. Es existieren sowohl einfache statische Strategien wie z.B. die Intel SpeedStep Technologie [Int03, Int04], die eine niedrige Versorgungsspannung während des Batteriebetriebs und eine hohe Spannung während des Netzbetriebs einstellen, als auch dynamische Verfahren, die die Versorgungsspannung dynamisch in Abhängigkeit von der Auslastung des Prozessors variieren [Fle01]. All diese Strategien sind für Echtzeitsysteme ungeeignet, da es aufgrund der längeren Ausführungszeiten der Prozesse in niedrigen Leistungsmodi zu Fristverletzungen kommen kann. Daher ist es nötig, die genannten Hardware-Strategien zu deaktivieren und dem Betriebssystem, dem Scheduler oder gar einem Prozeß die Kontrolle über den Leistungsmodus des Prozessors zu geben. Der Leistungsmodus wird, z.B. bei einem Intel Pentium M Prozessor, durch das setzen eines Prozessorregisters eingestellt. Dies kann z.B. mit einem Applet für Windows-Systeme (siehe [Int04]) bewerkstelligt werden.

Für die Anpassung der Versorgungsspannung durch den Scheduler gibt es zwei verschiedene Modelle. Ein verbreitetes Modell geht von der Annahme aus, daß die Spannung stufenlos verändert werden kann [GCW95, MAAM02, MC03, PS01, SC99, WWDS94, YDS95], während man beim anderen Modell nur eine bestimmte Anzahl diskreter Spannungsstufen zur Verfügung hat [KL00, KL03, LK03, LS00]. Natürlich können die Algorithmen für das erste Modell teilweise auf diskrete Leistungsstufen angepaßt werden (z.B. [SC99]) und die für das zweite Modell können auf Prozessoren mit kontinuierlicher Leistungseinstellung ausgeführt werden. Das erste Modell ermöglicht (zumindest teilweise) einen analytischen Lösungsansatz (z.B. [YDS95]), während das zweite Modell diskrete Ver-

fahren erzwingt (z.B. [LK03]). Beide Modelle können verfeinert werden, indem die eventuell nötige Zeit und Energie [Wol01] für einen Wechsel der Spannung betrachtet werden (z.B. [ZMC03]). Die in dieser Arbeit vorgestellten Algorithmen behandeln Prozessoren mit diskreten Spannungsstufen. Die nötige Umschaltenergie und -zeit können wahlweise vernachlässigt oder berücksichtigt werden.

Neben den genannten Arbeiten für Einprozessorsysteme existieren zahlreiche Erweiterungen für Mehrprozessorsysteme [LJ00, KP97, YWM^{+}01, YMW^{+}02, ZMC03]. Bei Mehrprozessorsystemen liegt der Hauptaugenmerk jedoch auf der Zuordnung der Prozesse zu den Prozessoren (Allokation) [KP97], da diese den durch die Prozeßeinplanung auf den Prozessoren erzielbaren Energieverbrauch entscheidend einschränkt. Nachdem die Allokation erfolgt ist, wird auf jedem Prozessor ein Einprozessor-Scheduler verwendet (siehe z.B. [RKKL03]). [UKK00b] gehen auf die unterschiedlichen Anforderungen bei lose bzw. eng gekoppelten Mehrprozessorsystemen ein und stellen eine auf Simulated Annealing basierende Heuristik für eng gekoppelte Systeme vor.

Unsal und Koren [UK03] geben einen Überblick über aktuelle Techniken für energie- und leistungsbewußtes Design von Echtzeitsystemen. Sie teilen die Arbeiten in die Kategorien *Compiler-*, *Betriebssystem-* und *Netzwerkebene* ein. Ziel der Compilertechniken (z.B. [LLM^{+}01, MSW01, LMD^{+}04, MWV^{+}04]) ist die Reduktion des Energieverbrauchs durch eine Optimierung der Hauptspeicherzugriffe oder eine Minimierung der Anzahl der auszuführenden Maschinenbefehle. Die hier vorliegende Arbeit liegt, wie die meisten der oben aufgeführten Arbeiten, in der Betriebssystemebene, in der es vorwiegend um die Entwicklung energiebewußter Scheduling-Algorithmen geht. In der Netzwerkebene werden beispielsweise neue Kommunikationsprotokolle [DHSS02], Datenreplikationstrategien [UKK00a] und Routing-Algorithmen [DWH03] zur Energieeinsparung entwickelt.

In ihrem Epilog schlagen Unsal und Koren auch explizit vor, daß zukünftige Arbeiten die Wahrscheinlichkeitsverteilungen der Ausführungszeiten - wie in dieser Arbeit - berücksichtigen sollten. Probleme, die sich bei energiebewußten Systemen im Hinblick auf den Hauptspeicher ergeben, werden in [LCNS99] behandelt.

2.2 Optimierung

Die in dieser Arbeit zu lösende Optimierungsaufgabe wurde, wie z.B. in [Foh94], als Suchproblem modelliert, das sowohl mit auf Dynamischer Programmierung als auch mit auf Simulated Annealing basierenden Algorithmen gelöst wird.

2.2.1 Dynamische Programmierung

Bellmann [Bel57] beschreibt ein Verfahren namens *Dynamische Programmierung* zur Optimierung mehrstufiger Entscheidungsprozesse. Das Verfahren wird bis heute in zahlreichen Varianten eingesetzt, die in die zwei Gruppen *wertiterativ* (engl. *value iteration*) und *strategieiterativ* (engl. *policy iteration*) eingeteilt werden können [SB98]. Während bei der ersten Gruppe eine Folge von Wertfunktionen erzeugt wird, die gegen die optimale Wertfunktion konvergiert, und aus der eine optimale Strategie erzeugt werden kann, erzeugen die Verfahren der zweiten Gruppe eine Folge von Strategien, die zu einer optimalen Strategie konvergiert deren Wertfunktion optimal ist.

Schon sehr früh wurden Variationen des Verfahrens entwickelt, die die beiden Hauptprobleme der Dynamischen Programmierung lösen sollten: exponentieller Speicherbedarf und Fehlen von Zwischenlösungen. Beispielsweise kombiniert das *Heuristic Search*-Verfahren von Martelli und Montanari [MM73, MM78] die Ansätze von Dynamischer Programmierung mit denen von *Branch and Bound*. Die von ihnen angegebene Komplexität im schlechtesten Fall ist höher als die der reinen Dynamischen Programmierung, aber im Normalfall vermeidet ihr Algorithmus die Erzeugung und Bewertung vieler Zustände. Die Größe der Einsparungen hängt jedoch stark von der Güte der für ihren Algorithmus benötigten Schätzfunktion ab. Auch diverse andere Abwandlungen der Dynamischen Programmierung zielen auf die Beschleunigung des Verfahrens und/oder die Erzeugung von Zwischenlösungen ab. Der Artikel von Barto und Bradtke [BBS95] enthält eine Übersicht der Varianten *Synchrone-*, *Asynchrone-*, *Gauss-Seidel-* und *Echtzeit Dynamische Programmierung*. Auch viele Algorithmen im Bereich der Künstlichen Intelligenz, wie etwa [BG03, HZ01, Kor90], beruhen auf dem Prinzip der Dynamischen Programmierung, und es existieren zahlreiche Erweiterungen der Dynamischen Programmierung, z.B. für nicht-deterministische Probleme [Ber87] oder Echtzeit-Problemstellungen [BBS95]. Bonet und Geffner [BG03] führen eine Markierungs-Hashing-Tabelle ein, um das Konvergenzverhalten des in [BBS95] vorgestellten Algorithmus (*Real-Time Dynamic Programming*) für nicht-deterministische Probleme zu verbessern.

Die Umsetzung der Dynamischen Programmierung erfolgt auch in der hier vorliegenden Arbeit nicht in der ursprünglichen tabellenbasierten Weise. Eine Modifikation führt dazu, daß schon während der Optimierung eine Folge besser werdender Lösungen erzeugt wird. Die zweite Änderung erlaubt es, den entstandenen Algorithmus mit Simulated Annealing zu kombinieren, um schneller gute Zwischenlösungen zu erhalten. Der Preis für diese Effizienzsteigerung ist jedoch, daß das Auffinden einer optimalen Lösung, aufgrund der zufälligen Suchreihenfolge, nicht mehr in endlicher Zeit garantiert werden kann.

2.2.2 Simulated Annealing

Kirkpatrick, Gellat und Vecchi [KGV83] stellen erstmals ein Verfahren zur Optimierung kombinatorischer Probleme vor, das das Abkühlen von Metallen simuliert (*Simulated Annealing*). Für die Anwendung des Verfahrens müssen eine exakte Beschreibung der Systemkonfiguration (Zustand), ein Generator für zufällige Schritte von einer Konfiguration zu einer anderen, eine Bewertungsfunktion für die Konfigurationsübergänge sowie ein Abkühlplan für den Optimierungsvorgang gegeben sein. Im Laufe der Optimierung erzeugt der Algorithmus zufällige Übergänge und bewertet diese. Übergänge zu besseren Konfigurationen werden dabei immer ausgeführt, während Übergänge zu schlechteren Konfigurationen nur mit einer von der aktuellen Temperatur und dem Wert der Verschlechterung abhängigen Wahrscheinlichkeit umgesetzt werden. Diese Wahrscheinlichkeit sinkt mit fallender Temperatur und steigender Verschlechterung, wodurch es dem Algorithmus anfangs möglich ist, aus lokalen Optima zu entkommen, er jedoch zum Ende des Abkühlvorgangs konvergiert.

Simulated Annealing wird in vielen Bereichen angewendet und es gibt auch zahlreiche Arbeiten die es für Scheduling-Probleme verwenden. DiNatale und Stankovic [NS95] verwenden Simulated Annealing für die Einplanung und Jitter-Kontrolle von Echtzeit-Prozessen auf Mehrprozessor-Systemen. Steinhöfel [Ste99] setzt Simulated Annealing zur Minimierung der Dauer von Job-Shop-Scheduling Problemen ein. [Cat98, Cra95] beschäftigt sich mit der Lösung von Scheduling-Problemen mit Ressourcen. [Cat98] stellt Untersuchungen zum Konvergenzverhalten von Simulated Annealing an und vergleicht die Ergebnisse für Simulated Annealing mit denen des Metropolis-Algorithmus.

2.3 Einordnung dieser Arbeit

Das im nächsten Kapitel vorgestellte Modell erweitert das um nichtdeterministische Ausführungsdauern angereicherte Design-to-time Modell um Prozessoren mit mehreren Leistungsstufen. Das Modell umfaßt periodische Prozesse mit Datenabhängigkeiten, Bereitzeiten und Fristen. Ihre Ausführung ist nicht unterbrechbar und kann durch die Verwendung alternativer Implementierungen erfolgen, die sich hinsichtlich ihrer Qualität bzw. ihres Energiebedarfs und ihrer Ausführungsdauer unterscheiden. Die Prozessoren können beliebig viele diskrete Leistungsstufen besitzen, und der Zeit- und Energieaufwand für einen Wechsel des Modus kann berücksichtigt werden.

Die Einplanung von Systemen, die mit diesem Modell spezifiziert werden können, erfolgt durch einen zweiphasigen Algorithmus. In der statischen Optimierungsphase wird mittels einer neuen Variante der Dynamischen Programmierung

oder Simulated-Annealing ein flexibler Plan erzeugt, der die Einhaltung der Echtzeitbedingungen garantiert und die gewählte Zielfunktion minimiert. Als Zielfunktion kann die Maximierung der durchschnittlich je Hyperperiode gelieferten Qualität oder die Minimierung der durchschnittlich während einer Hyperperiode verbrauchten Energie gewählt werden. Die modulare Aufteilung des Optimierers in drei Teile verleiht der Dynamischen Programmierung die Eigenschaften eines Anytime-Algorithmus, und sie erlaubt die Verwendung anderer qualitäts- bzw. energiebewußter Schedulingalgorithmen für die Suche nach einem optimalen, flexiblen Plan.

In der dynamischen Einplanungsphase wird der flexible Plan durch einen laufzeit- und energieeffizienten Scheduler interpretiert, der abhängig von der verstrichenen Zeit für jede Prozeßinstanz die optimale Methoden ausführt.

Kapitel 3

Modellierung

Dieses Kapitel führt das verwendete Anwendungs- und Hardwaremodell und die zu optimierenden Zielfunktionen ein. Anschließend wird das Modell der flexiblen Schedules vorgestellt.

Für Echtzeit-Scheduling müssen sowohl die auszuführende Anwendung als auch die Hardware, auf der die Anwendung laufen soll, modelliert werden. Das Modell der Anwendung enthält Informationen über ihre Struktur und über ihre zeitlichen Nebenbedingungen. Die Struktur ist durch die Prozesse, die zur Anwendung beitragen, deren Abhängigkeiten und die möglichen Implementierungen (*Methoden*) für die Prozesse gegeben. Ein Prozeß stellt eine abstrakte, periodische Aufgabe dar, die für die erfolgreiche Ausführung der Anwendung erfüllt werden muß. Er kann von anderen Prozessen, die die gleiche Periodendauer haben, datenabhängig sein. Eine einem Prozeß zugeordnete Methode ist die Implementierung eines Algorithmus, der die Aufgabe des Prozesses erfüllt. Zeitliche Nebenbedingungen sind die Bereitzeiten und die Fristen der Prozesse, sowie der Berechnungsaufwand der verschiedenen Implementierungen. Die Ausführungszeit einer Methode hängt von der Rechenleistung (Taktrate) des Prozessors ab, auf dem sie ausgeführt wird. Daher werden im Anwendungsmodell keine Methodenausführungszeiten, sondern Instruktionseinheiten angegeben, aus denen sich zusammen mit dem Leistungsmodus des Prozessors die Ausführungszeit berechnen läßt. Neben der Modellierung der Anwendung müssen auch noch die Prozessoren spezifiziert werden, auf denen die Anwendung ausgeführt werden soll. Das Modell umfaßt heterogene Multiprozessorsysteme, wobei für jeden der Prozessoren mehrere Taktraten, mit denen er betrieben werden kann, angegeben werden können. Die für die Abarbeitung einer Instruktionseinheit benötigte Energie hängt von der Taktrate ab (siehe Anhang C) und sie kann durch die geeignete Wahl der Taktrate gesenkt werden, ohne die Echtzeitbedingungen zu verletzen.

3.1 Anwendungs- und Hardwaremodell

Die in dieser Arbeit betrachteten Anwendungen können aus mehreren Prozessen, die periodisch ausgeführt werden müssen, bestehen. Ein Prozeß beschreibt eine Aufgabe die zu erledigen ist. Es können unterschiedliche Methoden, die die Aufgabe des Prozesses erfüllen, angegeben werden, und er kann von anderen Prozessen, die die gleiche Periodendauer haben, datenabhängig sein. Datenabhängigkeiten von Prozessen mit anderer Periodendauer sind nicht sinnvoll. Bei längerer Periodendauer des Vorgängerprozesses führen sie zu Fristverletzungen des Nachfolgeprozesses, und bei kürzerer dazu, daß die vom Vorgänger produzierten Daten mit immer größerer Verzögerung verarbeitet werden.

Die Hyperperiode ist das kleinste gemeinsame Vielfache aller im Modell enthaltenen Periodendauern. Sobald das Ende einer Hyperperiode erreicht ist, liegt wieder die gleiche Situation vor, wie zu ihrem Beginn. Für die Erzeugung eines optimalen Schedules reicht es also aus, alle Instanzen innerhalb einer Hyperperiode optimal zu schedulen. Daher werden alle Prozesse während der Vorverarbeitung innerhalb einer Hyperperiode instanziert und eingeplant. Um die Größe des Suchraumes zu reduzieren, wird aus den spezifizierten Datenabhängigkeiten, Bereitzeiten und Fristen eine Präzedenzrelation zwischen den Prozeßinstanzen definiert.

3.1.1 Prozessoren

Das Prozessormodell umfaßt die zur Verfügung stehenden Prozessoren und ihre möglichen Leistungsstufen. Für jede Leistungsstufe müssen die Anzahl der Instruktionseinheiten, die der Prozessor in einer Millisekunde bearbeitet, und der Energieverbrauch pro Millisekunde bei Berechnungen sowie im Leerlauf angegeben werden. Optional können die beim Übergang von einer Leistungsstufe zu einer anderen benötigten Zeiten und Energien für jeden möglichen Übergang angegeben werden.

Definition 1 ((Heterogenes) Mehrprozessorsystem, Prozessor). *Eine Menge* $Z = \{Z_0, \ldots, Z_n\}$ *von Prozessoren heißt* (heterogenes) Mehrprozessorsystem. *Ein Vektor* $Z_i = (N, L_{i,0}, \ldots, L_{i,m_i})$ *von Leistungsmodi* $L_{i,j}$ *heißt* Prozessor*, wobei* N *der Prozessortyp und* m_i *die Anzahl der Leistungsmodi* $L_{i,j}$ *des Prozessortyps* Z_i *ist.*

Definition 2 (Leistungsmodus). *Ein Vektor* $L_{i,j} = (r(i,j), e_b(i,j), e_s(i,j), (e_w(i,j,0), \ldots, e_w(i,j,m_i)), (t_w(i,j,0), \ldots, t_w(i,j,m_i)))$ *heißt* j-ter Leistungsmodus des Prozessors Z_i*. Er spezifiziert die Anzahl* $r(i,j)$ *der pro Millisekunde bearbeiteten Instruktionseinheiten (Rechenleistung), den Energieverbrauch je Millisekunde* $e_b(i,j)$ *bei der Bearbeitung eines Prozesses und im Schlafmodus*

$e_s(i, j)$*, sowie die Energien pro Millisekunde* $e_w(i, j, k)$ *und Zeiten* $t_w(i, j, k)$*, die für den Übergang von Modus* j *zu Modus* k*, mit* $0 \leq k \leq m_i$ *benötigt werden.*

Definition 3 (Instruktionseinheit). *Eine* Instruktionseinheit *(IU) ist die Anzahl von Instruktionen, die von einem Prozessor mit 1 MHz Taktrate in einer Millisekunde durchschnittlich bearbeitet wird.*

Das Einprozessorsystem aus Abbildung 1.3 entspricht der folgenden formalen Spezifikation:

$$\begin{aligned} Z &= \{Z_0\} \\ Z_0 &= (\text{CPU}, L_{0,0}, L_{0,1}) \\ L_{0,0} &= (1.0, 4.0, 0.4, (0,0), (0,0)) \\ L_{0,1} &= (0.5, 1.0, 0.1, (0,0), (0,0)) \end{aligned}$$

Es ist ein Prozessor vom Typ CPU verfügbar, der zwei Leistungsmodi besitzt. Der erste Modus arbeitet im Mittel eine Instruktionseinheit pro Millisekunde ab und verbraucht dabei vier Energieeinheiten bzw. eine Energieeinheit in einer Millisekunde Leerlauf. Der zweite Modus arbeitet im Mittel eine halbe Instruktionseinheit pro Millisekunde ab und verbraucht dabei eine Energieeinheit bzw. eine zehntel Energieeinheit in einer Millisekunde Leerlauf. Die Energien und Zeiten für die möglichen Moduswechsel werden im Beispiel vernachlässigt und sind daher jeweils mit 0 angegeben.

3.1.2 Methoden

Das Methodenmodell umfaßt die Spezifikation aller im Anwendungsmodell zur Verfügung stehenden Methoden für die Ausführung der zu erledigenden Aufgaben. Für jede Methode werden ihre diskrete Wahrscheinlichkeitsverteilung der benötigten Instruktionseinheiten, die von ihr zu erwartende Qualität und der kompatible Prozessortyp angegeben.

Definition 4 (Methode). *Ein Vektor* $M_{i,j} = (N, A_{i,j}, q_{i,j}, K_{i,j})$ *heißt* Methode. N *ist der Name der Methode. Der Vektor* $A_{i,j} = ((p_{i,j,0}, a_{i,j,0}), \ldots, (p_{i,j,k}, a_{i,j,k}))$ *gibt die Wahrscheinlichkeit* $p_{i,j,m}$ *an, mit der* x *Instruktionseinheiten, mit* $a_{i,j,m-1} < x \leq a_{i,j,m}$ *und* $1 \leq m \leq k$*, benötigt werden.* $p_{i,j,0}$ *ist die Wahrscheinlichkeit, daß zwischen* 0 *und* $a_{i,j,0}$ *Instruktionseinheiten benötigt werden. Der Eintrag* $q_{i,j}$ *gibt die zu erwartende Qualität der Methode an und* $K_{i,j}$ *ist der Prozessortyp auf dem die Methode ausgeführt werden kann.*

Die formale Spezifikation der Methoden aus Abbildung 1.3 sieht daher folgendermaßen aus:

$$
\begin{aligned}
M_{0,0} &= (\text{bilinear}, ((1.0, 2)), 10, \{Z_0\}) \\
M_{1,0} &= (\text{Zeilen einfügen}, ((0.7, 1), (0.2, 4), (0.1, 7)), 8, \{Z_0\}) \\
M_{2,0} &= (\text{JPG-1}, ((0.9, 6), (0.1, 12)), 11, \{Z_0\}) \\
M_{2,1} &= (\text{JPG-2}, ((0.2, 6), (0.8, 9)), 12, \{Z_0\}) \\
M_{3,0} &= (\text{Sendertreiber}, ((0.6, 4), (0.4, 5)), 7, \{Z_0\})
\end{aligned}
$$

3.1.3 Anwendung und Prozesse

Das Anwendungsmodell enthält alle für die Anwendung auszuführenden Aufgaben sowie deren zeitliche Nebenbedingungen und Abhängigkeiten voneinander. Im Prozeßmodell wird für jeden Prozeß die Menge der Methoden, die für die durch ihn zu erfüllende Aufgabe verwendet werden können, angegeben.

Definition 5 (Anwendung). *Ein Vektor* $Anw = (N, (P_0, p_0, b_0, f_0), \ldots, (P_n, p_n, b_n, f_n))$ *von Prozessen heißt* Anwendung. N *ist der Name der Anwendung. Die Tupel* (P_i, p_i, b_i, f_i)*,* $0 \leq i \leq n$*, spezifizieren die für die Anwendung nötigen Prozesse. Dabei ist* p_i *die Periodendauer,* b_i *die Bereitzeit relativ zum Periodenbeginn und* f_i *die Frist relativ zur Bereitzeit des Prozesses* P_i*.*

Definition 6 (Prozeß). *Ein Tripel* $P_i = (N, M_i, D_i)$ *heißt* Prozeß. N *ist der Name des Prozesses. Die Menge* $M_i = \{M_{i,j|0 \leq j \leq n_i}\}$ *enthält alle Methoden, die für die Ausführung der durch* P_i *spezifizierten Aufgabe geeignet sind, und die Menge* $D_i = \{D_{i,j} | 0 \leq j \leq m_i\}$ *enthält alle Prozesse, die vor der Ausführung von Prozeß* P_i *abgearbeitet sein müssen.*

Damit ist die formale Spezifikation der in Abbildung 1.3 gezeigten Anwendung und der darin enthaltenen Prozesse:

$$
\begin{aligned}
Anw &= (\text{Live-Video}, ((P_0, 40ms, 0ms, 40ms), (P_1, 40ms, 0ms, 40ms), \\
&\quad (P_2, 40ms, 0ms, 40ms), (P_3, 40ms, 0ms, 40ms))) \\
P_0 &= (\text{Skalieren}, \{M_{0,0}\}, \{\}) \\
P_1 &= (\text{Text einblenden}, \{M_{1,0}\}, \{P_0\}) \\
P_2 &= (\text{Kodieren}, \{M_{2,0}, M_{2,1}\}, \{P_1\}) \\
P_3 &= (\text{Senden}, \{M_{3,0}\}, \{P_2\})
\end{aligned}
$$

3.1.4 Instanzierung

Zu jedem Prozeß P_i mit Periodendauer p_i werden die ersten $\frac{p}{p_i}$ Instanzen kreiert, die innerhalb der Hyperperiode p eingeplant werden müssen. Da die Situation für die Einplanung zu Beginn jeder neuen Hyperperiode identisch ist, kann ein für die erste Hyperperiode gefundener Schedule einfach wiederholt werden, um alle Instanzen der ursprünglichen Prozesse einzuplanen.

Für jeden Prozeß i wird die Menge seiner Instanzen $P_{i,j}$ erzeugt und die Elemente werden initialisiert:

$$P_i = \{P_{i,0}, \ldots, P_{i,\frac{p}{p_i}-1}\}$$

Die Bereitzeit einer Instanz j, $0 \leq j \leq \frac{p}{p_i} - 1$, ist der Beginn ihrer Periode plus der Bereitzeit des Prozesses:

$$b_{i,j} = j \cdot p_i + b_i$$

Die Frist einer Instanz j, $0 \leq j \leq \frac{p}{p_i} - 1$, ist ihre Bereitzeit plus die Frist ihres Prozesses, beschränkt auf das Periodenende der Instanz:

$$f_{i,j} = \min\{b_{i,j} + f_i, (j+1) \cdot p_i\}$$

Die minimale Ausführungszeit einer Instanz ist die minimale Ausführungszeit der Methoden des zugehörigen Prozesses[1]

$$\text{minET}(P_{i,j}) = \min\left\{ t \,\middle|\, (p,a) \in A_{i,k} \wedge M_{i,k} \in M_i \wedge t = \frac{a}{r_{max}(M_{i,k})} \right\}$$

Die Ausführungsdauer einer Methode $M_{i,k}$ im schlimmsten Fall mit der höchsten Leistungsstufe ist

$$WCET(M_{i,k}) := max\left\{ \frac{a}{r_{max}(M_{i,k})} \,\middle|\, (p,a) \in A_{i,j} \right\}$$

Die minimale worst-case Ausführungszeit einer Instanz ist die minimale worst-case Ausführungszeit ihrer Methoden:

$$\text{minWCET}(P_{i,j}) = \min\{WCET(M_{i,k}) | M_{i,k} \in M_i\}$$

Die maximale worst-case Ausführungszeit einer Instanz ist die maximale worst-case Ausführungszeit ihrer Methoden:

$$\text{maxWCET}(P_{i,j}) = \max\{WCET(M_{i,k}) | M_{i,k} \in M_i\}$$

[1] $r_{max}(M_{i,k})$ ist die Rechenleistung der höchsten Leistungsstufe des Prozessors, der mit der Methode $M_{i,k}$ kompatibel ist. $r_{min}(M_{i,k})$ ist die Rechenleistung der niedrigsten Leistungsstufe des Prozessors, der mit der Methode $M_{i,k}$ kompatibel ist.

Die maximale Qualität einer Instanz ist die maximale Qualität ihrer Methoden:

$$\text{maxQ}(P_{i,j}) = \max\{q_{i,k} | M_{i,k} \in M_i\}$$

Die Datenabhängigkeiten der Prozesse werden auf ihre Instanzen übertragen:

$$P_{i_1} \in D_{i_2} \Rightarrow P_{i_1,j} \in D_{i_2,j}$$

Im weiteren wird die Doppelindizierung der Prozeßinstanzen durch eine Einfachindizierung ersetzt, da ihre Zuordnung zu den Prozessen nicht mehr nötig ist. Um Verwechslungen zu vermeiden, werden die einfach-indizierten Instanzen mit I bezeichnet. Alle Eigenschaften wie z.B. Bereitzeit, Frist und Datenabhängigkeiten der doppelt indizierten Instanzen werden übertragen.

$$I = \{I_0, \ldots, I_{m-1}\} = \bigcup_{i=0}^{n} \bigcup_{j=0}^{\frac{p}{p_i}-1} \{P_{i,j}\}$$

3.1.5 Berechnung der effektiven Bereitzeiten und Fristen

Nach der Erzeugung der Instanzen, die in der ersten Hyperperiode liegen, werden für diese sogenannte *effektive Bereitzeiten* und *effektive Fristen* berechnet, um den Suchraum zu verkleinern. Die effektiven Bereitzeiten geben an, wann eine Instanz unter Berücksichtigung des Rechenbedarfs und der Bereitzeiten ihrer Vorgänger frühestens starten kann. Die effektiven Fristen geben an, wann eine Instanz unter Berücksichtigung des Rechenbedarfs und der Fristen ihrer Nachfolger spätestens beendet sein muß, damit alle Nachfolger eingeplant werden können. Die Berechnung der effektiven Bereitzeiten beginnt mit den Instanzen ohne Vorgänger und die Berechnung der Fristen beginnt mit den Instanzen ohne Nachfolger. Danach werden die Zeiten sukzessive für deren Nachfolger bzw. Vorgänger berechnet.

Berechnung der effektiven Bereitzeiten $\hat{b}_i$ und der effektiven Fristen $\hat{f}_i$:

$$\begin{aligned} \hat{b}_i &= \max(\{b_i\} \cup \{\hat{b}_j + minET(I_j) | I_j \in D_i\}) \\ \hat{f}_i &= \min(\{f_i\} \cup \{\hat{f}_j - minWCET(I_j) | I_i \in D_j\}) \end{aligned}$$

3.1.6 Bestimmung der Präzedenzrelation

Die Datenabhängigkeiten, die Reihenfolge der Instanzen und die zeitlichen Beziehungen zwischen Instanzen verschiedener Prozesse werden zu einer Präzedenzre-

lation $<_I$ zusammengefaßt:

$$\begin{aligned} <_I &: I \times I \rightarrow \{0,1\} \\ &(I_i, I_j) \mapsto \max\{<_{I_1} (I_i, I_j), <_{I_2} (I_i, I_j), <_{I_3} (I_i, I_j)\} \end{aligned}$$

Dabei ist $<_{I_1}$ die Präzedenz durch Datenabhängigkeiten

$$<_{I_1} (I_i, I_j) = \begin{cases} 1 & , \text{falls } I_i \in D_j \\ 0 & , \text{sonst} \end{cases}$$

$<_{I_2}$ die Präzedenz aufgrund der Instanzreihenfolge

$$<_{I_2} (I_i, I_j) = \begin{cases} 1 & , \text{falls } I_i = P_{l,p} \wedge I_j = P_{l,q} \wedge p < q \\ 0 & , \text{sonst,} \end{cases}$$

und $<_{I_3}$ die durch Fristen und Bereitzeiten gegebene Präzedenz

$$<_{I_3} (I_i, I_j) = \begin{cases} 1 & , \text{falls } \hat{f}_i \leq \hat{b}_j \\ 0 & , \text{sonst.} \end{cases}$$

Damit eine Instanz ausgeführt werden kann, muß ihre Bereitzeit erreicht sein und es müssen alle bezüglich $<_I$ kleineren Instanzen bereits erfolgreich beendet sein. Die Präzedenzrelation $<_I$ verringert die Anzahl der zu untersuchenden Schedules und beschleunigt dadurch sie Suche nach einem optimalen Schedule.

Für die Anwendung aus Abbildung 1.3 muß aufgrund der identischen Periodendauern nur jeweils eine Instanz je Prozeß kreiert werden. Das Model sieht nach der Instanzierung und Berechnung der effektiven Bereitzeiten und Fristen folgendermaßen aus:

3.2 Formales Modell der Beispielanwendung

Zusammengefaßt sieht das formale, instanzierte Modell der Beispielanwendung aus Abbildung 1.3 wie folgt aus:

$$\begin{aligned}
Anw &= (\text{Live-Video}, ((I_0, ---, 0ms, 20ms), (I_1, ---, 2ms, 26ms),\\
&\quad (I_2, ---, 3ms, 35ms), (I_3, ---, 9ms, 40ms)))\\
I_0 &= (\text{Skalieren}, \{M_{0,0}\}, \{\})\\
I_1 &= (\text{Text einblenden}, \{M_{1,0}\}, \{I_0\})\\
I_2 &= (\text{Kodieren}, \{M_{2,0}, M_{2,1}\}, \{I_1\})\\
I_3 &= (\text{Senden}, \{M_{3,0}\}, \{I_2\})
\end{aligned}$$

$$\begin{aligned}
M_{0,0} &= (\text{bilinear}, ((1.0, 2)), 10, \{Z_0\})\\
M_{1,0} &= (\text{Zeilen einfügen}, ((0.7, 1), (0.2, 4), (0.1, 7)), 8, \{Z_0\})\\
M_{2,0} &= (\text{JPG-1}, ((0.9, 6), (0.1, 12)), 11, \{Z_0\})\\
M_{2,1} &= (\text{JPG-2}, ((0.2, 6), (0.8, 9)), 12, \{Z_0\})\\
M_{3,0} &= (\text{Sendertreiber}, ((0.6, 4), (0.4, 5)), 7, \{Z_0\})
\end{aligned}$$

$$\begin{aligned}
Z &= \{Z_0\}\\
Z_0 &= (\text{CPU}, L_{0,0}, L_{0,1})\\
L_{0,0} &= (1, 4, 0.4, (0, 0), (0, 0))\\
L_{0,1} &= (0.5, 1, 0.1, (0, 0), (0, 0))
\end{aligned}$$

3.3 Schedule

Ein Schedule gibt für jede Prozeßinstanz an, wann sie mit welcher Methode auf welchem Prozessor mit welchem Leistungsmodus gestartet werden soll. In einem zulässigen Schedule ist die Startzeit einer Prozeßinstanz immer größer gleich ihrer effektiven Bereitzeit, die ausgeführte Methode ist vor der effektiven Frist beendet und mit dem gewählten Prozessor kompatibel. Alle bezüglich $<_I$ kleineren Instanzen müssen bereits beendet sein und es muß ausreichend Leerlaufzeit für die eingeplanten Leistungsmoduswechsel zur Verfügung stehen.

Definition 7 (Schedule, zulässiger Schedule). *Eine Menge* $S = \{(s_i, m_i, z_i, v_i) \in \mathbb{N} \times M \times Z \times V | i \in \{0 \dots m-1\}\}$ *heißt* Schedule. *Jedes Tupel* (s_i, m_i, z_i, v_i) *spezifiziert, daß Instanz* I_i *zum Zeitpunkt* s_i *mit Methode* m_i *und Leistungsmodus*

v_i auf Prozessor z_i gestartet und bis zu ihrer Beendigung ausgeführt wird. Ein Schedule heißt zulässiger Schedule*, wenn für alle $i \in \{0 \ldots m-1\}$ gilt:*

- $\hat{b}_i \leq s_i$ *(effektive Bereitzeit erreicht)*
- $s_i + \frac{1}{r(v_i)} \cdot WCET(m_i) < \hat{f}_i$ *(effektive Frist eingehalten)*
- $z_i = K(m_i)$ *(geeigneter Prozessor)*
- $\forall k \in \{0, \ldots, m-1\} : I_k <_I I_i \Rightarrow s_k + \frac{1}{r(v_k)} \cdot ET(m_k) < s_i$ *(Präzedenzen erfüllt)*
- $\forall t \in \mathbb{N}_0 : \forall z \in Z : |\{k | z_k = z \wedge \boldsymbol{1}_{z_k}(m_k, t) = 1\}| \leq 1$ *(maximal eine Methode gleichzeitig auf einem Prozessor)*[2]
- v_i *ist ein Leistungsmodus von* z_i *(zulässiger Modus)*
- $\forall t_1, t_2 \in \mathbb{N}, z_h \in Z : (t_1 < t_2 \wedge v_{z_h}(t_1) = L_{h,j} \wedge L_{h,j} \neq L_{h,k} \wedge L_{h,k} = v_{z_h}(t_2)) \Rightarrow |\{t \in]t_1, \ldots, t_2[| \forall m_i \in M : \boldsymbol{1}_{z_h}(m_i, t) = 0\}| \geq t_w(h, j, k)$ *(ausreichende Leerlaufzeit für Leistungsmoduswechsel)*

Für die Instanzen innerhalb der ersten Hyperperiode der Anwendung aus Abbildung 1.3 ist beispielsweise folgender Schedule zulässig:

$$\begin{aligned} S_{\text{Live-Video}} = \{ \quad & (0, M_{0,0}, Z_0, L_{0,0}), (10, M_{1,0}, Z_0, L_{0,0}), \\ & (20, M_{2,1}, Z_0, L_{0,0}), (30, M_{3,0}, Z_0, L_{0,0}) \ \} \end{aligned}$$

Die wiederholte Anwendung dieses Schedules $S_{\text{Live-Video}}$ liefert den folgenden Schedule $S^{\infty}_{\text{Live-Video}}$ für die unendliche Ausführung der Anwendung:

$$\begin{aligned} S^{\infty}_{\text{Live-Video}} = \{ \quad & (s_{i,j}, m_{i,j}, z_{i,j}, v_{i,j}) | i, j \in \mathbb{N}, s_{i,j} = 40 \cdot j + 10 \cdot i, \\ & m_{0,j} = M_{0,0}, m_{1,j} = M_{1,0}, m_{2,j} = M_{2,1}, m_{3,j} = M_{3,0}, \\ & z_{i,j} = Z_0, v_{i,j} = L_{0,0} \ \} \end{aligned}$$

3.4 Zielfunktionen

Eine Einplanung (zulässiger Schedule) muß bei den in dieser Arbeit betrachteten Anwendungen nicht nur gültig sein, sondern zusätzlich eine gegebene Zielfunktion optimieren. Diese kann sowohl eindimensional als auch mehrdimensional (*Mehrzieloptimierung*) sein. Es ist also ein Schedule S aus der Menge der zulässigen Schedules $\mathbb{S}$ zu finden, für den die Zielfunktion optimal ist.

[2] $\mathbf{1}_{z_k}(m_i, t)$: Methode m_i wird zum Zeitpunkt t auf Prozessor z_k ausgeführt (Wert 1) oder nicht (Wert 0).

Energiebewußtes Scheduling und *qualitätsbewußtes Scheduling* sind Beispiele für eindimensionale Zielfunktionen. Beim energiebewußten Scheduling ist die Energie, die im Durchschnitt pro Zeiteinheit verbraucht wird, zu minimieren. Diese durchschnittliche Energie errechnet sich aus der Summe der Energien, die vom Prozessor für die Bearbeitung der Methoden, während auftretender Leerlaufzeit und für Versorgungsspannungsanpassungen verbraucht werden. Auch im Beispiel aus Abbildung 1.3 ist diese Zielfunktion zu optimieren. Beim qualitätsbewußten Scheduling soll die Qualität maximiert werden, die von den in der Anwendung enthaltenen und ausgeführten Methoden im Durchschnitt pro Zeiteinheit geliefert wird. Es reicht dazu aus, den Schedule für die Prozeßinstanzen innerhalb einer Hyperperiode so zu optimieren, daß der Erwartungswert der Zielfunktion innerhalb der Hyperperiode optimal ist, weil es keine zulässigen Schedules gibt, bei denen eine Prozeßinstanz in zwei Hyperperioden ausgeführt wird.

Bei einer Mehrzieloptimierung sind mehrere eindimensionale Zielfunktionen vorgegeben, deren optimale Einplanungen sich widersprechen können. Ein Beispiel hierfür ist die gemeinsame Optimierung der beiden Zielfunktionen für energiebewußtes und qualitätsbewußtes Scheduling. Für einen minimalen Energieverbrauch muß der Prozessor solange wie möglich mit der niedrigsten Taktrate betrieben werden. Dies bedeutet aber, daß nur die schnellsten Methoden, die in der Regel die schlechteste Ergebnisqualität besitzen, eingeplant werden können, um keine Fristen zu verletzen. Aus diesem Grund wird einer der Zielfunktionen eine höhere Priorität zugewiesen als der anderen. Erhält beispielsweise die Qualitätsmaximierung die höhere Priorität, so werden Lösungen - unabhängig von ihrem Energieverbrauch - bevorzugt, die eine höhere Qualität liefern als eine andere. Bei Lösungen mit gleicher Qualität wird jedoch diejenige mit dem geringeren Energieverbrauch bevorzugt. Zusätzlich kann für die Qualität eine Schranke angegeben werden, die nicht unterschritten werden darf, und für die Energie eine Schranke, die nicht überschritten werden darf. Bei einer Mehrzieloptimierung mit mehr als zwei Zielen erhält jedes Ziel eine eindeutige Priorität. Die dadurch gegebene lexikographische Ordnung bildet eine Totalordnung auf der Wertemenge der Mehrzielfunktion, und ermöglicht die Anwendung der in dieser Arbeit beschriebenen Optimierungsverfahren.

3.4.1 Energiebewußtes Scheduling

Die für einen Schedule S zu minimierende Zielfunktion beim energiebewußten Scheduling $E : \mathbb{S} \to \mathbb{R}$ gibt im zeitkontinuierlichen Modell den durchschnittlichen Leistungsverbrauch bzw. im zeitdiskreten Modell den durchschnittlichen

Energieverbrauch je Zeiteinheit an.[3]

$$E(S) = \lim_{n\to\infty} \frac{\int_0^{n\cdot p} p(t)dt}{n \cdot p} \text{ bzw. } E(S) = \lim_{n\to\infty} \frac{\sum_{t=1}^{n\cdot p} e(t)}{n \cdot p}$$

Dabei ist $p(t)$ die zum Zeitpunkt t aufgenommene Leistung und $e(t)$die Summe der Energien, die von den Prozessoren im Zeitschritt t verbraucht werden:

$$e(t) = \sum_{Z_i \in Z} \Big(\; rechnend(Z_i,t) \cdot e_b(i, v(i,t)) + leerlauf(Z_i,t) \cdot e_l(i, v(i,t)) + wechsel(Z_i, j, k) \cdot e_w(i,j,k) \; \Big)$$

Der Wert der energiebewußten Zielfunktion für den oben angegebenen Schedule der Anwendung aus Abbildung 1.3 ist:

$$\begin{aligned} E(S_{Live-Video}) &= \lim_{n\to\infty} \frac{n \cdot [\overbrace{(2 + 2.2 + 6.8 + 4.4)}^{\text{erwartete Laufzeiten}} \cdot e_b(0,0) + 4.6 \cdot e_l(0,0)]}{n \cdot 40} \\ &= \lim_{n\to\infty} \frac{15.4 \cdot 4 + 4.6 \cdot 0.4}{40} \\ &= 1.586 \end{aligned}$$

3.4.2 Qualitätsbewußtes Scheduling

Die für einen Schedule S zu maximierende Zielfunktion beim qualitätsbewußtem Scheduling $Q(S) : \mathbb{S} \to \mathbb{R}$ gibt die durchschnittlich erzielte Qualität (je Zeiteinheit) an. Auch hier ist nur die Betrachtung vollständig abgeschlossener Hyperperioden p sinnvoll.

$$Q(S) = \lim_{n\to\infty} \frac{\int_0^{n\cdot p} \hat{q}(t)dt}{n \cdot p} \text{ bzw. } Q(S) = \lim_{n\to\infty} \frac{\sum_{t=1}^{t_0} q(t)}{n \cdot p}$$

Dabei ist $\hat{q}(t)$ die Summe der von den Methoden, die im Zeitpunkt t beendet werden, gelieferten Qualitäten:

$$\hat{q}(t) = \sum_{\substack{(s_{i,j}, m_{i,j}, z_{i,j}, v_{i,j}) \in S \\ \text{mit } s_{i,j} + ET(m_{i,j}) = t}} p(ET(m_{i,j})) \cdot q(m_{i,j})$$

[3]Für die Zielfunktion werden jeweils nur vollständig abgeschlossene Hyperperioden p berücksichtigt, da nur nach diesen Zeiträumen von der Anwendung ein sinnvolles Ergebnis geliefert wird.

$q(t)$ ist die Summe der von den Methoden, die innerhalb des Intervalls $]t-1,t]$, also im Zeitschritt t, beendet werden, gelieferten Qualitäten:[4]

$$q(t) = \int_{x\in]t-1,t]} \hat{q}(x)$$

Der Wert der qualitätsbewußten Zielfunktion für den oben angegebenen Schedule der Anwendung aus Abbildung 1.3 ist:

$$\begin{aligned} Q(S_{Live-Video}) &= \lim_{n\to\infty} \frac{n \cdot [10+8+12+7]}{n \cdot 40} \\ &= \lim_{n\to\infty} \frac{37}{40} \\ &= 0.925 \end{aligned}$$

3.4.3 Mehrzieloptimierung

Bei einer Mehrzieloptimierung ist die Zielfunktion $F(S) : \mathbb{S} \to \mathbb{R}^{n+1}$ durch die eindimensionalen Zielfunktionen $F_i(S)$, einzuhaltende Schranken $s_{M,i}$ und die Angabe $o_{M,i}$, ob maximiert oder minimiert werden soll, gegeben. Im Falle von $o_{M,i} = min$ ist $s_{M,i}$ eine einzuhaltende obere Schranke und im Falle von $o_{M,i} = max$ ist $s_{M,i}$ eine einzuhaltende untere Schranke. Die Priorität ist durch die Reihenfolge der eindimensionalen Zielfunktionen gegeben.

$$\begin{aligned} F_M(S) &= (F_0(S), \ldots, F_n(S)) \\ s_M(S) &= (s_{M,0}, \ldots, s_{M,n}) \in \mathbb{R}^{n+1} \\ o_M(S) &= (o_{M,0}, \ldots, o_{M,n}) \in \{<,>\}^{n+1} \end{aligned}$$

Die Totalordnung (lexikographische Ordnung) $>_{F_M}$ auf der Wertemenge $\mathbb{R}^{n+1}$ der zu maximierenden Mehrzielfunktion $F_M(S)$ ist dann gegeben durch:

$$\begin{aligned} \forall v, w \in \mathbb{R}^{n+1} : v >_{F_M} w :\Leftrightarrow \quad & \exists i \in \{0, \ldots, n+1\} : (v_i \; o_{M,i} \; w_i) \\ & \wedge \forall j \in \{0, \ldots, i-1\} : v_j = w_j \end{aligned}$$

3.4.4 Beschränkung auf pareto-optimale Methoden

Um unnötigen Optimierungsaufwand zu vermeiden, werden nur pareto-optimale Methoden für die Ausführung in Betracht gezogen. Pareto-optimal bedeutet, daß

[4] Zum Zeitpunkt $t = 0$ wird keine Qualität geliefert, weil zu diesem Zeitpunkt keine Methode fertiggestellt sein kann.

die Methode bezüglich einer gegebenen Halbordnung größer ist als alle mit ihr bezüglich der Halbordnung vergleichbaren Methoden. Alle Methoden die nicht pareto-optimal sind, werden vor der Optimierung aus dem Modell entfernt.

Im Falle der Qualitätsmaximierung ist eine Methode pareto-optimal, die garantiert eine bessere Qualität liefert als alle anderen Methoden, wobei auch die Qualität der Methoden, die in der verbleibenden Zeit eingeplant werden können, berücksichtigt werden muß. Falls immer nur die worst-case Laufzeiten der Methoden betrachtet werden, so sind Methoden pareto-optimal, für die es keine vergleichbaren Alternativen gibt, oder die eine höhere Qualität und zugleich eine geringere worst-case Laufzeit haben als vergleichbare Methoden. Eine pareto-optimale Methode liefert also selbst eine höhere Qualität und läßt aber zugleich den Nachfolgern mehr oder gleich viel Zeit, und daher können diese mindestens die gleiche Qualität liefern.

Im Falle der Energieminimierung ist eine Methode pareto-optimal, deren erwarteter Energieverbrauch niedriger ist als der aller anderen Methoden, wobei auch der erwartete Energieverbrauch der nachfolgenden Methoden berücksichtigt werden muß. Falls immer nur die worst-case Laufzeiten der Methoden betrachtet werden, so sind Methoden pareto-optimal, für die es keine vergleichbaren Alternativen gibt, oder die einen niedrigeren erwarteten Energieverbrauch und zugleich eine geringere worst-case Laufzeit haben. Eine pareto-optimale Methode verbraucht also selbst weniger Energie und läßt aber zugleich den Nachfolgern mehr oder gleich viel Zeit, um diese langsamer und daher mit weniger Energie laufen zu lassen.

Definition 8 (Halbordnung auf Verteilungsfunktionen). *Eine Verteilungsfunktion F_1 ist* größer *als eine Verteilungsfunktion F_2, wenn F_1 in mindestens einem Punkt größer und ansonsten punktweise mindestens gleichgroß ist wie F_2:*

$$F_1 > F_2 :\Leftrightarrow (\exists x \in \mathbb{R}_0^+ : F_1(x) > F_2(x)) \wedge (\forall x \in \mathbb{R}_0^+ : F_1(x) \geq F_2(x))$$

Die Betrachtung der worst-case Zeiten reicht für Methoden mit nicht-deterministischen Laufzeiten nicht aus. Für diesen Fall ersetzt die Forderung nach einer größeren Verteilungsfunktion der Laufzeiten die Forderung nach einer kleineren worst-case Zeit.

Falls beide Verteilungen diskret sind, reicht es aus die beiden Verteilungen an allen Stützstellen zu vergleichen. Abbildung 3.1 zeigt zwei vergleichbare Verteilungsfunktionen A und B sowie zwei unvergleichbare Verteilungsfunktionen A und C.

Definition 9 (Halbordnung auf diskreten Dichtefunktionen). *Eine diskrete Dichtefunktion $A_{i,j}$ ist* größer *als eine diskrete Dichtefunktion $A_{k,l}$, wenn $A_{i,j}$ in mindestens einem Punkt größer und ansonsten punktweise mindestens gleichgroß ist*

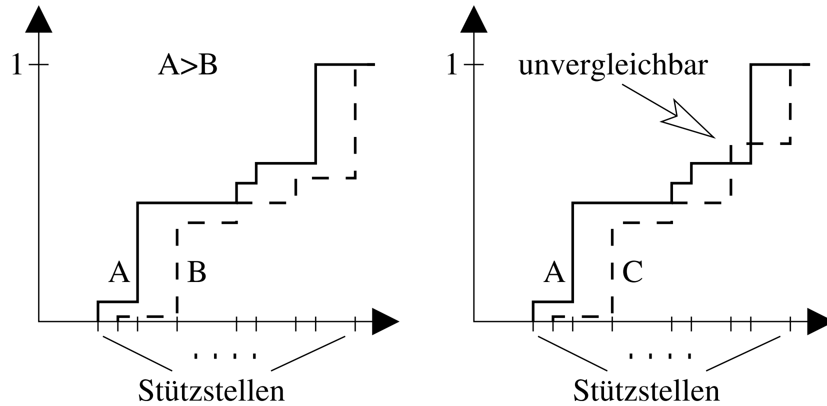

Abbildung 3.1: Vergleichbare und unvergleichbare Verteilungsfunktionen

wie $A_{k,l}$. *Sei* $supp = \{a_x | ((p, a_x) \in A_{i,j} \vee (p, a_x) \in A_{k,l}) \wedge a_x \leq a_{x+1}\}$, *dann gilt:*

$$A_{i,j} > A_{k,l} \;:\Leftrightarrow (\exists a_x \in supp : \sum_{\substack{(p_o,a_o)\in A_{i,j} \\ \text{mit } a_o \leq a_x}} p_o > \sum_{\substack{(p_h,a_h)\in A_{k,l} \\ \text{mit } a_h \leq a_x}} p_h) \;\wedge$$
$$(\forall a_x \in supp : \sum_{\substack{(p_o,a_o)\in A_{i,j} \\ \text{mit } a_o \leq a_x}} p_o \geq \sum_{\substack{(p_h,a_h)\in A_{k,l} \\ \text{mit } a_h \leq a_x}} p_h)$$

Definition 10 (Halbordnung auf Methoden). *Eine Methode* $M_{i,j}$ *ist* größer *als eine Methode* $M_{k,l}$, *wenn* $M_{i,j}$ *selbst eine höhere oder gleiche Qualität liefert bzw. niedrigere oder gleiche Energie verbraucht als* $M_{k,l}$ *und wenn die Verteilungsfunktion der Laufzeiten von* $M_{i,j}$ *größer ist als die von* $M_{k,l}$, *d.h.*

$$M_{i,j} > M_{k,l} \;:\Leftrightarrow\; q_{i,j} \geq q_{k,l} \wedge A_{i,j} > A_{k,l} \textit{ (Qualität)}$$
$$M_{i,j} > M_{k,l} \;:\Leftrightarrow\; e_{i,j} \leq e_{k,l} \wedge A_{i,j} > A_{k,l} \textit{ (Energie)}$$

Definition 11 (Pareto-optimale Methode). *Eine Methode* $M_{i,j} \in M_i$ *heißt* pareto-optimal *bezüglich der Halbordnung* $<$, *wenn es keine Methode* $M_{i,k} \in M_i$ *gibt, die bezüglich dieser Halbordnung größer ist, d.h.:*

$$\neg \exists M_{i,k} \in M_i : M_{i,j} < M_{i,k}$$

Satz 1. *Optimale Lösungen bestehen nur aus pareto-optimalen Methoden.*

Beweisskizze für qualitätsbewußte Einplanung: Sei L eine optimale Lösung, die eine nicht pareto-optimale Methode m_1 enthält und sei m_2 eine pareto-optimale

Methode, die größer als m_1 ist. Ersetzt man in L die Methode m_1 durch m_2, so ergibt sich wieder eine Lösung L', da m_2 eine größere Verteilungsfunktion besitzt (Nachfolger bleiben schedulebar), und L' ist aufgrund der Pareto-Optimalität von m_2 besser als L. Daher kann L keine optimale Lösung gewesen sein. □

3.5 Abschließende Bemerkungen zur Modellierung

Das in diesem Kapitel beschriebene Modell der *Prozesse*, *Methoden* und *Prozessoren*, zusammen mit den eingeführten *Zielfunktionen*, bildet die Eingabe der im nächsten Kapitel beschriebenen Optimierungs- und Einplanungsalgorithmen. Das Modell enthält periodische Prozesse, die abstrakte Aufgaben darstellen und die Datenabhängigkeiten, Bereitzeiten und Fristen besitzen können. Für die Erfüllung der Aufgaben können zu jedem Prozeß alternativ ausführbare Methoden angegeben werden, die sich hinsichtlich ihres Energieverbrauches bzw. der von ihnen gelieferten Qualität unterscheiden. Die Methoden sind nicht unterbrechbar und ihre Ausführungsdauer ist durch eine diskrete Wahrscheinlichkeitsverteilung spezifiziert. Die beiden definierten Zielfunktionen beschreiben die Optimierungsaufgaben, zulässige Schedules mit minimalem durchschnittlichen Energieverbrauch bzw. mit maximaler durchschnittlicher Qualität zu finden. Die auf den Methoden definierte Halbordnung kann dazu verwendet werden, bereits vor der Optimierung Methoden zu entfernen, die aufgrund ihrer Eigenschaften nicht in optimalen Lösungen enthalten sein können.

Kapitel 4

Methodik und Durchführung

Dieser Abschnitt beschreibt zunächst die gemeinsame, rekursive Modellierung des Schedulingproblems und des Optimierungsproblems. Danach folgt der Ablauf des Gesamtalgorithmus, der in zwei Phasen geteilt ist: Erstellen eines bezüglich der Zielfunktion optimalen, flexiblen Schedules und Interpretation des Optimierungsergebnisses zur Laufzeit.

4.1 Modellierung für Dynamische Programmierung

Der in dieser Arbeit vorgestellte Algorithmus basiert auf dem von Bellmann vorgestellten Prinzip der *Dynamischen Programmierung* [Bel57]. In diesem Abschnitt bezeichnet N die Anzahl der Instanzen, M die Anzahl der Methoden je Instanz, K die Anzahl der Laufzeiten je Methode, C die Anzahl der Prozessormodi, I die Menge der einzuplanenden Instanzen und HP die Dauer des Einplanungszeitraumes (Hyperperiode der einzuplanenden periodischen Prozesse). Alle Berechnungen wurden exemplarisch für die Zielfunktion zur Energieminimierung auf einem Monoprozessorsystem[1] durchgeführt.

Definition 12 (Dynamische Programmierung). Dynamische Programmierung *ist ein Lösungsverfahren für Optimierungsprobleme, die folgenden Bedingungen genügen:*

1. *Das Optimierungsproblem ist rekursiv darstellbar.*
2. *Optimale Lösungen bestehen aus optimalen Lösungen zu kleineren Teilproblemen, auf die in der Rekursion zugegriffen wird. (Optimalitätsprinzip)*
3. *Die Rekursion greift mehrfach auf gleiche Teilprobleme zurück.*

[1] Die Verwendung eines Mehrprozessorsystems führt zu einem weiteren Ansteigen der Komplexität.

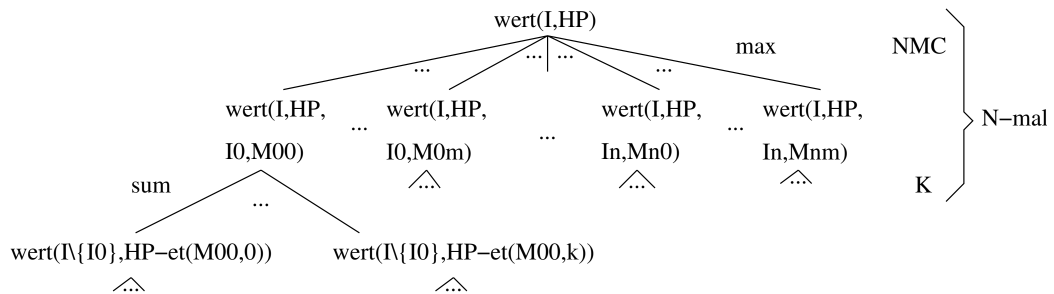

Abbildung 4.1: Rekursive Zusammensetzung einer optimalen Lösung

Während die beiden ersten Bedingungen für die Anwendbarkeit des Verfahrens notwendig sind, kennzeichnet die dritte Bedingung die Problemklasse, für die ihr Einsatz sinnvoll ist.

4.1.1 Rekursive Problembeschreibung

Das Optimierungsproblem für die Energieminimierung läßt sich wie folgt rekursiv beschreiben:

$$wert(I_v, s_v) = \min_{i \in I_v, j \in M_i} wert(j) + \sum_{(p,a) \in A_j} p \cdot wert(I_v \setminus \{i\}, s_v - a)$$

Dabei ist $wert(I_v, s_v)$ der für die Ausführung der Instanzmenge I_v zu erwartende Energieverbrauch, wenn die Ausführung zum Zeitpunkt s_v beginnt. Für die Berechnung wird auf $N \cdot M \cdot K \cdot C$ Teilprobleme zugegriffen (siehe Abbildung 4.1). Die Rekursionstiefe ist N, da in jedem Schritt genau eine Instanz eingeplant wird. Somit ist die Komplexität für die rein rekursive Berechnung des Optimums $O((NMKC)^N)$.

4.1.2 Komplexitätsanalyse bei Dynamischer Programmierung

Bei der Dynamischen Programmierung wird durch geeignete Auflösung der Rekursion jedes Teilproblem nur einmal gelöst und die Lösung in einer Tabelle gespeichert. Daher bestimmt die Anzahl der verschiedenen Teilprobleme die Komplexität. Ein Teilproblem ist durch die Menge der einzuplanenden Instanzen und den Startzeitpunkt eindeutig spezifiziert. Dynamische Programmierung löst die Teilprobleme in einer Reihenfolge, die garantiert, daß die Berechnung jeder Teillösung nur auf bereits gelöste, optimierte Teilprobleme zugreift. Um das zu erreichen, werden die Teilmengen der Instanzmenge nach zunehmender Kardinalität

geordnet und für jede solche Menge werden die Teilprobleme zu den möglichen Startzeitpunkten gelöst.

