Lehrstuhl für Realzeit–Computersysteme

Optimierte Auslegung verteilter Realzeitsysteme

Dipl.–Ing. univ. Herbert Thielen

Lehrstuhl für Realzeit–Computersysteme

Optimierte Auslegung verteilter Realzeitsysteme

Dipl.–Ing. univ. Herbert Thielen

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor–Ingenieurs (Dr.–Ing.)

genehmigten Dissertation.

Vorsitzender: Univ.–Prof. Dr.–Ing. W. Boeck

Prüfer der Dissertation:
1. Univ.–Prof. Dr.–Ing. G. Färber
2. Univ.–Prof. Dr.–Ing. Dr.–Ing. habil. F. Schneider

Die Dissertation wurde am 30.05.2000 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 28.10.2000 angenommen.

Vorwort

Die vorliegende Arbeit entstand hauptsächlich während meiner Tätigkeit als wissenschaftlicher Assistent am Lehrstuhl für Prozeßrechner, heute Lehrstuhl für Realzeit–Computersysteme, der Technischen Universität München.

Mein besonderer Dank gilt Herrn Prof. Dr.–Ing. Georg Färber, der diese Arbeit ermöglichte, und der durch viele wertvolle Gespräche und durch seine Geduld wesentlich zum Gelingen beigetragen hat.

Für die Übernahme des Zweitberichts möchte ich Herrn Prof. Dr.–Ing. Friedrich Schneider herzlich danken.

Mein Dank gilt auch allen ehemaligen Kollegen am Lehrstuhl für Prozeßrechner, insbesondere Herrn Prof. Dr.–Ing. Jürgen Quade und Herrn Dr.–Ing. Klaus Gresser für ihre wertvolle Unterstützung und die vielen hilfreichen Diskussionen.

Weiterhin möchte ich mich auch bei meinem jetzigen Arbeitgeber für die kulante Arbeitsfreistellung zur Fertigstellung der Dissertation bedanken.

Ganz besonderer Dank gilt aber meiner Frau Astrid und meinen Kindern Sonja, Carolina und Jonas, die mich mit ihrer Zuversicht, ihrer Liebe und Kraft in meiner Arbeit immer wieder bestärkten.

Heroldsberg, im Mai 2000

Inhaltsverzeichnis

1 Einleitung **1**

2 Stand der Forschung **5**
2.1 Entwurf von Realzeitsystemen 5
2.2 Optimierungsverfahren 7

3 Systemmodell und Realzeitnachweis **10**
3.1 Beschreibung des automatisierten Entwurfsverfahrens 10
3.2 Formale Beschreibung der Hardware–Komponenten 11
3.3 Formale Beschreibung der Software–Strukturen 13
3.4 Partitionierung 16
3.4.1 Automatisierung mit stochastischen Verfahren 17
3.4.2 Echtzeitnachweis 18
3.5 Ergebnis des Entwurfsverfahrens 19
3.6 Untersuchte Modelle 19

4 Anwendung stochastischer Optimierungsverfahren **21**
4.1 Problemspezifische Funktionen 21
4.1.1 Startpunktwahl 21
4.1.2 Bewertungsfunktion 22
4.1.3 Änderungsfunktion 34
4.2 Simulated Annealing 40
4.2.1 Algorithmus 40
4.2.2 Verhalten des Algorithmus 41
4.3 Threshold Accepting 46
4.3.1 Algorithmus 46
4.3.2 Verhalten des Algorithmus 47
4.4 Record–to–Record–Traveling 50
4.4.1 Algorithmus 50
4.4.2 Verhalten des Algorithmus 50
4.5 Great Deluge Algorithm 54
4.5.1 Algorithmus 54

4.5.2 Verhalten des Algorithmus . . . 55
4.6 Genetische Algorithmen . . . 59
4.6.1 Biologisches Vorbild . . . 59
4.6.2 Grundalgorithmus . . . 61
4.6.3 Kodierung . . . 62
4.6.4 Genetische Operatoren . . . 65
4.6.5 Population und Erbgutaustausch . . . 67
4.6.6 Verhalten des Algorithmus . . . 68

5 Heuristik zur Systemauslegung **75**
5.1 Unabhängige Tasks . . . 75
5.1.1 Ermittlung der nötigen Gesamt–Rechenleistung . . . 75
5.1.2 Ermittlung der nötigen Rechneranzahl . . . 76
5.1.3 Ermittlung der Task–Rechenlasten . . . 76
5.1.4 Ausnutzung von Restkapazitäten . . . 77
5.1.5 Taskallokation . . . 79
5.2 Tasks mit Kommunikationsbeziehungen . . . 80
5.3 Einbeziehung von Restriktionen . . . 81
5.3.1 Allokation auf denselben Rechner . . . 81
5.3.2 Allokation auf unterschiedliche Rechner . . . 82
5.3.3 Speziell erforderlicher Rechnertyp . . . 82
5.4 Algorithmus . . . 82
5.5 Aufwandsabschätzung . . . 85
5.6 Anwendungsergebnisse . . . 85

6 Diskussion der Ergebnisse **87**
6.1 Gegenüberstellung der Verfahren . . . 87
6.1.1 Ergebnisse aus Beispiel E . . . 87
6.1.2 Ergebnisse mit Beispiel R . . . 90
6.1.3 Heuristikergebnis als Startpunkt . . . 91
6.2 Möglichkeiten der Verbesserung . . . 95

7 Zusammenfassung **97**

Anhang **99**

A Implementierte Algorithmen **99**
A.1 Simulated Annealing . . . 99
A.2 Threshold Accepting . . . 101
A.3 Record–to–Record–Traveling . . . 101
A.4 Great Deluge Algorithm . . . 101
A.5 Genetische Algorithmen . . . 104

A.5.1 Populationsstruktur . . . 104
A.5.2 Initiale Population . . . 105
A.5.3 Fitness und Selektion . . . 105
A.6 Vergleichsverfahren . . . 105
A.6.1 Monte–Carlo . . . 106
A.6.2 Hill–Climbing . . . 106

B Verwendete Beispiele **108**
B.1 Systemmodell . . . 108
B.2 Taskmodell E . . . 108
B.3 Taskmodell R . . . 110

Abkürzungen **112**

Glossar **113**

Variablenverzeichnis **116**

Literaturverzeichnis **119**

Abbildungsverzeichnis

1.1 Rechnergestützte Systemauslegung . . . 2

3.1 Beispielhaftes Rechnersystem . . . 12
3.2 Attribute des TDMA–Kommunikationssystems . . . 13
3.4 Beispiel eines Taskgraphen . . . 14
3.3 Taskparameter . . . 14
3.5 Beispiel für einen Ereignisstrom . . . 15
3.6 Optimierungsschleife . . . 16
3.7 Beispiel für verbotene Zonen . . . 17

4.1 Anpassung d. Rechenleistung . . . 23
4.2 Preis–/Leistungskurve von 80x86–Boards . . . 25
4.3 Preis–/Leistungs–Verhältnis von 80x86–Boards . . . 25
4.4 Preisvergleich von Komplett–PC's . . . 26
4.5 Kostenkurve, erweitert für virtuelle Rechner . . . 28
4.6 Kostenkurve mit lastbezogenem Anteil (Ausschnitt) . . . 30
4.7 Kommunikationskosten für ein Kommunikationselement . . . 32
4.8 Spielraumverteilung bei Kommunikation . . . 36
4.9 Algorithmus „Simulated Annealing" . . . 40
4.10 SA: Einfluß von α in Beispiel E . . . 42
4.11 SA: Einfluß von α in Beispiel R . . . 42
4.12 SA: Einfluß von c_{temp} in Beispiel E . . . 43
4.13 SA: Einfluß von c_{temp} in Beispiel R . . . 44
4.14 SA: Kostenverlauf für unterschiedliche Werte von c_{temp} bei Modell R . . . 44
4.15 Algorithmus „Threshold Accepting" . . . 46
4.16 TA: Einfluß von α in Beispiel E . . . 47
4.17 TA: Einfluß von α in Beispiel R . . . 48
4.18 TA: Einfluß von c_{temp} in Beispiel E . . . 49
4.19 TA: Einfluß von c_{temp} in Beispiel R . . . 49
4.20 Algorithmus „Record–To–Record–Traveling" . . . 50
4.21 RTR: Einfluß von Abweichung D in Beispiel E . . . 51
4.22 RTR: Einfluß von Abweichung D in Beispiel R . . . 51
4.23 RTR: Einfluß von $t_{\text{s,max}}$ in Beispiel E . . . 52

4.24 RTR: Einfluß von $t_{\mathrm{s,max}}$ in Beispiel R . . . 53
4.25 Algorithmus „Great Deluge“ . . . 54
4.26 GD: Einfluß von Absenkung D in Beispiel E . . . 55
4.27 GD: Einfluß von Absenkung D in Beispiel R . . . 56
4.28 GD: Einfluß von $t_{\mathrm{s,max}}$ in Beispiel E . . . 57
4.29 GD: Einfluß von $t_{\mathrm{s,max}}$ in Beispiel R . . . 58
4.30 Duplikation von DNS . . . 59
4.31 Bildung von Keimzellen . . . 60
4.32 Optimierungszyklus der GA . . . 62
4.33 Einfaches Crossover . . . 66
4.34 Selektion mit Roulette–Verfahren . . . 68
4.35 GA: Einfluß von G_{break} auf Modell E . . . 69
4.36 GA: Kostenverlauf bei $G_{\mathrm{break}} = 300$ . . . 70
4.37 GA: Einfluß der Populationsgröße in Beispiel E . . . 71
4.38 GA: Einfluß von Populationsgröße und variablem G_{break} . . . 71
4.39 GA: Weitere Parameter–Varianten . . . 72
4.40 GA: Differenzen der Selektionswahrscheinlichkeiten . . . 73

5.1 Bestimmung der Worst–Case–Auslastung . . . 76
5.2 Beispiel für RZAF–Summen . . . 78
5.3 Beispiel für Startwertsuche . . . 84

6.1 Kostenvergleich für Beispiel E . . . 88
6.2 Aufwandsvergleich für Beispiel E . . . 88
6.3 Kostenvergleich für Beispiel R . . . 91
6.4 Aufwandsvergleich für Beispiel R . . . 92
6.5 Kostenvergleich mit Heuristik–Ergebnis als Startpunkt . . . 93
6.6 Aufwandsvergleich mit Heuristik–Ergebnis als Startpunkt . . . 93
6.7 RTR: Einfluß von D mit Startpunkt aus Heuristik . . . 94

A.1 „Simulated Annealing“, implementierte Version . . . 100
A.2 „Record–to–Record–Traveling“, implementierte Version . . . 102
A.3 „Great Deluge Algorithm“, implementierte Version . . . 103
A.4 GA: Populationsstruktur . . . 104
A.5 Algorithmus „Monte–Carlo“ . . . 106
A.6 Algorithmus „Hill–Climbing“ . . . 107

B.1 Task in Beispiel E . . . 109

Tabellenverzeichnis

5.1 Gruppenbildung für das Beispiel aus Abbildung 5.3 85

6.1 Ergebnisvergleich für Beispiel E 89
6.2 Ergebnisvergleich für Beispiel R 90
6.3 Ergebnisvergleich mit Heuristik–Wert als Startpunkt 92

B.1 Rechnertypen der Beispiel–Modelle 109
B.2 Task–Parameter Modell R 111

1 Einleitung

Verteilte Realzeitsysteme gewinnen immer mehr an Bedeutung. Dies hat mehrere Ursachen: Zum einen lassen sich kürzere Reaktionszeiten durch die Verteilung der Last auf mehrere Verarbeitungseinheiten (Prozessoren) erreichen, zum zweiten lassen sich die Systeme durch Verwendung der Komponenten mit dem besten Preis–/Leistungsverhältnis kostengünstiger realisieren, und zum dritten steigt dadurch die Verfügbarkeit des Systems an. Überdies hat die Kommunikationstechnik — insbesondere auch durch die Entwicklungen in den Bereichen der Feldbussysteme und der Internettechnologie — die Verteilung der Aufgaben auf mehrfach vorhandene Verarbeitungseinheiten vereinfacht. Auch der Einsatzbereich verteilter Realzeitsyteme hat sich erweitert. Zeitaspekte gewinnen insbesondere im Bereich der eingebetteten Systeme zunehmend an Bedeutung.

Nachteilig sind allerdings die zusätzlichen Aufwände, die durch die Kommunikation zwischen den Verarbeitungseinheiten erforderlich sind. Weiterhin sind technische Realisierungen von Systemen zunehmend durch wirtschaftliche Überlegungen geprägt. Der Kostendruck macht es notwendig, Systeme optimal für die gegebene Aufgabenstellung auszulegen. Optimal bedeutet in diesem Zusammenhang, keine teuren, unnötigen Reserven einzuplanen. Dies gilt insbesondere, wenn Echtzeitsystem in Massenprodukte (z. B. Konsumgüter, KFZ–Anwendungen) integriert werden [ZA95].

Betriebswirtschaftliche Aspekte spielen daher während der Planungsphase eines verteilten Echtzeitsystems eine maßgebliche Rolle. Eine manuelle Planung ist aufgrund der Komplexität der Aufgabenstellung allerdings unmöglich. Ziel dieser Arbeit ist es, ein rechnergestütztes Planungsverfahren für verteilte Echtzeitsysteme zu ermöglichen.

Bei einem rechnergestützten Planungsverfahren (Abbildung 1.1) werden in einem ersten Schritt die existierenden Anforderungen (Requirements) formal beschrieben. In einem zweiten Schritt ist es notwendig, die Softwarestruktur — als Ergebnis des Design–Prozesses — ebenfalls, zusammen mit den dynamischen Anforderungen an die Software, formal zu beschreiben (Taskmodell, Verarbeitungseinheiten innerhalb der Tasks, Beziehungen der Verarbeitungseinheiten untereinander, Synchronisationspunkte usw. und Ereignisströme). Wird nun in einem weiteren dritten Schritt die vorhandene Hardware (Systemkonfiguration, Rechenleistung) definiert, kann im vierten Schritt ein sogenannter Echtzeitnachweis durchgeführt werden. Hierbei wird berechnet, ob die gestellten

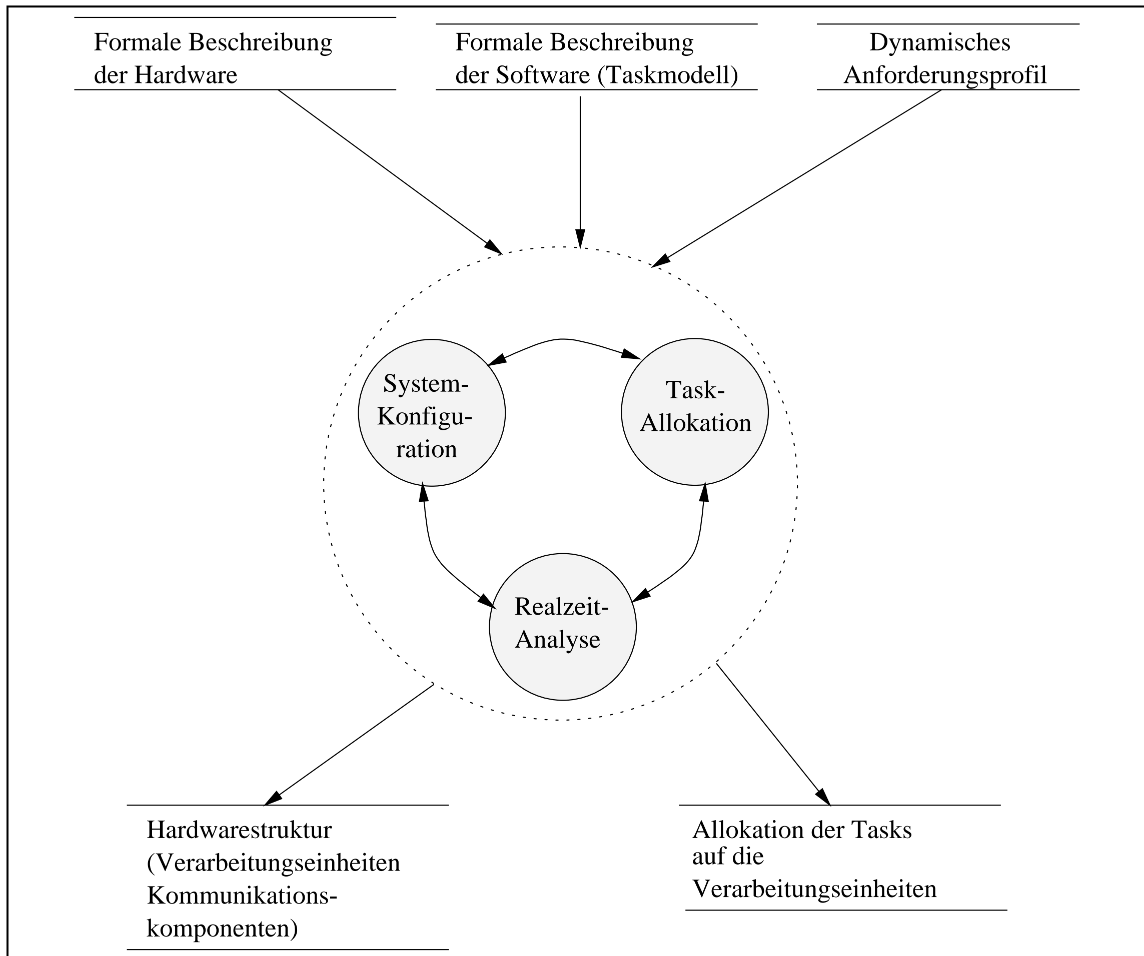


Abbildung 1.1: Rechnergestützte Systemauslegung

zeitlichen Anforderungen aufgrund der zur Verfügung stehenden Hardware und der Systemstruktur zu erfüllen sind oder nicht. Durch Heranziehen betriebswirtschaftlicher Anforderungen variiert man nun die bisher als vorhanden angenommene Systemstruktur (Hardware) und optimiert diese aufgrund der Kosten (also wiederholter Durchlauf der Schritte 3 und 4).

Eine rechnergestützte Planung für verteilte Realzeitsysteme ist sehr komplex. Zum einen werden die formalen Methoden zur Beschreibung von Anforderungen, der Systemstruktur und auch der Verarbeitungseinheiten (Hardware) — und zwar sowohl bezüglich der Funktionalität als auch der Kosten — benötigt, zum anderen sind Verfahren notwendig, mit denen aufgrund der angegebenen Informationen der Echtzeitnachweis durchgeführt werden kann, und drittens ist ein Optimierungsverfahren (Algorithmus) erforderlich, welches in endlicher Rechenzeit eine möglichst kostengünstige Auslegung des verteilten Systems berechnet.

Methoden zur formalen Beschreibung von Realzeitsystemen und ein Verfahren zum Nachweis der schritthaltenden Verarbeitung sind in [Gre93] dargestellt. Formale Methoden zur einfachen Beschreibung der Systemstruktur und die Eignung unterschiedlicher Optimierungsverfahren für die kostengünstige Auslegung eines verteilten Echtzeitsystems wurden in dieser Arbeit untersucht. Das automatische Entwurfsverfahren berücksichtigt dabei auch durch den Systemingenieur vorgegebene Anforderungen, die beispielsweise aus dem Bereich Fehlertoleranz stammen können (zwei Tasks müssen aus Fehlertoleranz-Gründen auf zwei unterschiedlichen Rechnern ablaufen). Damit befaßt sich die Arbeit mit einem Bereich des Entwurfs von Realzeitsystemen, der nahe bei der Realisierung bzw. Implementierung liegt. Zum einen ist das Problem selbst genau definiert, sodaß ein Modell des technischen Prozesses existiert, zum anderen ist auch die prinzipielle Lösung bekannt [Gre93].

Da die Variationsbreite zur Realisierung eines verteilten Systems unendlich groß ist (z. B. durch die Einsatzmöglichkeit unterschiedlichster Prozessoren und verschiedenster Verfahren zur Kommunikation zwischen den einzelnen Komponenten bzw. Rechenprozessen), wird der Lösungsraum auf einige Elemente eingeschränkt.

Zunächst wird im folgenden Kapitel ein Überblick über für die Arbeit relevante Entwurfsmethodiken und Optimierungsverfahren gegeben.

In Kapitel 3 wird das prinzipielle Verfahren zur automatischen optimalen Systemauslegung verteilter Realzeitsysteme vorgestellt. Hier werden auch die Modelle beschrieben, mit denen das Verfahren und insbesondere die verschiedenen Optimierungsalgorithmen untersucht wurden.

Die Abbildung der Problemstellung auf die Optimierungsverfahren und die detaillierte Beschreibung der Verfahren ist in Kapitel 4 enthalten. Auch die Auswertung der durchgeführten Experimente wird hier dargelegt.

Der Einsatz vorhandenen Wissens über das Modell kann hilfreich sein, um schneller und zu besseren Ergebnissen bei der Optimierung der Systemkonfiguration zu gelangen. Kapitel 5 zeigt, wie dies für eine heuristische Systemauslegung genutzt werden kann.

Die bei der Untersuchung der verschiedenen Optimierungsalgorithmen und der Konfigurations–Heuristik gewonnen Ergebnisse werden in Kapitel 6 einander gegenübergestellt und diskutiert. Aus diesen Ergebnissen werden mögliche Wege zur weiteren Verbesserung des automatischen Konfigurationsverfahrens abgeleitet.

Die Arbeit schließt in Kapitel 7 mit einer Zusammenfassung.

2 Stand der Forschung

Für die Optimierung von Realzeitsystemen sind einerseits die Entwicklungen auf dem Gebiet der Entwurfsmethodiken von Realzeitsystemen von Bedeutung, und auf der anderen Seite Untersuchungen zu allgemeinen Optimierungsverfahren, die für die gestellte Aufgabe Verwendung finden können. Eine Auswahl von Veröffentlichungen zu diesen Themen wird in den folgenden Abschnitten vorgestellt.

2.1 Entwurf von Realzeitsystemen

Wesentliche Bestandteile beim Entwurf von Realzeitsystemen sind die Partitionierung und die Systemanalyse (Verifikation), die beide in einem engen Zusammenhang stehen und aufeinander abgestimmt sein müssen.

Ohne Berücksichtigung des großen Bereiches Hardware–/Software–Codesign beschränkt sich die Partitionierung auf die Systemkonfiguration (z. B. Anzahl und Typ von Rechnerknoten) und die Allokation der Tasks auf die Rechnerknoten.

Zu standardisierten Spezifikationssprachen wie SDL [ITU94] existieren Ansätze zur Einbeziehung von Zeitbedingungen in die Spezifikation sowie darauf aufbauend Methoden zur Verifikation der Zeitbedingungen. Das Konzept von [DHHM95] erweitert die SDL–Systemspezifikation um Warteschlangenstationen, mit deren Hilfe eine Leistungsbewertung durch Simulation erfolgen kann. Ein analytischer Realzeitnachweis ist hier nicht vorgesehen.

Für ein verteiltes Rechnersystem aus gleichartigen Rechnerknoten beschreibt Kopetz in [Kop86] Verfahren zur Taskallokation unter Einhaltung der Realzeitbedingung. Das System setzt ein Zeitscheiben–Scheduling voraus und definiert eine spezifische Kommunikationsinfrastruktur.

Tindell, Burns und Wellings definieren in [TBW92] ein Verfahren zur Taskallokation in einem verteilten Realzeitsystem. Die Tasks werden periodisch aktiviert und kommunizieren über einen Token Bus. Dabei ist die Anzahl der Rechner fest vorgegeben, und die

Rechner sind identisch. Restriktionen bei der Taskallokation, z. B. Erfordernis bestimmter Rechner, werden berücksichtigt. Als Scheduling–Verfahren kommt Rate Monotonic Scheduling zum Einsatz.

Ein System zur dynamischen Taskallozierung und –migration, insbesondere bei Realzeitsystemen mit weichen Zeitbedingungen, beschreibt Triller in [Tri94]. Er unterscheidet lokale und globale Tasks: Letztere sind Tasks, die auf andere Prozessoren migrieren müssen, um ihre Deadlines einzuhalten. Als Scheduling–Verfahren wird Deadline Monotonic Priority Assignment eingesetzt [LW82]. Vorteile für Systeme mit harten Zeitbedingungen werden in dieser Arbeit nur dann gesehen, wenn keine statische Allokation möglich ist (z. B. wegen unvorhersehbarer Ereignisse), oder um den Komfort zu nutzen, keine statische Allokation vorgeben zu müssen. Lauzac und andere untersuchten in [LMM98] statische und dynamische Taskallozierung bei Rate Monotonic Priority Assignment [LL73] und kamen dabei ebenfalls zu dem Ergebnis, daß statische Allokation für harte Realzeitsysteme besser geeignet ist.

Wang stellt in [Wan99, WF98, WF99] ein auf MSC– (Message Sequence Charts) und SDL–Spezifikationen basierendes Verfahren vor, um die zeitlichen und funktionellen Aspekte eines harten Realzeitsystems zu beschreiben. Dabei werden auch Heuristiken für die Taskzuordnung an die Rechner sowie ein Analyseverfahren zum Nachweis der Realzeitfähigkeit entwickelt. Kommunikation und Präzedenzbeziehungen werden berücksichtigt; für das Scheduling kommt ein prioritätengesteuertes Verfahren zum Einsatz.

Analyseverfahren zum Nachweis der Einhaltung aller Realzeitbedingungen basieren entweder auf der Ermittlung der maximalen Prozessorauslastung im Worst–Case oder auf der Analyse der maximalen Antwortzeiten der Tasks. Bei der Analyse der Prozessorauslastung wächst die Komplexität sehr schnell mit der Anzahl der Tasks und der maximalen Deadline. In [MA98] wird gezeigt, daß die Analyseaufwand von Systemen mit Scheduling nach festen Prioritäten selbst bei konstanter Anzahl von Tasks unter Umständen sehr hoch sein kann. Auch bei der Analyse der maximalen Antwortzeiten, bei der für alle Tasks untersucht wird, ob die Antwortzeit im Worst–Case immer kleiner oder gleich der Task–Deadline ist, spielt die Komplexität der Analyse eine große Rolle. Daher werden oft hinreichende, aber nicht notwendige Bedingungen formuliert, mit deren Hilfe eine weniger komplexe Analyse möglich ist. Das Analyseergebnis ist damit nicht genau, sondern eine Abschätzung zur sicheren Seite hin [ABD+95].

Bei der Kommunikation von Tasks entstehen zwischen den Tasks Präzedenzbeziehungen. Werden diese Beziehungen bei der Analyse ignoriert, wird davon ausgegangen, daß alle Tasks im Worst–Case gleichzeitig lauffähig werden können [ATB+91]. Dies ist jedoch eine verschärfende Annahme, d. h. das System wird strenger bewertet als notwendig. Die Präzedenzbeziehungen können durch Offsets zur Startzeit der Tasks in der Analyse berücksichtigt werden [Bur93, Tin94]. Ein etwas anderer Ansatz wird in [BB97] verfolgt: Hier wird eine Menge von Tasks mit Offset ungleich Null durch eine

„composite task“ mit Offset Null in der Analyse repräsentiert.

Gresser entwickelt in [Gre93, Gre95] Methoden zur Realzeitanalyse ereignisgesteuerter Systeme, basierend auf einer formalen Beschreibung der technischen Prozesse in Form von sogenannten Ereignisströmen, sowie einer Beschreibung der Tasks und des Mehrrechnersystems. Die Allokation der Tasks auf die Rechner wird dabei bereits vorausgesetzt, als Scheduling–Verfahren wird Deadline–Scheduling eingesetzt. Kommunikation zwischen Tasks und daraus resultierende Präzedenzbeziehungen werden in der Analyse berücksichtigt. Ebenfalls sind Abhängigkeiten zwischen Ereignissen erlaubt, die unterschiedliche Tasks anregen. Das Analyseverfahren von Gresser wird für die Untersuchungen in dieser Arbeit verwendet; eine eingehendere Beschreibung ist in Kapitel 3 zu finden.

2.2 Optimierungsverfahren

Ziel der Optimierung ist das Auffinden einer Systemkonfiguration, mit der unter Minimierung der Kosten des Rechnersystems die Einhaltung der geforderten Randbedingungen, insbesondere der Realzeitbedingungen, möglich ist. Da bereits ohne Kostenoptimierung das Problem der Taskallokation unter Einhaltung von Realzeitbedingungen NP–vollständig ist, scheiden eine Berechnung des globalen Kostenminimums oder eine vollständige Suche wegen des zu hohen Aufwandes aus; stattdessen bieten sich heuristische Optimierungsverfahren zur Ermittlung einer (suboptimalen) Systemkonfiguration an.

Das **Monte–Carlo–Verfahren** ist ein sehr einfaches Optimierungsverfahren: Ausgehend von einem zufälligen Startpunkt wird eine bestimmte Anzahl von Konfigurationsänderungen durchgeführt; die am besten bewertete Konfiguration stellt das Optimierungsergebnis dar. Es handelt sich also um eine rein zufällige Suche ohne zielgerichtete Komponenten. Daher eignet sich das Verfahren zumindest zur vergleichenden Beurteilung zielgerichteter Ansätze in anderen Verfahren.

Beim **Hill–Climbing–Verfahren** werden ebenfalls ausgehend von einem zufälligen Startpunkt zufällig ausgewählte Konfigurationsänderungen versucht, doch wird hier nur eine solche Änderung verwendet, die eine bessere Bewertung erhält als die zuvor verwendete Konfiguration; diese verbesserte Konfiguration dient dann als Ausgangspunkt für die nächste Iteration. Hill–Climbing enthält also eine stark zielgerichtete Komponente, hat aber den Nachteil, daß es in lokalen Optima stagniert.

Kirkpatrick, Gellatt und Vecchi [KGV83] entwickelten ein Verfahren namens **Simulated Annealing** in Anlehnung an den physikalischen Vorgang des Abkühlens einer Metallschmelze (Annealing): Wenn zu schnell abgekühlt wird, kann bei einer bestimmten Temperatur kein Gleichgewichtszustand erreicht werden, und es entstehen durch Fehler

in der Gitterstruktur (lokale Energieminima) amorphe Strukturen im Metall. Erfolgt das Abkühlen langsamer, so ist zunächst noch genügend Energie vorhanden, um aus Strukturen mit lokalen Energieminima zu einem globalen Energieminimum zu finden, und das Material erhält eine optimale Gitterstruktur. Übertragen auf das Optimierungsproblem werden daher anfangs (bei „hoher Temperatur") viele Konfigurationsänderungen erlaubt, auch wenn sie zu einer schlechteren Bewertung führen, während mit allmählicher „Abkühlung" immer weniger Verschlechterungen akzeptiert werden und schließlich nur noch Verbesserungen erlaubt sind. Der Algorithmus wird in Kapitel 4.2.1 näher beschrieben.

Simulated Annealing wurde bereits für die Optimierung unterschiedlicher NP–harter Probleme angewendet, z. B. in der Partitionierung elektronischer Schaltungen, für die Allokation von Tasks [TBW92], für das Routing von Schaltungslayouts oder für Traveling–Salesman–Probleme (z. B. [Ott94]).

Dueck und Scheuer [DS90, DSW93] formulierten **Threshold Accepting** als vereinfachte Abwandlung von Simulated Annealing (s. Kap. 4.3). In den dort durchgeführten Untersuchungen zum Traveling–Salesman–Problem erreichten sie damit bessere Ergebnisse als mit Simulated Annealing, wobei gleichzeitig weniger Versuche nötig waren. Auch zur Generierung von fehlerkorrigierenden Codes, die ein sehr unstetiges Verhalten der Bewertungsfunktion aufweisen, wurde das Verfahren erfolgreich eingesetzt. Da das Akzeptanzkriterium gegenüber Simulated Annealing deterministisch ist, konnte durch Einsatz einer ebenfalls deterministischen Änderungsfunktion auch ein insgesamt vorhersagbares Verhalten des Optimierungsverfahrens für das Traveling–Salesman–Problem entwickelt werden. Dieses zeigte mit der stochastischen Variante vergleichbare Resultate bei einer kleineren Anzahl von nötigen Versuchen, ermittelte aber Ergebnisse mit einer höheren Varianz. Der in der Literatur dargestellte Vorteil von Threshold Accepting, daß im Vergleich zu Simulated Annealing wegen der vereinfachten Akzeptanzbedingung (Wegfall einer Exponentialfunktion) eine kürzere Rechenzeit nötig sei, greift bei dem in der vorliegenden Arbeit untersuchten Problem nicht, da hier der Hauptanteil der benötigten Rechenzeit in der Realzeitanalyse liegt.

Für Simulated Annealing wie auch für Threshold Accepting wurde die Konvergenz der Optimierung theoretisch nachgewiesen

Weitere von Dueck aus Threshold Accepting abgeleitete Varianten sind **Record–to–Record–Traveling** und **Great–Deluge** [Due93]. Anwendungen auf Traveling–Salesman–Probleme und im Chip Placement ergaben etwa doppelt so lange Rechenzeiten wie bei Threshold Accepting bei vergleichbaren Ergebnissen.

Zu Simulated Annealing wurden weitere Varianten beschrieben, z. B. von Ingber [Ing93, Ing89], die insbesondere eine schnellere Konvergenz der Optimierung durch adaptive Modifikation von Optimierer–Parametern ermöglichen. In [IR92] erhalten Ingber und Rosen damit bei Anwendung auf eine Suite mehrparametriger Testfunktionen im Vergleich zu Genetischen Algorithmen verläßlicher und schneller gute Ergebnisse.

Vorbild für die **genetischen Algorithmen** ist die Darwinsche Selektionstheorie, die davon ausgeht, daß sich die verschiedenen ihren jeweiligen Biotopen sehr gut angepaßten Arten der Tier– und Pflanzenwelt durch kleine Veränderungen (Mutationen) und Auswahl der am besten Angepaßten (Selektion) aus gemeinsamen Ur–Vorfahren (Einzellern) gebildet haben. Insbesondere die geschlechtliche Fortpflanzung wird bei den genetischen Algorithmen imitiert: Hier wird das Erbgut eines neuen Lebewesens aus der Mischung der elterlichen Erbinformationen zusammengestellt; die Nachkommen haben dadurch neue, von denen der Eltern leicht abweichende Eigenschaften.

Aus dem Verständnis der biologischen Evolution wurden im wesentlichen zwei algorithmische Modelle abgeleitet: Rechenberg [Rec73] entwickelte die „Evolutionsstrategien“, die insbesondere von Schwefel [Sch76] weiterentwickelt wurden. Wegen der Beschränkung auf Optimierungsprobleme mit realwertigen Parametern wird dieser Ansatz hier nicht weiter untersucht. Unter dem Begriff „Genetische Algorithmen“ wird üblicherweise die allgemeinere Umsetzung der biologischen Abläufe in einen Optimierungsalgorithmus von Holland [Hol73, Hol75] verstanden. Aufbauend auf diese Ideen wurden unzählige Variationen entwickelt und für verschiedene Anwendungsfälle untersucht. Eine Einführung in die Evolutionsstrategien und die Genetischen Algorithmen mit einer Skizzierung verschiedener Anwendungsfälle findet sich in [SHF94], weitere Übersichten sind in [Abl87, Hei94, Mut82, Wal94] gegeben. Wesentliche Untersuchungen und Weiterentwicklungen der Genetischen Algorithmen wurden von Goldberg durchgeführt [Gol89].

Ein Vorteil der Genetischen Algorithmen ist die leichte Parallelisierbarkeit, sodaß der Optimierungsprozeß durch Einsatz von speziellen Mehrprozessorsystemen (z. B. [Sch93]) oder vernetzten Workstations stark beschleunigt werden kann.

Die Genetischen Algorithmen finden in vielen komplexen Optimierungsproblemen Anwendung, z. B. Stundenplanoptimierung [CDM92], Allokation von Tasks [AD95] oder der Schaltungspartitionierung [Hul92, TB91].

Bei der Anwendung zur Schaltungspartitionierung [Hul92] definiert Hulin Simulated Annealing als Spezialfall der Genetischen Algorithmen, bei dem die Population nur aus zwei Elementen besteht und nur Mutation angwendet wird. In einem kurzen Vergleich mit seiner problemangepaßten Version der Genetischen Algorithmen zeigten diese eine schnellere Konvergenz als Simulated Annealing. Gleichzeitig stellt Hulin aber fest, daß ohne die spezielle Kodierung die Genetischen Algorithmen für seine Problemstellung nur mäßige Lösungen erzielen.