Alle Teilprobleme und ihre Lösungen werden in einer Tabelle gespeichert. Die Optimierung beginnt in der letzten Zeile der Tabelle. Die darin enthaltenen Zellen werden nacheinander so optimiert, daß die angegebene Instanzmenge ab dem spezifizierten Startzeitpunkt zulässig eingeplant und der Wert der Zielfunktion optimiert wird. Dazu werden alle $N \cdot M \cdot C$ möglichen Kombinationen von Instanz, Methode und Leistungsmodus betrachtet. In den höheren Zellen wird für die Berechnung des Wertes der Zielfunktion auch noch auf die Ergebnisse von K Teilproblemen zugegriffen, d.h. die Optimierung einer Zelle besitzt die Komplexität $O(N \cdot M \cdot K \cdot C)$. Da jede Zeile HP Zellen besitzt, ist die Komplexität zur Optimierung einer Zeile $O(N \cdot M \cdot K \cdot C \cdot HP)$. Für jede Teilmenge der Instanzmenge enthält die Tabelle eine Zeile, wobei die Mächtigkeit der Teilmengen in der Tabelle von unten nach oben monoton steigt. Da es 2^N verschiedene Teilmengen der Instanzmenge gibt und die Zeilen in einem Durchgang von unten nach oben abgearbeitet werden, ist die Gesamtkomplexität der Dynamischen Programmierung $O(N \cdot M \cdot K \cdot C \cdot 2^N \cdot HP) = O(M \cdot K \cdot C \cdot 2^N \cdot HP)$. Die Auswertungsreihenfolge ist in Abbildung 4.2 graphisch dargestellt.

$\lvert I_v \rvert$	s_v I_v	0	1	...	HP	
N	$\{I_0, \ldots, I_N\}$	wert(...) $O(NMKC)$ 1.$\longrightarrow$	wert(...)	...	wert(..)	
...	...	...	...	...	...	
2	$\{I_0, I_1\}$ ... $\{I_{N-1}, I_N\}$	wert(...) ... wert(...)	wert(...) ... wert(...)	... ...	wert(..) ... wert(..)	
1	$\{I_0\}$ ... $\{I_N\}$	wert(...) ... wert(...)	wert(...) ... wert(...)	... ...	wert(..) ... wert(..)	$O(2^N)$ $\uparrow$ 3.
				2. $\longrightarrow$	$O(HP)$	

Abbildung 4.2: Komplexität und Auswertungsreihenfolge bei Dynamischer Programmierung

4.2 Anforderungsgetriebene Dynamische Programmierung

Die Komplexität des Problems ist auch bei Verwendung von Dynamischer Programmierung noch zu hoch, um für realistische Anwendungsgrößen in vernünftiger Zeit ein (optimales) Ergebnis zu erhalten, da dieses erst vorliegt, wenn die Tabelle vollständig gefüllt ist. Aus diesem Grund wird in dieser Arbeit eine *anforderungsgetriebene Dynamische Programmierung* entwickelt. Im Gegensatz zur normalen Dynamischen Programmierung beginnt die Optimierung bei der anforderungsgetriebenen Dynamischen Programmierung bei der Lösung des Gesamtproblems. Die dabei angeforderten Lösungen der Teilprobleme werden sukzessive erzeugt, optimiert und gespeichert, so daß nur Teilprobleme optimiert werden, die auch benötigt werden. Diese Vorgehensweise spart viele Berechnungen ein und es liegt schnell eine erste Lösung vor, wenn das Problem einplanbar ist.

4.2.1 Optimierungseinsparungen

Verschiedene Gründe führen dazu, daß in der Tabelle enthaltene Teilprobleme nicht gelöst werden müssen. Beispielsweise treten nicht alle Teilmengen der Instanzmenge auf, weil die Instanzen in einer die Präzedenzrelation erfüllenden Reihenfolge eingeplant werden müssen. Der folgende Satz enthält eine Abschätzung für die Einsparungen, die sich aus den Instanznummern der Instanzen eines periodischen Prozesses ergeben:

Satz 2. *Seien p die Hyperperiode, p_i die Periodendauern der Prozesse aus der Menge P und N die Anzahl der Instanzen. Durch die Instanznummern reduziert sich die Anzahl der zu untersuchenden Instanzmengen auf* $\prod_{i \in P}(\frac{p}{p_i}+1) \leq 2^N$.

Beweis: $\forall k \in \mathbb{N}_1 : k+1 \leq 2^k$, da $1+1 = 2 \leq 2 = 2^1$ und $\forall k \geq 2 : k+1 = (k-1)+1+1 \leq 2^{k-1}+1+1 = 2^{k-1}+2 \leq 2^{k-1} \cdot 2 = 2^k$, und somit:

$$\prod_{i \in P}(\frac{p}{p_i}+1) \leq \prod_{i \in P} 2^{\frac{p}{p_i}} = 2^{\sum_{i \in P} \frac{p}{p_i}} = 2^N$$

□

Diese Abschätzung ist scharf, da bei einer Menge von periodischen Prozessen, die alle die gleiche Periodendauer besitzen, die linke und rechte Seite gleich sind. Dann ist aber die durch die Zahl der Prozesse bedingte Komplexität relativ gering, da es pro Prozeß nur eine Instanz gibt. Je unterschiedlicher die Periodendauern der Prozesse sind, desto größer sind die Einsparungen, die sich aus der Reihenfolge der Instanzen eines Prozesses ergeben.

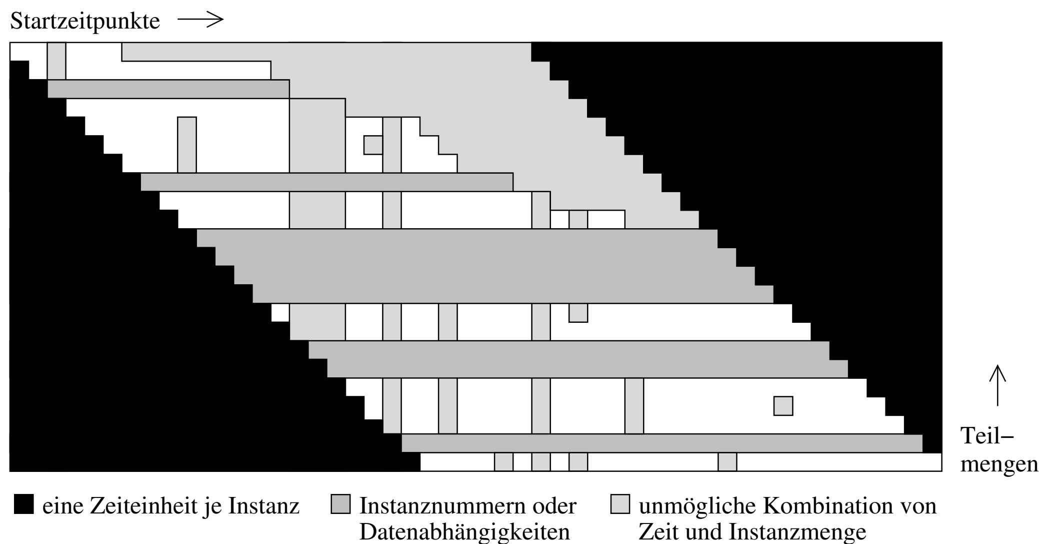

Abbildung 4.3: Einsparungen durch anforderungsgetriebene Dynamische Programmierung (schematisch)

Weitere Einschränkungen resultieren aus den Datenabhängigkeiten zwischen den Instanzen, da auch sie eine Halbordnung auf der Menge der Instanzen definieren. Die Auswirkungen sind ähnlich zu denen der Instanzreihenfolge, d.h. während der Optimierung müssen weniger Teilmengen der Instanzmenge betrachtet werden.

Zusätzlich zu den Einsparungen aufgrund der Präzedenzrelation führt die anforderungsgetriebene Dynamische Programmierung zu großen Einsparungen bei den zu betrachtenden Startzeitpunkten. Viele Startzeitpunkte treten aufgrund der diskreten Spezifikation der Methodenlaufzeiten nicht oder nur in Kombination mit einigen Instanzmengen auf und müssen daher nicht oder zumindest nicht immer betrachtet werden.

Schließlich können viele Startzeitpunkte nicht mit allen Teilmengen der Instanzmenge einen gültigen Schedule liefern. So müssen zum Beispiel die Startzeitpunkte $\{\max(1, p-|I_i|), .., p-1\} \cup \{0, \ldots, |I_i|-1\}$ nicht in Zusammenhang mit der Instanzmenge I_i betrachtet werden, da für jede Instanz mindestens eine Zeiteinheit benötigt wird, und die verbleibende Zeit damit nicht für die verbleibenden Instanzen ausreichen würde.

Die genannten Auswirkungen auf die Tabelle zur Dynamischen Programmierung sind in Abbildung 4.3 schematisch dargestellt. Anforderungsgetriebene Dynamische Programmierung bearbeitet und löst nur die Teilprobleme in den weißen Bereichen der Tabelle. Die dunkelgrauen Zeilen der Tabelle können aufgrund der

Präzedenzrelation der Instanzen (Instanzreihenfolge, Datenabhängigkeiten, Bereitzeiten und Fristen) nicht auftreten, weil die ihnen zugeordneten Teilmengen der Instanzmenge diese Präzedenzrelation verletzen. Der schwarze Bereich tritt nicht auf, weil die Bearbeitung jeder Instanz mindestens eine Zeiteinheit erfordert, und der hellgraue Bereich stellt die Kombinationen von Startzeitpunkten und Teilmengen der Instanzmenge dar, bei denen die verbleibende Zeit nicht für die einzuplanenden Instanzen ausreicht. Die ebenfalls hellgrau eingezeichneten Spalten können nicht auftreten, weil die Startzeitpunkte der darin enthaltenen Zellen keine mögliche Summe von spezifizierten Methodenlaufzeiten sind.

4.2.2 Umsetzung der anforderungsgetriebenen Dynamischen Programmierung

Bei der Umsetzung der anforderungsgetriebenen Dynamischen Programmierung kann der Speicherbedarf, den die Tabelle in Abbildung 4.2 besitzt, vermieden werden, weil sie nur ein kleinen Teil der Tabelleneinträge benötigt. Abbildung 4.4 zeigt eine Lösung und die von ihr referenzierten Teillösungen, wobei Teillösungen mit gleicher Mächtigkeit der Instanzmenge nach hinten in einer Ebene angeordnet wurden.[2] Durch diese neue Anordnung greift eine Teillösung ausschließlich auf Teillösungen in der nächsttieferen Ebene zu, die zusätzlich einen größeren Startzeitpunkt haben. Manche Teillösungen verwenden direkt oder indirekt einige gleiche (kleinere) Teillösungen, weil z.B. nur die Reihenfolge einiger Instanzen vertauscht wurde. Der von den Teillösungen gebildete gerichtete, azyklische Multilevel-Graph stellt eine Lösung des Gesamtproblems dar.

Definition 13 (gerichteter Multilevel-Graph). *Ein Graph heißt* gerichteter Multilevel-Graph*, wenn seine Knoten so in Ebenen angeordnet werden können, daß alle Kanten nach unten gerichtet und zwischen adjazenten Ebenen sind.*

Die Idee ist nun, nur Teillösungen zu erzeugen und in einer dynamischen Datenstruktur zu speichern, die tatsächlich von einer Lösung angefordert werden. Auf diese Weise erhält man sehr schnell eine initiale Lösung. Anschließend werden nach und nach alle anderen möglichen Lösungen und deren Teillösungen erzeugt und optimiert. Es werden also nur die benötigten Zellen angelegt und über eine Abbildung (Hash-Tabelle) verwaltet, die die Beschreibung des Teilproblems (vorher: Tabellenindex) seiner aktuellen Lösung zuordnet. Die große statische Tabellenstruktur entfällt dadurch. Auch ist es nicht nötig, referenzierte Teillösungen schon bei ihrer ersten Anforderung zu optimieren. Es reicht aus, in den Teillösungen Verweise auf die referenzierenden Lösungen zu speichern

[2] Zur übersichtlicheren Darstellung wurden Pfeile zu weiter hinten liegenden Teillösungen nicht eingezeichnet.

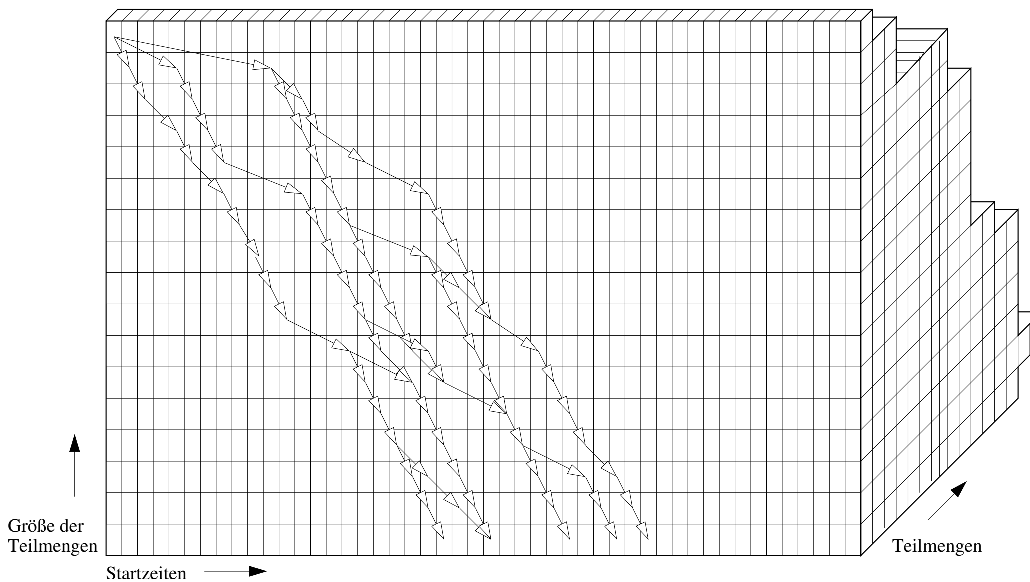

Abbildung 4.4: Referenzierung von Teillösungen (schematisch)

und diese beim einer eintretenden Verbesserung der Teillösung zu aktualisieren. Sobald eine Teillösung erstmals angefordert wird, wird sie dynamisch erzeugt, in einem sogenannten Bedingungsgraph-Knoten gespeichert und der Verweis auf diesen Knoten in die Hash-Tabelle eingetragen. Der Verweis auf die anfordernde Teillösung (Knoten) wird in der neuen Teillösung vermerkt. Sobald sich die Güte einer Teillösung ändert, werden die anfordernden Teillösungen aktualisiert (Abbildung 4.5). Dieser Fall tritt ein, wenn innerhalb eines Knotens eine neue, bessere Entscheidung getroffen wurde, oder wenn eine von der Lösung referenzierte Teillösung verbessert wurde.

Diese neue Strategie für die Knotenerzeugung führt dazu, daß schon nach sehr wenigen Schritte eine erste - noch nicht optimale - Lösung für das Gesamtproblem berechnet ist und ausgegeben werden kann. Die Güte (verbrauchte Energie, gelieferte Qualität) der gelieferten Lösungen verbessert sich, je mehr Knoten erzeugt und optimiert werden und damit erfüllt die anforderungsgetriebene Dynamische Programmierung die Bedingungen eines Anytime-Algorithmus [Zil93].

Definition 14 (Anytime-Algorithmus). *Ein Algorithmus ist ein* Anytime-Algorithmus*, wenn er*

- *(nach der Initialisierung) jederzeit beendet werden kann und dabei ein Ergebnis liefert, und*
- *die Güte des Ergebnisses bezüglich der Ausführungsdauer monoton steigt.*

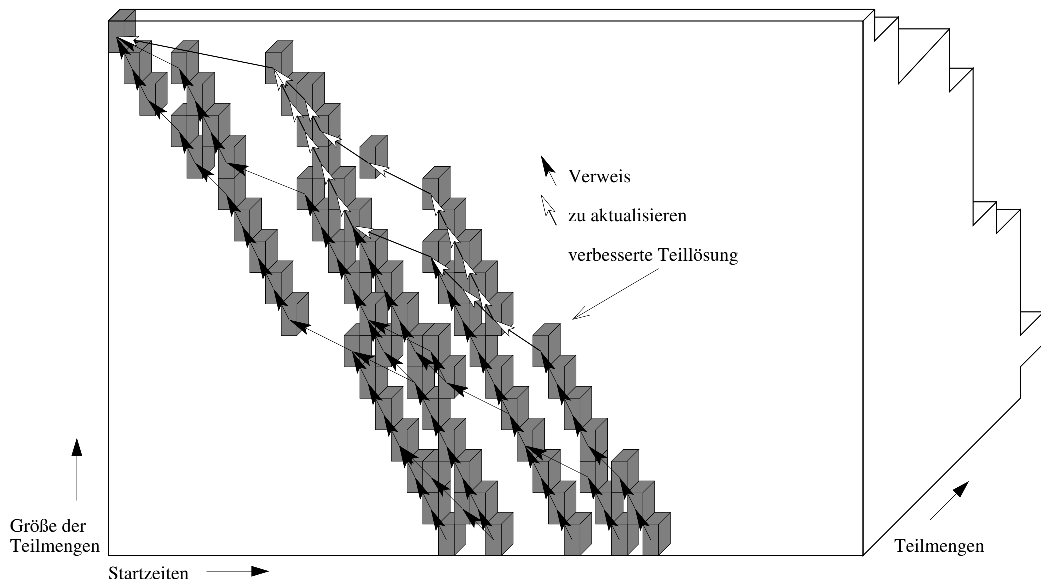

Abbildung 4.5: Dynamische Erzeugung angeforderter Zellen (schematisch)

Da Anytime-Algorithmen - nach Ermittlung einer Initiallösung - ständig Lösungen liefern (und oft nicht terminieren) können sie nicht anhand ihrer worst-case Laufzeit gemessen werden. Stattdessen bestimmt man für jeden Anytime-Algorithmus Leistungsprofile, die den Verlauf der Güte der Lösungen über die verstrichene Zeit beschreiben.

Definition 15 (Leistungsprofil, Halbordnung auf Leistungsprofilen). *Eine Abbildung $L_A : \mathbb{R}^+ \rightarrow \mathbb{R}_0^+$, die jeder positiven Ausführungszeit t eines Anytime-Algorithmus A die von A gelieferte Güte zuordnet, heißt* Leistungsprofil von A. L_A *ist* besser *als L_B, wenn für alle $t \in \mathbb{R}^+$ gilt:*

$$\begin{aligned} L_A(t) &\leq L_B(t) \text{ (Energie)} \\ L_A(t) &\geq L_B(t) \text{ (Qualität).} \end{aligned}$$

Typische Formen von Leistungsprofilen sind in Abbildung 4.6 zu sehen. Ein Leistungsprofil L_1 ist besser als ein Leistungsprofil L_2, wenn die von L_1 gelieferte Güte für jede Optimierungszeit besser oder gleich der von L_2 ist. In vielen Fällen sind Leistungsprofile nicht vergleichbar, sondern jedes Profil dominiert das andere für gewisse Zeitintervalle (Abbildung 4.7).

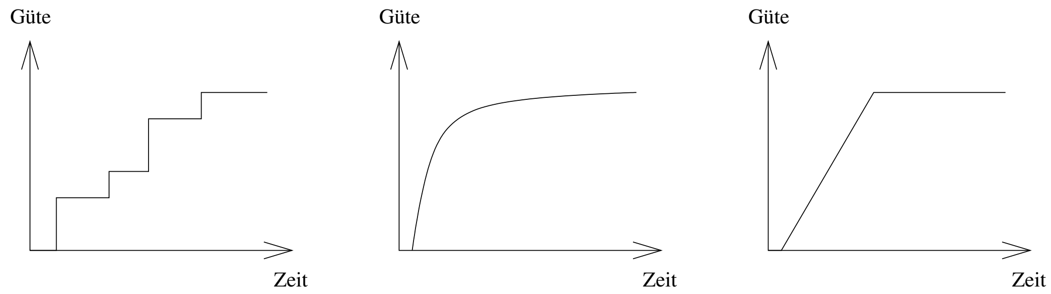

Abbildung 4.6: Einige Typen von Leistungsprofilen

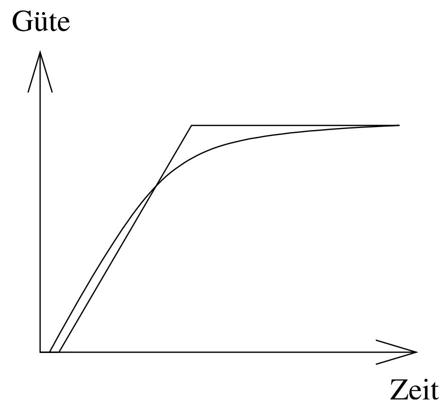

Abbildung 4.7: Unvergleichbare Leistungsprofile

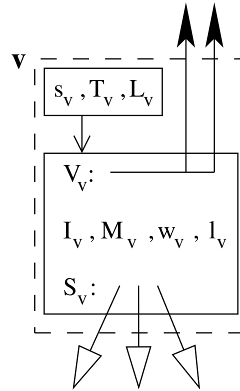

Abbildung 4.8: Aufbau eines Bedingungsgraph-Knotens

Aufbau von Bedingungsgraph-Knoten

Nun werden die bei der Umsetzung verwendeten Bedingungsgraphen vorgestellt, deren Knoten (siehe Abbildung 4.8) die Zellen der Tabelle, die bei Dynamischer Programmierung verwendet wird, ersetzen.

Definition 16 (Bedingungsgraph-Knoten). *Ein* Bedingungsgraph-Knoten $v = (s_v, T_v, L_v, I_v, M_v, l_v, w_v, S_v, V_v, E_v) \in \{0 \dots p\} \times \wp(I) \times L \times I \times M \times L \times \mathbb{R}_0^+ \times \wp((\{0 \dots p\} \times \wp(I))^\infty) \times \wp(I^\infty \times L) \times \wp(([0 \dots 1] \times \mathbb{N})^\infty) =: \mathbb{K}$ *enthält*

- *die spätestmögliche Startzeit* s_v*,*
- *die zu schedulende Instanzmenge* T_v*,*
- *den eingestellten Leistungsmodus des Prozessors* L_v*,*
- *die zu startende Instanz* I_v*,*
- *die zu startende Methode* M_v*,*
- *der Leistungsmodus* l_v *für die Methode* M_v*,*
- *den Erwartungswert* w_v *der Zielfunktion ab* v*,*
- *die Schlüsselmenge (Startzeiten, Instanzmengen und Leistungsmodi) der Söhne* S_v*,*
- *die Menge der Väter* V_v*,*
- *die Menge* E_v *der gewichteten, möglichen Endzeitpunkte der Methode* M_v

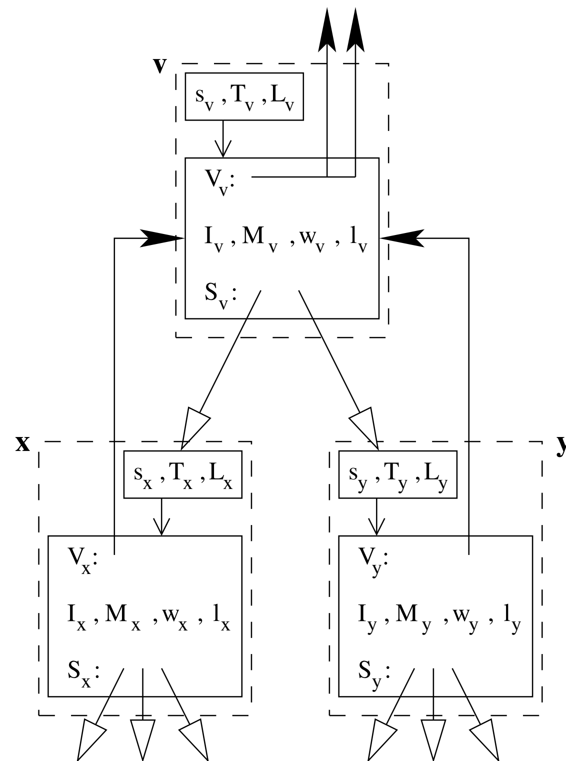

Abbildung 4.9: Ausschnitt eines Bedingungsgraphen

Die Information in einem Knoten v bedeutet, daß es beginnend mit diesem Knoten möglich ist, die Instanzmenge I_v erfolgreich einzuplanen, wenn v spätestens ab dem Zeitpunkt s_v ausgeführt wird und sich der Prozessor im Leistungsmodus L_v befindet. Dabei ist für einen in $wurzel$ verwurzelten Berechnungsgraph BG der Wert w_{wurzel} für die Zielfunktion zu erwarten. Insbesondere bedeutet dies, daß w_{wurzel} der zu erwartende Wert der Zielfunktion für die komplette Anwendung und damit der zu optimierende Wert ist. Der im folgenden vorgestellte Optimierungsalgorithmus setzt voraus, daß der Wert w_v für jeden Knoten allein aufgrund der Information im Knoten und der Werte seiner Söhne berechnet werden kann, d.h.

$$w : (s_v, L_v, I_v, M_v, l_v, \{w_u | u \in S_v\}) \mapsto w_v$$

Definition 17 (Schlüssel eines Knotens). *Das Tripel (s_v, T_v, L_v) wird als* Schlüssel *des Knotens v bezeichnet. Der Schlüssel gibt an, daß beginnend im Knoten v die Instanzmenge T_v eingeplant werden soll, wenn der Knoten spätestens zum Zeitpunkt s_v aufgerufen wird und der Prozessor sich im Leistungsmodus L_v befindet.*

Definition 18 (Zulässiger Knoten). *Ein Knoten v heißt* zulässig*, wenn mit $s_0 := \max\{s_v + t_w(CPU, L_v, l_v), b_{I_v}\}$ gilt:*

- *$I_v \in T_v$ (Prozeßinstanz I_v ist noch zu schedulen)*
- *$M_v \in M_{I_v}$ (Methode M_v implementiert I_v)*
- *$\{I_1 \in I | I_1 <_I I_v\} \cap T_v = \emptyset$ (alle Vorgänger von I_v sind bereits eingeplant)*
- *$\forall (s_u, T_u, L_u) \in S_v : T_u = T_v \setminus \{I_v\} \wedge L_u = l_v \wedge \exists (p, a) \in A_{M_v} : s_u = s_0 + \frac{a}{r(l_v)}$ (für alle möglichen Ausführungszeiten gibt es einen Sohnschlüssel, der die verbleibenden Prozeßinstanzen einplant), oder falls $T_v \setminus \{I_v\} = \emptyset$ gilt $S_v = \emptyset$*
- *$s_0 + WCET(M_v) \cdot \frac{r_{max}(M_v)}{r(l_v)} \leq f_{I_v}$ (die Frist von I_v wird auch im schlimmsten Fall eingehalten)*

In dieser Darstellung wird eine Lösung des Einplanungsproblems durch einen Bedingungsgraphen, dessen Wurzelknoten den Schlüssel $(0, I, L_{0,0})$ besitzt und zu dessen Knoten jeweils alle Söhne vorhanden und zulässig sind, repräsentiert. Jeder Knoten enthält die in der durch seinen Schlüssel beschriebenen Situation zu treffende Einplanungsentscheidung.

Definition 19 (Sohn). *Ein Knoten u heißt* Sohn *eines Knotens v, wenn der Schlüssel von u in der Menge der Schlüssel der Söhne S_v von v enthalten ist.*

Definition 20 (Bedingungsgraph). *Ein azyklischer, gerichteter Graph mit einer Knotenmenge B heißt* Bedingungsgraph*, wenn gilt:*

- *Es gibt genau einen Wurzelknoten $r \in B$. Er besitzt den Schlüssel* $(0, I, L_{0,0})$.
- *Alle Knoten $u \in B \setminus \{r\}$ sind Sohn eines anderen Knotens $v \in B$*
- *Alle Knoten besitzen paarweise verschiedene Schlüssel.*

Definition 21 (vollständiger, zulässiger Bedingungsgraph). *Ein Bedingungsgraph heißt* vollständig*, wenn er alle Söhne aller Knoten enthält. Ein Bedingungsgraph heißt* zulässig*, wenn er nur zulässige Knoten enthält.*

Definition 22 (Lösung, optimale Lösung). *Ein Bedingungsgraph heißt* Lösung*, wenn er vollständig und zulässig ist. Eine Lösung L heißt* optimal *für eine zu maximierende Zielfunktion w_{max} bzw. eine zu minimierende Zielfunktion w_{min}, wenn gilt:*

$$w_{max}(L) = \max_{L_x \in alleLoesungen} \{w_{max}(L_x)\}$$

bzw.

$$w_{min}(L) = \min_{L_x \in alleLoesungen} \{w_{min}(L_x)\}.$$

Die in dieser Arbeit vorgestellten Algorithmen generieren verschiedene Instanz-Methode-Leistungsstufe-Kombinationen für den Wurzelknoten und fügen dem Graphen sukzessive zulässige Knoten hinzu oder entfernen sie. Sie vervollständigen dadurch zulässige Bedingungsgraphen und versuchen auf diese Weise eine optimale Lösung zu finden. Der Suchraum, in dem es eine (von möglicherweise mehreren) optimale Lösung zu finden gilt, besteht daher aus der Menge aller zulässigen Bedingungsgraphen. Abbildung 4.10 zeigt eine stark vereinfachte, abstrakte Skizze des Suchraumes.

Nachbarschaft von Bedingungsgraphen

Seien G und G' zwei Bedingungsgraphen. Dann bezeichnet $V_G^{=}(G')$ die Menge der Knoten in G die auch in G' enthalten sind. $V_G^{\neq}(G')$ ist die Menge der Knoten in G die den gleichen Schlüssel besitzen wie Knoten in G' aber mit einer anderen Instanz oder Methode beginnen oder die Methode mit einer anderen Leistungsstufe ausführen. Die Menge $V_G^{\Delta}(G')$ beinhaltet die Knoten in G, die einen Schlüssel besitzen für den es keinen Knoten in G' gibt.

$$V_G^{=}(G') = \{v \in G | \exists v' \in G' : key_{v'} = key_v \wedge I_{v'} = I_v \wedge M_{v'} = M_v \wedge l_{v'} = l_v\}$$
$$V_G^{\neq}(G') = \{v \in G | \exists v' \in G' : key_{v'} = key_v \wedge (I_{v'} \neq I_v \vee M_{v'} \neq M_v \vee l_{v'} \neq l_v)\}$$
$$V_G^{\Delta}(G') = \{v \in G | \neg \exists v' \in G' : key_{v'} = key_v\}$$

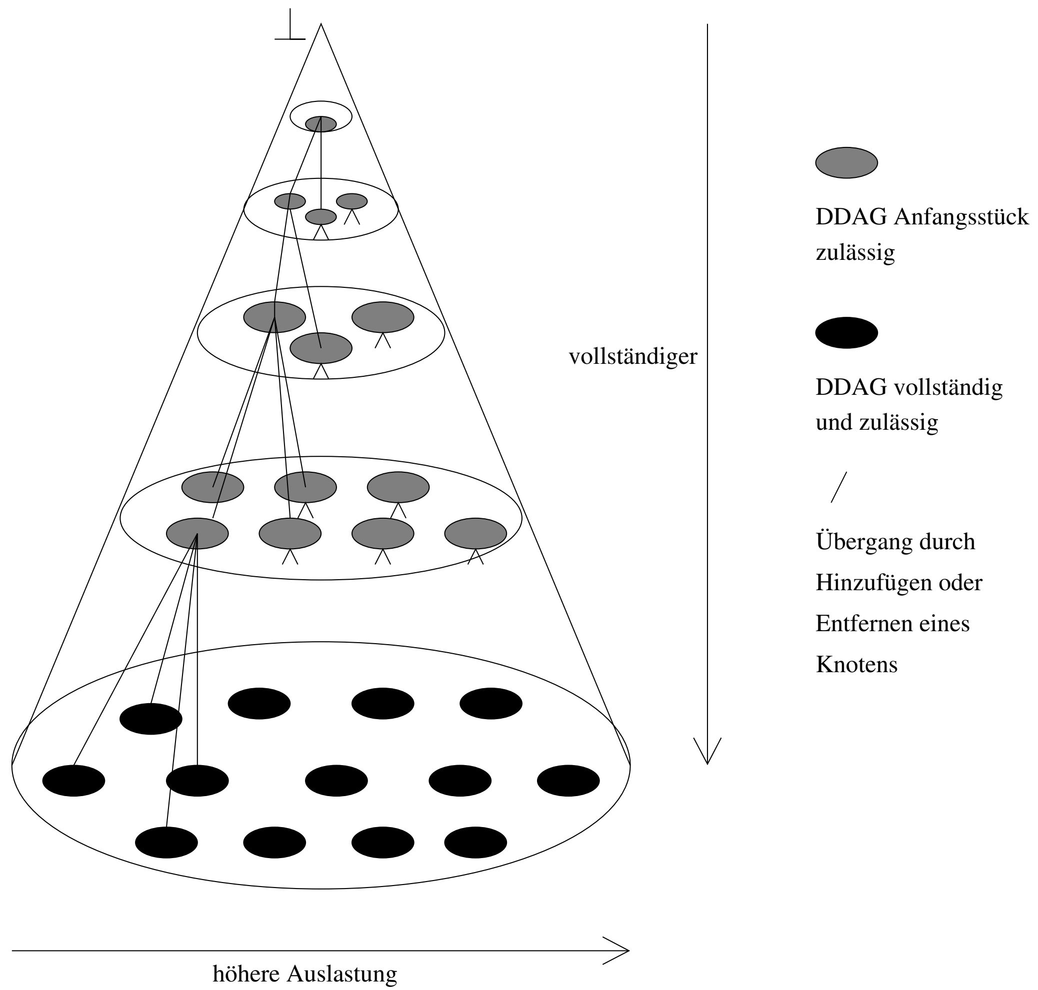

Abbildung 4.10: Bedingungsgraph Suchraum

Definition 23 (Abstand von Bedingungsgraphen). *Der* Abstand $d(B, B')$ *zweier Bedingungsgraphen B und B' ist die Anzahl der Knoten in B und B', die sich hinsichtlich ihres Schlüssels oder der gewählten Instanz und Methode unterscheiden, oder die nur in einem der beiden Graphen enthalten sind:*

$$d(B, B') := \left| \{ v \in V^{\neq}_{B'}(B) \cup V^{\Delta}_{B'}(B) \cup V^{\Delta}_{B}(B') \} \right|$$

Definition 24 (Nachbarschaft von Bedingungsgraphen). *Die Menge $N(B)$ der Bedingungsgraphen, die von einem Bedingungsgraphen B Abstand 1 haben, heißt* Nachbarschaft *von B:*

$$N(B) := \{ B' | d(B, B') = 1 \}$$

Definition 25 (Nachbarlösung). *Eine Lösung L' heißt* Nachbarlösung *der Lösung L, wenn es genau einen Schlüssel gibt, der in beiden Lösungen enthalten ist und für den unterschiedliche Instanzen oder Methoden gewählt wurden, d.h.*
$|V^{\neq}_{L}(L')| = 1$.

Eigenschaften optimaler Lösungen

Satz 3. *Jeder vollständige Teilgraph eines optimalen Bedingungsgraphen ist selbst optimal für den spezifizierten Schlüssel.*

Beweis: Diese Eigenschaft ist leicht durch Widerspruch zu zeigen. Angenommen ein vollständiger Teilgraph eines Bedingungsgraph T_1 des optimalen Bedingungsgraphen G_1 ist nicht optimal, dann gibt es einen optimalen Teil-Graphen T_2, der besser ist als T_1 und damit ist der Graph G_2 der entsteht, wenn in G_1 T_1 durch T_2 ersetzt wird, besser als G_1. Dies widerspricht der Optimalität von G_1. □

Der nächste Satz beschreibt eine Eigenschaft der Nachbarschaft, die für den Einsatz des im nächsten Abschnitt beschriebenen Optimierungsverfahrens Simulated Annealing notwendig ist.

Satz 4 (Erreichbarkeit des Optimums). *Jede optimale Lösung G_{opt} kann von jedem beliebigen Bedingungsgraphen G oder dem leeren Graphen durch iteratives Entfernen und Hinzufügen von Knoten, d.h. durch Übergänge innerhalb der Nachbarschaften, erreicht werden.*

Beweis: Sei G_{opt} die zu erreichende Lösung, G der aktuelle Entscheidungsgraph oder der leere Graph. Man erhält G_{opt} durch iteratives Entfernen aller Knoten, die in G aber nicht in G_{opt} sind, und anschließendes iteratives Hinzufügen aller Knoten, die in G_{opt} aber nicht in G sind. □

4.3 Optimierungsalgorithmen

Im Rahmen dieser Arbeit wurde neben der anforderungsgetriebenen Dynamischen Programmierung, die dazu dient optimale Lösungen zu finden, ein zweiter Algorithmus zum schnelleren Auffinden guter - wenn auch nicht zwingenderweise optimaler - Lösungen entworfen. Zu beiden Verfahren existieren zahlreiche Varianten. Alle in dieser Arbeit entwickelten Varianten besitzen die Anytime-Eigenschaft, d.h. sie zeichnen sich dadurch aus, daß sie abbrechbar sind, und daß sie bessere Lösungen liefern je später der Abbruch erfolgt. Ein Anytime-Algorithmus ist besser als ein anderer, wenn er in der gleichen Zeit bessere Lösungen liefert. Während der Algorithmus der anforderungsgetriebenen Dynamischen Programmierung das Finden einer optimalen Lösung garantiert, ist der auf Simulated Annealing basierende Algorithmus darauf ausgelegt, bessere Lösungen als der erste zu finden, wenn nicht genügend Zeit für eine vollständige Optimierung zur Verfügung steht.

4.3.1 Anforderungsgetriebene Dynamische Programmierung

Ein Algorithmus der das Auffinden einer optimalen Lösung garantiert, muß sicherstellen, daß er alle Elemente des Suchraumes bewertet, oder aber, daß er nur Elemente des Suchraumes nicht bewertet, die nicht optimal sein können. Dieses Verhalten ist durch einen rekursiven Ansatz umsetzbar, der, beginnend mit der Wurzel, alle zulässigen Knoten und deren Söhne aufzählt. Falls eine Abschätzung des bestmöglichen Knotenwertes ohne Kenntnis seiner Söhne einen schlechteren als den besten bekannten Wert für den Schlüssel des Knotens ergibt, kann die rekursive Aufzählung seiner Söhne eingespart werden.

Die anforderungsgetriebene Dynamische Programmierung ist rekursiv implementiert. Zunächst wird der Knoten für den Ausgangsschlüssel angefordert. Da dieser nicht existiert, werden er und alle benötigten Söhne erzeugt. Diese Knoten bilden die Initiallösung. Anschließend werden alle Knoten beginnend bei den Blättern optimiert, d.h. es werden alle möglichen bzw. erfolgversprechenden Kombinationen von Instanzen, Methoden und Leistungsstufen untersucht.

```
double optimize() {
    k := (0,I);
    w_opt := optimize(k);
    return w_opt;
}

double optimize(Schlüssel k) {
    w_k := -infinity;

    // untersuche alle Instanz-Methode Kombinationen
    forall (I_v,M_v,L_v) zulässig für k do
        if (w_schätz > w_k) then
```

```
        if (|I_v| > 1) then

          // hole die optimalen Werte der Söhne
         forall sk(i) Schlüssel eines Sohnes von v do
         if (!isMarkedAsOptimized(sk(i))) then

         // Sohn wurde noch nicht optimiert
                 w_sk(i) := optimize(sk(i));
             fi;
          od;
          w_neu := max(w_k,w(M_v,L_v,w_sk(0),...,w_sk(m));
        else

            // berechne den Wert der Senke
            w_neu = w(M_v);
        fi;

        // neuer Wert besser als der alte?
        if (w_neu > w_k) then
            Knoten v := new Knoten(k,I_v,M_v);
            w_k := w_neu;
            save(k,v,w_k);
        od;
    fi;
od;

// Schlüssel k ist vollständig optimiert
markAsOptimized(k);
return w_k;
}
```

Da in der Implementierung immer nur Aktualisierungen durchgeführt werden, wenn ein Vaterknoten angefragt wird, ist der Aufwand gering. Durch die rekursive Anforderung der Knoten wird jeder Knoten nur einmal aktualisiert, und zwar dann, wenn alle seine Söhne optimiert sind.

4.3.2 Simuliertes Ausglühen

Der zweite Algorithmus verwendet ein Verfahren das in Analogie zum Ausglühen von Metallen entwickelt wurde. Beim Simulierten Ausglühen (Simulated Annealing) wird immer eine bereits gefundene Lösung modifiziert und dann anhand der Änderung der Zielfunktion entschieden, ob die Ausgangs- oder die modifizierte Lösung beibehalten wird. Hierbei werden teilweise auch Verschlechterungen in Kauf genommen, um in komplexen Suchräumen nicht vorzeitig in einem lokalen Optimum zu konvergieren.

```
SimulatedAnnealing(temperature,stopTemperature,coolControl,constTempSteps) {

  // Initiallösung berechnen
  L = getInitialSolution();

  do { // bis zum Erreichen des Stoppkriteriums
    do {// während konstanter Temperatur
```

```
      // Opferknoten zufällig auswählen
      Node toChange = chooseVictim();

      // Zufällige Änderung durchführen
      newChoice = randomChoice();
      L' = change(L,toChange,newChoice);

      // Änderung der Wertfunktion bestimmen
      double valueDiff = quality(L') - quality(L);

      // bessere, neue Knoten akzeptieren
      if (valueDiff >= 0.0) {
        L = L';
      }

      // Verschlechterungen eventuell akzeptieren
      if (Math.exp(valueDiff / temperature) > (randGen.nextDouble())) {
        L = L';
      }

   steps++;
    } while (steps < this.constTempSteps ); // Temperatur konstant halten

    // Parameter aktualisieren
    steps = 0;
    temperature = temperature * coolControl;

  } while ( this.temperature > this.stopTemperature );  // abkühlen
}
```

Da Simulated Annealing den Suchraum nicht systematisch abarbeitet, sondern in jedem Schritt einen Knoten aus einer beliebigen Ebene ändert, ist die Anzahl der jeweils neu zu erzeugenden und zu aktualisierenden Knoten größer als im Falle der Dynamischen Programmierung.

4.3.3 Abschneiden von Teil-Bedingungsgraphen

Viele unnötige Teil-Bedingungsgraphen können durch die Überprüfung von Abbruchbedingungen vermieden werden. Da die Kosten für die Erzeugung und Optimierung eines Teil-Bedingungsgraphen sehr hoch sind, lohnt es sich diese einfach zu berechnenden Abbruchbedingungen zu prüfen. Die erste Bedingung ist, daß die Rekursion abgebrochen werden kann, falls die verbleibende Zeit bei höchster Prozessorleistung nicht mehr für die Ausführung der schnellsten Methoden der einzuplanenden Prozesse ausreicht. Diese Bedingung muß nur für die worst-case Ausführungszeiten der schnellsten Methoden geprüft werden, weil in diesem Fall kein zulässiger Plan existieren kann, der alle Prozesse im schlimmsten Fall rechtzeitig ausführt, d.h. es werden Fristen überschritten.

$$\text{Abbruchbedingung 1 (Restzeit):} \sum_{I_i \in T_v} minWCET(I_i) > HP - s_v$$

Die zweite Bedingung resultiert aus der Tatsache, daß eine weitere Optimierung keinen Nutzen bringt, wenn die Summe der besten Qualitäten bzw. niedrigsten Energien der einzuplanenden Prozesse kleiner oder gleich der Qualität bzw. größer oder gleich der Energie der besten bereits bekannten Lösung ist.

$$\text{Abbruchbedingung 2a (Qualität): } \sum_{I_i \in T_v} maxQ(I_i) \leq w_{v_{opt}}$$

$$\text{Abbruchbedingung 2b (Energie): } \sum_{I_i \in T_v} minE(I_i) \geq w_{v_{opt}}$$

Die dritte Bedingung für einen Abbruch der Optimierung ist erfüllt, wenn es eine Instanz I_i gibt, für die die Zeit zwischen dem spätestmöglichen Ende der ausgeführten Methode m_v bis zur Frist $DL(I_i)$ nicht für den schlechtesten Fall der schnellsten Methode von I_i in der höchsten Leistungsstufe ausreicht.

Abbruchbedingung 3 (Fristverletzung):

$$\exists I_i \in T_v : s_v + \text{WCET}(m_v) > DL(I_i) - \text{minWCET}(I_i)$$

4.3.4 Vergröberung des Zeitrasters

Bei der Optimierung kann die Auflösung des verwendeten Zeitrasters vergröbert werden, um den Aufwand zu verringern. Für ein gröberes Zeitraster als 1 werden die Ausführungszeiten der Methoden erst während der Optimierung an das Raster angepaßt, um die Verluste möglichst gering zu halten. Jeder mögliche Endzeitpunkt einer eingeplanten Methode wird auf den nächsten gleichen oder größeren Rasterpunkt gerundet. Für Endzeitpunkte die zusammenfallen, wird derselbe Sohn verwendet. Dadurch sollten während der Optimierung bei einer Rastervergröberung weniger Knoten benötigt werden, und die Lösungsgraphen sollten weniger Knoten enthalten.

4.4 Nachbearbeitung

Um den Speicherbedarf der Datenstruktur zur Laufzeit der Anwendung gering zu halten, wird die Anzahl der Knoten des Bedingungsgraphen minimiert und der Graph anschließend in eine Moore-Maschine transformiert.

4.4.1 Bedingungsgraph Kompaktifizierung

Da die für die Lösungen verwendete Graphstruktur groß werden kann, wird sie vor der Anwendungsausführung verkleinert, ohne dabei den Wert der Zielfunktion zu verändern. Die verkleinerte Lösung enthält nur noch die zur Laufzeit der

Anwendung benötigte Information und keine Daten, die nur für die Optimierung nötig sind. Abschließend wird aus dieser Lösung ein Endlicher Automat erzeugt werden.

Zusammenfassung von Knoten mit gleicher Ausführungssequenz

Beginnend mit den Blättern des Bedingungsgraphen können Knoten $v_1, \ldots, v_k$ zusammengefaßt werden, für die gilt:

1. $T_{v_1} = T_{v_i}$
2. $L_{v_1} = L_{v_i}$
3. $I_{v_1} = I_{v_i}$
4. $m_{v_1} = m_{v_i}$
5. $l_{v_1} = l_{v_i}$

Bei der Zusammenfassung werden die Knoten gelöscht und durch den neuen Knoten v ersetzt mit:

1. $s_v = \max\{s_{v_i} | i = 1..k\}$
2. $T_v = T_{v_1}$
3. $L_v = L_{v_1}$
4. $I_v = I_{v_1}$
5. $m_v = m_{v_1}$
6. $l_v = l_{v_1}$
7. $S_v = \bigcup_{i=1...k} sons_{v_i}$

Die Verweise in den Vaterknoten auf $v_1, \ldots, v_k$ werden durch den Verweis auf v ersetzt.

Bei Bedingungsgraphen, die mittels Simulated Annealing erzeugt wurden, kann es vorkommen, das der Graph Knoten v_1, v_2 mit $T_{v_1} = T_{v_2}$, $L_{v_1} = L_{v_2}$, $s_{v_1} < s_{v_2}$, $w_{v_1} < w_{v_2}$ (bei zu maximierender Zielfunktion) bzw. $w_{v_1} > w_{v_2}$ (bei zu minimierender Zielfunktion) und $I_{v_1} \neq I_{v_1}$ oder $M_{v_1} \neq M_{v_1}$ oder $l_{v_1} \neq l_{v_2}$ enthalten, d.h. obwohl v_1 mehr Zeit zur Verfügung hat, liefert er ein schlechteres Ergebnis. In diesem Fall wird v_1 vor der Anwendung obiger Kompaktifizierung gelöscht und Verweise auf ihn werden durch Verweise auf v_2 ersetzt.

Berechnung der Knoten- und Kantengewichte

Falls die Knoten- und Kantengewichte des Graphen während der Kompaktifizierung erhalten bleiben sollen, z.B. um den Graphen zu analysieren, sind folgende Berechnungen durchzuführen:

Berechnung der Gewichte der in den Knoten v eingehenden Kanten:

$$p_{v' \to v} = p \text{ , mit } (p, n) \in E_{v'} \wedge n = s_v$$

Berechnung der Knotengewichte:

$$p_v = 1 \text{ , falls } v \text{ die Wurzel des Graphen ist}$$
$$p_v = \sum_{v' \in V_v} p_{v'} \cdot p_{v' \to v} \text{ , sonst.}$$

Beim Zusammenfassen einer Knotenmenge $K = \{v_1, \ldots, v_n\}$ zu einem Knoten v_x werden die Gewichte neu berechnet. Gewicht der aus dem neuen Knoten ausgehenden Kanten:

$$\forall v_y \in \bigcup_{v \in K} sons_v : p_{v_x \to v_y} = \frac{1}{\sum_{v \in K} p_v} \cdot \sum_{v \in K} p_v \cdot p_{v \to v_y}$$

Gewicht der in den neuen Knoten eingehenden Kanten:

$$\forall v_y \in \bigcup_{v \in K} fathers_v : p_{v_y \to v_x} = \sum_{v \in K} p_{v_y \to v}$$

Gewicht des neuen Knotens:

$$p_{v_x} = \sum_{v \in K} p_v$$

Abbildung 4.11 zeigt einen optimalen Bedingungsgraphen für die Anwendung aus Abbildung 1.3 und Abbildung 4.12 zeigt den zugehörigen kompakten Bedingungsgraphen. Die Abbildungen 4.13,4.14,4.15 und 4.16 zeigen Bedingungsgraphen, die für ein anderes Beispiel erstellt wurden, und sie verdeutlichen den Größenunterschied zwischen nicht kompaktifizierten und kompaktifizierten Bedingungsgraphen.

Transformation Bedingungsgraph → Moore-Maschine

Ein (minimierter) Graph G kann leicht in einen endlichen Automaten mit Ausgabe (Moore-Maschine) umgewandelt und anschließend in einer kompakten Zustandsübergangstabelle kodiert werden.

Die Moore-Maschine $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ besteht aus

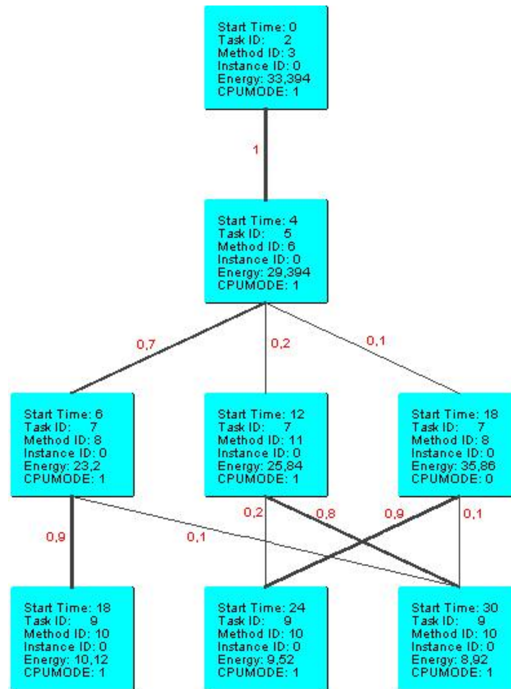

Abbildung 4.11: Optimaler Bedingungsgraph

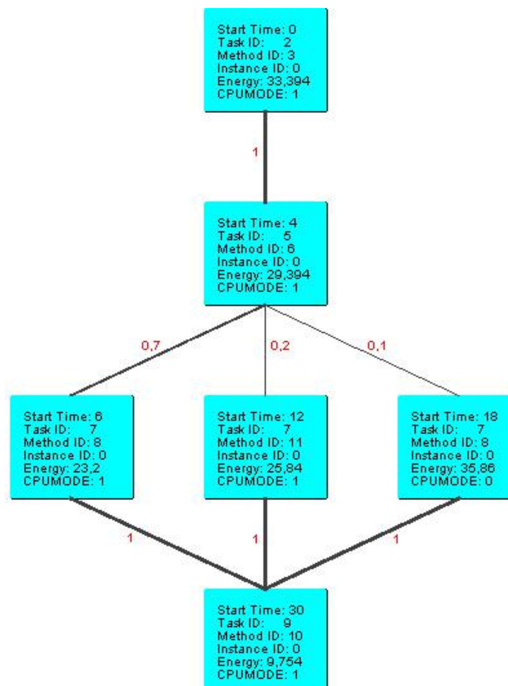

Abbildung 4.12: Kompakter optimaler Bedingungsgraph

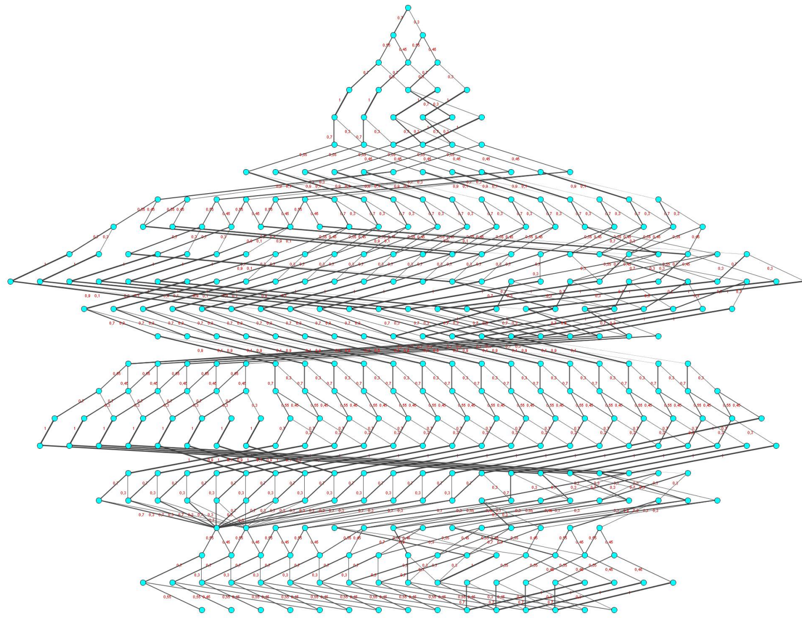

Abbildung 4.13: Optimaler Bedingungsgraph (DVD)

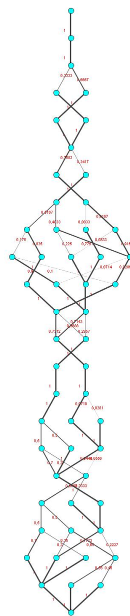

Abbildung 4.14: Kompakter, optimaler Bedingungsgraph (DVD)

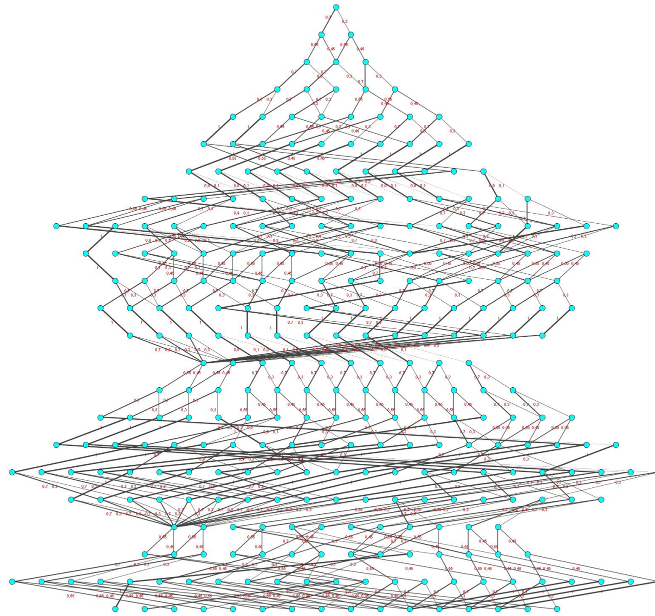

Abbildung 4.15: Fast optimaler Bedingungsgraph (DVD, Simulated Annealing)

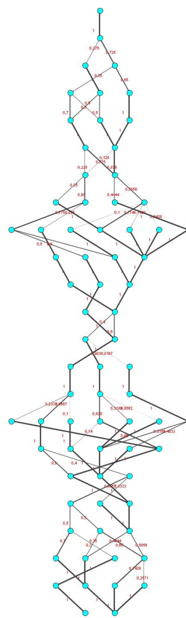

Abbildung 4.16: Kompakter, fast optimaler Bedingungsgraph (DVD, Simulated Annealing)

- der Zustandsmenge $Q = \{q | q = vertex(v) \land v \in G\} \cup \{error\}$,
- dem Eingabealphabet $\Sigma = \{0 \dots (HP - 1)\}$,
- dem Ausgabealphabet $\Delta = \{(t, m, l) | t \in I \land m \in M \land l \in L\}$,
- der Übergangsfunktion $\delta : (q, s) \mapsto q'$,
- der Ausgabefunktion $\lambda : q \mapsto (I_{vertex^{-1}(q)}, M_{vertex^{-1}(q)})$ und
- dem Startzustand $q_0 = vertex(root(G))$.

Für die Definition der Zustandübergangsfunktion δ wird noch die Menge der nach aufsteigender Startzeit geordneten Nachfolger von q benötigt:

$$
\begin{aligned}
succ: \quad & Q \to 2^Q \text{ mit} \\
& q \mapsto \{z_i \in Q | 1 \leq i \leq \#succ(q) \land vertex^{-1}(z_i) \in succ(vertex^{-1}(q) \\
& \qquad\qquad \land \forall_{i=1..k-1} : s(z_i) < s(z_{i+1})\} \\
& \text{mit } s(z_i) := s_{vertex^{-1}(z_i)}
\end{aligned}
$$

$$
\delta(q, s) = \begin{cases} z_i & \text{, falls } s(z_{i-1}) < s \leq s(z_i) \land i \in \{1..\#succ(q)\} \\ error & \text{, falls } s > s(z_{\#succ(q)}) \\ q_0 & \text{, sonst.} \end{cases}
$$

Abbildung 4.17 zeigt den endlichen Automaten für den Bedingungsgraphen aus Abbildung 4.12. Der im Fehlerfall allen Zuständen folgende Zustand $error$ aus Übersichtlichkeitsgründen nicht dargestellt.

4.5 Dynamisches Scheduling

Der dynamische Scheduler interpretiert die eingegebene Moore-Maschine. Abbildung 4.18 zeigt den Pseudoquelltext des Schedulers. Zu Beginn der Ausführung der Anwendung wechselt er in den Startzustand und beginnt die nötigen Methoden auszuführen. Zunächst versetzt er den angegebenen Prozessor in den vorgegebenen Leistungsmodus. Danach wartet er bis die Bereitzeit des eingeplanten Prozesses erreicht ist und weist dann der gewählten Methode den Prozessor zu. Sobald die Methode ihre Berechnungen beendet hat, verzweigt der Scheduler entsprechend der verstrichenen Zeit in den nächsten Zustand. Hat die Methode ihre spezifizierte worst-case Laufzeit eingehalten, kommt die nächste Methode zur Ausführung. Andernfalls wird der Zustand $error$ erreicht, d.h. die dem Optimierer

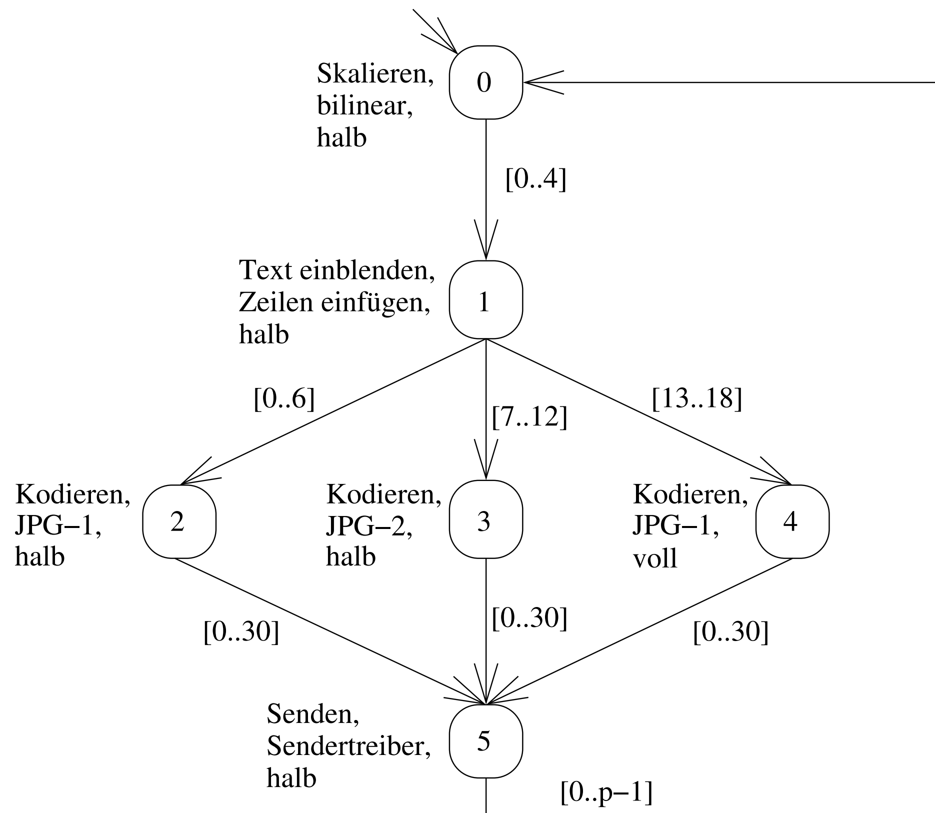

Abbildung 4.17: Endlicher Automat für Live-Video Anwendung

```
int schedule(Automaton A) {
    State v = A.getStartstate();

    while ((v != null) && (!STOP_SIGNALED)) do

        // schedule specified method
        setProcessorSpeed(l_v);
        waitForReleaseTimeOf(I_v);
        run(M_v);

        // state transition
        Time t = currentTime MOD p;
        v = delta(v,t)

        // recovery needed?
        if (v == error) then
            v = recover(v,t);
        fi;
    od;
}
```

Abbildung 4.18: Pseudoquelltext des dynamischen Schedulers

übergebene Spezifikation der Methode ist fehlerhaft. Dieser Fall kann eintreten, wenn die modellierte Anwendung noch in der Entwicklung ist und sich die Laufzeiten der Methoden gegenüber der Spezifikation geändert haben. In diesem Fall kann das Programm abgebrochen werden, wenn eine weitere Ausführung katastrophale Folgen hätte, oder es wird versucht mit dem am besten geeigneten Zustand fortzufahren (siehe z.B. Anhang A). Sind jedoch keine Schäden zu befürchten, wird die neue Laufzeit in das Profil der Methode aufgenommen und kann bei einer erneuten Optimierung berücksichtigt werden. Abbildung 4.19 zeigt den Pseudoquelltext der Fehlerbehandlung.

4.6 Abschließende Bemerkungen zur Methodik und Durchführung

Das Einplanungs- und Optimierungsproblem wurde in der Notation der Dynamischen Programmierung modelliert. Anschließend wurde eine neue Variante der

```
State recover(State v, Time t) {
    if (HARMFUL) then

        // harmful deadline miss: stop execution
        v = null;
    else

        // non-harmful deadline miss: ignore
        t = s(z_(#succ(v)));
        v = delta(v,t);
    fi;
    return v;
}
```

Abbildung 4.19: Pseudoquelltext der Fehlerbehandlung

Dynamischen Programmierung entwickelt, die (meist) zu großen Zeiteinsparungen während der Optimierung führt. Zusätzlich besitzt diese Variante die vorteilhafte Eigenschaft eines Anytime-Algorithmus, bereits während der Optimierung Lösungen mit steigender Güte zur Verfügung zu stellen. Nach der Vorstellung der beiden in dieser Arbeit verwendeten Optimierungsverfahren (*anforderungsgetriebene Dynamische Programmierung* und *Simulated Annealing*) wurden Möglichkeiten für weitere Verkleinerungen des Suchraumes angeführt, und die Nachbearbeitung der flexiblen Schedules (*Bedingungsgraphen*) zur Reduzierung ihres Speicherbedarfs wurde vorgestellt. Die Beschreibung des *dynamischen Schedulers*, der während der Anwendungsausführung verwendet wird, komplettiert den Gesamtalgorithmus.

Das nächste Kapitel enthält Details zur strukturellen Aufteilung der Optimierungsverfahren in mehrere Komponenten, die teilweise in beiden Verfahren anwendbar sind.

Kapitel 5

Strukturelle Umsetzung der Optimierer

Beide Optimierungsalgorithmen haben die Aufgabe im Suchraum eine optimale Lösung zu finden. Es ist nicht möglich einen Bedingungsgraphen in einem Schritt zu erzeugen, weil er sukzessive durch Erweitern von Anfangsstücken erzeugt werden muß. Anfangsstücke können aber nicht mit der Zielfunktion bewertet werden, da diese nur für vollständige Bedingungsgraphen definiert ist. Aus diesen Gründen wird eine Nachbarschaft auf der Menge der Lösungen definiert, und die Optimierungsalgorithmen (*Optimierer*) führen nur Schritte innerhalb dieser Nachbarschaft aus und bewerten sie (*Makroschritt*). Die für solche Übergänge benötigten Zwischenschritte (*Mikroschritt*) im Raum der Anfangsstücke werden ohne Auswertung der Zielfunktion automatisch durch einen zweiten Algorithmus (*Knotenspeicher*) ausgeführt. Sowohl die Optimierer als auch der Knotenspeicher werden bei der Erzeugung neuer Knoten von einem dritten Algorithmus (*Lotse*) unterstützt, der in den meisten Fällen einen Scheduling- oder Aufzählalgorithmus enthält.

Das Zusammenspiel der drei Komponenten Optimierer, Knotenspeicher und Lotse ist in Abbildung 5.1 zu sehen. Zu Beginn fordert der Optimierer beim Knotenspeicher eine Initiallösung an. Danach entscheidet der Optimierer, an welcher Stelle diese Lösung modifiziert werden soll, und der Lotse bestimmt, welche Änderung vorgenommen werden soll. Um den Schritt auszuführen, erzeugt der Knotenspeicher sowohl den angeforderten Knoten als auch rekursiv alle noch nicht existierenden Söhne des neuen Knotens. Falls der Knoten und alle seine Söhne erfolgreich erzeugt werden konnten, so erhält der Optimierer die neue Lösung, ansonsten wird die alte Lösung beibehalten. Falls eine neue Lösung erzeugt wurde, entscheidet der Optimierer anhand des Wertes der Zielfunktion, ob die alte oder die neue Lösung beibehalten werden soll, und der nächste Optimierungsschritt kann ausgeführt werden.

Ein Makroschritt erfordert in der Regel die Ausführung mehrerer Mikroschrit-

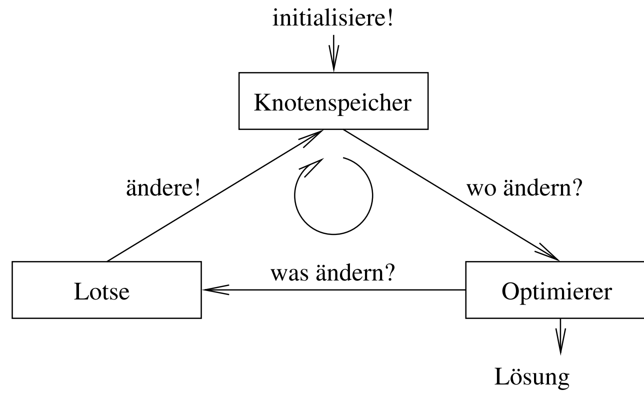

Abbildung 5.1: Zusammenspiel von Optimierer, Knotenspeicher und Lotse

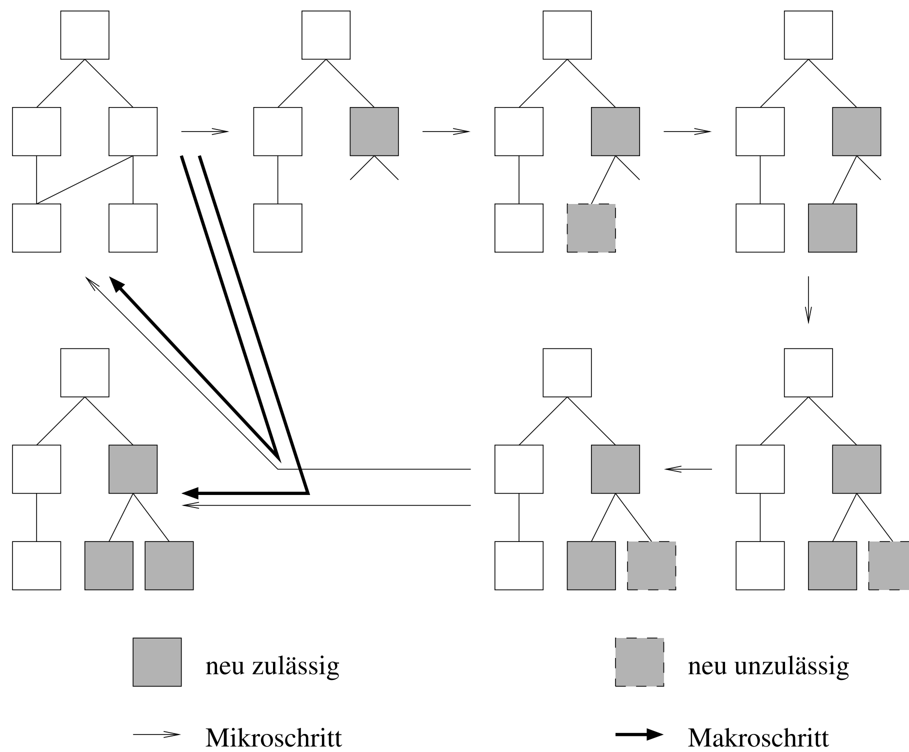

Abbildung 5.2: Mikro- und Makroschritte

te. Ein Makroschritt wird ausgeführt, wenn die in einem Knoten einer Lösung geschedulte Instanz, Methode oder Leistungsstufe geändert werden soll, um eine neue Lösung zu erreichen. In den meisten Fällen ist eine solche Änderung mit dem Ersetzen/Erzeugen des im geänderten Knoten beginnenden Teil-Bedingungsgraphen verbunden, d.h. es muß eine Reihe von Mikroschritten ausgeführt werden um die fehlenden Söhne zu suchen oder zu erzeugen. Das Resultat des Makroschrittes ist die erste von einem Mikroschritt erreichte Lösung, oder der Ausgangsgraph des Makroschrittes, falls es keine Lösung gibt, die die gewünschte Änderung enthält, oder, wenn die neue Lösung vom Optimierer abgelehnt wird. Abbildung 5.2 zeigt mögliche Mikroschritte zur Ausführung eines Makroschrittes.

5.1 Lotsen

Ein Lotse wählt die Instanz, Methode und Leistungsstufe aus, die am wahrscheinlichsten in der Optimallösung auftritt. Aufgrund der Struktur des Bedingungsgraphen muß er jedoch nicht alle möglichen Ausführungszeiten der Methoden betrachten, sondern kann sich auf jeweils eine der möglichen Zeiten beschränken und zu dieser einen Schedule für die verbleibende Instanzmenge suchen.

Die Eingabe für einen Lotsen ist die Beschreibung der Ausgangssituation. Eine Situation ist durch den Startzeitpunkt, die aktuelle Leistungsstufe und die verbleibende Instanzmenge, sowie deren Zeitbedingungen und Datenabhängigkeiten, charakterisiert. Außerdem erhält der Lotse die Information, welche Instanz, Methode und Leistungsstufe zuletzt in dieser Situation (von ihm) gewählt wurde. Die Ausgabe des Lotsen ist die als erste auszuführende Instanz, sowie die dafür zu verwendende Methode und Leistungsstufe. Da für die meisten implementierten Lotsen Scheduling-Heuristiken verwendet werden, die die Einhaltung der Echtzeitbedingungen der verbleibenden Instanzen nicht garantieren, ist Backtracking notwendig. Falls der Lotse in eine unzulässige Situation kommt, werden die zuvor eingeplanten Instanzen, Methoden und Leistungsstufen mittels Backtracking ausgetauscht.

Definition 26 (vollständiger, deterministischer Lotse). *Ein Lotse heißt* vollständig*, wenn er bei k-maligem Aufruf alle möglichen, zulässigen Instanz-Methode-Leistungsstufe-Kombinationen einmal vorschlägt, wobei k die Anzahl der möglichen Kombinationen ist. Ein Lotse heißt* deterministisch*, wenn er für eine bestimmte Situation und eine bestimmte zuletzt verwendeten Instanz-Methode-Leistungsstufe-Kombination immer dieselbe neue Instanz-Methode-Leistungsstufe-Kombination vorschlägt.*

Aktuell sind ein Zufallslotse (MC) sowie die Scheduling-Heuristiken Earliest-

(EDF, GREEDY) Kombination	Zeitbedarf	(EDF, ASC_T) Kombination	Zeitbedarf
(Kodieren, JPG-1, halb)	24 ms	(Kodieren, JPG-2, voll)	9 ms
(Kodieren, JPG-2, voll)	18 ms	(Kodieren, JPG-1, halb)	12 ms
(Kodieren, JPG-1, halb)	12 ms	(Kodieren, JPG-2, voll)	18 ms
(Kodieren, JPG-2, voll)	9 ms	(Kodieren, JPG-1, halb)	24 ms
null	-	null	-

Tabelle 5.1: Vorschläge der Lotsen (EDF, GREEDY) und (EDF, ASC_T)

Deadline-First (EDF), Earliest-Releasetime-First (ERF) und Rate-Monotonic (RM) implementiert. Im Gegensatz zu den letzten drei Lotsen ist der Zufallslotse weder vollständig noch deterministisch.

5.1.1 Sortierte Lotsen (EDF, ERF, RM)

Bei den sortierten Lotsen handelt es sich um vollständige, deterministische Lotsen. Vor Beginn der Optimierung werden dabei die Prozeßinstanzen entsprechend des Sortierkriteriums angeordnet. Im Falle des EDF-Lotsen bedeutet dies, daß die Prozeßinstanzen nach monoton steigender Frist sortiert werden. Bei ERF erfolgt die Sortierung nach steigender Bereitzeit und bei RM nach steigender Periodendauer des ursprünglichen periodischen Prozesses.

Auch für die Methoden, die für eine Prozeßinstanz verwendbar sind, wird ein Sortierkriterium spezifiziert. Ein Beispiel ist das Greedy-Kriterium (GREEDY), bei dem die Methoden und Leistungsstufen nach monoton steigender worst-case Ausführungszeit sortiert werden. Während der Optimierung wird aus dem geordneten Vektor die am weitesten vorne stehende Prozeßinstanz vorgeschlagen, deren Datenabhängigkeiten erfüllt sind. Als Methode wird die Methode mit der längsten worst-case Ausführungszeit vorgeschlagen, deren Ausführung auch im schlimmsten Fall noch vor der Frist der Instanz beendet ist. Eine andere Strategie ist das Vorschlagen der Kombinationen nach ansteigender worst-case Ausführungszeit (ASC_T). Der Vorteil dieser zweiten Strategie ist, daß auch dann schnell Lösungen gefunden werden, wenn die Fristen sehr knapp sind. Die Greedy-Strategie führt in diesem Fall oft zu Backtracking. Tabelle 5.1 zeigt die Verschlagsreihenfolge der Lotsen (EDF, GREEDY) und (EDF, ASC_T) für folgende Ausgangssituation: sechs Millisekunden verstrichen, Prozesse *Kodieren* und *Senden* noch einzuplanen.

Wird dem Lotsen bei seinem Aufruf die zuletzt gewählte Kombination aus Instanz, Methode und Leistungsstufe übergeben und ist er im Modus `EXHAUSIVE`, so versucht er zunächst, ob für diese Instanz eine weitere ausführbare Methode

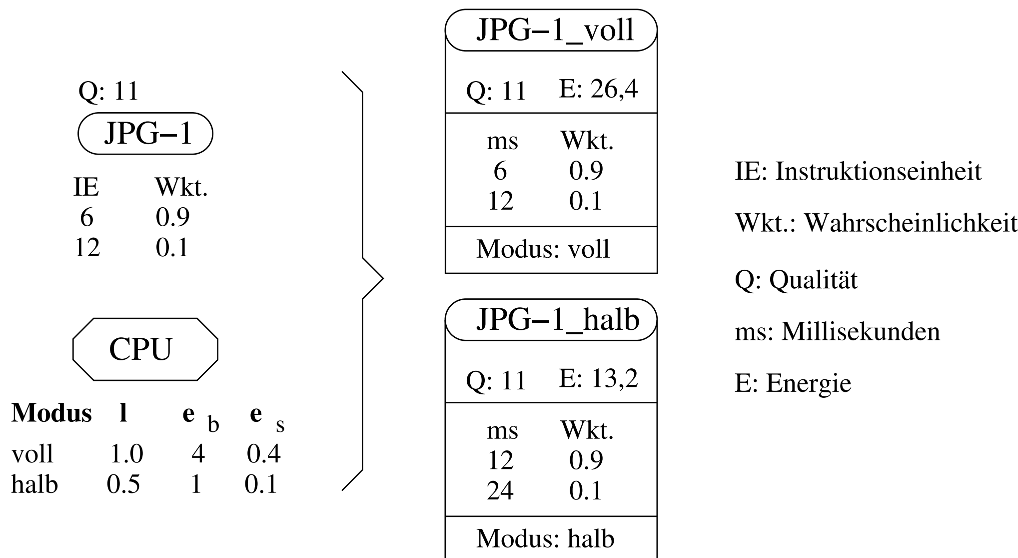

Abbildung 5.3: Kodierung der Leistungsstufen in Methodenkopien

oder eine andere Leistungsstufe existiert. Trifft dies zu, so schlägt er die gleiche Instanz vor, die zuletzt gewählt war, und wählt die Methode oder Leistungsstufe, die der zuletzt gewählten Kombination am ähnlichsten ist und die auch im schlimmsten Fall rechtzeitig vor der Frist beendet ist. Existiert keine solche Kombination, oder ist der Lotse im Modus `INIT`, so schlägt er als nächste Instanz die Instanz vor, deren Datenabhängigkeiten erfüllt sind und die der zuletzt gewählten Instanz am nächsten ist. Die Wahl der Methode erfolgt in diesem Fall auf die gleiche Art wie beim Aufruf ohne zuletzt gewählte Kombination aus Instanz, Methode und Leistungsstufe. Sobald die Kombination bestimmt ist, kann der Lotse implementierte Abbruchbedingungen überprüfen, bevor er das Ergebnis an den Knotenspeicher übergibt. Falls eine der Abbruchbedingungen erfüllt ist, wiederholt der Lotse obige Kombinationsberechnung so lange, bis er eine Kombination findet, die keine der Abbruchbedingungen erfüllt. Findet er keine solche Kombination, so liefert er als Ergebnis `null` zurück, d.h. die Optimierung dieses Schlüssels ist beendet.

In der Implementierung werden die Leistungsstufen in den Methoden kodiert. Für jede Leistungsstufe des Prozessors wird eine Kopie jeder Methode gespeichert, die die Ausführungszeitverteilung und den Energieverbrauch der Methode bei der Ausführung in dieser Leistungsstufe enthält. Abbildung 5.3 zeigt die Vervielfältigung für die Methode *JPG-1* aus dem Beispiel in Abbildung 1.3. Abbildung 5.4 zeigt den Pseudoquelltext für sortierte Lotsen.

```
getProposal(key, instance, method, mode,
            instanceComparator, methodComparator) {

    do {
        tryAgain = false;
        proposalInstance = null;
        proposalMethod = null;
        newSelection = null;

        switch (mode) {

        case EXHAUSTIVE:

            // try to get next method of current instance
            if (method != null) {

                // try to get next method
                proposalMethod = instance.getMethodAfter(method);
            }
            /* fall through! */

        case INIT:

            // get next instance and first method
            if (proposalMethod == null) {

                // get instance ready to schedule with earliest deadline
                ArrayList sortedReadyInstances =
                    key.getSortedReadyInstances(instanceComparator);
                proposalInstance =
                      sortedReadyInstances.getInstanceAfter(Instance);
                proposalMethod =
                    proposalInstance.getFirstMethod(methodComparator);
            }
        }

        // check proposal for feasibility and
        // superiority potential
        boolean maybeBetter = estimate(key,
                    proposalInstance, proposalMethod);

        if (!maybeBetter) {
            tryAgain = true;
            instance = proposalInstance;
            method = proposalMethod;
        }
    } while (tryAgain);

    return (proposalInstance,proposalMethod);
}
```

Abbildung 5.4: Pseudoquelltext der sortierten Lotsen

Kombination	Zeitbedarf
(Kodieren, JPG-2, voll)	18 ms
(Kodieren, JPG-1, halb)	24 ms
(Kodieren, JPG-2, voll)	18 ms
(Kodieren, JPG-1, halb)	12 ms
(Kodieren, JPG-2, voll)	9 ms
(Kodieren, JPG-2, voll)	9 ms
(Kodieren, JPG-1, halb)	24 ms
...	...
(Kodieren, JPG-1, halb)	12 ms
null	-

Tabelle 5.2: Vorschläge des Lotsen (ZUFALL, ZUFALL)

5.1.2 Zufallslotse (MC)

Der Zufallslotse wählt im Modus `EXHAUSTIVE` zufällig eine Instanz, Methode und Leistungsstufe aus und überprüft die Zulässigkeit und die Abbruchbedingungen. Falls die gewählte Kombination unzulässig oder mindestens eine Abbruchbedingung erfüllt ist, wird erneut eine zufällige Kombination gewählt. Tabelle 5.2 zeigt eine mögliche Vorschlagsreihenfolge des Lotsen (ZUFALL, ZUFALL) für die Ausgangssituation: sechs Millisekunden verstrichen, Prozesse *Kodieren* und *Senden* noch einzuplanen.

Abbildung 5.5 zeigt den Pseudoquelltext des Zufallslotsen. Die Anzahl der Versuche kann durch den Wert der Variablen `recursionDepth` beschränkt werden. Im Modus `INIT` wird nur die Instanz zufällig gewählt. Die gewählte Methode ist dann die an erster Stelle des Methodenvektors stehende Methode. Auch im Modus `INIT` ist die Anzahl der Versuche beschränkt.

5.1.3 Dynamische Scheduling-Algorithmen als Lotsen

Neben den oben erwähnten, implementierten Lotsen ist es auch möglich beliebige dynamische, energie- bzw. qualitätsbewußte Schedulingalgorithmen als Lotsen zu verwenden. Dynamische Scheduling-Algorithmen liefern für jeden Schlüssel die als nächstes auszuführende Instanz, Methode und Leistungsstufe (sofern unterstützt). Ein Lotse, der durch einen dynamischen Scheduling-Algorithmus implementiert ist, liefert als initialen Bedingungsgraphen das gleiche Ergebnis, das er auch während der Anwendungsausführung erzeugen würde. Erweitert man den Algorithmus um Backtracking, z.B. durch temporäres Entfernen der bereits gewählten Instanzen, so kann er auch während der Optimierung des Bedingungsgra-

```
public Selection getSelectionProposal(NodeKey key,MRTask task,MRMethod method) {

    if ((scheduler.getConfig().getMaxRetries() != UNLIMITED)
        && (recursionDepth > scheduler.getConfig().getMaxRetriesValue())) {
   return null;
    }

    recursionDepth++;

    MRTask proposalTask = null;
    MRMethod proposalMethod = null;
    Selection newSelection = null;

    // get tasks ready to be scheduled
    readyTasks = key.getSortedReadyMRTasks(null);
    int noOfTasks = readyTasks.size();

    // choose random task
    proposalTask = (MRTask)readyTasks.get(random(noOfTasks));

    // choose method
    switch (mode) {

    case EXHAUSTIVE:
      // choose method randomly
      int noOfMethods = proposalTask.getMethods().size();

      proposalMethod = (MRMethod)proposalTask.getMethods().get(
                                          random(noOfMethods));
      newSelection = new Selection(proposalTask,proposalMethod);
      break;

    case INIT:
      proposalMethod = (MRMethod)proposalTask.getMethods().get(0);
      newSelection = new Selection(proposalTask,proposalMethod);
      break;

    default:
      // error
      throw new RuntimeException("Mode not supported!");
    }

    boolean maybeBetter = forwardCheck(key,newSelection);

    if (!maybeBetter) {
      newSelection = this.getSelectionProposal(key,newSelection.getTask()
                                          ,newSelection.getMethod());
    }

    recursionDepth--;

    return newSelection;
}
```

Abbildung 5.5: Pseudoquelltext des Zufallslotsen

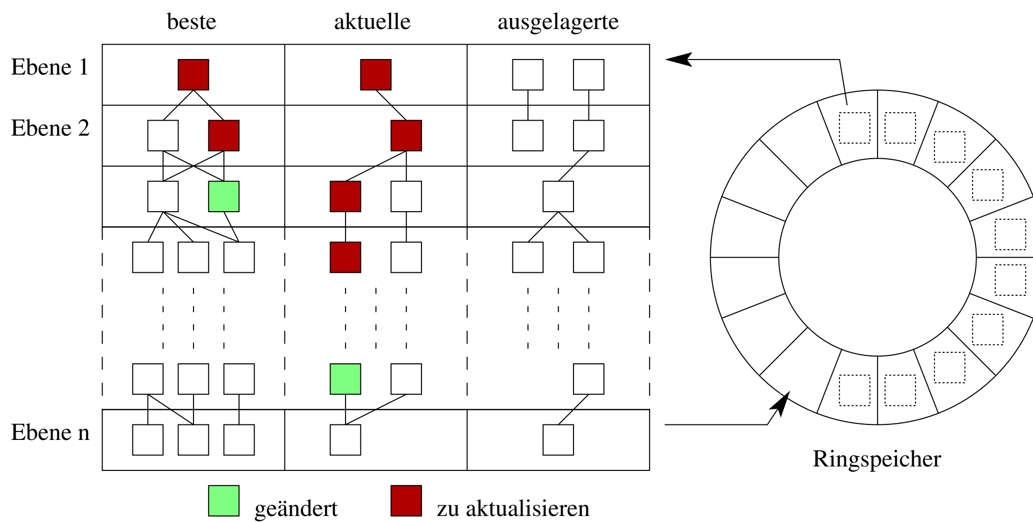

Abbildung 5.6: Aufbau des Knotenspeichers

phen eingesetzt werden.

5.2 Knotenspeicher

Der Knotenspeicher ist eine Datenstruktur, die für die gesamte Verwaltung der Knoten, d.h. ihre Erzeugung, Modifikation, Aktualisierung und Zerstörung, zuständig ist. Er initialisiert fehlende (Teil-)Bedingungsgraphen und speichert die beste gefundene Lösung. Dadurch werden die Optimierer von dieser Aufgabe entlastet und können einfacher umgesetzt werden kann.

5.2.1 Aufbau des Knotenspeichers

Der Knotenspeicher ist in $n = |\text{Instanzen}|$ Ebenen unterteilt (Abbildung 5.6). Die Nummer einer Ebene gibt die Anzahl der Instanzen an, die inklusive dieser Ebene bereits eingeplant sein müssen. Jede Ebene ist in die drei Bereiche *beste*, *aktuelle*, und *ausgelagerte* Knoten unterteilt. Die beste bisher gefundene Lösung wird im ersten Bereich gehalten. Der zweite Bereich enthält die aktuell vom Optimierer betrachtete Lösung. Der dritte Bereich enthält Knoten, die in keiner der beiden Lösungen enthalten sind, die aber vielleicht in einem der folgenden Optimierungsschritte wieder benötigt werden könnten.

Die Knoten werden vor Beginn der Optimierung erzeugt und in einem Ringspeicher abgelegt und verwaltet, solange sie noch nicht initialisiert sind. Sobald

ein Knoten zur Initialisierung angefordert wird, wird er an der durch den Entnahmezeiger markierten Stelle entnommen. Knoten die nicht mehr benötigt werden, werden an der durch den Einfügezeiger markierten Stelle wieder in den Ringspeicher eingefügt. Die Anzahl der im Ringspeicher zu Beginn der Optimierung verfügbaren Knoten kann konfiguriert werden. Je mehr Knoten zur Verfügung stehen, desto schneller wird die Optimierung in der Regel voranschreiten. Die Anzahl der Knoten sollte jedoch so gewählt werden, daß der durch die Knoten belegte Speicher kleiner ist, als der physikalisch verfügbare Hauptspeicher.[1] Sobald die Optimierung beendet ist, werden alle Knoten wieder gelöscht.

5.2.2 Konsistenzsicherung im Knotenspeicher

Der Knotenspeicher muß die Konsistenz der nach außen sichtbaren Knoten sicherstellen, d.h. der Zielfunktionswert dieser Knoten muß dem Wert entsprechen, der sich aus der rekursiven Berechnungsvorschrift ergibt. Sobald also ein direkter oder indirekter Sohn eines Knotens geändert wird, oder sich dessen Zielfunktionswert ändert, muß der Vater aktualisiert werden, bevor er nach außen zugänglich gemacht werden kann. Aus Leistungsgründen ist es wünschenswert, so wenige Aktualisierungen wie möglich durchzuführen. Daher werden bei einer Knotenanfrage nur diejenigen Knoten aktualisiert die in der gleichen Ebene oder darunter liegen. Dies führt z.B. bei der anforderungsgetriebenen Dynamischen Programmierung zu erheblichen Einsparungen, weil dieses Verfahren die Knoten in der Reihenfolge einer Tiefensuche erzeugt, ändert und optimiert. Daher muß jeder Knoten nur einmal aktualisiert werden, weil er erst nach Abschluß der Optimierung aller seiner Söhne wieder angefragt wird. Bei Simulated Annealing sind die Einsparungen geringer, da dabei die Knoten in zufällig gewählten Ebenen geändert werden.

Um sicherzustellen, daß im Bereich der besten Knoten tatsächlich immer der beste Knoten für einen Schlüssel abgelegt ist, muß nach jeder Aktualisierung eines aktuellen Knotens überprüft werden, ob dessen Zielfunktionswert besser ist als der des bisher besten Knotens. Falls dies der Fall ist wird der aktuelle Knoten repliziert und das Replikat verdrängt den bisher besten Knoten. Das Replikat erbt dessen Väter, die aufgrund des geänderten Zielfunktionswertes als zu aktualisieren markiert werden müssen. Abbildung 5.7 zeigt ein Beispiel für eine Qualitätsinversion aufgrund der Verbesserung eines Knotens. Vor dem Ersetzen des sowohl im besten als auch im aktuellen Graphen enthaltenen Knotens 4 durch den Knoten 6, ist die Qualität $Q1$ von Knoten 1 höher als die von Knoten 2. Wird

[1] Falls die Anzahl der minimal benötigten Knoten nicht bekannt ist, kann der Ringspeicher während der Optimierung vergrößert und mit zusätzlichen Knoten gefüllt werden, sobald er leer ist. Dies kann allerdings dazu führen, daß Knoten auf den Hintergrundspeicher ausgelagert werden, wodurch sich erhebliche Leistungseinbußen ergeben können.

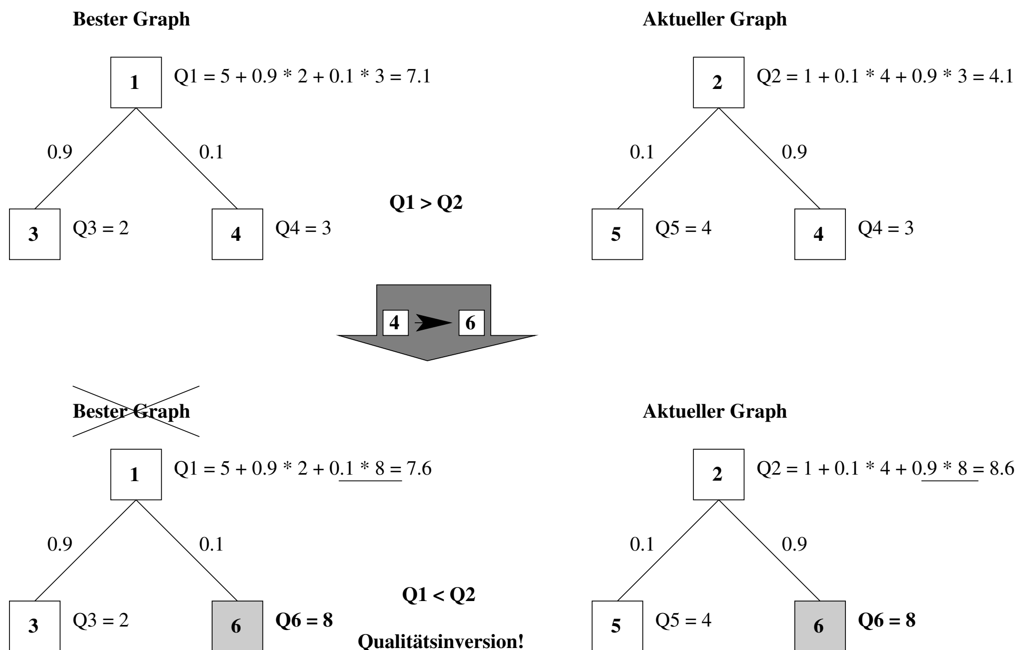

Abbildung 5.7: Qualitätsinversion durch Qualitätsänderung eines Sohnes

Knoten 4 durch Knoten 6 ersetzt, so profitiert Knoten 2 - aufgrund der höheren Gewichtung des ersetzen Sohnes - stärker als Knoten 1. Nach der Aktualisierung ist Knoten 2 besser als Knoten 1 und verdrängt diesen aus dem Bereich der besten Knoten.

Abbildung 5.8 zeigt die fünf Zustände, die von den Knoten während der Optimierung angenommen werden können. In der Initialisierungsphase des Knotenspeichers werden die Knoten erzeugt und gelangen dadurch in den Zustand *kreiert*. Sobald der Optimierer die im Knoten zu setzende Instanz sowie die zu setzende Methode und Leistungsstufe bestimmt hat, wird der Zustand des Knotens zu *initialisiert*, falls die gesetzten Werte zu keiner Fristverletzung führen. Im Falle einer Fristverletzung oder einer ungültigen Kombination werden die Werte nicht gesetzt und der Zustand des Knotens bleibt *kreiert*. Der Knotenspeicher sucht bzw. erzeugt im Erfolgsfall die benötigten Söhne des Knotens. Falls ein Sohn nicht gefunden wird und auch kein zulässiger Sohn erzeugt werden kann, so ist auch der initialisierte Knoten unzulässig und er wird wieder in den Zustand *kreiert* versetzt. Wenn alle Söhne vorhanden sind, wird der Knoten in den Zustand *zu aktualisieren* gesetzt, damit sein Zielfunktionswert entsprechend der Werte der Söhne zu einen späteren Zeitpunkt aktualisiert wird. Während der Optimierung oder aufgrund der Anforderung einer Zwischenlösung kann der Knotenspeicher den Knoten aktualisieren, wodurch sich sein Zustand zu *aktualisiert* ändert. Sobald ein

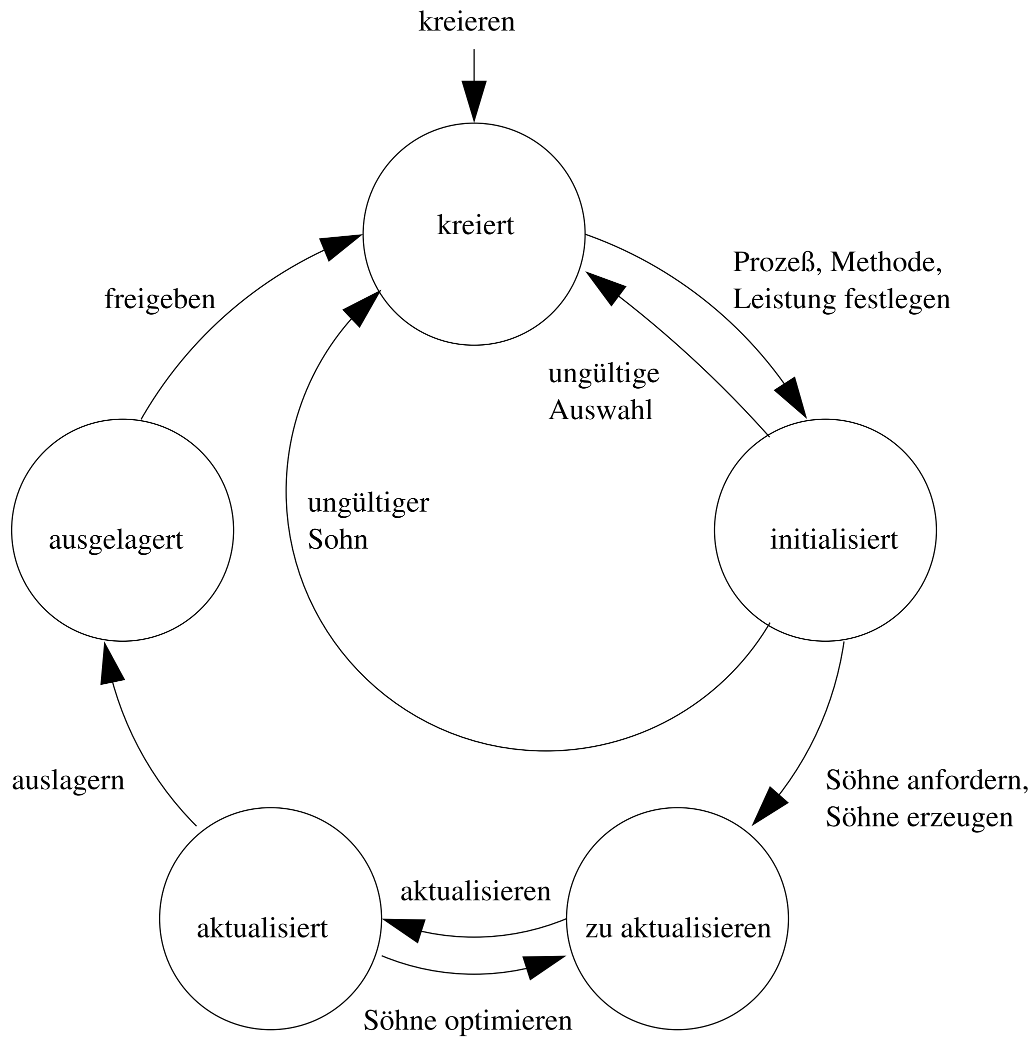

Abbildung 5.8: Knotenzustände

direkter oder indirekter Sohn des Knotens geändert oder aktualisiert wird, wird der Knoten wieder in den Zustand *zu aktualisieren* versetzt, damit der Zielfunktionswert an die Änderung angepaßt wird. Im Laufe der Optimierung werden oft Knoten erzeugt, die besser sind, als schon vorhandene Knoten mit dem gleichen Schlüssel. In diesem Fall wird der schlechtere Knoten nicht gelöscht, aber ausgelagert. In diesem Zustand *ausgelagert* führen Änderungen in Söhnen, die sich noch im aktiven Teil des Knotenspeichers befinden, nicht mehr zu Aktualisierungen des Knotens, aber der letzte eingetragene Zielfunktionswert und die gewählte Kombination aus Instanz, Methode und Leistungsstufe bleiben gespeichert. Dadurch kann bei einem späteren, erneuten Versuch, diese Kombination zu wählen, schnell festgestellt werden, ob sie zulässig ist. Sind alle dem Knotenspeicher zur Verfügung stehenden Knoten in Verwendung, so werden die ausgelagerten Knoten freigegeben, d.h. der Zielfunktionswert und die gewählte Kombination werden gelöscht. Die so freigegebenen Knoten besitzen dann wieder den Status *kreiert* und können erneut initialisiert werden.

Sobald ein Knoten erzeugt wird, dessen Schlüssel in keinem besten Knoten enthalten ist, oder falls der Wert des neuen Knotens besser als der des bisher besten Knotens ist, wird der neue Knoten in den Bereich der besten Knoten kopiert. Dabei verdrängt er den eventuell bereits dort gespeicherten Knoten und erbt dessen Väter. Diese werden als aktualisierungsbedürftig markiert, da ihr Zielfunktionswert angepaßt werden muß. Um unnötige Aktualisierungen zu vermeiden, werden diese solange verzögert, bis ein Knoten aus der Ebene der Väter oder einer höheren Ebene angefordert oder geändert wird.

Abbildung 5.9 zeigt den Verlauf der Knotenspeicheraktionen für eine Reihe von Knotenänderungen. Zuerst wird ein Knoten in der dritten Ebene neu erzeugt und, da er und alle seine Söhne zulässig sind, in den Bereich der aktuellen Knoten eingeordnet. Sein Vater wird als zu aktualisieren markiert. Bei der Berechnung des Zielfunktionswertes des neuen Knotens stellt der Knotenspeicher fest, daß der neue Knoten besser als der bisherige beste Knoten für diesen Schlüssel ist. Daher repliziert er den neuen Knoten und fügt ihn anstelle der bisherigen besten Knotens in den Bereich der besten Knoten ein. Der replizierte Knoten erbt dessen Väter, die als zu aktualisieren markiert werden. Nun wird ein Knoten in der vorletzten Ebene geändert. Da alle darunterliegenden Knoten aktualisiert sind, kann die Änderung ohne weiteren Aufwand durchgeführt werden. Die Väter des neuen Knotens werden wiederum als zu aktualisieren markiert. Die darauf folgende Änderung betrifft einen Knoten in der dritten Ebene. Da es zu aktualisierende Knoten in darunterliegenden Ebenen gibt, werden diese erst vom Knotenspeicher aktualisiert und anschließend wird die Änderung durchgeführt.

Abbildung 5.10 zeigt den Pseudoquelltext der zentralen Funktionen des Knotenspeichers. Die Methode `getBestNode(NodeKey key)` liefert den besten bisher gefundenen Knoten für den Schlüssel `key`. Nach der Bestimmung der Ebe-

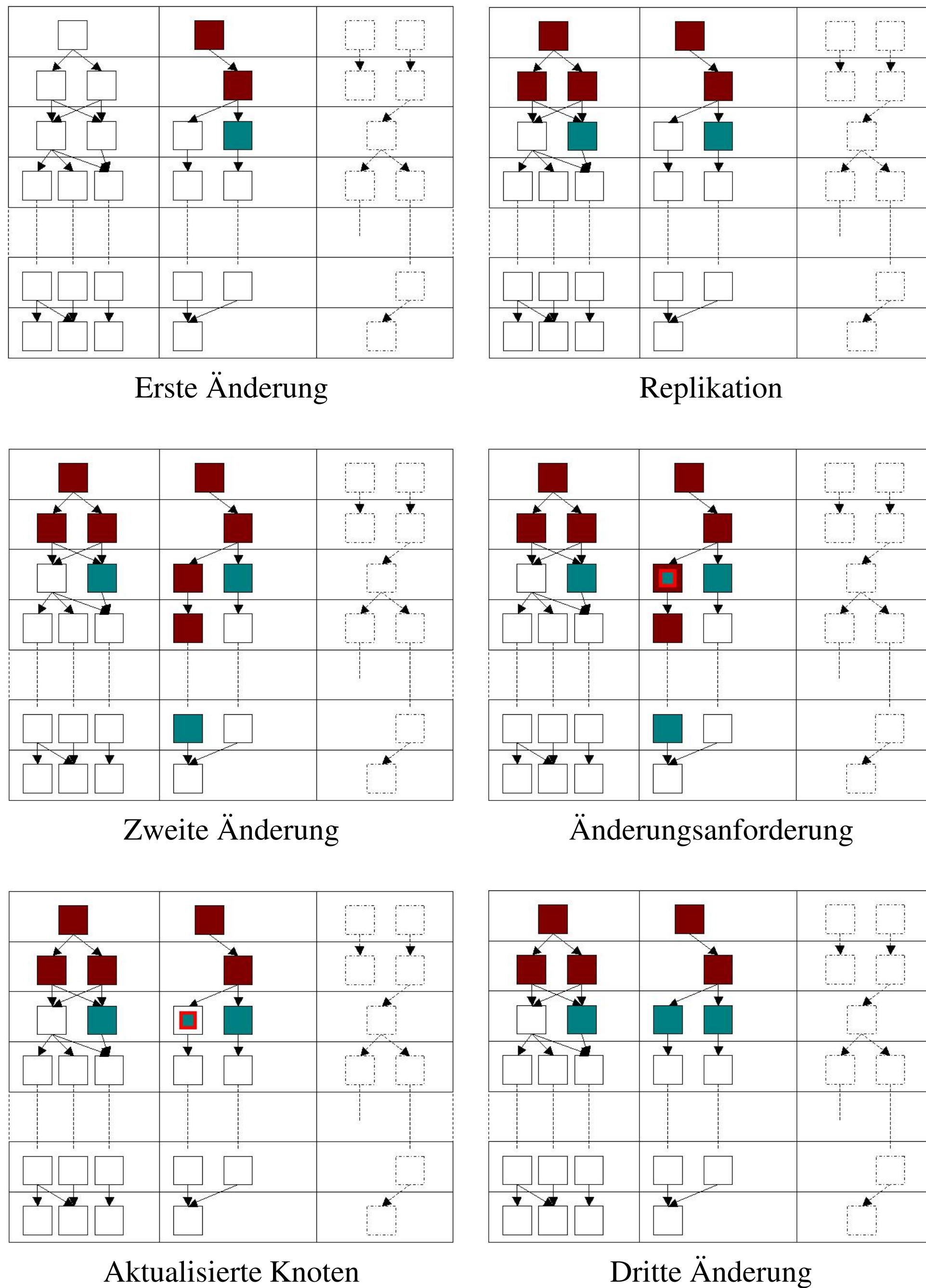

Abbildung 5.9: Änderung mehrerer Knoten und verzögerte Aktualisierung

```
public Node getBestNode(NodeKey key) {

  // get level of desired node
  Node bestNode = null;
  Level currentLevel = null;

  int levelIndex = levels.size() - key.getTasks().cardinality();
  currentLevel = (Level)levels.elementAt(levelIndex);

  // update outdated levels below levelIndex (highest level is level 0!)
  updateLevelsTo(levelIndex);

  // get node from level
  bestNode = currentLevel.getBestNode(key,null,null);
  return bestNode;
}


public Node getNode(NodeKey key) {
  return getNode(key, null, null);
}


public Node getNode(NodeKey key, MRTask task, MRMethod method) {

  // get level of desired node
  Node currentNode = null;
  int levelIndex = levels.size() - key.getTasks().cardinality();
  Level currentLevel = null;
  currentLevel = (Level)levels.elementAt(levelIndex);

  // update outdated levels below levelIndex (highest level is level 0!)
  updateLevelsTo(levelIndex);

  // get node from level
  currentNode = currentLevel.getCurrentNode(key, task, method);

  return currentNode;
}
```

Abbildung 5.10: Pseudoquelltext der zentralen Knotenspeichermethoden

ne, in der der Knoten gespeichert sein muß, wird der Knotenspeicher bis zu dieser Ebene aktualisiert, und anschließend wird der Knoten aus der entsprechenden Ebene angefordert. Falls es keinen solchen Knoten gibt, versucht der Knotenspeicher diesen zu initialisieren und gibt dann den neunen Knoten zurück. Schlägt auch die Initialisierung fehl, so wird der Wert `null` zurückgeliefert. Das gleiche Verhalten besitzt die Methode `getNode(NodeKey key)`, mit dem Unterschied, daß hier der aktuell vom Optimierer betrachtete Knoten für den Schlüssel gesucht und gegebenenfalls initialisiert wird. In der Implementierung wird dazu lediglich die Methode `getNode(NodeKey key, MRTask task, MRMethod method)` mit dem Wert `null` für die Parameter `task` und `method` aufgerufen. Ist für den Parameter `method` und eventuell den Parameter `task` der Wert `null` angegeben, so wird bei der Suche bzw. Initialisierung die Methode bzw. Instanz-Methode-Kombination verwendet, die der Initialisierungslotse für den übergebenen Schlüssel vorschlägt. Wird kein Knoten mit dieser Instanz-Methode-Kombination gefunden, und kann er auch nicht erzeugt werden, so wird ebenfalls der Wert `null` zurückgegeben.

5.2.3 Monitoring

Eine weitere wichtige Aufgabe des Knotenspeichers ist das Erzeugen von Monitorereignissen zur Bewertung des Optimierungsfortschritts. Monitoringereignisse können nach jeweils einer konfigurierbaren Anzahl von erzeugten/geänderten Knoten oder in periodischen Zeitabständen erzeugt werden. Um den exakten Zielfunktionswert der Wurzelknoten des besten und des aktuellen Graphen zu ermitteln, müssen alle Knoten aktualisiert werden. Aus diesem Grund beeinflußt zu häufiges Erzeugen von Monitorereignissen sowohl die Konvergenzeigenschaften der Optimierungsalgorithmen (Anzahl der Aktualisierungen) als auch die Optimierungsdauer (Zeit für Aktualisierungen).

Falls der Monitorwert nicht aktuell sein muß, kann auch eine Schnellauswertung der Knoten vorgenommen werden. In diesem Fall werden die Knoten nicht aktualisiert, sondern es wird lediglich rekursiv beginnend mit den (möglicherweise veralteten) Wurzelknoten der Wert der Zielfunktion berechnet und ausgegeben. Diese Vorgehensweise verhindert, daß durch das Monitoring die Anzahl der Aktualisierungen verändert wird, aber es kann vorkommen, daß veraltete oder, falls auf ungültige Söhne zugegriffen wird, keine Zielfunktionswerte zurückgegeben werden.

5.3 Optimierer

Da die Aufzählung der möglichen Schedules durch die Lotsen geschieht und die Verwaltung und Initialisierung der Knoten vom Knotenspeicher übernommen werden, müssen die Optimierer nur bestimmen, welche Knoten variiert werden sollen und in welcher Reihenfolge.

5.3.1 Anforderungsgetriebene Dynamische Programmierung

Bei der anforderungsgetriebenen Dynamischen Programmierung wird der Suchraum systematisch durchschritten. Für die Optimierung kann jeder vollständige, deterministische Lotse in Kombination mit dem Knotenspeicher eingesetzt werden. Zu Beginn der Optimierung fordert der Optimierer einen Wurzelknoten vom Knotenspeicher an. Diese Anforderung löst die Mikroschritte aus, die zum Erzeugen eines initialen Bedingungsgraphen nötig sind. Falls der Knotenspeicher auf diese Anfrage `null` zurückliefert ist das Systemmodell nicht einplanbar und die Optimierung wird abgebrochen. Im anderen Fall wird die Optimierung gestartet. Die anforderungsgetriebene Dynamische Programmierung fordert dazu rekursiv ab dem Wurzelknoten alle Söhne in Tiefensuchreihenfolge an. Wird bei dieser Anforderungsfolge ein Knoten erreicht, dessen Söhne bereits alle als optimiert markiert sind, wird der Lotse aufgerufen, der eine neue Kombination aus Prozeß, Methode und Leistungsstufe liefert, die im Knoten gesetzt werden soll. Das Setzen der Kombination führt im Knotenspeicher zur Erzeugung der fehlenden Teilbäume des geänderten Knotens. Alle noch nicht als optimiert markierten Söhne des Knotens werden wiederum rekursiv optimiert. Wenn alle möglichen Kombinationen im betrachteten Knoten gesetzt wurden, wird er als optimiert markiert.

Die Umsetzung der anforderungsgetriebenen Dynamischen Programmierung reduziert durch die spezielle Anforderungs- und Änderungsreihenfolge der Knoten die Anzahl der Knotenaktualisierungen im Zusammenspiel mit dem Knotenspeicher auf ein Minimum. Sie fordert immer erst alle möglichen Söhne eines Knotens an und optimiert diese, bevor der Knoten selbst ausgewertet wird. Daher muß jeder Knoten nur einmal nach Beendigung der Optimierung seiner Söhne aktualisiert werden, wenn der Knotenspeicher genügend Knoten zur Verfügung hat. Falls der Knotenspeicher während der Optimierung bereits optimierte Knoten löschen muß, müssen diese natürlich bei Bedarf neu erzeugt und optimiert werden.

Abbildung 5.11 zeigt den Pseudocode für die Initialisierung und die iterative Optimierung des Wurzelknotens. Dazu wird zunächst ein Knoten mit dem Schlüssel des Wurzelknotens angefordert, um den Knotenspeicher zu initialisieren. Falls der Wert `null` zurückgegeben wird, konnte der Knotenspeicher keinen

zulässigen Wurzelknoten erzeugen, d.h. das Systemmodell ist nicht einplanbar, und die Optimierung wird abgebrochen. Wenn ein Wurzelknoten zurückgegeben wurde, wird der Lotse initialisiert, und in der Schleife werden alle vom Lotsen gelieferten Instanz-Methode-Leistungsstufen-Kombinationen für den Wurzelknoten an den Knotenspeicher übergeben. Für jede Kombination, die der Knotenspeicher initialisieren kann (Rückgabewert ungleich `null`) wird die rekursive Optimierung seiner Söhne, die in Abbildung 5.12 dargestellt ist, aufgerufen. In der äußeren Schleife werden alle Söhne des Knotens abgearbeitet. Jeder Sohn der im Knotenspeicher noch nicht als optimiert gekennzeichnet ist, wird in der inneren Schleife optimiert, indem alle für ihn möglichen Instanz-Methode-Leistungsstufe-Kombinationen an den Knotenspeicher übergeben werden und seine Söhne durch den Rekursionsaufruf optimiert werden. Nach der Optimierung eines Knotens wird er als optimiert markiert, damit er bei einer zweiten Referenzierung nicht erneut optimiert wird. Abbildung 5.13 enthält den Kode für das Akzeptieren von neuen/geänderten Knoten. Im Falle der anforderungsgetriebenen Dynamischen Programmierung muß jeder zulässige Knoten akzeptiert werden, um den gesamten Suchraum abzudecken.

5.3.2 Simulated Annealing

Die Zielsetzung des Simulated Annealing-Ansatzes unterscheidet sich von der der anforderungsgetriebenen Dynamischen Programmierung. Bei Simulated Annealing wird der Suchraum durch zufällige Schritte innerhalb einer auf ihm definierten Nachbarschaft durchwandert. Ein zufälliger Schritt von einem Bedingungsgraphen zu einem anderen wird dabei immer ausgeführt, wenn der neue Graph besser ist als der alte, oder mit der Wahrscheinlichkeit

$$1 - e^{\frac{-\Delta(w)}{t}} ,$$

falls der neue Graph schlechter als der alte ist. $\Delta(w)$ bezeichnet dabei den Wert der Verschlechterung. Die Wahrscheinlichkeit, einen schlechteren Graphen zu akzeptieren ändert sich während der Optimierung. Sie wird durch die Parameter t_{start}, k, f und t_{stop} kontrolliert. t_{start} ist die Starttemperatur. Je höher die Temperatur t ist desto höher ist die Wahrscheinlichkeit, daß ein schlechterer Graph akzeptiert wird. k gibt die Anzahl der Schritte an, während derer die Temperatur konstant gehalten wird, und f bestimmt, um welchen Faktor sie nach jeweils k Schritten verringert wird. Der letzte Parameter t_{stop} bestimmt schließlich, bei welcher Temperatur die Optimierung beendet werden soll.

Das Akzeptieren jedes Verbesserungsschrittes bewirkt, daß man zu immer besseren Lösungen (lokales Optimum) kommt, während das gelegentliche Zulassen von Verschlechterungen es ermöglicht, lokale Optima wieder zu verlassen, um ein globales Optimum erreichen zu können.

```
private boolean calculateSchedule() {

  useGuide(INIT_GUIDE);
  Node rootNode = nodepool.getNode(rootKey);

  if (rootNode == null) {
    // no initial configuration found -> taskgraph not schedulable
    return false;
  }

  // examine all possible tasks to get the optimal solution
  MRTask task = null;
  MRMethod method = null;

  // set normal guide
  useGuide(STANDARD_GUIDE);

  // check guide properties
  String errStr = guide.hasRequiredProperties(fullCoverage,deterministic);
  if (errStr != null) {
    return false;
  }

  // examine all possible tasks  and methods to get the optimal solution
  Selection nextSel = new Selection(rootNode.getActiveTask(),
                                  rootNode.getActiveMethod());

  while (nextSel != null) {

    task = nextSel.getTask();
    method = nextSel.getMethod();

    // get node and optimize it
    useGuide(INIT_GUIDE);
    rootNode = nodepool.getNode(rootKey, task, method);
    useGuide(STANDARD_GUIDE);

    // optimize root node
    if (rootNode != null) {
      optimize(rootNode);
      rootNode = nodepool.getNode(rootKey, task, method);
    }

    nextSel = guide.getSelectionProposal(rootKey,task,method);
  }

  rootNode = nodepool.getBestNode(rootKey);
  resultDAG = new DecisionDAG(rootNode,null);
  return true;
}
```

Abbildung 5.11: Initialisierung und Iteration über alle möglichen Startkombinationen

```
private void optimize(Node node) {

  Vector sons = node.getSons();
  if (sons == null) {
    // node is leaf and thus has no sons
    return;
  }

  // optimize all sons recursively
  for (int index = 0; index < sons.size(); index++) {
    Node currentSon = (Node) sons.get(index);
    NodeKey currentKey = currentSon.getKey();
    MRTask task = null;
    MRMethod method = null;
    Node currentSonAlternative = null;

    // examine all possible tasks and methods to get the optimal solution
    Selection nextSel = new Selection(currentSon.getActiveTask(),
                                  currentSon.getActiveMethod());

    if (nodepool.keyIsOptimized(currentKey) == false) {

      // son needs to be optimized
      while (nextSel != null) {

        task = nextSel.getTask();
        method = nextSel.getMethod();

        // get node and optimize it
        useGuide(INIT_GUIDE);
        currentSonAlternative = nodepool.getNode(currentKey, task, method);
        useGuide(STANDARD_GUIDE);

        // optimize son
        if (currentSonAlternative != null) {
          optimize(currentSonAlternative);
        }

        nextSel = guide.getSelectionProposal(currentKey,task,method);
      }

      // mark currentKey as optimized
      nodepool.addOptimzedKey(currentKey);
    }
  } // end son loop

  // get node again to update quality
  if ((node.getStatus() != Node.BEST_NODE)
      && (node.getStatus() != Node.CURRENT_NODE)) {
    node = nodepool.getNode(node.getKey(),node.getActiveTask(),
                                      node.getActiveMethod());
  }

  return;
}
```

Abbildung 5.12: Rekursive Optimierung der Knoten

```
public Node acceptNode(Node newNode, Node oldNode) {

  if (newNode.getValue() != Node.INFEASIBLE) {
    return newNode;
  }
  return oldNode;
}
```

Abbildung 5.13: Jeder zulässige Knoten wird akzeptiert.