3 Systemmodell und Realzeitnachweis

3.1 Beschreibung des automatisierten Entwurfsverfahrens

Ein Echtzeitsystem ist dadurch gekennzeichnet, daß ein Rechensystem die durch einen technischen Prozeß erzeugten Ereignisse (Inputs) verarbeitet und zu den richtigen Zeitpunkten Ausgaben (Outputs) generiert. Die Aufgabe des Rechensystems ist damit, innerhalb der durch den technischen Prozeß definierten maximal zulässigen Reaktionszeit (Deadline) gültige Ausgaben bereit zu stellen. Dabei soll das Rechensystem auf der einen Seite robust und zuverlässig, auf der anderen Seite auch kostengünstig sein.

Aufgrund der Anforderungen (Komplexität der Aufgabe, Kosten) kann das Rechensystem als ein Ein–Prozessor Echtzeitrechner oder als ein verteiltes Echtzeitsystem aufgebaut werden, das aus mehreren Verarbeitungseinheiten besteht, die über ein Kommunikationssystem gekoppelt sind. Während ein einzelner Echtzeitrechner einfacher zu realisieren ist, ist ein verteiltes System leistungsfähiger und bietet Vorteile im Bereich Fehlertoleranz. Jedoch ist die Auslegung des Systems, insbesondere die Auswahl der Hardware–Struktur (Verarbeitungseinheiten und Kommunikationskomponenten) und die Verteilung der einzelnen Rechenprozesse (Tasks) auf die Verarbeitungseinheiten, aufgrund der Parametervielfalt und des sich daraus ergebenden immens großen Lösungsraumes derart komplex, daß nur ein automatisiertes Verfahren in Betracht kommt.

Das automatisierte Entwurfsverfahren hat zur Aufgabe, aufgrund einer formalen Beschreibung der Software–Struktur, des dynamischen Anforderungsprofils und einer formalen Beschreibung der zur Verfügung stehenden Hardware–Komponenten eine möglichst kostengünstige Hardware–Struktur (Instanziierung der Hardware–Komponenten) zu berechnen, mit der die gestellte Aufgabe in Echtzeit gelöst werden kann. Außerdem muß die Verteilung der Tasks auf die ausgewählten Verarbeitungseinheiten durchgeführt werden, wobei hier von einer statischen Allokation ausgegangen wird. Statische Allokation bedeutet, daß Tasks während ihrer ganzen Lebenszeit einer Verarbeitungseinheit zugeordnet sind und nicht auf einen anderen Rechner migrieren.

Konkret werden durch das automatisierte Entwurfsverfahren die folgenden Aufgaben

bearbeitet:

- Festlegung der nötigen Rechenelemente.
  Ausreichend leistungsfähige Rechnerknoten in der notwendigen Anzahl sollen automatisch ausgewählt werden.

- Zuordnung der Tasks zu den Rechnerknoten.
  Gegebenenfalls müssen hierbei Restriktionen (z. B. Fehlertoleranz, Zugriff auf gemeinsame Daten) berücksichtigt werden. Die Taskallokation beeinflußt wesentlich die nötige Rechnerzahl und –leistung.

- Parametrisierung des Kommunikationssystems.
  Länge und Periode der Zeitscheiben jedes Rechners für den Zugriff auf das Kommunikationsmedium beim TDMA–Verfahren werden bestimmt.

- Generierung von erforderlichen Zwischendeadlines.
  Zwischendeadlines, die bei der Kommunikation von Tasks über Rechnergrenzen hinweg nötig werden, müssen bei Bedarf automatisch festgesetzt werden.

3.2 Formale Beschreibung der Hardware–Komponenten

Um eine kostengünstige Hardwarestruktur des verteilten Echtzeitsystems generieren zu können, müssen dem Entwurfssystem die zur Verfügung stehenden Hardwarekomponenten (Typen) bekannt sein.

Prinzipiell werden zwei Komponentenarten unterschieden:

1. Verarbeitungseinheiten (Rechenelemente)
2. Kommunikationskomponenten.

Für jedes Rechenelement, auf das Tasks mit rechnerübergreifender Kommunikation alloziert werden, wird automatisch ein Kommunikationselement eingeplant. Damit ergibt sich schließlich ein Rechnersystem wie es beispielhaft in Abbildung 3.1 dargestellt ist. Gezeigt ist ein Rechensystem bestehend aus fünf Rechnern, wobei die Rechner A, B und C miteinander kommunizieren.

Um die einzelnen Hardwarekomponenten voneinander zu unterscheiden und im Gesamtsystem später bewerten zu können, ist es notwendig, diese zu attributieren. Die Verarbeitungseinheiten besitzen die folgenden Attribute:

Verarbeitungsleistung: Als ein Teilschritt des Entwurfsverfahrens wird der Nachweis geführt, daß die gestellte Aufgabe mit einer angenommenen Hardwarestruktur in

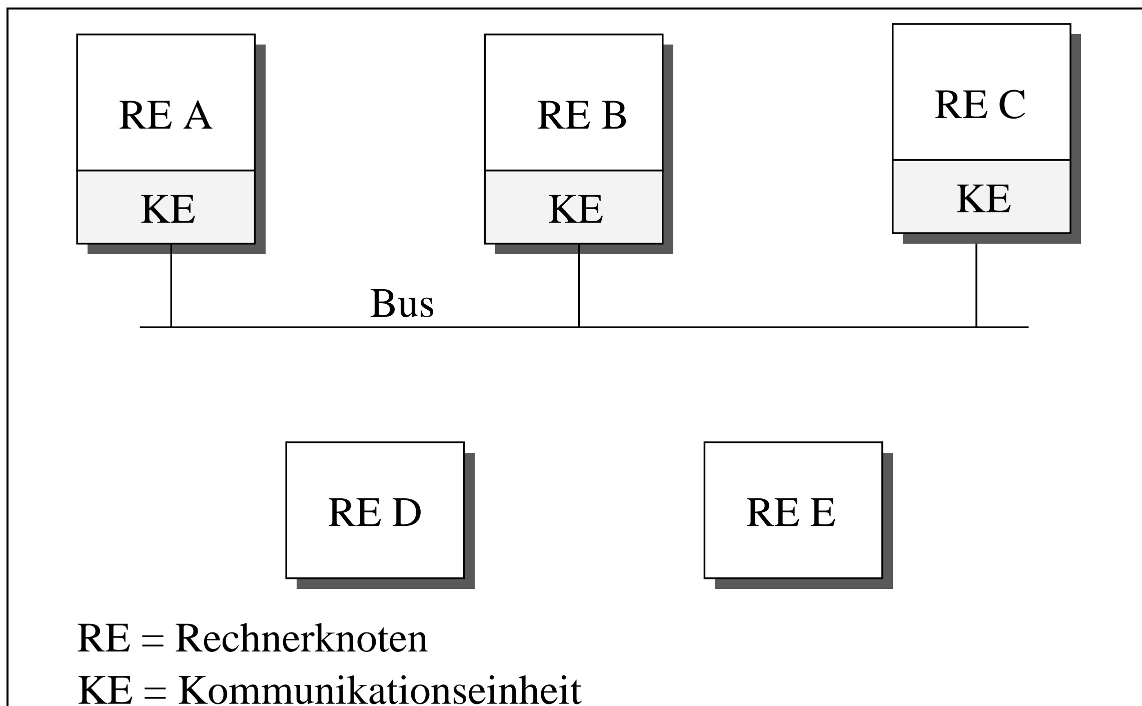


Abbildung 3.1: Beispielhaftes Rechnersystem

Echtzeit bearbeitet werden kann (Echtzeitnachweis). Dazu ist es aber erforderlich, die Verarbeitungszeit t_V (Rechenzeit) einer Task auf der Verarbeitungseinheit (Prozessor) zu kennen. Da es zu aufwendig wäre, die Verarbeitungszeiten für jeden Prozessortyp anzugeben, wird die Rechenzeit t_V für einen „Normrechner“ angegeben und die Verarbeitungsleistung einer CPU wiederum als Faktor relativ zu ebenfalls diesem Normrechner spezifiziert. Aus diesen Angaben kann dann während des automatisierten Systementwurfs die reale Verarbeitungszeit berechnet (skaliert) werden.

Vereinfachend wird hier zunächst von einer konstanten Verarbeitungsleistung ausgegangen. Das führt zwar zu Ungenauigkeiten, die aber durch die ohnehin einzukalkulierende Sicherheitsreserve toleriert werden können. Ungenauigkeiten ergeben sich, wenn in der Realität die Verarbeitungsleistung eines Prozessors abhängig vom zu bearbeitenden Rechenprozeß (Integeroperationen, Floating–Point Operationen) und der Vorgeschichte (Cache) schwankt. Insbesondere führt dies auch dazu, daß die Rechenzeiten nicht für alle Tasks gleichmäßig skalieren, wenn statt des „Normrechners“ auf einen anderen Rechnertyp gewechselt wird.

Hardwarekosten: Kosten werden als einheitenlose Absolutwerte angegeben. Je nach Aufgabenbereich umfassen diese Kosten auch „Fixkosten“, die für alle Rechnertypen gleich sind (z. B. Gehäuse, Stromversorgung etc. bei Komplettrechnern), oder nur direkt von der Verarbeitungseinheit abhängige Kosten (z. B. bei Prozes-

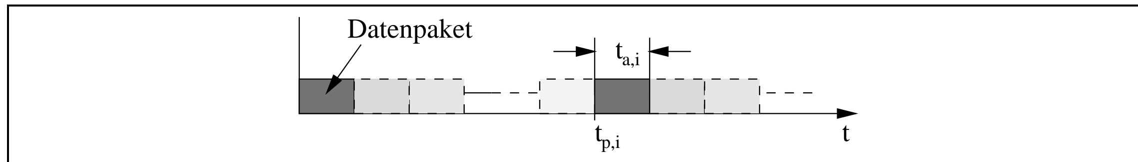


Abbildung 3.2: Attribute des TDMA–Kommunikationssystems

sorboards, die zu einem Gesamtsystem zusammengefaßt werden).

spezifische Eigenschaften: Manche Tasks setzen spezifische Eigenschaften der Verarbeitungseinheiten voraus. Zu diesen spezifischen Eigenschaften gehören beispielsweise besondere Hardwarekomponenten (z. B. spezialisierte Koprozessoren oder besondere Peripheriekontroller).

Die Attribute der Kommunikationskomponenten sind:

Übertragungsleistung: Ähnlich wie die Rechenleistung wird auch die Übertragungsleistung des Kommunikationssystems spezifiziert. Beispielhaft wird von einem Kommunikationssystem ausgegangen, welches seine Daten über ein TDMA Verfahren (Time Division Multiple Access) austauscht. Bei diesem Verfahren bekommt jeder Kommunikationsteilnehmer einen Zeitschlitz fester Länge und mit konstanter Periode zugeordnet.

Kosten: Auch die Kosten für unterschiedliche Typen von Kommunikationskomponenten müssen als einheitenlose Absolutwerte angegeben werden. Aus Gründen der Kompatibilität der Kommunikationskomponenten wird vereinfachend davon ausgegangen, daß bei jedem Rechnerelement mit externer Kommunikation der gleiche Kommunikationskomponententyp verwendet wird.

Zykluszeit: Mit konstanter Periode bekommt das Kommunikationssystem des Teilnehmers einen Zeitschlitz konstanter Dauer zugeteilt. Die Zeit zwischen zwei derartigen Zeitschlitzen wird als Zyklus– oder Periodenzeit $t_{p,i}$ angegeben (siehe Abbildung 3.2).

Zugriffszeit: Die Dauer des Zeitschlitzes $t_{a,i}$ (access time) bestimmt den Anteil des Rechners am Kommunikationssystem.

3.3 Formale Beschreibung der Software–Strukturen

Die Beschreibung der Software–Struktur erfolgt auf Basis von **Taskgraphen** [Gre93], die Beschreibung des dynamischen Softwareverhaltens wird durch Ereignisströme modelliert. Ein Rechenabschnitt ohne Betriebssystemaufruf wird dabei als ein Taskknoten bzw. eine Task dargestellt. Es wird davon ausgegangen, daß Tasks bei der Initialisierung

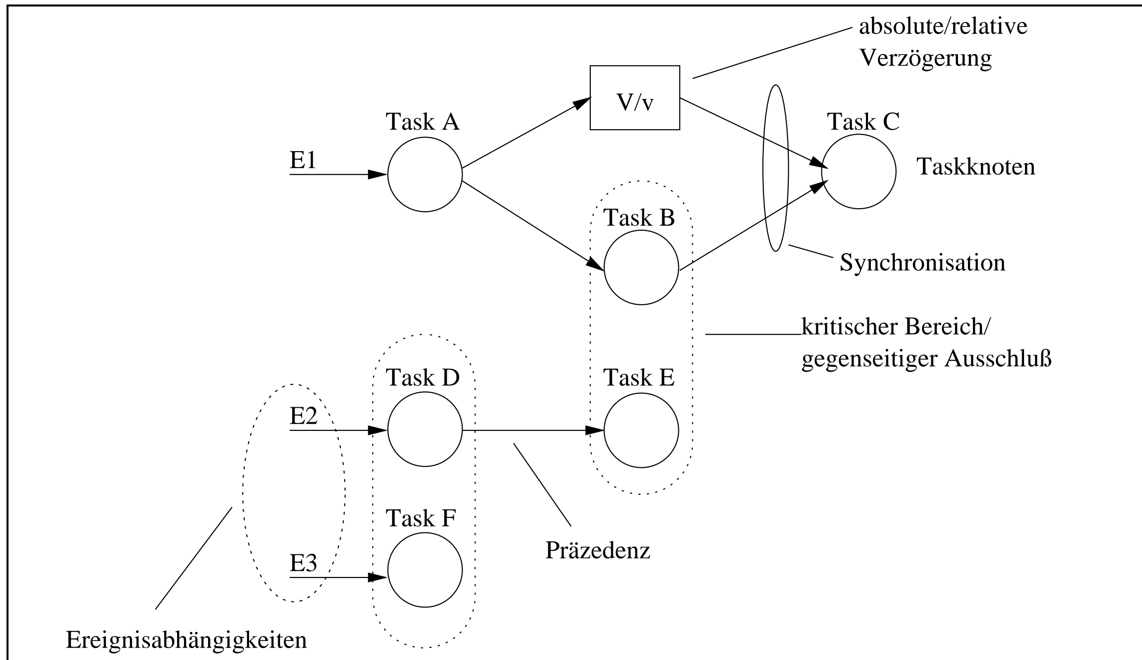


Abbildung 3.4: Beispiel eines Taskgraphen [Gre93]

einmal statisch erzeugt werden und während der gesamten Systemlaufzeit vorhanden sind.

Für jeden Taskknoten muß die Angabe der Verarbeitungszeit t_v und der Deadline t_{Rzul} erfolgen (Abbildung 3.3). Mit dem Eintreffen einer Anforderung E_i wird die Task rechenbereit (lauffähig). Spätestens nach Ablauf der Zeit t_{Rzul} muß die Task die Verarbeitungszeit t_v abgeleistet haben.

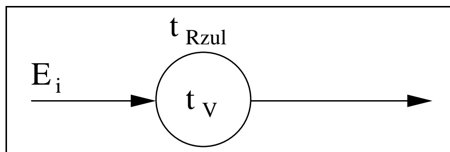


Abbildung 3.3: Taskparameter nach [Gre93]

Im Taskgraphen lassen sich des weiteren Abhängigkeiten zwischen Ereignissen (Alternative) und auch zwischen Tasks (Präzedenzen, gegenseitiger Ausschluß) spezifizieren. Ebenso können Verzögerungen (absolut und relativ), beispielsweise ausgelöst durch die Zeitverwaltung oder durch Peripherie–/Kommunikations–Rückmeldungen, modelliert werden.

Kommunikation zwischen Tasks wird durch Angabe der zu übertragenden Bytes modelliert. Die nach [Gre93] nötigen Zwischendeadlines bei rechnerübergreifender Kommunikation werden bei Bedarf automatisch bestimmt.

Die formale Spezifikation der Taskstruktur wurde erweitert, so daß Randbedingungen, wie beispielsweise die Anforderung, daß zwei oder mehr Tasks nur gemeinsam auf ei-

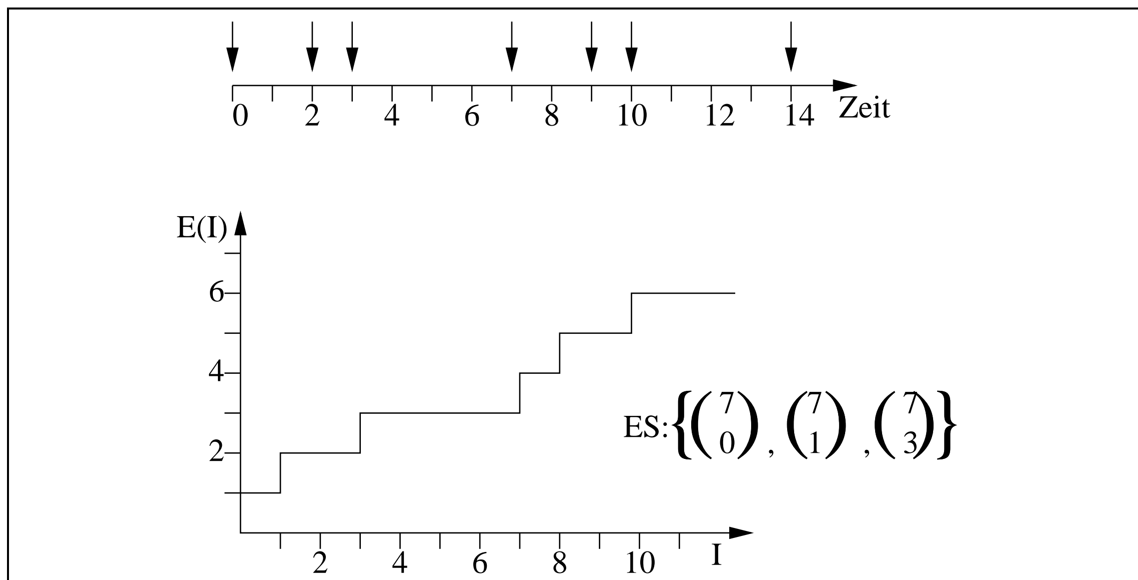


Abbildung 3.5: Beispiel für einen Ereignisstrom nach [Gre93]

nem Rechnerknoten ablauffähig sind, modelliert werden können. Eine weitere modellierbare Anforderung betrifft die Fehlertoleranz. So läßt sich formal spezifizieren, daß zwei oder mehr redundante Tasks *nicht* gemeinsam auf einem Rechnerknoten alloziert werden dürfen.

Für den später notwendigen Echtzeitnachweis ist das dynamische Verhalten des Systems wichtig. Dazu werden die möglichen Anforderungen an das System formal in Form von **Ereignisströmen** [Gre93] beschrieben. Ereignisströme geben die maximal mögliche Anzahl von Ereignissen in Abhängigkeit des Zeitintervalls an. Ein Ereignisstrom wird in Form von Ereignistupeln beschrieben, wobei jedes Tupel aus dem Zyklus z_i und dem Intervall a_i besteht. Die Position innerhalb des Ereignisstroms kennzeichnet wiederum die maximale Anzahl der im Intervall a_i des Zyklus z_i zu erwartenden Ereignisse. In Abbildung 3.5 treten beispielsweise die Ereignisse zu den Zeiten 0, 2 und 3 auf und wiederholen sich mit Zyklus 7. Das erste Tupel hat immer den Intervallwert 0, welcher angibt, daß innerhalb des Zyklus z_i generell mit mindestens einem Ereignis zu rechnen ist. Das zweite Tupel mit dem Intervallwert 1 gibt an, daß in einem beliebigen Intervall von einer Zeiteinheit mit zwei Ereignissen gerechnet werden muß. Das dritte Tupel schließlich besagt, daß innerhalb eines Intervalls von drei Zeiteinheiten maximal drei Ereignisse zu erwarten sind.

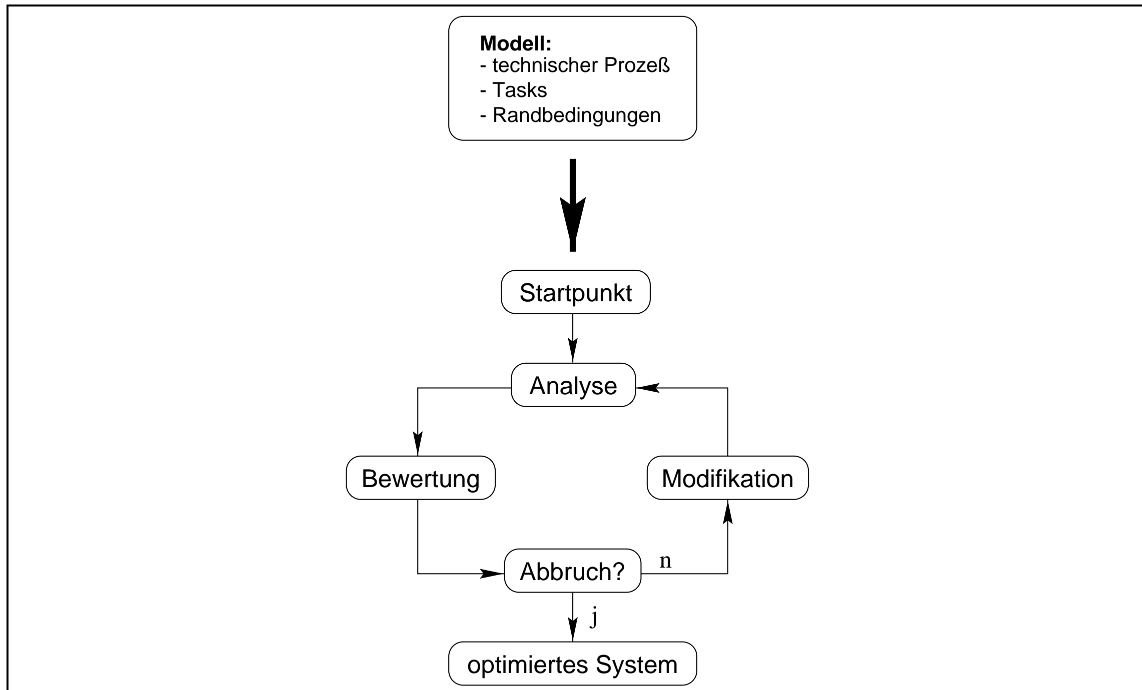


Abbildung 3.6: Optimierungsschleife

3.4 Partitionierung

Bei der Partitionierung werden Aufgaben auf Ressourcen verteilt. Ressourcen sind hierbei die einzelnen Rechnerknoten, die Aufgaben sind durch die Tasks bzw. Taskknoten im Taskgraphen gekennzeichnet.

Das prinzipielle Vorgehen bei diesem Schritt ist, daß das Entwurfssystem eine Rechner–Hardware–Konfiguration vorschlägt, eine Allokation der Tasks auf die vorhandenen Rechnerknoten durchführt und schließlich analysiert, ob alle Realzeitbedingungen eingehalten werden. Können die Realzeitbedingungen eingehalten werden, ist eine gültige Rechnerkonfiguration gefunden worden, die anhand verschiedener Kriterien — im wesentlichen jedoch der Kosten — bewertet wird. Sind die Realzeitbedingungen jedoch nicht erfüllt, so wird so lange eine erneute Taskzuteilung ausprobiert, bis entweder eine gültige Zuteilung gefunden wurde oder eine weitere Überprüfung sinnlos erscheint. Danach wird eine andere Rechner–Hardware–Konfiguration vorgeschlagen und überprüft.

Abbildung 3.6 faßt den prinzipiellen Ablauf der Systemoptimierung zusammen: Eine initiale Konfiguration wird hinsichtlich der Einhaltung der Realzeitbedingungen analysiert. Das Ergebnis der Realzeitanalyse sowie die Überprüfung weiterer Restriktionen (wie z. B. Einschränkungen bei der Taskallokation) werden in einer kostenorientierten Bewertung zu einem Gütewert zusammengefaßt.

Ist das Ergebnis „hinreichend gut“, wird die Optimierungsschleife verlassen; ansonsten wird in weiteren Iterationen versucht, durch Parameteränderungen zu einem besseren Ergebnis zu gelangen.

3.4.1 Automatisierung mit stochastischen Verfahren

Wegen der hohen Komplexität des Problems und der vielen Suboptima ist eine rein zielgerichtete Automatisierung der Optimierungsaufgabe nicht möglich; es bieten sich die stochastischen Optimierungsverfahren an. In diesem Abschnitt soll nur ein grober Überblick über die Bearbeitung der Optimierungsschleife aus Abbildung 3.6 gegeben werden.

Durch die Optimierung sind im wesentlichen zwei Parametergruppen zu bestimmen:

1. Die Parameter zur Systemkonfiguration (Rechnerknoten und Kommunikation) und
2. die Parameter zur Taskallokation.

Zunächst wird angenommen, daß rein quantitativ ebensoviele Rechner zur Verfügung stehen, wie Tasks auf dem System existieren. Das Optimierungsverfahren gibt eine (z. B. anfangs zufällige) Verteilung der zu bearbeitenden Tasks auf die zur Verfügung stehenden Rechnerknoten vor, wobei mehrere Tasks auch einem Rechner zugeteilt werden können (die Tasks bilden also den abhängigen Parameter und die Lokalisation der Tasks auf Rechner variiert). Aufgrund dieser Verteilung von Taskgruppen auf die Rechnerknoten werden die daraus resultierenden Realzeitanforderungen berechnet, und es wird eine kostengünstige Hardware ausgesucht. Aufgrund der erfolgten Hardwareauswahl kann eine Bewertung des Systems vorgenommen werden.

Bei allen stochastischen Optimierungsverfahren ist im Prinzip ein beliebiger, z. B. auch zufällig erzeugter Startpunkt möglich. Allerdings wird abhängig vom gewählten Optimierungsverfahren durch die Wahl eines günstigen Startpunktes eine schnellere Optimierung erreicht (siehe auch Kapitel 4.1.1).

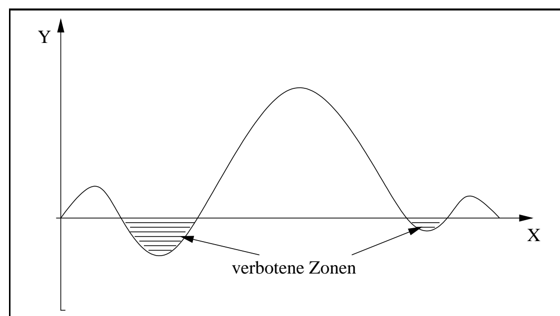


Abbildung 3.7: Beispiel für verbotene Zonen

Parameteränderungen werden jeweils zufällig und in *kleinen* Schritten durchgeführt; die genauen Änderungsmöglichkeiten werden in Kapitel 4.1.3 beschrieben. Auch Verschlechterungen werden dabei — allerdings mit allmählich abnehmender und von dem Betrag der Verschlechterung abhängender Wahrscheinlichkeit — akzeptiert, um ein Festlaufen in lokalen Optima zu vermeiden.

Weiterhin werden auch ungültige Konstellationen, bei denen einzelne Restriktionen verletzt sind, nicht unbedingt von vornherein ausgeschlossen, sondern mit einer entsprechend schlechten Bewertung versehen. Dadurch können auch „verbotene Zonen“ auf dem Weg zum globalen Optimum durchquert werden (Abbildung 3.7).

Da das globale Optimum der Konfiguration in der Regel nicht bekannt ist, wird beim Abbruchkriterium „Die Lösung ist hinreichend gut“ mit „Es wurde über längere Zeit keine Verbesserung erreicht“ gleichgesetzt, wobei die genaue Definition von „längere Zeit“ wiederum spezifisch für die einzelnen Optimierungsverfahren ist.

3.4.2 Echtzeitnachweis

Für den Echtzeitnachweis wird das Analyseverfahren aus [Gre93] verwendet. Dafür wird das Vorliegen eines geeigneten Modells des technischen Prozesses sowie der steuernden Tasks vorausgesetzt (siehe Kapitel 3.1). Mit diesem Verfahren ist die Analyse eines Systems mit folgenden Eigenschaften möglich:

- lose gekoppeltes Mehrrechnersystem mit Rechnerknoten unterschiedlicher Leistungsfähigkeit
- Bussystem mit TDMA–Zugriff
- Taskpräzedenzen
- Asynchrone Kommunikation von Tasks über einen Mailbox–Mechanismus (`send`, `receive`)
- Taskzuteilung nach frühesten Antwortzeiten (Earliest Deadline First)
- Taskanregung durch aperiodische und periodische Ereignisse des technischen Prozesses
- Berücksichtigung ununterbrechbarer Bereiche des Betriebssystems und der Anwendertasks
- Berücksichtigung von gegenseitigem Ausschluß (nicht rechnerübergreifend, d. h. nur lokale Semaphore)
- Berücksichtigung von Prioritätsinversion an Semaphoren
- Interrupt–Service–Routinen mit unterschiedlichen Hardware–Prioritäten
- Block–DMA und Cycle–Stealing–DMA
- Feste Zuordnung der Tasks auf die Rechnerknoten (keine dynamische Task–Verschiebung)

Daneben sind — wie bereits beschrieben — weitere Randbedingungen für die Taskallokation einstellbar, die für das Analyseverfahren nicht relevant sind:

- Tasks dürfen nicht auf denselben Rechner alloziert werden, z. B. Replikate einer fehlertoleranten Task
- Tasks müssen auf denselben Rechner alloziert werden, z. B. wenn ein Zugriff auf gemeinsame globale Variablen erforderlich ist
- Tasks müssen auf einen bestimmten Rechnertyp alloziert werden, z. B. wenn spezielle Hardware–Komponenten benötigt werden

Beim Echtzeitnachweis wird die Ereignisfunktion (Abbildung 3.5 auf Seite 15) mit der Rechenzeit t_V für jede Ausführung einer Task multipliziert und um die Deadline $t_{R\mathrm{zul}}$ nach rechts verschoben. Das ergibt die Rechenzeitanforderungsfunktion RZAF (siehe z. B. Abbildung 5.2 auf Seite 78). Die Rechenzeitanforderungsfunktion für das Gesamtsystem erhält man durch Addition der einzelnen Rechenzeitanforderungsfunktionen [Fär99]. Im Nachweisverfahren wird jetzt verifiziert, daß zu jedem Zeitpunkt der Funktionswert dieser Funktion kleiner ist als die zum Zeitpunkt zur Verfügung stehende Rechenleistung (grafisch betrachtet darf die Rechenzeitanforderungsfunktion nicht die Winkelhalbierende schneiden).

3.5 Ergebnis des Entwurfsverfahrens

Innerhalb des Entwurfsverfahrens werden die (Hardware–) Kosten zunächst aufsummiert. Dies führt zu einer diskreten Bewertungsfunktion, die jedoch für den Systemvergleich bei kleinen Änderungen zu grob ist. Daher fließen auch andere Komponenten in die Bewertung mit herein, wie beispielsweise die Auslastung des Rechnersystems. Die Einbeziehung derartiger Qualitätsmerkmale in die Bewertungsfunktion führt zu einer stetigen Kostenfunktion, die für einen Systemvergleich herangezogen werden kann. Aufgrund dieser Bewertungsfunktion wird dann eine kostenoptimale Systemkonfiguration ausgegeben.

3.6 Untersuchte Modelle

Für die Experimente, die den Untersuchungen in den folgenden Abschnitten zugrunde liegen, wurden zwei Taskmodelle unterschiedlicher Komplexität verwendet. Die beiden Taskmodelle decken von ihrer Struktur her die wesentlichen Aspekte des Entwurfs eines verteilten Mehrrechnersystems ab. Sicher kann damit nicht auf die generelle Eignung der

untersuchten Algorithmen für sämtliche möglichen Konstellationen geschlossen werden; tendentielle Aussagen lassen sich aber auch aus diesen Beispielen ableiten.

Beispiel–Modell „E“: Ein sehr einfaches (E) System bestehend aus 100 identischen Tasks. Die Tasks sind unabhängig, also kommunikationslos; das Kommunikationssystem ist in diesem Modell ohne Bedeutung. Die Tasks werden durch identische, periodische Ereignisströme angeregt; die Deadline ist gleich der Periode. Dadurch entstehen keine Restkapazitäten im Sinne von Kapitel 5.1.4 (Seite 77), und das globale Optimum ist einfach zu berechnen. Eine Beschreibung dieses Modells befindet sich in Anhang B.2.

Beispiel–Modell „R“: Dieses Modell enthält Restriktionen (R) und ein gemischtes Taskprofil mit unterschiedlichen Tasklaufzeiten und Ereignisströmen. Abgesehen von der Kommunikation ist das Taskmodell an das realitätsnahe Anwendungsbeispiel in [Gre93] angelehnt. Die Parameter und eine detaillierte Beschreibung sind in Anhang B.3 wiedergegeben.

Für die Systemmodelle wurde eine statische Auswahl von Rechnerkomponenten vorgegeben (Anhang B.1).

Aus Vereinfachungsgründen wurden bei allen Experimenten bezüglich der Analyse folgende Einschränkungen gegenüber [Gre93] getroffen:

- Rechenzeiten für Interrupt–Service–Routinen werden vernachlässigt.
- Zeitunschärfe, DMA und ununterbrechbare Bereiche werden in der Realzeit–Analyse nicht berücksichtigt.
- Es sind keine Ereignisabhängigkeiten vorhanden.
- Die Experimente umfassen keine Kommunikation, die nötigen Behandlungsmechanismen sind aber in den folgenden Kapiteln dargestellt.

4 Anwendung stochastischer Optimierungsverfahren

In diesem Kapitel wird die Anwendung der stochastischen Optimierungsverfahren auf die gegebene Problemstellung dargestellt. Zunächst werden die für alle Verfahren gültigen problemspezifischen Funktionen behandelt. Anschließend werden die einzelnen Algorithmen beschrieben und die Ergebnisse der Anwendung auf die Beispielmodelle aus Kapitel 3.6 vorgestellt.

4.1 Problemspezifische Funktionen

Alle im folgenden untersuchten stochastischen Optimierungsverfahren benötigen an das Problem angepaßte Funktionen, die in den folgenden Abschnitten entwickelt werden: Eine Funktion zur Ermittlung eines *Startpunktes*, eine *Bewertungsfunktion* (oft auch als *Zielfunktion* oder *Gütefunktion* bezeichnet), die zur Bewertung einer untersuchten Konfiguration dient, und eine *Änderungsfunktion* zur Ermittlung eines „benachbarten“ Punktes im Lösungsraum.

4.1.1 Startpunktwahl

Als Startpunkt der Optimierung kann eine zufällig ausgewählte Konfiguration, eine vom Anwender vorgegebene Konfiguration oder eine durch eine Heuristik bestimmte Konfiguration verwendet werden.

Im allgemeinen Fall stellt die Startwertsuche schon ein Problem für sich selbst dar, um eine erste „gültige“ Konfiguration zu finden, die alle Randbedingungen einhält. Werden dagegen Restriktionsverletzungen in der Bewertungsfunktion erfaßt und auf entsprechend erhöhte Kosten abgebildet, wie es hier der Fall ist, kann prinzipiell jede beliebige, „ungültige“ Konfiguration als Startwert verwendet werden. Ein extrem schlechter Startpunkt kann dabei zu einer schlechteren Konvergenz der Optimierungsverfahren führen;

andererseits erschwert ein Startpunkt auf einem lokalen Optimum das Herausfinden des Optimierungsverfahrens aus diesem lokalen Optimum hin zu einem noch besseren Ergebnis.

Um eine bessere Vergleichbarkeit der Optimierungsverfahren zu ermöglichen, wurde in den Untersuchungen in diesem Kapitel jeweils von einem vorgegebenen Startpunkt ausgegangen.

4.1.2 Bewertungsfunktion

Da das Optimierungsziel eine Systemrealisierung mit *minimalen Kosten* unter Einhaltung aller Randbedingungen darstellt, muß die Bewertungsfunktion *kostenorientiert* arbeiten.