Abbildung 5.14 zeigt den Pseudocode des Simulated Annealing-Algorithmus. Die äußere Schleife kontrolliert den Verlauf der Temperatur mittels der Funktion in Abbildung 5.15 und die innere Schleife führt alle Schritte aus, die bei konstanter Temperatur durchgeführt werden. Jeder Schritt beginnt mit der zufälligen Auswahl eines Opferknotens durch die Funktion aus Abbildung 5.16, dessen Instanz-Methode-Leistungsstufe-Kombination geändert werden soll. Anschließend wird durch den Zufallslotsen zufällig die neu zu setzende Kombination bestimmt und, falls der neue Knoten zulässig ist, wird mit der Funktion aus Abbildung 5.17 bestimmt, ob er beibehalten wird. Knoten die besser sind als der letzte für denselben Schlüssel betrachtete Knoten werden immer akzeptiert, während Knoten, die eine Verschlechterung bewirken, nur mit einer gewissen Wahrscheinlichkeit, die von der Größe der Verschlechterung und der aktuellen Temperatur abhängt, akzeptiert werden.

Die hier vorgestellte Simulated Annealing bewertet im Gegensatz zum klassischen Simulated Annealing nicht komplette Lösungen, sondern trifft die Entscheidungen jeweils für Teillösungen. Diese Vorgehensweise ist durch den monotonen Zusammenhang der Güte der Lösung und der Güte der in ihr enthaltenen Teillösungen möglich, denn jede Verbesserung einer Teillösung stellt eine Verbesserung der Gesamtlösung dar. Die Beschränkung auf die Bewertung von Teillösungen reduziert die für einen Optimierungsschritt nötige Zeit erheblich, da im Falle des klassischen Simulated Annealing immer alle Knoten der Lösung aktualisiert werden müssen, um einen Schritt zu bewerten. Der automatische Abgleich der besten bisher gefundenen Lösung mit den neuen Teillösungen beschleunigt die Konvergenz des Verfahrens zusätzlich, weil auch die Güte der besten Lösung steigt, wenn sie das in der aktuellen Lösung verbesserte Teilproblem enthält. D.h. die Güte der besten Lösung steigt, obwohl die Güte der neuen, aktuellen Lösung immer noch weit schlechter ist als die der besten Lösung.

```
private boolean calculateSchedule() {

  // read the preferences
  temperature = ((SimAnnealSchedulerParams)params).getInitTemperature();
  stopTemperature = ((SimAnnealSchedulerParams)params).getStopTemperature();
  coolControl = ((SimAnnealSchedulerParams)params).getCoolControl();
  constTempSteps = ((SimAnnealSchedulerParams)params).getConstTempSteps();

  this.steps = 0;

  // get rootnode (this generates an initial configuration,too)
  useGuide(INIT_GUIDE);
  Node rootNode = nodepool.getNode(rootKey);

  if (rootNode == null) {
    return false;
  }

  // apply simulated annealing to get a better solution
  do { // until stop-criterium is reached
    do {// until equilibrium is reached

      // get victim node
      Node toChange = chooseVictim();

      // "perturb" the node
      useGuide(STANDARD_GUIDE);
      Selection newChoice = getSelectionProposal(toChange.getKey(),
                                                 toChange.getActiveTask(),
                                                 toChange.getActiveMethod());

      if (newChoice != null) {
        // generate the new node
        useGuide(INIT_GUIDE);
        nodepool.getNode(toChange.getKey(),newChoice.getTask(),
                                          newChoice.getMethod());
      } else {
        steps--;
      }

    } while (( (steps++) < this.constTempSteps )&& (!isTimeUp)); // keep temp?

    // update parameters
    updateControlParameters(); // cool down
  } while ( this.temperature > this.stopTemperature );  // stop?

  rootNode = nodepool.getBestNode(rootKey);
  resultDAG = new DecisionDAG(rootNode,null);
  return true;
}
```

Abbildung 5.14: Pseudoquelltext des Simulated Annealing-Algorithmus

```
private void updateControlParameters() {
  steps = 0;
  temperature = temperature * coolControl;
}
```

Abbildung 5.15: Pseudoquelltext der Temperaturanpassung

```
private Node chooseVictim() {

  if (victimRootNode == null) {
    victimRootNode = nodepool.getNode(rootKey);
  }

  Node victim = victimRootNode;

  // choose level of change
  int rootLevel = victimRootNode.myLevelNumber;
  int deepestLevel = nodepool.getMRTasks().size() - 1;
  int noOfLevels = deepestLevel - rootLevel +1;

  // same probability for all levels
  int randLevelNo = randGen.nextInt(noOfLevels);

  for (int i = 0; i < randLevelNo; i++){
    Vector ewSons = victim.getExistingWeightedSons();

    // randomly choose a son
    int noOfVictim = randGen.nextInt(ewSons.size());
    Iterator sonIter = ewSons.iterator();

    for ( int j = 0; j <= noOfVictim; j++ ) {
      victim = ((WeightedLink)sonIter.next()).getTarget();
    }
  }

  // force update up to victim's level
  nodepool.getNode(victim.getKey());
  return victim;
}
```

Abbildung 5.16: Pseudoquelltext der Auswahl des Opferknotens

```
public Node acceptNode(Node newNode, Node oldNode) {

  if (oldNode == null) {
    return newNode;
  }

  // calculate gain of new node
  double valueDiff = nodepool.getTargetFunction().gain(oldNode.getValue(),
                                                       newNode.getValue());

  // accept improvements
  if (valueDiff >= 0.0) {
    return newNode;
  }

  // probabilistic acceptance of degradation
  if ((Math.exp((-1.0 * valueDiff) / temperature) > (randGen.nextDouble()))
      && (newNode.getValue() != Node.INFEASIBLE)) {
    return newNode;
  } else {
    return oldNode;
  }
}
```

Abbildung 5.17: Pseudoquelltext für Akzeptanz von Knoten

5.4 Abschließende Bemerkungen zur strukturellen Umsetzung der Optimierer

Die beiden Optimierungsverfahren wurden in drei Komponenten unterteilt, die unterschiedliche Aufgabenstellungen erfüllen:

- Die Komponente *Lotse* dient zur Aufzählung der Instanz-Methode-Leistungsstufe-Kombinationen, die in einer durch die verstrichene Zeit und die verbleibende Instanzmenge spezifizierten Situation ausführbar sind. Als Lotsen können einfache Aufzählalgorithmen, Zufallsgeneratoren oder aber auch dynamische Scheduler, die um Backtracking erweitert wurden und das vorliegende Systemmodell behandeln können, eingesetzt werden.

- Die zweite Komponente ist der *Knotenspeicher*, der die gesamte Verwaltung der gefundenen (Zwischen-)Lösungen übernimmt. Er generiert Knoten, initialisiert sie und aktualisiert ihre Zielfunktionswerte, wenn dies nötig ist. Zusätzlich stellt er die Anytime-Funktionalität zur Verfügung, indem er die beste gefundene Lösung speichert. Beim Speichern der besten Lösung überprüft er auch jeweils, ob eine in der aktuellen Lösung enthaltene Teillösung besser ist, als die in der besten Lösung für dieselbe Situation gespeicherte Lösung. Ist dies der Fall, so setzt er die bessere Teillösung in die beste Lösung ein und aktualisiert sie. Durch diese Vorgehensweise kann die ge-

speicherte beste Lösung besser sein, als alle bis dahin gefundenen Lösungen, wodurch insbesondere die Konvergenzgeschwindigkeit bei der Verwendung von Simulated Annealing beschleunigt wird.

- Die dritte Komponente ist der *Optimierer*, für den die Verfahren *anforderungsgetriebene Dynamische Programmierung* und *Simulated Annealing* implementiert wurden. Ihre Aufgabe ist es, festzulegen, welcher von den in der aktuellen Lösung enthaltene Knoten als nächstes modifiziert werden soll, um zu einer besseren Lösung zu gelangen.

Die Optimierung erfolgt in einem jederzeit abbrechbaren Zyklus, der mit der Suche nach einer Initiallösung durch den Knotenspeicher beginnt. Anschließend bestimmt der Optimierer, den zu ändernden Knoten, der Lotse berechnet die durchzuführende Änderung, und der Knotenspeicher führt sie schließlich durch. Dieser Zyklus wird durchlaufen, bis die Optimierung abgebrochen wird, eine optimale Lösung gefunden wurde, oder verifiziert wurde, daß keine Lösung existiert. Der Abbruch der Optimierung kann dabei auch automatisch beim Erreichen einer Zeit-, Energie- oder Qualitätsschranke erfolgen. Durch die konfigurierbare Größe des Knotenspeichers und die Wiederverwendung nicht mehr benötigter Knoten ermöglicht das vorgestellte Verfahren eine Abwägung zwischen dem Speicherbedarf und der Dauer der Optimierung. Je mehr Knoten gleichzeitig im Knotenspeicher gehalten werden können, desto schneller ist die Optimierung in der Regel beendet.

Das nächste Kapitel untersucht die Auswirkungen der Modelleigenschaften, wie etwa Anzahl der Methoden je Prozeß, auf die Optimierungsdauer und die erzielbare durchschnittliche Qualität bzw. den erzielbaren durchschnittlichen Energieverbrauch. Weitere Untersuchungen geben Aufschluß über die Auswirkungen der Parameter des Knotenspeichers und des Optimierers, sowie des verwendeten Lotsen auf die Optimierungsdauer und die Anytime-Profile der Optimierungsverfahren.

Kapitel 6

Untersuchungen

Die Untersuchungen in diesem Kapitel geben Aufschluß über verschiedene Eigenschaften der Optimierungsalgorithmen und Systemmodelle (Graphen). Abschnitt 6.1 beschreibt die Vorgehensweise bei der Versuchsdurchführung. Die Ergebnisse über die Änderung des Optimierungsaufwandes bei unterschiedlichen Modelleigenschaften und verschiedenen Parametern der Optimierer befinden sich in Abschnitt 6.2. Die anschließenden Abschnitte 6.3 und 6.4 stellen die Auswirkungen der Modelleigenschaften auf die erzielbare Qualität und den erzielbaren Energieverbrauch dar. Die Leistungsprofile der beiden Varianten des Optimierers sind in Abschnitt 6.5 dargestellt und letzte Abschnitt 6.6 des Kapitels beinhaltet einen Vergleich des in dieser Arbeit vorgestellten Algorithmus mit anderen Algorithmen.

6.1 Versuchsdurchführung

Um die Auswirkungen von Systemeigenschaften auf das erzielbare Ergebnis zu untersuchen, müssen verschiedene Voraussetzungen erfüllt sein:

1. Messung über viele Graphen
2. Verwendung unterschiedlicher Graphtypen (keine Spezialfälle)
3. Vergleichbarkeit innerhalb einer Graphreihe (keine Dominanz durch andere Eigenschaften)
4. Skalierung der Ergebnisse der Graphreihen
5. Mittlung der Ergebnisse aller Graphreihen

Der erste Punkt stellt sicher, daß die erzielten Ergebnisse aussagekräftig sind, während der zweite verhindern soll, daß die Ergebnisse nur Spezialfälle abdecken,

z.B. nur Graphen mit 20 Instanzen. Die Vergleichbarkeit innerhalb einer Graphreihe ist für die isolierte Untersuchung der Auswirkung einer Eigenschaft nötig. Sie verhindert, daß eine andere Eigenschaft die Ergebnisse beeinflußt. Z.B. ist es bei der Untersuchung der mit unterschiedlich vielen Methoden je Instanz erzielbaren Qualität notwendig, andere Grapheigenschaften, wie etwa die Anzahl der Instanzen, konstant zu halten. Um die Ergebnisse vergleichen zu können, müssen sie innerhalb einer Graphreihe skaliert werden. Anschließend sind sie in einem vergleichbaren Wertebereich, und durch Mittelung der Ergebnisse der Graphreihen kann die grundlegende Form der Auswirkung des jeweils untersuchten Parameters ermittelt werden.

Für die folgenden Untersuchungen wurden jeweils 20 bis 90 Graphen mit dem PASCHA-Graphgenerator zufällig erzeugt (Punkte 1 und 2). Anschließend wurde jeder Graph im Hinblick auf die zu untersuchende Eigenschaft modifiziert, z.B. sukzessives Entfernen von Methoden, um eine Graphreihe zu erzeugen. Die Ergebnisse der durch Modifikation eines Graphen entstandenen Graphen sind vergleichbar, da sich die Graphen (möglichst) nur bezüglich dieser einen Eigenschaft unterscheiden (Punkt 3). Die Graphen wurden anschließend mit den Optimierern eingeplant und die Ergebnisse so skaliert, daß der Maximalwert 1 wird (Punkt 4). Für die Auswertung wurden zu den skalierten Ergebnissen der Graphreihen jeweils Minimum, Maximum, Durchschnitt und Standardabweichung[1] für jeden Wert der untersuchten Grapheigenschaft berechnet (Punkt 5). Abbildung 6.1 veranschaulicht den Ablauf der Untersuchungen für Grapheigenschaften.

Für die Untersuchung der Auswirkungen von Parametern des Optimierers wird zunächst eine Menge von Graphen mit dem Graphgenerator erzeugt, und anschließend wird jeder Graph mit den möglichen Werten für den Parameter eingeplant. Der restliche Ablauf gleicht dem bei der Untersuchung der Modelleigenschaften. Abbildung 6.2 veranschaulicht den Ablauf der Untersuchungen für Parameter des Optimierers.

Die Legende in Abbildung 6.3 zeigt die Linientypen für Minimum, Durchschnitt, Maximum und Standardabweichungen.

[1] Aufgrund der automatisierten Auswertung ist die berechnete Standardabweichung auch bei nicht normalverteilten Größen eingezeichnet.

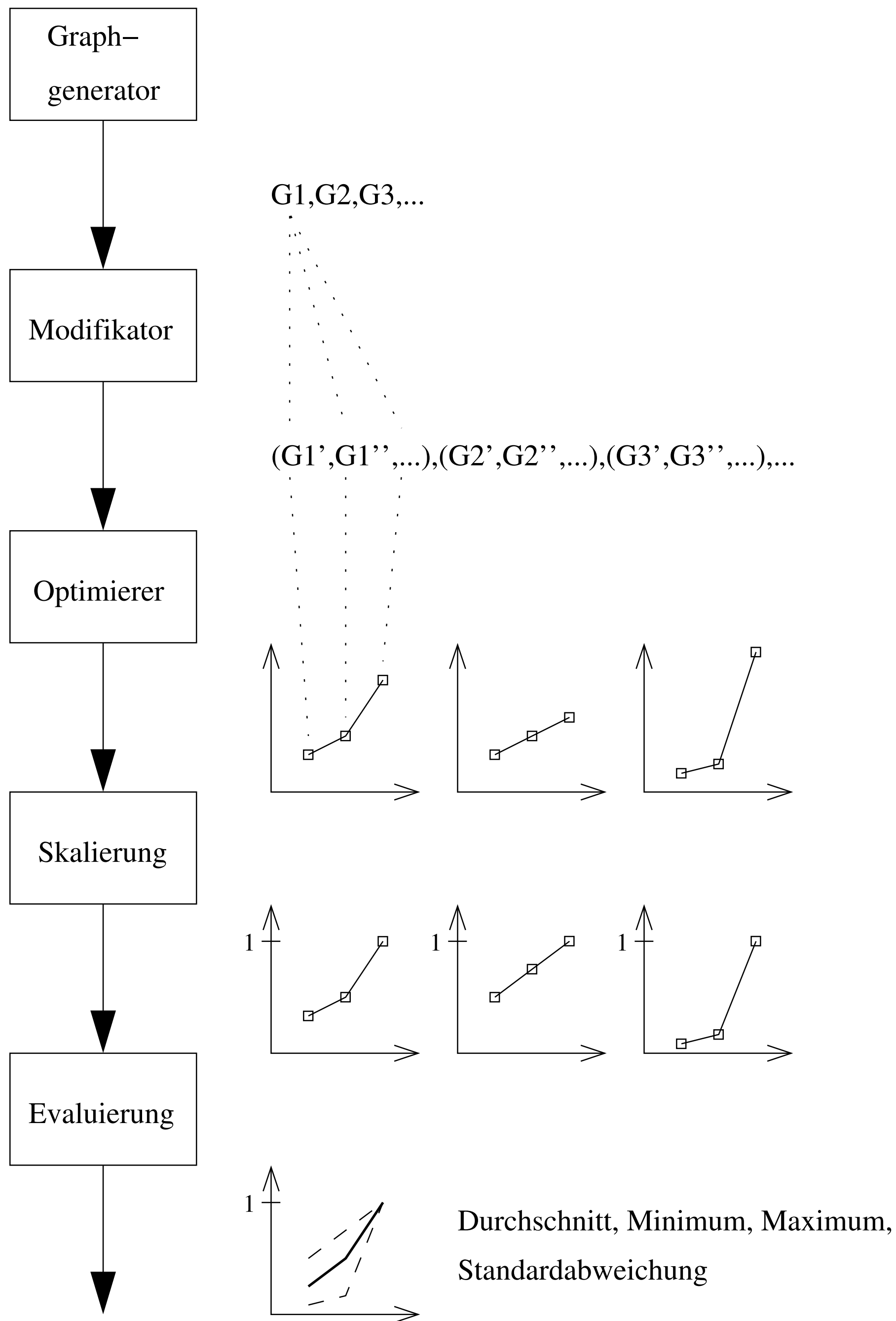

Abbildung 6.1: Ablauf einer Untersuchung für Modelleigenschaften

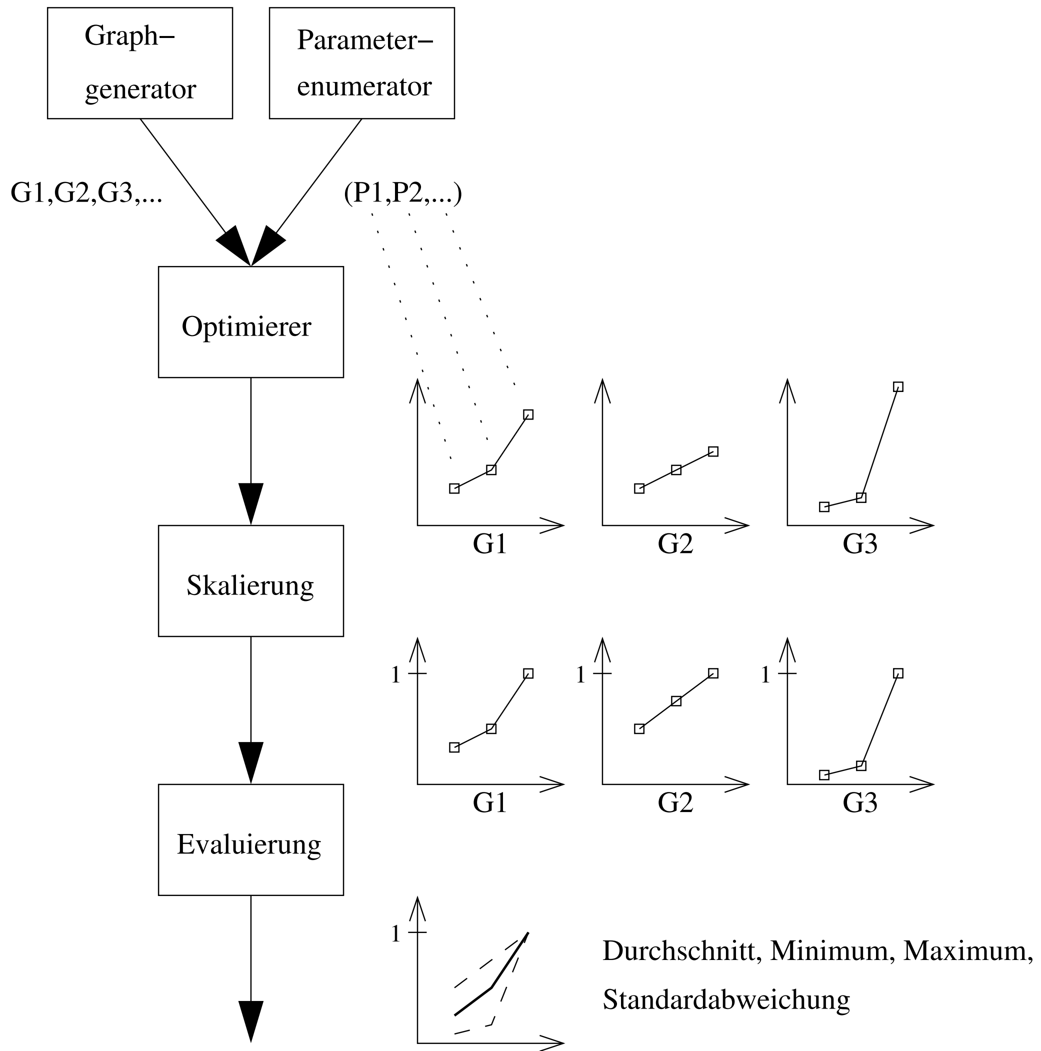

Abbildung 6.2: Ablauf einer Untersuchung für Parameter des Optimierers

Abbildung 6.3: Legende der Graphen

6.2 Optimierungsdauer

Dieser Abschnitt beinhaltet die Auswirkungen von Grapheigenschaften und Parametern des Optimierers auf die Optimierungsdauer. Die durchgeführten Untersuchungen zeigen, wie sich die Dauer der Optimierung verändert, wenn Eigenschaften des Anwendungsgraphen, wie z.B. die Anzahl der Instanzen, oder die Parameter des Optimierers, etwa der verwendete Lotse, verändert werden.

Alle Messungen wurden auf einem AMD Athlon Prozessor mit 800 MHz Taktfrequenz mit der virtuellen Maschine Sun JVM 1.4.2 durchgeführt. Zu Beginn jeder Untersuchung wurden zwei Optimierungen von Graphen durchgeführt, deren Ergebnis nicht in die Meßdaten einfließt, da die Meßergebnisse in den ersten Durchläufen aufgrund des Just-in-time-Compilers der verwendeten JVM stark schwanken können. Aufgrund der hohen Anzahl an Optimierungsläufen (60 bis 900 Läufe je Untersuchung) wurden nur Optimierungen mir einer Dauer kleiner 80 Sekunden berücksichtigt. Insgesamt liegen den folgenden Untersuchungen mehr als 5000 Optimierungsauswertungen zugrunde.

6.2.1 Grapheigenschaften

Dieser Abschnitt zeigt den Einfluß einiger Grapheigenschaften auf die Dauer einer vollständigen[2] Optimierung mit dem Verfahren der anforderungsgetriebenen Dynamischen Programmierung. Als Lotse wurde `EDF/ASC_T` (earliest-deadline first für Instanzen, Methodensortierung nach steigender worst-case Dauer) verwendet und die Überprüfung der Abbruchbedingungen war aktiviert.

Anzahl der Instanzen

Diese Untersuchung zeigt die Abhängigkeit der Optimierungsdauer für Graphen mit unterschiedlichen Instanzzahlen. Die Zahl der Instanzen pro Hyperperiode variiert in Zehnerschritten zwischen 10 und 120. Für jede Instanzzahl wurden 20 Graphen zufällig mit dem Graphgenerator erzeugt. Die Instanzen der Graphen bilden jeweils eine Kette von Datenabhängigkeiten. Jede Instanz besitzt zwei Methoden, die wiederum bis zu zwei Werte für die Ausführungsdauer enthält.

Abbildung 6.4 zeigt die Dauer der Qualitätsoptimierung in Millisekunden in Abhängigkeit von der Instanzzahl. Der erwartete, steiler werdende Anstieg der Optimierungsdauer ist gut zu erkennen. Die Optimierung von Graphen mit bis zu 120 Instanzen ist jedoch mit durchschnittlich ca. drei Sekunden problemlos möglich.

[2] Finden einer optimalen Lösung und Verifikation der Optimalität

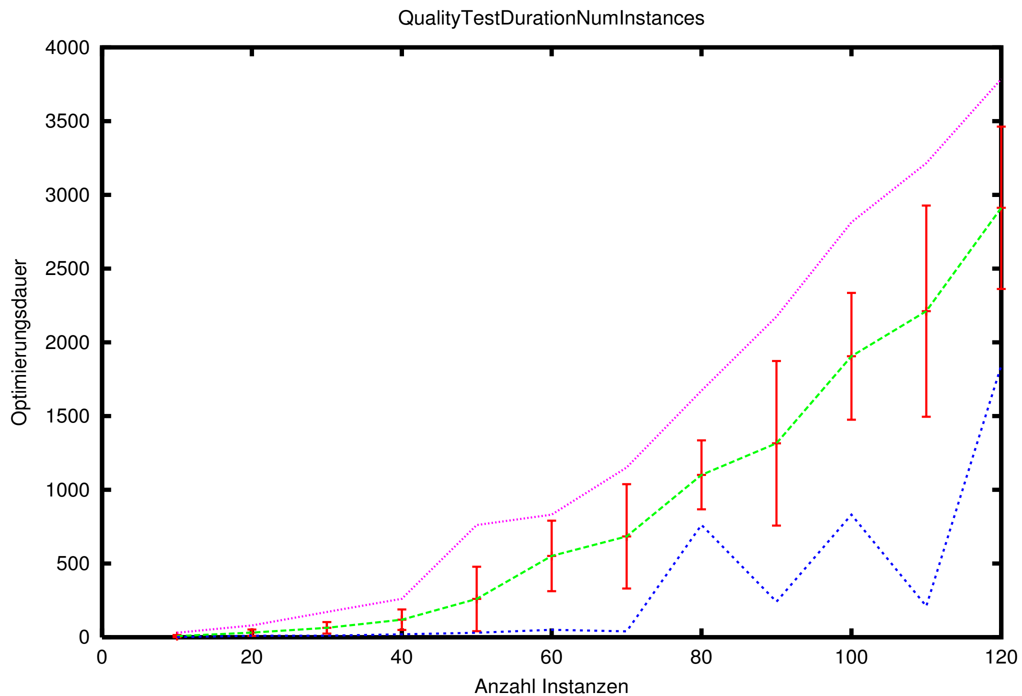

Abbildung 6.4: Anzahl der Instanzen → Optimierungsdauer in Millisekunden

Anzahl der Zusammenhangskomponenten

Diese Untersuchung zeigt die Entwicklung der Dauer der Qualitätsoptimierung für Graphen mit unterschiedlich vielen Zusammenhangskomponenten. Die Zahl der Zusammenhangskomponenten nimmt die Werte 1, 2, 4 und 6 an. Mit dem Graphgenerator wurden 20 Graphen mit jeweils 40 Instanzen zufällig erzeugt, die jeweils aus einer Kette von datenabhängigen Instanzen bestehen. Im Modifikationsschritt wird diese Kette in zwei, vier oder sechs möglichst gleichlange Stücke geteilt. Jede Instanz besitzt zwei Methoden, die wiederum bis zu zwei Werte für die Ausführungsdauer enthält. Die Ergebnisse der aus einem Graphen entstandenen Graphen sind aufgrund der ähnlichen Topologie (Kettenstücke) vergleichbar.

Abbildung 6.5 zeigt die Entwicklung der Optimierungsdauer in Abhängigkeit von der Anzahl der Kettenstücke. Die Dauern für die Optimierungen von Graphen mit demselben Ursprungsgraphen wurden so normiert, daß der Maximalwert 1 ist. Die Untersuchung zeigt den großen Einfluß der Datenabhängigkeiten auf die Optimierungsdauer deutlich.

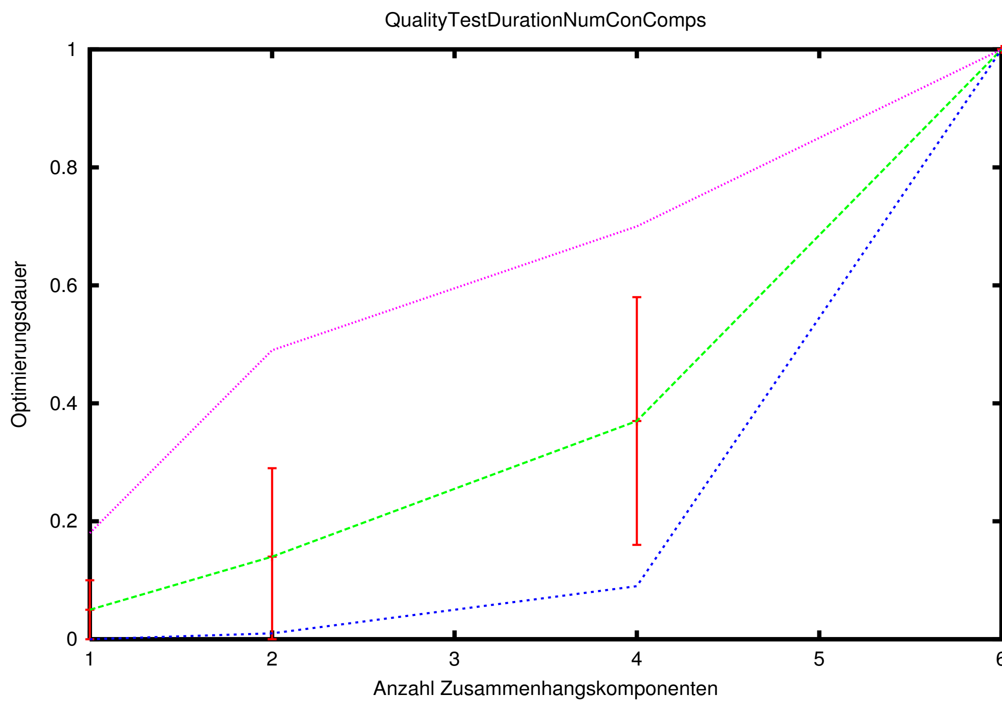

Abbildung 6.5: Anzahl der Zusammenhangskomponenten → Optimierungsdauer

Anzahl der Methoden je Instanz

Diese Untersuchung zeigt die Entwicklung der Dauer der Qualitätsoptimierung für Graphen mit unterschiedlich vielen Methoden je Instanz. Die Anzahl der Methoden je Instanz nimmt die Werte 1, 2, 3, 4 und 5 an. Mit dem Graphgenerator wurden 75 Graphen zufällig erzeugt, die je zehn Methoden pro Instanz besitzen. Die Anzahl der Instanzen lag zwischen 6 und 20 und die Graphen enthielten vier Zusammenhangskomponenten. Jede Methode besaß bis zu zehn Ausführungsdauern. Im Modifikationsschritt wird sukzessive, zufällig eine der Methoden mit den längeren worst-case Ausführungszeiten entfernt, um die Einplanbarkeit des Graphen zu erhalten. Die Ergebnisse der aus einem Graphen entstandenen Graphen bleiben dadurch vergleichbar.

Abbildung 6.6 zeigt, daß die Optimierungsdauer mäßig mit der Anzahl der Methoden je Instanz steigt. Dies ist darin begründet, daß zusätzliche Methoden aufgrund ihrer längeren worst-case Ausführungsdauer oft nicht eingeplant werden können, oder aber daß durch die Pareto-Optimalität zusätzlicher Methoden mehrere andere Methoden nicht mehr berücksichtigt werden müssen.

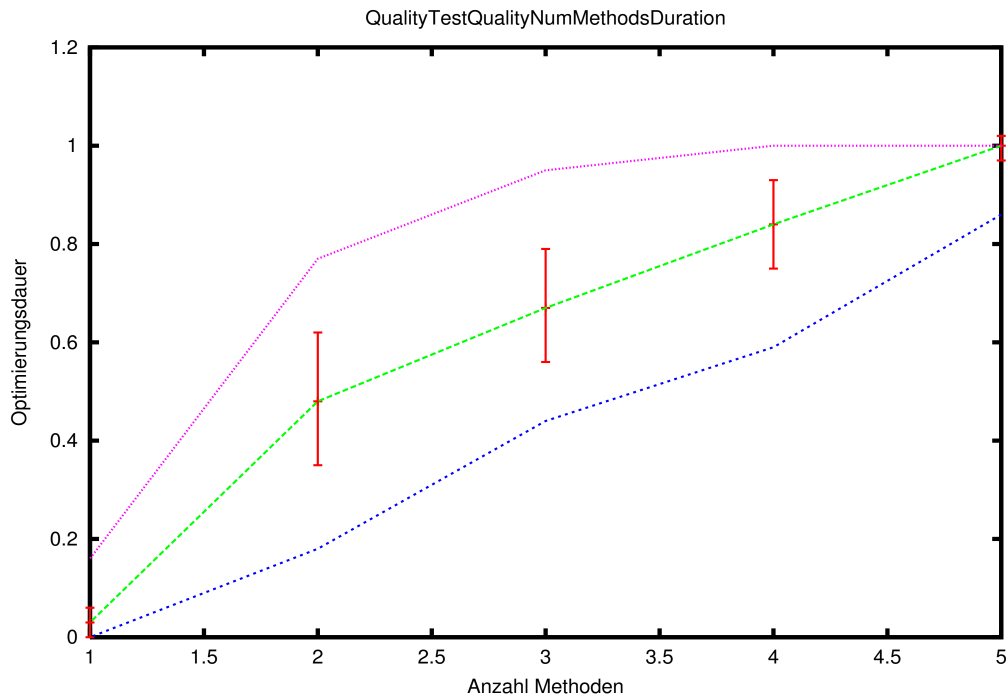

Abbildung 6.6: Anzahl der Methoden → Optimierungsdauer

Anzahl der Ausführungszeiten je Methode

Diese Untersuchung zeigt die Entwicklung der Dauer der Qualitätsoptimierung für Graphen mit unterschiedlich vielen Ausführungszeitspezifikationen je Methode. Die Zahl der Ausführungszeiten je Methode nimmt die Werte 1, 2, 3, 4 und 5 an. Mit dem Graphgenerator wurden 70 Graphen zufällig erzeugt, die bis zu fünf Ausführungszeiten pro Methode besitzen.[3] Pro Instanz enthalten die Graphen zehn Methoden. Die Anzahl der Instanzen lag zwischen 6 und 20 und die Graphen enthielten vier Zusammenhangskomponenten. Im Modifikationsschritt wird sukzessive die mittlere der Ausführungszeiten entfernt, um die Einplanbarkeit und worst-case Auslastung des Graphen zu erhalten. Die Ergebnisse der aus einem Graphen entstandenen Graphen bleiben dadurch vergleichbar.

Abbildung 6.7 den Einfluß der Anzahl der Ausführungsdauern je Methode. Der Anstieg der Optimierungsdauer wird durch die Mehrfachverwendung gleicher Teillösungen gedämpft.

[3]Bei worst-case Ausführungszeiten kleiner als fünf ist es aufgrund des diskreten Zeitmodells nicht möglich die geforderte Anzahl zu erreichen.

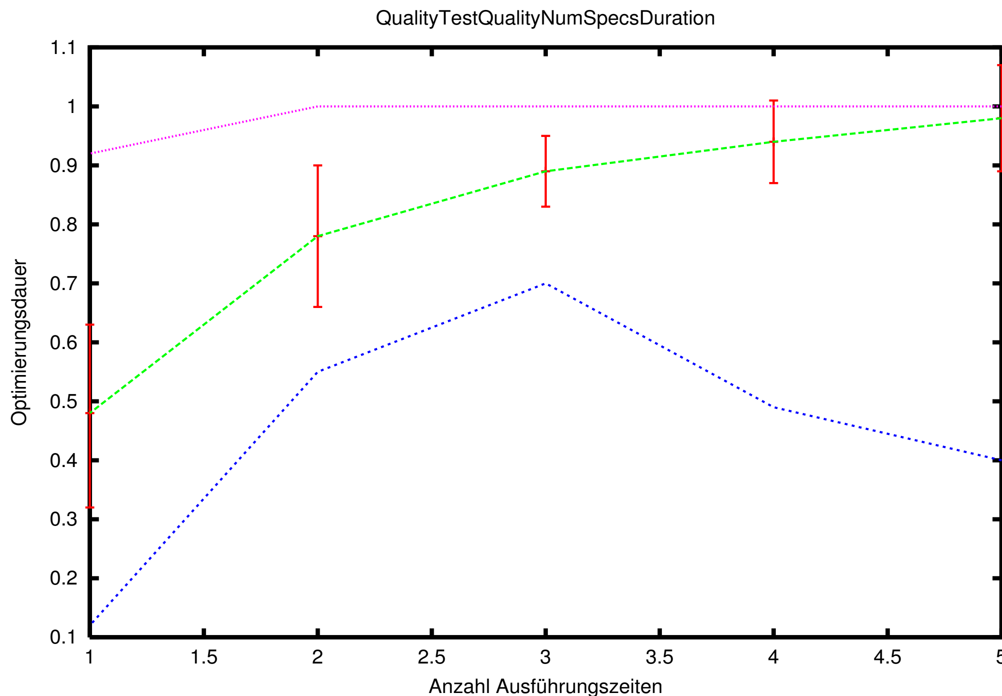

Abbildung 6.7: Anzahl der Ausführungszeiten je Methode → Optimierungsdauer

Minimale worst-case Auslastung

Diese Untersuchung zeigt die Entwicklung der Dauer der Qualitätsoptimierung für Graphen mit unterschiedlicher minimaler worst-case Auslastungen[4]. Die Last nimmt die Werte 0.1, 0.2, ..., 0.9 sowie 0.95 an. Mit dem Graphgenerator wurden 90 Graphen zufällig erzeugt. Die Anzahl der Instanzen liegt zwischen 6 und 20, jede Instanz besitzt drei Methoden mit drei Ausführungsdauern. Im Modifikationsschritt wurden die Periodendauern so angepaßt, daß die spezifizierte Last erzielt wurde. Die Ergebnisse der aus einem Graphen entstandenen Graphen bleiben dadurch vergleichbar.

Abbildung 6.8 zeigt, daß die Optimierungsdauer bei mittlerer Auslastung am größten ist. Ein Grund hierfür ist die große Freiheit bei der Optimierung von Graphen mit kleiner Auslastung, weil hier schnell gute Lösungen gefunden und dadurch von den Abbruchbedingungen große Teile des Suchraums ausgeschlossen werden. Bei hoher Auslastung gibt es nur wenige zulässige Einplanungen, d.h. der Suchraum ist hier relativ klein. Bei mittlerer Auslastung hingegen gibt es viele Lösungen aber es ist dennoch schwierig gute Lösungen zu finden.

[4]*minimale worst-case Last*: Die Last, wenn jeweils die Methode mit der kürzesten worst-case Ausführungszeit gewählt wird und diese immer ihre worst-case Ausführungszeit benötigt

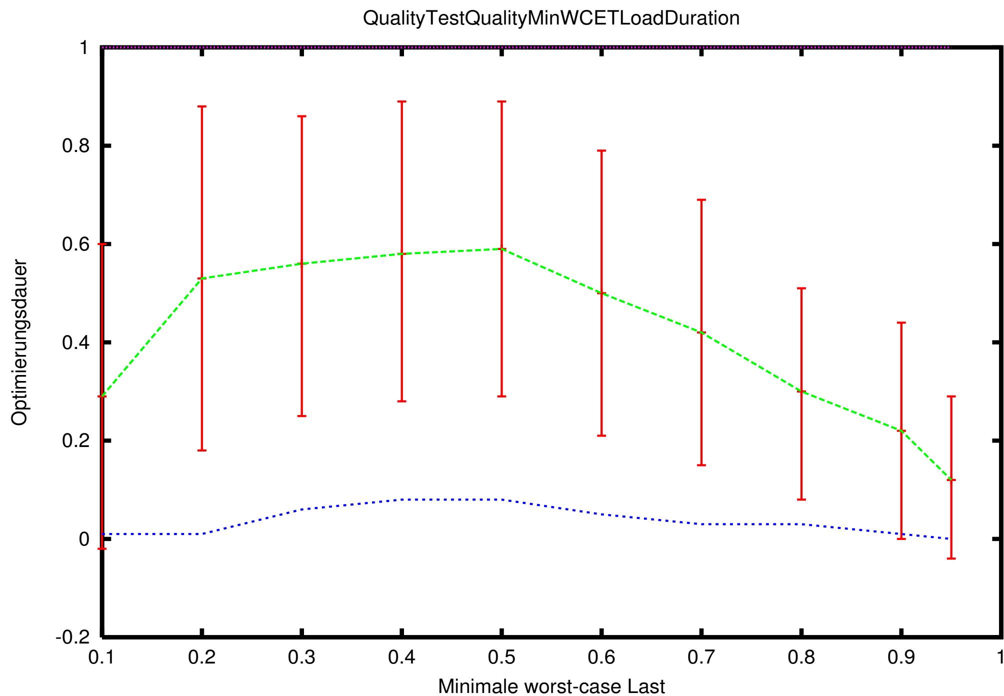

Abbildung 6.8: Minimale worst-case Auslastung → Optimierungsdauer

Verhältnis durchschnittliche Ausführungsdauer zu worst-case Ausführungsdauer

Diese Untersuchung zeigt die Entwicklung der Dauer der Qualitätsoptimierung für Graphen mit unterschiedlichem Verhältnis von durchschnittlicher Ausführungszeit zu worst-case Ausführungszeit. Das Verhältnis nimmt die Werte 0.5, 0.6, 0.7, 0.8, 0.9 sowie 0.95 an. Mit dem Graphgenerator wurden 50 Graphen zufällig erzeugt. Die Anzahl der Instanzen liegt zwischen 6 und 20, jede Instanz besitzt drei Methoden. Im Modifikationsschritt wurden für jede Methode nur die kürzeste und die längste Ausführungszeit beibehalten und ihre Wahrscheinlichkeiten so angepaßt, daß der spezifizierte Wert erreicht wurde. Die Ergebnisse der aus einem Graphen entstandenen Graphen bleiben dadurch vergleichbar.

Abbildung 6.9 zeigt den geringen Einfluß des Verhältnisses durchschnittliche Ausführungsdauer zu worst-case Ausführungsdauer. Bei der Annäherung an den Wert 1 sinkt die Optimierungsdauer etwas, da eine höhere durchschnittliche Ausführungsdauer zu einer Verkleinerung des Suchraumes führt.

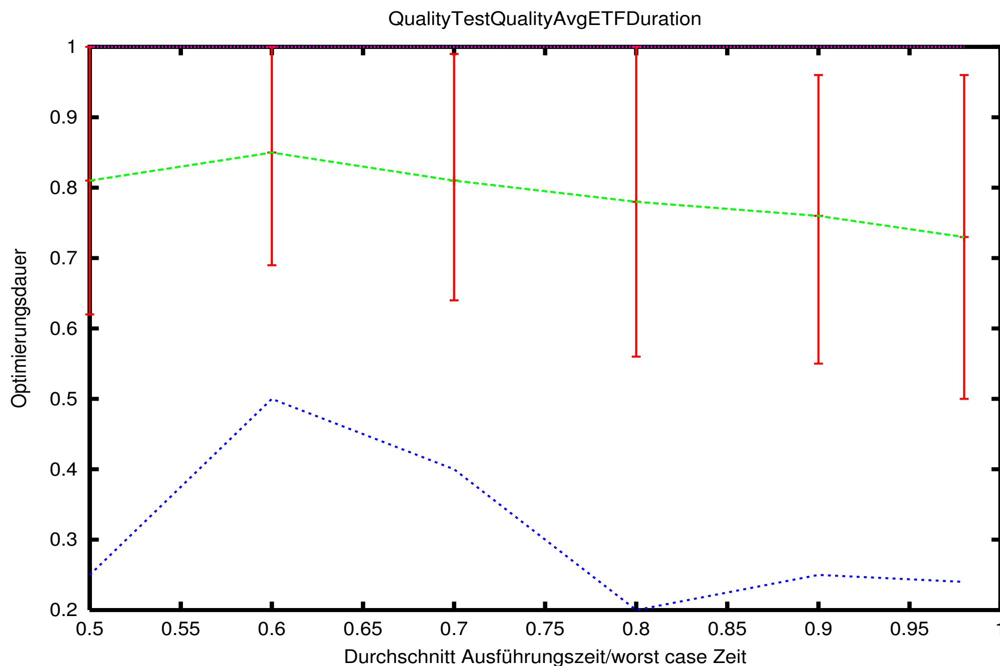

Abbildung 6.9: Durchschnittliche Ausführungsdauer/worst-case Ausführungsdauer → Optimierungsdauer

Anzahl der Leistungsstufen

Diese Untersuchung zeigt die Entwicklung der Dauer der Energieoptimierung für Graphen mit unterschiedlich vielen Prozessorleistungsstufen. Die Optimierungsdauer wurde für 1, 2, 4 und 8 Leistungsstufen untersucht. Mit dem Graphgenerator wurden 25 Graphen mit jeweils 40 Instanzen zufällig erzeugt, die jeweils aus einer Kette von datenabhängigen Instanzen bestehen. Jede Instanz besitzt zwei Methoden, die wiederum bis zu zwei Werte für die Ausführungsdauer enthält. Im Modifikationsschritt wurde die Anzahl der Leistungsstufen auf den spezifizierten Wert gesetzt. Die Ergebnisse der aus einem Graphen entstandenen Graphen bleiben dadurch vergleichbar.

Abbildung 6.10 zeigt den relativ linearen Anstieg der Optimierungsdauer mit der Anzahl der Leistungsstufen.

6.2.2 Parameter des Optimierers

Dieser Abschnitt untersucht den Einfluß einiger Parameter der Optimierer auf die Optimierungsdauer. Die zugrundeliegenden Graphen enthalten jeweils 20 Instanzen mit je vier Methoden, die bis zu drei Ausführungsdauern besitzen. Die Graphen wurden zuerst mit der anforderungsgetriebenen Dynamischen Programmie-

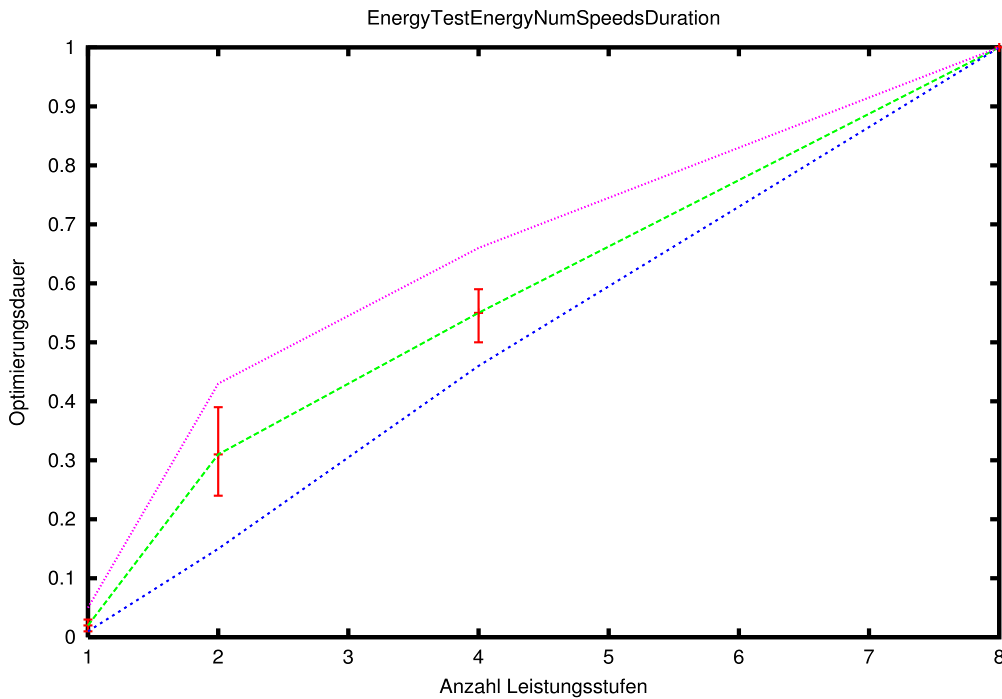

Abbildung 6.10: Anzahl der Leisungsstufen → Optimierungsdauer

rung optimiert, und die maximal erzielbare Qualität gespeichert. Danach wurden die Graphen mit Simulated Annealing optimiert, wobei jeweils die Optimierungsdauer bis zum Erreichen von 95,5% der ermittelten maximalen Qualität gemessen wurde.

Lotse

Diese Untersuchung zeigt den Einfluß des gewählten Lotsen auf die Optimierungsdauer. Für die anforderungsgetriebene Dynamische Programmierung wurden die Lotsen als Initialisierungs- und Optimierungslotse eingesetzt. Beim Simulated Annealing Optimierer wurden nur die Lotsen für die Initialisierung ausgetauscht, da der Optimierungslotse hier auf den Lotsen *Zufall/Zufall* festgelegt ist. Mit dem Graphgenerator wurden 30 Graphen zufällig erzeugt. Die auf der x-Achse aufgetragenen Lotsen wurden für die automatische Auswertung numerisch kodiert. Z.B. wird der Lotse *(ERF, GREEDY)* durch den Wert 14 repräsentiert. Tabelle 6.1 zeigt die Kodierung der Lotsen für die beiden folgenden Untersuchungen.

Abbildung 6.11 zeigt, daß die Wahl des Lotsen für die anforderungsgetriebene Dynamische Programmierung nur geringe Auswirkungen hat. Der Lotse *EDF/Greedy* liefert die kürzeste Optimierungsdauer.

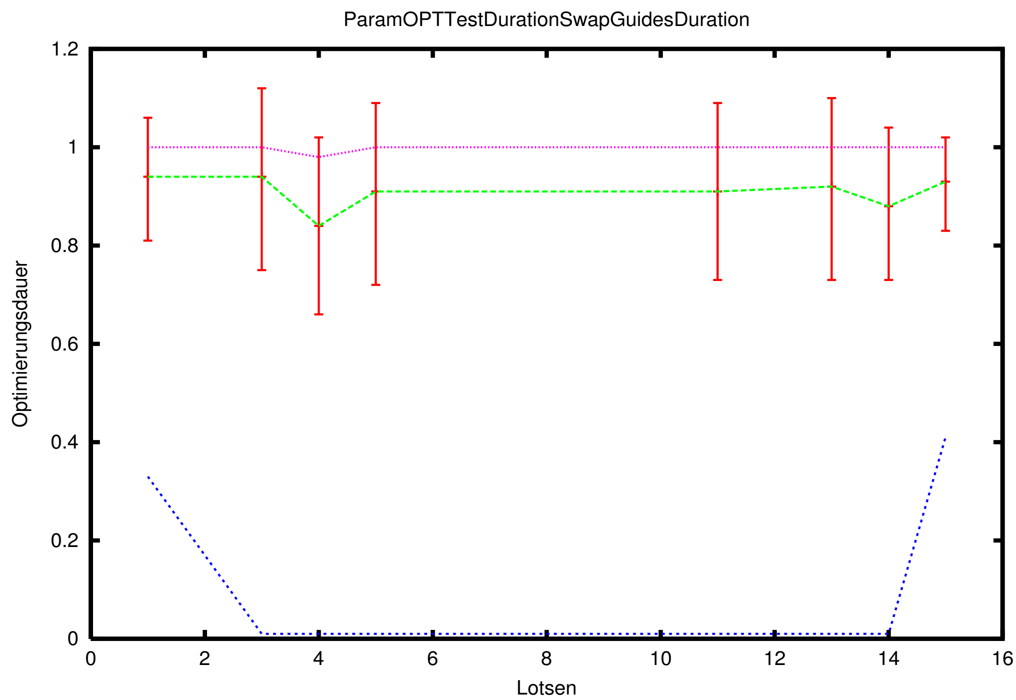

Abbildung 6.11: ADP: Lotse → Optimierungsdauer (Kodierung siehe Tabelle 6.1)

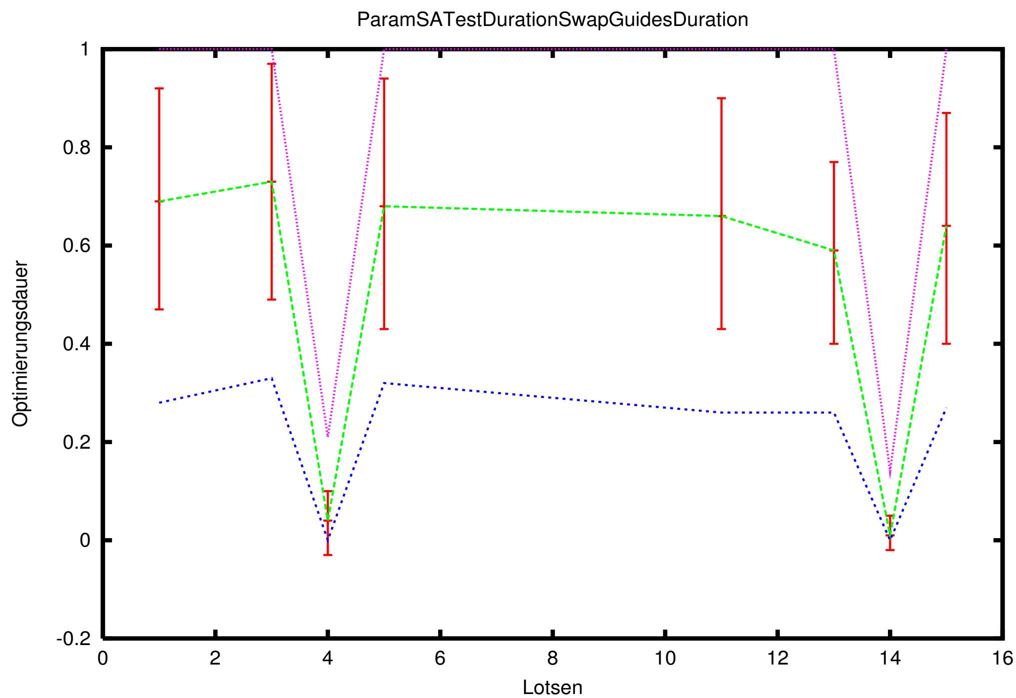

Abbildung 6.12: SA: Initialisierungslotse → Optimierungsdauer (Kodierung siehe Tabelle 6.1)

	steigende Dauer (*ascWCET*)	sinkende Dauer (*descWCET*)	Greedy (*GREEDY*)	Zufall (*PERTURB*)
EDF	1	3	4	5
ERF	11	13	14	15

Tabelle 6.1: Kodierung der Lotsen in den Untersuchungen

Der Einfluß des Initialisierungslotsen auf die Optimierungsdauer bei Simulated Annealing ist in Abbildung 6.12 zu sehen. Hier ist der Einfluß der Lotsen größer, und die besten Ergebnisse liefert der *ERF/Greedy* Lotse. Insgesamt schneiden die *ERF*-Lotsen etwas besser ab als die *EDF*-Lotsen.

Größe des Knotenspeichers

Diese Untersuchung zeigt die Auswirkungen der Größe[5] des Knotenspeichers auf die Optimierungsdauer. Mit dem Graphgenerator wurden 30 Graphen zufällig erzeugt. Zuerst wurde für jeden Graphen eine Optimierung mit der Knotenspeichergröße 100000 durchgeführt, um die Anzahl der benötigten Knoten zu bestimmen. In den anschließenden Tests wurde dann jeweils ein Bruchteil der ermittelten Knotenzahl eingestellt.

Für die anforderungsgetriebene Dynamische Programmierung zeigt Abbildung 6.13 einen starken Anstieg der Optimierungsdauer, wenn die Knotenspeichergröße nur 30% oder weniger der benötigten Knotenanzahl beträgt. Ab 40% der benötigten Knotenanzahl ist der Schiebefenstereffekt zu erkennen, d.h. die meisten der während der Optimierung aus dem Knotenspeicher verdrängten Knoten werden später nicht mehr benötigt. Der konstante Verlauf des Graphen bei mehr als der benötigten Knotenzahl zeigt, das ein zu groß gewählter Knotenspeicher keine negativen Auswirkungen auf die Optimierungsdauer hat. Der Knotenspeicher sollte also immer so groß wie möglich gewählt werden.

Abbildung 6.14 zeigt deutlich, daß die Optimierungsdauer bei Simulated Annealing nur wenig von der Größe des Knotenspeichers abhängt. Zum einen bedeutet dies, daß dieser Optimierer von einem großen Knotenspeicher nur wenig profitiert, zum anderen heißt es jedoch auch, daß Simulated Annealing für komplexe Systemmodelle nur wenig unter einem relativ kleinem Knotenspeicher leidet.

Aktivierung Abbruchbedingung worst-case Zeit

Diese Untersuchung zeigt die Auswirkungen der Überprüfung der Abbruchbedingung, ob die verbleibende Zeit zur Einplanung der verbleibenden Instanzen aus-

[5] Anzahl der gleichzeitig verfügbaren Knoten

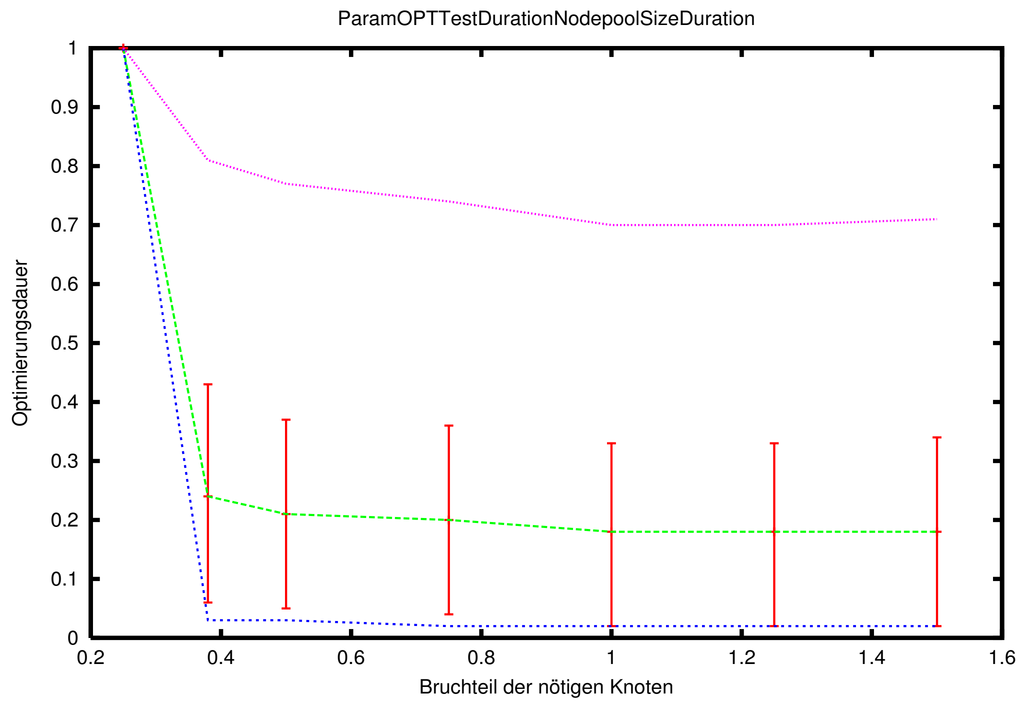

Abbildung 6.13: ADP: Knotenspeichergröße → Optimierungsdauer

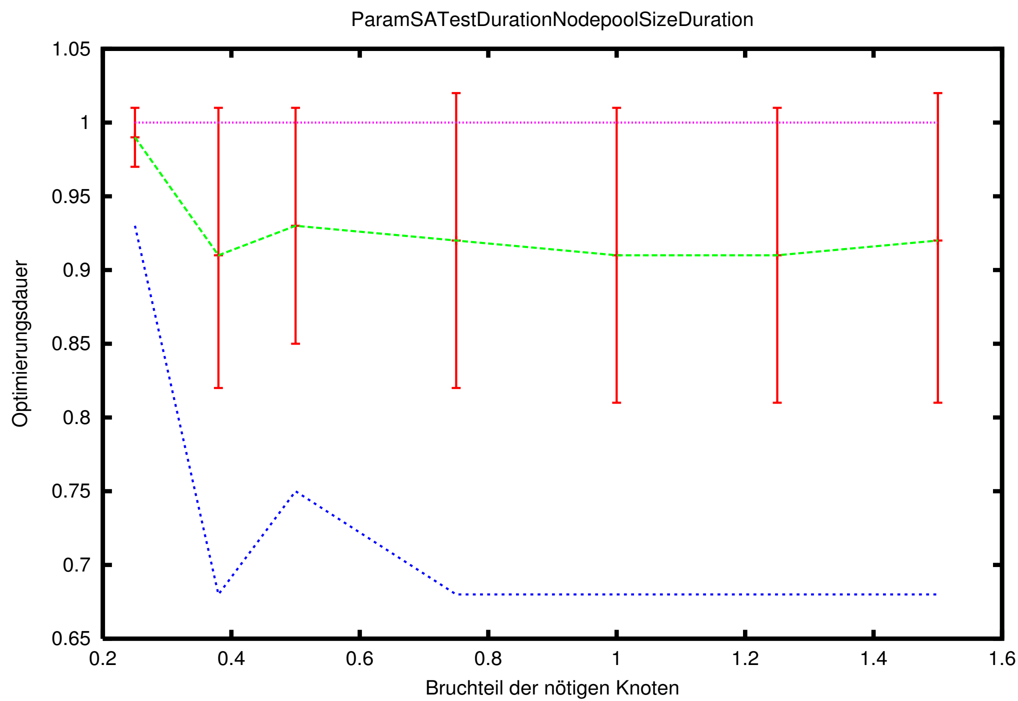

Abbildung 6.14: SA: Knotenspeichergröße → Optimierungsdauer

reicht, auf die Optimierungsdauer. Mit dem Graphgenerator wurden 90 Graphen zufällig erzeugt. Die Graphen besitzen jeweils zehn Instanzen mit je drei Methoden je Instanz und drei Ausführungsdauern je Methode. Sie enthalten jeweils zwei Zusammenhangskomponenten, die durch Datenabhängigkeiten gebildet werden. Der Wert 0 auf der x-Achse repräsentiert die deaktivierte Überprüfung der Bedingung und der Wert 1 steht für die aktivierte Überprüfung der Bedingung.

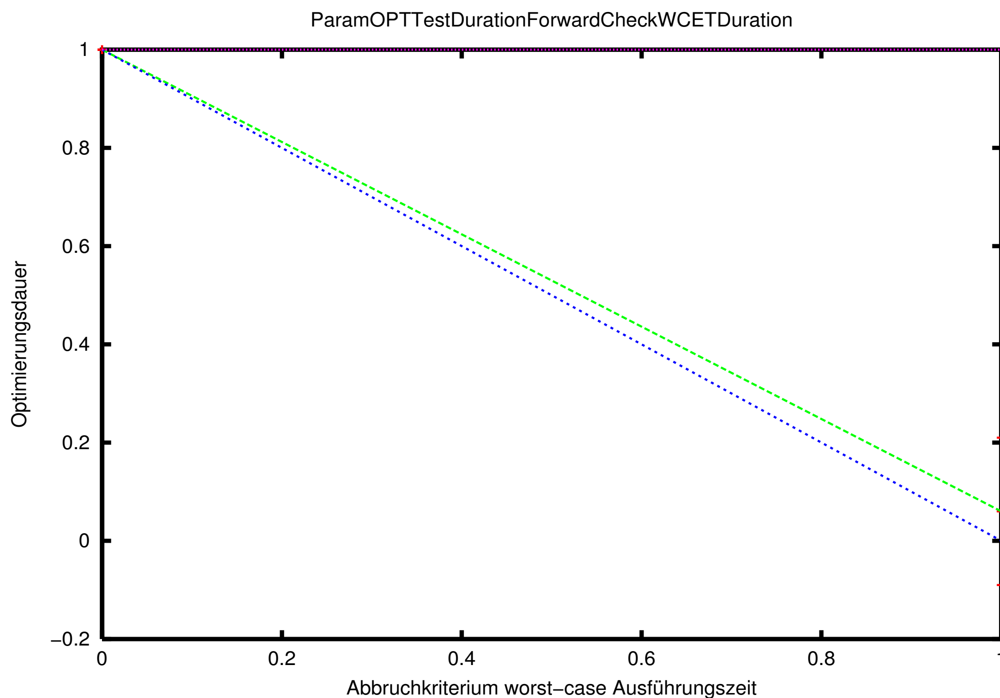

Abbildung 6.15: Aktivierung der Zeit-Abbruchsbedingung → Optimierungsdauer

Abbildung 6.15 zeigt den Einfluß der Überprüfung auf die Dauer der Qualitätsoptimierung mit anforderungsgetriebener Dynamischen Programmierung. Diese Abbruchbedingung erlaubt es sehr viele Knoten bei der Optimierung auszuschließen und führt dadurch zu sehr großen Zeiteinsparungen.

Abbildung 6.16 zeigt den ebenfalls deutlichen, positiven Einfluß der Abbruchbedingung auf die Dauer der Qualitätsoptimierung mit Simulated Annealing.

Aktivierung Abbruchbedingung Qualität

Diese Untersuchung zeigt die Auswirkungen der Überprüfung der Abbruchbedingung, ob noch eine höhere Qualität als die bereits erreichte erzielt werden kann, auf die Optimierungsdauer. Mit dem Graphgenerator wurden 90 Graphen zufällig erzeugt. Die Graphen besitzen jeweils zehn Instanzen mit je drei Methoden je

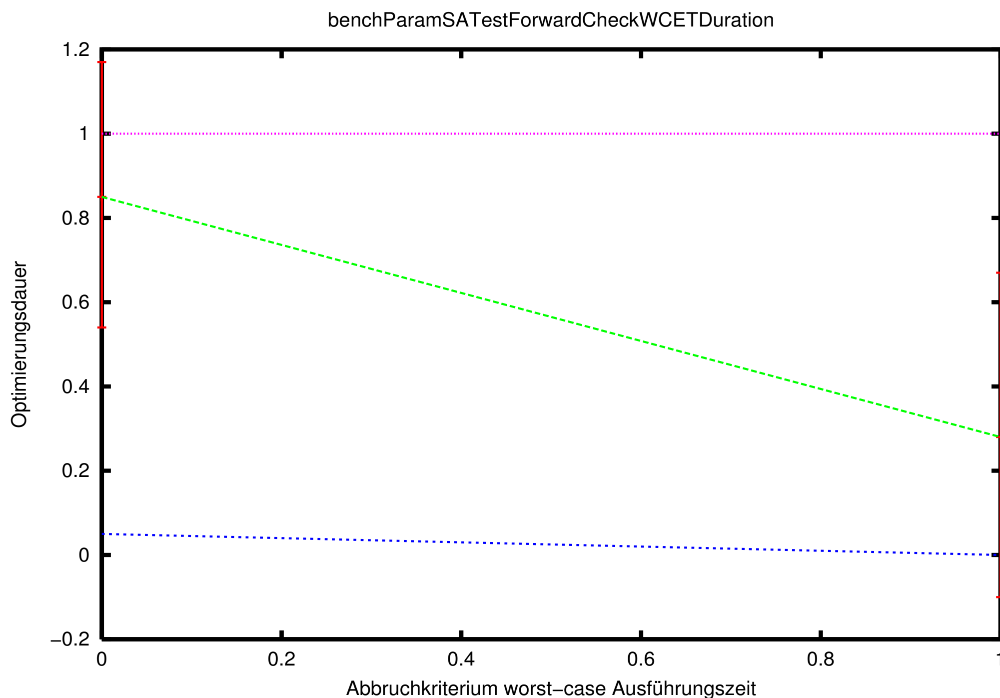

Abbildung 6.16: Aktivierung der Zeit-Abbruchsbedingung → Optimierungsdauer

Instanz und drei Ausführungsdauern je Methode. Sie enthalten jeweils zwei Zusammenhangskomponenten, die durch Datenabhängigkeiten gebildet werden. Der Wert 0 auf der x-Achse repräsentiert die deaktivierte Überprüfung der Bedingung und der Wert 1 steht für die aktivierte Überprüfung der Bedingung.

Abbildung 6.17 zeigt den Einfluß der Überprüfung auf die Dauer der Qualitätsoptimierung mit anforderungsgetriebener Dynamischen Programmierung. In einigen Fällen reduzierte die Überprüfung der Bedingung die Optimierungsdauer durch den Ausschluß von Knoten, in vielen Fällen war die für die Auswertung der Bedingung benötigte Zeit jedoch größer als die Zeitersparnis durch die ausgeschlossenen Knoten. Im Durchschnitt führt die Auswertung der Bedingung zu einer leichten Erhöhung der Optimierungsdauer.

Abbildung 6.18 zeigt den ebenfalls geringen Einfluß der Abbruchbedingung auf die Dauer der Qualitätsoptimierung mit Simulated Annealing. Hier führt die Überprüfung der Bedingung im Durchschnitt zu leichten Zeiteinsparungen.

Zeitraster

Diese Untersuchung zeigt die Auswirkungen der Weite des Zeitrasters auf die Optimierungsdauer. Mit dem Graphgenerator wurden 60 Graphen zufällig erzeugt. Die Graphen besitzen jeweils zehn Instanzen mit je drei Methoden je Instanz und

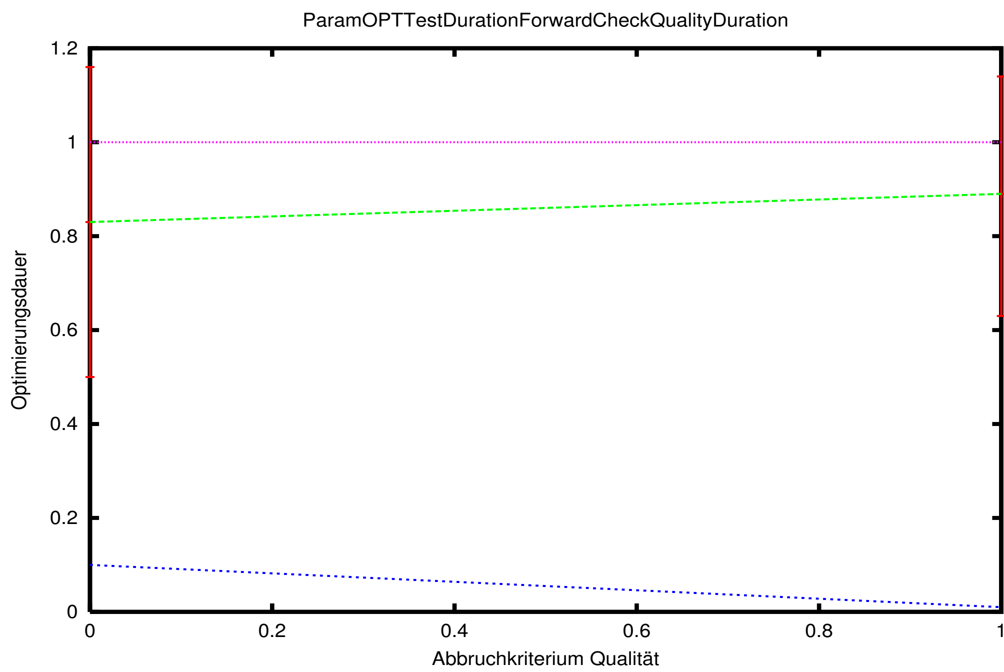

Abbildung 6.17: Aktivierung der Qualität-Abbruchsbedingung → Optimierungsdauer

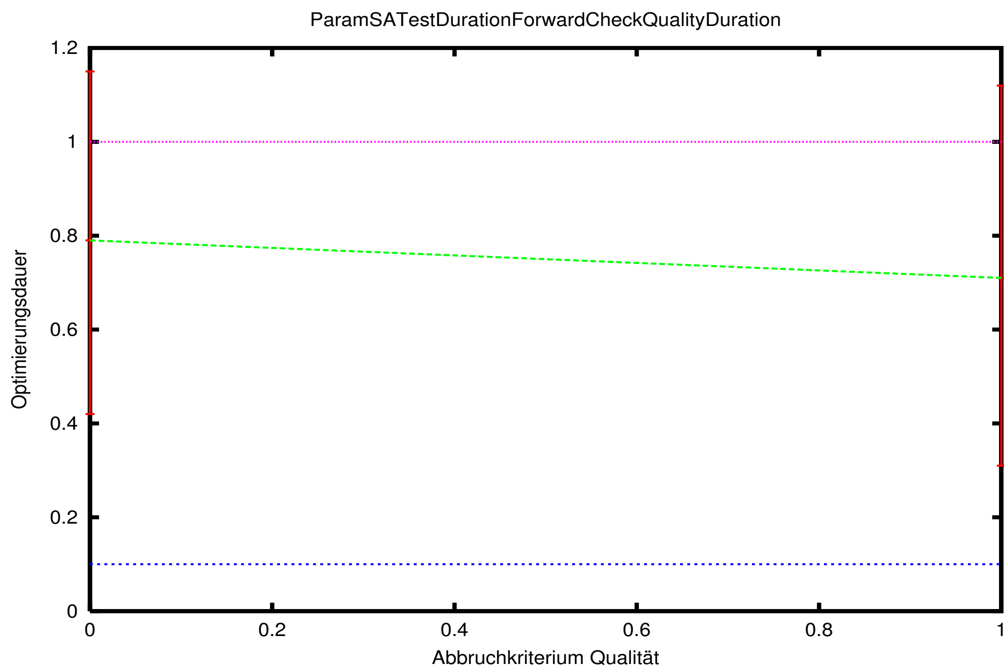

Abbildung 6.18: Aktivierung der Qualität-Abbruchsbedingung → Optimierungsdauer

drei Ausführungsdauern je Methode. Sie enthalten jeweils zwei Zusammenhangskomponenten, die durch Datenabhängigkeiten gebildet werden.

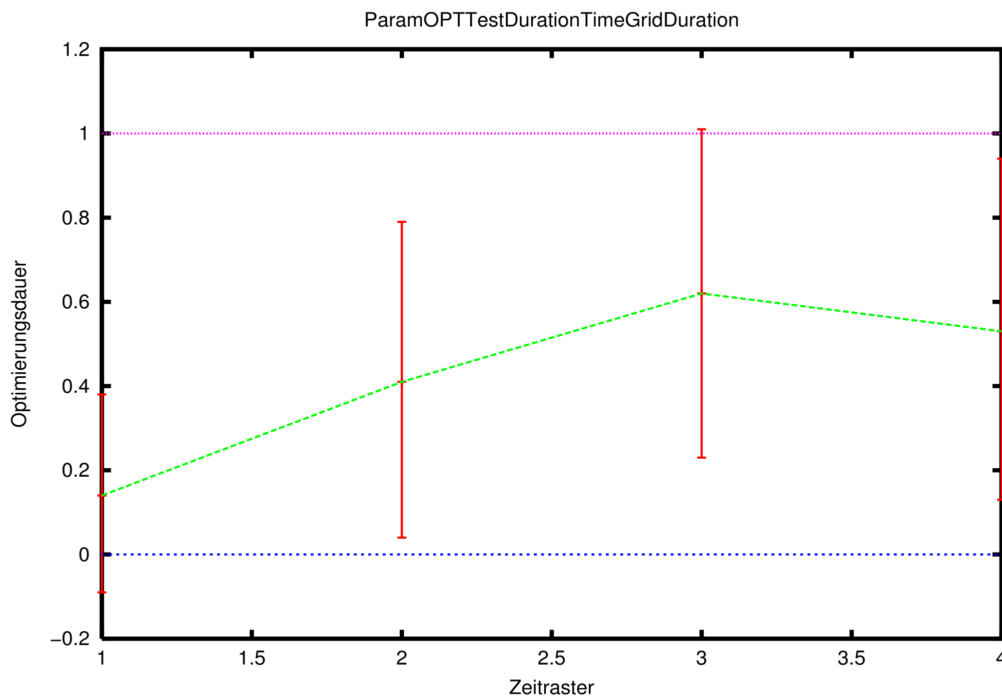

Abbildung 6.19: Feinheit des Zeitrasters → Optimierungsdauer

Abbildung 6.19 zeigt den Einfluß der Zeitrasterweite auf die Dauer der Qualitätsoptimierung mit der anforderungsgetriebenen Dynamischen Programmierung. Überraschenderweise führt die Vergröberung des Rasters durchschnittlich zu einer Erhöhung[6] der Optimierungsdauer. Eine nähere Analyse zeigte, daß die Anzahl der Knotenaktualisierungen bei einer Rastervergröberung sinkt, d.h. es werden weniger zulässige Knoten erzeugt. Die Anzahl der insgesamt erzeugten Knoten - und damit die Optimierungsdauer - steigt jedoch sehr stark an. Die Ursache für die vielen erzeugten, ungültigen Knoten sind die ohnehin schon relativ knappen Fristen der Prozesse, die nach einer zusätzlichen Vergröberung des Zeitrasters nicht mehr eingehalten werden können und zu häufigem Backtracking führen.

Abbildung 6.20 zeigt den ebenfalls negativen Einfluß der Zeitrasterweite auf die Dauer der Qualitätsoptimierung mit Simulated Annealing. Die Absenkung der Optimierungsdauer bei den Rasterwerten 3 und 4 ist durch die nicht mehr einplanbaren Graphen und sehr viele Graphen, die nicht innerhalb der Zeitbeschränkung

[6] Die Absenkung der Optimierungsdauer bei der Rasterweite 4 ist durch Nulleinträge für nicht mehr einplanbare Graphen bedingt.

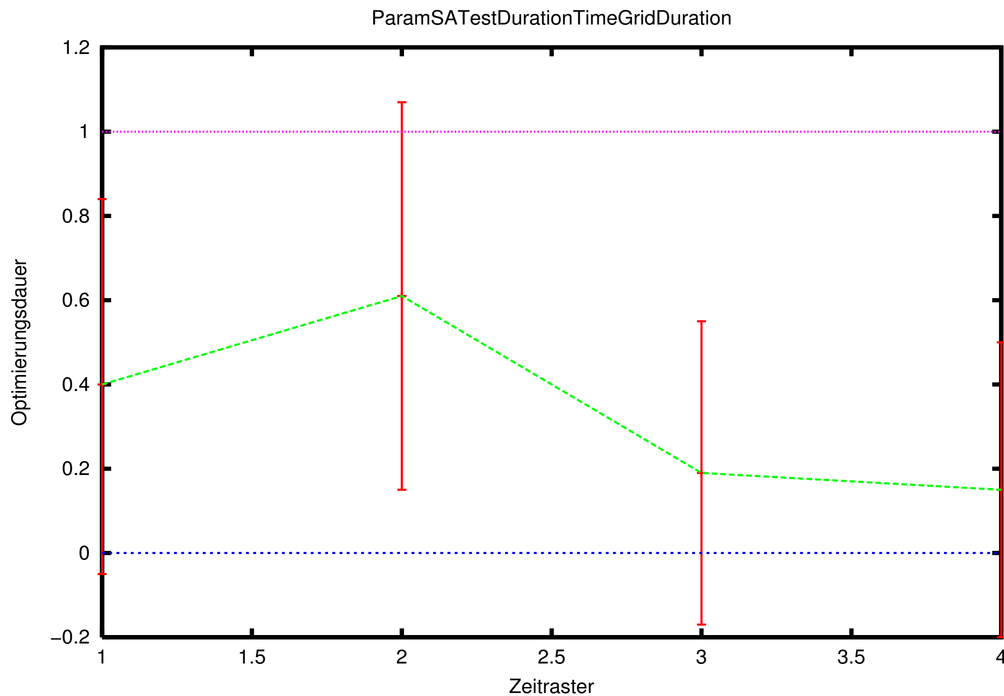

Abbildung 6.20: Feinheit des Zeitrasters → Optimierungsdauer

mit der geforderten Qualität eingeplant werden konnten, bedingt. Diese Graphen fließen in der automatisierten Auswertung ebenfalls mit Dauer 0 ein.

6.3 Qualität

Die Untersuchungen in diesem Abschnitt sollen die Auswirkungen einiger Modeleigenschaften auf die erzielbare maximale Qualität aufzeigen. Die durchgeführten Untersuchungen zeigen, wie sich die optimal erzielbare durchschnittliche Qualität verändert, wenn Eigenschaften des Anwendungsgraphen, wie z.B. die Anzahl der Instanzen oder die Anzahl der Methoden je Instanz, verändert werden. Die Optimierung erfolgte mit dem Verfahren der anforderungsgetriebenen Dynamischen Programmierung.

6.3.1 Anzahl der Methoden je Instanz

Diese Untersuchung zeigt die Entwicklung der maximal möglichen Qualität für Graphen mit unterschiedlich vielen Methoden je Instanz. Die Anzahl der Methoden je Instanz nimmt die Werte 1, 2, 3, 4 und 5 an. Mit dem Graphgenerator wurden 75 Graphen zufällig erzeugt, die je zehn Methoden pro Instanz besitzen.

Die Anzahl der Instanzen lag zwischen 6 und 20 und die Graphen enthielten vier Zusammenhangskomponenten. Jede Methode besaß bis zu zehn Ausführungsdauern. Im Modifikationsschritt wird sukzessive, zufällig eine der Methoden mit den längeren worst-case Ausführungszeiten entfernt, um die Einplanbarkeit des Graphen zu erhalten. Die Ergebnisse der aus einem Graphen entstandenen Graphen bleiben dadurch vergleichbar.

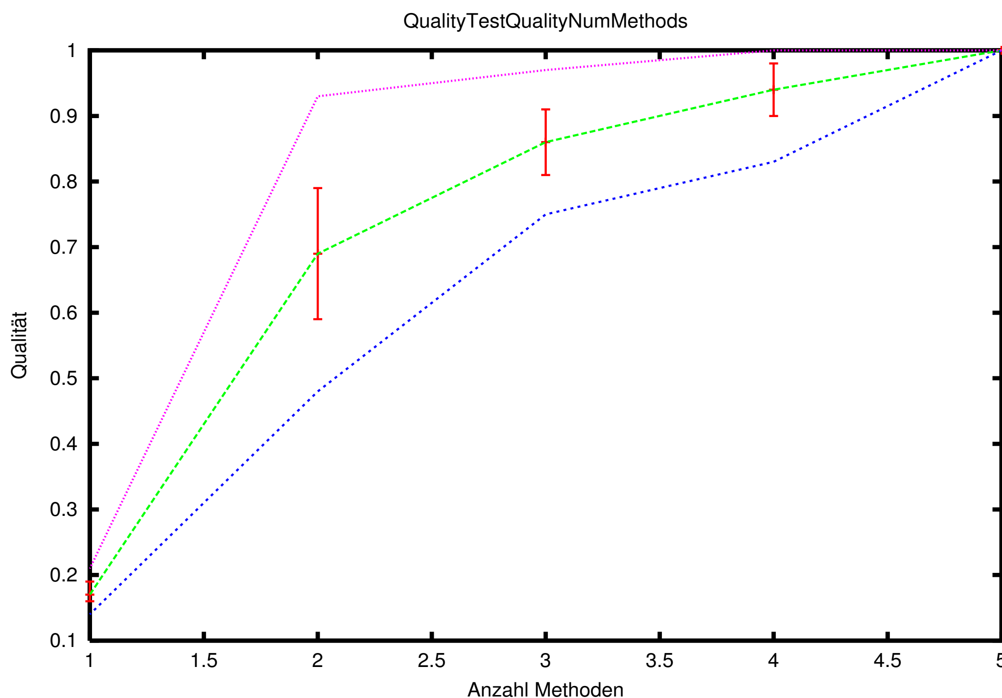

Abbildung 6.21: Anzahl der Methoden je Instanz → Qualität

Abbildung 6.21 zeigt, daß der Zugewinn an Qualität mit zunehmender Methodenzahl immer langsamer ansteigt. In den durchgeführten Tests konnte bereits mit drei Methoden je Instanz die vorgegebene Zeit sehr gut genutzt werden, und daher sind zusätzliche Methoden nur selten zur Qualitätssteigerung einplanbar.

6.3.2 Anzahl der Ausführungszeiten je Methode

Diese Untersuchung zeigt die Entwicklung der maximal möglichen Qualität für Graphen mit unterschiedlich vielen Ausführungszeitspezifikationen je Methode. Die Zahl der Ausführungszeiten je Methode nimmt die Werte 1, 2, 3, 4 und 5 an. Mit dem Graphgenerator wurden 75 Graphen zufällig erzeugt, die bis zu zehn

Ausführungszeiten pro Methode besitzen.[7] Pro Instanz enthalten die Graphen vier Methoden. Die Anzahl der Instanzen betrug 20 und die Graphen enthielten zwei Zusammenhangskomponenten. Im Modifikationsschritt wird sukzessive die mittlere der Ausführungszeiten entfernt, um die Einplanbarkeit und worst-case Auslastung des Graphen zu erhalten. Die Ergebnisse der aus einem Graphen entstandenen Graphen bleiben dadurch vergleichbar.

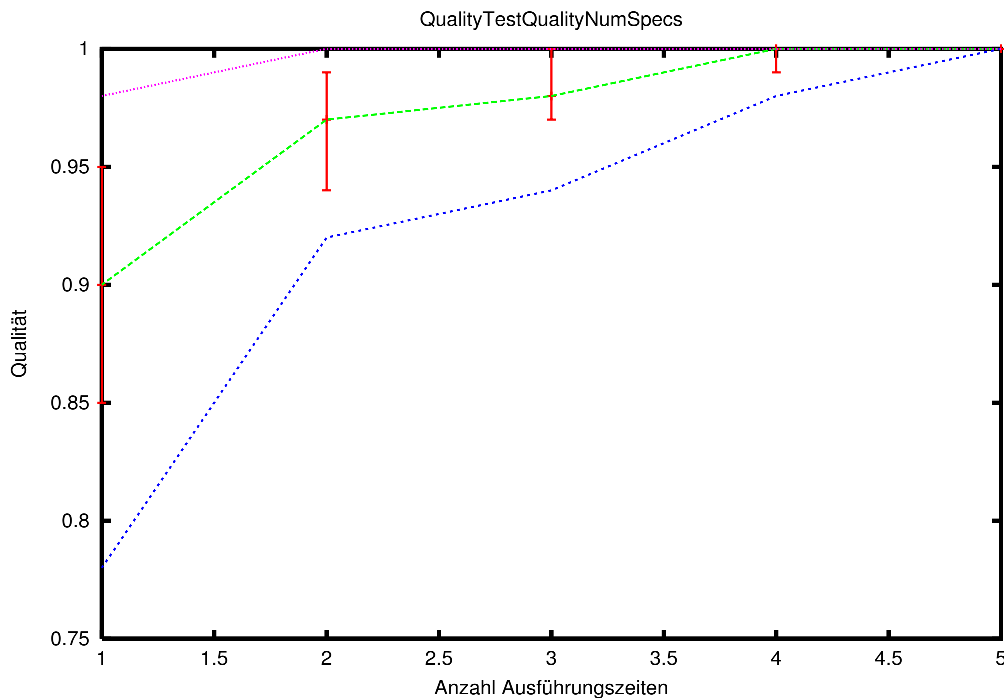

Abbildung 6.22: Anzahl der Ausführungszeiten je Methode → Qualität

Abbildung 6.22 zeigt den Einfluß der Anzahl der Ausführungsdauern auf die erzielbare Qualität. Die Spezifikation einer Ausführungsdauer zusätzlich zur worst-case Ausführungsdauer bringt den größten Qualitätsgewinn, während noch detailliertere Verteilungen nur einen sehr geringen Einfluß haben.

6.3.3 Minimale worst-case Auslastung

Diese Untersuchung zeigt die Entwicklung der maximal möglichen Qualität für Graphen mit unterschiedlicher minimaler worst-case Auslastungen. Dabei nimmt die Last die Werte 0.1, 0.2, . . . , 0.9 sowie 0.95 an. Mit dem Graphgenerator wurden 70 Graphen zufällig erzeugt. Die Anzahl der Instanzen betrug 15, jede Instanz

[7] Bei worst-case Ausführungszeiten kleiner als zehn ist es aufgrund des diskreten Zeitmodells nicht möglich die geforderte Anzahl zu erreichen.

besitzt drei Methoden mit drei Ausführungsdauern. Im Modifikationsschritt wurden die Periodendauern so angepaßt, daß die spezifizierte Last erzielt wurde. Die Ergebnisse der aus einem Graphen entstandenen Graphen bleiben dadurch vergleichbar.

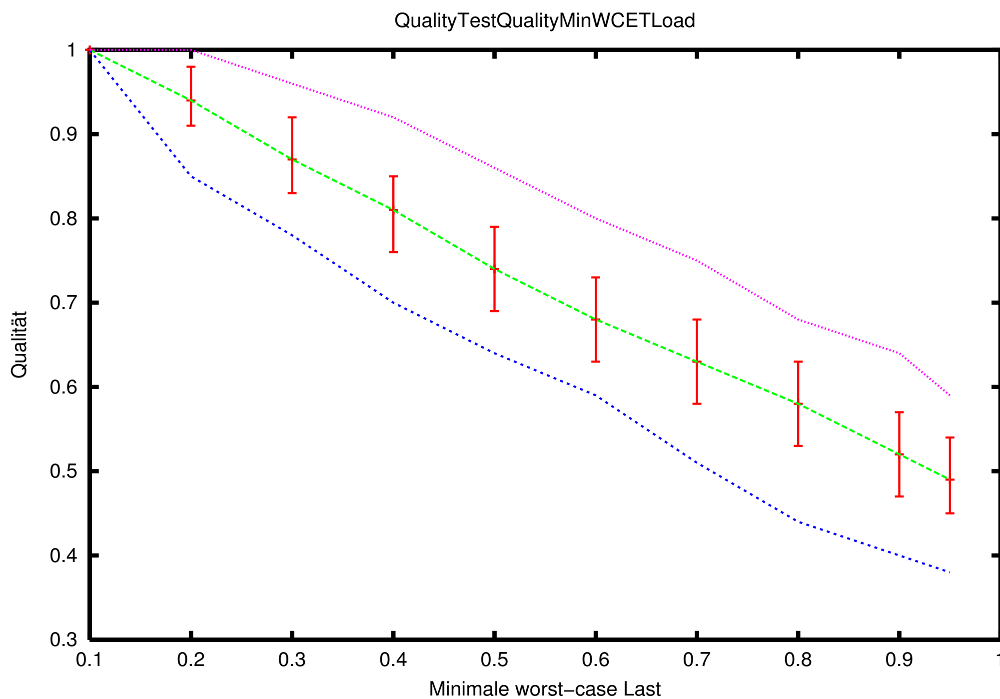

Abbildung 6.23: Minimale worst-case Auslastung → Qualität

Abbildung 6.23 zeigt die kontinuierliche Abnahme der erzielbaren Qualität bei steigender (worst-case) Last. Die Qualität nimmt sehr stark ab, da immer häufiger die schnellsten Methoden eingeplant werden müssen, um keine Fristen zu verletzen.

6.3.4 Verhältnis durchschnittliche Ausführungsdauer zu worst-case Ausführungsdauer

Diese Untersuchung zeigt die Entwicklung der maximal möglichen Qualität für Graphen mit unterschiedlichem Verhältnis von durchschnittlicher Ausführungszeit zu worst-case Ausführungszeit. Das Verhältnis nimmt die Werte 0.5, 0.6, 0.7, 0.8, 0.9 sowie 0.95 an. Mit dem Graphgenerator wurden 75 Graphen zufällig erzeugt, die bis zu zehn Ausführungszeiten pro Methode besitzen. Pro Instanz enthalten die Graphen vier Methoden. Die Anzahl der Instanzen betrug 20 und die Graphen enthielten zwei Zusammenhangskomponenten. Im Modifikationsschritt

wurden für jede Methode nur die kürzeste und die längste Ausführungszeit beibehalten und ihre Wahrscheinlichkeiten so angepaßt, daß der spezifizierte Wert erreicht wurde. Die Ergebnisse der aus einem Graphen entstandenen Graphen bleiben dadurch vergleichbar.

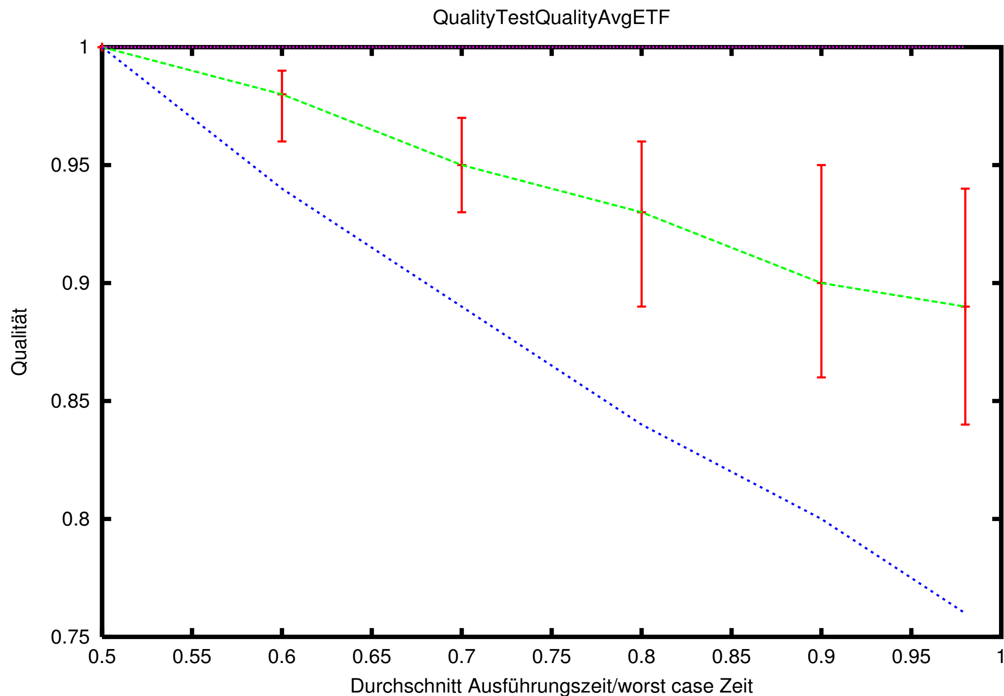

Abbildung 6.24: Durchschnittliche Ausführungszeit/worst-case Ausführungszeit $\longrightarrow$ Qualität

Abbildung 6.24 zeigt die ebenfalls sinkende erreichbare Qualität, wenn das Verhältnis von durchschnittlicher Ausführungsdauer zu worst-case Dauer steigt.

6.3.5 Zeitraster

Diese Untersuchung zeigt den Einfluß einer Vergröberung des Zeitrasters auf die erzielbare Qualität. Es wurden 60 mit dem Graphgenerator erzeugte Graphen untersucht.

Abbildung 6.25 zeigt, daß eine mäßige Vergröberung des Zeitrasters nur zu einer geringen Senkung der erzielbaren Qualität führt, jedoch konnten bereits bei einem Zeitraster von vier Zeiteinheiten einige der untersuchten Graphen nicht mehr eingeplant werden. Da die Vergröberung des Rasters auch die Optimierungsdauer erhöht ist der Wert 1 die beste Einstellung der Rasterweite.

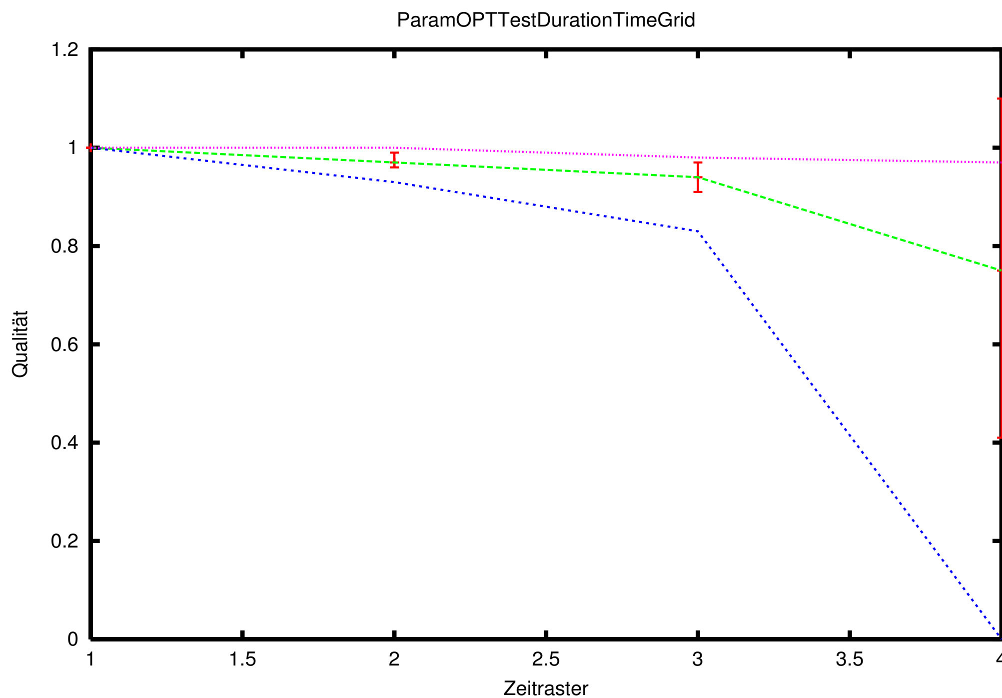

Abbildung 6.25: Feinheit des Zeitrasters → Qualität

6.4 Energieverbrauch

Die Untersuchungen in diesem Abschnitt sollen die Auswirkungen einiger Modelleigenschaften auf den erzielbaren minimalen, erwarteten Energieverbrauch aufzeigen. Die durchgeführten Untersuchungen zeigen, wie sich der optimal erzielbare, durchschnittliche Energieverbrauch verändert, wenn Eigenschaften des Anwendungsgraphen, wie z.B. die Anzahl der Instanzen oder die Anzahl der Methoden je Instanz, verändert werden. Die Optimierung erfolgte mit dem Verfahren der anforderungsgetriebenen Dynamischen Programmierung.

6.4.1 Anzahl der Methoden je Instanz

Diese Untersuchung zeigt die Entwicklung des Energieverbrauchs für Graphen mit unterschiedlich vielen Methoden je Instanz. Die Anzahl der Methoden je Instanz nimmt die Werte 1, 2, 3, 4 und 5 an. Mit dem Graphgenerator wurden 75 Graphen zufällig erzeugt, die je zehn Methoden pro Instanz besitzen. Die Anzahl der Instanzen lag zwischen 6 und 20 und die Graphen enthielten vier Zusammenhangskomponenten. Jede Methode besaß bis zu zehn Ausführungsdauern. Der Prozessor besitzt nur eine Leistungsstufe. Im Modifikationsschritt wird sukzessive, zufällig eine der Methoden mit den längeren worst-case Ausführungszeiten

entfernt, um die Einplanbarkeit des Graphen zu erhalten. Die Ergebnisse der aus einem Graphen entstandenen Graphen bleiben dadurch vergleichbar.

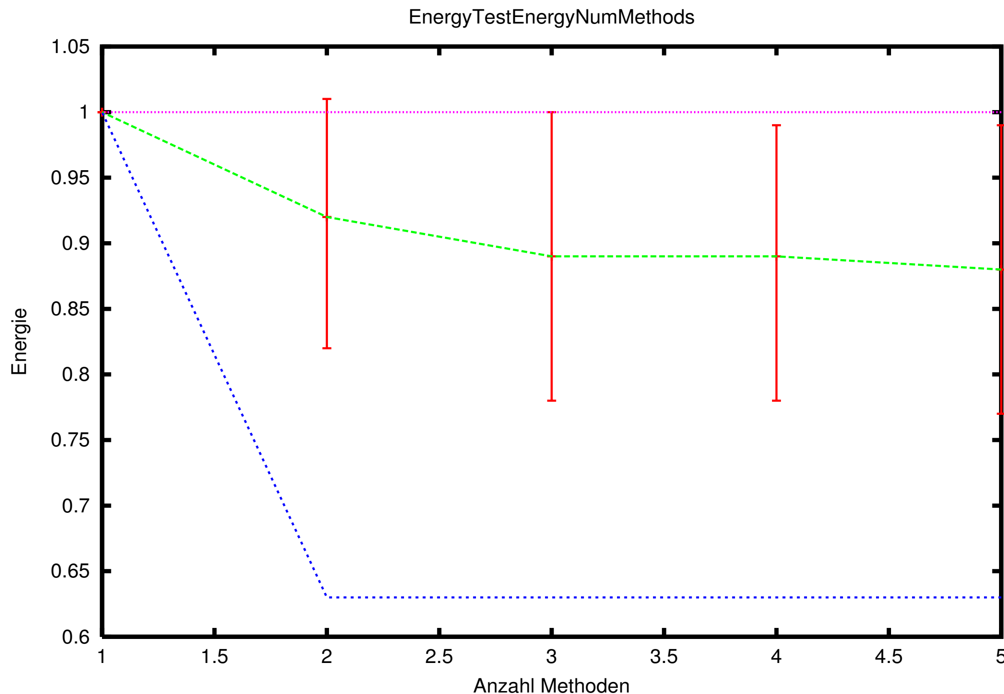

Abbildung 6.26: Anzahl der Methoden je Instanz → Energie

Abbildung 6.26 zeigt, daß der minimale zu erwartende Energieverbrauch durch die Spezifikation mehrerer Methoden je Instanz gesenkt werden kann. Die Angabe von mehr als drei Methoden je Instanz bewirkt jedoch nur noch geringe Einsparungen.

6.4.2 Anzahl der Ausführungszeiten je Methode

Diese Untersuchung zeigt die Entwicklung des Energieverbrauchs für Graphen mit unterschiedlich vielen Ausführungszeitspezifikationen je Methode. Die Zahl der Ausführungszeiten je Methode nimmt die Werte 1, 2, 3, 4 und 5 an. Mit dem Graphgenerator wurden 70 Graphen zufällig erzeugt, die bis zu fünf Ausführungszeiten pro Methode besitzen.[8] Pro Instanz enthalten die Graphen zehn Methoden. Die Anzahl der Instanzen lag zwischen 6 und 20 und die Graphen enthielten vier Zusammenhangskomponenten. Der Prozessor besitzt nur eine Leistungsstufe.

[8] Bei worst-case Ausführungszeiten kleiner als fünf ist es aufgrund des diskreten Zeitmodells nicht möglich die geforderte Anzahl zu erreichen.

Im Modifikationsschritt wurde sukzessive die mittlere der Ausführungszeiten entfernt, um die Einplanbarkeit und worst-case Auslastung des Graphen zu erhalten. Die Ergebnisse der aus einem Graphen entstandenen Graphen bleiben dadurch vergleichbar.

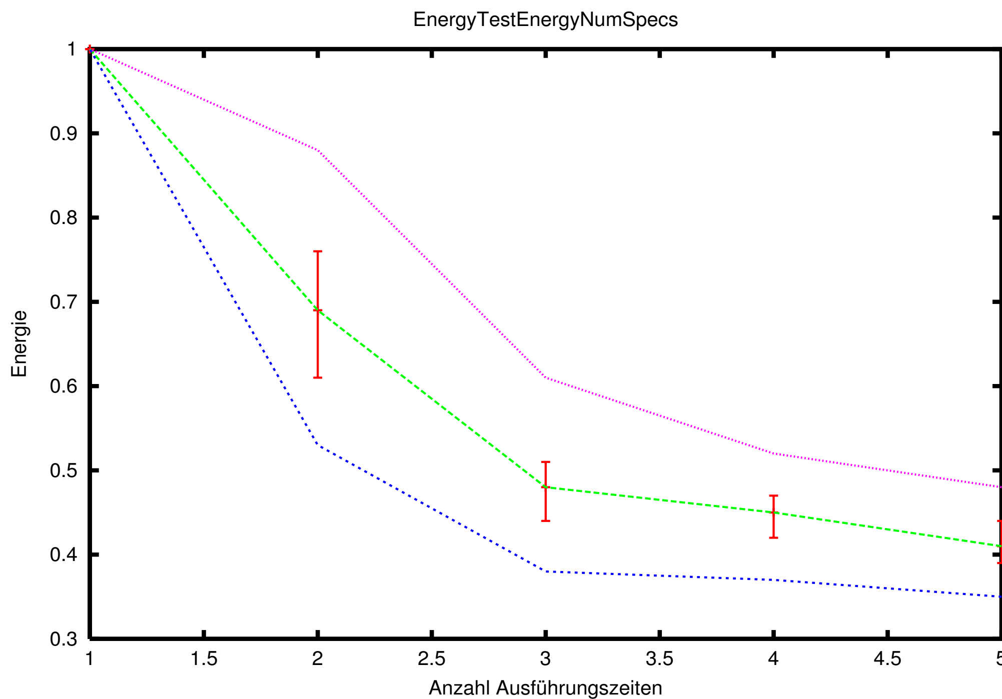

Abbildung 6.27: Anzahl der Ausführungszeiten je Methode → Energie

Abbildung 6.27 zeigt, daß eine Erhöhung der Anzahl der Ausführungsdauern auf drei große Energieeinsparungen ermöglicht. Bei einer weiteren Verfeinerung der Spezifikation der Ausführungsdauer ergeben sich jedoch nur geringe Einsparungen.

6.4.3 Minimale worst-case Auslastung

Diese Untersuchung zeigt die Entwicklung des Energieverbrauchs für Graphen mit unterschiedlicher minimaler worst-case Auslastung. Die Last nimmt die Werte 0.1, 0.2, . . . , 0.9 sowie 0.95 an. Mit dem Graphgenerator wurden 90 Graphen zufällig erzeugt. Die Anzahl der Instanzen liegt zwischen 6 und 20, jede Instanz besitzt drei Methoden mit drei Ausführungsdauern. Der Prozessor besitzt nur eine Leistungsstufe. Im Modifikationsschritt wurden die Periodendauern und der Energieverbrauch pro Zeiteinheit so angepaßt, daß die spezifizierte Last erzielt wurde. Die Ergebnisse der aus einem Graphen entstandenen Graphen bleiben dadurch vergleichbar.

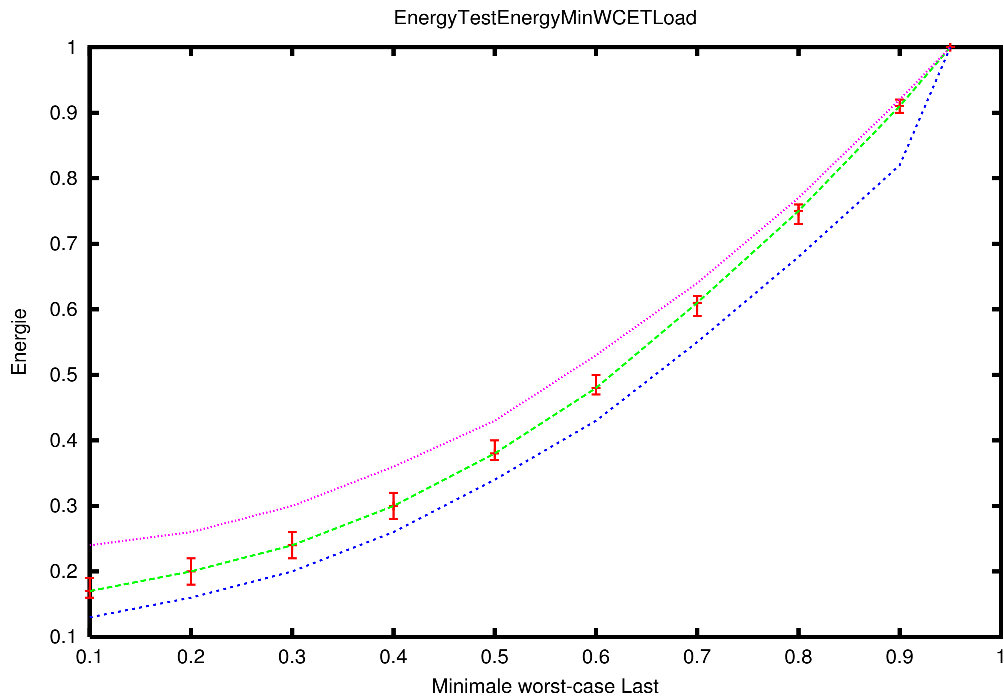

Abbildung 6.28: Minimale worst-case Auslastung → Energie

Abbildung 6.28 zeigt den Anstieg des Energieverbrauchs mit steigender Last. Da der Prozessor länger im Modus rechnend betrieben werden muß und zusätzlich weniger oft energiesparende Methoden gewählt werden können, ist die Steigerung des Energieverbrauchs überproportional zur Steigerung der Auslastung.

6.4.4 Verhältnis durchschnittliche Ausführungsdauer zu worst-case Ausführungsdauer

Diese Untersuchung zeigt die Entwicklung des Energieverbrauchs für Graphen mit unterschiedlichem Verhältnis von durchschnittlicher Ausführungszeit zu worst-case Ausführungszeit. Das Verhältnis nimmt die Werte 0.5, 0.6, 0.7, 0.8, 0.9 sowie 0.95 an. Mit dem Graphgenerator wurden 50 Graphen zufällig erzeugt. Die Anzahl der Instanzen liegt zwischen 6 und 20, jede Instanz besitzt drei Methoden. Der Prozessor besitzt nur eine Leistungsstufe. Im Modifikationsschritt wurden für jede Methode nur die kürzeste und die längste Ausführungszeit beibehalten und ihre Wahrscheinlichkeiten so angepaßt, daß der spezifizierte Wert erreicht wurde. Die Ergebnisse der aus einem Graphen entstandenen Graphen bleiben dadurch vergleichbar.

Abbildung 6.29 zeigt den Anstieg des Energieverbrauchs bei häufigerem Auf-

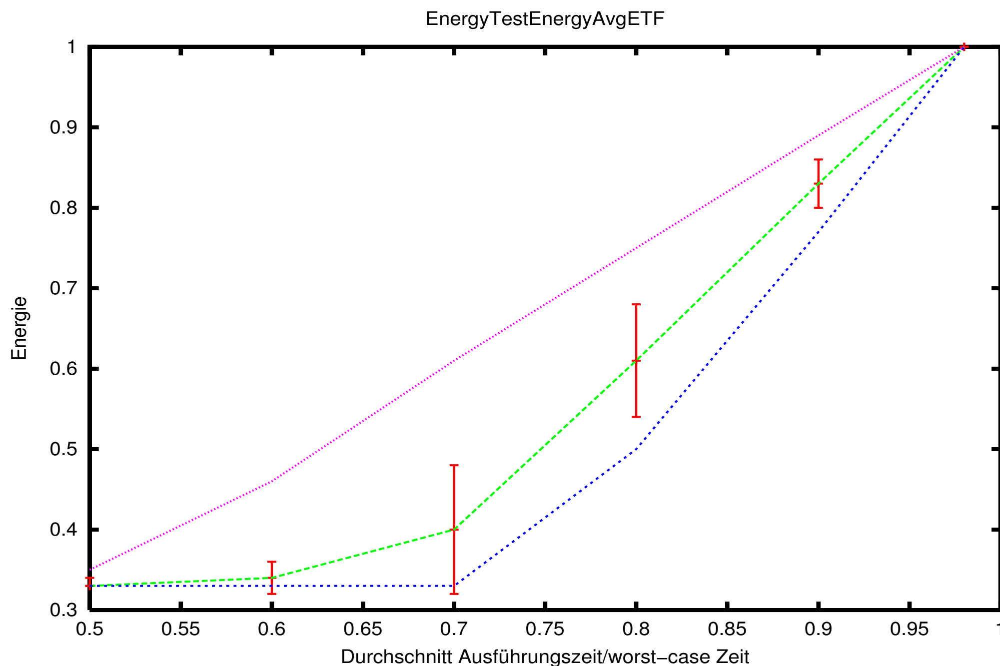

Abbildung 6.29: Durchschnittliche Ausführungszeit/worst-case Ausführungszeit → Energie

treten der worst-case Ausführungsdauer. Ab einem Wert von 0.7 ist der Anstieg annähernd linear, da der Energieverbrauch linear mit der Zeit in der der Prozessor rechnet steigt und die Zeit nur noch selten reicht, um energiesparende Methoden einzuplanen.

6.4.5 Anzahl der Leistungsstufen

Diese Untersuchung zeigt die Entwicklung des Energieverbrauchs für Graphen mit unterschiedlich vielen Prozessorleistungsstufen. Der Energieverbrauch wurde für 1, 2, 4 und 8 Leistungsstufen untersucht. Mit dem Graphgenerator wurden 25 Graphen mit jeweils 40 Instanzen zufällig erzeugt, die jeweils aus einer Kette von datenabhängigen Instanzen bestehen. Jede Instanz besitzt zwei Methoden, die wiederum bis zu zwei Werte für die Ausführungsdauer enthält. Im Modifikationsschritt wurde die Anzahl der Leistungsstufen auf den spezifizierten Wert gesetzt. Die Ergebnisse der aus einem Graphen entstandenen Graphen bleiben dadurch vergleichbar.

Abbildung 6.30 zeigt einen deutlichen Abfall des Energieverbrauchs bis zu vier Leistungsstufen. Eine weitere Verdopplung der Anzahl der Leistungsstufen ermöglicht hingegen fast keine zusätzlichen Energieeinsparungen.

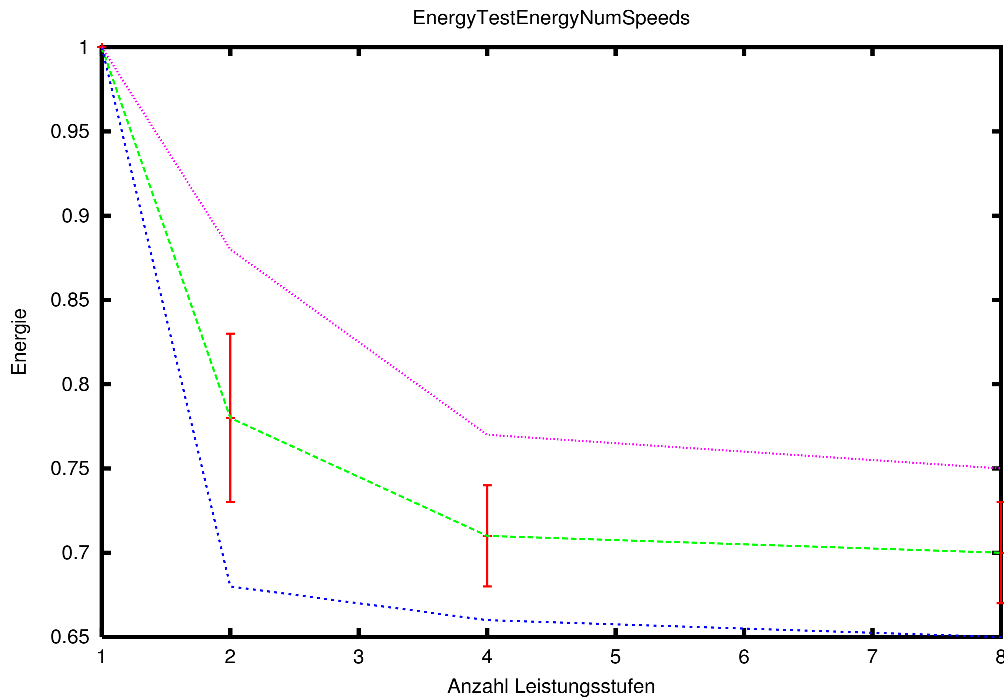

Abbildung 6.30: Anzahl Leistungsstufen → Energie

6.5 Anytime-Profile

Die folgenden Untersuchungen zeigen den Aufwand, der mit den Optimierern nötig ist, um einen bestimmten Bruchteil der maximal erzielbaren Qualität zu erreichen. Die Messungen bestätigen den angenommenen Verlauf der beiden Varianten des Optimierers. Während bei der anforderungsgetriebenen Dynamischen Programmierung die Konvergenz anfangs recht langsam verläuft, findet der Simulated Annealing Optimierer zu Beginn der Optimierung schnell gute Lösungen aber er benötigt sehr lange, um faßt optimale Lösungen zu finden. Die Messungen erfolgten mit 65 Graphen bei unterschiedlichen Knotenspeichergrößen. Die Graphen besitzen vier Methoden je Instanz und bis zu drei Ausführungsdauern je Methode. Die Einplanung erfolgte zunächst mit der anforderungsgetriebenen Dynamischen Programmierung, um die maximal erzielbare Qualität für jeden Graphen zu bestimmen. Bei den anschließenden Einplanungen mit Simulated Annealing wurde die Optimierung beim Erreichen von 95,5% der maximalen Qualität gestoppt.

Die in diesem Abschnitt enthaltenen Ergebnisse stammen aus der Untersuchung der Optimierungsdauer bei unterschiedlichen Knotenspeichergrößen. Ein Vergleich mit den in den anderen Untersuchungen ermittelten Profilen zeigt jedoch, daß die hier gezeigten Profile sehr ähnlich zu denen aus den anderen Unter-

suchungen sind.

6.5.1 Anforderungsgetriebene Dynamische Programmierung

Für die Bestimmung des Optimierungsaufwands wurden jeweils die benötige Anzahl von Knoten, die Anzahl der Knotenaktualisierungen und die Dauer der Optimierung gemessen. Durch die extremen Maximalwerte der Optimierungsdauer ist der Verlauf der Durchschnittswerte nur schwer zu erkennen. Für die Auswertungen wurden daher die Ergebnisse für die untersuchten Graphen so skaliert, daß die maximale Knotenzahl, Aktualisierungszahl und Optimierungsdauer jeweils den Wert 1 erhalten.