Für die Optimierungsverfahren ist die Bildung *eines* Kostenwertes K_{ges} nötig. Dieser setzt sich zusammen aus Kostenwerten für die Rechner K_{R} und für die Kommunikationselemente K_{K}:

$$K_{\text{ges}} = K_{\text{R}} + K_{\text{K}}$$

Die weitere Aufschlüsselung von K_{R} und K_{K} erfolgt in den anschließenden Abschnitten.

4.1.2.1 Anpassung der Rechenleistung

Zur Reduktion der Komplexität des Optimierungsproblems wird die Rechenleistung bzw. die zu verwendenden Rechnertypen nicht als zu optimierender Systemparameter betrachtet, sondern aus der Realzeitanalyse der momentanen Konfiguration ermittelt (vgl. Kapitel 3.4.2). Die Analyse liefert für jeden Rechner r den größten Quotienten aus Rechenzeitanforderung t_v und Intervall $t_{R\text{zul}}$ in der Rechenzeitanforderungsfunktion. Dieser Quotient $t_v/t_{R\text{zul}}$ ist eine Abschätzung des relativen Leistungsfaktors l_r, um den die Leistungsfähigkeit des Rechners r verkleinert oder vergrößert werden muß, damit die Realzeitbedingungen *gerade noch* eingehalten werden; bei $t_v/t_{R\text{zul}} = 1$ wird die Winkelhalbierende in der Rechenzeitanforderungsfunktion gerade berührt.

Eine genaue Berechnung der nötigen Leistungsfaktoren wäre mit dem in [Gre93] vorgeschlagenen iterativen Verfahren möglich, würde jedoch zu einer deutlichen Erhöhung der Analysezeiten führen. An dieser Stelle genügt aber bereits die Abschätzung. Da in jedem Optimierungsschritt die Analyse erneut aufgerufen wird und damit auch die Leistungsfaktoren neu bestimmt werden, ist eine iterative Annäherung ohnehin implizit gegeben.

Bei $l_r < 1$ wird geprüft, ob für Rechenelement r ein leistungsschwächerer, preiswerterer Typ vorhanden ist. Falls ja, wird dieser für die weitere Bewertung und den nächsten Optimierungsschritt verwendet.

Im Falle $l_r > 1$ muß Rechenelement r auf jeden Fall durch einen leistungsfähigeren Typ ersetzt werden. Ist kein ausreichend leistungsfähiger Typ vorhanden, wird temporär ein *virtueller Rechner* mit der genau erforderlichen Leistung eingeführt. Da ein solches Rechenelement ja nicht verfügbar ist, ist diese Parameterbelegung ungültig und wird durch eine entsprechend schlechte Bewertung gekennzeichnet: Sie wird mit Kosten belegt, die wesentlich über den Kosten des leistungsfähigsten realen Rechenelmentes liegen (vgl. Kostenansatz auf Seite 27).

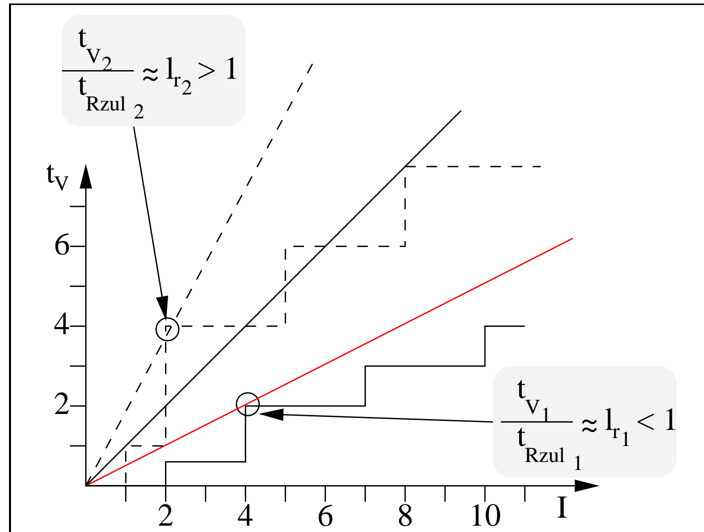


Abbildung 4.1: Anpassung d. Rechenleistung

Die Suche nach einem Rechnertyp passender Leistung gilt nur unter der Annahme, daß sich die Taskrechenzeiten umgekehrt proportional zum Leistungswert des Rechnertyps verhalten; dies kann aber bei Realzeitsystemen in der Regel angenommen werden. Ist dies nicht der Fall, d. h. skalieren die Taskrechenzeiten verschiedener Tasks unterschiedlich beim Wechsel des Rechnertyps, müssen die Taskrechenzeiten im Systemmodell für die unterschiedlichen Rechnertypen angegeben sein (vgl. Kapitel 3.2 auf Seite 11). Der gerade verwendete Rechnertyp muß dann nacheinander durch alle verfügbaren Rechnertypen ersetzt werden, und die Realzeitanalyse muß dabei jedesmal erneut durchgeführt und l_r von neuem bestimmt werden. Schließlich wird der Rechnertyp mit größtem $l_r < 1$ als neuer Typ verwendet.

Bei der Ersetzung des Rechnertyps bleiben ggf. vorhandene Anforderungen von Tasks nach bestimmten Hardware–Voraussetzungen (z. B. Peripherie, Speicher, Koprozessoren) unberücksichtigt; eine dadurch neu entstehende Restriktionsverletzung spiegelt sich später in dem ermittelten Kostenwert wieder (s. Seite 29).

Eine zu den Rechenelementen analoge Anpassung der Kommunikationselemente ist bei dem angenommenen Kommunikationsverfahren nicht sinnvoll, da hier zum einen alle Kommunikationselemente die gleiche (physikalische) Bandbreite haben müssen, und zum anderen die Netto–Datenrate eines Kommunikationselementes wesentlich durch sei-

ne TDMA–Parameter (Zeitanteil und Gesamtzyklus) gegeben sind. Die Lastfaktoren der Kommunikationselemente werden daher erst für die Berechnung der Kommunikationskosten herangezogen (Kapitel 4.1.2.3). Durch die stark wachsende Kommunikationskapazität (z. B. bei modernen parallelen Bussystemen) tritt diese Problematik hier auch zunehmend in den Hintergrund.

4.1.2.2 Rechnerkosten

In die Rechnerkosten K_{R} fließen mehrere Anteile ein, die in den folgenden Abschnitten entwickelt werden.

Die Rechnerkosten sind zunächst natürlich durch die Hardwarekosten K_{HW} für die verwendeten Rechner bestimmt. Da diese aber in relativ groben Schritten diskretisiert sind (vgl. Abbildung 4.2), würde eine kleine Parameteränderung, beispielsweise die Verschiebung einer Task von einem auf einen anderen Rechner, unter Umständen zu keiner Änderung des Kostenwertes führen und damit zu keiner erkennbaren qualitativen Veränderung. Aus diesem Grund wird ein weiterer, rechenlastabhängiger Kostenwert K_{Last} hinzugefügt.

Wie bereits erwähnt wurde, können zeitweilig auch ungültige Konstellationen untersucht werden, bei denen einzelne Allokations–Restriktionen verletzt sind, womit sie natürlich nicht die endgültige Lösung darstellen können. Diese Restriktionsverletzungen werden in dem Term K_{Restrikt} bewertet. Damit ergibt sich für die Rechnerkosten K_{R}:

$$K_{\mathrm{R}} = K_{\mathrm{HW}} + K_{\mathrm{Last}} + K_{\mathrm{Restrikt}}$$

Die einzelnen Kostenelemente dieser Formel werden in den folgenden Abschnitten aufgeschlüsselt.

Hardware–Kosten

Wie in Kapitel 4.1.2.1 erläutert wurde, wird nach der Realzeitanalyse für jeden Rechner r in der Tabelle der verfügbaren Rechnertypen ein bezüglich der Leistung gerade ausreichender Typ gesucht; damit sind auch die Hardware–Kosten $K_{\mathrm{r,real}}$ für diesen Rechner festgelegt.

Als ein Beispiel für eine solche Tabelle, dargestellt als Kurve der Kosten über die Rechenleistung, zeigt Abbildung 4.2 eine reale Kostenerhebung von PC–Mainboards auf ix86–Basis (Mainboard mit Prozessor, ohne Speicher; Stand Juni 1994). Eine Erhebung

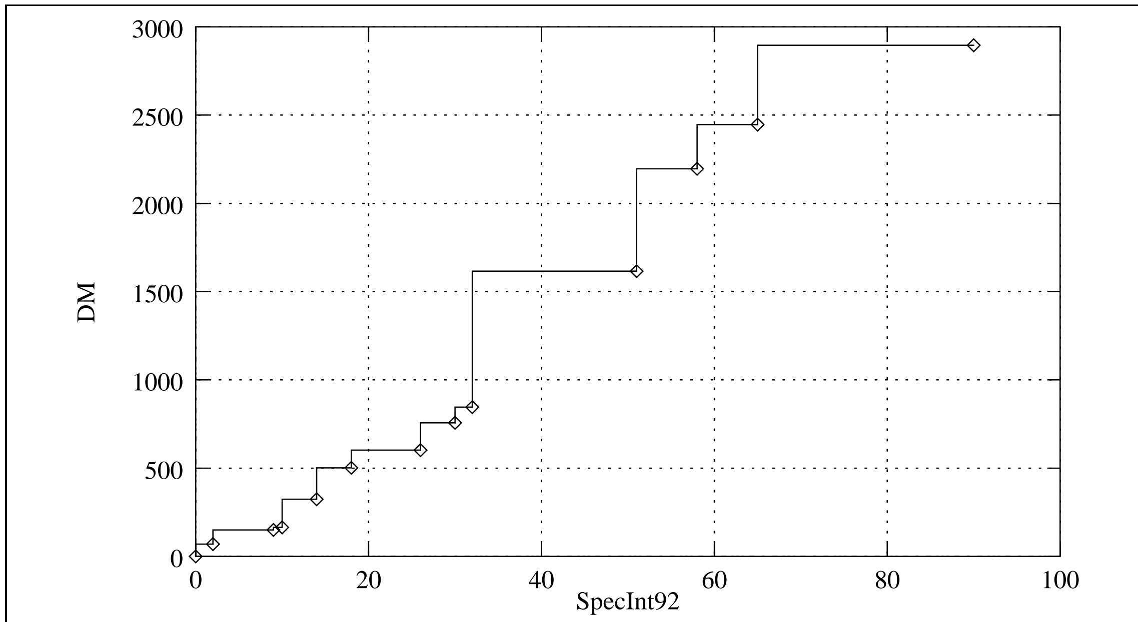


Abbildung 4.2: Preis–/Leistungskurve von 80x86–Boards

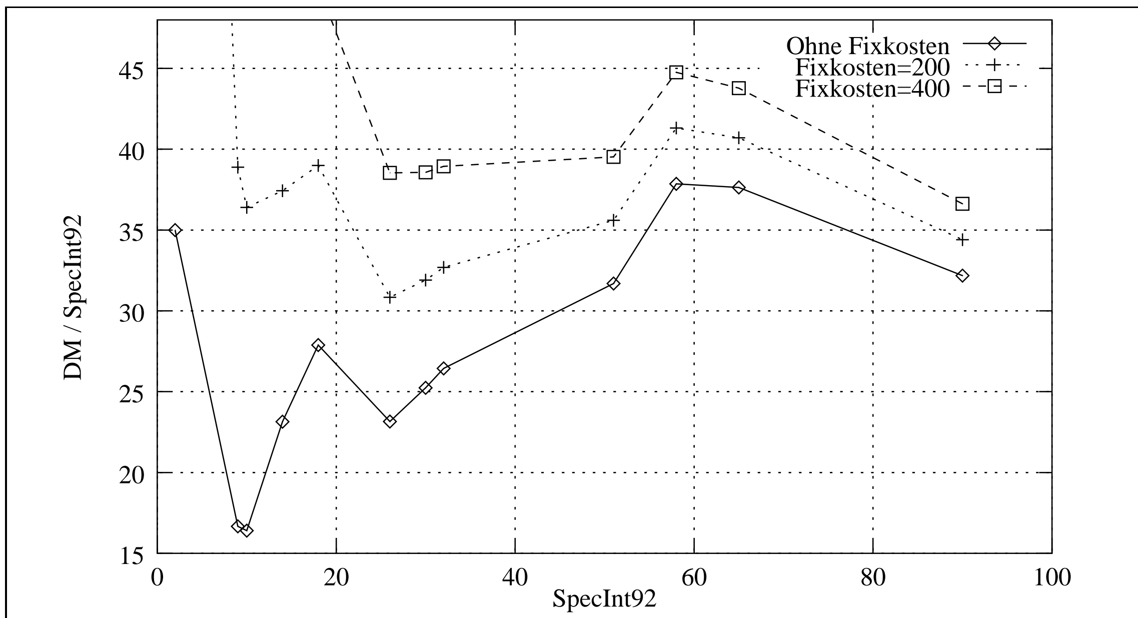


Abbildung 4.3: Preis–/Leistungs–Verhältnis von 80x86–Boards mit unterschiedlichen Fixkosten

neueren Datums ergab eine qualitativ ähnliche Kurve mit quantitativen Änderungen entsprechend dem Preisverfall und der Leistungssteigerung im PC–Bereich.

Zu diesen leistungsabhängigen Kosten kommen natürlich weitere Kosten hinzu (Stromversorgung, Gehäuse, Hauptspeicher, Kommunikationskontroller, Peripherieanbindung, Installation, ...), die je nach Aufgabenstellung stark variieren können (z. B. steckbare Prozessor–Boards im gemeinsamen Gehäuse gegenüber Single–Prozessor–Komplettrechnern; Spritzwasser– oder Explosionsschutz; unterbrechungsfreie Stromversorgung; anwendungsbedingte Peripheriekarten). Durch diese von der benötigten Rechenleistung unabhängigen Fixkosten verschiebt sich der Punkt des besten Preis–/Leistungsverhältnisses zu höheren Leistungswerten, wie Abbildung 4.3 veranschaulicht: Bereits bei 400 DM Fixkosten ist in diesem Beispiel der leistungsstärkste Rechner relativ zur Rechenleistung am preisgünstigsten.

Daraus kann aber nicht die Vereinfachung abgeleitet werden, daß generell der leistungsstärkste Rechnertyp verwendet werden sollte. Denn insbesondere bei den neuesten High–End Prozessoren sind die Preisunterschiede wiederum so hoch, daß diese Schwelle in der Regel nicht erreicht wird. Abbildung 4.4 zeigt dazu einen Vergleich dreier Komplett–PC's (ohne Monitor; Stand Frühjahr 1997) eines einzigen Anbieters, die sich lediglich in der Taktrate des Prozessors unterscheiden: Das günstigste Kosten–/Leistungsverhältnis (der Einfachheit halber nur auf die Prozessor–Taktrate bezogen) weist hier die mittlere Konfiguration auf.

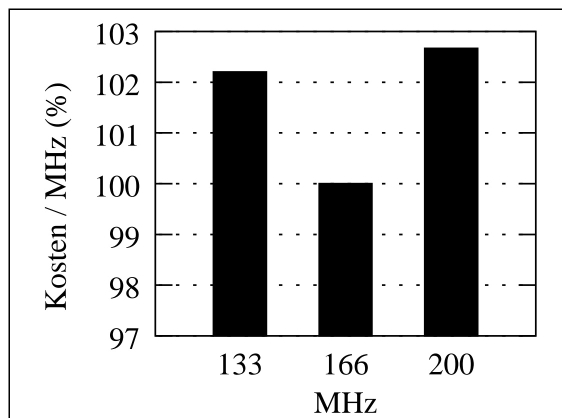


Abbildung 4.4: Preisvergleich von Komplett–PC's

Durch die in jedem Optimierungsschritt erfolgende Anpassung der ausgewählten Rechnertypen an die erforderliche Leistung wird vermieden, daß die Realzeitbedingungen verletzt werden, solange nicht mehr Rechenleistung erforderlich ist, als der stärkste verfügbare Rechner bereitstellt. Für diesen Fall wird ein „virtueller Rechner" ausreichender Leistung bewertet. Die Hardware–Kosten K_{HW} aller Rechner r ergeben sich damit aus der Summe der Kosten für die realen Rechner K_{real} und für die virtuellen Rechner K_{virt}:

$$K_{HW} = \sum_r \begin{Bmatrix} K_{r,real} \\ K_{r,virt} \end{Bmatrix}$$

Virtuelle Rechner

Wenn bei der Realzeitanalyse festgestellt wird, daß eine höhere Rechenleistung benötigt wird, als der leistungsstärkste zur Verfügung stehende Rechner bereitstellen kann, wird für die Bewertung ein „virtueller Rechner“ erzeugt, der genau diese geforderte Rechenleistung zur Verfügung stellt. Die Bewertung dieses Rechners erfolgt in Abhängigkeit der Rechenleistung.

Die untersuchte Konfiguration ist damit ungültig, da so nicht realisierbar. Es muß daher sichergestellt sein, daß die Kosten K_{virt} eines virtuellen Rechners auf jeden Fall höher sind als die Kosten der realen Rechner, die das auf den virtuellen Rechner allozierte Tasksystem zeitgerecht bearbeiten könnten. Daraus ergeben sich zwei Forderungen:

1. Es muß ein Kosten*sprung* nach dem leistungsfähigsten Rechner um *mindestens* die Kosten des leistungsschwächsten Rechners stattfinden.
2. Der weitere Kosten*anstieg* muß abhängig von der erforderlichen Leistung so stark sein, daß das virtuelle System teurer bewertet wird als eine real mögliche Lösung.

Die genaue Definition von K_{virt} hängt damit von der vorliegenden Kostenkurve der realen Rechner ab und müßte jeweils darauf angepaßt werden. In der Praxis ist aber ein kräftiger Kostensprung nach dem leistungsstärksten Rechner (Rechenleistung $P_{\max}$) und eine anschließend mit der geforderten Leistung P schnell steigende Kostenfunktion entsprechend nachfolgender Formel für die Optimierung ausreichend:

$$K_{\text{virt}} = K_{\text{virt}}(P) = a_v * K_{\text{real,max}} * \left(\frac{P}{P_{\max}}\right)^{e_v}$$

Der Parameter a_v (mit $a_v > 1$) bewirkt den Kostensprung, und e_v ($e_v > 1$) überproportional ansteigende Kosten. Als Beispiel ist in Abbildung 4.5 auf der nächsten Seite für Werte größer 90 SpecInt92 die Erweiterung von Abbildung 4.2 für $a_v = e_v = 2$ dargestellt.

Lastabhängige Kosten

Da bei kleinen Konfigurationsänderungen wie z. B. dem Verschieben einer Task von einem auf einen anderen Rechner in vielen Fällen die Anzahl und auch die Typen der verwendeten Rechner unverändert bleiben, kann die Auswirkung einer solchen Änderung nicht mit den Hardware–Kosten K_{HW} bewertet werden. Es ist daher ein weiterer, lastabhängiger Kostenwert K_{Last} nötig, der auch die Erkennung geringer Auslastungsänderungen eines Rechners ermöglicht.

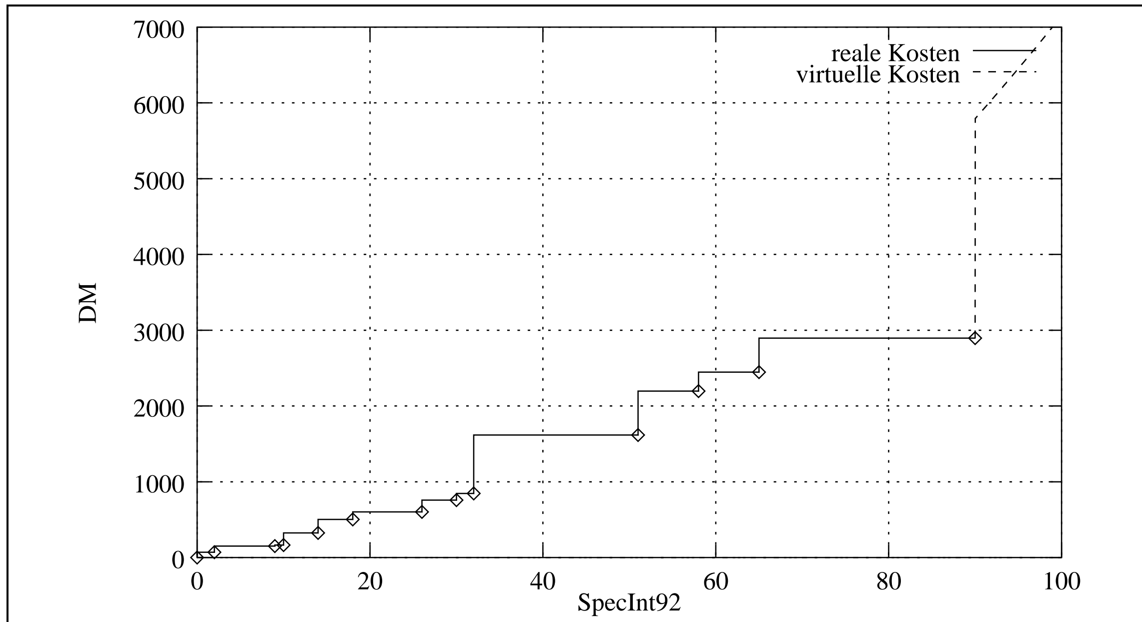


Abbildung 4.5: Kostenkurve, erweitert für virtuelle Rechner

Eine einfache Implementierung ist die Quadratsumme der relativen Auslastungen l_r aller nicht–virtuellen Rechner (die Auslastung virtueller Rechner wird bereits in K_{virt} berücksichtigt):

$$K_{\text{Last}} = a_l * \sum_{r|\text{nicht virtuell}} l_r^2$$

Die relativen Auslastungen l_r sind dabei schon auf die nach der Realzeitanalyse angepaßten Rechnertypen bezogen (siehe Kapitel 4.1.2.1).

Durch die Verwendung der quadratischen Summe wird eine gleichmäßige Auslastung aller Rechner günstiger bewertet, da mit $\sum l_r^2$ gleichzeitig die Varianz der relativen Auslastungen minimiert wird. Der verfügbare Spielraum wird so gleichmäßig über alle Rechner verteilt, was für zukünftige Erweiterungen wünschenswert sein kann. Weiterhin bewirkt dies auch eine erhöhte Toleranz gegen Fehler bei der Laufzeitermittlung und damit eine höhere Sicherheit zur Einhaltung der Realzeitanforderungen.

Der Gewichtungsfaktor a_l muß so eingestellt werden, daß der Term K_{Last} deutlich kleiner als K_{HW} bleibt; da dies jedoch von dem Anwendungsproblem und der verwendeten Kostenkurve abhängt, muß a_l jeweils den spezifischen Anwendungen angepaßt oder mittels einer Heuristik z. B. auf einen Bruchteil der geschätzten Rechner–Hardwarekosten gesetzt werden.

Der anwendungsspezifisch zu bestimmende Parameter a_l kann vermieden werden, wenn K_{Last} als Term zur Interpolation der Hardware–Kosten definiert wird. Da die verwendeten Rechnertypen schon bei der Bestimmung der Hardware–Kosten entsprechend der erforderlichen Leistung angepaßt werden (Kapitel 4.1.2.1 auf Seite 22), kann sich die relative Worst–Case–Last l_r nur in den Grenzen

$$l_{r,\min} = \frac{P_{r^-}}{P_r} \quad < \quad l_r \quad \leq \quad 1$$

bewegen, wobei P_r die Rechenleistung des für Rechner r verwendeten Typs und P_{r^-} die Rechenleistung des nächstschwächeren Typs ist. Für diesen Bereich von l_r soll $K_{\text{Last},r}(l_r)$ als monotone Funktion definiert werden. An der unteren Grenze kann sicherlich $K_{\text{Last},r}(l_{r,\min}) = 0$ gewählt werden; an der Grenze zum nächst stärkeren Rechner dürfen dessen reale Kosten nicht überschritten werden:

$$K_{\text{Last},r}(1) + K_{\text{HW},R} \leq K_{\text{HW},R^+}$$

R ist dabei der Rechnertyp, der für Rechner r verwendet wird, und R^+ der nächst leistungsstärkere Typ. Mit dem Ziel, die Optimierung besonders auf solche Rechner zu konzentrieren, bei denen die mögliche Kostenersparnis, d. h. der Kostensprung zum nächst schwächeren Rechner R^- besonders groß ist, wird $K_{\text{Last},r}(1)$ sinnvollerweise in Abhängigkeit von diesem Kostensprung definiert. Insgesamt ergibt sich damit:

$$K_{\text{Last}} = \sum_{r|\text{nicht virtuell}} K_{\text{Last},r}(l_r)$$

mit

$$K_{\text{Last},r}(l_r) = \left(\frac{l_r - l_{r,\min}}{1 - l_{r,\min}}\right)^2 * \min \left\{ \begin{array}{l} a_b * (K_{\text{real},R} - K_{\text{real},R^-}) \\ a_t * (K_{\text{real},R^+} - K_{\text{real},R}) \end{array} \right\} \quad ; 0 \leq a_b, a_t \leq 1$$

Abbildung 4.6 auf der nächsten Seite zeigt einen Ausschnitt aus Abbildung 4.2 für $a_b = a_t = 0.8$. Mit dem Parameter a_b wird eingestellt, wie stark die Kostendifferenz zum nächstbilligeren Rechnertyp R^- berücksichtigt werden soll, und durch a_t die minimal verbleibende Höhe des Kostensprungs zwischen den Rechnern R und R^+; die Festlegung der beiden Parameter kann relativ unabhängig von der Anwendung erfolgen.

Restriktionsverletzungen

Restriktionen bei der Taskallokation, z. B. das Verbot der Zuteilung redundanter Replikate einer fehlertoleranten Task auf denselben Rechner oder die Mißachtung der Forderung nach einem bestimmten Rechnertyp, können entweder direkt in der Änderungsfunktion (siehe Kapitel 4.1.3 auf Seite 34) berücksichtigt werden, oder aber auch zur

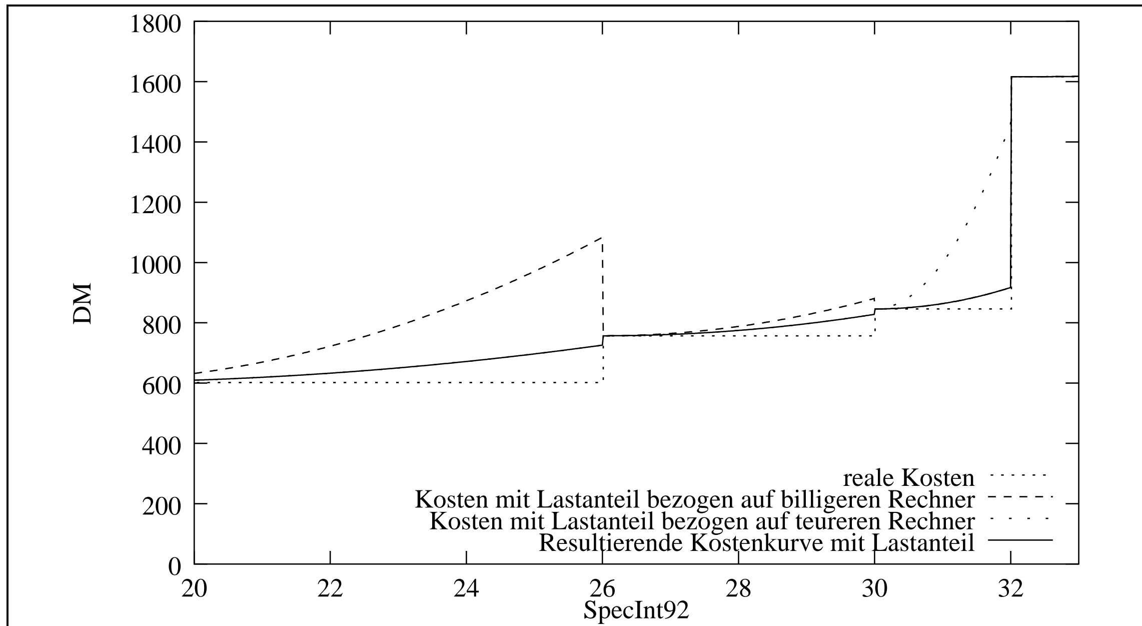


Abbildung 4.6: Kostenkurve mit lastbezogenem Anteil (Ausschnitt)

„Überbrückung verbotener Bereiche" zunächst zugelassen werden, denn es ist möglich, daß bei der Optimierung der „Fangbereich" eines lokalen Optimums nur dadurch verlassen werden kann. Da Restriktionsverletzungen auch bei der Rechnerleistungsanpassung (Kapitel 4.1.2.1) entstehen können, müssen sie auf jeden Fall entsprechend ungünstig bewertet werden.

Die Anzahl N_r der Restriktionsverletzungen geht mit

$$K_{\text{Restrikt}} = a_r * N_r$$

in die Rechnerkosten K_{R} ein. Der Gewichtungsfaktor a_r muß so eingestellt werden, daß der Anteil von K_{Restrikt} an K_{R} einerseits genügend hoch ist, um ungültige Konfigurationen sicher wieder zu verlassen, andererseits aber nicht so hoch ist, daß diese Konfigurationen doch wegen zu schlechter Bewertung von vornherein ausgeschlossen werden.

4.1.2.3 Kommunikationskosten

Die Realzeitanalyse des Nachrichtentransportsystems wird für jedes Kommunikationselement in analoger Weise zu der Analyse der Recheneinheiten durchgeführt, und es wird ebenfalls ein relativer Auslastungswert l_k für den Worst Case bestimmt. Dennoch kann hier nicht analog zur Kostenbestimmung für die Rechner eine Leistungsanpassung vorgenommen werden:

- Es wird vom Systemmodell nur *ein* Bussystem unterstützt, da andernfalls irgendeine Art von Nachrichten–Routing nötig ist, was die Realzeitanalyse wenn nicht unmöglich, so doch erheblich komplexer macht.
- Alle verwendeten Kommunikationselemente müssen zueinander kompatibel sein, d. h. beispielsweise mit gleichem Bustakt arbeiten. Daher könnte eine Leistungsanpassung nicht unabhängig für jedes Kommunikationselement durchgeführt werden, sondern müßte gleichzeitig für alle Einheiten auf den Typ erfolgen, der die maximal erforderliche Kommunikationsleistung zur Verfügung stellen kann. Der ausgewählte Typ müßte auch für alle verwendeten Rechnertypen zur Verfügung stehen.
- Die Nachrichtenübertragungszeiten werden neben der Bandbreite des Busses wesentlich durch die Zugriffsparameter (Länge des Zeitschlitzes zum Senden und TDMA–Zykluszeit) bestimmt.

Aus diesen Gründen, und da in der Praxis bei den Kommunikationselementen ohnehin eine geringere Auswahl als bei den Rechnertypen gegeben ist, wird ein fixer Kommunikationselemente–Typ vorausgesetzt. Modell, Analyse und Optimierung sind jedoch analog erweiterbar. In die Gesamtkosten geht nur die Worst–Case–Auslastung der Kommunikationselemente nach folgender Formel ein:

$$K_{\mathrm{K}} = a_k * \sum_k \begin{cases} l_k^2 & ; l_k \leq 1 \text{ (Leistung ausreichend)} \\ a_v * l_k^{e_v} & ; l_k > 1 \text{ (Überlast)} \end{cases}$$

Hierbei ist l_k der relative Auslastungsfaktor des Kommunikationselementes, der in der gleichen Weise wie der Rechenleistungsfaktor l_r (vgl. Kapitel 4.1.2.1) aus der Realzeitanalyse ermittelt wird. Mit dem Gewichtungsfaktor a_k ($a_k > 0$) wird das Gewicht von K_{K} an K_{ges} eingestellt. a_k muß anwendungsspezifisch bestimmt werden.

Wenn ein Kommunikationselement nicht überlastet ist, geht l_k analog zur Worst–Case–Rechnerlast quadratisch in die Summe ein; im Überlastfall kann wie bei der Kostenfunktion für virtuelle Rechner (K_{virt} auf Seite 27) mit a_v ($a_v > 1$) ein Kostensprung und mit e_v ($e_v > 1$) ein starkes, überproportionales Ansteigen der Kosten mit l_k festgelegt werden.

In Abbildung 4.7 ist K_{K} in Abhängigkeit von l_k für die Werte $k = 1$, $a_k = 1000$, $a_v = 2$ und $e_v = 4$ dargestellt.

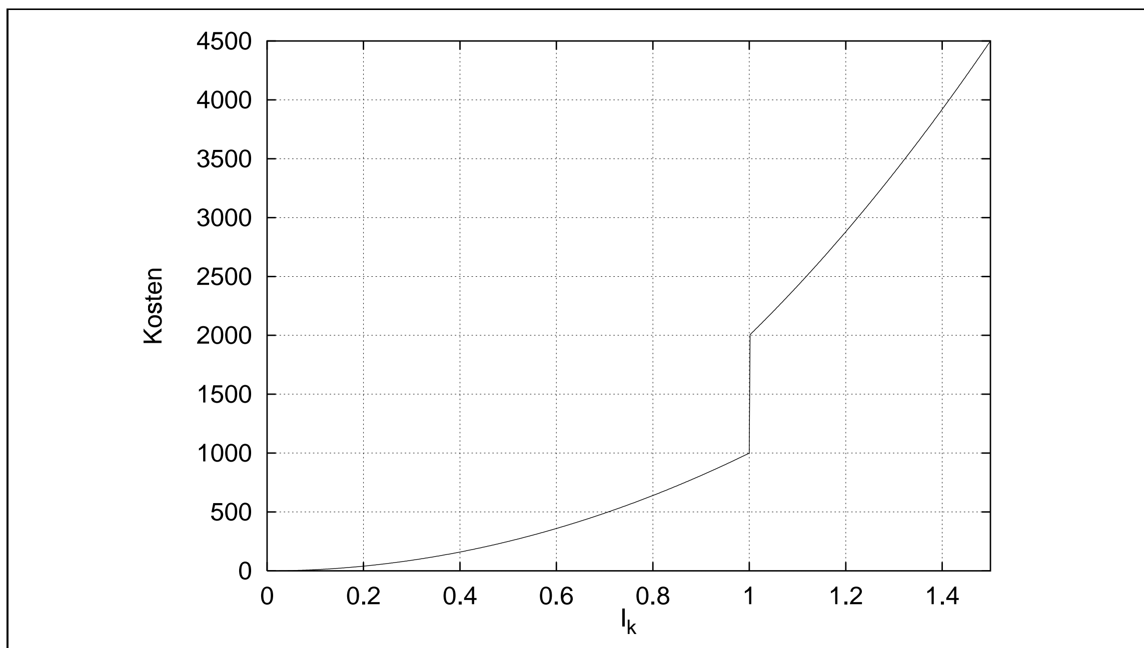


Abbildung 4.7: Kommunikationskosten für ein Kommunikationselement

4.1.2.4 Zusammenfassung

Zusammengefaßt ergibt sich aus den vorherigen Abschnitten folgende Bewertungsfunktion:

$$
\begin{aligned}
K_{\text{ges}} = & \sum_{r|\text{real}} K_{\text{HW,real}}(r) && \text{(reale Rechner)}\\
& + \sum_{r|\text{virtuell}} a_v * K_{\text{real,max}} * \left(\frac{P_r}{P_{\max}}\right)^{e_v} && \text{(virtuelle Rechner)}\\
& + \sum_{r|\text{real}} \left(\frac{l_r - l_{r,\min}}{1 - l_{r,\min}}\right)^2 * \min \left\{ \begin{array}{l} a_b * (K_{\text{real},R} - K_{\text{real},R^-}) \\ a_t * (K_{\text{real},R^+} - K_{\text{real},R}) \end{array} \right\} && \text{(Auslastung)}\\
& + a_r * N_r && \text{(Restriktionsverletzungen)}\\
& + a_k * \left(\sum_{k|l_k<=1} l_k^2 + \sum_{k|l_k>1} a_v * l_k^{e_v} \right) && \text{(Kommunikation)}
\end{aligned}
$$

Applikationsspezifisch sind dabei folgende Parameter:

- a_v Sprungfaktor für virtuelle Kosten
- e_v Exponent für virtuelle Kosten
- a_b Faktor zur Kosteninterpolation zu billigeren Rechnern
- a_t Faktor zur Begrenzung der Kosteninterpolation bzgl. teureren Rechnern
- a_r Gewichtungsfaktor für Restriktionsverletzungen
- a_k Gewichtungsfaktor für Kommunikationskosten

4.1.3 Änderungsfunktion

Mit der Änderungsfunktion werden aus der momentanen Konfiguration durch Parameteränderungen neue Konfigurationen erzeugt. Die durchgeführten Änderungen sollen *klein* sein, um nicht ziellos im Parameterraum herumzuspringen. Weiterhin sollen sie *zufällig* durchgeführt werden, d. h. ein zielgerichtetes Vorgehen wird an dieser Stelle explizit vermieden, um die Gefahr zu verringern, in lokalen Minima „festzufahren“.

Die Zufälligkeit der Änderungen kann dabei insofern beschränkt werden, daß bei der Parameteränderung ungültige Konstellationen von vornherein vermieden werden. Bei einigen Optimierungsproblemen, insbesondere wenn der gültige Lösungsraum sehr beschränkt ist, wurde allerdings festgestellt, daß eine Mißachtung von Restriktionen in der Änderungsfunktion und eine entsprechende schlechte Einstufung in der Bewertungsfunktion zu besseren Ergebnissen führte [CDM92].

Für das vorliegende Problem wird in jedem Optimierungsschritt eine der folgenden Änderungen zufällig ausgewählt:

1. Eine zufällig ausgewählte Task oder Taskgruppe wird auf einen anderen Rechner verschoben. Dabei können auch Rechner wegfallen oder zusätzliche Rechner hinzukommen. Ist die Task bzw. Taskgruppe Teil eines Präzedenzsystems, hat dies auch Auswirkungen auf das rechnerübergreifende Nachrichtenaufkommen.
2. Die Länge der Zeitscheibe eines Rechners für den TDMA–Buszugriff wird verändert.
3. Die Länge des gesamten Zugriffszyklus zum Nachrichtentransportsystem wird verändert; die Zeitscheiben der einzelnen Rechner werden proportional angepaßt.

Die Wahrscheinlichkeiten für die Wahl der Änderungsfunktion sind einstellbar. Bei den Genetischen Algorithmen sind diese Änderungen Bestandteil des Operators „Mutation“; zusätzliche Änderungen erfolgen hier durch den problemunabhängigen Operator „Crossover“ (vgl. Kapitel 4.6). Der *Typ* eines Rechners ist kein freier Parameter für die Änderungsfunktion, da er wie in Abschnitt 4.1.2.1 beschrieben dynamisch angepaßt wird.

4.1.3.1 Taskallokation

Die Taskallokation hat maßgeblichen Einfluß auf die Kosten, da damit die Anzahl und nötige Leistung der Rechner bestimmt und das Nachrichtenaufkommen auf dem Nachrichtentransportsystem festgelegt wird.

Die Änderung der Taskallokation läuft in folgenden Schritten ab:

1. Zufällige Auswahl einer Task (unter Beachtung von Restriktionen; vgl. Kapitel 4.1.3.3)

2. Mit einstellbarer Wahrscheinlichkeit:
   a) Verschiebung auf einen zufällig ausgewählten anderen Rechner, der bereits in der Konfiguration vorhanden ist, oder
   b) Verschiebung auf einen neuen Rechner (Wahrscheinlichkeit p_n)
3. Entfernung des bisherigen Rechners, falls
   a) nach der Verschiebung keine Task mehr auf diesen Rechner alloziert ist, oder
   b) mit der Wahrscheinlichkeit p_k, wobei die noch auf diesem Rechner befindlichen Tasks auf zufällig ausgewählte andere Rechner verschoben werden.

Die Wahrscheinlichkeit p_n muß relativ gering gewählt werden, um zu häufige Erweiterungen des Rechnerpools zu vermeiden, während die Allokation auf die bisherigen Rechner noch nicht optimiert ist. Zwar wird dadurch auch eine evtl. nötige Ausweitung des Rechnerpools verlangsamt (z. B. bei schlechtem Startwert mit lediglich einem Rechner), dies kann aber durch eine verbesserte Startwertwahl umgangen werden (siehe Kapitel 4.1.1).

Analog dazu muß auch die Wahrscheinlichkeit p_k zur Entfernung eines nicht–leeren Rechners klein gehalten werden. Da diese Änderung umso größere Auswirkungen hat, je mehr Tasks auf diesem Rechner alloziert waren, wird p_k in Abhängigkeit von der Taskanzahl n_t festgelegt zu

$$p_k(n_t) = p_{k_0} * \frac{1}{{n_t}^{e_k}}$$

p_{k_0} und e_k sind dabei vom Anwender festzulegende Optimierer–Parameter.

4.1.3.2 Zwischendeadlines bei rechnerübergreifender Kommunikation

Das zugrunde liegende Taskmodell erlaubt es, in Präzedenzsystemen, bei denen alle Tasks auf dem gleichen Rechner liegen, jeder Task die *gleiche* Deadline zu vergeben (Kapitel 3.1). Dies ist nicht mehr möglich, sobald das Präzedenzsystem Rechnergrenzen überschreitet: Wegen der unabhängig voneinander durchgeführten lokalen Prozessor– und Netzwerkzuteilungen sind jetzt Deadlines für die Vorgänger–Tasks auf einem Rechner und für die über das Kommunikationssystem übermittelte Nachricht so zu vergeben, daß für die Nachfolgertasks auf anderen Rechnern genügend Rechenzeit bis zur Deadline verbleibt.

Es ist nicht sinnvoll, diese Aufgabe der Festlegung von „Zwischendeadlines“ für Tasks und Nachrichten von vornherein dem Anwender aufzubürden, da dadurch ein restriktiveres System als nötig modelliert würde: Diese Task–Zwischendeadlines müßten ja

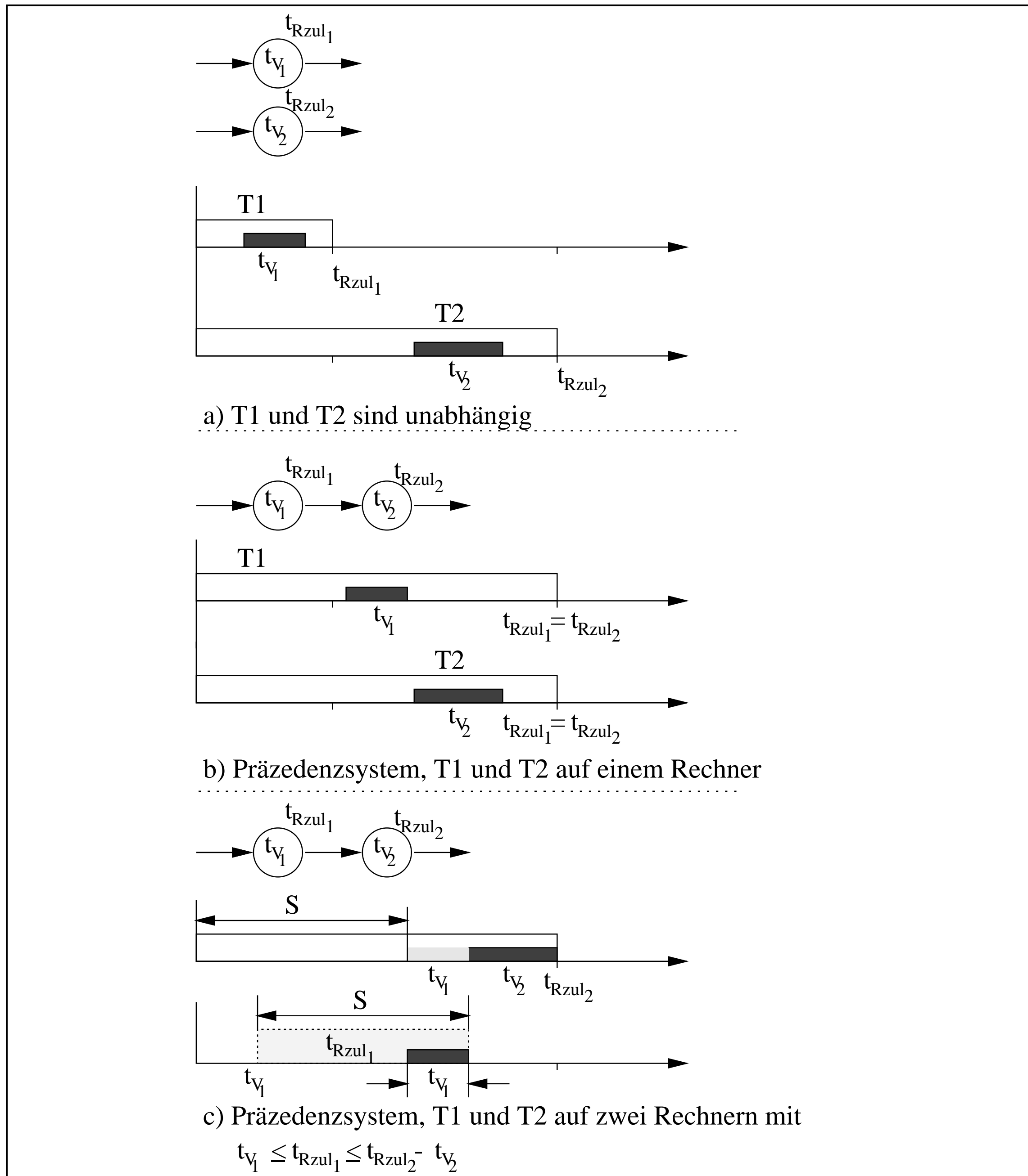


Abbildung 4.8: Spielraumverteilung bei Kommunikation

selbst dann eingehalten werden, wenn das gesamte Präzedenzsystem auf einen Rechner alloziert würde.

Abbildung 4.8 auf der vorherigen Seite illustriert diese Zusammenhänge: Für das Präzedenzsystem aus T1 und T2 (b) sind die Task–Deadlines $t_{R\mathrm{zul}}$ identisch. Eine (mögliche) Zwischendeadline sollte nicht von vornherein vorgesehen werden, da sich sonst generell eine restriktivere Situation ähnlich wie bei unabhängigen Tasks (a) ergibt. Nur wenn wirklich rechnerübergreifende Kommunikation stattfindet, muß ein Zwischendeadline t_{Rzul_1} unter Berücksichtigung des vorhandenen Spielraums $t_S = t_{\mathrm{Rzul}_2} - t_{v_2} - t_{v_1}$ gesetzt werden (in diesem Beispiel unter Vernachlässigung der Kommunikationszeiten).

Allgemein ergibt sich der insgesamt zu vergebende Spielraum t_S eines linearen Präzedenzsystems mit einheitlicher Deadline $t_{R\mathrm{zul}}$ zu

$$t_S = t_{R\mathrm{zul}} - \sum_i t_{v_i} - \sum_n t_{k_n}$$

mit den Rechenzeiten t_{v_i} für ein gesamtes Präzedenz–Teilsystem i und den Nachrichtenübertragungszeiten t_{k_n} für über Rechnergrenzen hinweg übermittelte Nachrichten n. Bei I Teilsystemen und $N = I - 1$ Nachrichten muß der Spielraum t_S auf $(2I-1)$ Bereiche verteilt werden, d. h. $(2I-2)$ Zwischendeadlines sind festzulegen. Eine zeitliche Verschiebung einer Zwischendeadline nach hinten bedeutet dabei eine Entlastung des sendenden Systems (mehr Spielraum vorhanden) und eine Belastung des empfangenden Systems (weniger Spielraum vorhanden).

Zur Beschränkung der Komplexität wird die Vergabe der Zwischendeadlines nicht optimiert, sondern so festgelegt, daß der Spielraum proportional zur Bearbeitungszeit auf die Teilsysteme und Nachrichtenübermittlungen verteilt wird.

4.1.3.3 Behandlung von Restriktionen

Bezüglich der Task–Allokation sind im Systemmodell folgende Einschränkungen vorgesehen:

1. Eine Task ist auf einen bestimmten Rechnertyp festgelegt, da beispielsweise bestimmte Hardware–Voraussetzungen gegeben sein müssen.

Diese Restriktion wird bereits in der Änderungsfunktion eingehalten, da es ansonsten bei einer etwas größeren Anzahl von Tasks und Rechnern sehr unwahrscheinlich wird, eine gültige Konfiguration zu erhalten: Eine Task, die auf einen bestimmten Rechnertyp angewiesen ist, wird von einer Verschiebung auf einen anderen Rechner ausgenommen, und zusätzlich wird der Typ dieses Rechners nach der Realzeitanalyse nicht an die erforderliche Leistung angepaßt. Zwar wird so dem Anwender die Verantwortung überlassen, Tasks dieser Art so zu verteilen, daß auch die Realzeitbedingungen eingehalten werden;

in der Praxis dürfte dieser Fall allerdings relativ selten sein, sodaß diese Einschränkung erlaubt ist.

2. Zwei oder mehrere Tasks müssen auf denselben Rechner alloziert werden (z. B. wegen gemeinsamer kritischer Bereiche, Kommunikation über globale Variablen etc.).

Auch diese Restriktion wird schon in der Änderungsfunktion eingehalten, indem die ganze Taskgruppe wie eine einzige Task bezüglich der Verschiebung auf einen anderen Rechner behandelt wird. Die Taskgruppe wird dadurch unbeweglicher gegenüber der Verschiebung einzelner Tasks, da durch die Verschiebung der ganzen Gruppe eine größere Kostenänderung hervorgerufen wird, wodurch bei fortgeschrittener Optimierung diese Änderung mit höherer Wahrscheinlichkeit verworfen wird. Andererseits würde aber bei temporär erlaubter Aufspaltung der Taskgruppe eine große Zahl von Optimierungsschritten nötig, bis alle Tasks wieder auf einem Rechner alloziert wären.

3. Zwei oder mehrere Tasks müssen auf unterschiedliche Rechner alloziert werden, z. B. redundante Replikate fehlertoleranter Tasks

Lediglich diese letzte Einschränkung wird alternativ in der Änderungsfunktion beachtet oder in der Bewertungsfunktion berücksichtigt (siehe Seite 29), da hier die Wahrscheinlichkeit hoch ist, eine nicht erlaubte Allokation im Laufe der Optimierung wieder zu korrigieren.

4.1.3.4 Änderung der Kommunikations–Parameter

Wie auf Seite 34 beschrieben wurde, kann in der Änderungsfunktion eine der beiden folgenden Änderungsmöglichkeiten für die Kommunikations–Parameter ausgewählt werden:

1. Die Länge der Zeitscheibe $t_{a,i}$ eines Rechners i für den TDMA–Buszugriff wird verändert.

Dazu wird ein zufälliger neuer Wert $t'_{a,i}$ im Bereich

$$(1-k_t)\,t_{a,i} \leq t'_{a,i} \leq (1+k_t)\,t_{a,i}$$

mit einstellbarem Parameter k_t $(0 \leq k_t \leq 1)$ gewählt. Der gesamte Zyklus t_p wird entsprechend verlängert oder verkürzt.

2. Die Länge des gesamten Zugriffszyklus zum Nachrichtentransportsystem wird verändert.

Hier wird nach der gleichen Methode wie oben der gesamte Zyklus t_p modifiziert; die einzelnen Zeitscheiben $t_{a,i}$ werden proportional angepaßt.

Der Gesamtzyklus t_p wird weiterhin bei Erzeugung eines neuen Rechners um die Zeitscheibe von dessen Kommunikationselement vergrößert sowie bei Entfernung eines Rechners um die entsprechende Zeitscheibe verkleinert.

Die Änderungen der Zeitparameter des Kommunikationssystems erfolgen in einem quasi–kontinuierlichen Wertebereich. Stochastische Optimierungsalgorithmen für diskrete Parameterräume sind für diese Probleme nicht optimal geeignet. Hier wäre möglicherweise der Einsatz der Evolutionsstrategien nach [Rec73] sinnvoll; die Kombination mit diesem Verfahren wurde allerdings nicht weiter untersucht.

4.2 Simulated Annealing

4.2.1 Algorithmus

Der Algorithmus nach Kirkpatrick, Gelatt und Vecchi [KGV83] ist in Abbildung 4.9 dargestellt. Ausgehend von einem als ersten Lösungsansatz zufällig ausgewählten Startpunkt X_0 im Parameterraum wird durch eine kleine, zufällige Änderung der Parameterwerte ein benachbarter Punkt X_{test} ermittelt. Entsprechend der Analogie zum physikalischen Vorgang werden die Kosten $\text{Cost}(X)$ X als Energie aufgefaßt, und die Energiedifferenz ΔC entscheidet über Annahme oder Verwerfung der geprüften Parameteränderung. Als Annahmebedingung wird bei Simulated Annealing der Metropolis–Algorithmus [MRTT53] verwendet: Bei einer Kostenverbesserung ($\Delta C < 0$) wird die neue Lösung X_{test} auf jeden Fall Ausgangspunkt für die nächste Iteration. Um nicht in einem lokalen Optimum steckenzubleiben, wird aber auch eine Kostenverschlechterung mit der Wahrscheinlichkeit $e^{-\Delta C/T}$ in Abhängigkeit von der momentanen Temperatur T akzeptiert; T wird im Laufe des Optimierungsvorgangs allmählich nach einem Abkühlplan erniedrigt.

Wähle zufälligen Startpunkt $X = X_0$
Wähle Anfangstemperatur $T = T_0$
repeat
  repeat
    $X_{\text{test}} = \text{ChangeOf}(X)$
    $\Delta C = \text{Cost}(X_{\text{test}}) - \text{Cost}(X)$
    if $\Delta C \leq 0$ **then**
      $X = X_{\text{test}}$
    else
      $r = \text{random}()$ (gleichverteilt; $0 < r < 1$)
      if $r < e^{-\Delta C/T}$ **then**
        $X = X_{\text{test}}$
      end if
    end if
  until „Temperaturausgleich“
  Erniedrige T nach Abkühlplan
until „keine Verbesserung mehr erzielbar“

Abbildung 4.9: Algorithmus „Simulated Annealing“

Die Anfangstemperatur T_0 soll so eingestellt werden, daß zunächst nahezu alle Änderungen akzeptiert werden. Dies kann automatisch erfolgen, indem solange „aufgeheizt“ wird, bis das Verhältnis von angenommenen zu abgelehnten Änderungen annähernd 100% beträgt [KGV83, Sec88].

Als Abkühlfunktion wird in [KGV83] eine empirisch ermittelte Reihe oder eine einfache Multiplikation mit einer Konstanten vorgeschlagen:

$$\text{CoolDown}(T) = \alpha * T \quad \text{mit} \quad 0 < \alpha < 1$$

Diese Funktion wird in der Literatur häufig benutzt mit Werten für α im Bereich von 0.8 - 0.95 [TBW92, Sec88, WLL88]. Zur Beschleunigung des Optimierungsvorgangs wird der Faktor α auch dynamisch in Abhängigkeit von der Geschwindigkeit der Kostenverbesserung angepaßt [Sec88].

Das Temperaturniveau wird solange konstant gehalten, bis „Temperaturausgleich“ erreicht ist. Dies wird in der Literatur so definiert, daß die Anzahl von akzeptierten Änderungen [KGV83] oder auch nur von Änderungen mit Kostenverbesserungen [TBW92] einen anwendungsabhängig gewählten Wert übersteigt, oder — um annähernd endlose Schleifen insbesondere bei niedrigen Temperaturen zu vermeiden — bis die Anzahl der insgesamt durchgeführten Versuche auf einem Temperaturniveau eine obere Grenze überschreitet. Geeignete Werte werden in der Regel experimentell bestimmt.

Der Algorithmus wird abgebrochen, wenn über mehrere (üblicherweise drei) Temperaturniveaus hinweg keine Änderung mehr akzeptiert wurde, was gleichzeitig bedeutet, daß auch keine Verbesserung mehr erzielbar ist.

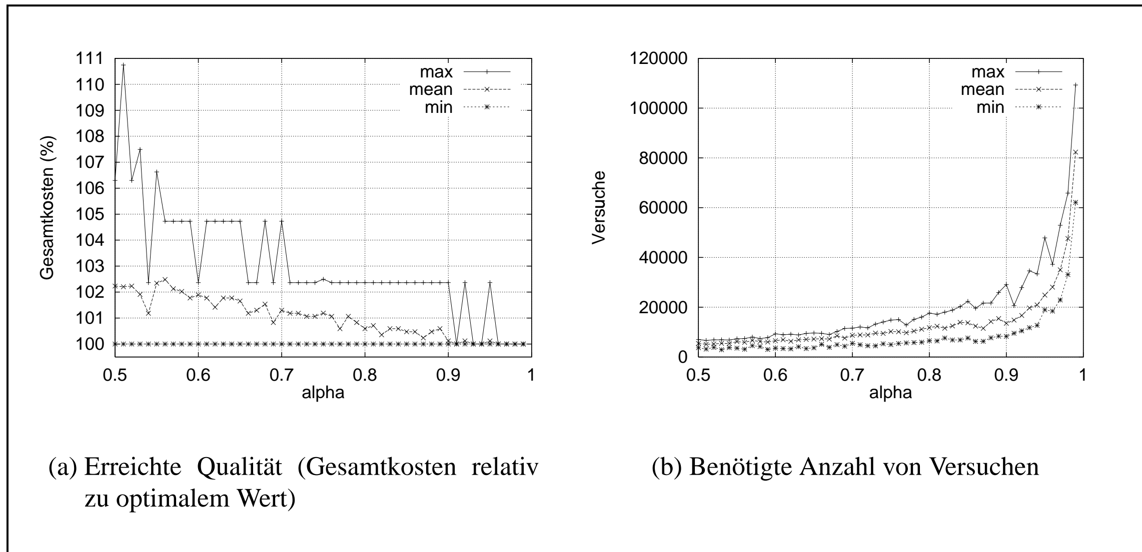
4.2.2 Verhalten des Algorithmus

Die folgenden Experimente wurden jeweils für die Beispielmodelle E und R durchgeführt. Als Startkonfiguration waren immer alle Tasks auf einem (virtuellen) Rechner angeordnet. Dies ist für die Optimierung ein relativ ungünstiger Startwert, da dem System zunächst einige Rechner hinzugefügt werden müssen, bevor überhaupt eine realisierbare (suboptimale) Konfiguration erreicht wird.

Die Optimierung wurde mit jeder Parametereinstellung 20 mal mit unterschiedlichen Startwerten für den Zufallszahlengenerator durchlaufen.

4.2.2.1 Abkühlung α

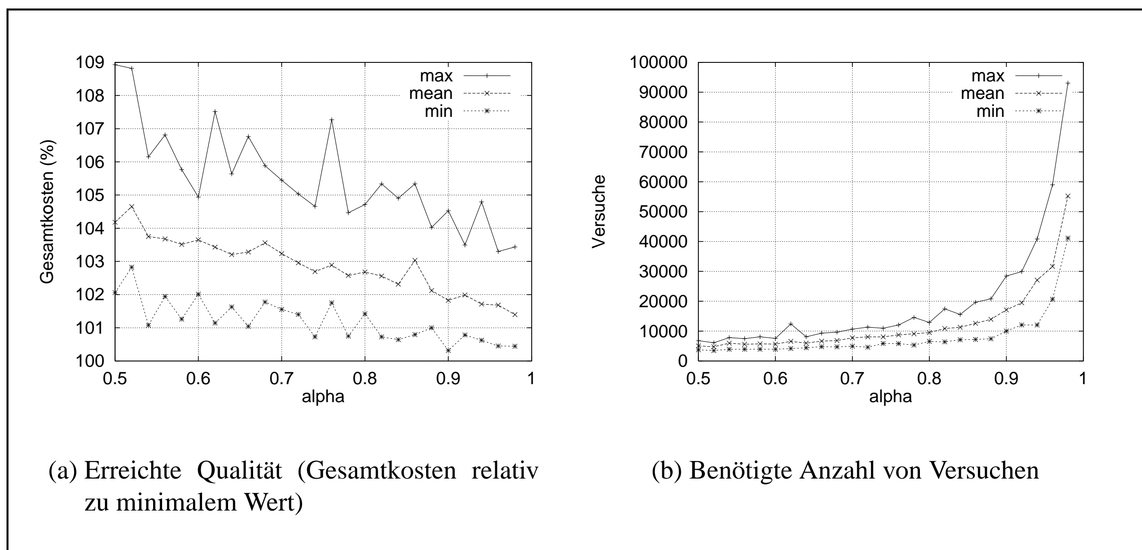
Abbildung 4.10 zeigt den Einfluß des Abkühlungsfaktors α auf die erreichte Qualität sowie die durchgeführte Anzahl von Versuchen für das Beispielmodell E. Als Anfangstemperatur T_0 wurde ein Wert von 15000 verwendet; dies entspricht in etwa der anfänglichen maximal auftretenden Kostendifferenz, sodaß zunächst sicher alle Änderungen das Akzeptanzkriterium erfüllen (vgl. Algorithmus in Abbildung A.1 auf Seite 100). Das Temperaturniveau wird mit α gesenkt, sobald entweder mehr als $c_{temp} = 50$ Änderungen akzeptiert oder mehr als $t_{max} = 1000$ Versuche auf diesem Niveau durchgeführt wurden.



(a) Erreichte Qualität (Gesamtkosten relativ zu optimalem Wert)

(b) Benötigte Anzahl von Versuchen

Abbildung 4.10: SA: Einfluß von α in Beispiel E



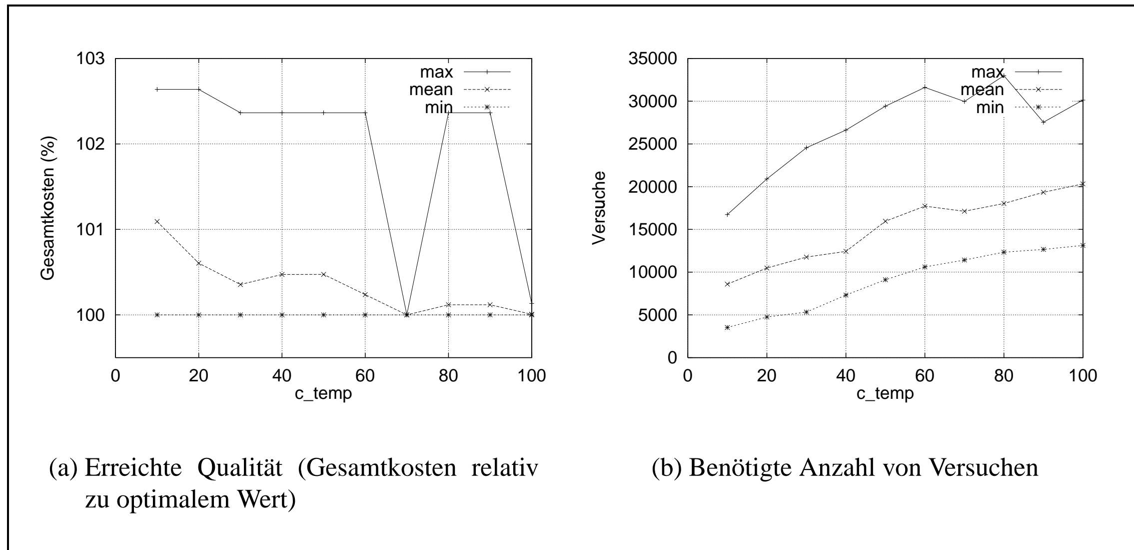
(a) Erreichte Qualität (Gesamtkosten relativ zu minimalem Wert)

(b) Benötigte Anzahl von Versuchen

Abbildung 4.11: SA: Einfluß von α in Beispiel R

Die Optimierung wird abgebrochen, sobald auf einem Temperaturniveau $t_{\max}$ Versuche durchgeführt werden, ohne daß eine neue Konfiguration akzeptiert wurde.

Wie Abbildung 4.10(a) zu entnehmen ist, wird für dieses einfache Beispiel für jeden Wert von α mindestens einmal das globale Optimum gefunden. Mit steigendem α, d. h. mit langsamerer Abkühlung wird das Optimum häufiger gefunden, und auch die schlechtesten Kostenwerte nähern sich dem Optimum an, sodaß der Mittelwert bereits ab $\alpha = 0.9$ nahezu identisch mit dem optimalen Wert ist. Andererseits nimmt die Anzahl der Versuche, bis die Optimierung abgebrochen wird, ab diesem Wert von α rasch zu



(a) Erreichte Qualität (Gesamtkosten relativ zu optimalem Wert)

(b) Benötigte Anzahl von Versuchen

Abbildung 4.12: SA: Einfluß von c_{temp} in Beispiel E

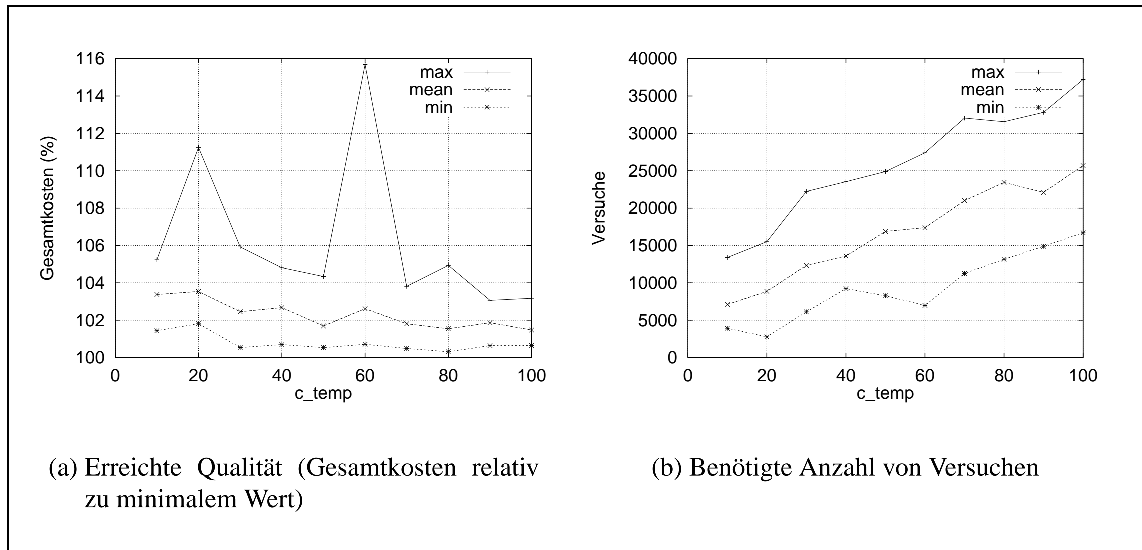
(Abbildung 4.10(b)).

Für Modell R ist die Abhängigkeit des Optimierungsverfahrens von α in Abbildung 4.11 dargestellt ($T_0 = 3000$, $c_{\text{temp}} = 50$, $t_{\text{max}} = 1000$). Wegen des komplexeren Modells wird das (hier experimentell ermittelte) Optimum nur in seltenen Fällen gefunden. Ab $\alpha = 0.9$ liegt aber auch der Mittelwert der Kosten in einem Bereich von nur 2% über dem Optimum; die Anzahl der Versuche steigt auch hier ab diesem Parameterwert schnell an.

4.2.2.2 Temperaturausgleich

Der zweite wesentliche Parameter von Simulated Annealing ist die Anzahl der akzeptierten Änderungen c_{temp}, die gefordert wird, bevor „Temperaturausgleich“ angenommen und das Temperaturniveau vermindert wird. Während die im Mittel erreichte Qualität sich mit wachsendem c_{temp} nur unwesentlich verbessert, steigt die Anzahl der Versuche relativ stark, aber nahezu linear an (Abbildung 4.12; $\alpha = 0.9$). Dies ist verständlich, da auf jedem Temperaturniveau mehr Versuche durchgeführt werden, die Anzahl der durchlaufenen Temperaturniveaus aber praktisch nur von α abhängt: Erst bei einer sehr kleinen Temperatur wird das Abbruchkriterium erfüllt, daß keine Änderung mehr auf einem Temperaturniveau akzeptiert wurde.

Auch das komplexere Modell R zeigt bei der Variation von c_{temp} ein ähnliches Verhalten (Abbildung 4.13; $\alpha = 0.9$). Insgesamt hat der Parameter c_{temp} — insbesondere bei der Anzahl der benötigten Versuche — in den untersuchten Fällen einen geringeren Einfluß als der Parameter α. Für eine Bestätigung dieser Tendenz müßten allerdings die Ver-



(a) Erreichte Qualität (Gesamtkosten relativ zu minimalem Wert)

(b) Benötigte Anzahl von Versuchen

Abbildung 4.13: SA: Einfluß von c_{temp} in Beispiel R

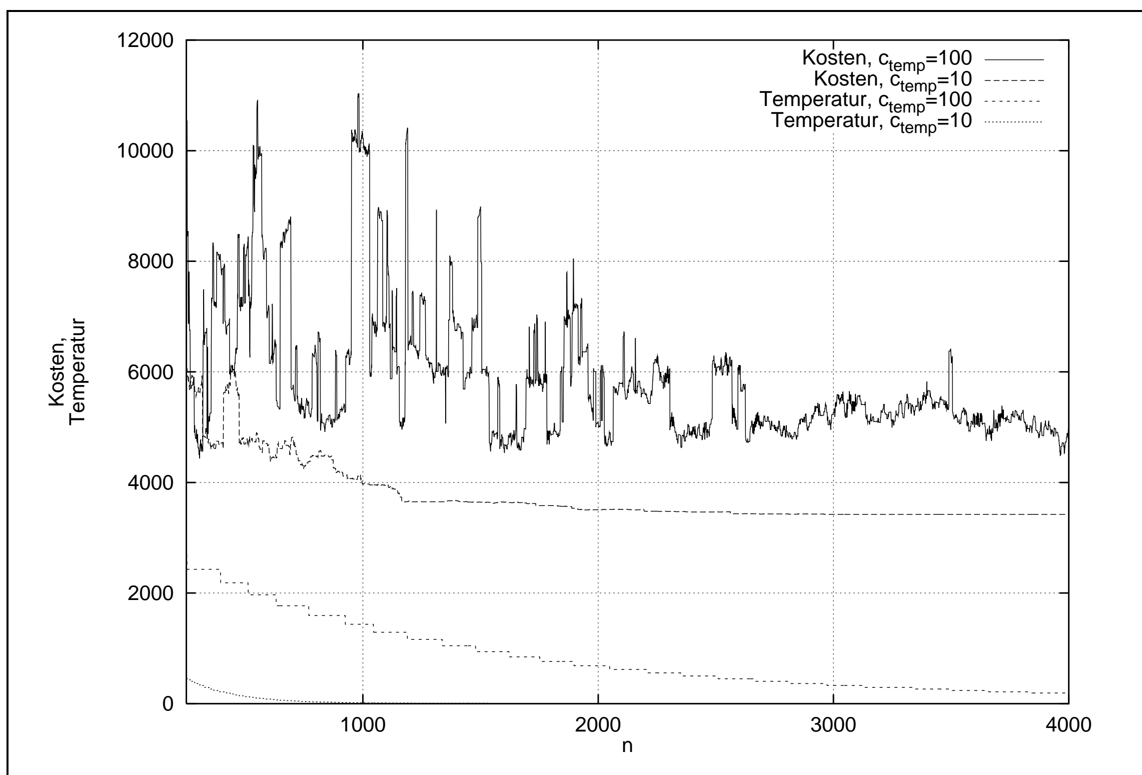


Abbildung 4.14: SA: Kostenverlauf für unterschiedliche Werte von c_{temp} bei Modell R

suchsreihen auch für andere Werte des jeweils fixen Parameters durchgeführt werden.