Optimierungsdauer für Qualitätsbruchteile

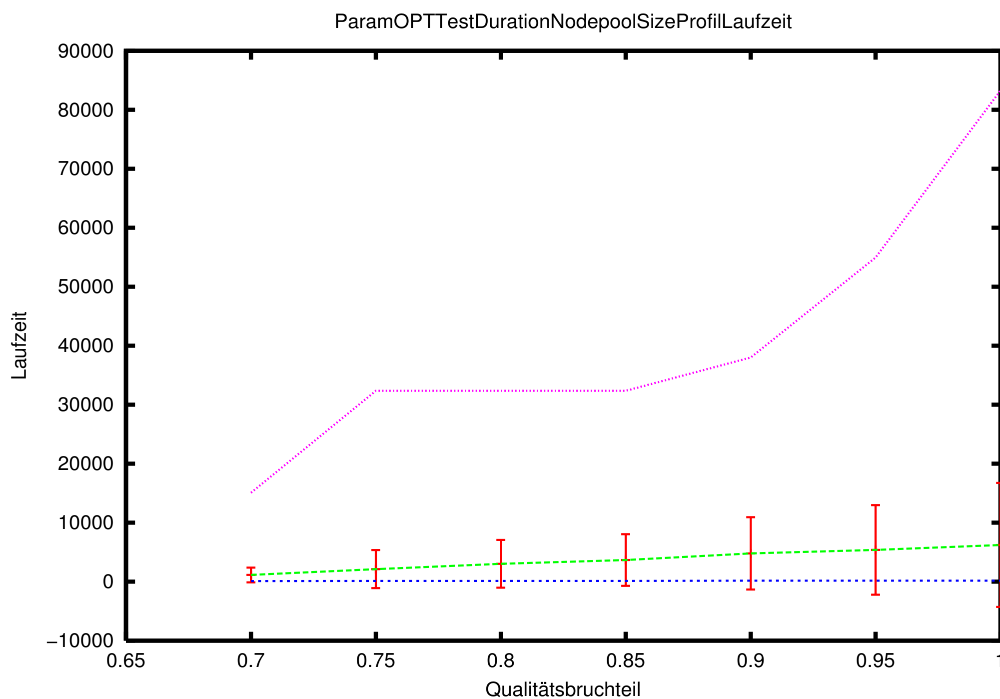

Abbildung 6.31: Qualitätsbruchteil → Optimierungsdauer in Millisekunden

Abbildung 6.31 zeigt die unskalierten Werte der Optimierungsdauer in Millisekunden. Anhand der Minimal-, Durchschnitts- und Maximalwerte ist deutlich die hohe Schwankung der absoluten Zeiten zu erkennen.

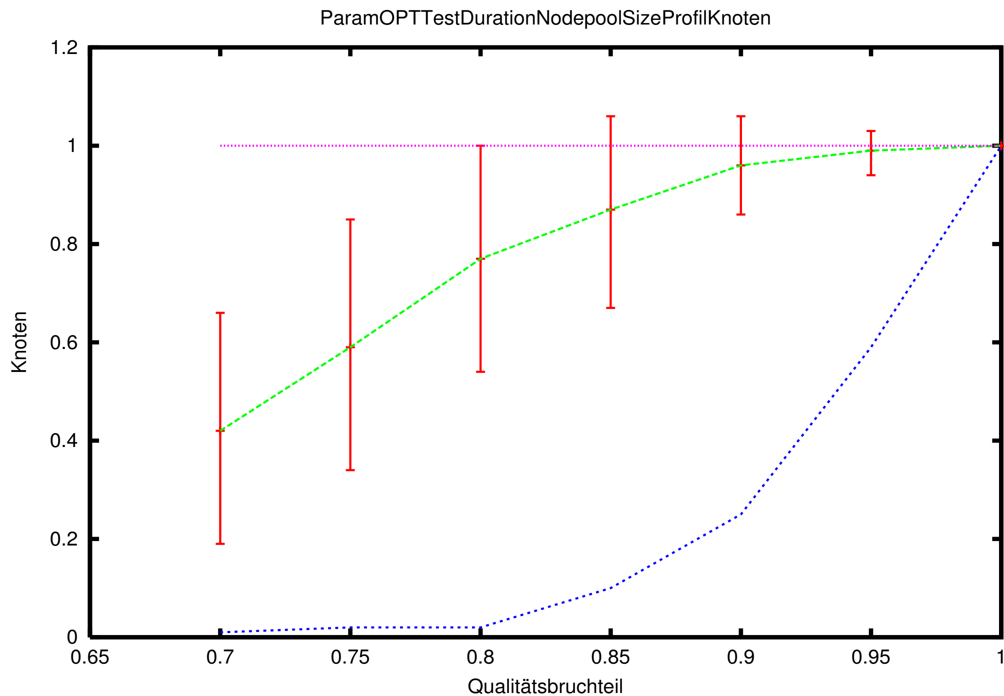

Abbildung 6.32: Qualitätsbruchteil → erzeugte Knoten

Anzahl der Knoten für Qualitätsbruchteile

In Abbildung 6.32 ist zu sehen, wie sich die Zahl der Knoten entwickelt, die für das Erreichen bestimmter Bruchteile der maximal möglichen Qualität nötig ist. Anfangs müssen für Qualitätssteigerungen relativ viele neue Knoten erzeugt werden. Im weiteren Verlauf der Optimierung können jedoch immer mehr bereits optimierte Teillösungen wiederverwendet werden, wodurch die die Konvergenz gegen Ende der Optimierung beschleunigt wird.

Anzahl der Aktualisierungen für Qualitätsbruchteile

Abbildung 6.33 zeigt, wie sich die Zahl der durchgeführten Knotenaktualisierungen entwickelt, die für das Erreichen bestimmter Bruchteile der maximal möglichen Qualität nötig ist. Da bei der anforderungsgetriebenen Dynamischen Programmierung nur in etwa eine[9] Aktualisierung je erzeugtem Knoten stattfindet, gleicht der Verlauf des Graphen dem der Knotenanzahl.

[9]Durch das für die Auswertung notwendige Monitoring der erreichten Qualität sind zusätzliche Aktualisierungen notwendig.

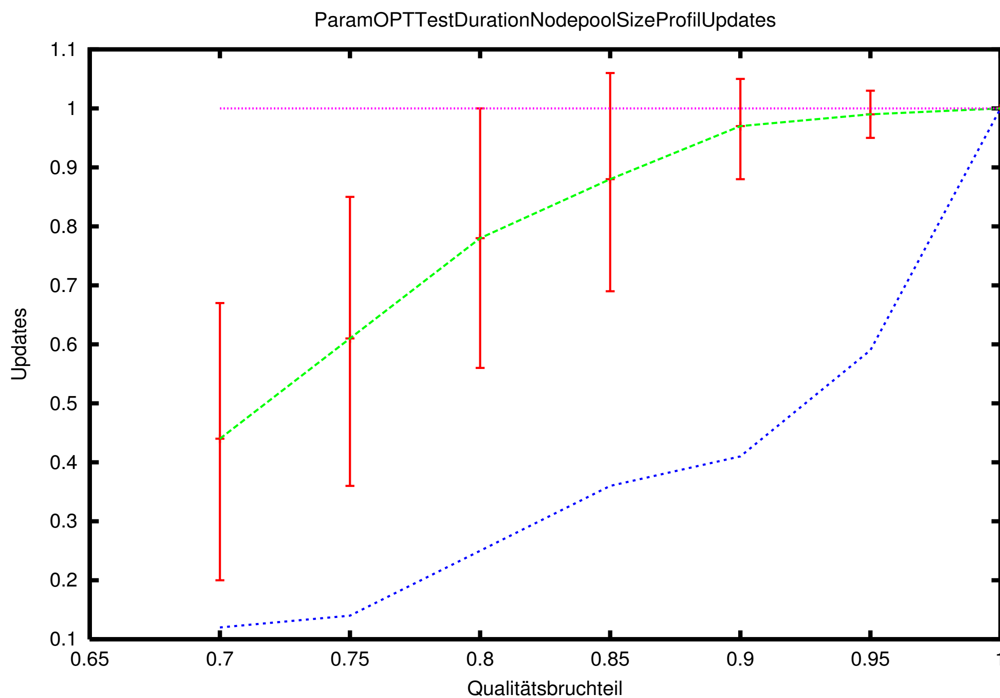

Abbildung 6.33: Qualitätsbruchteil → durchgeführte Aktualisierungen

Skalierte Optimierungsdauer für Qualitätsbruchteile

In Abbildung 6.34 ist die Entwicklung der Optimierungsdauer für das Erreichen bestimmter Qualitätsbruchteile aufgezeigt. Da die Anzahl der erzeugten Knoten ungefähr gleich der Anzahl der Aktualisierungen ist, ist auch die Entwicklung der Optimierungsdauer sehr ähnlich zur Entwicklung der notwendigen Knoten.

6.5.2 Simulated Annealing

Auch bei Simulated Annealing wurden jeweils die benötige Anzahl von Knoten, die Anzahl der Knotenaktualisierungen und die Dauer der Optimierung gemessen. Durch die ebenfalls extremen Maximalwerte der Optimierungsdauer ist der Verlauf der Durchschnittswerte auch hier nur schwer zu erkennen. Für die Auswertungen wurden daher wiederum die Ergebnisse für die untersuchten Graphen so skaliert, daß die maximale Knotenzahl, Aktualisierungszahl und Optimierungsdauer jeweils den Wert 1 erhalten.

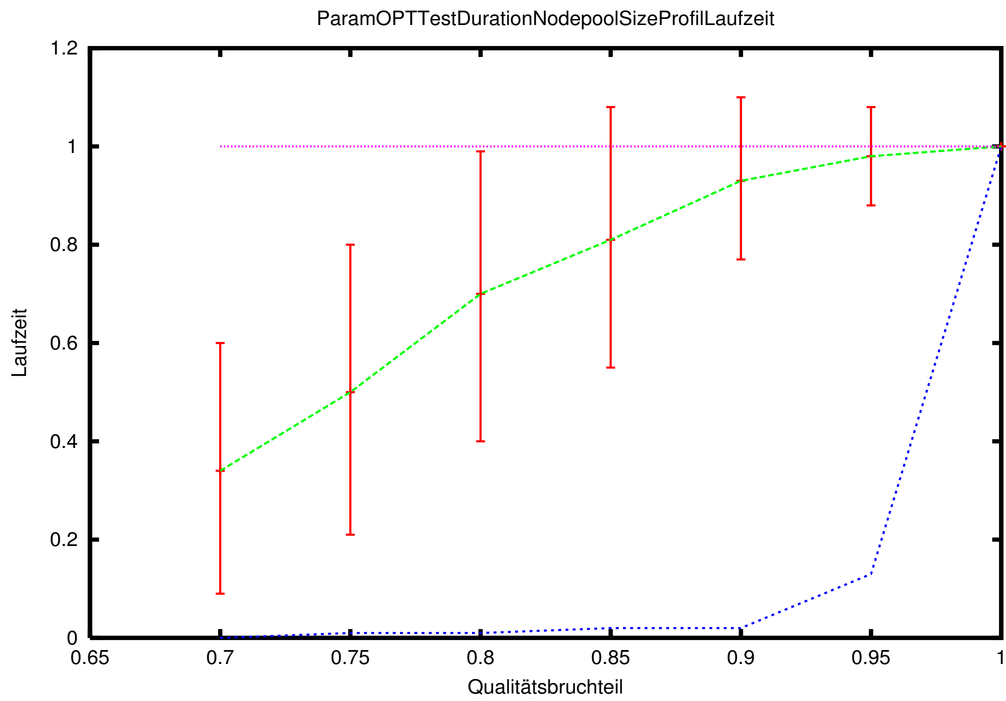

Abbildung 6.34: Qualitätsbruchteil → Optimierungsdauer

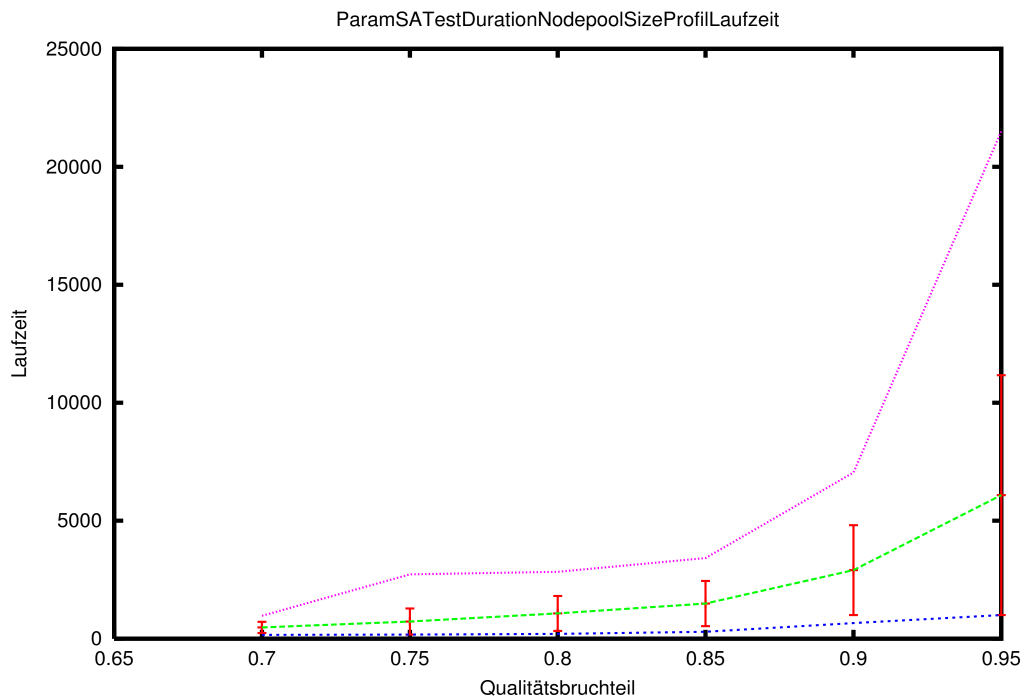

Abbildung 6.35: Qualitätsbruchteil → Optimierungsdauer in Millisekunden

Optimierungsdauer für Qualitätsbruchteile

Abbildung 6.35 zeigt die unskalierten Werte der Optimierungsdauer in Millisekunden. Anhand der Minimal-, Durchschnitts- und Maximalwerte ist deutlich die hohe Schwankung der absoluten Zeiten zu erkennen.

Anzahl der Knoten für Qualitätsbruchteile

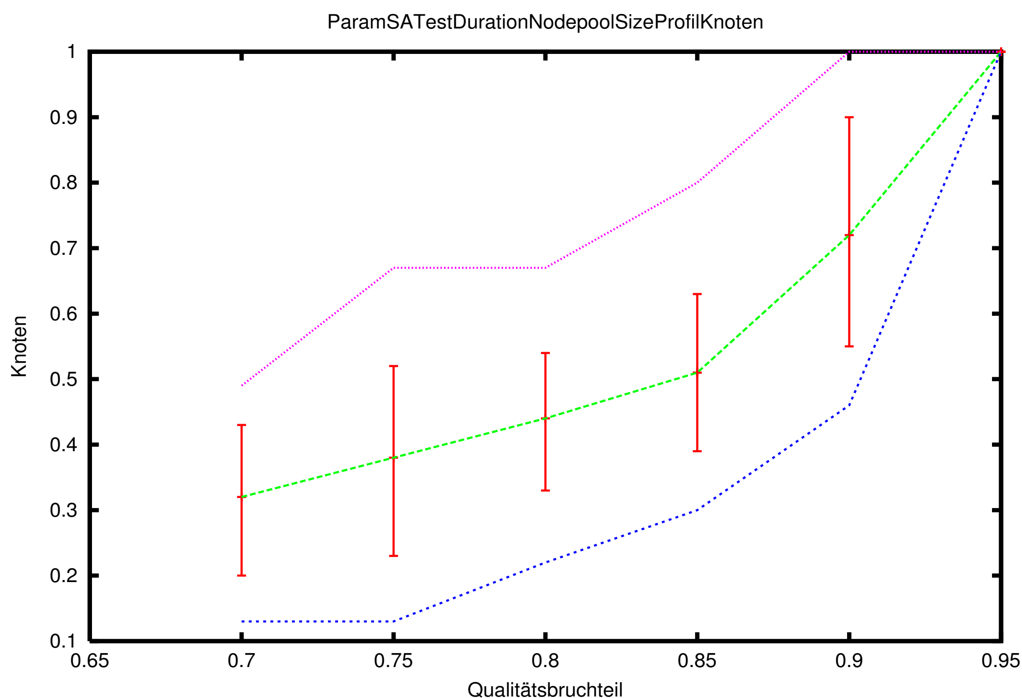

Abbildung 6.36: Qualitätsbruchteil → erzeugte Knoten

In Abbildung 6.36 ist zu sehen, wie sich die Zahl der Knoten entwickelt, die für das Erreichen bestimmter Bruchteile der maximal möglichen Qualität nötig ist. Der Verlauf ist hier entgegengesetzt zu dem der anforderungsgetriebenen Dynamischen Programmierung. Aufgrund der zufälligen Abarbeitung des Suchraumes bei Simulated Annealing ist der die Zahl der für gute Lösungen benötigten Knoten relativ gering. Für das Finden (fast) optimaler Lösungen hingegen steigt die Zahl der benötigten Knoten sehr stark an, da solche Lösungen nur mit einer geringen Wahrscheinlichkeit untersucht werden.

Anzahl der Aktualisierungen für Qualitätsbruchteile

Abbildung 6.37 zeigt, wie sich die Zahl der durchgeführten Knotenaktualisierungen entwickelt, die für das Erreichen bestimmter Bruchteile der maximal mögli-

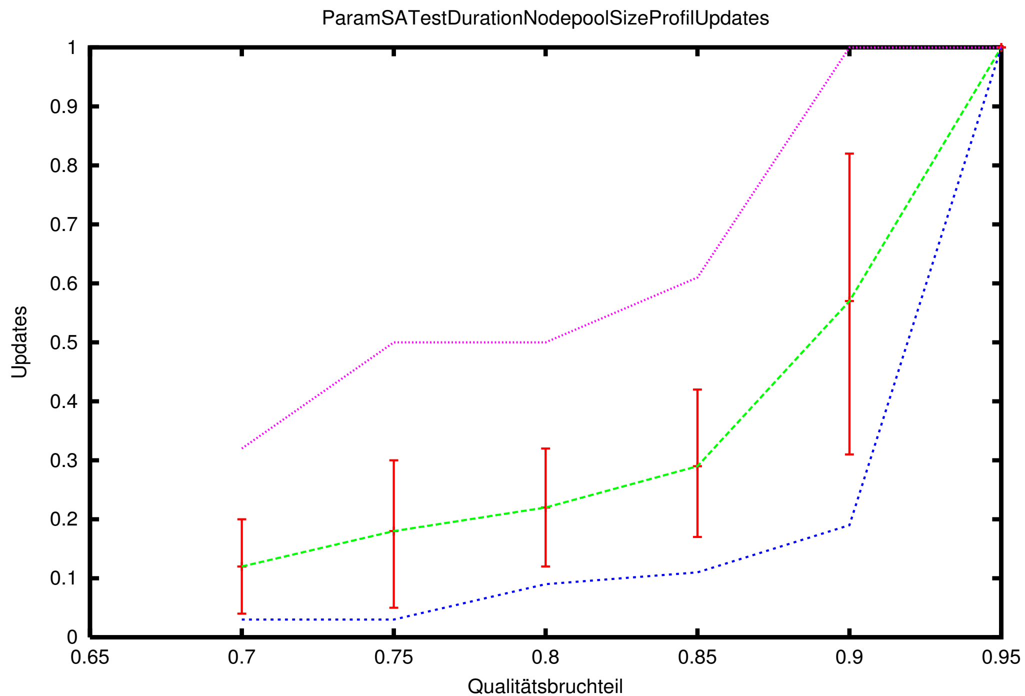

Abbildung 6.37: Qualitätsbruchteil → durchgeführte Aktualisierungen

chen Qualität nötig ist. Da bei Simulated Annealing aufgrund der zufälligen Änderung von Knoten mehrere Aktualisierungen je erzeugtem Knoten stattfinden, verläuft der Graph der Aktualisierungen steiler als der der erzeugten Knoten.

Skalierte Optimierungsdauer für Qualitätsbruchteile

In Abbildung 6.38 ist die Entwicklung der Optimierungsdauer für das erreichen bestimmter Qualitätsbruchteile aufgezeigt. Da die Anzahl der erzeugten Knoten von der Anzahl der Aktualisierungen dominiert wird, ist auch die Entwicklung der Optimierungsdauer sehr ähnlich zur Entwicklung der notwendigen Aktualisierungen.

Die Untersuchung aus [Ram04] zeigt die möglichen Einergieeinsparungen bei der Erweiterung des Systemmodells um Leistungsstufen, alternative Methoden und Ausführungszeitverteilungen. Für die Untersuchung wurden die Durchschnittswerte des Energieverbrauchs für 10 Graphen mit bis zu zwei Methoden je Instanz und jeweils zwei Ausführungsdauern je Methode berechnet und aufgetragen. Die vier Säulen `Methode A` stehen für eine zufällige aber feste Auswahl einer Methode je Instanz, die Säulen `Methode B` für die feste Auswahl der jeweils anderen Methode. Bei den Säulen `Methoden A+B` blieben beide Methoden im Systemmodell erhalten. `Methode A` und `Methode B` sollen den Einfluß der

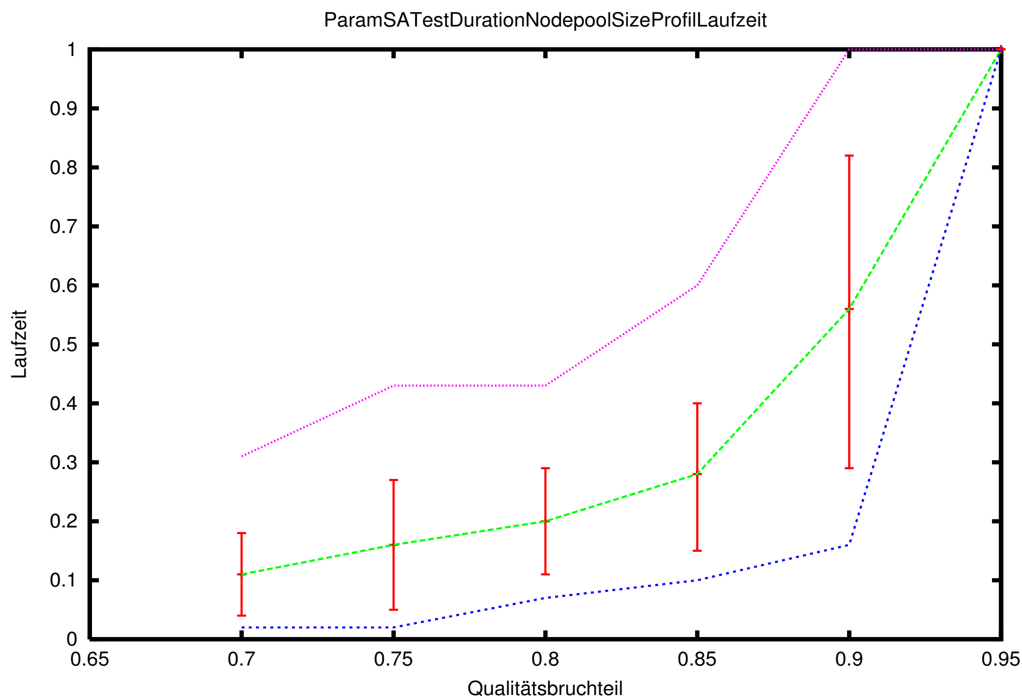

Abbildung 6.38: Qualitätsbruchteil → Optimierungsdauer

Methodenauswahl durch den Entwickler einer Anwendung darstellen, während `Methode A+B` die automatische, dynamische Auswahl durch den in dieser Arbeit entwickelten Optimierer erlaubt. Die Spalte `WCET` steht für Systemmodelle in denen nur jeweils die worst-case Ausführungsdauern der Methoden spezifiziert sind, während die Modelle in den Spalten `Verteilung` jeweils eine weitere Ausführungsdauer je Methode beinhalten. Die vordere Reihe der Säulen enthält Systemmodelle die Prozessoren mit zwei Leistungsstufen besitzen, die hintere Reihe solche mit nur einer Leistungsstufe.

Abbildung 6.39 zeigt, daß die Einsparungen durch die automatische Methodenauswahl immer am größten sind. Bei der manuellen Auswahl sind die Einsparungen sehr stark vom Geschick bzw. der Erfahrung des Anwendungsentwicklers abhängig. Verfeinerungen der Spezifikation der Ausführungsdauer führen nur in Kombination mit mehreren Leistungsstufen zu Einsparungen. Die Einführung einer zweiten Leistungsstufe ermöglicht jedoch immer Energieeinsparungen. Die größten Einsparungen ergeben sich durch die Kombination der automatischen Methodenauswahl mit Ausführungszeitverteilungen und mehreren Leistungsstufen.

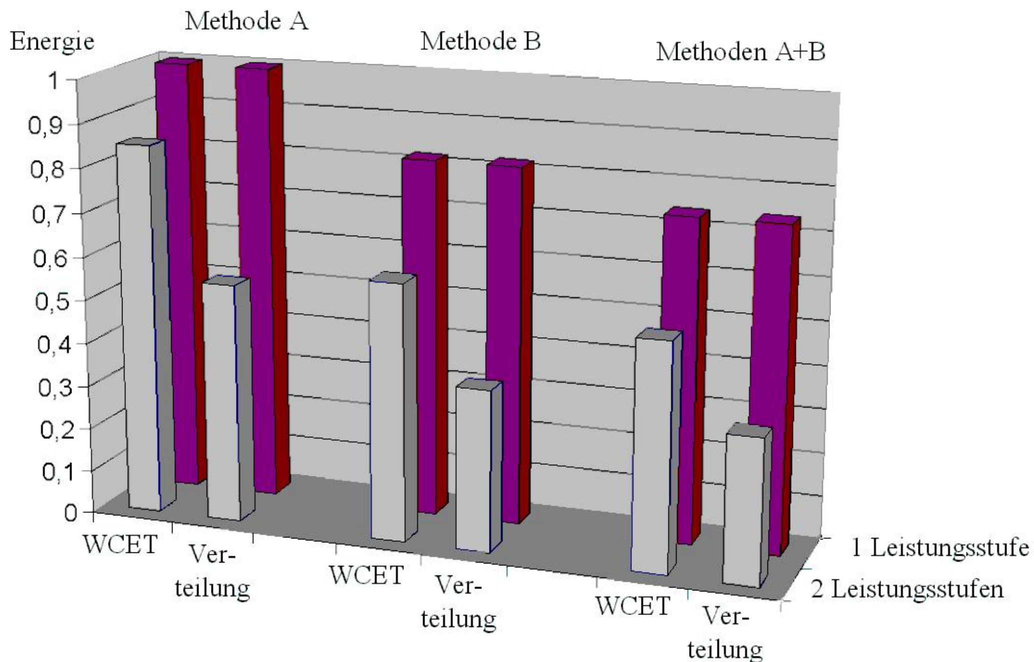

Abbildung 6.39: Energieeinsparungen durch erweitertes Modell

6.6 Vergleich mit anderen energiebewußten Schedulern

Die folgenden beiden Untersuchungen vergleichen die Energieeinsparungen anderer Arbeiten mit den Einsparungen, die diese Arbeit ermöglicht. Die Vergleiche wurden mit einem Systemmodell durchgeführt, das zu allen Arbeiten kompatibel ist, dadurch aber leider relativ einfach gehalten werden mußte. Die Untersuchungen zeigen jedoch, daß der in dieser Arbeit entwickelte Optimierer die Energieeinsparmöglichkeiten optimal ausnutzen kann. Hierbei muß aber erwähnt werden, daß die anderen Arbeiten teils flexiblere oder unflexiblere Systemmodelle umfassen und sie daher oft als laufzeiteffiziente Heuristiken entwickelt wurden, während der hier vorgestellte Optimierer sehr viel Zeit für die Offline-Optimierung benötigt.

Die Ergebnisse des in dieser Arbeit entwickelten Algorithmus - in diesem Abschnitt mit `MRDtT` bezeichnet - stellen jeweils den berechneten zu *erwartenden Energieverbrauch* dar. Die Ergebnisse für die anderen Arbeiten wurden durch die Mittelung des *gemessenen Energieverbrauchs* während mehrerer Simulationsläufe bestimmt.

Abbildung 6.40 zeigt die Entwicklung des Energieverbrauchs bei der Erhöhung der Leistungsstufenanzahl des Prozessors. Der Vergleich wurde für die Algorith-

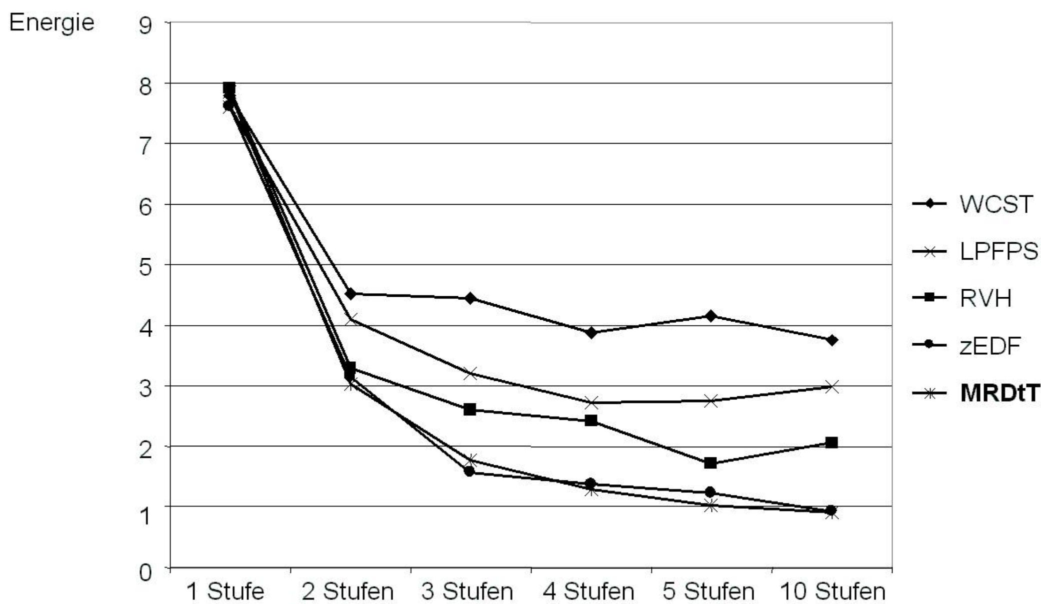

Abbildung 6.40: Energieverbrauch bei unterschiedlich vielen Leistungsmodi

men `WCST` und `RVH` [LS00], `LPFPS` [SC99], `zEFD` [PS01] und `MRDtT` durchgeführt. Aufgrund der vollständigen Optimierung vor der Anwendungsausführung kann der hier vorgestellte Algorithmus `MRDtT` die Leistungsstufen besser nutzen als die zum Vergleich herangezogenen Algorithmen.

Abbildung 6.41 zeigt die durchschnittlichen/erwarteten Energieverbräuche für die Algorithmen `WCST` [LS00], `LPFPS` [SC99], `RVH` [LS00], `sRM` [PS01], `sEDF` [PS01], `dRM` [PS01], `dEDF` [PS01], `zEDF` [PS01] und `MRDtT`. Auch hier ist die optimale Energieeinsparung durch die Offline-Optimierung bei `MRDtT` zu erkennen.

Ein Vergleich mit den Design-to-time-Schedulern von Decker, Garvey und Lesser [Dec96, GL93, GL96, GL95] war aufgrund der nicht zugänglichen Implementierung und der für eine Nachimplementierung zu ungenauen Dokumentation leider nicht möglich.

6.7 Bewertung der Untersuchungen

Die Untersuchungen haben die positiven Auswirkungen der Zusammenführung der Modelle mit nicht-deterministischen Ausführungszeiten, Methodenauswahl und diskreten Leistungsstufen auf die qualitäts- und energiebewußte Einplanung von harten Echtzeitanwendungen gezeigt. Bei der Modellierung von Anwendun-

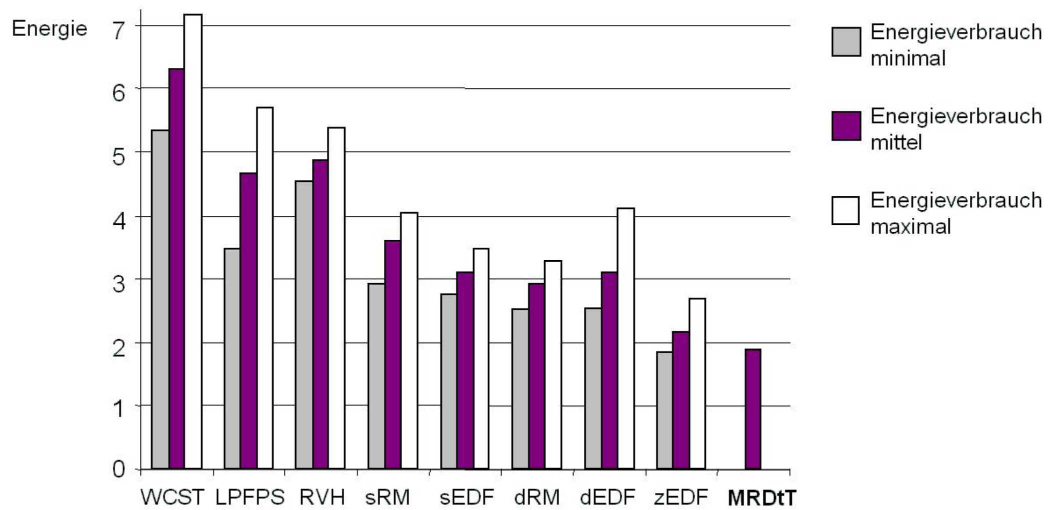

Abbildung 6.41: Energieverbrauch verschiedener Scheduling-Verfahren

gen ist es ausreichend, nur etwa drei Methoden je Instanz, drei Ausführungsdauern je Methode und bis zu vier Leistungsstufen zu verwenden. Diese Werte schränken die Optimierungsdauer stark ein und liefern dennoch fast optimale Resultate.

Die erwarteten Auswirkungen der Modelleigenschaften konnten durch die Untersuchungen zur Optimierungsdauer gestützt werden. Auch der grundlegende Verlauf der Optimierung (Anytime-Profil) bei den beiden Optimierungsalgorithmen anforderungsgetriebene Dynamische Programmierung und Simulated Annealing wurde bestätigt. Die anforderungsgetriebene Dynamische Programmierung konvergiert anfangs langsam gegen eine optimale Lösung, weil in diesem Stadium noch viele Teilprobleme nicht optimal gelöst sind. Gegen Ende der Optimierung wird die Konvergenz jedoch durch die bereits optimierten Teillösungen beschleunigt. Das gegenteilige Verhalten zeigt der Simulated Annealing-Ansatz. Hier werden durch die zufällige Durchwanderung des Suchraumes bereits früh gute Lösungen gefunden, aber die Konvergenzgeschwindigkeit sinkt im Verlauf der Optimierung. Die ermittelten absoluten Optimierungsdauern für die untersuchten Graphen legen den Einsatz der anforderungsgetriebenen Dynamischen Programmierung für Graphen mit weniger als 200 Instanzen und vielen Datenabhängigkeiten nahe, bei größeren Graphen bietet das Simulated Annealing-Verfahren aufgrund der anfangs höheren Konvergenz Vorteile.

Kapitel 7

Zusammenfassung

Die Arbeit befaßt sich mit der energie- und qualitätsbewußten Einplanung von Echtzeitanwendungen in eingebetteten Systemen. Sie stellt eine Zusammenführung eines auf mehreren diskreten Leistungsstufen basierenden Prozessormodells mit dem Design-to-time-Modell in einer um diskrete Laufzeitverteilungen erweiterten Form vor. Das Modell umfaßt periodische Prozesse mit Datenabhängigkeiten, Bereitzeiten und Fristen. Für jeden Prozeß können mehrere alternative Methoden angegeben werden, die sich hinsichtlich ihrer Ausführungszeitverteilungen und der von ihnen gelieferten Qualität oder verbrauchten Energie unterscheiden. Für die beiden Optimierungsziele der Minimierung des durchschnittlichen Energieverbrauchs und der durchschnittlich gelieferten Qualität eines zulässigen Schedules wurde ein modular aufgebauter Optimierungsalgorithmus entwickelt, der die Eigenschaften eines Anytime-Algorithmus besitzt, d.h. er liefert eine Folge von besser werdenden Lösungen und er kann jederzeit abgebrochen werden, wobei dann die beste bis dahin berechnete Lösung verfügbar ist.

Die energiebewußte Einplanung nutzt mehrere Möglichkeiten zur Energieeinsparung. Durch das vorgestellte Modell werden erwartete Ausführungszeitschwankungen bereits während der statischen Optimierung berücksichtigt und genutzt. Ebenso können unterschiedliche Implementierungen für einen Prozeß spezifiziert werden, die aufgrund unterschiedlichen Zeit- und Energiebedarfs für Einsparungen genutzt werden können. Auch die Möglichkeit weniger Energie zu verbrauchen, indem der Prozessor in eine niedrigere Leistungsstufe versetzt wird, wird genutzt. Bei der qualitätsbewußten Einplanung werden die modellierten Ausführungszeitschwankungen und die von den modellierten Methoden gelieferten unterschiedlichen Qualitäten genutzt, um die durchschnittliche gelieferte Qualität zu verbessern.

Die komplexe, zeit- und energieintensive Optimierung erfolgt nur einmal vor der Ausführung der Anwendung und sie kann auf jedem Java-fähigen Rechner durchgeführt werden. Sie ist daher nicht an die Rechenleistung und den Energie-

vorrat des Zielsystems gebunden.

Die Architektur des Optimierers erlaubt die Verwendung anderer energie- und qualitätsbewußter Schedulingalgorithmen als Lotsen. Die Initiallösung entspricht dann den von diesem Algorithmen erzeugten Schedules, sie kann jedoch weiter optimiert werden und der minimale, der zu erwartende und der maximale Zielfunktionswert sind bereits vor der Anwendungsausführung bekannt. Für die Optimierung der Bedingungsgraphen stehen eine auf Dynamischer Programmierung und eine auf Simulated Annealing basierende Variante zur Verfügung. Die erste Variante bietet eine deterministische Suche nach einem optimalen Bedingungsgraphen und aufgrund der vollständigen Abarbeitung des Suchraumes verifiziert sie auch die globale Optimalität der gelieferten Lösung. Neben ihrer Anwendung auf gemäßigt komplexe Anwendungsmodelle dienen die von ihr gefundenen Ergebnisse als Referenzwerte bei Benchmarks zur Bewertung der zweiten Variante. Simulated Annealing ist aufgrund der probabilistischen Suchreihenfolge darauf ausgerichtet für komplexe Anwendungsmodelle schnell gute Bedingungsgraphen zu liefern. Das Auffinden einer optimalen Lösung hingegen kann nur mit Wahrscheinlichkeit 1 bei unendlicher Laufzeit garantiert werden. Beide Varianten werden durch den Knotenspeicher unterstützt, der die Verwaltung der Knoten der Bedingungsgraphen übernimmt. Er speichert automatisch die besten gefundenen Teillösungen, die jederzeit angefordert werden können, und stellt dadurch die Anytime-Funktionalität zu Verfügung. Das Zusammenführen der besten Teillösungen beschleunigt die Konvergenz des Simulated Annealing Verfahrens.

Die statische Optimierung der Bedingungsgraphen erlaubt die Implementierung eines sehr effizienten, dynamischen Schedulers. Der Scheduler interpretiert den Bedingungsgraphen zur Laufzeit mit vernachlässigbarem bzw. während der Optimierung berücksichtigtem Zeit- und Energieaufwand.

Die an Zufallsanwendungsmodellen durchgeführten Untersuchungen zeigen, das bei der energiebewußten Einplanung alle drei im Modell enthaltenen Effekte Einsparungen ermöglichen. Die größten Einsparungen werden erzielt, wenn alle Effekte gemeinsam genutzt werden. Ebenso kann durch die Modellierung der nicht-deterministischen Ausführungszeiten und durch die dynamische Methodenauswahl die Qualität gesteigert werden, die von einer qualitätsbewußten Anwendung unter vorgegebenen Zeitbedingungen geliefert wird. Die Experimente bestätigen den berechneten Verlauf des Optimierungsaufwandes durch variierende Modelleigenschaften, wie etwa Anzahl der Instanzen oder Anzahl der Zusammenhangskomponenten. Auch zeigen die Untersuchungen, daß es bei den vom Anwendungsentwickler beeinflußbaren Modelleigenschaften Anzahl der Methoden je Instanz, Anzahl der Ausführungsdauern je Methode und Anzahl der Leistungsstufen ausreicht etwa drei Werte zu spezifizieren, um fast optimale Ergebnisse zu erhalten und die Optimierungsdauer zu verringern.

Der vorgestellte zweiphasige Algorithmus ist durch die Abtrennung der auf-

wendigen Optimierung von der dynamischen Einplanung besonders für eingebettete Echtzeitsysteme geeignet, bei denen es auf die effiziente Nutzung der Ressourcen Rechenleistung und Energie ankommt. Das Konzept der flexiblen Schedules ermöglicht die Implementierung eines dynamischen Schedulers, dessen Zeit und Energiebedarf bereits bei der Optimierung der Anwendung berücksichtigt werden kann. Aufgrund der Modellierung der nicht-deterministischen Ausführungszeiten der Methoden kann er jedoch trotz statischer Optimierung die gewonnene Flexibilität zur Steigerung der erzielten Qualität oder zur Verringerung der benötigten Energie ausnutzen.

Die Praktikabilität des zweiteiligen Ansatzes der statischen Optimierung und der dynamischen Einplanung wurde in einer Fallstudie mit Fußball spielenden Robotern gezeigt (siehe Anhang A). Dazu wurde die existierende Software modelliert und es wurde ein Werkzeug entwickelt, um die Ausführungszeiten der implementierten Methoden einfach messen zu können. Die Roboteranwendung wird als einzige Echtzeittask in einem mit RTAI um Echtzeitfähigkeiten erweiterten Linuxsystem ausgeführt. Die Anwendung erhält dadurch während ihrer Ausführung exklusiven Zugriff auf den Prozessor und sie führt das Scheduling der internen Prozesse selbst aus. Anhang B enthält eine Erweiterung der Bedingungsgraphen für die Einplanung auf Mehrprozessorsystemen. Anhang C stellt den theoretischen Zusammenhang zwischen der Taktfrequenz und dem Leistungsverbrauch bei CMOS-Schaltungen vor, und Anhang D zeigt, wie die mögliche Qualität bzw. der mögliche Energieverbrauch eines Systemmodells mit CPLEX abgeschätzt werden können. Die Anhänge E und F beschreiben die Integration der hier vorgestellten Algorithmen in das am Lehrstuhl für Rechnerstrukturen entwickelte Framework PASCHA.

Anhang A

Fallstudie RoboCup

Die Praktikabilität des hier vorgestellten Ansatzes wird durch den Einsatz im Projektstudium „RoboCup“ gezeigt. Das Kapitel beginnt mit einem Abschnitt über den für diese Fallstudie verwendeten Ablauf zur Datengewinnung, Optimierung und dem Einsatz des Algorithmus. Der zweite Abschnitt enthält Details über die entwickelte qualitätsbewußte Anwendung.

A.1 Allgemeiner Ablauf

Das in diesem Rahmen entwickelte, einfache Werkzeug [RCH04] hat auch das Ziel, die Akzeptanz und Anwendbarkeit des Algorithmus in folgenden Projektstudien zu erleichtern und zu fördern. Für Projekte, bei denen die Hardware noch nicht festgelegt ist, oder bei denen die Methodenlaufzeiten nicht gemessen werden können, ist der hier vorgestellte Ansatz nicht einsetzbar. In solchen Fällen können Werkzeuge wie z.B. [MCKT04] oder [SE04] verwendet werden, um die benötigten Daten zu bestimmen.

Die Ziele für das in dieser Fallstudie verwendete Werkzeug waren:

1. einfache, wahlweise graphisch unterstützte Modellierung der Anwendung
2. Vorgabe geeigneter Standardparameter für den Optimierer, bzw. leichte, verständliche Parametrisierbarkeit
3. Automatisierung des Optimiervorganges
4. Verwendung eines flexiblen, etablierten Eingabeformates
5. Plattformunabhängigkeit des Werkzeugs, d.h. kein Zwang die Optimierung auf der Zielplattform berechnen zu lassen
6. Erweiterbarkeit auf andere Betriebssysteme

7. Unterstützung bei der Ermittlung der benötigten Meßwerte

Voraussetzung (1) wird durch die wahlweise Modellierung der Anwendung mit einer XML-Datei (4) oder mit Hilfe des graphischen Editors aus dem Rahmenwerk PASCHA (siehe Anhang E) erfüllt. Die Parameter des Optimierers (2) wurden so eingestellt, daß sie für viele Anwendungen geeignet sind. Der Optimierungsvorgang wurde aus dem Rahmenwerk PASCHA extrahiert und in ein eigenes Programm, das mit neueren Versionen des Optimierers aktualisiert werden kann, gekapselt (3). Es ist in Java implementiert, um auf vielen Systemen (5) einsetzbar zu sein, und es erzeugt übersetzbaren Quelltext für das ausgewählte Echtzeitbetriebssystem. Die für die Quelltexterzeugung nötigen Dateien wurden für das Betriebssystem Linux mit der Echtzeiterweiterung RTAI bereits erstellt. Für den Einsatz anderer Betriebssysteme sind diese Dateien, die z.B. die systemspezifischen Anweisungen für die Allokation von Hauptspeicher enthalten, neu zu erstellen (6). Einer der wichtigsten Punkte ist jedoch die Unterstützung des Entwicklers bei der Gewinnung der benötigten Meßwerte (7). Ohne Automatisierung ist der Aufwand hierfür beachtlich, weil die Messungen sowohl für neue Algorithmen, die in der Anwendung eingesetzt werden, als auch bei jeder Änderung der Implementierung eines solchen Algorithmus erneut durchgeführt werden müssen.

A.1.1 Modellbildung

Abbildung A.1 zeigt das Datenflußdiagramm des Werkzeuges. Für die Bildung des Anwendungsmodells wird der äußere Zyklus (gestrichelt) des Datenflußdiagramms durchlaufen.

Der Anwendungsentwickler erstellt eine *Konfiguration* im XML-Format, die Informationen über die in der Anwendung enthaltenen Prozesse, deren Echtzeitbedingungen und die für sie verwendbaren Methoden beinhaltet. Abbildung A.2 zeigt einen Ausschnitt einer Konfigurationsdatei.

Die Schlüsselworte `<RNT>...</RNT>` umschließen die Spezifikation eines periodischen Prozesses. Sie enthält

- seinen Namen `<Name>...</Name>`,
- die Periodendauer `<Period>...</Period>`,
- seine Bereitzeit `<ReleaseTime>...</ReleaseTime>`,
- seine Frist `<Deadline>...</Deadline>`,
- seine Datenabhängigkeiten `<Dependency>...</Dependency>`
- und die verwendbaren Methoden `<RNM>...</RNM>`.

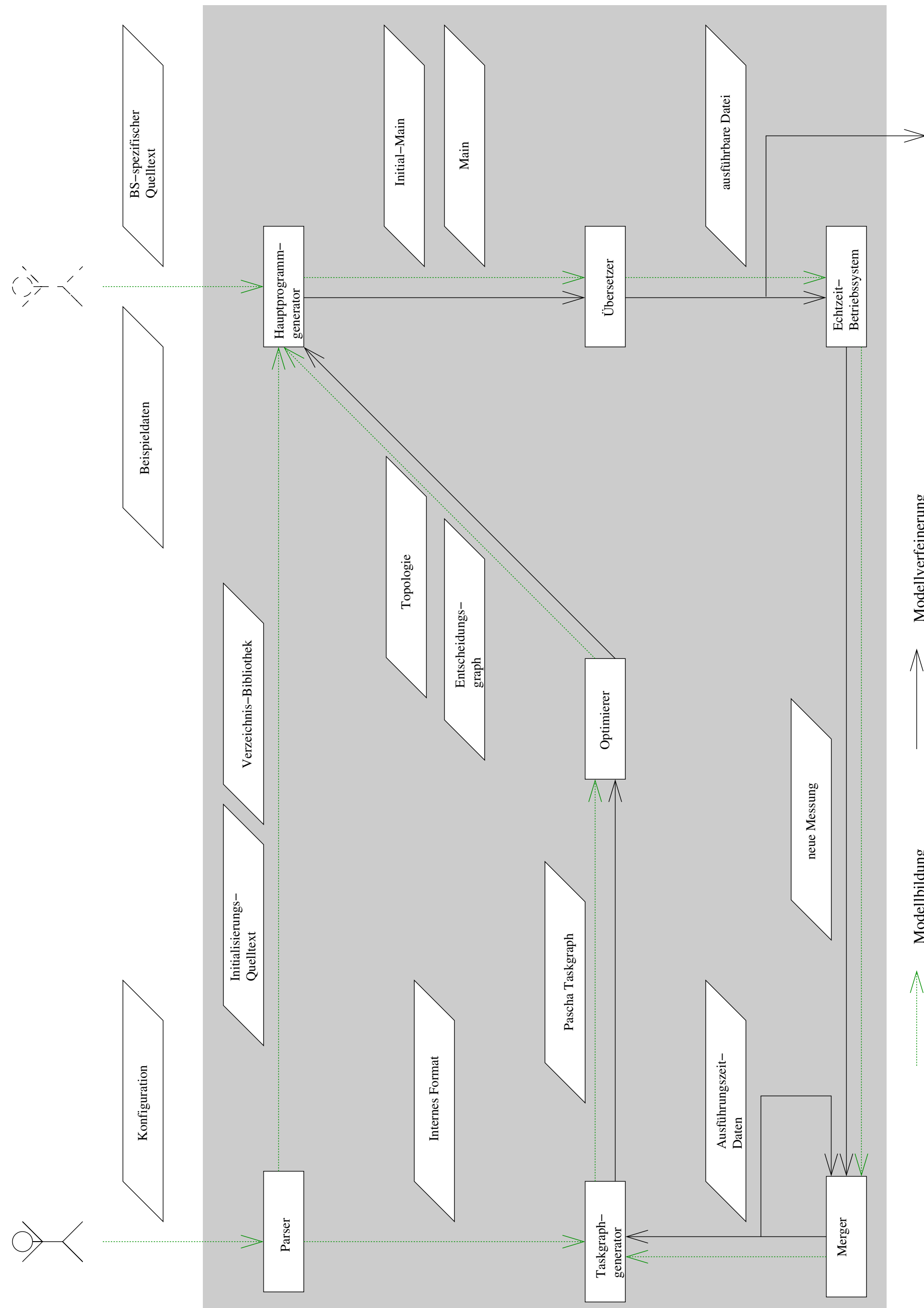

Abbildung A.1: Datenflußdiagramm des Werkzeuges für die Anwendungsoptimierung

Die Spezifikationen von Methoden enthalten

- ihren Namen `<Name>...</Name>`,
- die von ihnen gelieferte Qualität `<Quality>...</Quality>`
- sowie den Quelltext für ihren Aufruf `<CallCode>...</CallCode>`.

Eine weitere Möglichkeit zur Spezifikation einer Anwendung ist die Verwendung von PASCHA-Taskgraphen anstatt der Konfigurationsdatei, die mit dem graphischen Editor des PASCHA-Rahmenwerks erstellt und geändert werden können. Im Rahmen der Fallstudie und für den Prototypen des Werkzeugs ist jedoch die direkte Erstellung der Konfiguration vorteilhaft, da dies mit einem einfachen Texteditor bewerkstelligt werden kann, der auf jedem Rechner verfügbar ist und Änderungen nur selten vorzunehmen sind.

Falls für einige im Modell enthaltenen Methoden noch keine Ausführungszeitverteilungen existieren, muß der Anwendungsentwickler für alle Prozesse ohne eingehende Datenabhängigkeiten Beispieldaten angeben, um die automatische Messung der Verteilungen zu ermöglichen. Falls die Anwendung auf einem Betriebssystem ausgeführt werden soll für das noch keine Datei mit dem *betriebssystemspezifischen Quelltext* existiert, muß der Anwendungsentwickler auch diese erstellen.

Anschließend startet er das Werkzeug, das die nun folgenden Schritte automatisch durchführt (Abbildung A.1). Die Konfiguration wird in ein *internes Format*, das die Daten zu den Prozessen und Methoden enthält, und in den *Initialisierungsquelltext* und eine *Verzeichnisbibliothek*, die die Pfade zu den Implementierungen der Methoden, aufgetrennt. Der Taskgraph-Generator wandelt das *interne Datenformat* in einen *PASCHA-Taskgraphen* ohne Ausführungszeitinformationen um. Aus diesem Graph generiert das Werkzeug einen Reihenfolgegraph (*Topologie*) für die einzuplanenden Prozeßinstanzen, aus der dann durch den Hauptprogrammgenerator zusammen mit den anderen bereits erzeugten Dateien eine *initiale Main-Methode* (z.B. C-Code) erstellt wird. Die initiale Main-Methode wird übersetzt und mit den vom Benutzer spezifizierten *Beispieldaten* auf dem Zielsystem ausgeführt. Während dieser Ausführung wird eine *neue Messung* der Ausführungszeiten der Methoden durchgeführt. Die Messung beginnt mit der Ausführung der Prozesse, die Datenquellen sind, und führt dazu alle für diese Prozesse verwendbaren Methoden aus. Die Ergebnisse, die die Methoden liefern, dienen als Eingabe für die noch nicht gemessenen Prozesse. Die ermittelten *Ausführungszeitdaten* werden vom Merger gespeichert und schließlich vom Taskgraph-Generator zum PASCHA-Taskgraph hinzugefügt.

```
<Configuration>
   <IncludeDir>
    <Dir>/usr/include/qt</Dir>
  </IncludeDir>
  <HeaderFile>
    <File>robot.h</File>
  </HeaderFile>
  <LibFile>
    <File>-lqt-mt</File>
  </LibFile>
  <InitCode></InitCode>
  <CommandLine>
    <Argument>src/config/heulsuse.xml</Argument>
  </CommandLine>
    <TaskStructure>
      <RNT>
        <Name>SeedSearch</Name>
        <Period>400</Period>
        <ReleaseTime>0</ReleaseTime>
        <RNM>
          <Name>getSkipSeedSearch</Name>
          <Quality>10</Quality>
          <CallCode>SkipSeedSearch(img);</CallCode>
        </RNM>
        <RNM>
          <Name>getStandardSeedSearch</Name>
          <Quality>10</Quality>
          <CallCode>standardSeedSearch(img);</CallCode>
        </RNM>
        <RNM>
          ...
        </RNM>
        <RDD>
          <Dependency>Grabber</Dependency>
        </RDD>
      </RNT>
      <RNT>
        ...
      </RNT>
    </TaskStructure>
  </Configuration>
```

Abbildung A.2: Konfiguration der Bestandteile einer Anwendung

Abbildung A.3: RoboCup-Roboter im Spielfeld

A.1.2 Anwendungsoptimierung und Modellverfeinerung

Der angereicherte Taskgraph kann dann mit dem DTT-Optimierer eingeplant werden und analog zum ersten Durchlauf wird daraus eine optimierte *Main*-Methode erstellt. Da auch die daraus generierte Anwendung Ausführungszeitdaten liefern kann, ist es möglich den Zyklus öfter zu durchlaufen, um so die gewonnenen Informationen für eine bessere Annäherung an das tatsächliche Verhalten der Prozesse zu ermöglichen. Da alle Methoden ohne Unterbrechung ausgeführt werden, kann die Messung der Ausführungszeiten in der Main-Methode implementiert werden. Eine Anpassung des Methodenquelltextes für die Laufzeitmessung ist daher nicht nötig.

A.2 RoboCup-Anwendung

Die Beispielanwendung zum Thema „RoboCup“[1] wurde von Studenten im Rahmen einer Projektarbeit erstellt. Bei RoboCup geht es darum, Roboter zu entwickeln, die autonom Fußball spielen. Jeder Roboter verfügt über zahlreiche Sensoren, wie etwa eine Kamera oder Sonarsensoren, und muß sich aufgrund der durch diese gewonnenen Informationen auf dem Spielfeld zurechtfinden. Die in den Robotern eingebetteten Rechner verfügen über 128 MB Speicher und einen AMD K6-2 Prozessor mit einer festen Taktfrequenz von 400MHz. Die Ansteuerung der Roboter erfolgt über eine serielle Schnittstelle, über die auch Sensordaten der Odometrie und des Sonars empfangen werden. Die von der Kamera auf-

[1] www.robocup.org

genommenen Bilder werden mit einem PC-104+-Framegrabber digitalisiert, der auf der EBX-Hautplatine montiert ist. Als Betriebssystem kommt Linux mit der Echtzeiterweiterung RTAI zum Einsatz. Abbildung A.3 zeigt einen unserer Roboter und einen Teil des Spielfeldes. Ziel war es, die Entwickler von der Auswahl der zahlreichen Methoden zu entlasten und den bisher verwendeten statischen Ablaufplan der Prozesse durch den in dieser Arbeit vorgestellten qualitätsbewußten, dynamischen Echtzeitscheduler zu ersetzen. Die Zielfunktion bei der Optimierung des Schedules ist die Qualitätsmaximierung der Anwendung, da eine Energieminimierung der Anwendung aufgrund der festen Taktfrequenz des Prozessors und des sehr hohen Energieverbrauchs der Hardware keinen signifikanten Nutzen bringt.

A.2.1 Modellierung

Das in C/C++ implementierte Programm wurde zur Verwendung des in dieser Arbeit verwendeten Modells umstrukturiert. Alle Methoden wurden auf parameterlose Signaturen umgestellt. Der Datenaustausch erfolgt über gemeinsame Datenobjekte. Die Gültigkeit der Objekte wird durch den Optimierer, der die spezifizierten Datenabhängigkeiten berücksichtigt, und durch die nicht-unterbrechbare Ausführung der Methoden automatisch gewährleistet. Der ursprüngliche Quelltext wurde an mehreren Stellen aufgetrennt, um die einzelnen Teilprozesse des Projekts zu erzeugen.