Der Kosten– und Temperaturverlauf über die Anzahl n der Versuche ist zum Vergleich für unterschiedliche c_{temp} in Abbildung 4.14 ausschnittsweise wiedergegeben (Werte für

$250 \leq n \leq 4000$ aus dem jeweils besten Durchlauf der Versuchsreihe). Während die Kosten der akzeptierten Konfiguration für $c_{\text{temp}} = 100$ entsprechend der noch hohen Temperatur von $T = 2700$ anfangs stark schwanken, ist der Kostenverlauf für $c_{\text{temp}} = 10$ an dieser Stelle (mit $T = 450$) auf niedrigerem Niveau schon wesentlich glatter. Eine Kreuzung der beiden Kostenkurven für $n > 4000$ findet nicht statt: Zwar wird mit $c_{\text{temp}} = 100$ schließlich ein besserer Kostenwert gefunden, aber erst bei $n = 25250$ wird der beste Wert von $c_{\text{temp}} = 10$ unterschritten — hier wurde aber schon bei $n = 11834$ abgebrochen. Mit $c_{\text{temp}} = 10$ konvergiert die Optimierung schneller, aber mit $c_{\text{temp}} = 100$ wird schließlich — mit wesentlich höherem Aufwand — ein besseres Ergebnis erreicht.

4.3 Threshold Accepting

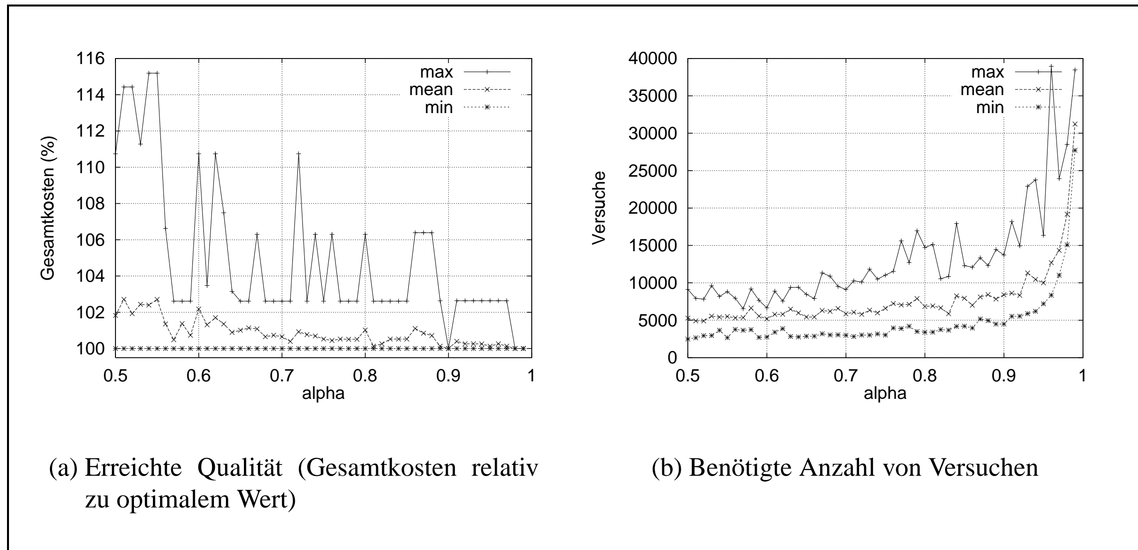
4.3.1 Algorithmus

Dueck und Scheuer [DS90, DSW93] formulierten „Threshold Accepting“ als Vereinfachung von Simulated Annealing. Der Unterschied zu Simulated Annealing liegt im wesentlichen in der Akzeptanzbedingung von Änderungen: Diese ist nun nicht mehr zufallsbehaftet, sondern eine Änderung wird *immer* dann akzeptiert, wenn die Kostenerhöhung eine Schwelle (Threshold) T nicht überschreitet (Abbildung 4.15). Verbesserungen werden immer angenommen. Durch die deterministische Akzeptanzbedingung kann Threshold Accepting aus lokalen Minima nicht mehr herausfinden, falls die Kostenerhöhung zu allen „Nachbarpunkten“ X_{test} größer als die Schwelle T ist, während dies bei Simulated Annealing — wenn auch mit geringer Wahrscheinlichkeit — gelingen kann.

Wähle zufälligen Startpunkt $X = X_0$
Wähle Anfangsschwelle $T = T_0$
repeat
    repeat
        $X_{\text{test}} = \text{ChangeOf}(X)$
        $\Delta C = \text{Cost}(X_{\text{test}}) - \text{Cost}(X)$
        if $\Delta C < T$ **then**
            $X = X_{\text{test}}$
        end if
    until „lange keine Verbesserung oder zu viele Versuche“
    Erniedrige Schwelle T
until „keine Verbesserung mehr erzielbar“

Abbildung 4.15: Algorithmus „Threshold Accepting“

Die innere Schleife (bis „lange keine Verbesserung“) wird in den Beispielen in [DS90] nach einer festen Anzahl n von Versuchen beendet. Als „Schwellenfahrplan“ wird eine empirisch ermittelte Sequenz von Werten verwendet oder eine triviale Sequenz von T_0 bis 0 mit festem Abstand ΔT zwischen den einzelnen Werten, wobei ca. 30 bis 100 verschiedene Werte angenommen werden. Die Wahl des „Schwellenfahrplans“ soll im Gegensatz zum „Temperaturfahrplan“ bei Simulated Annealing relativ geringe Auswirkungen auf das Optimierungsergebnis haben. Am Ende der Schwellwert–Sequenz ($T = 0$) wird die Optimierung beendet.



(a) Erreichte Qualität (Gesamtkosten relativ zu optimalem Wert)

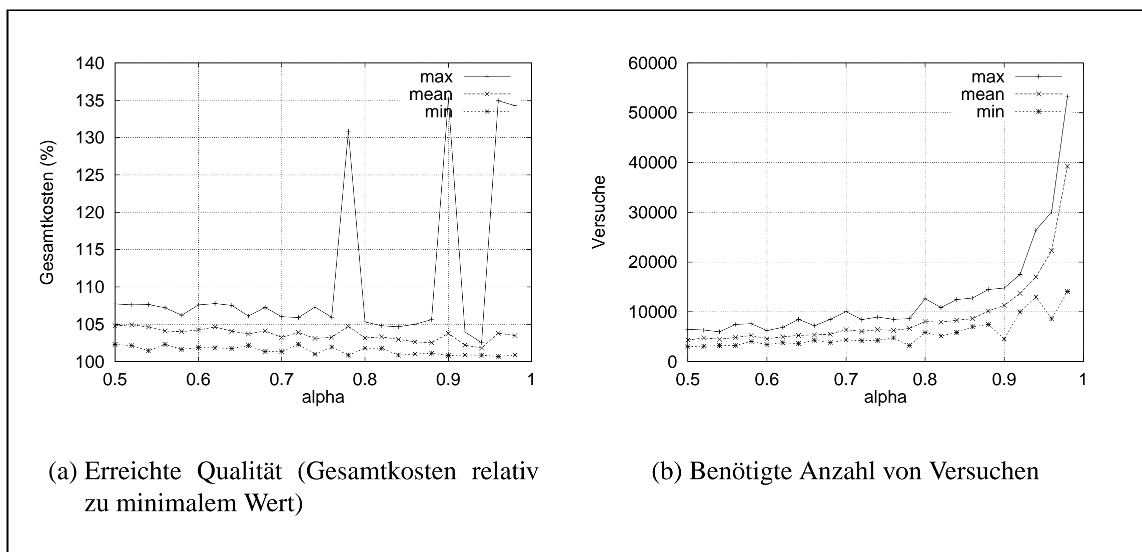
(b) Benötigte Anzahl von Versuchen

Abbildung 4.16: TA: Einfluß von α in Beispiel E

4.3.2 Verhalten des Algorithmus

Wegen der Ähnlichkeit mit Simulated Annealing ist zu erwarten, daß Threshold Accepting ein analoges Verhalten bei Änderung der Parameter zeigt. Dies wurde durch die Versuche bestätigt.

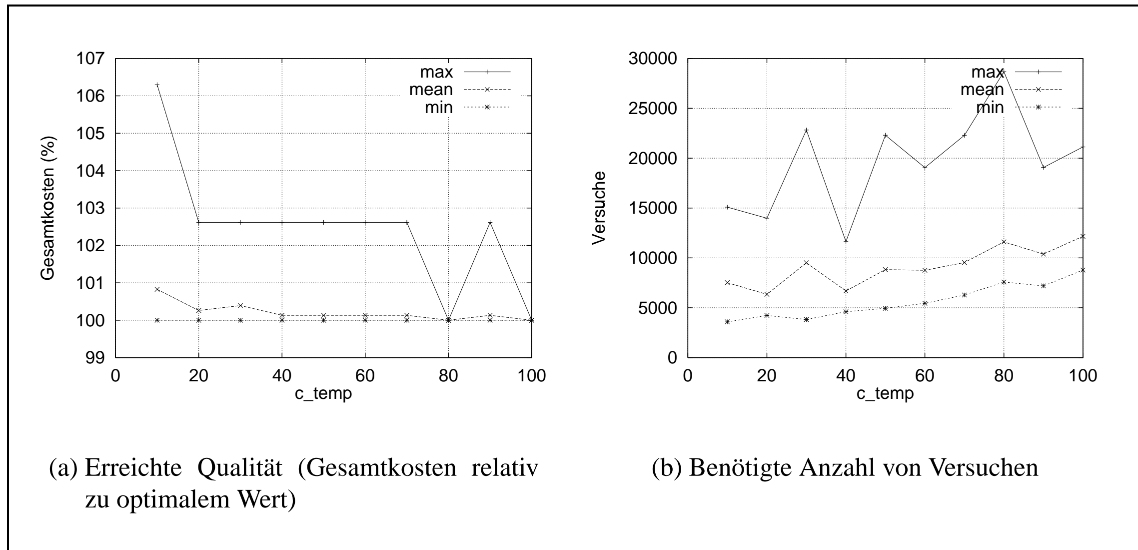
Für die gleichen Parameterwerte wie bei Simulated Annealing zeigt Abbildung 4.16 die Abhängigkeit von α bei der Optimierung von Modell E und Abbildung 4.17 von Modell R. Qualitativ ergibt sich in etwa der gleiche Verlauf wie bei Simulated Annealing, und auch die Mittelwerte der erreichten Gesamtkosten stimmen ungefähr überein. Durch die einfachere Akzeptanzbedingung werden aber insbesondere bei niedrigeren Schwellenwerten T seltener Verschlechterungen akzeptiert, sodaß insgesamt das Verfahren nach etwas weniger Versuchen terminiert. Dies äußert sich auch in stärkeren Ausschlägen der maximalen Kostenwerte.



(a) Erreichte Qualität (Gesamtkosten relativ zu minimalem Wert)

(b) Benötigte Anzahl von Versuchen

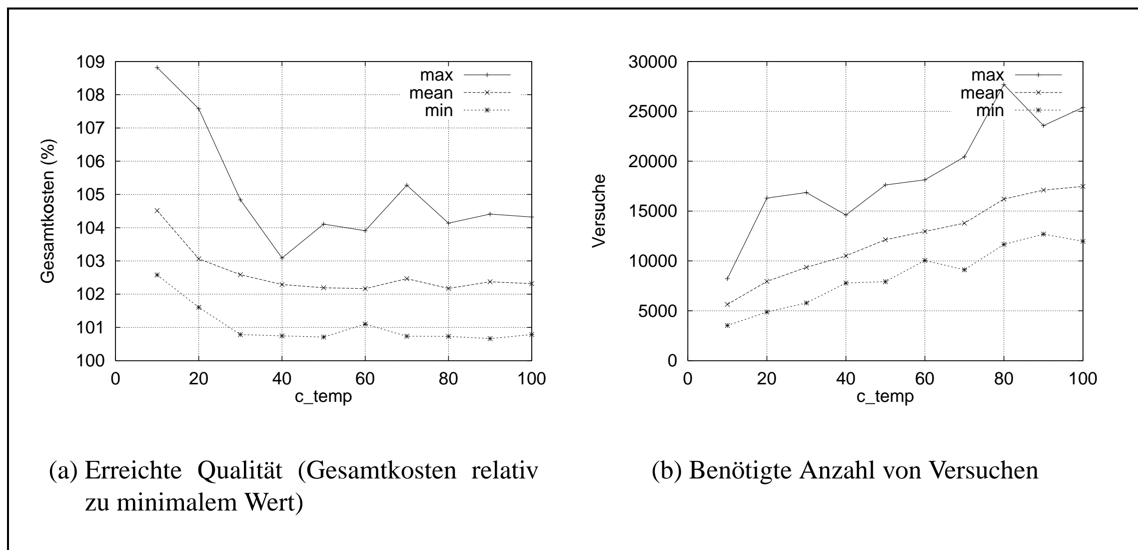
Abbildung 4.17: TA: Einfluß von α in Beispiel R



(a) Erreichte Qualität (Gesamtkosten relativ zu optimalem Wert)

(b) Benötigte Anzahl von Versuchen

Abbildung 4.18: TA: Einfluß von c_{temp} in Beispiel E



(a) Erreichte Qualität (Gesamtkosten relativ zu minimalem Wert)

(b) Benötigte Anzahl von Versuchen

Abbildung 4.19: TA: Einfluß von c_{temp} in Beispiel R

Das Verhalten des Algorithmus bei Änderungen von c_{temp} ist in Abbildung 4.18 (Modell E) und Abbildung 4.19 (Modell R) wiedergegeben. Insbesondere bei Modell R ist hier gegenüber Simulated Annealing eine deutliche Verschlechterung der mittleren Qualität für kleine c_{temp} zu erkennen, was wiederum auf die frühere Erfüllung der Abbruchbedingung zurückzuführen ist.

4.4 Record–to–Record–Traveling

4.4.1 Algorithmus

Eine weitere Vereinfachung stellte Dueck in [Due93] mit „Record–to–Record–Traveling“ vor: Nicht die Kostendifferenz zwischen einem neuen Punkt X_{test} im Parameterraum und dem aktuellen Punkt X_{test} wird für die Akzeptanzbedingung herangezogen, sondern die Kostendifferenz zwischen dem neuen Punkt X_{test} und dem bisherigem Bestwert („Record“) an Punkt X_R. Diese Differenz darf eine festgelegte Abweichung nicht überschreiten (Abbildung 4.20). Damit ist kein „Temperaturfahrplan“ oder „Schwellenfahrplan“ mehr nötig, und gegenüber Threshold Accepting entfällt ein Schleifenkörper. Auch bei dieser Variante können lokale Minima nicht mehr verlassen werden, wenn sie mehr als D „tief“ sind.

```
Wähle zufälligen Startpunkt X_R = X = X_0
Wähle maximale Abweichung D
repeat
  X_test = ChangeOf(X)
  ΔC = Cost(X_test) − Cost(X_R)
  if ΔC < D then
    X = X_test
    if ΔC < 0 then
      X_R = X_test
    end if
  end if
until „lange keine Verbesserung oder zu viele Versuche“
```

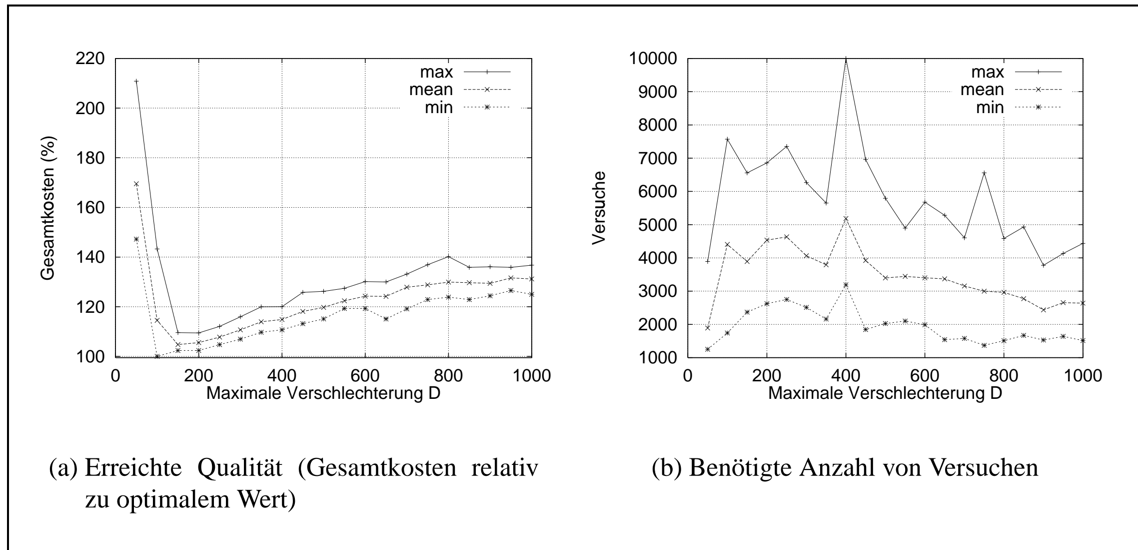
Abbildung 4.20: Algorithmus „Record–To–Record–Traveling“

Wie bei Threshold Accepting wurde in [Due93] die Abbruchbedingung einfach als feste Anzahl von Versuchen implementiert.

4.4.2 Verhalten des Algorithmus

Für dieses Optimierungsverfahren wurde die Variation der maximalen Abweichung D vom bisherigen Bestwert untersucht sowie die maximale Anzahl $t_{\text{s,max}}$ von Änderungsversuchen ohne Kostenverbesserung, nach der die Optimierung beendet wird. Die mit $t_{\text{max}} = 60000$ eingestellte absolute Obergrenze von Versuchen wurde nie erreicht.

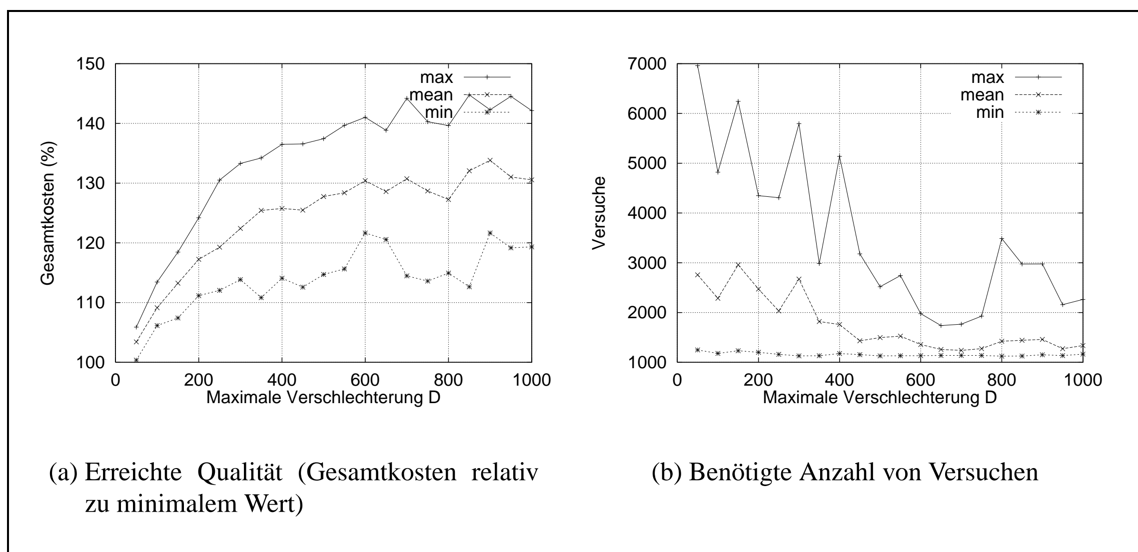
Für die Optimierung von Modell E zeigt sich bei Variation von D (mit $t_{\text{s,max}} = 1000$), daß dieses Modell offenbar ausgeprägte lokale Minima hat: Wird D kleiner als die Kosten für den schwächstmöglichen Rechner (Typ C2; vgl. Tabelle B.1 auf Seite 109), wird



(a) Erreichte Qualität (Gesamtkosten relativ zu optimalem Wert)

(b) Benötigte Anzahl von Versuchen

Abbildung 4.21: RTR: Einfluß von Abweichung D in Beispiel E

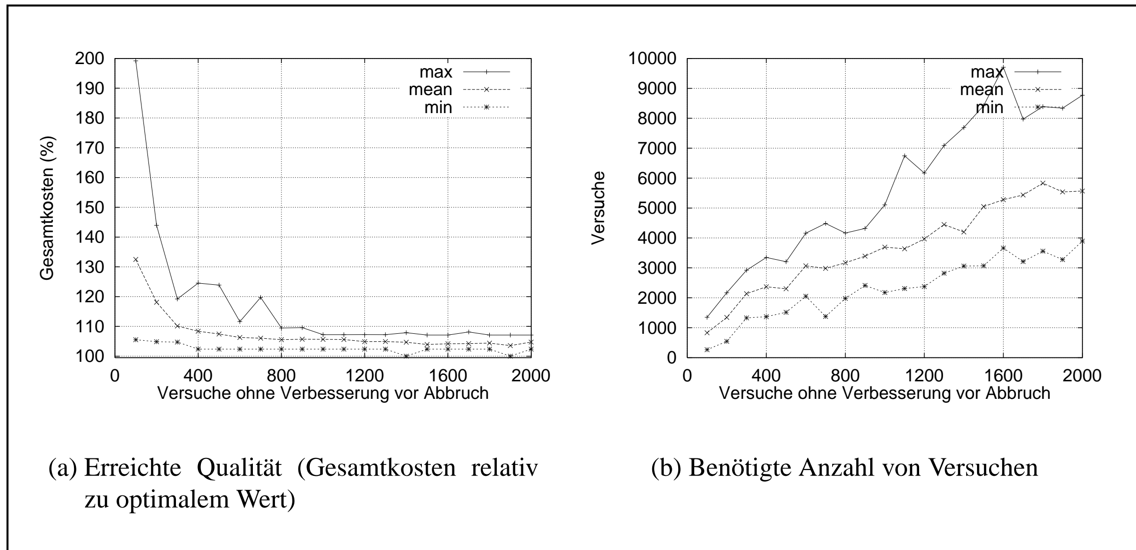


(a) Erreichte Qualität (Gesamtkosten relativ zu minimalem Wert)

(b) Benötigte Anzahl von Versuchen

Abbildung 4.22: RTR: Einfluß von Abweichung D in Beispiel R

die erreichbare Qualität deutlich schlechter, da eine zeitweise Kostenverschlechterung bei Hinzunahme eines weiteren Rechners nicht toleriert wird (Abbildung 4.21(a)); das Verhalten des Algorithmus wird für $D \to 0$ dem Verfahren „Hill–Climbing" immer ähnlicher. Andererseits wird die erreichbare Qualität aber auch wieder — jedoch langsamer — schlechter, je weiter D den optimalen Wert von ca. 150 überschreitet, da dann die zielgerichtete Komponente immer schwächer wird; mit $D \to \infty$ entartet das Verfahren zu „Monte–Carlo". Mit der erreichten Qualität korrespondiert auch direkt die Anzahl der benötigten Versuche (Abb. 4.21(b)).



(a) Erreichte Qualität (Gesamtkosten relativ zu optimalem Wert)

(b) Benötigte Anzahl von Versuchen

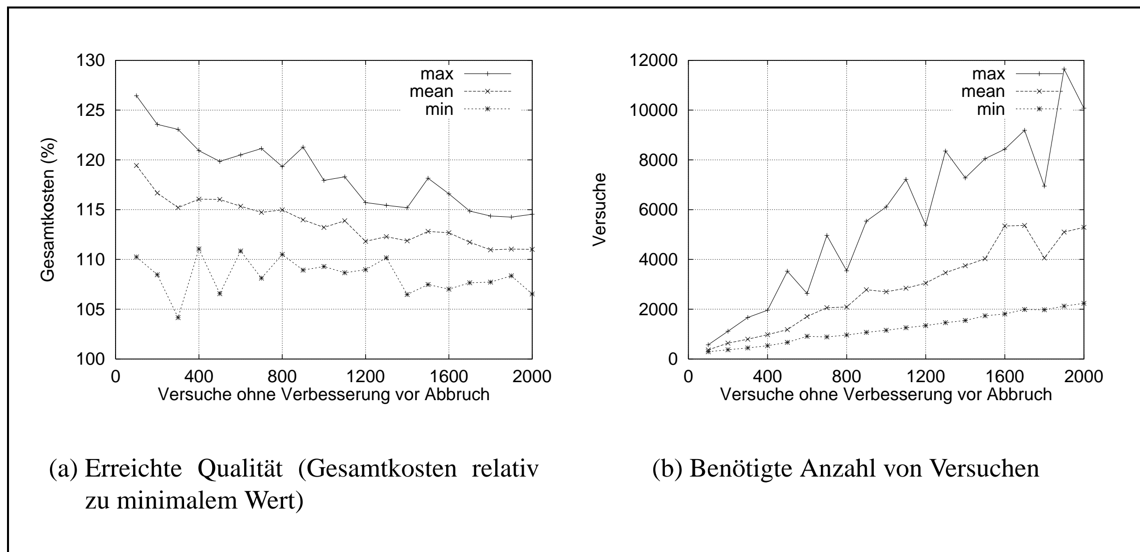
Abbildung 4.23: RTR: Einfluß von $t_{s,\max}$ in Beispiel E

Da Modell R eine feinere Verteilung bezüglich der nötigen minimalen Rechnerleistung für die einzelnen Tasks hat (vgl. Tabelle B.2 auf Seite 111), ist hier keine Qualitätsverschlechterung für kleine Werte von D zu verzeichnen (Abbildung 4.22(a)).

Mit den gewählten Parametern ist die erreichte Güte im Mittel um einiges schlechter als bei Simulated Annealing oder Threshold Accepting, allerdings werden auch wesentlich weniger Versuche benötigt. Die Wahl der maximalen Abweichung D hat einen deutlich größeren Einfluß als die Parameterwahl bei den vorhergehenden Optimierungsverfahren.

Der Einfluß von $t_{s,\max}$ auf die Qualität ist etwas geringer: Ab einem Wert von $t_{s,\max} = 500$ zeigt sich bei Modell E praktisch kein Einfluß mehr auf die erreichbare Güte (Abbildung 4.22(a); $D = 150$), obwohl die Anzahl der Versuche bis zum Abbruch weiter steigt (Abbildung 4.22(b)). Unter diesem Wert terminiert der Algorithmus einfach zu früh, sodaß die Wahrscheinlichkeit, eine Konfiguration nahe am Optimum zu finden, rasch kleiner wird.

Bei dem komplexeren Modell R ist der Einfluß von $t_{s,\max}$ noch etwas kleiner als bei Modell E, aber über den gesamten untersuchten Wertebereich erkennbar (Abbildung 4.24; $D = 150$). Deutlich sichtbar ist hier, daß mit der Erhöhung von $t_{s,\max}$ und der damit verursachten höheren Anzahl durchgeführter Versuche zwar eine deutliche Tendenz der mittleren Kosten zu kleineren Werten hin gegeben ist, damit aber eben nur die *Wahrscheinlichkeit* steigt, daß innerhalb einer begrenzten Anzahl von Versuchen ein dem Optimum naher Wert gefunden wird: Die kostengünstigste Konfiguration wurde hier bei einem der 20 Optimierer–Läufe mit $t_{s,\max} = 300$ gefunden.



(a) Erreichte Qualität (Gesamtkosten relativ zu minimalem Wert)

(b) Benötigte Anzahl von Versuchen

Abbildung 4.24: RTR: Einfluß von $t_{s,max}$ in Beispiel R

4.5 Great Deluge Algorithm

Ebenfalls in [Due93] stellte Dueck ein weiteres Optimierungsverfahren vor, das wie Record–To–Record–Traveling keinen Temperatur– oder Schwellenfahrplan benötigt. Statt eines Minimierungsproblems wie bei Threshold Accepting und Record–to–Record–Traveling wird hier ein Maximierungsproblem betrachtet. Hierfür werden keine Kostendifferenzen, sondern die absoluten Kosten zur Akzeptanzentscheidung herangezogen: Für das Maximierungsproblem werden alle Punkte mit Kosten über dem aktuellen Niveau („Wasserstand“) akzeptiert; das Niveau wird bei Akzeptanz einer Änderung um einen festen Betrag („Regen“) erhöht.

4.5.1 Algorithmus

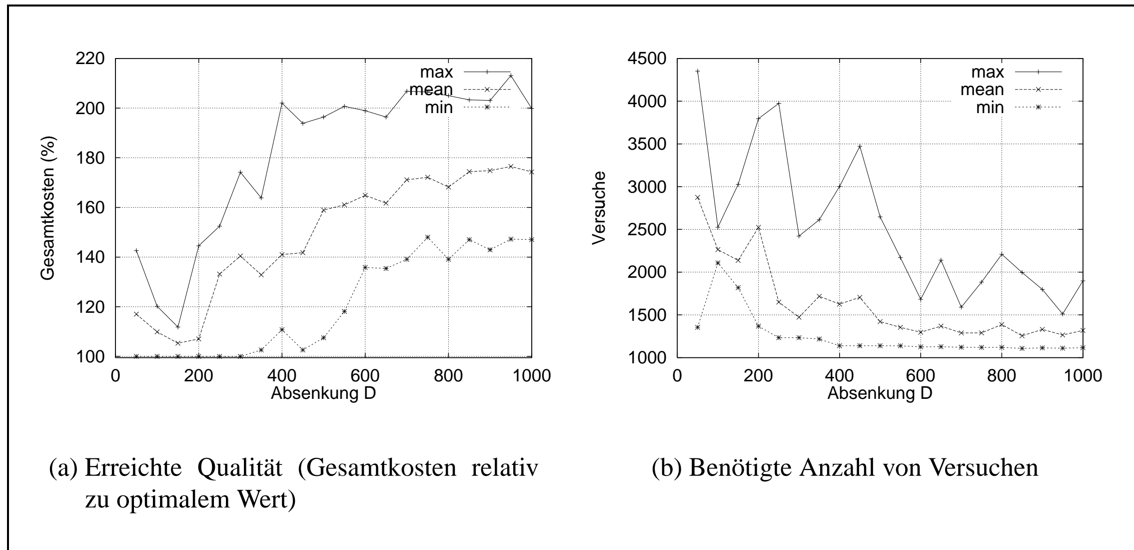
Abbildung 4.25 zeigt den auf ein Minimierungsproblem transformierten Algorithmus. Neben dem Parameter D für die Niveau–Verringerung ist wegen der Anwendung auf eine Minimierung zusätzlich ein Anfangsniveau N_0 nötig, das z. B. auf ein Vielfaches der Kosten des Startpunktes gesetzt werden kann (z. B. $N_0 = 1.5\text{Cost}(X_0)$).

```
Wähle zufälligen Startpunkt X = X_0
Wähle Verringerung D
Wähle Anfangsniveau N = N_0
repeat
  X_test = ChangeOf(X)
  if Cost(X_test) < N then
    X = X_test
    N = N − D
  end if
until „lange keine Verbesserung oder zu viele Versuche“
```

Abbildung 4.25: Algorithmus „Great Deluge“

Für die Abbruchbedingung wird wiederum eine feste Anzahl von Versuchen verwendet. In [Due93] wurde durchweg ein gutes Optimierungsverhalten für das Traveling–Salesman–Problem erreicht, wenn für die Verringerung D ein Wert von etwas weniger als 1% der mittleren Differenz zwischen $\text{Cost}(X)$ und Niveau N verwendet wurde. Zur Beschleunigung der Konvergenz wurde auch vorgeschlagen, die lineare Niveauänderung um D durch eine dynamische Anpassung der Niveausenkung nach z. B.

$$N = N - \max\left(D, \frac{N - \text{Cost}(P)}{500}\right)$$



(a) Erreichte Qualität (Gesamtkosten relativ zu optimalem Wert)

(b) Benötigte Anzahl von Versuchen

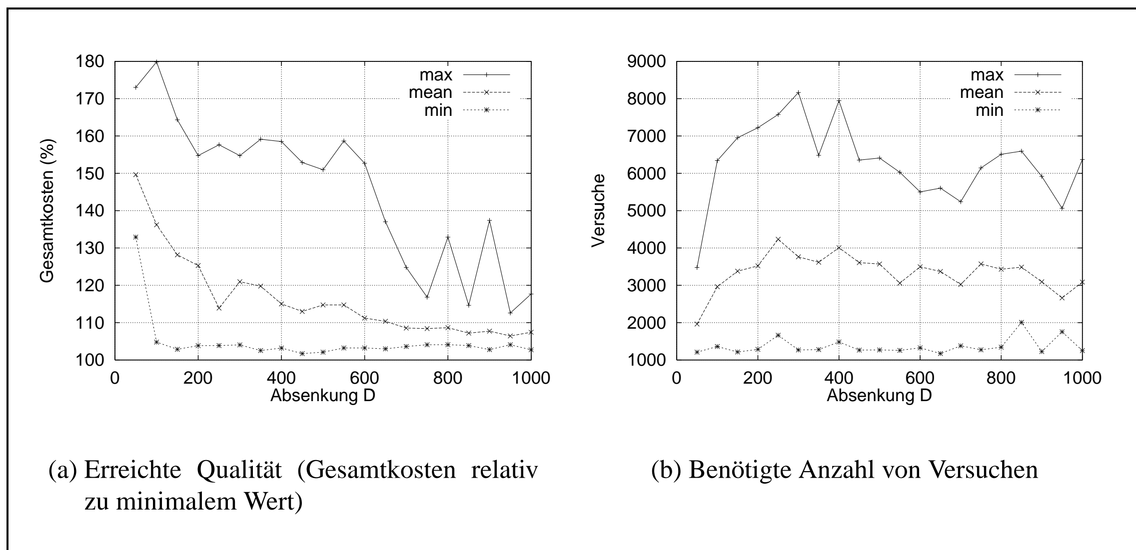
Abbildung 4.26: GD: Einfluß von Absenkung D in Beispiel E

zu ersetzen, um so anfangs bei einem schlechten Startwert und starken Verbesserungen das Niveau schneller zu senken (im Beispiel mit mindestens 0,2% der Niveau–Kosten–Differenz).

4.5.2 Verhalten des Algorithmus

Die wesentlichen Parameter dieses Optimierungsverfahrens sind die Absenkung D, um die der „Wasserstand" erniedrigt wird, wenn die Kosten der aktuellen Konfiguration unter diesem Pegel liegen, und die Anzahl $t_{\text{s,max}}$ von Versuchen, nach denen die Optimierung beendet wird, wenn keine Verbesserung des Optimums mehr erfolgte. Als initialer Wasserstand wurde jeweils der Kostenwert der Startkonfiguration verwendet.

In Abbildung 4.26 ist das Verhalten des Verfahrens bei Variation von D für Modell E dargestellt ($t_{\text{s,max}} = 1000$). Daß sich bei den Kosten — zumindest bei Mittelwert und Maximum — wie bei Record–To–Record–Traveling ein Optimum bei $D \sim 150$ ergibt, hat allerdings nichts mit der Ähnlichkeit der Verfahren zu tun: Da als Startwert die sehr schlechte Konfiguration „alle Tasks auf einem Rechner" gewählt wurde, ergibt sich wegen der Kosten des virtuellen Rechners ein sehr hoher initialer Wasserpegel $N_0 = \text{Cost}(X_0)$. Innerhalb der ersten wenigen hundert Versuche fallen die Kosten wegen der Hinzunahme weiterer Rechner und der dadurch schnell sinkenden Kosten für virtuelle Rechner sehr schnell ab; die Wahrscheinlichkeit, daß durch Wegfall von Rechnern wieder virtuelle Rechner benötigt werden, ist jedoch gering, sodaß der Kostenwert sich nun nicht mehr in allzugroßen Bereichen verändert. Der Wasserpegel sinkt aber nur linear ab und trifft z. B. bei $D = 50$ daher erst nach etwa 2000 Versuchen überhaupt



(a) Erreichte Qualität (Gesamtkosten relativ zu minimalem Wert)

(b) Benötigte Anzahl von Versuchen

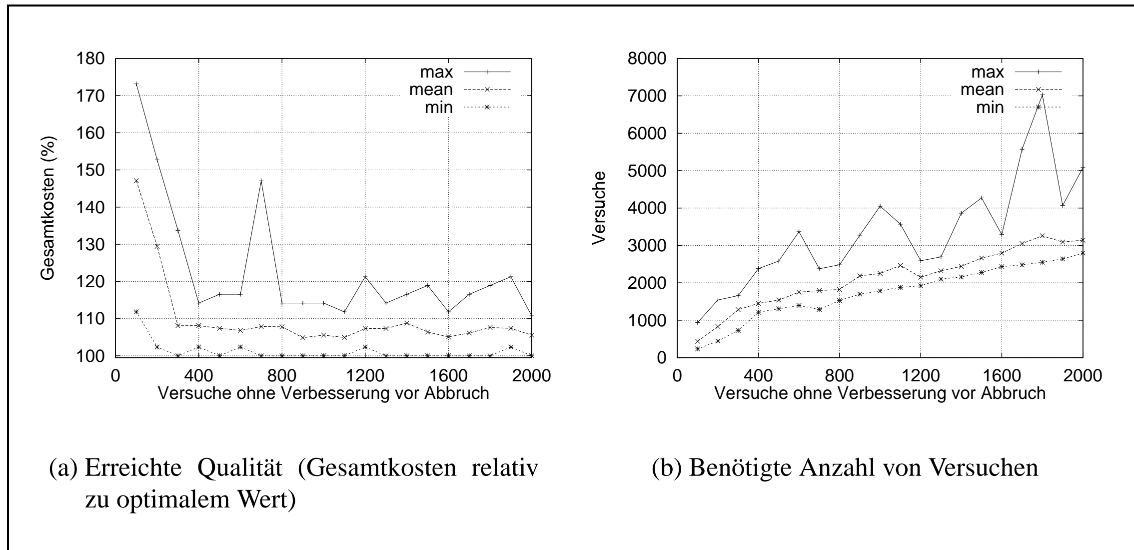
Abbildung 4.27: GD: Einfluß von Absenkung D in Beispiel R

mit dem aktuellen Kostenwert zusammen; erst ab diesem Punkt wirkt die zielgerichtete Komponente des Optimierungsverfahrens sehr stark. Mit einer gewissen Wahrscheinlichkeit ist aber bereits bei weniger als 2000 Versuchen das Abbruchkriterium erfüllt ($t_{s,\max} = 1000$), sodaß für einige Durchläufe mit $D = 50$ nur wenige zielgerichteten Konfigurationsänderungen (d. h. Ablehnungen von Verschlechterungen) erfolgen, was den schlechten Mittelwert erklärt. Für $D = 100$ werden die Konfigurationsänderungen bereits nach ca. 1000 Versuchen durch den Wasserpegel begrenzt, für $D = 150$ schon nach ca. 750 Versuchen, und in diesem Fall wird die Optimierung sicher nicht mehr zuvor abgebrochen.

Für größere Werte von D erfolgt dagegen die Absenkung wiederum so schnell, daß das Verfahren immer früher in lokalen Minima stagniert, die aufgrund der schnellen Pegelsenkung nicht mehr verlassen werden können. Dies spiegelt sich auch in den minimalen Werten für die Anzahl der durchgeführten Versuche wider (Abbildung 4.26(b)), die mit den maximalen Kostenwerten korrespondieren.

Eine Verbesserung des Verhaltens für $D < 150$ würde sich vermutlich ergeben, wenn bei starken Kostenverbesserungen eine dynamische Pegelabsenkung erfolgen würde, oder wenn ein besserer, kostengünstigerer Startwert gewählt würde; auch eine Erhöhung von $t_{s,\max}$ würde ein zu frühes Abbrechen verhindern, allerdings nicht die Anzahl nicht zielgerichteter Konfigurationsänderungen erniedrigen.

Der gleiche Effekt wie bei Modell E ist für kleine D auch bei Modell R zu beobachten (Abbildung 4.27). Durch den noch höheren initialen Kostenpegel als bei Modell E erfolgt eine Kostenbegrenzung noch später; dadurch tritt insbesondere bei mittleren Werten von D der Effekt auf, daß schon vor der Kostenbegrenzung eine relativ gute Konfiguration



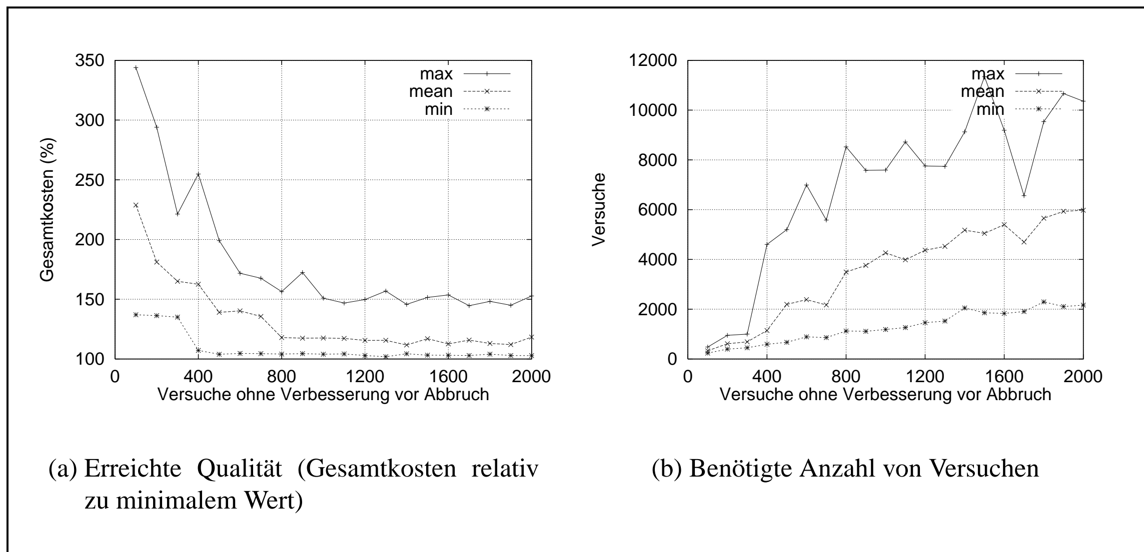
(a) Erreichte Qualität (Gesamtkosten relativ zu optimalem Wert)

(b) Benötigte Anzahl von Versuchen

Abbildung 4.28: GD: Einfluß von $t_{s,max}$ in Beispiel E

gefunden wird, von der sich das Optimierungsverfahren aber wegen des noch hohen Kostenpegels so weit entfernen kann, daß bei Einsetzen der Kostenbegrenzung dieser gute Wert nicht mehr erreicht werden kann — das Verfahren stagniert in einem lokalen Minimum. Für höhere Werte von D tritt dies seltener auf, da der Kostenpegel schneller sinkt.

Bezüglich des Parameters $t_{s,max}$ gilt ebenfalls die oben erläuterte Problematik: Wird $t_{s,max}$ zu klein gewählt, terminiert der Algorithmus bereits, bevor der Kostenpegel soweit gesunken ist, daß Konfigurationsänderungen zielgerichtet erfolgen, sodaß nur schlechte Kostenwerte erreicht werden (Abb. 4.28(a) für Modell E und Abb. 4.29(a) für Modell R; jeweils mit $D = 150$). Oberhalb dieser kritischen Werte hat $t_{s,max}$ nur geringen Einfluß auf die erreichbare Qualität; die Anzahl der benötigten Versuche wächst in etwa im gleichen Maße wie $t_{s,max}$ (Abb. 4.28(b) und 4.29(b)).



(a) Erreichte Qualität (Gesamtkosten relativ zu minimalem Wert)

(b) Benötigte Anzahl von Versuchen

Abbildung 4.29: GD: Einfluß von $t_{s,\max}$ in Beispiel R

4.6 Genetische Algorithmen

Für dieses Optimierungsverfahren wird eine Analogie zur Vererbungs- und Selektionslehre ausgenutzt, um die gestellte Optimierungsaufgabe zu lösen. Im nächsten Abschnitt werden zunächst die Vorgänge in der Biologie grob gezeigt, und in den folgenden Abschnitten die Übertragung auf die Genetischen Algorithmen beschrieben.

4.6.1 Biologisches Vorbild

Bei biologischen Individuen wird die Erbinformation, die die „Baupläne“ der Organismen enthält, in der doppelsträngig organisierten DNS (Desoxyribonukleinsäure) als Folge von vier verschiedenen Basen kodiert [Dud94] (Abb. 4.30). Durch den speziellen Aufbau des Doppelstrangs aus komplementären Basenpaaren kann durch Auftrennen des Doppelstrangs und Ergänzung der entsprechenden Basen eine identische Reduplikation der DNS erreicht werden, was vor jeder Zellteilung nötig ist, um die Erbanlagen von Zellgeneration zu Zellgeneration weiterzugeben. Träger der DNS im Zellkern sind die **Chromosomen**, die Teile des Zellkerns sind.

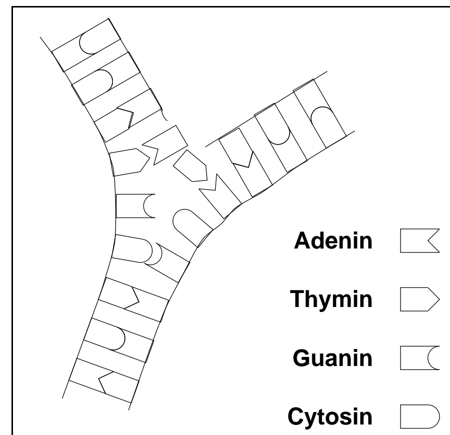


Abbildung 4.30: Duplikation von DNS

Eine definierte Basensequenz innerhalb der DNS, die die Information für ein Produkt oder für eine Funktion enthält (in Form der Bauanleitung für ein oder mehrere Proteine), wird als **Gen** bezeichnet; die Gesamtheit aller Gene eines Organismus ist das **Genom** [DTV90]. Die Zustände eines Gens, die das zugehörige Erbmerkmal unterschiedlich ausprägen (z. B. blaue/braune Augen), werden **Allele** genannt. Die Allele aller Gene auf den Chromosomen eines bestimmten Lebewesens sind im **Genotyp** zusammengefaßt [Dud94].

Die Anzahl der verschiedenen Chromosomen im Chromosomensatz verschiedener Tier– und Pflanzenarten schwankt zwischen zwei und einigen Hundert (z. B. beim Menschen: 46 Chromosomen), die in der Regel doppelt (diploid) vorhanden sind: jeweils ein Chromosom von der Mutter und eines vom Vater [Dud94] (z. B. bei der Frau: 23 Chromosomenpaare). Da somit jedes Gen zweimal vorhanden ist, kann jeder Zellkern zwei Allele eines Gens besitzen. Welches Allel äußerlich erkennbar wird, hängt davon ab, welches der beiden Allele **dominant** bzw. **rezessiv** ist. Die erkennbaren Allele bestimmen den **Phänotyp**, das äußere Erscheinungsbild eines Lebewesens. Da dieses nicht von allen

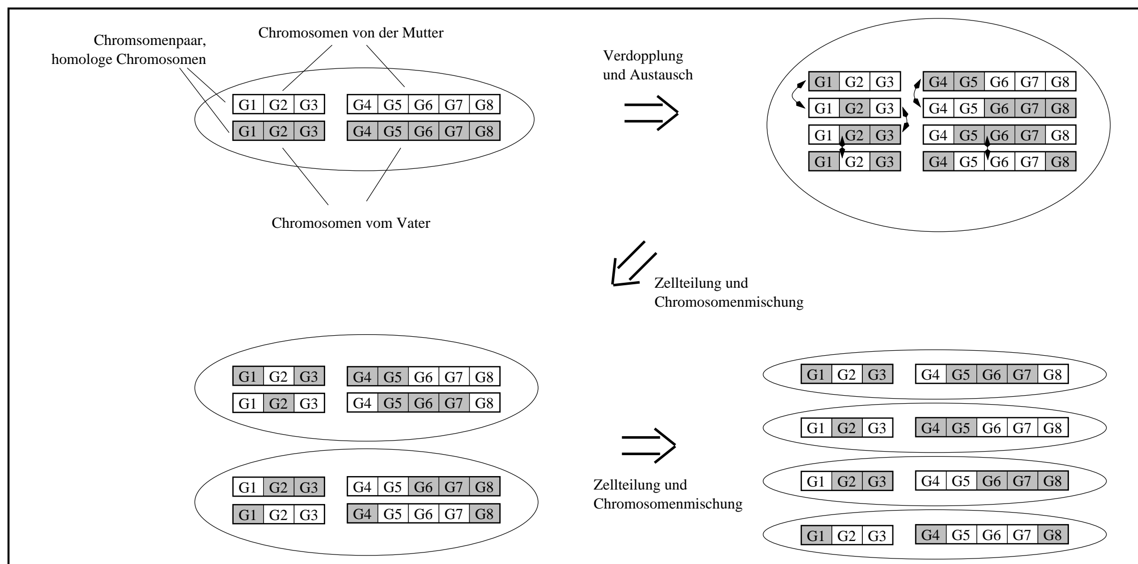


Abbildung 4.31: Bildung von Keimzellen

Allelen bestimmt wird, lassen sich sich daraus nicht ohne weiteres Rückschlüsse auf den Genotyp fassen [Len76].

Bei der Bildung der Geschlechtszellen der tierischen Organismen (Ei– bzw. Samenzellen) wird in der **Meiose** (Reduktionsteilung) der diploide Chromosomensatz halbiert und in einen haploiden Chromosomensatz umgewandelt. Dabei werden aus einer diploiden Urgeschlechtszelle vier (verschiedene) haploide elterliche Keimzellen gebildet: Zunächst werden die jeweils beiden homologen Chromosomen verdoppelt und paaren sich; es bilden sich **Überkreuzungsstellen** (Chiasma), an denen ein Stückaustausch zwischen den homologen Chromosomen erfolgt (**Faktentausch, Crossing Over**). Am Ende der ersten Reifungsteilung werden die homologen Chromosomen voneinander getrennt. In der folgenden zweiten Reifungsteilung werden die Chromosomenpaare voneinander getrennt, es enstehen die vier haploiden Keimzellen. Während der Zellteilungen werden die homologen Chromosomen jeweils zufällig auf die neuen Zellen verteilt (Abbildung 4.31).

Beim Faktentausch werden die Anteile aus Vater– und Mutter–Chromosom beim Menschen bei den meisten Chromosomen an ein bis vier Kreuzungsstellen gebildet (für 40% der Chromosomen an 2 Kreuzungsstellen, für 30% der Chromosomen an 3 Kreuzungsstellen). Die Anzahl der Kreuzungsstellen richtet sich auch nach der Länge der Chromosomen; das längste menschliche Chromosom bildet 10 bis 12 Chiasmata [Len76].

Es erfolgt also während der Meiose eine **Rekombination** der Chromosomen. Dies erfolgt zum ersten durch Mischung der vom Vater bzw. von der Mutter ererbten ganzen Chromosomen, und zum zweiten durch eine neue Zusammenstellung der Chromosomen aus den väterlichen und mütterlichen Genen.

Die Veränderung der *Struktur* einzelner Chromosomen wird als **Chromosomenaberration** bezeichnet und erfolgt durch Verlust, Austausch oder Verdopplung eines Chromosomenstücks [Dud94]. Bei der **Deletion** geht ein Teilstück eines Chromosoms verloren. Eine **Inversion** führt zu zum „verdrehten" Einbau eines Chromsomenstücks, d. h. die Gene dieses Teilstücks sind in ihrer Reihenfolge verdreht. Die **Duplikation** führt durch Einbau *beider* Teilstücke der am Faktentausch beteiligten Chromosomen zur Verlängerung des Chromosoms (und zur Deletion des beteiligten Chromosoms). **Translokation** schließlich bedeutet den Faktentausch zwischen nichthomologen Chromosomen, d. h. zwischen Chromosomen, die gänzlich verschiedene Gene enthalten. Chromosomenaberrationen führen in der Biologie sehr häufig zu stark krankhaften Erscheinungen (Mißbildungen).

Bei der **Mutation** wird unterschieden zwischen der **Genmutation**, der Veränderung einzelner Gene innerhalb eines Chromosoms, und der **Chromosomenmutation**, einer Änderung des Chromsomensatzes durch Hinzufügen oder Entfernen ganzer Chromosomen. Mutationen werden meist durch äußere Einflüsse hervorgerufen (Strahlung, chemische Stoffe).

Die beiden haploiden Chromosomensätze der väterlichen und mütterlichen Keimzellen verschmelzen bei der Befruchtung wieder zu einem diploiden Chromosomensatz. Nach der Darwin'schen Selektionslehre werden dabei günstige Konstellationen innerhalb einer Population von Individuen eher überleben und weitervererbt werden als ungünstige Konstellationen.

4.6.2 Grundalgorithmus

Bei den Genetischen Algorithmen existiert entsprechend eine Population von künstlichen Individuen, deren Erbinformation statt in Chromosomen in **Strings** finiter Länge aus einem finiten Alphabet gespeichert ist [Gol89]. Die Gene entsprechen einem bestimmten **Feature**, ein Allel damit einem **Feature–Wert**. Die Datenstruktur, in der die Erbinformation gespeichert wird, kann mit dem Genotyp verglichen werden; das Gegenstück zum Phänotyp ist schließlich der daraus extrahierte Parametersatz, d. h. eine Lösungsmöglichkeit des Optimierungsproblems.

Genetische Algorithmen arbeiten oft nur mit einem Chromosom, nicht mit einem ganzen Chromosomensatz, und dieses ist pro Individuum nur einfach (haploid) vorhanden. Somit spielt der Begriff der *dominanten* bzw. *rezessiven Gene* hier in der Regel keine Rolle.

Aus der Sicht eines „Individuums" läßt sich ein Optimierungszyklus in die folgenden Phasen gliedern (Abbildung 4.32):

Erbgutaustausch: Von Partnern wird deren String (Erbgut) empfangen, und der eigene

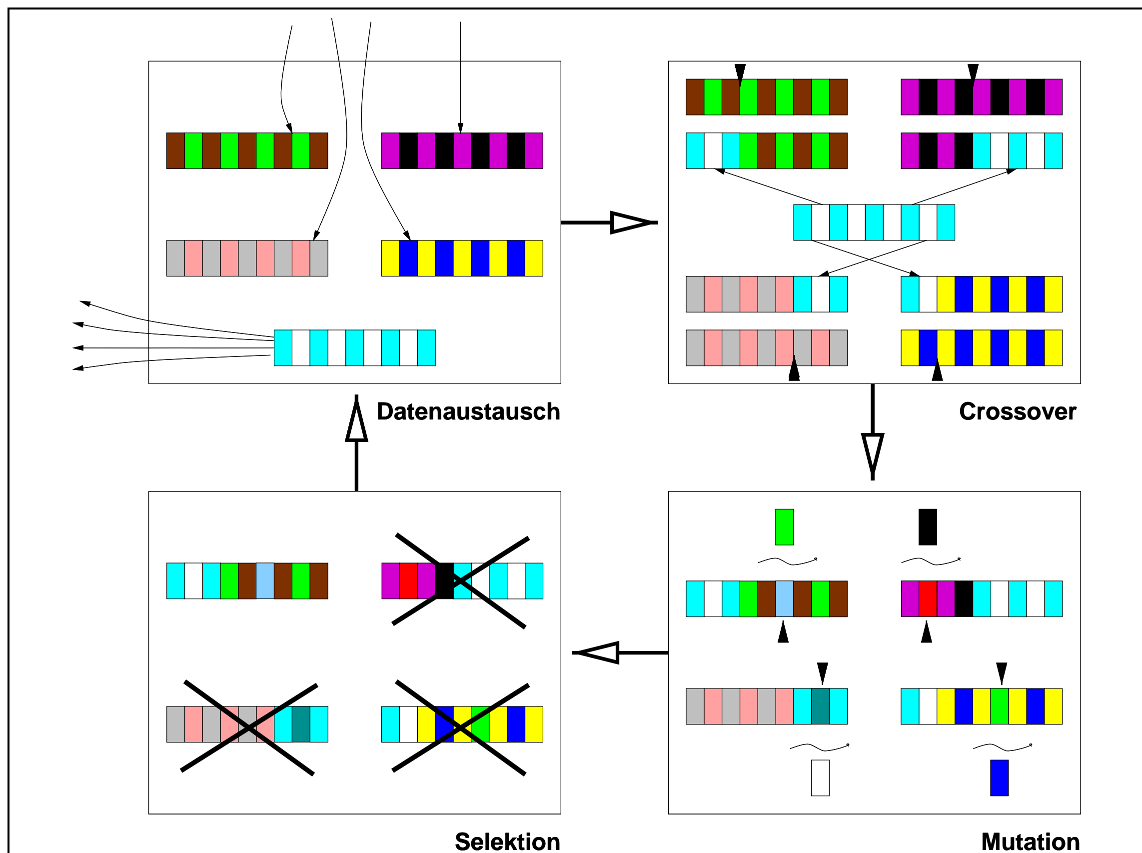


Abbildung 4.32: Optimierungszyklus der GA

String wird an Partner weitergegeben (Verschmelzung).

Crossover: Die eigenen Features werden mit denen des erhaltenen Erbmaterials gemischt; daraus ergibt sich ein neuer String.

Mutation: Der neue String wird zufällig verändert. Außer der Mutation können auch noch andere genetische Operatoren angewendet werden.

Selektion: Aus allen so neu gebildeten Strings, d. h. gleichzeitig Individuen, werden die in der Population überlebenden ausgewählt.

4.6.3 Kodierung

Durch die Kodierung werden die zu optimierenden Parameter auf das in den Genetischen Algorithmen verwendete Erbgut, die Strings, abgebildet. Sie bestehen aus einer festen Länge von Zeichen aus einem finiten Alphabet. Der strukturelle Aufbau der Strings, insbesondere auch die Abbildung auf die zu optimierenden Parameter, muß für alle Individuen identisch sein — andernfalls wären die Vererbungsmechanismen ja bedeutungslos.

Da die verwendeten Zeichen in Analogie zu Genen behandelt werden, sollte darauf geachtet werden, daß ein kleinstmögliches Alphabet verwendet wird, das eine „natürliche“ Kodierung des Problems erlaubt [Gol89]. Häufig wird das binäre Alphabet $\{0,1\}$ verwendet, das ohne weitere Konvertierung eine natürliche Darstellung ganzer Zahlen im Bereich $[0,2^l]$ mit einem String der Länge l gestattet. Auch Parameter andere Intervalle können durch eine lineare Skalierung leicht in ein Intervall $[0,2^l]$ konvertiert und damit binär dargestellt werden. Zur Kodierung mehrerer Parameter in einem String werden einfach die binären Kodierungen aneinandergehängt; die Anzahl der zur Kodierung verwendeten Bits kann pro Parameter variieren.

Für die genetischen Operatoren werden die Strings zwischen zwei Zeichen des Kodierungsalphabets aufgetrennt. Bei einer binären Kodierung bedeutet dies, daß die Auftrennung mitten in einem Parameterwert passieren kann: Mehrere „Allele“ repräsentieren hier den Parameterwert. Oft ist dies tolerierbar oder gar erwünscht; es kann dadurch aber auch zu unerlaubten Parameterwerten kommen (z. B. ist bei einer binären Kodierung der ganzen Zahlen $0 \ldots 9$ mit 4 Bit der binäre Wert 1111 nicht erlaubt). Diese unerlaubten Werte müssen dann durch einen Filteroperator korrigiert werden [CDM92] . Wird stattdessen ein Alphabet verwendet, das die Parameterwerte vollständig enthält, kann dieses Problem nicht auftreten, da immer ganze Parameter zwischen unterschiedlichen Strings ausgetauscht werden: Ein „Gen“ entspricht jetzt einem Parameter.

Auch die Position der kodierten Parameter auf dem String ist von Bedeutung, da weit auseinander liegende Zeichen bei Crossover–Operationen häufiger getrennt werden als benachbarte Zeichen. Die Positionierung kann durch den Inversionsoperator modifiziert und damit ebenfalls Gegenstand der Optimierung werden.

Im vorliegenden Problem liegen zwei unterschiedliche Parameterklassen vor: Die Taskallokation, bei der eine Beziehung Task zu Rechner beschrieben wird, und die Einstellung der TDMA–Parameter für die einzelnen Kommunikationselemente.

4.6.3.1 Taskallokation

Da die Anzahl der Tasks fix ist, die Anzahl der Rechner dagegen während der Optimierung variiert, wird eine Kodierung der (Task–)Parameter „Task x ist alloziert auf Rechner y“ vorgenommen. Die reziproke Kodierung der (Rechner–)Parameter „Rechner y enthält Tasks $x_1, x_2, \ldots$ “ wäre wegen der nicht konstanten Stringlänge ungünstig.

Für n Tasks ist die maximale Anzahl von verwendeten Rechenelementen ebenfalls n. Die Rechner erhalten die Indizes $1 \ldots n$. Um beim Crossover immer gültige Werte zu erhalten, wird als Kodierungsalphabet das Intervall $[1,n]$ verwendet. Die Implementierung ist damit einfach möglich als Array von n Integerwerten, das für die Tasks $1 \ldots n$ den jeweils zugewiesenen Rechnerindex enthält.

4.6.3.2 Kommunikationsparameter

Für jeden Rechner i mit rechnerübergreifender Kommunikation muß für sein Kommunikationselement die TDMA–Zykluszeit $t_{p,i}$ und die Länge der Zugriffszeit $t_{a,i}$ innerhalb des Zyklus angegeben werden. Um trotz der variierenden Anzahl der verwendeten Rechner zu einer konstanten Stringlänge zu kommen, wird jeder rechnerübergreifend kommunizierenden Task j ein TDMA–Zeitanteil $t_{a,j}$ zugeordnet; die insgesamt benötigte Zugriffszeit für den Rechner i kann dann für die Analyse aufsummiert werden:

$$t_{a,i} = \sum_j t_{a,j}$$

Die Werte $t_{a,j}$ können in einem zweiten String kodiert werden (d. h. dann existieren zwei „Chromosomen“), oder als zusätzliches Element in das für die Kodierung der Taskallokation verwendete Array aufgenommen werden. Im ersten Fall werden die TDMA–Parameter unabhängig von der Taskallokation vererbt, was zu einer größeren Genvielfalt führt. Die zweite Variante hat dagegen den Vorteil, daß bei einer Verschiebung von Tasks auf einen anderen Rechner die möglicherweise bereits aufeinander abgestimmten Zeitanteile am TDMA–Zyklus mitgenommen werden. In der beispielhaften Implementierung wurde die erste Variante verwendet.

Zur Vereinfachung wird im System nur eine für alle Rechner identische Zykluszeit t_p zugelassen, die sich damit aus der Summe der Zugriffszeiten berechnen läßt und nicht eigens optimiert werden muß:

$$t_p = \sum_i t_{a,i}$$

Alle Zeitwerte werden in der Optimierung als Integerwerte gespeichert. Reale Zeiten können daraus durch Multiplikation mit einem einstellbaren Zeitfaktor (z. B. 1 μs) berechnet werden. Der Umrechnungsfaktor muß so gewählt werden, daß einerseits die Auflösung der internen Darstellung ausreichend groß ist, andererseits bei den Rechenoperationen kein Überlauf stattfindet. Dies stellt in der Praxis bei Verwendung von 32 bit oder gar 64 bit–Werten aber keine Einschränkung dar.

4.6.3.3 Initialbelegung

Für die Genetischen Algorithmen ist eine große „Genvielfalt“ wünschenswert, damit über die Vererbungsmechanismen viele Parameterkombinationen getestet werden können. Dies kann über eine zufällige Auswahl der Startparameter erfolgen.

Bezüglich der Taskallokation wird dazu für jede Task eine zufällige Rechnernummer aus dem Bereich $[1, n]$ gewählt. Für die den Tasks zugeordneten TDMA–Zeitanteilen muß vom Anwender eine Obergrenze angegeben werden; für Tasks mit Kommunikation wird dann eine Zeit zwischen 1 und dem angegebenen Maximalwert zufällig bestimmt.

Alternativ zur zufälligen Auswahl der Kommunikationsparameter ist auch eine Abschätzung auf Grund der im Modell angegebenen Nachrichtengröße und der Anregung aus dem Ereignisstrom möglich. Dies gilt insbesondere für einfache, periodische Ereignisse ohne Ereignisabhängigkeiten.

4.6.4 Genetische Operatoren

Bei den Genetischen Algorithmen spielen nur Änderungen an der String–Zusammensetzung eine Rolle, die die Struktur als solche nicht verändern, denn dann wäre keine Abbildung auf die Parameter mehr möglich. Somit werden keine Analogien zur Deletion und Duplikation gebildet. Desgleichen wird auch auf die Translokation verzichtet, da bei verschiedenen Strings noch nicht einmal das gleiche Alphabet verwendet werden muß — eine Übertragung von einem String auf einen anderen erscheint damit nicht sinnvoll.

4.6.4.1 Crossover

Durch die Crossover–Operation wird das Erbgut der Nachkommen aus den Eltern–Strings gebildet. Im einfachsten Fall werden an einer zufällig gewählten Schnittstelle die Strings beider Elternpaare aufgeschnitten, und die Teilstrings über Kreuz wieder zusammengesetzt. Werden auch die beiden übrigen Teilstrings zusammengesetzt, können durch diese Operation direkt die Strings zweier Nachkommen entstehen (Abbildung 4.33). Der Schnitt erfolgt dabei immer zwischen zwei Zeichen des gewählten Alphabets.

Bei nur einer Schnittstelle ist direkt ersichtlich, daß weit voneinander entfernt liegende Gene häufig getrennt werden, benachbarte Gene dagegen selten. Diese Problematik kann durch die Verwendung von mehreren Schnittstellen vermindert werden. Bei sehr vielen Trennstellen sinkt allerdings auch die Wahrscheinlichkeit, daß Gruppen zusammenpassender Gene („building blocks“) gemeinsam weitergegeben werden.

Zur Durchführung der Experimente wurde nach jedem Gen eine Schnittstelle gebildet, d. h. für jedes Gen zufällig entschieden, von welchem Elternteil es übernommen wird.

Insbesondere für Reihenfolgeprobleme (z. B. Traveling–Salesman–Problem) ist diese Art von Crossover aber ungeeignet. Hierfür wurden Varianten wie **Partially Matched**

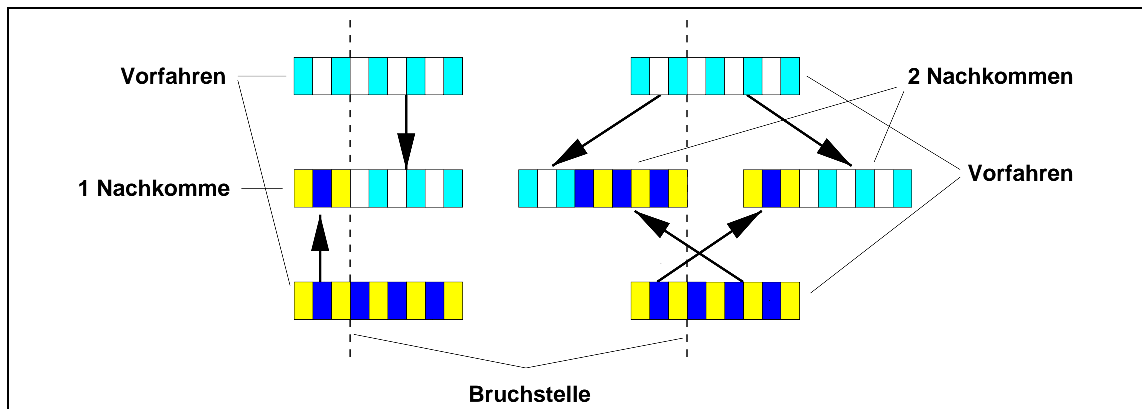


Abbildung 4.33: Einfaches Crossover

Crossover (PMX), **Order Crossover** und **Cycle Crossover** entwickelt [Gol89], die außer den Parameter–Werten auch die *Reihenfolge* der Parameter der beiden Eltern–Strings mischen.

4.6.4.2 Mutation

Durch die Mutation werden einzelne Features in *einem* String verändert, die Operation ist also unär. Mit Hilfe der Mutation können Parameterwerte in die Population eingebracht werden, die vorher noch in keinem String vorhanden waren.

Die einfache Mutation, bei der ein einzelnes Feature zufällig beeinflußt wird, wird durch die Änderungsfunktion aus Kapitel 4.1.3 realisiert. Lediglich die Änderung der TDMA–Parameter wird analog auf die hier den Tasks zugeordneten Parameter übertragen.

Eine komplexere Variante der Mutation ist die **Inversion**, die die Reihenfolge der Parameter beeinflußt. Hierbei kann ein Teilstring bestimmter Länge mit einem anderen Teilstring der gleichen Länge vertauscht werden („Swapping“ in [Wal94]). Eine andere Variante schneidet einen Teilstring aus und setzt ihn an der gleichen Stelle spiegelverkehrt wieder ein. Für einen sinnvollen Einsatz der Inversion ist es nötig, zusätzlich zum Parameter*wert* auch die Parameter*bedeutung* im String zu speichern, z. B. durch einen Parameterindex [Gol89]. Dadurch verändert die Inversion die Reihenfolge der Parameter; ohne den zusätzlichen Parameterindex würden lediglich wie bei der einfachen Mutation einige Parameterwerte zufällig neu belegt.

Durch diese Reihenfolge–Änderung wird die Kodierung der Parameter in Strings modifiziert, was beim Crossover die Wahrscheinlichkeit des „Zusammenhalts“ von Eigenschaften verändert; es handelt sich also um eine Meta–Optimierung. Vor der Crossover–Operation muß einer der beiden zu kombinierenden Strings in die Parameter–Reihenfolge des anderen konvertiert werden.

Der Inversionsoperator wurde in den Experimenten nicht berücksichtigt.

4.6.4.3 Fitness und Selektion

Durch die Selektion werden die Partner zur Bildung der nächsten Generation ausgewählt. Der Parameter, der zu einer höheren Auswahlwahrscheinlichkeit für ein Individuum führt, wird bei den Genetischen Algorithmen als *Fitness* bezeichnet. Die Fitness spiegelt die Qualität des Parametersatzes wieder; eine höhere Fitness bedeutet eine höhere Qualität.

Für das vorliegende Problem wird die Fitness mit der Kostenfunktion aus Kapitel 4.1.2 berechnet: Da die Kosten minimiert werden sollen, wird als Fitness f_i der Kehrwert der Kosten K_i verwendet:

$$f_i = \frac{1}{K_i}$$

Die Auswahl eines Individuums erfolgt häufig nach einem gewichteten Roulette–Verfahren [Gol89, Sch93], bei dem die Sektoren des „Glücksrades“ entsprechend der String–Fitness eingestellt werden (Abbildung 4.34). Dieses Verfahren wird auch hier eingesetzt. Die Wahrscheinlichkeit zur Auswahl eines Strings i ist dann das Verhältnis der eigenen Fitness zur Gesamtsumme der Fitnesswerte aller betrachteten Strings:

$$p_i = \frac{f_i}{\sum_j f_j}$$

4.6.5 Population und Erbgutaustausch

Die Struktur der Population bestimmt, aus welcher Menge von Individuen Partner für Crossover herausgesucht werden, und wie aus den neu gebildeten Individuen die nächste Generation zusammengesetzt wird.

Bei einer *nicht überlappenden* Population (z. B. [Gol89]) wird aus einer Generation i von n Individuen die nächste Generation $i+1$ gebildet, indem n mal mit dem Roulette–Verfahren zwei Strings der Generation i ausgewählt werden, aus denen durch Anwendung von Crossover und Mutation ein Nachfolge–String der Generation $i+1$ erzeugt wird. Die Eltern– und die Nachkommen–Generation sind also vollständig getrennt voneinander, die Populationsgröße ist konstant. Die Selektion erfolgt global aus allen Individuen.