Abbildung A.4 zeigt das derzeitige Datenflußmodell der RoboCup-Anwendung, die aus 21 Prozessen und 24 Methoden besteht. Zu Beginn jeder Hyperperiode verarbeitet die Anwendung zunächst die von der Hardware gelieferten Sensordaten. Anschließend werden die berechneten Daten, wie etwa Roboter-, Gegner- und Ballposition, in das Weltmodell des Roboters eingetragen und mit Methoden der Künstlichen Intelligenz ausgewertet. Am Ende der Hyperperiode sendet die Anwendung die berechneten Aktionen an die Aktoren des Roboters und andere Mitspieler. Alle Prozesse werden periodisch mit einer Periodendauer von 210 Millisekunden ausgeführt. Die Ausführung der Prozesse erfolgt in sechs Stufen:

1. Einlesen des Kamerabildes und der Sonar- und Odometriedaten
2. Vorverarbeitung der Sensordaten
3. Verarbeitung der Sensordaten mit verschiedenen Algorithmen
    - Hinderniserkennung
    - Roboterpositionsbestimmung
    - Gegnererkennung
    - Ballsuche

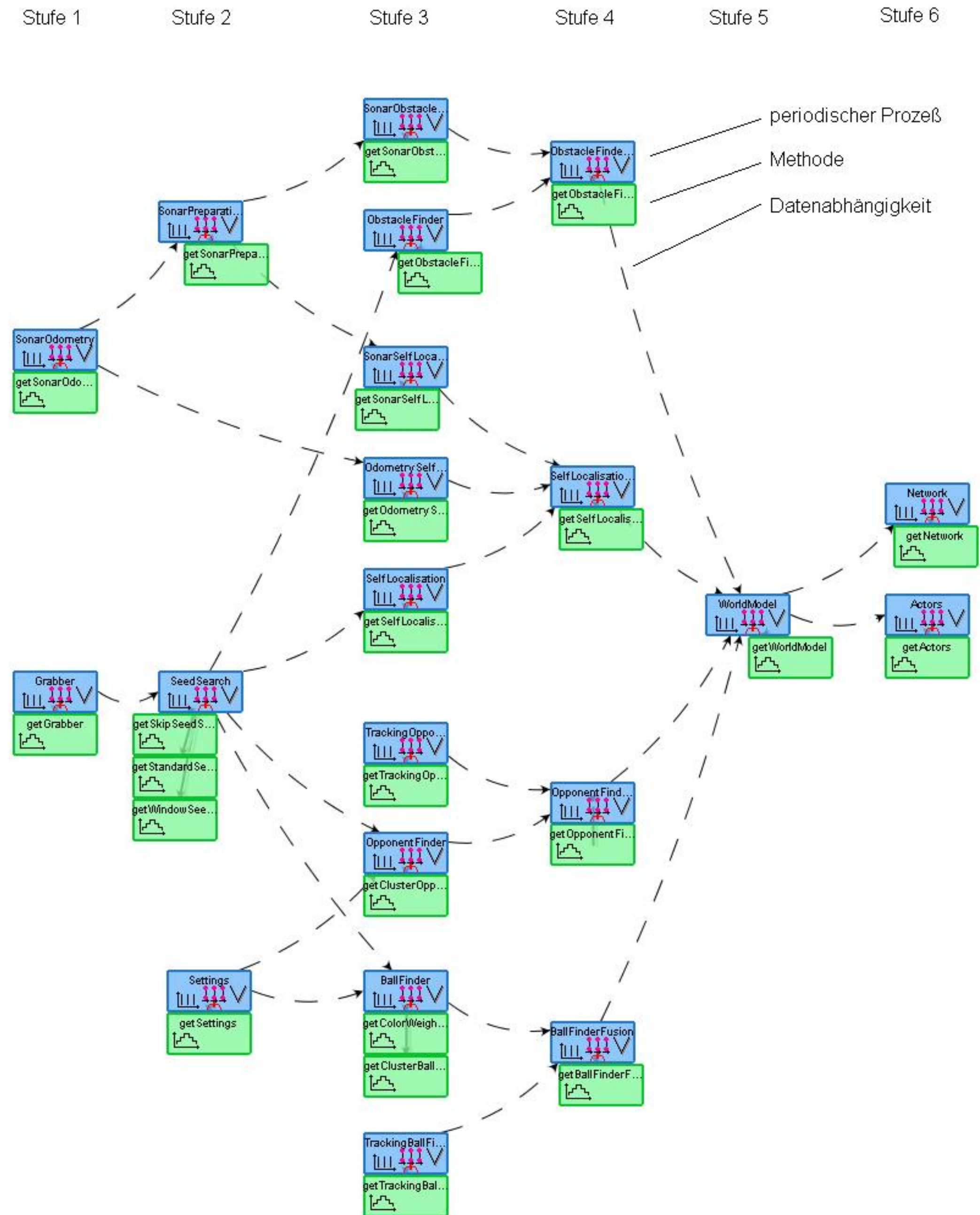

Abbildung A.4: Datenflußmodell der RoboCup-Anwendung

4. Fusion der von der Datenverarbeitung gelieferten Daten

5. Aktualisierung des Weltmodells und Auswertung des Weltmodells

6. Ansteuern der Aktoren und Netzwerkkommunikation

Für die Bestimmung von Farbübergängen im Kamerabild (*SeedSearch*) stehen die drei Methoden *getSkipSeedSearch*, *getStandardSeedSearch* und *getWindowSeedsearch* mit unterschiedlichem Zeitbedarf und unterschiedlicher Qualität zur Verfügung. Für den Prozeß *BallFinder*, dessen Aufgabe das Finden des Balles im Kamerabild ist, kann eine einfache Farbschwerpunktmethode (*getColorWeightBallFinder*) mit geringem Zeitbedarf und geringer Genauigkeit, oder eine zuverlässigere, langsamere mit Farbclustern arbeitende Methode (*getClusterBallFinder*) eingesetzt werden.

A.2.2 Initiale Messung der Methodenlaufzeiten

Für die Einplanung der Prozesse müssen die Ausführungszeitverteilungen aller verwendbaren Methoden bekannt sein. Da eine formale Analyse oder Abschätzung aufgrund der Komplexität und der ständigen Änderungen der Implementierung nicht in angemessener Zeit durchführbar ist, werden die Verteilungen gemessen. Um diese Messungen durchzuführen, müssen die von den Methoden jeweils benötigten Eingabedaten zur Verfügung stehen. Das hier vorgestellte Werkzeug erfordert jedoch nur, daß Testeingabedaten für alle Methoden, die zu Quellprozessen der Anwendung gehören, spezifiziert werden. Anschließend führt es diese Methoden aus, speichert die berechneten Ausgaben und verwendet sie als Eingabe für die Nachfolgeprozesse. In der Anwendung RoboCup bestehen die Eingaben für die Quellmethoden z.B. aus gespeicherten Bildern oder aus vom Simulator gelieferten Daten. Tabelle A.1 zeigt die gemessenen Ausführungszeiten der Methoden in Millisekunden und die Häufigkeit ihres Auftretens. Bei Einträgen mit der Ausführungszeit 0 ms handelt es sich um Läufe, deren Ausführungsdauer unter der Meßgenauigkeit von einer Millisekunde liegt.

A.2.3 Optimierung

Sobald der Taskgraph-Generator aus der geparsten Konfiguration und den gemessenen Ausführungszeiten das Systemmodell im PASCHA-Format erstellt hat, beginnt die Optimierung. Der Hauptprogrammgenerator liest den vom Optimierer erzeugten Bedingungsgraphen ein, und er erzeugt daraus zusammen mit dem betriebssystemspezifischen Kode die Main-Methode der Anwendung, die den dynamischen Scheduler enthält. Während der Ausführung der Main-Methode können

```
- <ProfileData>
- <Method id="1">
  <Profile duration="0">147</Profile>
  </Method>
- <Method id="2">
<Profile duration="0">147</Profile>
  </Method>
- <Method id="3">
  <Profile duration="0">147</Profile>
  </Method>
- <Method id="4">
  <Profile duration="0">147</Profile>
  </Method>
- <Method id="5">
  <Profile duration="0">21</Profile>
  <Profile duration="5">1</Profile>
  <Profile duration="7">1</Profile>
  <Profile duration="8">1</Profile>
  <Profile duration="11">1</Profile>
  <Profile duration="16">2</Profile>
  <Profile duration="18">2</Profile>
  <Profile duration="20">1</Profile>
  <Profile duration="24">2</Profile>
  <Profile duration="26">1</Profile>
  <Profile duration="30">1</Profile>
  <Profile duration="32">1</Profile>
  <Profile duration="34">1</Profile>
  <Profile duration="35">1</Profile>
  <Profile duration="39">1</Profile>
  <Profile duration="40">1</Profile>
  <Profile duration="45">1</Profile>
  <Profile duration="48">2</Profile>
  <Profile duration="49">79</Profile>
  <Profile duration="50">26</Profile>
  </Method>
- <Method id="6">
  <Profile duration="0">147</Profile>
  </Method>
- <Method id="7">
  <Profile duration="0">147</Profile>
  </Method>
- <Method id="8">
  <Profile duration="0">147</Profile>
  </Method>
- <Method id="9">
  <Profile duration="0">146</Profile>
  <Profile duration="1">1</Profile>
  </Method>
- <Method id="10">
  <Profile duration="0">147</Profile>
  </Method>
- <Method id="11">
  <Profile duration="0">147</Profile>
  </Method>
- <Method id="12">
  <Profile duration="0">147</Profile>
  </Method>
- <Method id="13">
  <Profile duration="0">21</Profile>
  <Profile duration="76">1</Profile>
  <Profile duration="79">104</Profile>
  <Profile duration="80">1</Profile>
  <Profile duration="81">2</Profile>
  <Profile duration="91">9</Profile>
  <Profile duration="100">1</Profile>
  <Profile duration="101">1</Profile>
  <Profile duration="102">1</Profile>
  <Profile duration="107">1</Profile>
  <Profile duration="109">1</Profile>
  <Profile duration="111">1</Profile>
  <Profile duration="112">1</Profile>
  <Profile duration="116">1</Profile>
  <Profile duration="117">1</Profile>
  </Method>
- <Method id="14">
  <Profile duration="17">9</Profile>
  <Profile duration="18">55</Profile>
  <Profile duration="19">62</Profile>
  <Profile duration="20">13</Profile>
  <Profile duration="21">7</Profile>
  <Profile duration="22">1</Profile>
  </Method>
- <Method id="15">
  <Profile duration="26">62</Profile>
  <Profile duration="27">79</Profile>
  <Profile duration="28">6</Profile>
  </Method>
- <Method id="16">
  <Profile duration="119">3</Profile>
  <Profile duration="120">18</Profile>
  <Profile duration="121">55</Profile>
  <Profile duration="122">51</Profile>
  <Profile duration="123">19</Profile>
  <Profile duration="124">1</Profile>
  </Method>
- <Method id="17">
  <Profile duration="1">147</Profile>
  </Method>
- <Method id="18">
  <Profile duration="0">146</Profile>
  <Profile duration="1">1</Profile>
  </Method>
- <Method id="19">
  <Profile duration="0">147</Profile>
  </Method>
- <Method id="20">
  <Profile duration="0">147</Profile>
  </Method>
- <Method id="21">
  <Profile duration="0">147</Profile>
  </Method>
- <Method id="22">
  <Profile duration="0">147</Profile>
  </Method>
- <Method id="23">
  <Profile duration="2">134</Profile>
  <Profile duration="3">13</Profile>
  </Method>
- <Method id="24">
  <Profile duration="0">147</Profile>
  </Method>
  </ProfileData>
```


Tabelle A.1: Gemessene Ausführungszeiten der Methoden

erneut Ausführungszeiten gemessen werden, und der Optimierungszyklus kann zur Verfeinerung des Bedingungsgraphen beliebig oft wiederholt werden.

Abbildung A.5 zeigt den initialen Schedule der RoboCup-Anwendung. Die Periodendauer der Prozesse wurde aufgrund der Ausführungsdauern[2] der Methoden auf 210 Millisekunden festgelegt. Der Schedule zeigt, daß in Abhängigkeit von der Ausführungsdauer der vier zu Beginn eingeplanten Prozesse unterschiedliche Methoden für die Suche nach Farbübergängen (*SeedSearch*) im Videobild eingesetzt werden. Wenn mehr Zeit zur Verfügung steht, wird die Methode *getStandardSeedSearch* verwendet, ansonsten die schnellere *getSkipSeedSearch*-Methode. Für die Suche des Balls (Prozeß *BallFinder*) wird immer die bessere Methode *getClusterBallFinder* eingesetzt.

A.2.4 Ausführung der Zielanwendung

Abbildung A.6 zeigt einen groben Überblick der Architektur des Linux/RTAI-Systems. Durch die RTAI-Erweiterung (Mikrokernel) wird der Linux-Kernel unterbrechbar gemacht, indem eine Software-Schicht (*hardware abstraction layer*) eingeführt wird, die alle Hardware-Interrupts verwaltet und zu passenden Zeitpunkten an den Standard-Kernel weiterleitet. Der Linux-Kernel wird als niedrig priorisierter Prozeß im Mikro-Kernel ausgeführt, während alle Echtzeitprozesse als Kernelmodule implementiert werden, die entsprechen ihrer Dringlichkeit mit höherer Priorität ausgeführt werden. Durch diese Vorgehensweise ist sichergestellt, daß Echtzeitprozesse nicht durch den Standard-Kernel oder im Benutzerbereich ablaufende Prozesse unterbrochen werden.

Durch das `LXRT`-Modul ermöglicht die RTAI-Erweiterung die Implementierung von Echtzeitprozessen in der Programmiersprache C++ im Benutzerbereich. Für harte Echtzeitprozesse ist dabei zu beachten, daß keine API-Aufrufe verwendet werden dürfen, die den Standard-Kernel aufrufen.

Abbilding A.7 zeigt den Quelltext der nötig ist, um einen RTAI-Echtzeitprozeß zu erzeugen. In den Zeilen `(1)` bis `(8)` wird eine Prozeßstruktur definiert und als Scheduler wird ein FIFO-Scheduler angelegt (first-in-first-out). Anschließend wird in den Zeilen `9` bis `14` die RoboCup-Anwendung geladen und schließlich als einziger Echtzeitprozeß gestartet.

Abbildung A.8 zeigt den Quelltext der `Main`-Methode der RoboCup-Anwendung, die den anwendungsinternen Dispatcher implementiert.[3] Da die RoboCup-

[2]Ausführungsdauern von null Millisekunden wurden für die Optimierung auf eine Millisekunde erhöht.

[3]In der Prototypimplementierung des Werkzeuges werden die Bedingungsgraphen in Bäume umgewandelt und im XML-Format gespeichert. Dies ist im Fall der RoboCup-Anwendung akzeptabel, da das Anwendungsmodell und der aus der Umwandlung entstehende Bedingungsbaum klein genug sind, um im Hauptspeicher gehalten werden zu können. Für andere, größere An-

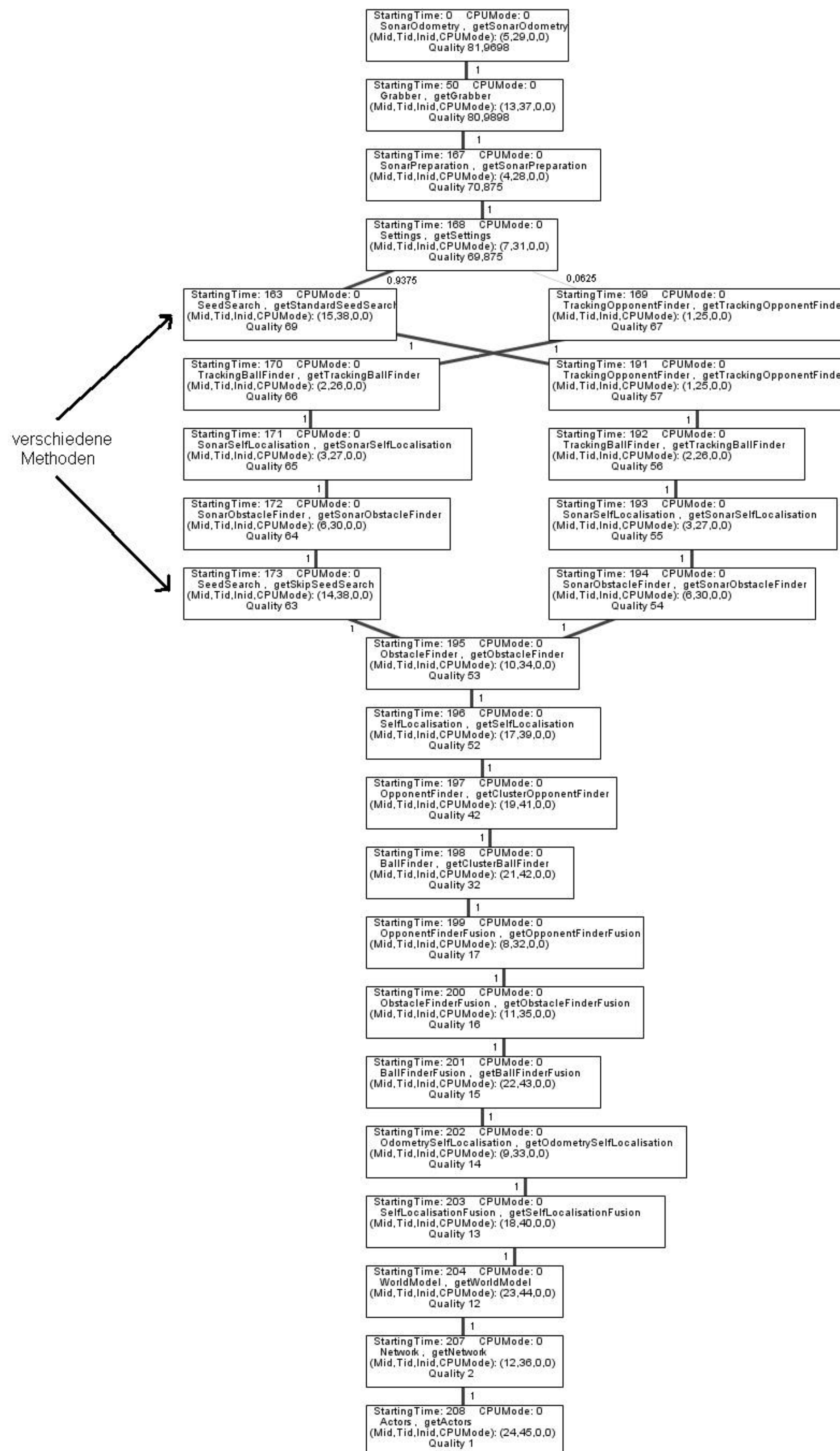

Abbildung A.5: Initialer Schedule der RoboCup-Anwendung

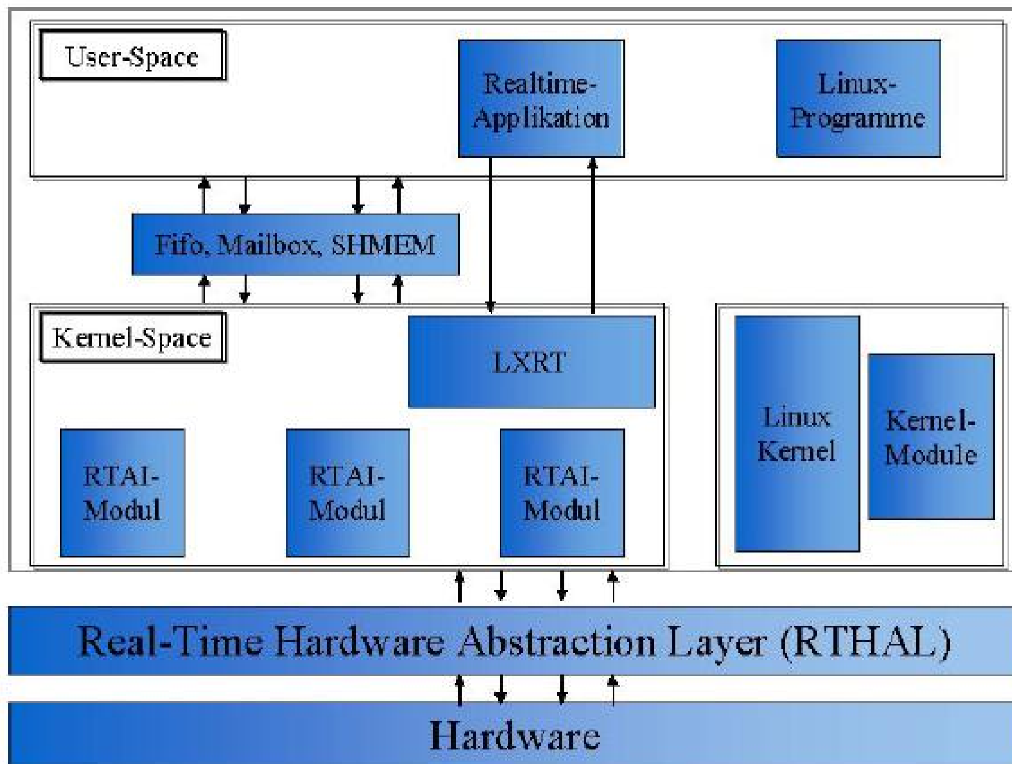

Abbildung A.6: Architektur der Echtzeiterweiterung RTAI für Linux

```
(1)   RT_TASK* mytask = 0;
(2)   struct sched_param mysched;
(3)   mysched.sched_priority=
(4)       sched_get_priority_max(SCHED_FIFO)-1;
(5)   if(sched_setscheduler(0,SCHED_FIFO,&mysched )==-1){
(6)       printf("ERROR IN SETTING THE POSIX SCHEDULER");
(7)       exit(1);
(8)   }
(9)   if (!(mytask = rt_task_init(nam2num("RoboCup"),
(10)                  priority, stack_size, msg_size ))) {
(11)      printf("CANNOT INIT TASK");
(12)      exit(1);
(13)  }
(14)  //Sporadische Task: rt_set_oneshot_mode();
(15)  rt_set_periodic_mode();
(16)  mlockall(MCL_CURRENT | MCL_FUTURE);
(17)  start_rt_timer(PERIOD);
(18)  rt_task_make_periodic(mytask,
(19)                        rt_get_time()+PERIOD,
(20)                        PERIOD);
(21)  //main-loop..
(22)  stop_rt_timer();
(23)  rt_task_delete(mytask);
```

Abbildung A.7: Quelltext zur Erzeugung des RTAI-Prozesses

```
int main(char**argv, int argc) {
  XMLTree G;
  generateXMLTreeFromFile("result.xml", G);
  while (true) {
    Node current=G.getRoot();
    while(current!=NULL) {
      execNode(current);
      branch(current, getRelativeTime());
    }
  }
}
```

Abbildung A.8: Main-Methode der RoboCup-Anwendung

Anwendung der einzige Echtzeitprozeß ist, kann er nicht durch andere Prozesse unterbrochen werden und verfügt dadurch über die Kontrolle des Prozessors. Am Anfang der `Main`-Methode wird der optimierte Bedingungsgraph eingelesen. In der äußeren `while`-Schleife wird der Wurzelknoten des Graphen geholt. Die innere Schleife führt den im jeweils aktuellen Knoten spezifizierten Prozeß mit der angegebenen Methode und Leistungsstufe aus und verzweigt nach deren Ende zum passenden Sohn. Falls der aktuelle Knoten ein Blatt ist, sind alle Prozeßinstanzen der aktuellen Hyperperiode ausgeführt, und in der äußeren Schleife wird erneut der Wurzelknoten geholt. Falls der aktuelle Knoten kein Blatt war, aber - z.B. aufgrund einer Zeitüberschreitung der ausgeführten Methode - kein geeigneter Sohn gefunden werden kann, so wird bei geringer Überschreitung der Sohn gewählt, der die größte Startzeit besitzt. Aufgrund der Laufzeitverteilungen der nachfolgend ausgeführten Methoden bestehen gute Chancen, daß die Zeitüberschreitung wieder eingeholt wird und das Spiel reibungslos fortgesetzt werden kann. Selbst wenn die Zeit nicht eingeholt werden kann, drohen keine Hardwareschäden.

A.2.5 Laufzeitmessung während der Anwendungsausführung

Die gemessenen Ausführungszeitverteilungen der Methoden können während der Ausführung der optimierten Main-Methode weiter verfeinert oder ersetzt werden. Dazu wurde der Dispatcher so erweitert, daß er auf Wunsch vor jeder Methodenausführung die aktuelle Systemzeit speichert und nach der Fertigstellung der Methode die verstrichene Zeit speichert. Beim Beenden der Anwendung werden die Daten in eine Protokolldatei geschrieben. Durch das kontinuierliche Messen der Ausführungszeiten während eines Spiels werden mehr relevante Situationen und Eingabedaten geliefert, als dies der Simulator ermöglicht. Neben dem Finden

---

wendungen ist jedoch eine Weiterentwicklung des Werkzeuges zur Verarbeitung von azyklischen, gerichteten Bedingungsgraphen erstrebenswert.

selten auftretender Ausführungszeiten werden dadurch auch die Wahrscheinlichkeiten der anderen Ausführungszeiten genauer an die Gegebenheiten des Spiels angepaßt, und eine zweite Optimierung des so entstehenden Modells kann eine für das Spiel maßgeschneiderte Main-Methode liefern.

Anhang B

Erweiterung auf heterogene Mehrprozessorsysteme

Die Optimierungsalgorithmen können auch für das Scheduling auf heterogenen Mehrprozessorsystemen erweitert werden. Die verbreitetet Vorgehensweise, zuerst das Allokationsproblem zu lösen und dann auf jedem Prozessor das Einprozessor-Scheduling-Verfahren anzuwenden, ist nur möglich, wenn die Allokationslösung datenabhängige Prozesse auf dem gleichen Prozessor plaziert. Die folgende Modellierung ermöglicht jedoch die gemeinsame Optimierung des Allokations- und des Schedulingproblems. Kommunikationskosten werden nicht berücksichtigt bzw. als konstant angenommen.

Definition 27 (Bearbeitungszustand). *Sei $P = \{p_1, \ldots, p_n\}$ die Menge der Prozessoren und $C(p_i)$ die Menge der Leistungsmodi des i-ten Prozessors. Der Vector $B_v = ((s_1, t_1, m_1, c_1), \ldots, (s_n, t_n, m_n, c_n)) \in \prod_{p_i \in P}(\mathbb{N}_0 \times AllTasks \cup \{\perp\} \times AllMethods \cup \{\perp\} \times C(p_i))$ stellt den* Bearbeitungszustand *der begonnenen Instanzen und Methoden dar. Der Vektor enthält Quadrupel (s_i, t_i, m_i, c_i), die angeben zu welchem Zeitpunkt s_i die Instanz t_i mit der Methode m_i und Leistungsmodus c_i auf Prozessor p_i gestartet wurde.*

Definition 28 (Mehrprozessor-Bedingungsgraph-Knoten). *Ein* Mehrprozessor-Bedingungsgraph *ist ein gerichteter azyklischer Multi-Level-Graph mit genau einer Wurzel. Jeder* Mehrprozessor-Bedingungsgraph-Knoten *v enthält*

- *die spätestmögliche Startzeit s_v,*
- *die zu schedulende Instanzmenge T_v,*
- *den Bearbeitungszustand der begonnenen Methoden und die Leistungsmodi der Prozessoren B_v,*

- *die zu startenden Instanzen, Methoden, zugeordneten freien Prozessoren und Leistungsmodi* $I_{v_{ext}} = ((I_1, M_1, p_1, c_1), \dots, (I_m, M_m, p_m, c_m))$,
- *den Erwartungswert* w_v *der Zielfunktion ab* v,
- *die Schlüsselmenge der Söhne* S_v,
- *die Menge der Väter* V_v.

Definition 29 (Schlüssel eines Knotens). *Das Tripel* (s_v, T_v, B_v) *wird als* Schlüssel *des Knotens* v *bezeichnet. Der Schlüssel gibt an, daß beginnend im Knoten* v *die Instanzmenge* T_v *noch eingeplant werden soll, wenn der Knoten spätestens zum Zeitpunkt* s_v *aufgerufen wird und der Bearbeitungsstatus der begonnenen Methoden* B_v *entspricht.*

Definition 30 (Zulässiger Knoten). *Ein Knoten* v *heißt* zulässig, *wenn mit* $s_0 := \max\{s_v, b_{I_v}\}$ *gilt:*

- $I_v \in T_v$ *(Prozeßinstanz* I_v *ist noch zu schedulen)*
- $M_v \in M_{I_v}$ *(Methode* M_v *implementiert* I_v*)*
- $\{I_1 \in I | I_1 <_I I_v\} \cap T_v = \emptyset$ *(alle Vorgänger von* I_v *sind bereits eingeplant)*
- $\forall (s_u, T_u, B_u) \in S_v : T_u = T_v \setminus \{I_v\} \wedge \exists (p, a) \in A_{M_v} : s_u = s_0 + a$ *(für alle möglichen Ausführungszeiten gibt es einen Sohnschlüssel, der die verbleibenden Prozeßinstanzen einplant), oder falls* $T_v \setminus \{I_v\} = \emptyset$ *gilt* $S_v = \emptyset$ *und* B_u *ist der Bearbeitungszustand* B_v *in dem die in* v *eingeplanten Instanzen* $I_{v_{ext}}$ *eingesetzt wurden.*
- $s_0 + WCET(M_v) \leq f_{I_v}$ *(die Frist von* I_v *wird auch im schlimmsten Fall eingehalten)*

Der Schlüssel eines Sohnes w des Knoten v, der durch die Beendigung der Instanz $t_i \in B_v \cup I_{v_{ext}}$ zum Zeitpunkt $f_{i,j} >= s_w$ mit $\forall k \neq i : \exists f_{k,l} \geq f_{i,j}$ entsteht, ist gegeben durch

$$\begin{aligned}
s_w &= s_v + f_{i,j} \\
T_w &= T_v \setminus \{t_i\} \\
B_w &= (\dots, (s_w, t_i = \bot, m_i = \bot, c_i), \dots) \cup I_{v_{ext}}.
\end{aligned}$$

Die Wahrscheinlichkeit $p_v(w)$ für den Aufruf von w nach v ist

$$p_v(w) = p(f_{i,j}) \cdot \prod_{k \neq i} \sum_{f_{k,l} \geq f_{i,j}} p(f_{k,l}).$$

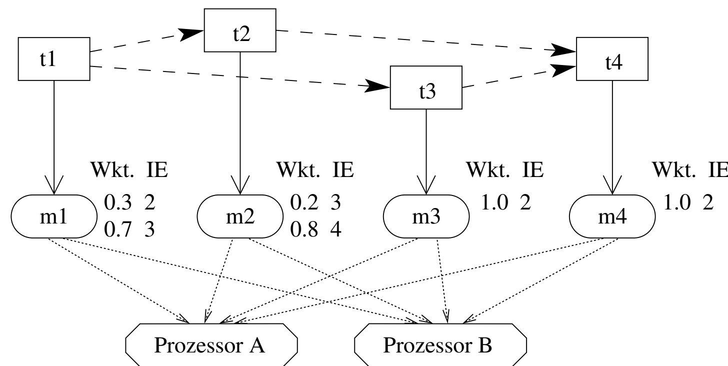

Abbildung B.1: Beispielanwendung für Mehrprozessorscheduling

Definition 31 (Sohn). *Ein Knoten u heißt* Sohn *eines Knotens v, wenn der Schlüssel von u in der Menge der Schlüssel der Söhne S_v von v enthalten ist.*

In dieser Darstellung wird eine Lösung des Einplanungsproblems durch einen Bedingungsgraphen, dessen Wurzelknoten den Schlüssel

$$(0, I, ((0, \perp, \perp, 1), \ldots, (0, \perp, \perp, 1)))$$

besitzt und zu dessen Knoten jeweils alle Söhne vorhanden und zulässig sind, repräsentiert. Jeder Knoten enthält die in der durch seinen Schlüssel beschriebenen Situation zu treffende Einplanungsentscheidung. Da bei jedem Übergang von einem Knoten zu einem seiner Söhne genau eine Methode beendet wird, ist die Höhe des Mehrprozessor-Bedingungsgraphen wie beim Einprozessor-Bedingungsgraphen gleich der Anzahl der einzuplanenden Prozeßinstanzen. In einem Knoten können mehrere Instanzen zu freien Prozessoren zugewiesen werden, aber es ist auch zulässig keine Instanz einem Prozessor zuzuweisen. Der durch das Allokationsproblem zusätzlich entstandene Freiheitsgrad spiegelt sich im höheren Verzweigungsgrad der Knoten wider.

Abbildung B.1 zeigt eine Anwendung, die auf einem Zweiprozessorsystem eingeplant werden soll. Sie enthält vier Prozesse mit gleicher Periodendauer. Für jeden Prozeß ist eine Methode verfügbar, die auf jedem der beiden Prozessoren ausgeführt werden kann. Die Struktur der Datenabhängigkeiten erlaubt die parallele Ausführung der Prozesse `t2` und `t3`.

Abbildung B.2 zeigt einen möglichen Mehrprozessor-Bedingungsgraphen. Die Ausführung beginnt mit Prozeß `t1` auf Prozessor `A`. Nach der Beendigung der Methode `m1` sind wieder beide Prozessoren verfügbar und der Scheduler führt `t2` auf Prozessor `A` und `t3` auf Prozessor `B` aus. Da `m3` vor `m2` beendet ist, aber für die Ausführung von `t4` auch `m2` beendet sein muß, kann im Sohnknoten `t4` nicht ausgeführt werden. Erst in der letzten Ebene, die durch die Beendigung von `m2` erreicht wird, führt der Scheduler `m4` aus.

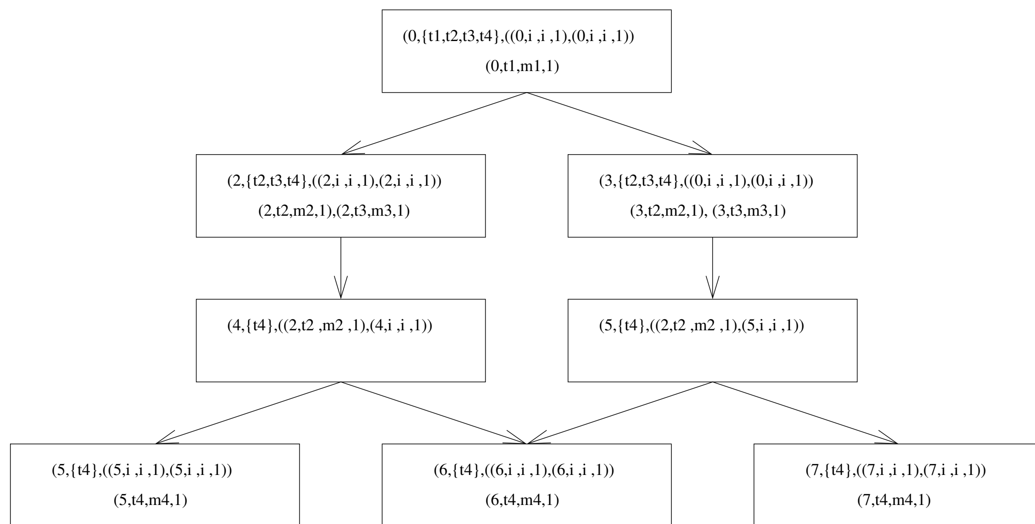

Abbildung B.2: Bedingungsgraph für Mehrprozessorscheduling

Anhang C

Energieverbrauch von Prozessoren

Ein Prozessor kann mehrere Modi besitzen, die sich in ihrer Rechenleistung und ihrem Energieverbrauch unterscheiden. Für jeden Modus wird die pro Zeiteinheit verbrauchte Energie angegeben, wenn der Prozessor aktiv e_a ist oder inaktiv e_i ist. Außerdem wird angegeben, wieviel Energie und Zeit für einen Moduswechsel nötig sind.

Die Regelung der Versorgungsspannung eines Prozessors ermöglicht große Energieeinsparungen, weil der Leistungsverbrauch P von CMOS-Schaltungen proportional zum Quadrat der Versorgungsspannung V ansteigt.[1] Mit den Bezeichnern C für circuit output load capacitance, N für die Anzahl der Schaltvorgänge während eines Taktzyklus und der Taktfrequenz f ist der Leistungsverbrauch P durch diese Formel gegeben:

$$P = C \cdot N \cdot V^2 \cdot f \tag{C.1}$$

Die Verzögerung δ des Schaltkreises ergibt sich nach [KL03] aus Formel C.2, wobei K eine vom Prozeß und der Gate-Größe abhängige Konstante bezeichnet, V_T die Schwellwertspannung (threshold voltage) ist und α zwischen 1 und 2 variiert.

$$\delta = \frac{C \cdot V}{K \cdot (V - V_T)^\alpha} \tag{C.2}$$

Da die Taktfrequenz umgekehrt proportional zur Verzögerung ist, ergibt sich aus den Formeln C.1 und C.2 der Zielkonflikt zwischen der verbrauchten elektrischen Leistung und der Rechenleistung eines Schaltkreises. Der Faktor $S(V)$, der angibt wieviel langsamer die Schaltung bei der Versorgungsspannung V im Vergleich zur maximalen Rechenleistung bei der maximalen Spannung V_{max} ist,

[1] Dieser Abschnitt entstand in enger Anlehnung an [KL03].

kann durch Einsetzen in C.2 folgendermaßen berechnet werden:

$$S(V) = \frac{V}{V_{max}} \cdot \left(\frac{V_{max} - V_T}{V - V_T} \right)^{\alpha} . \tag{C.3}$$

Der Faktor $P(V)$, der angibt wieviel höher der Leistungsverbrauch bei der Versorgungsspannung V im Vergleich zu dem der maximalen Spannung V_{max} ist, ergibt sich durch Einsetzen in C.1 zu[2]

$$P(V) \propto \left(\frac{V}{V_{max}} \right)^2 \cdot \left(\frac{1}{S(V)} \right) . \tag{C.4}$$

Mit den Formeln C.3 und C.4 ist der Faktor der für eine Instruktionseinheit benötigte Energie $E(V)$ bei der anliegenden Versorgungsspannung V folgendermaßen bestimmt:

$$E(V) \propto P(V) \cdot S(V) = \left(\frac{V}{V_{max}} \right)^2 . \tag{C.5}$$

Modellbeschränkungen

Weitere Effekte, die den Energieverbrauch des Prozessors beeinflussen, wie z.B. Leckströme, werden vernachlässigt. Dies gilt auch für den Energieverbrauch weiterer Systemkomponenten wie etwa Hauptspeicher, Zusatzkarten oder Bildschirm. Für alle Prozesse wird angenommen, daß alle Ressourcen ausreichend vorhanden sind. Diese Annahme ist bei Systemen mit harten Echtzeitbedingungen sinnvoll. Z.B. macht die Möglichkeit von Seitenfehlern beim Speicherzugriff eine Garantie für die Einhaltung von Fristen unmöglich, wenn bei der Messung der Ausführungszeiten diese nur selten auftreten. Der Speicherbedarf jedes Prozesses muß also bekannt sein, und in dem hier verwendeten nicht unterbrechbaren Einprozessormodell reicht es dann aus, wenn das System die Menge Hauptspeicher besitzt, die der Methode mit der größten Anforderung plus den zu speichernden Daten entspricht. Bei Mehrprozessorsystemen mit k Prozessoren muß der Speicherbedarf der k Methoden, die den größten Speicherbedarf besitzen und parallel ausgeführt werden, gedeckt werden.

[2] $\propto$: proportional zu

Anhang D

Schrankenbestimmung bei komplexen Anwendungen

Da es für größere Anwendungen nicht möglich ist den gesamten Suchraum zu erkunden, kann die erzielbare maximale Qualität bzw. die minimal nötige Energie mit CPLEX [ILO00] nach oben bzw. unten abgeschätzt.

Für die Abschätzungen werden einige Relaxationen eingeführt:

- Methoden sind unterbrechbar
- Ausführungszeitverteilungen werden durch kürzeste Ausführungsdauer ersetzt
- keine Datenabhängigkeiten
- keine Bereitzeiten
- keine Fristen

Für die Berechnung der Schätzwerte erfolgt eine Formulierung des Einplanungsproblems als ILP-Modell, das mit dem Programm CPLEX optimiert wird.

D.1 Qualitätsabschätzung

Abbildung D.1 zeigt die ILP-Darstellung für die Berechnung der oberen Schranke der erzielbaren Qualität. Zeilen `3` und `4` repräsentieren die Zielfunktion, daß die Summe der von den Instanzen gelieferten Qualitäten `Qi` zu maximieren ist. Anschließend werden die Nebenbedingungen für die Optimierung beschrieben. Für jede Instanz muß genau eine Methode `xij` eingeplant werden (Zeilen `8-10`), die von einer Instanz gelieferte Qualität ist die Qualität `qij` der gewählten Methode

(Zeilen `13-15`) und die Ausführungszeit `Di` der Instanz ist die Ausführungszeit `dij` der gewählten Methode (Zeilen `18-20`). Die Summe der Ausführungszeiten der Instanzen muß kleiner gleich der Dauer der Hyperperiode sein (Zeile `23`). Im letzen Block (Zeilen +25-28+) wird festgelegt, daß die Variablen zur Methodenauswahl nur die Werte `0` und `1` annehmen dürfen.

Aufgrund der vernachlässigten Nebenbedingungen und der Betrachtung der kürzesten Ausführungszeiten der Methoden (*best-case*) liefert die Optimierung des ILP-Modells eine obere Schranke für die maximal erzielbare Qualität.

```
( 1) // Summe der Qualitäten der ausgeführten
( 2) // Methoden ist zu maximieren
( 3) maximize
( 4)   Q0 + ... + Qz
( 5) such that
( 6)  // genau eine Methode je
( 7)  // Prozeßinstanz auswählen
( 8)   x00 + ... + x0m0 = 1
( 9)   ...
(10)   xz0 + ... + xzmz = 1
(11)   // die Qualität der Prozeßinstanz
(12)   // ist die der gewählten Methode
(13)   -Q0 + q00*x00 + ... + q0m0*x0m0 = 0
(14)   ...
(15)   -Qz + qz0*xz0 + ... + qzmz*xzmz = 0
(16)   // die Ausführungszeit der Prozeßinstanz
(17)   // ist die der gewählten Methode
(18)   -D0 + d00*x00 + ... + d0m0*x0m0 = 0
(19)   ...
(20)   -Dz + dz0*xz0 + ... + dzmz*xzmz = 0
(21)   // die Summe der Ausführungszeiten ist
(22)   // kleiner als das kgV der Perioden
(23)   D0 + ... + Dz <= Hyperperiod
(24) // xik sind binäre Variablen
(25) integers
(26)   x00 ... X0m0
(27)   ...
(28)   xz0 ... Xzmz
(29) end
```

Abbildung D.1: ILP für obere Qualitätsschranke

D.2 Energieabschätzung

Das ILP-Modell für die Berechnung einer unteren Schranke der benötigten Energie unterscheidet sich nur wenig von obigem Modell. Als Zielfunktion ist hier die Summe der von den Instanzen verbrauchten Energie `Ei` zu minimieren (Zeilen `3` und `4`). Die von einer Instanz verbrauchte Energie ist der erwartete Energieverbrauch `eij` der gewählten Methode `xij` (Zeilen`13-16`). Die Leistungsstufen des Prozessors werden implizit durch die Methoden dargestellt, indem für jede Leistungsstufe eine neue Methode im ILP-Modell eingeführt wird, deren Ausführungsdauer und Energieverbrauch der Ausführung der Methode in diesem Modus entspricht.

Um eine untere Schranke zu erhalten wird wiederum die jeweils kürzeste Ausführungszeit der Methoden für `dij` eingesetzt. Für die Berechnung der unteren Schranke kann die im Leerlauf verbrauchte Energie unbeachtet bleiben.

```
( 1) // Summe der Energie der ausgeführten
( 2) // Methoden ist zu minimieren
( 3) minimize
( 4)   E0 + ... + Ez
( 5) such that
( 6)  // genau eine Methode je
( 7)  // Prozeßinstanz auswählen
( 8)   x00 + ... + x0m0 = 1
( 9)   ...
(10)   xz0 + ... + xzmz = 1
(11)   // die Energie der Prozeßinstanz
(12)   // ist die der gewählten Methode
(13)   -E0 + e00*x00 + ... + e0m0*x0m0 = 0
(14)   ...
(15)   -Ez + ez0*xz0 + ... + ezmz*xzmz = 0
(16)   // die Ausführungszeit der Prozeßinstanz
(17)   // ist die der gewählten Methode
(18)   -D0 + d00*x00 + ... + d0m0*x0m0 = 0
(19)   ...
(20)   -Dz + dz0*xz0 + ... + dzmz*xzmz = 0
(21)   // die Summe der Ausführungszeiten ist
(22)   // kleiner als das kgV der Perioden
(23)   D0 + ... + Dz <= PeriodsLCM
(24) // xik sind binäre Variablen
(25) integers
(26)   x00 ... X0m0
(27)   ...
(28)   xz0 ... Xzmz
(29) end
```

Abbildung D.2: ILP für untere Energieschranke

Anhang E

Integration in das Framework PASCHA

Alle in dieser Arbeit vorgestellten Algorithmen sind in Java implementiert und in das Scheduling-Framework PASCHA integriert. Das Framework PASCHA ist unter der Adresse `lrs.fmi.uni-passau.de/~pascha` frei verfügbar. Nach dem Ausführen der Datei `setup.exe` unter Windows wird die Programmgruppe `PASCHA`, die die Teilprogramme des Systems enthält, angelegt.

E.1 Graphischer Editor

PASCHA erlaubt die komfortable Erstellung von Systemmodellen mit Hilfe eines graphischen Editors (Abbildung E.1). Das erstellte Modell kann durch Drücken des Knopfes `Check` auf Kompatibilität mit dem links daneben ausgewählten Scheduler überprüft werden. Für die in dieser Arbeit vorgestellten Algorithmen muß entweder der `OptimalSchedulerAlgorithm` oder der `SimAnnealSchedulerAlgorithm` ausgewählt sein. Nach der Überprüfung öffnet sich ein Fenster mit Informationen über die gefundenen inkompatiblen Teile des erstellten Modells.

Ein kompatibles Modell ist folgendermaßen aufgebaut. Es gibt genau eine Task, die den Wurzelknoten der Anwendung bildet. Darunter liegen die Tasks, die die Anwendung übersichtlich strukturieren. Alle Söhne dieser Tasks müssen die gleiche Periodendauer besitzen, und zwischen ihnen dürfen Datenabhängigkeiten bestehen. Alle periodischen Tasks müssen mindestens durch eine Methode Implementiert sein. Schließlich muß das Modell noch genau einen Prozessor enthalten, auf dem alle Methoden ausgeführt werden, da die derzeitige Implementierung nur Einprozessorsysteme unterstützt. Die Abbildungen E.2 a) bis d) zeigen die zulässigen Einstellungen. Umrandete Parameter sind frei wählbar, alle anderen

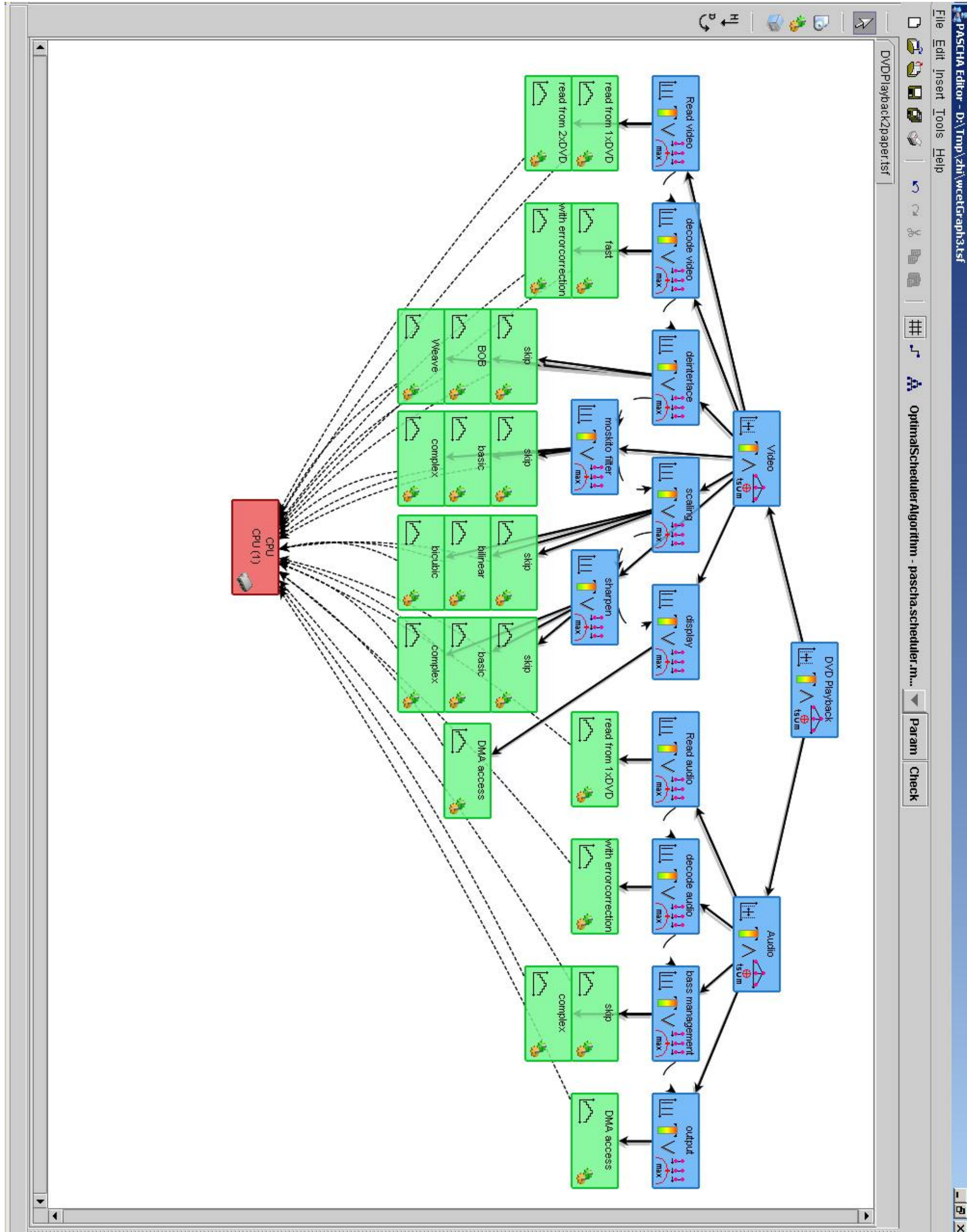

Abbildung E.1: Oberfläche des PASCHA-Editors

Property	Value
Task ID	1
Task Name	DVD Playback
Task Type	sporadic
Importance Type	mandatory
Base Priority	0
Log Type	AND
Interruptable	☑
Activate Rel.Time	☑
Release Time	0
Rel. Jitter defined	☐
Release Jitter	0
Start Prob. defined	☐
Start Probability	1.0
Activate Deadline	☐
Deadline Type	relative
Deadline	0
Utility Function	Edit...
Quality Function	TSUM
User Quality Funct.	UserQA-1
User-Def. Properties	Edit...

a) Wurzel- und Gruppenknoten

Property	Value
Task ID	20
Task Name	Read audio
Task Type	periodic
Importance Type	mandatory
Base Priority	0
Log Type	OR
Interruptable	☐
Period	50
Activate Rel.Time	☑
Release Time	0
Rel. Jitter defined	☐
Release Jitter	0
Start Prob. defined	☐
Start Probability	1.0
Cont. Rel. Jitter def.	☐
Cont. Release Jitter	0
Activate Deadline	☑
Deadline Type	relative
Deadline	0
Utility Function	Edit...
Quality Function	MAX
User Quality Funct.	UserQA-1
User-Def. Properties	Edit...

b) Periodische Knoten

Property	Value
Method ID	40
Method Name	DMA access
Method Type	discrete
Runtime Type	stochastic
Critical Sections	Edit...
Execution Times	Edit...
Activate WCDuration	☐
WorstCase Duration	0
Executing Processor	CPU
Create New	Edit...
User-Def. Properties	Edit...

c) Methoden

Property	Value
Resource ID	48
Resource Name	CPU
Resource Type	processor
Units	1
Processor Type	CPU
Interrupt Time	0
User-Def. Properties	Edit...
Power Management	Intel xScale
Speed Steps	4

d) Prozessor

Abbildung E.2: Parametereinstellungen im PASCHA-Editor

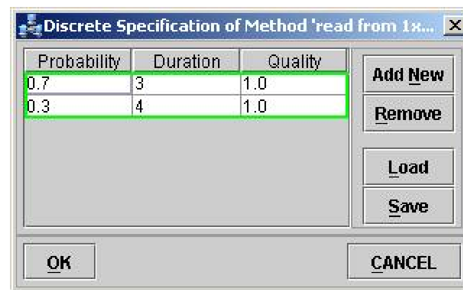

Abbildung E.3: Ausführungszeitverteilung und Qualität einer Methode

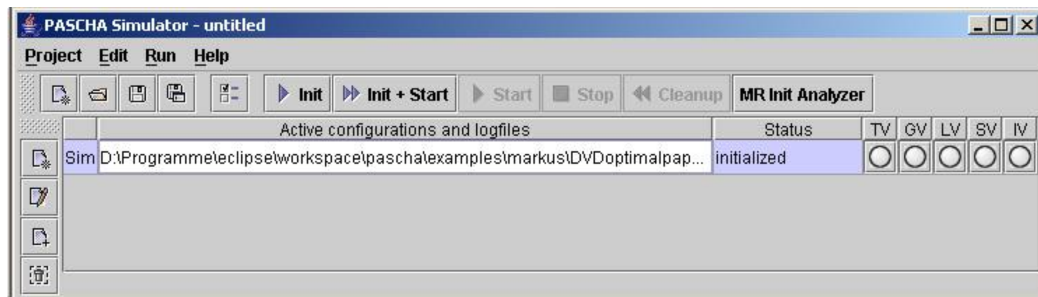

Abbildung E.4: Oberfläche des PASCHA-Simulators

müssen wie dargestellt eingestellt sein. Bei der Spezifikation (Abbildung E.3) der diskreten Ausführungszeitverteilung und der gelieferten Qualität muß die Summe der Wahrscheinlichkeiten `1` ergeben, und alle Qualitätseinträge müssen den gleichen, aber beliebigen, Wert haben.

E.2 Simulator

Um ein mit dem Editor erstelltes Systemmodell mit dem hier vorgestellten Algorithmus einzuplanen muß im Simulator (Abbildung E.4) eine Konfiguration erstellt werden. Die Konfiguration enthält den Dateinamen des einzuplanenden Modells, den zu verwendenden Scheduler und die Parameter des Schedulers. Für die anforderungsgetriebene Dynamische Programmierung ist der `OptimalSchedulerAlgorithm` auszuwählen und für den auf Simulated Annealing basierenden Algorithmus der `SimAnnealSchedulerAlgorithm`. Im Reiter `Guides & Optimizer` werden die zu verwendenden Lotsen und die Parameter der Suchalgorithmen eingestellt (siehe Abschnitt E.5).

Das Markieren einer Konfiguration und das Drücken des Knopfes `MR Init`

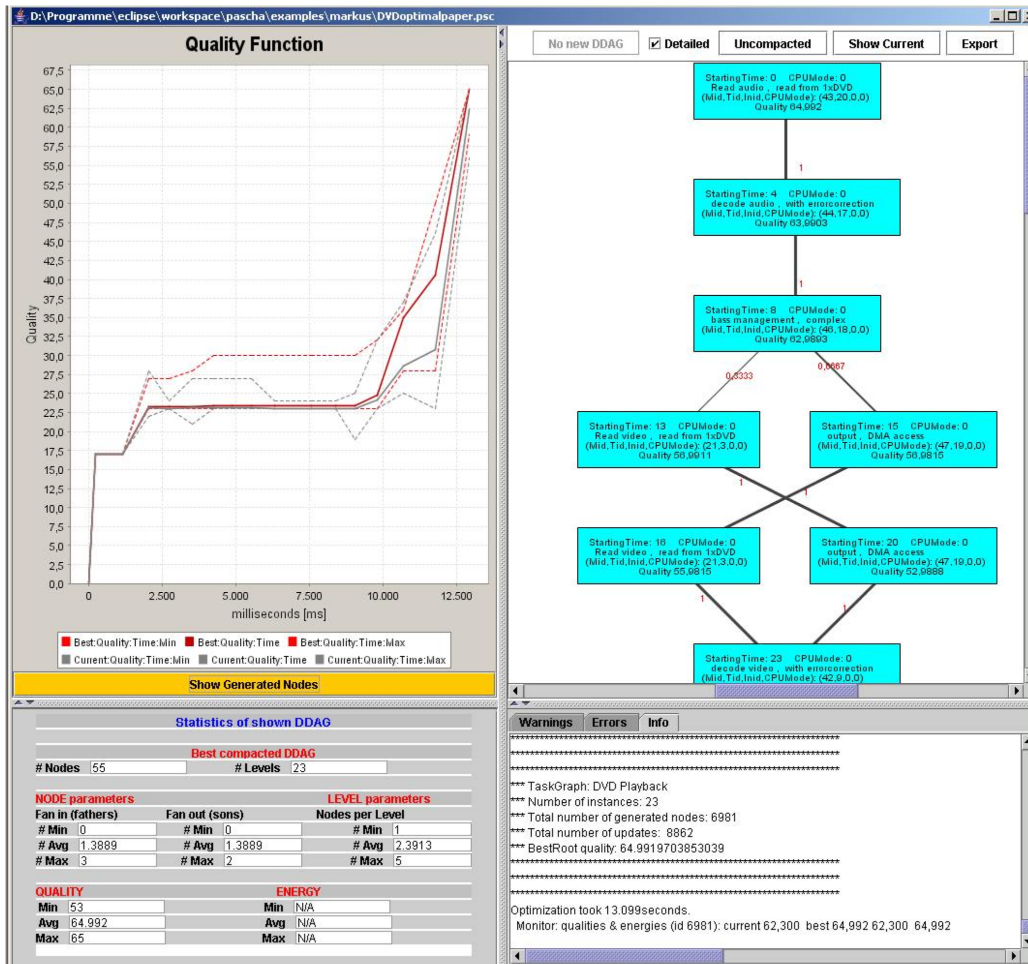

Abbildung E.5: Visualisiierung der Optimierungsphase

Analyzer startet die Visualisierung der Optimierungsphase. Durch Drücken des Init + Start wird die Optimierungsphase ausgeführt und direkt im Anschluß die Simulation gestartet. Falls eine der Simulationsvisualisierungen Zeitansicht (TV), Graphansicht (GV) oder Logansicht (LV) grün markiert ist, öffnet sich das Visualisierungsfenster mit den entsprechenden Ansichten. Durch drücken des Knopfes Stop wird die Simulation gestoppt und der Knopf Cleanup schließt alle Visualisierungsfenster.