Für die hier durchgeführten Experimente wird ein anderer Ansatz verwendet, bei dem n Eltern–Individuen in einer Generation existieren; jedes Individuum erhält von m anderen

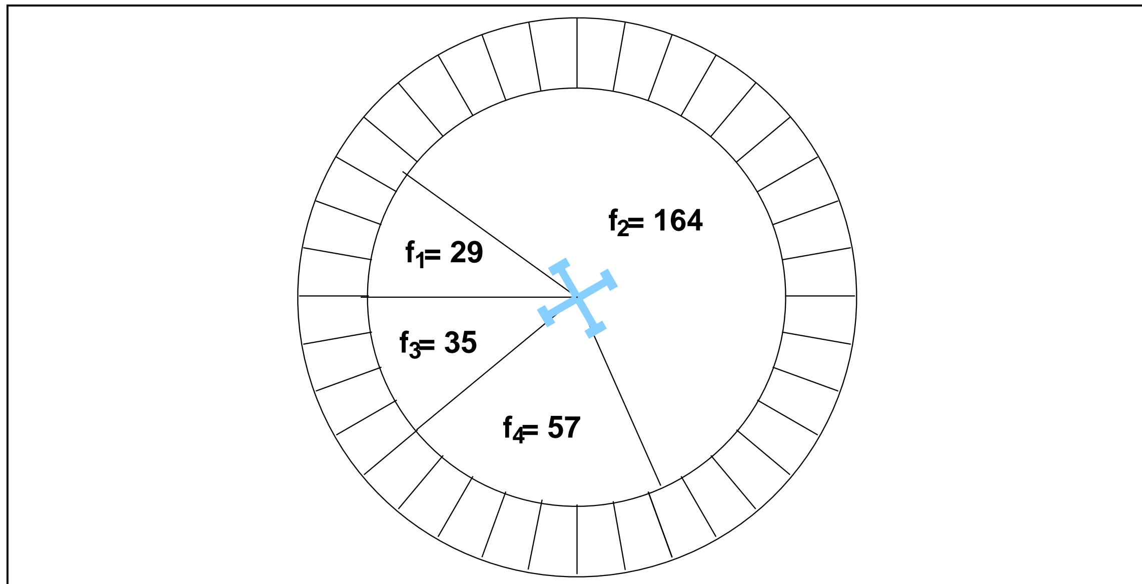


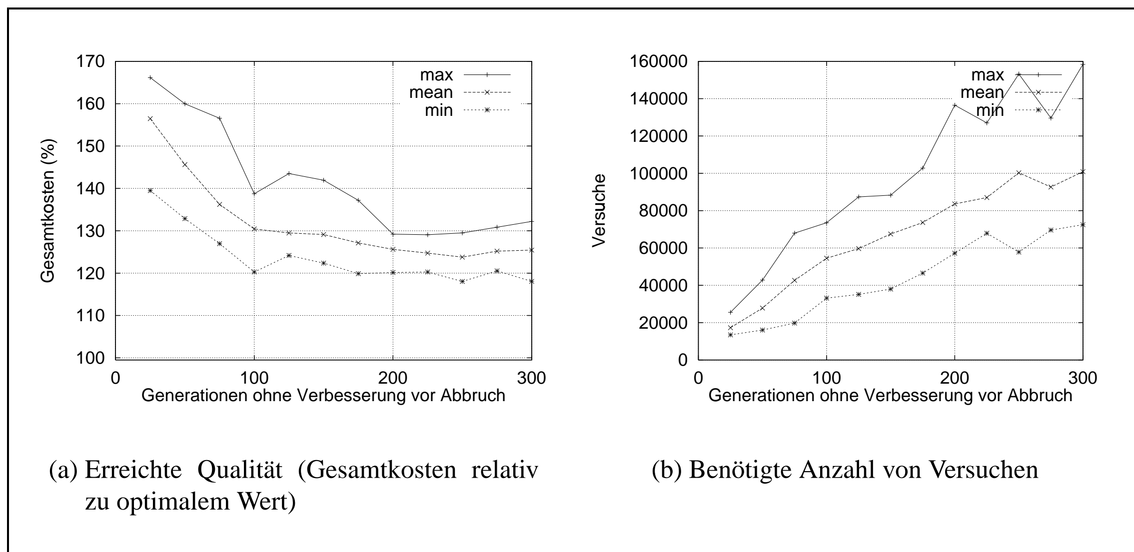
Abbildung 4.34: Selektion mit Roulette–Verfahren

Individuen deren String und bildet daraus jeweils in Kombination mit dem eigenen Erbgut m Nachfolger. Die Selektion erfolgt, ebenfalls mit dem Roulette–Verfahren, *lokal* aus dem eigenen Erbgut und dem der eigenen Nachfahren, d. h. aus $m+1$ Strings: Es wird nur ein String ausgewählt, der in der nächsten Generation das Individuum ersetzt und am Erbgutaustausch teilnimmt (vgl. Anhang A.5). Dieser Ansatz hat den Vorteil, daß bei einer Implementierung in parallele Optimierungsprozesse (z. B. bei einer vernetzten Mehrrechnerumgebung) die Kommunikation zwischen den Prozessen verringert wird. Der Nachteil ist allerdings, daß auch zwischen „Inseln“ mit durchgehend ungünstigen Bewertungen und mit durchgehend günstigen Bewertungen nicht unterschieden wird: Es werden jeweils gleich viele Nachfahren gebildet. Einen ähnlichen Ansatz mit Subpopulationen, die Individuen miteinander austauschen, wird in [TB91] vorgeschlagen.

4.6.6 Verhalten des Algorithmus

Für die genetischen Algorithmen wurde der Einfluß folgender Parameter untersucht:

- Abbruchkriterium: Anzahl der Generationen G_{break} ohne neuen Bestwert, nach denen die Optimierung beendet wird
- Populationsgröße G_{pop}
- Startpopulation
- Fitness–Funktion



(a) Erreichte Qualität (Gesamtkosten relativ zu optimalem Wert)

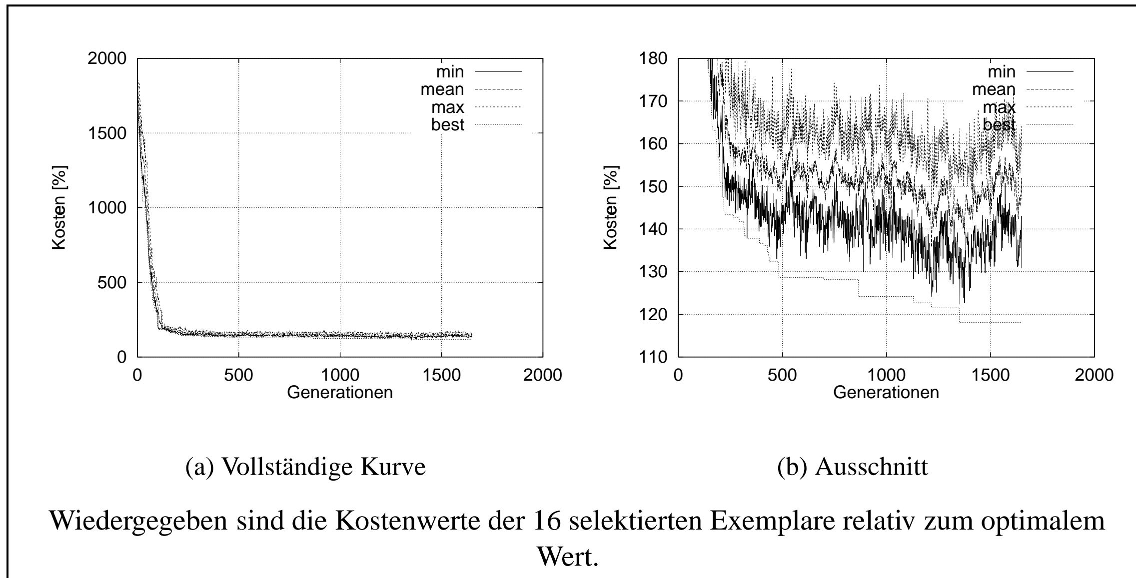
(b) Benötigte Anzahl von Versuchen

Abbildung 4.35: GA: Einfluß von G_{break} auf Modell E

Für die Variation von G_{break} wurde eine Population mit 112 Exemplaren, 16 Eltern–Exemplaren und jeweils 6 Nachkommen gewählt (ebene Kommunikation, siehe Anhang A.5.1). Die Selektion erfolgt lokal nach der Roulette–Methode; die Fitness ergibt sich aus dem Kehrwert der Kosten. Wie in den vorhergehenden Experimenten wurde von der Startposition „alle Tasks auf einem Rechner“ ausgegangen; implementationsbedingt waren damit zunächst alle Eltern–Exemplare identisch (Anhang A.5.2). Dies ist natürlich eine äußerst ungünstige Startpopulation für die Genetischen Algorithmen, da zunächst keine genetische Vielfalt bei den Exemplaren vorhanden ist. Um diesem Nachteil entgegenzuwirken, wurde mit einer Mutationsrate von 1 gearbeitet, d. h. jeder Nachkomme wurde auch der Mutation unterworfen.

Wie Abbildung 4.35(a) entnommen werden kann, verbessert sich die erzielbare Qualität mit wachsendem G_{break} zunächst relativ schnell. Bereits ab $G_{\text{break}} = 100$ tritt aber nahezu Stagnation ein, obwohl die Anzahl der Versuche weiter stark ansteigt (Abbildung 4.35(b)). Bei 96 neu erzeugten Exemplaren je Generation entsprechen dabei 10.000 Versuche ca. 104 Generationen; $G_{\text{break}} = 300$ entspricht also einem Abbruch nach 28800 Versuchen ohne Verbesserung.

In Abbildung 4.36(a) ist dazu für den besten Optimiererlauf aus Abb. 4.35 mit $G_{\text{break}} = 300$ der Kostenverlauf über die Generationen dargestellt. Wiedergegeben sind der beste bisher gefundene Kostenwert, sowie niedrigster, mittlerer und höchster Kostenwert der 16 in jeder Generation zur weiteren Fortpflanzung selektierten Exemplare. Ungefähr bis Generation 120 fallen die Kosten sehr schnell, da auch bei relativ kleinen Verbesserungen die virtuellen Kosten stark sinken; danach flacht mit Wegfall der virtuellen Rechner die Kurve ab. In dem vergrößerten Ausschnitt dieser Kurve in Abbildung 4.36(b) fällt auf, daß ein neuer Bestwert nur in seltenen Fällen auch in der nächsten Generation vertreten



(a) Vollständige Kurve

(b) Ausschnitt

Wiedergegeben sind die Kostenwerte der 16 selektierten Exemplare relativ zum optimalem Wert.

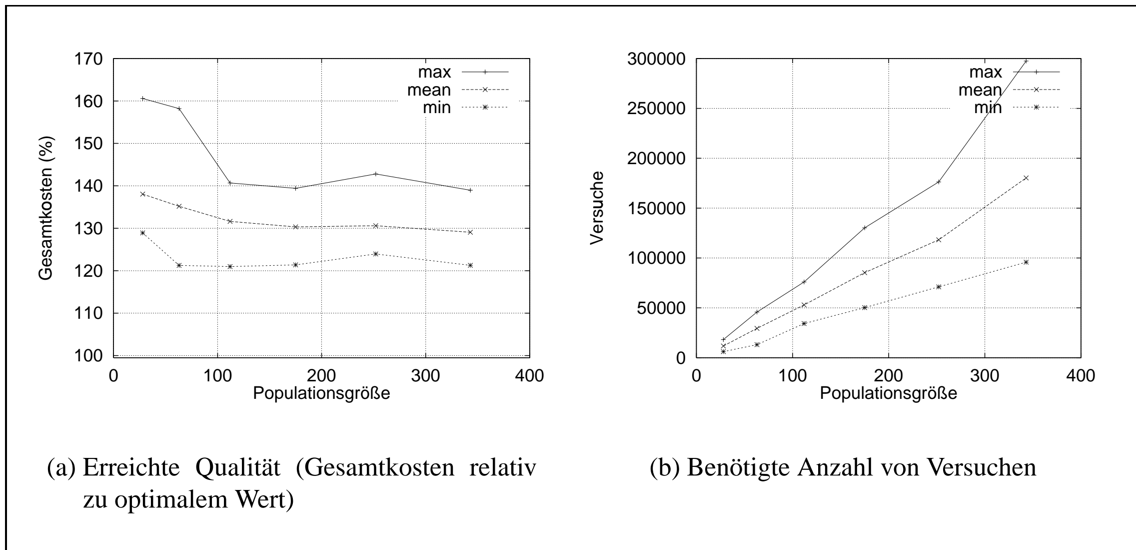
Abbildung 4.36: GA: Kostenverlauf bei $G_{\text{break}} = 300$

ist; dies beeinträchtigt natürlich die Konvergenzgeschwindigkeit der Optimierung.

Bei der Untersuchung des Einflusses der Populationsgröße wurde von dem linearen Kommunikationsmodell der Subprozesse (s. Anhang A.5.1) ausgegangen. Wegen der quadratischen Anzahl der Subprozesse wurden die Versuche für die Populationen $G_{\text{pop}} = (i^2, 6, 7i^2)$ mit $i = 2, 3, \ldots, 7$ durchgeführt. Abbildung 4.37(a) zeigt, daß sich die erreichte Qualität ab $G_{\text{pop}} = (16, 6, 112)$ kaum mehr verbessert, obwohl die Anzahl der durchgeführten Versuche stark ansteigt (Abb. 4.37(b)). Da hier $G_{\text{break}} = 100$ für alle G_{pop} konstant ist, steigt die maximale Anzahl der Versuche nach dem Auffinden des letzten Bestwertes um $6/7 * G_{\text{break}} \approx 86$ Versuche pro zusätzlichem Populationsexemplar an; allein hierdurch kann bereits eine Verbesserung des Ergebnisses erwartet werden.

Wird die Anzahl der Versuche ohne Verbesserung ungefähr konstant gehalten, z. B. ≈ 9600 bei $G_{\text{break}} = \lfloor 1600/i^2 \rfloor$ (d. h. für $G_{\text{pop}} = (16, 6, 112)$ ist wie zuvor $G_{\text{break}} = 100$), steigen die Kostenwerte mit der Populationsgröße sogar an, obwohl die Anzahl der durchgeführten Versuche nach wie vor ansteigt, wenn auch nicht so schnell wie zuvor (Abbildung 4.38). Zumindest in dem hier untersuchten Fall hat also der Parameter G_{break} einen größeren Einfluß auf die Güte als G_{pop}.

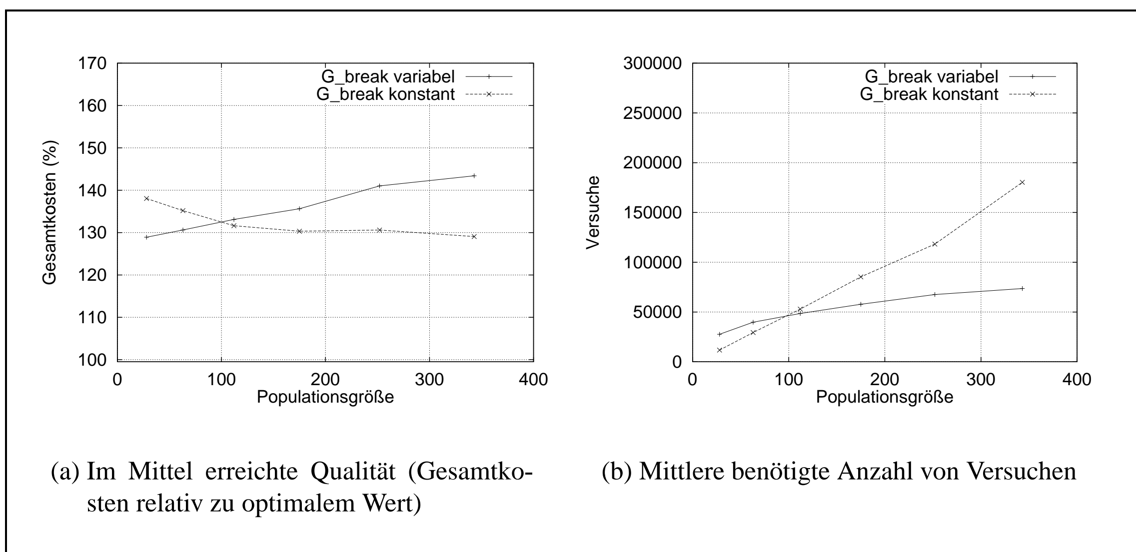
Abbildung 4.39 zeigt die Ergebnisse für weitere Parameter–Variationen, für die mit unterschiedlichen Werten von G_{break} die Optimierung jeweils 20 mal durchgeführt wurde. Zur besseren Übersichtlichkeit sind nur die Mittelwerte der Versuche abgebildet. Kurve 1 entspricht exakt dem Mittelwert aus Abbildung 4.35. Sie gibt die Werte für eine Population aus 16 Eltern–Individuen mit Kommunikation zu je 3 Nachbarn und Bildung von 2 Nachkommen je Crossover–Operation wieder; an der Selektion nehmen 112 Individuen teil.



(a) Erreichte Qualität (Gesamtkosten relativ zu optimalem Wert)

(b) Benötigte Anzahl von Versuchen

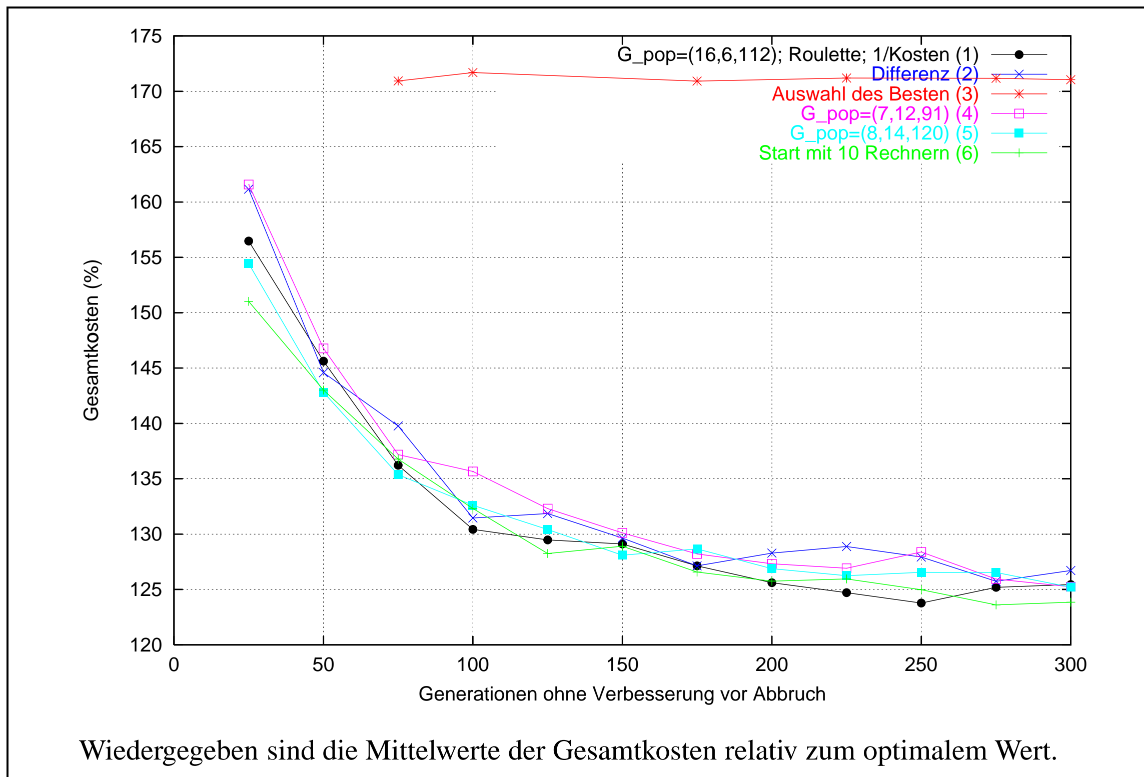
Abbildung 4.37: GA: Einfluß der Populationsgröße in Beispiel E; $G_{\text{break}} = 100$



(a) Im Mittel erreichte Qualität (Gesamtkosten relativ zu optimalem Wert)

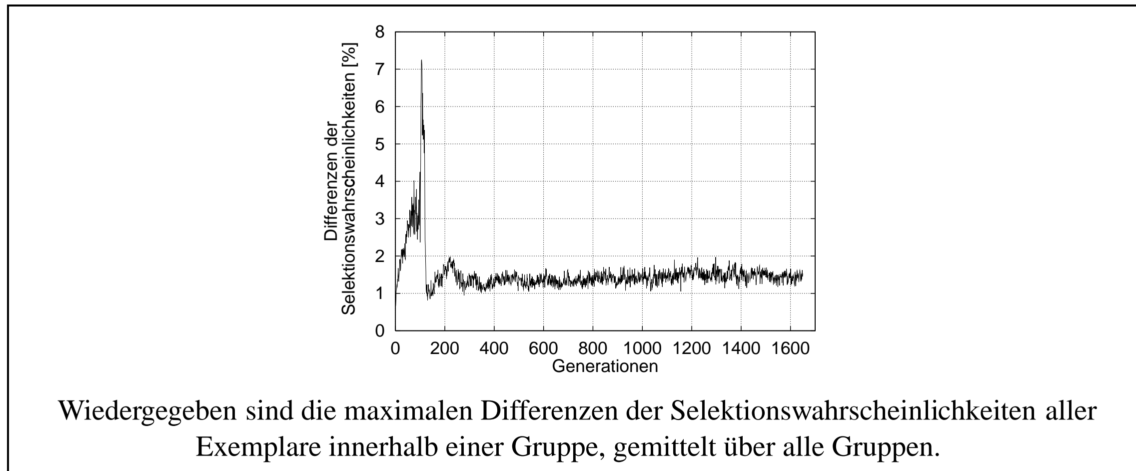
(b) Mittlere benötigte Anzahl von Versuchen

Abbildung 4.38: GA: Einfluß der Populationsgröße in Beispiel E; $G_{\text{break}} = 100$ bzw. abnehmend mit wachsender Population



Wiedergegeben sind die Mittelwerte der Gesamtkosten relativ zum optimalem Wert.

Abbildung 4.39: GA: Weitere Parameter–Varianten



Wiedergegeben sind die maximalen Differenzen der Selektionswahrscheinlichkeiten aller Exemplare innerhalb einer Gruppe, gemittelt über alle Gruppen.

Abbildung 4.40: GA: Differenzen der Selektionswahrscheinlichkeiten für Abbildung 4.36

Gegenüber dieser Optimierer–Konfiguration wurde in Kurve 2 eine Variation der Fitness–Funktion verwendet: Statt des Kehrwerts der Kosten wird die Differenz zum global schlechtesten Kostenwert (d. h. alle Tasks auf einem Rechner) verwendet (vgl. Anhang A.5.3), was aber zu keiner wesentlichen Änderung führt.

Kurve 3 zeigt eine Variation der Selektionsmethode: Es erfolgte jeweils die Auswahl der billigsten Konfiguration in einer lokalen Population statt der Auswahl nach der Roulette–Methode wie in Kurve 1 (diese Variante wurde nicht für alle Werte von G_{break} durchgeführt). Die Änderung des Auswahlverfahrens führte zu deutlich schlechteren Ergebnissen: Das Optimierungsverfahren stagnierte hier sehr früh in lokalen Optima.

Für die Kurven 4 und 5 wurde der Erbgutaustausch zwischen allen Eltern–Individuen ausgewählt, sodaß sich die lokalen Populationen gegenüber Kurve 1 von 7 auf 13 bzw. 15 Individuen vergrößerten, und die Gesamtpopulation wenig verkleinert bzw. vergrößert wurde (91 bzw. 120 Individuen). Hier ergeben sich wiederum keine wesentlichen Qualitätsunterschiede.

Schließlich gibt Kurve 6 die Werte für eine Variation des Startpunktes wieder: Hier entsteht die initiale Population durch eine zufällige Verteilung der Tasks auf 10 Rechner (gegenüber allen Tasks auf einem Rechner bei Kurve 1). Aber auch diese Änderung führte zu keinem wesentlich anderen Verhalten des Optimierers.

Neben dem oben genannten Problem, daß neue Bestwerte selten auch in der Nachkommengeneration vertreten sind, kann sich auch die gewählte Rekombinationsmethode wegen der möglichen Trennstellen des Erbguts nach jedem Gen (Zuordnung Task/Rechner) negativ auswirken, da die Erhaltung von günstigen Gruppen von Genen sehr unwahrscheinlich ist. Weiterhin führt die auf die lokalen Populationen beschränkte Selektion dazu, daß die an der Selektion beteiligten Individuen einander sehr ähnlich sind. Dies

wird in Abbildung 4.40 deutlich: Am Anfang der Optimierung haben alle Individuen eine nahezu identische Konfiguration, weshalb sich die Auswahlwahrscheinlichkeiten kaum unterscheiden. In der Folge variieren die Kostenwerte sehr stark, da durch Mutationen mehr Rechner hinzukommen und der zunächst hohe Anteil von virtuellen Kosten dadurch schnell fällt (vgl. Abbildung 4.36(a)). Danach ähneln sich die Kostenwerte stark, und die Selektionswahrscheinlichkeiten unterscheiden sich nur noch wenig.

Ein verbessertes Verhalten des Optimierungsverfahrens könnte z. B. durch folgende Maßnahmen erreicht werden:

Startwert: Die Exemplare sollten mutiert werden, bis keine identischen Exemplare mehr vorhanden sind und eine Mindest–Streuung der Kosten bzw. der Fitness erreicht ist. Die hierfür nötigen zusätzlichen Realzeitanalysen sind vernachlässigbar, da in der Folge viele Analysen identischer Konfigurationen vermieden werden. Durch die höhere Gen–Vielfalt wird von vornherein ein größerer Parameterraum durchsucht.

Fitness: Eine höhere Varianz der Fitness wird erreicht, indem die Fitness des Exemplars i als Differenz seines Kostenwertes zu dem schlechtesten Kostenwert $k_{\max}$ der Selektionsgruppe ausgedrückt wird. Um auch dem schlechtesten Exemplar s eine Fitness > 0 zuzuordnen, wird dieser Wert mit einem konstanten Faktor a_f (z. B. $a_f = 1.01$) multipliziert:

$$f_i = k_{\max} * a_f - k_i$$

Die dadurch ermittelten Fitness–Werte können allerdings nur in der Selektionsgruppe zur Bewertung herangezogen werden, da die Bezugswerte $k_{\max}$ zwischen verschiedenen Gruppen und auch in unterschiedlichen Generationen verschieden sein können.

Selektion: Die Selektion sollte aus den o.g. Gründen global über alle Exemplare einer Generation erfolgen. Um die Konvergenz zu beschleunigen, könnte eine Anzahl der auszuwählenden Exemplare nach der Fitness–Rangfolge bestimmt werden. Um dabei aber die Gen–Vielfalt nicht zu stark zu beschränken, und insbesondere um nicht zu leicht in einem lokalen Minimum zu stagnieren, ist jedoch immer noch ein Teil der Exemplare nach der Roulette–Methode auszuwählen.

Rekombination: Die Anzahl der Schnittstellen im Erbgut sollte auf eine bis wenige Stellen beschränkt sein, die zufällig ausgewählt werden. Dadurch haben günstige Zusammenstellungen von Genen die Chance, unzerstört übernommen zu werden.

5 Heuristik zur Systemauslegung

Mit der hier vorgestellten Heuristik wird versucht, durch Anwendung von einfachen Regeln und Benutzung von vorhandenem Wissen über das System schnell zu einer zwar nicht ganz optimalen, aber in der Nähe des Optimums liegenden Lösung zu gelangen.

5.1 Unabhängige Tasks

Ein sinnvoller Ausgangspunkt für die Optimierung ist sicherlich mit einem homogenen System von Rechnern des Typs mit bestem Preis–/Leistungsverhältnis gegeben, das gerade die nötige Gesamtrechenleistung zur Verfügung stellen kann.

Das Vorgehen soll zunächst für ein System von m unabhängigen Tasks, also ohne Berücksichtigung von Kommunikation oder Restriktionen, gezeigt werden.

5.1.1 Ermittlung der nötigen Gesamt–Rechenleistung

Die Worst–Case–Auslastung l eines Rechners ist durch das maximale Verhältnis $t_v(I)/I$ aus angeforderter Rechenzeit $t_v(I)$ und Intervall I in der Rechenzeitanforderungsfunktion gegeben. Die minimal nötige Gesamt–Rechenleistung P_{ges} wird also ermittelt, indem alle m Tasks *einem* Rechner beliebigen Typs zugeteilt werden, z. B. dem Rechnertyp R_{bPL} mit bestem Preis–/Leistungsverhältnis und Rechenleistung P_{bPL}. Durch die Realzeitanalyse wird dann die Worst–Case–Last l_{bPL} dieses Rechners bestimmt, und P_{ges} ergibt sich ungefähr zu

$$P_{\text{ges}} \approx l_{\text{bPL}} * P_{\text{bPL}}$$

Da hier Systeme betrachtet werden, für deren Bearbeitung mehrere Rechner nötig sind, gilt in der Regel $l_{\text{bPL}} > 1$. Abbildung 5.1 zeigt ein Beispiel für die Bestimmung von l_{bPL}: Die Worst–Case–Auslastung, hier mit l_x gekennzeichnet, ist gleich dem maximalen Verhältnis aus angeforderter Rechenzeit $t_v(I)$ und Intervallgröße I in der Rechenzeitanforderungsfunktion. Der ermittelte Wert von P_{ges} entspricht nur bei der Realzeitanalyse

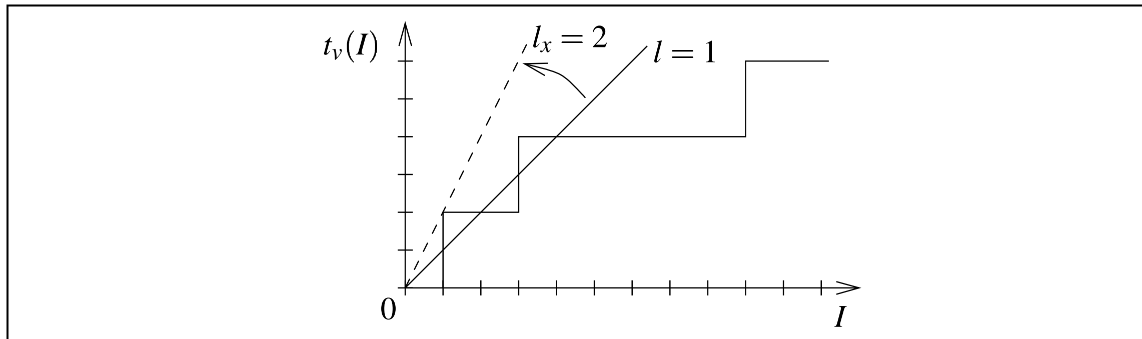


Abbildung 5.1: Bestimmung der Worst–Case–Auslastung

einfacher Tasks der genau erforderlichen Rechenleistung; sobald Elemente im Modell enthalten sind, bei denen sich angesetzte Rechenzeiten nicht umgekehrt proportional zur Rechenleistung verhalten, kann der exakte Wert nur durch iterative Analysen ermittelt werden.[1]

5.1.2 Ermittlung der nötigen Rechneranzahl

Um möglichst niedrige Gesamtkosten zu erreichen, wird zunächst ein homogenes System aus Rechnern des Typs R_{bPL} angestrebt. Für die nötige Rechneranzahl n_{bPL} gilt:

$$n_{\text{bPL}} \geq n_{\min} = \lceil l_{\text{bPL}} \rceil$$

Für den trivialen Fall $n_{\min} = 1$ wird der billigste Rechnertyp gewählt, der gerade noch genügend leistungsfähig ist[2]. Weiterhin ist n_{bPL} nach oben natürlich durch die Anzahl m der Tasks begrenzt. Ist $n_{\min} > m$, so existiert mindestens eine Task, für deren zeitgerechte Bearbeitung mehr Rechenleistung als P_{bPL} nötig ist; für solche Tasks müssen leistungsfähigere Rechner gewählt werden (siehe Abschnitt „Taskallokation“ unten), und es wird in diesem Fall $n_{\text{bPL}} = m$ gesetzt. Eine Parallelisierung des Problems in Teilprobleme, die maximal P_{bPL} benötigen, muß anwenderseitig erfolgen.

5.1.3 Ermittlung der Task–Rechenlasten

Für die Ermittlung einer günstigen Verteilung der Tasks auf die Rechner wird zunächst von jeder Task die durch sie hervorgerufene relative Rechenlast l_t (bezogen auf den Rechnertyp R_{bPL}) bestimmt. Dazu wird die zu untersuchende Task alleine auf einen Rechner

[1] Dies ist z. B. bei der Analyse von Interrupt–Service–Routinen, gegenseitigem Ausschluß und DMA–Vorgängen der Fall [Gre93].

[2] Es wird eine monoton steigende Kostenfunktion vorausgesetzt.

R_{bPL} alloziert und die Realzeitanalyse durchgeführt. Die zur zeitgerechten Bearbeitung der Task t nötige Rechenleistung P_t ergibt sich dann zu

$$P_t = l_t * P_{\text{bPL}}$$

Damit das System überhaupt realisierbar ist, muß gelten:

$$\forall t : P_t \leq P_{\max}$$

D. h. die für die zeitgerechte Bearbeitung der Task t nötige Rechenleistung darf die Rechenleistung $P_{\max}$ des leistungsstärksten Rechners nicht überschreiten; andernfalls wäre t mit den verfügbaren Rechnern nicht zeitgerecht zu bearbeiten, und das Modell müßte geändert werden.

5.1.4 Ausnutzung von Restkapazitäten

Die durch eine Gruppe von Tasks verursachte Worst–Case–Rechenlast l_g ergibt sich aus der Summe der Rechenzeitanforderungsfunktionen (RZAF) der einzelnen Tasks. Je nach Ausprägung der einzelnen RZAF's bewegt sich l_g in den Grenzen

$$\max_t(l_t) \leq l_g \leq \sum_t l_t$$

Im schlimmsten Fall entspricht die Worst–Case–Rechenlast der Taskgruppe also der Summe der einzelnen Task–Rechenlasten (z. B. bei identischen Taskparametern), im besten Fall der maximalen Einzel–Rechenlast: Eine Task läßt nach ihrem Worst–Case–Intervall dann soviel Rechenleistung ungenutzt, daß damit die Rechenzeitanforderungen aller anderen Tasks erfüllt werden können.

Das Beispiel in Abbildung 5.2 zeigt auf der linken Seite die RZAF's für drei periodische Tasks mit folgenden Parametern:

Task	Ereignisstrom	Rechenzeit	Deadline
1	$\binom{2}{0}$	1	2
2	$\binom{5}{0}$	0.5	1
3	$\binom{4}{0}$	1	4

Zusätzlich ist jeweils zum Vergleich die Winkelhalbierende eingezeichnet, d. h. die Gerade für $l = 1$.