E.3 Visualisierung der Optimierungsphase

Die Visualisierung der Optimierungsphase (Abbildung E.5) ist in vier Bereiche unterteilt. Der Verlauf der von den gefundenen Lösungen gelieferten Qualität bzw. verbrauchten Energie ist in der linken oberen Ecke dargestellt. Ein Knopf ermöglicht das Umschalten der Abszisse zwischen der Anzahl der erzeugten Knoten und der verstrichenen Zeit. Rechts daneben befindet sich die Anzeige der Bedingungsgraphen. Sobald der initiale Graph erzeugt wurde, wird er in diesem Bereich angezeigt. Im weiteren Verlauf der Optimierung können weitere Graphen durch den Knopf `Update` angefordert werden. Der Knopf `Show current` bzw. `Show best` führt zur Darstellung des letzten erhaltenen aktuellen Graphen bzw des letzten erhaltenen besten Graphen. Die Darstellung der kompaktifizierten bzw. nicht kompaktifizierten Graphen wird mit dem Knopf `Compacted` bzw. `Uncompacted` gewählt. Setzen oder Entfernen des Häkchens `Detailed` variiert die in den Knoten angezeigten Details und `Export` erlaubt das Speichern des dargestellten Graphen in eine Graphikdatei. Zu dem jeweils gezeigten Graphen werden im linken unteren Bereich statistische Daten wie etwa die Anzahl der Knoten oder die erzielte Qualität angezeigt. Das Fenster rechts unten zeigt vom Optimierer ausgegebene Informationen, Warnungen und Fehlermeldungen an.

E.4 Visualisierung der Simulation

Die Visualisierung der Simulation bietet verschiedene Ansichten. Die Graphansicht (Abbildung E.6) stellt detaillierte Informationen zu einem Simulationszeitpunkt dar. Beispielsweise sind Prozesse und Methoden, die gerade ausgeführt werden, mit einem grünen Dreieck markiert und die bereits erzielte Qualität wird angezeigt. Die Zeitansicht (Abbildung E.7) zeigt den zeitlichen Verlauf der Simulation, z.B. wann welcher Prozeß mit welcher Methode ausgeführt wurde, wann er bereit wurde und während welcher Zeit er blockiert war. Beim Prozessor zeigt eine rote Linie, wieviel Prozent der Energie, die bei Ausführung der angefallenen Instruktionseinheiten bei voller Leistung verbraucht worden wäre, verbraucht wurde. Die Höhe des schwarzen Balkens zeigt die Leistungsstufe des Prozessors und seine Auslastung an.

Eine einblendbare Logansicht zeigt alle vom Simulator und vom Scheduler erzeugten Ereignisse an, und die Statistikansicht kann genutzt werden, um z.B. die Anzahl der verpaßten Fristen anzuzeigen.

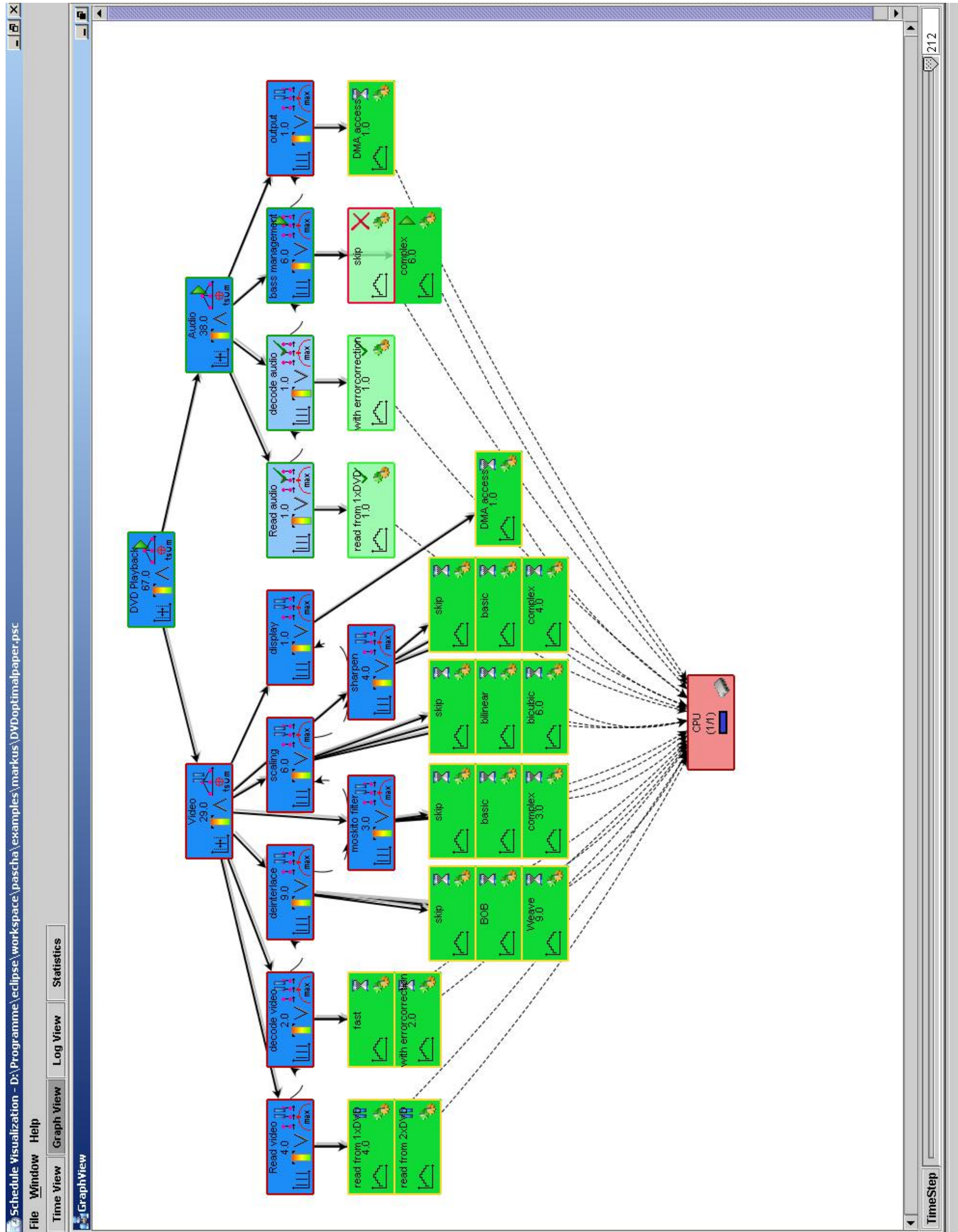

Abbildung E.6: Visualisierung eines Simulationszeitpunktes in der Graphansicht

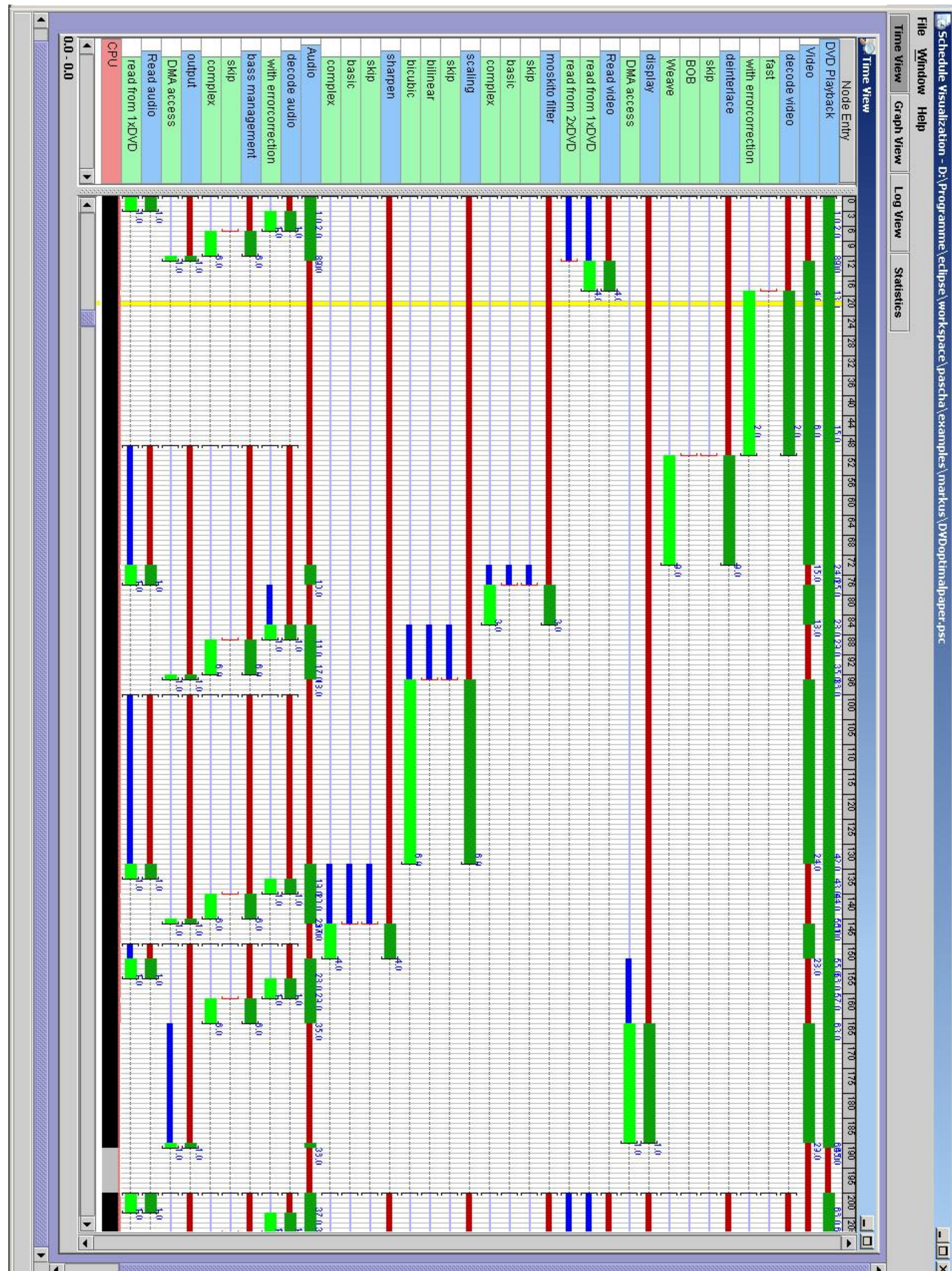

Abbildung E.7: Visualisierung des zeitlichen Verlaufes

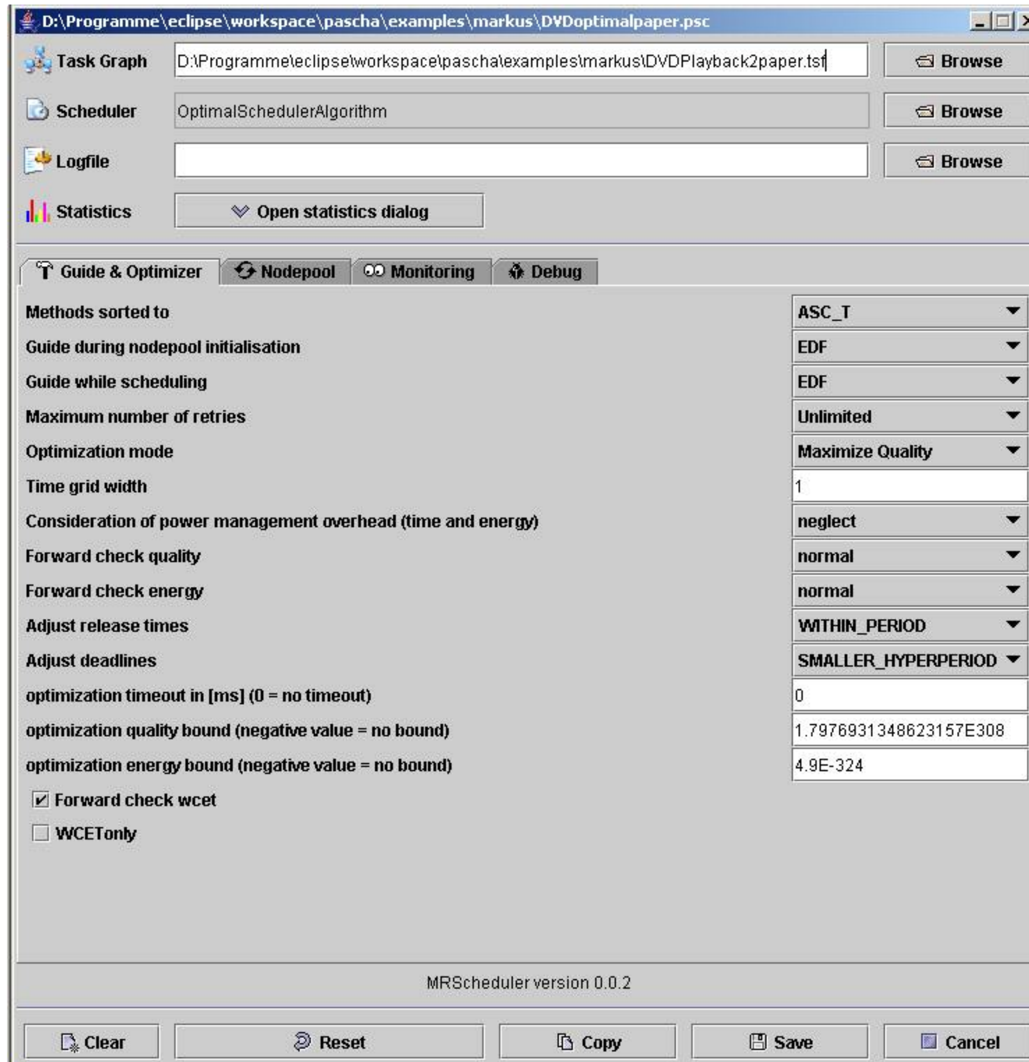

Abbildung E.8: Konfiguration des Optimierers (Lotse und Suchalgorithmus)

E.5 Parameter der Schedulingalgorithmen

Die folgende Auflistung zeigt die möglichen Einstellungen im Parameterdialog (Abbildung E.8) der beiden Schedulingalgorithmen. Die Standardwerte sind fett gedruckt.

Parameter des Reiters `Guides & Optimizer`

- Sortierung der Methoden im Lotsen (`Methods sorted to`)
  - nach ansteigender worst-case-Ausführungszeit (`ASC_T`)
  - nach fallender worst-case-Ausführungszeit (**`DESC_T`**)
  - nach ansteigender Qualität (`ASC_Q`)
  - nach sinkender Qualität (`DESC_Q`)
  - nach fallender worst-case-Ausführungszeit (`GREEDY_T`)
  - in zufälliger, deterministischer Reihenfolge (`PERTURB`)
- Initialisierungslotse (`Guide during nodepool initialization`)
  - nach ansteigender Frist (**`EDF`**)
  - nach ansteigender Bereitzeit (`ERF`)
  - in Modellreihenfolge (`Brute force`)
  - in zufälliger Reihenfolge (`Montecarlo`)
  - lexikographisch nach ansteigender Periodendauer und ansteigender Frist (`RMS/EDF`)
- Lotse während der Einplanung (`Guide during scheduling`)
  - siehe oben
- Max. Vorschlagsanzahl je Schlüssel (`Maximum number of retries`)
  - durch Lotsen bestimmt (**`Unlimited`**)
  - durch Scheduler bestimmt (`Scheduler`)
- Zeitraster (`Time grid width`)
  - bestimmt das Raster für die Zeiteinträge in den Schlüsseln (Standard **`1`**)
- Aufwand für Leistungsanpassung (`power management overhead`)

  - Aufwand vernachlässigen (**`neglect`**)
  - Aufwand schätzen (`estimate`)
  - Aufwand überschätzen (`overestimate`)

- Abbruchbedingungen für Qualität (`Forward check quality`)
  - keine Prüfung (`off`)
  - einfache Prüfung (**`normal`**)
  - komplexe Prüfung (`extended`)

- Abbruchbedingungen für Energie (`Forward check energy`)
  - keine Prüfung (`off`)
  - einfache Prüfung (**`normal`**)
  - komplexe Prüfung (`extended`)

- Einschränkung der Bereitzeiten (`adjust release times`)
  - am Periodenbeginn (`EQUALS_PERIODBEGIN`)
  - innerhalb der Periode (`WITHIN_PERIOD`)
  - größer gleich dem Periodenbeginn (**`GREATER_PERIODBEGIN`**)

- Einschränkung der Fristen (`adjust deadlines`)
  - am Periodenende (`EQUALS_PERIODEND`)
  - innerhalb der Periode (`WITHIN_PERIOD`)
  - innerhalb der Hyperperiode (**`SMALLER_HYPERPERIOD`**)

- Zeitlimit für Optimierung (`optimization timeout`)
  - Angabe in Millisekunden (Standard **`0`**)

- Abbruchwert für Qualität (`optimization bound quality`)
  - Optimierungsabbruch bei Überschreiten des Wertes (Standard **`-1`**)

- Abbruchwert für Energie (`optimization bound energy`)
  - Optimierungsabbruch bei Unterschreiten des Wertes (Standard **`-1`**)

- Prüfung der worst-case-Ausführungszeiten (`forward check wcet`)
  - Abbruch bei unzureichender Restzeit (Standard **`true`**)

- Beschränkung auf worst-case-Ausführungszeiten (`wcet only`)
  - nur worst-case-Analyse (Standard **`false`**)

Parameter des Reiters `Nodepool`

- Alte Knoten zwischenspeichern (`caching`)
  - bei `false` werden nicht mehr benötigte Knoten direkt an den Ringspeicher zurückgegeben (Standard **`true`**)
- Strategie der zu speichernden alten Knoten (`cache mode`)
  - Halten des ersten Knotens (`first cached node`)
  - Halten des besten Knotens (**`best cached node`**, zwingend bei ADP)
  - Halten des letzten Knotens (`last cached node`)
- Anzahl der verfügbaren Knoten (`number of nodes`)
  - Anzahl der gleichzeitig verfügbaren Knoten (Standard **`10000`**)
- Anzahl der verfügbaren Knoten (`cache resizing`)
  - Erlauben rekursiver Verdopplung der verfügbaren Knotenanzahl (Standard **`false`**)

Parameter des Reiters `Monitoring`

- Exaktes Monitoring
  - Aktivieren des Monitorings (`Default monitoring`, Seiteneffekt: alle Knoten werden aktualisiert)
  - Anfangsanzahl der Knoten zwischen Monitorpunkten (Standard **`0`**, kein Monitoring)
  - Anfangszeitintervall in Millisekunden zwischen Monitorpunkten (Standard **`0`**, kein Monitoring)
- Schnelles Monitoring
  - Aktivieren des schnellen Monitorings (`Quick and dirty monitoring`, Knoten werden nicht aktualisiert, kann veraltete Werte liefern)

  - Anfangsanzahl der Knoten zwischen Monitorpunkten (Standard **0**, kein Monitoring)
  - Anfangszeitintervall in Millisekunden zwischen Monitorpunkten (Standard **0**, kein Monitoring)

Parameter des Reiters `Simulated Annealing` (nur Simulated Annealing)

- Starttemperatur (`Start temperature`, Standard **1000000**)
- Stopptemperatur (`Stop temperature`, Standard **0,01**)
- Abkühlfaktor (`Cool down factor`, Standard **0,99**)
- Anzahl Schritte bei konstanter Temperatur (`Steps to take before temperature is decreased`, Standard **1000**)

Der Reiter `Debug` bietet diverse Hilfestellungen bei der Entwicklung und Fehlersuche an.

Anhang F

Klassendiagramme

Dieser Anhang beschreibt die Klassenstruktur der Implementierung der Optimierer, der Lotsen, des Knotenspeichers, des Systemmodells und der Parameter der Algorithmen. Ziel des Entwurfs ist die Erleichterung einer Erweiterung der Implementierung um neue Optimierer, Lotsen und Zielfunktionen.

F.1 Optimierer

Abbildung F.1 zeigt das Klassendiagramm für die Umsetzung der Optimierer[1]. Das Interface `MRScheduler` stellt sicher, daß die verwendeten Optimierer die vom Knotenspeicher benötigten Methoden implementieren. In `MRScheduler-Base` sind die von beiden Optimierern benötigten Methoden, wie z.B. das Laden und Speichern der Parameter oder den Dispatcher, der vom Simulator vor jedem Simulationsschritt aufgerufen wird, implementiert.

Die Klasse `OptimalSchedulerAlgorithm` enthält die anforderungsgetriebene Dynamische Programmierung und der Simulated Annealing Algorithmus befindet sich in der Klasse `SimAnnealSchedulerAlgorithm`. Die Methode `Init()` wird vom Simulator vor der Simulation des Systemmodells aufgerufen und sie implementiert die Optimierung des Systemmodells. Die Methode `acceptNode()` wird vom Knotenspeicher aufgerufen, wenn er einen neuen Knoten erzeugt hat.

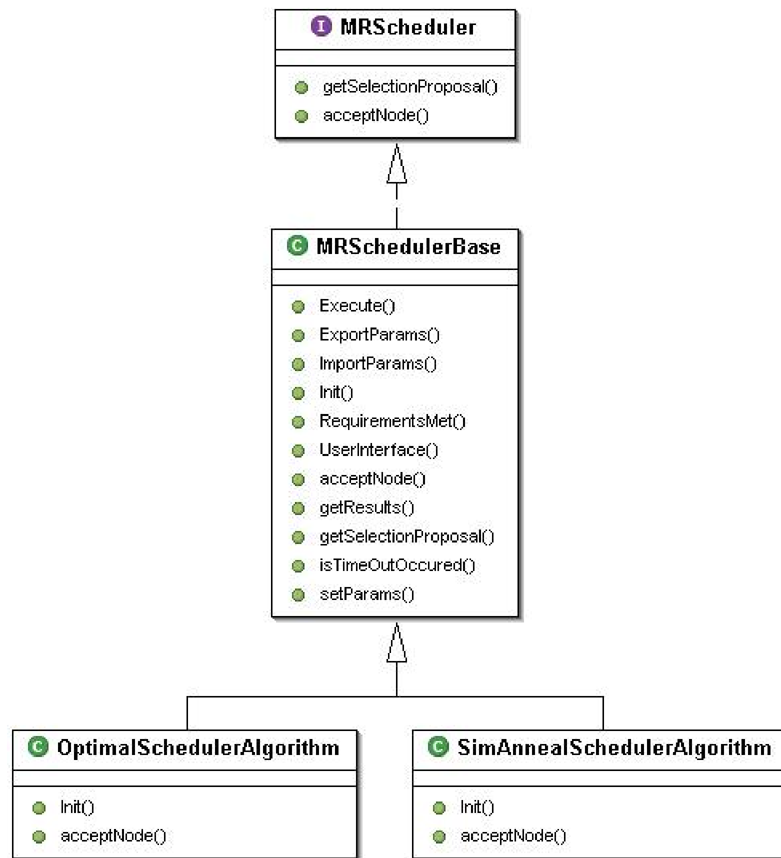

Abbildung F.1: Klassendiagramm der Optimierer

F.2 Lotsen

Abbildung F.2 zeigt das Klassendiagramm für die Umsetzung der Lotsen. Das Interface `Guide` erzwingt die Implementierung der beiden Methoden, die von den Optimierern und dem Knotenspeicher benötigt werden, um einen neuen Vorschlag für eine Kombination von Instanz, Methode und Leistungsstufe anzufordern, und zu Überprüfen, ob der Lotse mit dem Optimierer kompatibel ist. `GuideBase` enthält die von den Lotsen gemeinsam genutzten Methoden zum Überprüfen der Abbruchbedingungen (`forwardCheck()`) und der Kompatibilität.

Der Zufallslotse ist in der Klasse `MonteCarloGuide` implementiert und enthält einen Zufallsgenerator für die vorgeschlagenen Kombinationen aus In-

[1]Der Optimierer ist im Framework PASCHA als Scheduler sichtbar. Für das dynamische Nachladen der Scheduler im PASCHA-Simulator müssen diese Klassen mit dem Suffix `SchedulerAlgorithm` enden, um gefunden zu werden.

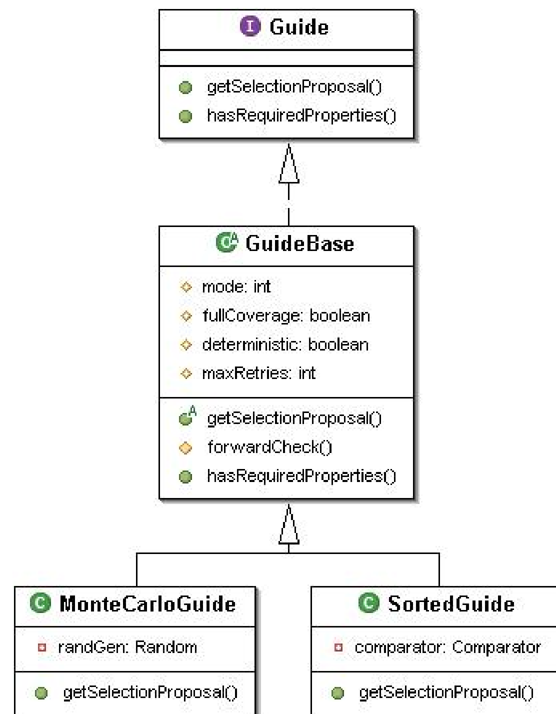

Abbildung F.2: Klassendiagramm der Lotsen

stanz, Methode und Leistungsstufe. Alle anderen implementierten Lotsen zählen die möglichen Kombinationen nur in unterschiedlichen Reihenfolgen auf. Sie sind daher gemeinsam in der Klasse `SortedGuide` implementiert, in der jeweils nur die gewünschte Sortierreihenfolge eingestellt wird.

F.3 Knotenspeicher

Abbildung F.3 enthält die Klassenstruktur für den Knotenspeicher und die Zielfunktionen. Die Klasse `Nodepool` bildet die Schnittstelle zu den Optimierern. Sie aggregiert Objekte vom Typ `Level`, die die Ebenen des Knotenspeichers darstellen und verwalten. In jedem `Level` werden die Schlüssel `NodeKey` der erzeugten Knoten in drei Bereichen für beste, aktuell betrachtete und ausgelagerte Knoten gehalten. Jeder gespeicherte `NodeKey` verweist auf einen Knoten (`Node`) der die für den Schlüssel gewählte Kombination aus Instanz, Methode und Leistungsstufe und den dadurch erzielten Wert der Zielfunktion enthält. Um die Integration neuer Zielfunktionen zu erleichtern, wurden alle sie betreffenden Berechnungen in eigene Klassen ausgelagert. Die Klasse `TargetFunction` enthält

Methoden, die allen Zielfunktionen gemeinsam sind und abstrakte Definitionen für Methoden, die von jeder Zielfunktion implementiert werden müssen. Hierzu gehören die Methoden zur Überprüfung der Abbruchbedingungen für Qualität bzw. Energie (`forwardCheck...()`), der Vergleich welcher von zwei Werten besser ist (`firstIsSuperiorToSecond()`) und um wieviel (`gain()`)und die rekursive Berechnung `calculateValue()` des Zielfunktionswertes eines Knotens.

In der Klasse `QualityTargetFunction` sind die Methoden für die Maximierung der zu erwartenden Qualität enthalten. Die Klasse `EnergyTargetFunction` implementiert die Zielfunktion, um den Energieverbrauch zu minimieren. Neben den allgemeinen Methoden enthält sie Methoden zur Berechnung des Energieverbrauchs während der Leerlaufzeiten zwischen und nach den ausgeführten Instanzen.

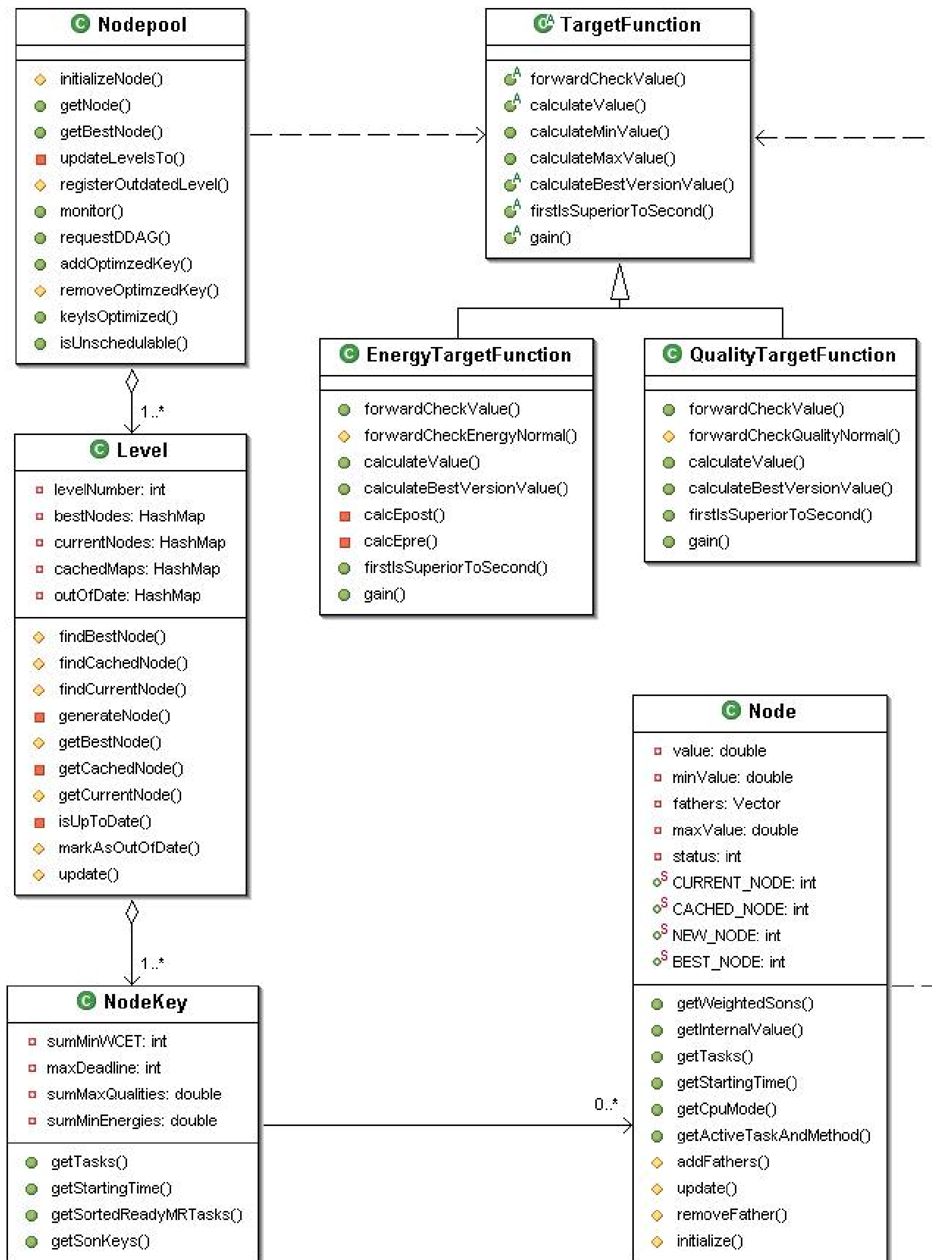

Abbildung F.3: Klassendiagramm des Knotenspeichers

F.4 Systemmodell

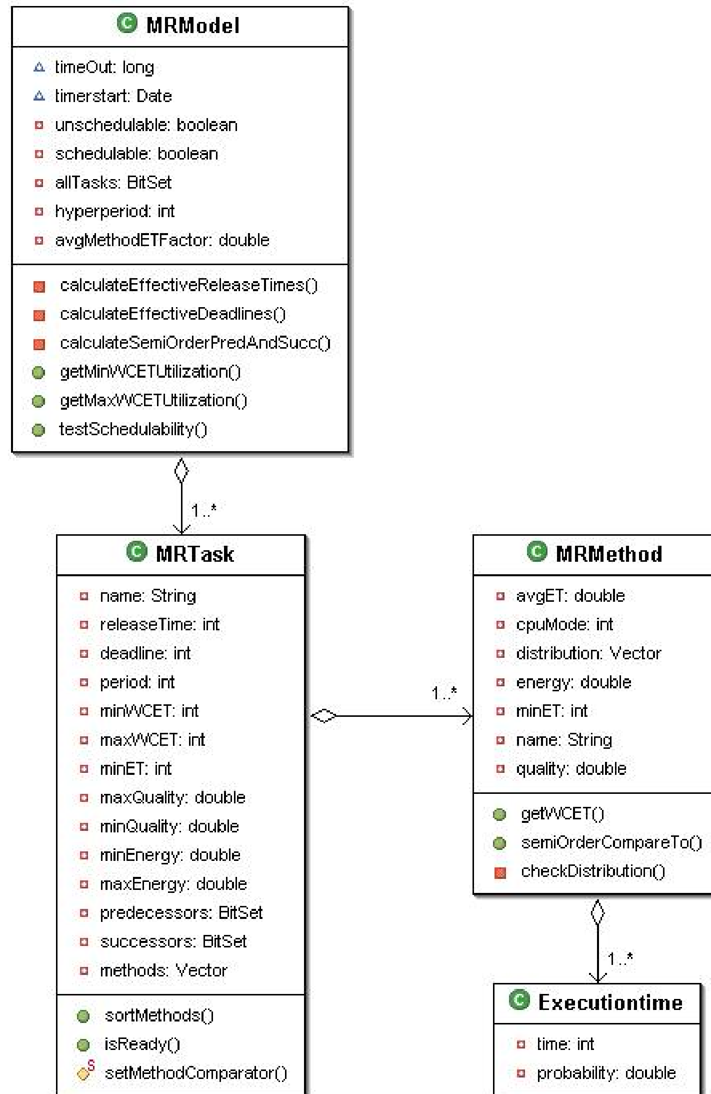

Abbildung F.4: Klassendiagramm des Anwendungsmodells

Abbildung F.4 enthält die Klassenstruktur des Systemmodells. Die oben genannten Klassen greifen auf das Systemmodell über die Klasse `MRModel` zu. Diese Klasse stellt eine Methode für den (zeitlich begrenzbaren) Test der Einplanbarkeit (`testSchedulability()`) des Modells zur Verfügung. Sie übernimmt die Instanzierung der von PASCHA gelieferten periodischen Tasks, sowie die Berechnung der effektiven Bereitzeiten und Fristen. Die von ihr aggregierten

Instanzen sind Objekte des Typs `MRTask`. Dieser Typ enthält Informationen über die Bereitzeit und Frist der Instanz, ihre Vorgänger und Nachfolger bezüglich der Datenabhängigkeiten und die für ihre Ausführung verwendbaren Methoden. Diese sind Objekte vom Typ `MRMethod`, der den Energieverbrauch der Methode bzw. die von ihr gelieferte Qualität. Die Methode `semiOrderCompareTo()` implementiert den Vergleich der Verteilungsfunktionen zweier Methoden bezüglich der zu optimierenden Zielfunktion. Die möglichen Ausführungsdauern und ihre Wahrscheinlichkeiten werden in der Klasse `Executiontime` gespeichert. Die Informationen zu den Leistungsmodi des Prozessors werden bei der Konvertierung des PASCHA-Modells in dieses Modell in die Methoden gespeichert.

F.5 Parameter der Algorithmen

Abbildung F.5 enthält die Klassenstruktur zum Speichern der Parameter der Algorithmen. Die Klasse `MRSchedulerBaseParams` enthält die von beiden Optimierern, den Lotsen und dem Knotenspeicher benötigten Parameter. Sie ist von der Klasse `Configuration` abgeleitet, um das Laden und Speichern der Parameter in PASCHA zu ermöglichen. Die zusätzlichen vom SimulatedAnnealing-Scheduler benötigten Parameter befinden sich in der Klasse `SimAnnealSchedulerParams`. Die (leere) Klasse `OptimalSchedulerParams` ist nur für Lade- und Speichervorgänge in PASCHA notwendig.

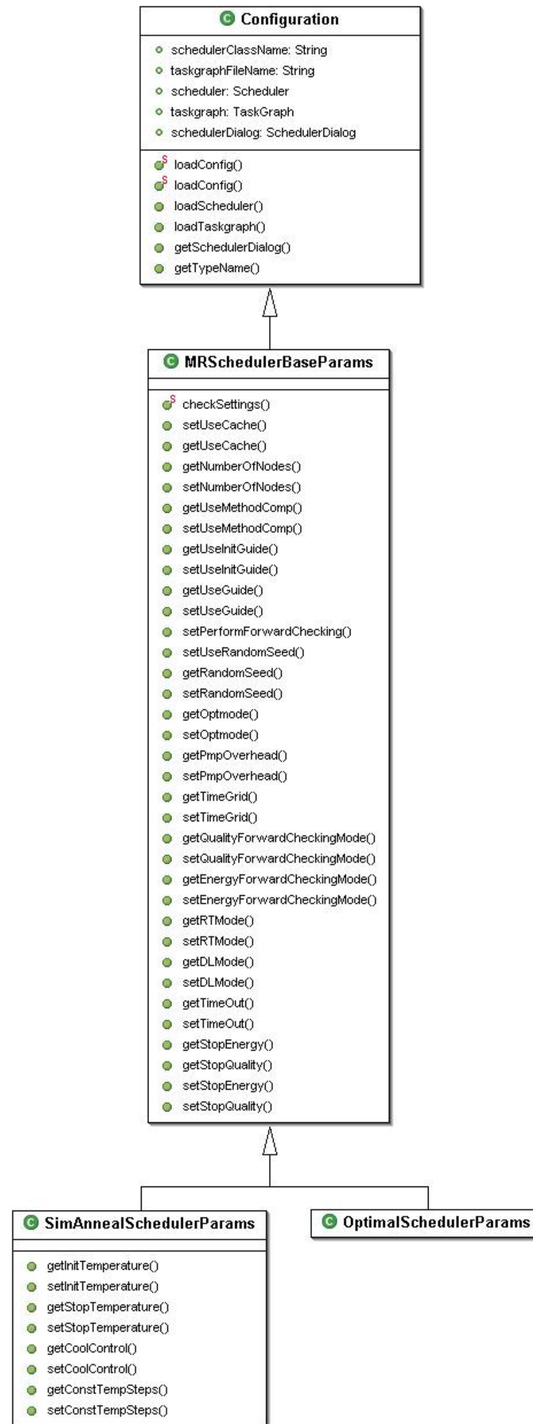

Abbildung F.5: Klassendiagramm der Parameter der Optimierer

Literaturverzeichnis

[BB95] BURD, T. D. und R. W. BRODERSEN: *Energy efficient CMOS microprocessor design*. In: *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, Seiten 288–297. IEEE Computer Society, 1995.

[BBS95] BARTO, A. G., S. J. BRADTKE und S. P. SINGH: *Learning to Act using Real-Time Dynamic Programming*. In: *Articial Intelligence Journal*, Band 72, Seiten 81–138, 1995.

[BDTL96] BASSO, A., I. DALGIC, F. TOBAGI und C. LAMBRECHT: *Study of MPEG-2 Coding Performance Based on a Perceptual Quality Metric*. In: *Proc. PCS'96, Australia*, Seiten 263–268, März 1996.

[Bel57] BELLMANN, RICHARD: *Dynamic Programming*. Princeton University Press, 1957.

[Ber87] BERTSEKAS, D. P.: *Dynamic Programming: Deterministic and Stochastic Models*. Englewood Cliffs, New Jersey: Prentice-Hall, 1987.

[BG03] BONET, B. und H. GEFFNER: *Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming*. In: *Proceedings of 13th International Conference on Automated Planning and Scheduling (ICAPS-2003), Trento (Italy)*, Seiten 12–21. AAAI Press, 2003.

[Bin97] BINNS, P.: *Incremental Rate Monotonic Scheduling for Improved Control System Performance*. In: *Proceedings of Third IEEE Real-time Technology and Applications Symposium (RTAS'97)*, Seiten 80–90, June 1997.

[Cam00] CAM, HASAN: *An on-line scheduling policy for IRIS real-time composite tasks*. In: *Journal of Systems and Software*, Band 52(1), Seiten 25–32. Elsevier Science Inc., 2000.

[Cat98] CATONI, OLIVIER: *Solving Scheduling Problems by Simulated Annealing*. In: *SIAM Journal on Control and Optimization*, Band 36(5), Seiten 1539–1575, 1998.

[CB97] CHARPILLET, FRANÇOIS und ANNE BOYER: *Progress: an Approach for Defining and Monitoring Non-deterministic Design to Time methods*. In: *Ninth IEEE International Conference on Tools with Artificial Intelligence. (Newport Beach, California, USA). IEEE Computer Society*, Seiten 502–507, 1997.

[CKT03] CHAKRABORTY, S., S. KÜNZLI und L. THIELE: *A General Framework for Analysing System Properties in Platform-Based Embedded System Designs*. In: *Proceedings of the 6th Design, Automation and Test in Europe (DATE)*, Seiten 190–195, München, Germany, März 2003.

[Cra95] CRABTREE, I. B.: *Resource scheduling - comparing simulated annealing with constraint propagation*. In: *BT Technology Journal*, Band 13(1), Seiten 121–127, Januar 1995.

[CSB92] CHANDRAKASAN, A., S. SHENG und R. BRODERSEN: *Low-Power CMOS Digital Design*. In: *IEEE Journal of Solid-State Circuits*, Band 27(4), Seiten 473–484, 1992.

[DB88] DEAN, T. und M. BODDY: *An analysis of time dependent planning*. In: *Proceedings of the Seventh National Conference on Artificial Intelligence*, Seiten 49–54. AAAI/MIT Press, 1988.

[Dec96] DECKER, KEITH: *TAEMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms*. In: O'HARE, G. und N. JENNINGS (Herausgeber): *Foundations of Distributed Artificial Intelligence, Chapter 16*, Seiten 429–448. Wiley Inter-Science, January 1996.

[DHSS02] DONCKERS, L., P.J.M. HAVINGA, G.J.M. SMIT und L.T. SMIT: *Energy Efficient TCP*. In: *Proceedings 2nd Asian International Mobile Computing conference (AMOC2002)*, Seiten 18–28. Malaysia, ACM Sigmobile, Mai 2002.

[DWH03] DULMAN, S., J. WU und P. HAVINGA: *An energy efficient multipath routing algorithm for wireless sensor networks*. In: *IEEE International Symposium on Autonomous Decentralized Systems (ISADS 2003)*. Pisa, Italy, April 2003.

[Fle01] FLEISCHMANN, MARC: *LongRun(TM) Power Management: Dynamic Power Management for Crusoe(TM) Processors,Transmeta Corporation*, 2001.

[Foh94] FOHLER, GERHARD: *Flexibility in Statically Scheduled Hard Real-Time Systems*. Doktorarbeit, Technische Universität Wien, Institut für Technische Informatik, Wien, Österreich, 1994.

[FW01] FEILER, PETER H. und JOHN J. WALKER: *Adaptive feedback scheduling of incremental and design-to-time tasks*. In: *Proceedings of the 23rd International Conference on Software Engineering*, Seiten 318–326. IEEE Computer Society, 2001.

[GCW95] GOVIL, K., E. CHAN und H. WASSERMANN: *Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU*. In: *Proceedings of the 1st Conference on Mobile Computing and Networking MOBICOM'95*, März 1995.

[GL93] GARVEY, A. und V. LESSER: *Design-to-time Real-Time Scheduling*. In: *IEEE Transactions on Systems, Man and Cybernetics, Special Issue on Planning, Scheduling and Control*, Band 23(6), Seiten 1491–1502, 1993.

[GL95] GARVEY, ALAN und VICTOR LESSER: *Design-to-time Scheduling with Uncertainty*. UMass CS Technical Report 95-03, Januar 1995.

[GL96] GARVEY, ALAN und VICTOR LESSER: *Issues in Design-to-time Real-Time Scheduling*. In: *AAAI Fall 1996 Symposium on Flexible Computation*, November 1996.

[Gup04] GUPTA, RAJESH: *Silicon for embedded multimedia processing*. In: GUPTA, RAJESH (Herausgeber): *Design & Test of Computers: Embedded Systems for Real-Time Multimedia*, Seite 345. Coveröffentlichung von IEEE CS und IEEE CASS, September-October 2004.

[HFL96] HULL, D., W. FENG und J. LIU: *Operating System Support for Imprecise Computation*. In: *Flexible Computation in Intelligent Systems: Results, Issues, and Opportunities*. Cambridge, Massachusetts, November 1996.

[Hor90] HORVITZ, ERIC J.: *Reasoning about Beliefs and Actions Under Computational Resource Constraints*. In: HENRION, M., R. D. SHACHTER, L. N. KANAL und J. F. LEMMER (Herausgeber):

Uncertainty in Artificial Intelligence 5, Seiten 301–324. Elsevier Science Publishers B.V., North Holland, 1990.

[HZ01] HANSEN, ERIC A. und SHLOMO ZILBERSTEIN: *LAO * : A heuristic search algorithm that finds solutions with loops.* In: *Artificial Intelligence*, Band 129(1-2), Seiten 35–62, 2001.

[ILO00] ILOG: *ILOG CPLEX 7.0 User's Manual.* ILOG S.A., 2000.

[Int03] INTEL: *Mobile Intel® Pentium® III Processors: Intel SpeedStep® Technology, http://www.intel.com/support/processors/mobile/pentiumiii/ss.htm, Intel Corporation*, 2003.

[Int04] INTEL: *Enhanced Intel® SpeedStep® Technology for the Intel® Pentium® M Processor, White Paper, Order Number: 301170-001, http://www.intel.com/design/intarch/papers/301174.htm*, März 2004.

[JHE04] JERSAK, M., R. HENIA und R. ERNST: *Context-Aware Performance Analysis for Efficient Embedded System Design.* In: *Proceedings of the Design Automation and Test in Europe, Paris, France*, März 2004.

[KGV83] KIRKPATRICK, S., C. D. GELATT und M. P. VECCHI: *Optimization by Simulated Annealing.* In: *Science*, Band 220(4598), Seiten 671–680, Mai 1983.

[KL00] KRISHNA, C. M. und YANN-HANG LEE: *Voltage-Clock-Scaling Adaptive Scheduling Techniques for Low Power in Hard Real-Time Systems.* In: *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000)*, Seiten 156–165. IEEE Computer Society, 2000.

[KL03] KRISHNA, C.M. und Y.-H. LEE: *Voltage-Clock-Scaling Adaptive Scheduling Techniques for Low Power in Hard Real-Time Systems.* In: *IEEE Transactions on Computers*, Band 52(12), Seiten 1586–1593, December 2003.

[Kor90] KORF, R. E.: *Real-Time Heuristic Search.* In: *Artificial Intelligence Journal*, Band 42, Seiten 198–211. Elsevier Science Publishers B.V. (North-Holland), 1990.

[KP97] KIROVSKI, DARKO und MIODRAG POTKONJAK: *System-level synthesis of low-power hard real-time systems*. In: *Proceedings of the 34th annual conference on Design automation conference*, Seiten 697–702. ACM Press, 1997.

[KS97] KRISHNA, C. M. und K. G. SHIN: *Real-Time Systems*. McGraw-Hill, ISBN 0-07-057043-4, 1997.

[LCNS99] LEVY, R., B. CRILLY, B. NARAHARI und R. SIMHA: *Memory Issues in Power-aware Design of Embedded Systems: An Overview*. In: *Second International Workshop on Compiler and Architecture Support for Embedded Systems, CASES'99, Washington D.C.*, 1999.

[Liu00] LIU, JANE W. S.: *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[LJ00] LUO, JIONG und NIRAJ K. JHA: *Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems*. In: *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, Seiten 357–364. IEEE Press, 2000.

[LK03] LEE, YANN-HANG und C. M. KRISHNA: *Voltage-Clock Scaling for Low Energy Consumption in Fixed-Priority Real-Time Systems*. In: *Real-Time Systems*, Seiten 303–317. Kluwer Academic Publishers, 2003.

[LL73] LIU, C. L. und JAMES W. LAYLAND: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. In: *Journal of the Association for Computing Machinery*, Band 20(1), Seiten 46–61, 1973.

[LLM+01] LORENZ, MARKUS, RAINER LEUPERS, PETER MARWEDEL, THORSTEN DRÄGER und GERHARD P. FETTWEIS: *Low-Energy DSP Code Generation Using a Genetic Algorithm*. In: *ICCD '01, Austin/Texas/USA*, September 2001.

[LMD+04] LORENZ, MARKUS, PETER MARWEDEL, THORSTEN DRAEGER, GERHARD FETTWEIS und RAINER LEUPERS: *Compiler based Exploration of DSP Energy Savings by SIMD Operations*. In: *ASPDAC 2004*, Seiten 839–842, 2004.

[LRT92] LEHOCZKY, J.P. und S. RAMOS-THUEL: *An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems*. In: *Proceedings of the 13th Real-Time Systems Symposium (RTSS'92), Phoenix, AZ*, Seiten 110–123, December 1992.

[LS00] LEE, SEONGSOO und TAKAYASU SAKURAI: *Run-time voltage hopping for low-power real-time systems*. In: *Proceedings of the 37th conference on Design automation*, Seiten 806–809. ACM Press, 2000.

[MAAM02] MELHEM, RAMI, NEVINE ABOUGHAZALEH, HAKAN AYDIN und DANIEL MOSSÉ: *Power Management Points In Power-Aware Real-Time Systems*. In: GRAYBILL, R. und R. MELHEM (Herausgeber): *Power Aware Computing*, Seiten 127–152. Plenum/Kluwer Publishers, 2002.

[Mar03] MARWEDEL, PETER: *Embedded System Design*. Kluwer Academic Publishers, 2003.

[MC03] MANZAK, ALI und CHAITALI CHAKRABARTI: *Variable voltage task scheduling algorithms for minimizing energy/power*. In: *IEEE Transactions on Very Large Scale Integrated Systems*, Band 11(2), Seiten 270–276. IEEE Educational Activities Department, 2003.

[MCKT04] MAXIAGUINE, A., S. CHAKRABORTY, S. KÜNZLI und L. THIELE: *Evaluating Schedulers for Multimedia Processing on Buffer-Constrained SoC Platforms*. IEEE Design & Test, 21(5):368–377, September 2004.

[MM73] MARTELLI, A. und U. MONTANARI: *Additive AND/OR Graphs*. In: *Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford, Californien*, Seiten 1–11, August 1973.

[MM78] MARTELLI, A. und U. MONTANARI: *Optimizing Decision Trees Through Heuristically Guided Search*. In: *Communications of the ACM*, Band 21, Seiten 1025–1039, Dezember 1978.

[MSW01] MARWEDEL, PETER, STEFAN STEINKE und LARS WEHMEYER: *Compilation techniques for energy-, code-size-, and run-time-efficient embedded software*. In: *International Workshop on Advanced Compiler Techniques for High Performance and Embedded Processors, Bucharest*, 2001.

[MWV+04] MARWEDEL, PETER, LARS WEHMEYER, MANISH VERMA, STEFAN STEINKE und URS HELMIG: *Fast, predictable and low energy memory references through architecture-aware compilation.* In: *ASPDAC 2004*, Seiten 4–11, 2004.

[NS95] NATALE, MARCO DI und JOHN A. STANKOVIC: *Applicability of Simulated Annealing Methods to Real-Time Scheduling and Jitter Control.* In: *IEEE Real-Time Systems Symposium*, Seiten 190–199, 1995.

[PL95] PARKS, THOMAS M. und EDWARD A. LEE: *Non-Preemptive Real-Time Scheduling Of Dataflow Systems.* In: *Proceedings of International Conference on Acoustics, Speech and Signal Processing (ICASSP 95), Detroit, Michigan.* IEEE, 1995.

[PS01] PILLAI, PADMANABHAN und KANG G. SHIN: *Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems.* In: *ACM Symposium on Operating Systems Principles*, Seiten 89–102, 2001.

[Ram03] RAMSAUER, MARKUS: *Using Simulated Annealing for Hard Real-Time Design-to-Time Scheduling.* In: *Embedded Systems and Applications*, Seiten 109–115. CSREA Press, 2003.

[Ram04] RAMSAUER, MARKUS: *Simultaneously Exploiting Dynamic Voltage Scaling, Execution Time Variations, and Multiple Methods in Energy-Aware Hard Real-Time Scheduling.* In: MÜLLER-SCHLOER, CHRISTIAN, THEO UNGERER und BERNHARD BAUER (Herausgeber): *ARCS*, Band 2981 der Reihe *Lecture Notes in Computer Science*, Seiten 213–227. Springer, 2004.

[RCH04] RAMSAUER, M., M. CODURO und R. HOFFMANN: *Tool Supported Development of Energy-aware or Quality-aware Real-time Applications for Embedded Systems.* In: *Work in Progress Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS 04), Catania, Italy*, Seiten 37–40, 2004.

[RJD98] RAGHUNATHAN, A., N. K. JHA und S. DEY: *High-level power analysis and optimization.* Boston, MA: Kluwer Academic Publishers, 1998.

[RKKL03] ROYCHOWDHURY, DIGANTA, ISRAEL KOREN, C. MANI KRISHNA und Y.-H. LEE: *A Voltage Scheduling Heuristic for Real-Time*

Task Graphs. In: *2003 International Conference on Dependable Systems and Networks*, Seiten 741–750. IEEE Computer Society, 2003.

[RMM03] RUSU, C.A., R. MELHEM und D. MOSSE: *Maximizing system value while satisying time and energy constraints.* In: *IBM Journal of Research and Development*, Band 47(5/6), Seiten 689–702. IBM Corporation, Riverton, NJ, USA, 2003.

[SB98] SUTTON, R. S. und A. G. BARTO: *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, Massachusetts, London, England, 1998.

[SC99] SHIN, YOUNGSOO und KIYOUNG CHOI: *Power conscious fixed priority scheduling for hard real-time systems.* In: *Proceedings of the 36th ACM/IEEE conference on Design automation conference*, Seiten 134–139. ACM Press, 1999.

[Sch03a] SCHWARZFISCHER, THOMAS: *Application of Simulated Annealing to Anytime Scheduling Problems with Additional Timing Constraints.* In: *Proceedings of the Fifth Metaheuristics International Conference (MIC), Kyoto*, August 2003.

[Sch03b] SCHWARZFISCHER, THOMAS: *Quality and Utility - Towards a Generalization of Deadline and Anytime Scheduling.* In: *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS), Triest*. AAAI Press, Juni 2003.

[Sch04] SCHWARZFISCHER, THOMAS: *Closed-Loop Online Scheduling with Timing Constraints and Quality Profiles on Multiprocessor Architectures.* In: *Proceedings of the Workshop on Integrating Planning into Scheduling of the 14th International Conference on Automated Planning and Scheduling (ICAPS), Whistler*, Juni 2004.

[SE04] STASCHULAT, JAN und ROLF ERNST: *Static Running Time Analysis with Symta/P.* http://www.ida.ing.tu-bs.de/research/projects/symta/symtaP-overview.pdf, 2004.

[SRS+01] SHEN, HONG, BADRINATH ROYSAM, CHARLES V. STEWART, JAMES N. TURNER und HOWARD L. TANENBAUM: *Optimal Scheduling of Tracing Computations for Real-time Vascular Landmark Extraction from Retinal Fundus Images.* In: *IEEE Transactions on Information Technology in Biomedicine*, Band 5(1), Seiten 77–91, 2001.

[Ste99] STEINHÖFEL, KATHLEEN: *Stochastic Algorithms in Scheduling Theory*. Doktorarbeit, DISKI 218, infix-Verlag, ISBN 3-89601-218-5, 1999.

[UK03] UNSAL, OSMAN S. und ISRAEL KOREN: *System-Level Power-Aware Design Techniques in Real-Time Systems*. In: *Proceedings of the IEEE, Special Issue on Real-Time Systems*, Band 91, Seiten 1055–1069, July 2003.

[UKK00a] UNSAL, OSMAN S., I. KOREN und C.M. KRISHNA: *Power-Aware Replication of Data Structures in Distributed Embedded Real-Time Systems*. In: *EHPC2000, 14th Annual International Parallel & Distributed Systems Symposium*, Seiten 839–846, Mai 2000.

[UKK00b] UNSAL, OSMAN S., ISRAEL KOREN und C. M. KRISHNA: *High-Level Power-Reduction Heuristics in Large Scale Real-Time Systems*. In: *IEEE International Workshop On Embedded Fault-Tolerant Systems*, September 2000.

[WGL97] WAGNER, THOMAS A., ALAN J. GARVEY und VICTOR R. LESSER: *Design-to-Criteria Scheduling: Managing Complexity through Goal-Directed Satisficing*. In: *Proceedings of the AAAI-97 Workshop on Building Resource-Bounded Reasoning Systems*, Juli 1997.

[WGL98] WAGNER, THOMAS A., ALAN J. GARVEY und VICTOR R. LESSER: *Criteria Directed Task Scheduling*. In: *Journal for Approximate Reasoning (Special Scheduling Issue)*, Band 19, Seiten 91–118. Elsevier Science Inc., Januar 1998.

[WL01] WAGNER, THOMAS A. und VICTOR R. LESSER: *Design-to-Criteria Scheduling: Real-Time Agent Control*. In: RANA, WAGNER & (Herausgeber): *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems: International Workshop on Infrastructure for Scalable Multi-Agent Systems (Part of Series: Lecture Notes in Artificial Intelligence)*, Band 1887, Seiten 128–137. Springer, 2001.

[Wol01] WOLF, WAYNE: *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufman Publishers, 2001.

[WWDS94] WEISER, MARK, BRENT WELCH, ALAN J. DEMERS und SCOTT SHENKER: *Scheduling for Reduced CPU Energy*. In: *Operating Systems Design and Implementation*, Seiten 13–23, 1994.

[YDS95] YAO, F., A. DEMERS und S. SHENKER: *A scheduling model for reduced CPU energy*. In: *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, Seiten 374–382. IEEE Computer Society, 1995.

[Yea98] YEAP, G. K.: *Practical Low Power Digital VLSI Design*. Boston, MA: Kluwer Academic Publishers, 1998.

[YMW+02] YANG, PENG, PAUL MARCHAL, CHUN WONG, STEFAAN HIMPE, FRANCKY CATTHOOR, PATRICK DAVID, JOHAN VOUNCKX und RUDY LAUWEREINS: *Managing dynamic concurrent tasks in embedded real-time multimedia systems*. In: *Proceedings of the 15th International Symposium on System Synthesis*, Seiten 112–119, 2002.

[YWM+01] YANG, P., C. WONG, P. MARCHAL, F. CATTHOOR, D. DESMET, D. VERKEST und R. LAUWEREINS: *Energy-aware Runtime Scheduling for Embedded Multiprocessor SoCs*. In: *IEEE Design & Test of Computers*, Band 19(3), September 2001.

[Zil93] ZILBERSTEIN, S.: *Operational Rationality through Compilation of Anytime Algorithms*. Doktorarbeit, Computer Science Division, University of California at Berkeley, 1993.

[ZMC03] ZHU, DAKAI, RAMI G. MELHEM und BRUCE R. CHILDERS: *Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multiprocessor Real-Time Systems*. In: *IEEE Transactions on Parallel and Distributed Systems*, Band 14, Seiten 686–700, Juli 2003.