Auf der rechten Seite sind die RZAF's für Gruppen aus diesen Tasks abgebildet: Für die Kombination von Task 1 mit Task 2 ergibt sich $l_{g,1+2} = 0.75$, da nach der Worst–Case–Rechenzeitanforderung von Task 2 bei $I = 1$ für die Anforderung von Task 1 bei

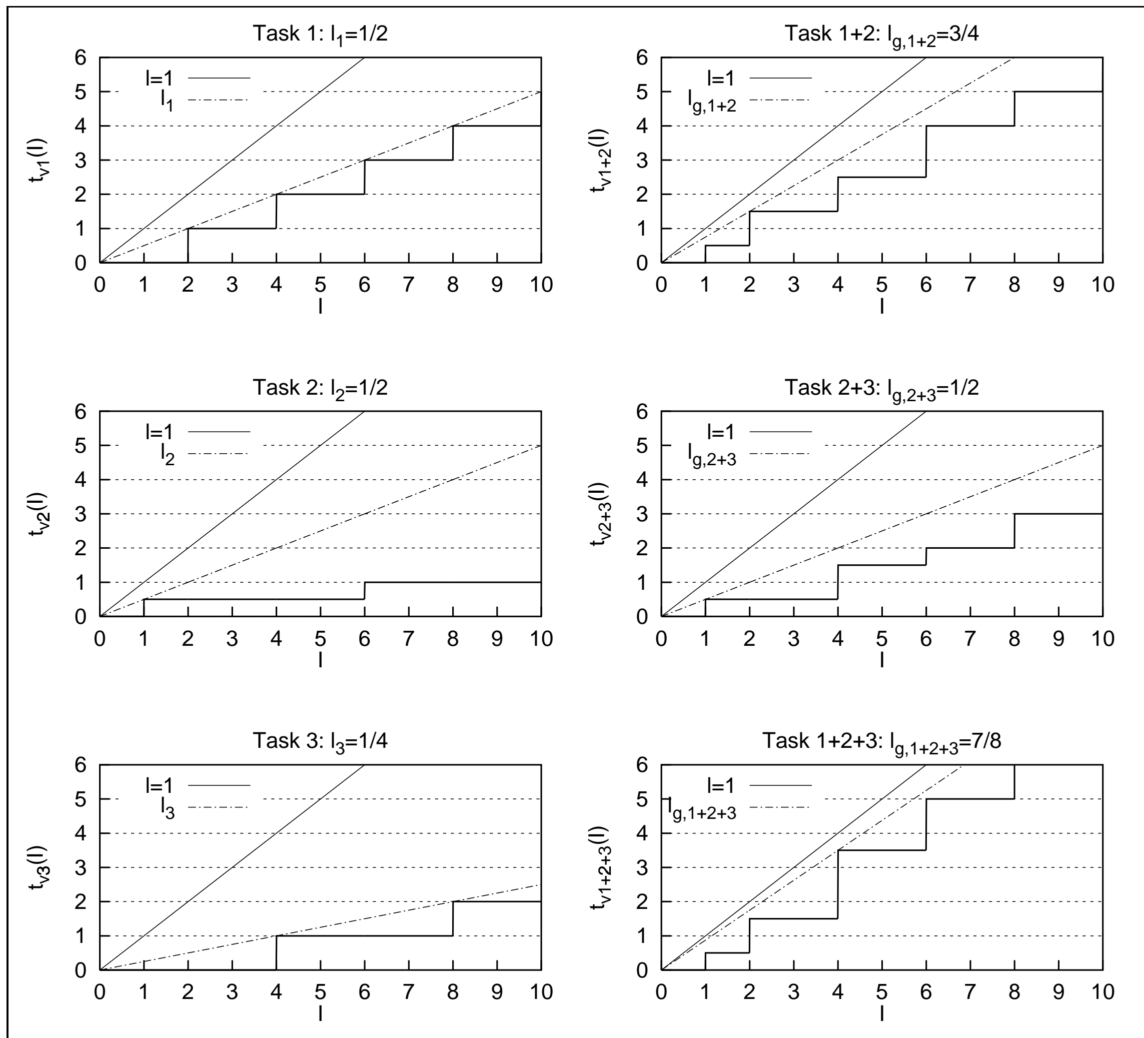


Abbildung 5.2: Beispiel für RZAF–Summen

$I = 2$ noch eine Rechenzeit $t_v = 0.5$ verfügbar ist (siehe $t_{v_2}(I)$). Die Rechenzeitanforderung von Task 3 kann sogar vollkommen mit der von Task 2 ungenutzten Rechenleistung erfüllt werden (siehe $t_{v_{2+3}}(I)$), sodaß $l_{g,2+3} = l_2 = 0.5$ ist. Die nicht abgebildete Kombination von Task 1 mit Task 3 ergäbe dagegen $l_{g,1+3} = l_1 + l_3 = 0.75$. Insgesamt ergibt sich für die Gruppe der 3 Tasks $l_{g,1+2+3} = 0.875$.

5.1.5 Taskallokation

Obige Abschätzung für l_g gilt analog (bei einem homogenen Rechnersystem aus Rechnertypen R_{bPL}) auch für die n zu bildenden Taskgruppen der n Rechner mit Worst–Case–Rechenlast l_r im Vergleich zur Allokation aller Tasks auf *einen* Rechner:

$$l_{\mathrm{bPL}} \leq \sum_{r=1}^{n} l_r$$

Für ein heterogenes Rechnersystem ist statt der Verwendung der relativen Worst–Case–Lasten die äquivalente Betrachtung der im Worst–Case tatsächlich benötigten Rechenleistung P_r günstiger:

$$P_{\mathrm{ges}} \leq \sum_{r=1}^{n} P_r$$

Um P_{ges} möglichst nahe zu kommen und somit auch möglichst wenige Rechner zu benötigen, werden die Tasks jeweils *dem* Rechner zugeteilt, bei dem sich dadurch die kleinste Erhöhung von P_r ergibt. Im einzelnen läuft die Taskallokation in folgenden Schritten ab:

1. Beginne mit einem Pool von n_{bPL} Rechnern des Typs R_{bPL}.
2. Sortiere die Tasks in der Reihenfolge absteigender P_t; bei gleichen P_t in der Reihenfolge ansteigender Intervallzeitpunkte I, bei denen P_t das erste mal in der RZAF auftritt.
3. Für alle Tasks t:
   a) Suche den Rechner r, bei dem sich die kleinste Erhöhung von P_r ergibt, P_r möglichst klein bleibt und nicht größer als P_{bPL} wird.
   b) Falls bei keinem Rechner $P_r \leq P_{\mathrm{bPL}}$ bleibt, suche den Rechner mit kleinster Erhöhung von P_r, bei dem $P_r \leq P_{\max}$ bleibt. Dieser Rechner wird später durch einen leistungsfähigeren Typ ersetzt.
   c) Falls auch jetzt noch kein Rechner gefunden wurde, nimm einen weiteren Rechner vom Typ R_{bPL} in den Rechnerpool und verwende diesen Rechner für t.

d) Alloziere t auf den gefundenen Rechner.

4. Für alle Rechner r: Wähle für r den Rechnertyp, der die benötigte Leistung P_r gerade zur Verfügung stellt.

Durch diese Strategie wird zunächst versucht, mit n_{min} Rechnern des Typs R_{bPL} auszukommen. Ist dies nicht möglich, wird zuerst für einzelne Rechner einen leistungsfähigerer Typ verwendet, bevor ein zusätzlicher Rechner hinzugenommen wird. Dadurch wird die Anzahl der Taskgruppen möglichst klein gehalten, und die Restkapazitäten an Rechnerleistung werden möglichst weit ausgenutzt.

Eine alternative Möglichkeit wäre, statt der Erhöhung der Rechenleistung in Schritt 3b direkt einen neuen Rechner vom Typ R_{bPL} in den Rechnerpool aufzunehmen. Dadurch würden zwar teurere Rechnertypen vermieden, dafür aber mehr Rechner benötigt, und somit könnten auch weniger Verdeckungseffekte genutzt werden. Welche der beiden Strategien letztlich ein preiswerteres System erzeugen, ist abhängig von dem zu optimierenden System.

5.2 Tasks mit Kommunikationsbeziehungen

Bei einem sehr leistungsfähigen Kommunikationssystem, bei dem die Zeiten für die Nachrichtenübertragung innerhalb eines Präzedenzsystems in der gleichen Größenordnung wie die Rechenzeiten liegen, können die Tasks eines Präzedenzsystems wie unabhängige Tasks im vorigen Abschnitt auf die Rechner verteilt werden.

Ist dies aber nicht der Fall, sollten Präzedenzsysteme nach Möglichkeit jeweils auf einen Rechner alloziert werden, sodaß die Kommunikation lokal bleibt. Die Taskallokation wird gegenüber dem Verfahren bei unabhängigen Tasks daher folgendermaßen erweitert:

1. Beginne mit einem Pool von n_{bPL} Rechnern des Typs R_{bPL}.

2. Ermittle für alle Präzedenzsysteme p die Worst–Case–Rechenleistung P_p bei Zuteilung des gesamten Präzedenzsystems auf einen Rechner. Für $P_p > P_{\max}$ muß eine Aufteilung in mehrere Gruppen g mit Worst–Case–Rechenleistung $P_g \leq P_{\max}$ durchgeführt werden (siehe unten), andernfalls bildet das gesamte Präzedenzsystem eine Gruppe (mit $P_g = P_p$). Unabhängige Tasks bilden ebenfalls jeweils eine Gruppe ($P_g = P_t$).

3. Sortiere die Taskgruppen in der Reihenfolge absteigender P_g.

4. Für alle Taskgruppen g: Alloziere g auf einen Rechner analog zu Schritt 3 auf der Seite vorher.

5. Wie Schritt 4.

Kann ein Präzedenzsystem auch auf dem leistungsfähigsten Rechnertyp nicht mehr zeitgerecht bearbeitet werden ($P_p > P_{\max}$), muß es in $n_p \geq \lceil P_p/P_{\max} \rceil$ Gruppen aufgeteilt werden. Während bei linearen Präzedenzsystemen die günstigsten Bruchstellen, die wenig Kommunikation über das Kommunikationssystem bei möglichst kleiner Anzahl von Gruppen ermöglichen, noch mit relativ relativ geringem Aufwand gesucht werden können, wird dies bei komplexeren Präzedenzsystemen schnell sehr aufwendig. Der Einfachheit halber werden daher für die Erzeugung einer Startkonfiguration die Tasks der Reihe nach zu einer Gruppe zusammengefaßt, und bei zu groß werdendem P_g wird eine neue Gruppe begonnen.

5.3 Einbeziehung von Restriktionen

5.3.1 Allokation auf denselben Rechner

Müssen mehrere Tasks auf denselben Rechner alloziert werden[3], werden sie zu einer Gruppe zusammengefaßt, die gleichzeitig mit den Taskgruppen aus Präzedenzsystemen und den unabhängigen Tasks alloziert wird. Eine solche Gruppe ist allerdings unteilbar, was ggf. in Schritt 2 auf der vorherigen Seite bei der Aufteilung der Präzedenzsysteme in mehrere Gruppen beachtet werden muß:

- Sind Tasks einer unteilbaren Gruppe Elemente von Präzedenzsystemen, werden diese Präzedenzsysteme zunächst mit der unteilbaren Gruppe zu *einer* Taskgruppe zusammengefaßt, sofern dadurch nicht andere Restriktionen verletzt werden (siehe unten).

- Muß eine Taskgruppe unterteilt werden, die unteilbare Gruppen enthält, werden diese in der Reihenfolge abnehmender relativer Lasten l_g herausgelöst, bis die entstandenen Taskgruppen auf Rechnern vom Typ R_{bPL} bearbeitbar sind, oder bis alle unteilbaren Taskgruppen herausgelöst wurden. Die (teilbaren) Rest–Präzedenzsysteme werden danach nötigenfalls wie im vorigen Abschnitt weiter unterteilt. Kann eine unteilbare Taskgruppe auch auf dem leistungsfähigsten Rechnertyp nicht zeitgerecht bearbeitet werden, muß das Taskmodell modifiziert werden.

[3]Dies ist beispielsweise bei Tasks der Fall, die einen gemeinsamen kritischen Bereich haben, da das Systemmodell nur rechnerlokale Semaphore vorsieht [Gre93, S. 24].

5.3.2 Allokation auf unterschiedliche Rechner

Existieren zwei oder mehrere Tasks, die auf unterschiedliche Rechner alloziert werden müssen[4], werden die Taskgruppen mit Tasks dieses Typs *vor* den bisher behandelten Taskgruppen alloziert. In Schritt 3a und 3b auf Seite 79 werden nur *die* Rechner betrachtet, auf die noch keine der zu vermeidenden Tasks alloziert wurde. Enthält ein Präzedenzsystem mehrere „unverträgliche“ Tasks, werden diese zuvor der Reihe nach aus der Taskgruppe herausgelöst, bis kein Konflikt mehr besteht.

5.3.3 Speziell erforderlicher Rechnertyp

Sind Tasks **auf einen bestimmten Rechnertyp** festgelegt[5], werden die Taskgruppen mit solchen Tasks *vor allen anderen* Taskgruppen alloziert. Dadurch werden Rechner des erforderlichen Typs frühzeitig in den Rechnerpool aufgenommen und können später auch durch andere Tasks genutzt werden. Schritt 3 auf Seite 79 ändert sich für diese Tasks folgendermaßen:

Für alle Taskgruppen g:

(a) Suche den Rechner r vom erforderlichen Rechnertyp R_T, bei dem sich die kleinste Erhöhung von P_r ergibt, P_r möglichst klein bleibt und nicht größer als P_T wird.

(b) Falls bei keinem Rechner $P_r \leq P_T$ bleibt, ersetze einen leeren Rechner vom Typ R_{bPL} durch einen Rechner vom Typ R_T, oder nimm einen weiteren Rechner vom Typ R_T in den Rechnerpool auf. Verwende diesen Rechner für g.

(c) Alloziere g auf den gefundenen Rechner und markiere den Rechner als Typ–fixiert.

Sind in einer Taskgruppe Tasks enthalten, die *unterschiedliche* Rechnertypen erfordern, müssen diese Gruppen zuvor durch sukzessives Herauslösen von auf den gleichen Typ festgelegten Tasks aufgespalten werden. Auch Taskgruppen, die auf dem erforderlichen Typ nicht zeitgerecht bearbeitbar sind, müssen in mehrere Gruppen aufgeteilt werden.

5.4 Algorithmus

Zusammengefaßt ergibt sich damit der folgende algorithmische Ablauf:

1. Bildung von Taskgruppen aus Präzedenzsystemen
2. Aufspaltung von Taskgruppen bei Fixer–Typ–Restriktionen

[4] Z.B. redundante Replikate fehlertoleranter Tasks
[5] Z.B. wegen bestimmter Hardware–Voraussetzungen

3. Aufspaltung von Taskgruppen bei Verschiedene–Rechner–Restriktionen

4. (Um–)Bildung von Taskgruppen bei Selber–Rechner–Restriktionen

5. Herauslösung kommunikations– und restriktionsloser Tasks aus Taskgruppen:

   Durch die Aufspaltung und Umbildung von Taskgruppen in den vorhergehenden Schritten können bei komplexeren Präzedenzsystemen einzelne Tasks in einer Gruppe verbleiben, die keine (direkte) Kommunikationsverbindung mehr zu den übrigen Tasks der Gruppe haben. Vorteilhafterweise werden diese Tasks abgetrennt und wie unabhängige Tasks alloziert.

6. Bestimmung der Worst–Case–Rechenleistungen P_g für alle Taskgruppen g

7. Aufspaltung von Taskgruppen bei $P_g > P_{\max}$ bzw. $P_g > P_T$ bei auf den Rechnertyp T fixierten Taskgruppen

8. Allokation in der Reihenfolge

   a) Taskgruppen mit Fixer–Typ–Restriktion

   b) Taskgruppen mit Verschiedene–Rechner–Restriktion

   c) restliche Taskgruppen

   Innerhalb dieser Gruppen werden die Taskgruppen in der Reihenfolge fallender P_g alloziert. Die Allokation erfolgt analog zu Schritt 3 auf Seite 79 unter Beachtung der bei Restriktionen nötigen Änderungen.

Abbildung 5.3 zeigt ein Beispiel, in dem alle Restriktionen enthalten sind: Task 2, 4, 5 und 8 müssen auf denselben Rechner alloziert werden, da sie in einem gemeinsamen kritischen Bereich liegen (Sema1), ebenso Task 3 und 7 (Sema2). Weiterhin benötigt Task 4 einen Rechner vom Typ F1, Task 6 vom Typ F2. Task 9a und 9b sind Replikate einer fehlertoleranten Task und dürfen daher nicht auf den selben Rechner alloziert werden.

Die in den Schritten 1 bis 5 gebildeten Taskgruppen sind in Tabelle 5.1 auf Seite 85 wiedergegeben. In Schritt 1 wird zunächst aus der unabhängigen Task T1 und den beiden Präzedenzsystemen jeweils eine Taskgruppe gebildet. Im nächsten Schritt muß Gruppe g_3 aufgespalten werden, da T4 und T6 unterschiedliche Rechner benötigen. Es sei $P_{\mathrm{T4}} > P_{\mathrm{T6}}$, sodaß T4 herausgelöst wird. In Schritt 3 müssen T9a und T9b getrennt werden, sodaß T9b nun eine eigene Gruppe (g_5) bildet. Im darauffolgenden Schritt müssen Tasks, die einen gemeinsamen kritischen Abschnitt haben, zu einer Gruppe zusammengefaßt werden. Um möglichst wenig rechnerübergreifende Kommunikation zu erzeugen, wird hier versucht, die beiden Präzedenzsysteme komplett zusammenzufassen. Dies scheitert allerdings an dem Konflikt zwischen T4 und T6, sodaß T6 alleine in g_3 verbleibt. Schritt 5 wird in diesem Beispiel nicht benötigt. Die in diesem konstruierten

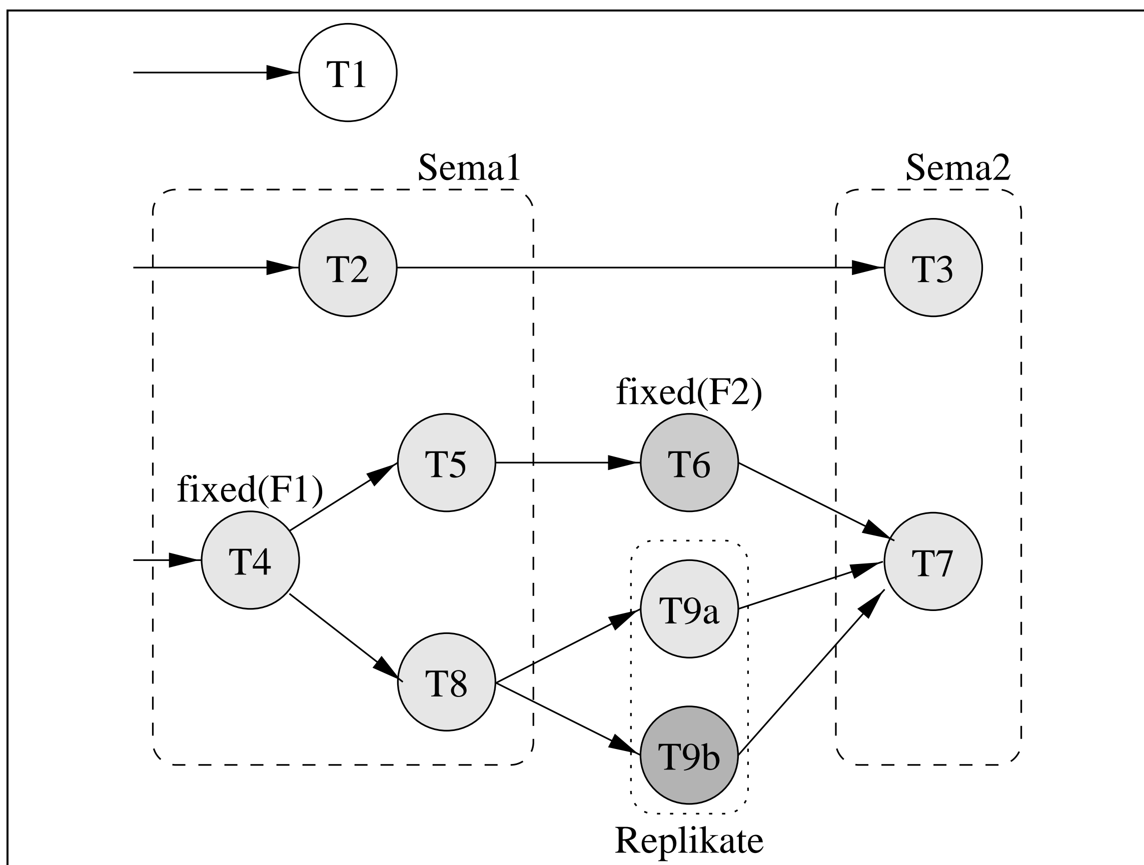


Abbildung 5.3: Beispiel für Startwertsuche. Die Schattierungen entsprechen den Taskgruppen in Tab. 5.1 auf der nächsten Seite.

Schritt	Taskgruppen				
	g_1	g_2	g_3	g_4	g_5
1	T1	T2,3	T4 ... T9b	–	–
2	T1	T2,3	T5 ... T9b (F2)	T4 (F1)	–
3	T1	T2,3	T5 ... T9a (F2)	T4 (F1)	T9b
4	T1	T2,4,5,8 (F1) T3,7 T9a	T6 (F2)	–	T9b

Tabelle 5.1: Gruppenbildung für das Beispiel aus Abbildung 5.3

Beispiel relativ groß gewordene Gruppe g_2 müßte ggf. in Schritt 7 wieder aufgeteilt werden. Unter der Annahme eines ausreichend leistungsfähigen Rechnertyps F1 werden die Taskgruppen in der Reihenfolge g_2 und g_3 (Typ–fixiert), g_5 (Replikat) und zuletzt g_1 alloziert.

5.5 Aufwandsabschätzung

Der initiale Mehraufwand an Realzeitanalysen für die Suche eines günstigen Startpunktes beträgt für ein System mit m restriktionslosen Taskgruppen (die nicht aufgeteilt werden müssen) und $n = n_{\text{bPL}}$ benötigten Rechnern

- eine Analyse zur Bestimmung von P_{ges}
- m Analysen zur Bestimmung der Worst–Case–Rechenlasten P_g der Taskgruppen
- $\approx m * n$ Analysen für die Plazierung der Taskgruppen[6]

Insgesamt sind also ca. $m * (n+1)$ Analysen erforderlich. Unter der Annahme, daß die Rechneranzahl proportional zur Zahl der Taskgruppen steigt, wächst die Zahl der Analysen mit $O(n^2)$.

5.6 Anwendungsergebnisse

Für das sehr einfache Modell E ist die mit Hilfe der Heuristik ermittelte Lösung leicht nachzuvollziehen. Die insgesamt benötigte Rechenleistung beträgt für dieses Beispiel 310 MIPS (s. Anh. B.2 auf Seite 108); aufgrund von Rundungen in der Analyse zu sicheren Seite hin wird ein knapp darüber liegender Wert errechnet. Somit geht die Heuristik

[6] Solange leere Rechner im System sind, werden für die Plazierung einer Taskgruppe statt m Analysen entsprechend weniger benötigt.

von einer minimal nötigen Anzahl von 32 Rechnern des preisgünstigsten Typs C10 aus. Nach Allokation von 96 Tasks muß für die letzten 4 Tasks zusätzliche Rechenleistung bereitgestellt werden; als Resultat ergibt sich ein System aus 28 Rechnern des Typs C10 mit je 3 Tasks (Auslastung 93%) und 4 Rechnern des Typs C14 (Auslastung 89%).

Diese Lösung liegt bezüglich der Gesamtkosten entsprechend der Kostenfunktion aus Kapitel 4.1.2.4 um 9,2% und bezüglich der reinen Hardwarekosten um 6,1% über dem Optimum. Zur Ermittlung der Lösung werden insgesamt 2042 Realzeit–Analysen durchgeführt[7]. Die Verwendung dieser suboptimalen Lösung als Startwert für eine weitere heuristische Optimierung wird im nächsten Kapitel diskutiert.

Bei dem komplexeren Modell R ermittelt die Heuristik ein System mit 14 Rechnern (8 mal C9, 2 mal C10 und 4 mal C18), auf das die in 64 Gruppen zusammengefaßten Tasks verteilt sind. Die Gesamtkosten überschreiten mit 3772,12 den minimalen bei den stochastischen Optimierungen gefundenen Wert um 13,5%, die Hardwarekosten um 15,9%. Für das Ergebnis sind 736 Analysen nötig[8].

[7] Benötigte Rechenzeit: 9,6s auf DEC Alphastation 500/400

[8] Benötigte Rechenzeit: 9,1s auf DEC Alphastation 500/400

6 Diskussion der Ergebnisse

6.1 Gegenüberstellung der Verfahren

In diesem Kapitel werden die Ergebnisse der einzelnen Optimierungsverfahren aus Kapitel 4 und der Heuristik aus Kapitel 5 zusammengefaßt dargestellt.

Teilweise werden zum Vergleich Ergebnisse zweier sehr einfacher stochastischer Verfahren verwendet: „Monte–Carlo“ und „Hill–Climbing“. Beide Verfahren nutzen die in Kapitel 4.1 beschriebenen Funktionen für die Generierung des Startwertes, für die Bewertung und für die Erzeugung einer geänderten Konstellation. Der Unterschied besteht nur in der Annahmebedingung einer geänderten Konstellation: bei „Monte–Carlo“ wird eine geänderte Konstellation *immer* als Ausgangspunkt für die nächste Iteration verwendet, und bei „Hill–Climbing“ werden *nur Verbesserungen* akzeptiert. Die ausführlichen Algorithmen sind im Anhang A.6 angegeben.

Beide Verfahren werden so parametrisiert, daß sie ungefähr gleich viele Versuche wie die anderen Optimierungsverfahren durchführen. Durch Vergleich mit „Monte–Carlo“ kann dann festgestellt werden, wie „zielgerichtet“ ein Optimierungsverfahren arbeitet; der Vergleich mit „Hill–Climbing“ zeigt, welche Auswirkungen die Fähigkeit zum Verlassen lokaler Minima hat.

6.1.1 Ergebnisse aus Beispiel E

Von den vorhergehenden Versuchen wurde für jedes Verfahren ein „guter“ Parametersatz ausgewählt (siehe Tabelle 6.1). Abbildung 6.1 zeigt für Beispiel E die niedrigsten (d. h. besten) und die durchschnittlich erreichten Kostenwerte (relativ zum optimalen Wert) innerhalb der durchgeführten 20 Versuche.

Bei Simulated Annealing und den davon abgeleiteten Verfahren wurde innerhalb der 20 Versuche das Optimum gefunden. Während bei Simulated Annealing und Threshold Accepting der Mittelwert ebenfalls fast das Optimum erreicht, also nahezu in jedem Durchlauf das Optimum gefunden wurde, werden bei Great Deluge (GD) und Record–

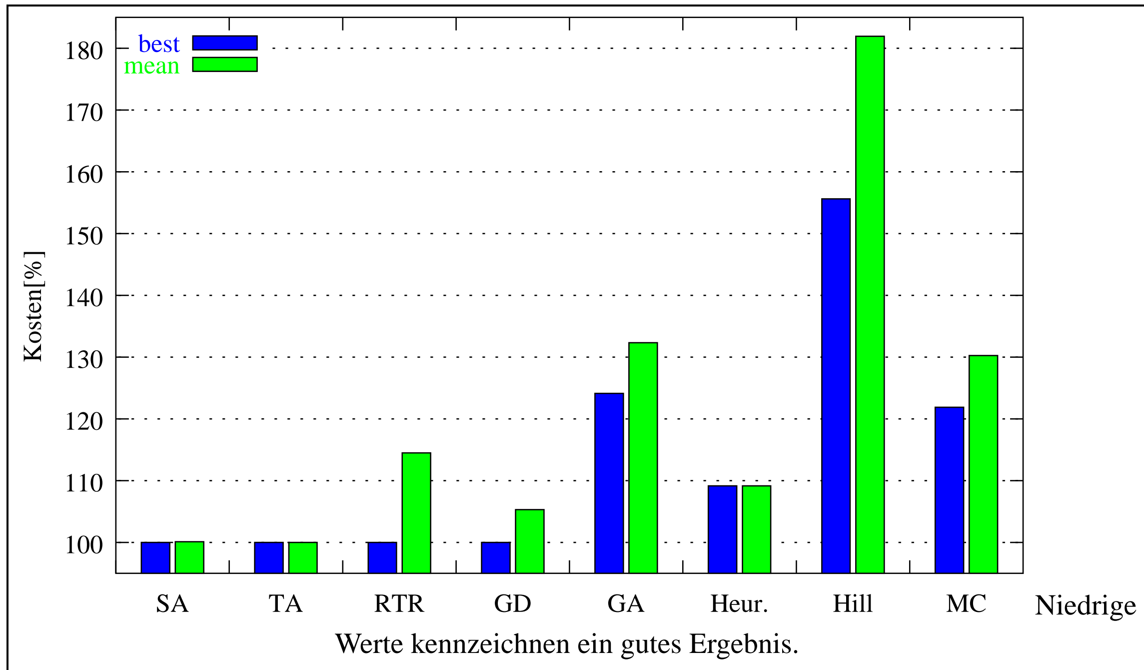


Abbildung 6.1: Kostenvergleich für Beispiel E

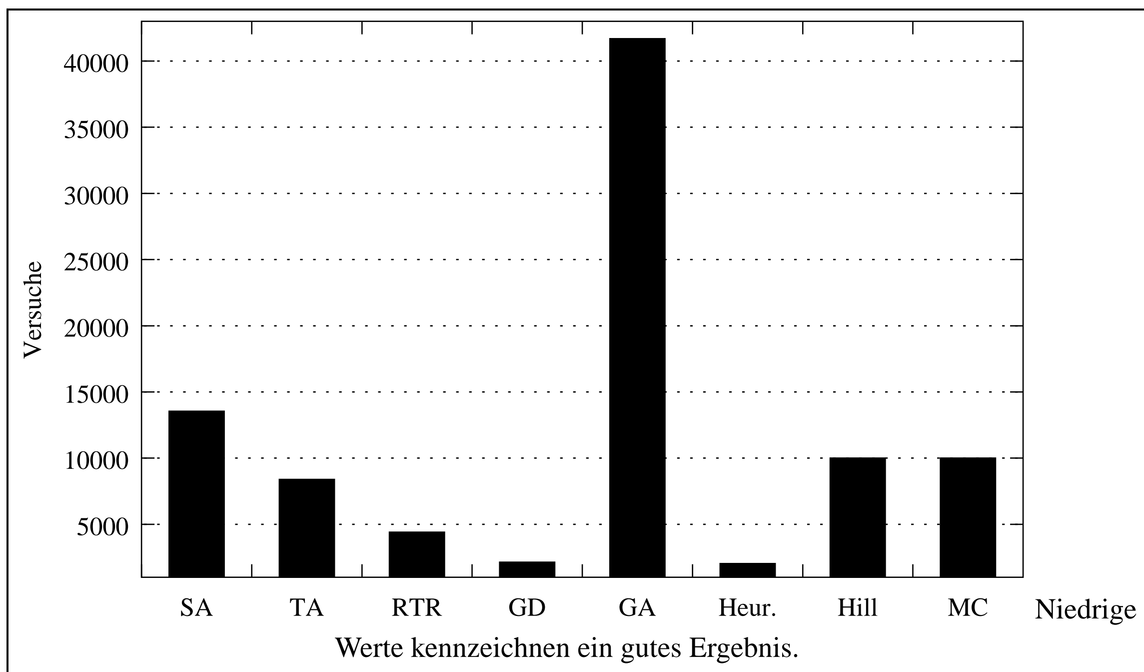


Abbildung 6.2: Aufwandsvergleich für Beispiel E

Verfahren	Parameter	Best-wert	Mittel-wert	Versuche
Simulated Annealing	$\alpha = 0.9$; $T_0 = 15000$; $c_{\text{temp}} = 50$; $t_{\max} = 1000$	100%	100.1%	13540
Threshold Accepting	$\alpha = 0.9$; $T_0 = 15000$; $c_{\text{temp}} = 50$; $t_{\max} = 1000$	100%	100%	8399
Record–To–Record–Traveling	$D = 100$; $t_{\text{s,max}} = 1000$	100%	114.5%	4405
Great Deluge	$D = 150$; $t_{\text{s,max}} = 1000$	100%	105.3%	2137
Genetische Algorithmen	$G_{\text{break}} = 100$	124.1%	132.3%	41693
Heuristik	—	109%	109%	2042
Hill Climbing	$t_{\text{s,max}} = 10000$	155.6%	181.9%	10000
Monte Carlo	$t_{\text{s,max}} = 10000$	121.9%	130.3%	10000

Tabelle 6.1: Ergebnisvergleich für Beispiel E

To–Record–Traveling (RTR) einige Durchläufe in lokalen Optima beendet. Dabei muß beachtet werden, daß bei RTR nur für den Parameter $D = 100$ das Optimum überhaupt gefunden wurde, wodurch sich auch der deutlich schlechtere Mittelwert ergibt (siehe dazu auch Abbildung 4.21(a) auf Seite 51).

Wie in Abbildung 6.2 auf der vorherigen Seite zu sehen ist, korrespondiert das gute Optimierungsergebnis von Simulated Annealing und Threshold Accepting auch mit einer erhöhten Anzahl von Versuchen, wobei Threshold Accepting bei gleicher Qualität deutlich weniger Versuche benötigt. Hervorstechend ist die trotz des brauchbaren Ergebnisses sehr niedrige Zahl von Versuchen bei Great Deluge.

Die Genetischen Algorithmen schneiden im Vergleich dazu deutlich schlechter ab und werden sogar von dem Monte–Carlo–Verfahren geschlagen (Abb. 6.1). Wie in Kapitel 4.6 schon dargelegt wurde, weisen die Genetischen Algorithmen wesentlich mehr Freiheitsgrade als Simulated Annealing und verwandte Verfahren auf und sind daher schwieriger an das Problem anzupassen. Möglicherweise könnte hier das Ergebnis durch eine andere Adaptierung der GA verbessert werden. Immerhin kann durch die leichte Parallelisierbarkeit der Zeitaufwand für die hohe Zahl von Versuchen durch entsprechenden Rechnereinsatz stark verringert werden. Eine zielgerichtete Optimierung ist allerdings bei der implementierten Variante nicht erkennbar.

Dagegen wurde mit der in Kapitel 5 vorgeschlagenen Heuristik ein recht guter, mit den Mittelwerten von RTR und GD vergleichbarer Wert erreicht (Abbildung 6.1). Die im Vergleich bereits sehr kleine Anzahl von nötigen Versuchen in Abbildung 6.2 darf nicht in vollem Umfang auf die benötigte Zeitdauer übertragen werden: Anfangs ist das in der Heuristik analysierte Modell noch nicht vollständig, sodaß die Realzeitanalysen wesentlich schneller durchgeführt werden als bei den vollständigen Analysen für die Optimie-

Verfahren	Parameter	Best- wert	Mittel- wert	Versuche
Simulated Annealing	$\alpha = 0.9$; $T_0 = 3000$; $c_{\text{temp}} = 50$; $t_{\max} = 1000$	100.3%	101.8%	17113
Threshold Accepting	$\alpha = 0.9$; $T_0 = 3000$; $c_{\text{temp}} = 50$; $t_{\max} = 1000$	100.8%	103.8%	11299
Record–To–Record–Traveling	$D = 50$; $t_{s,\max} = 1000$	100.4%	103.4%	2758
Great Deluge	$D = 1000$; $t_{s,\max} = 1000$	102.7%	107.4%	3087
Heuristik	—	113.5%	113.5%	736
Hill Climbing	$t_{s,\max} = 10000$	101.2%	103.4%	10000
Monte Carlo	$t_{s,\max} = 10000$	129.3%	141.4%	10000

Tabelle 6.2: Ergebnisvergleich für Beispiel R

rungsverfahren.

Das zum Vergleich verwendete Hill–Climbing–Verfahren erreicht die Werte der stochastischen Verfahren in keinem Durchlauf (Abb. 6.1): Das Beispiel enthält viele ausgeprägte lokale Minima, in denen dieses zielgerichtete Verfahren stockt. Auch gegenüber dem hier besseren Monte–Carlo–Verfahren führen die stochastischen Optimierungsverfahren — mit Ausnahme der Genetischen Algorithmen — schneller zu besseren Ergebnissen.

Bei diesem sehr einfachen Beispiel zeigen unter Berücksichtigung von erreichter Qualität und benötigtem Aufwand Great Deluge und Threshold Accepting die besten Ergebnisse. Great Deluge zeichnet sich dabei durch die wenigen Parameter, und damit eine sehr einfache Anwendung, sowie durch die Schnelligkeit aus. Die Stärken von Threshold Accepting liegen hauptsächlich in der Güte der Optimierung und der unkritischen Einstellung der Parameter. Für eine sinnvolle Anwendung der Genetischen Algorithmen müßten weitere Untersuchungen zur besseren Anpassung des Verfahrens an die Problemstellung durchgeführt werden.

6.1.2 Ergebnisse mit Beispiel R

Die exakten Daten im Vergleich für dieses Beispiel sind in Tabelle 6.2 zusammengefaßt. Das einfache Hill–Climbing–Verfahren liefert hier sehr gute Ergebnisse, da das Beispiel offenbar keine ausgeprägten lokalen Optima besitzt (Abbildung 6.3). Simulated Annealing und Threshold Accepting sind nur knapp besser als Hill Climbing, wobei Simulated Annealing allerdings deutlich mehr Versuche benötigt (Abbildung 6.4). Threshold Accepting ist im Mittel etwas schlechter, verhält sich aber auch bei diesem Beispiel unkritisch bezüglich Parameter–Änderungen (vgl. Kapitel 4.3.2). Mit $D = 50$ verhält sich

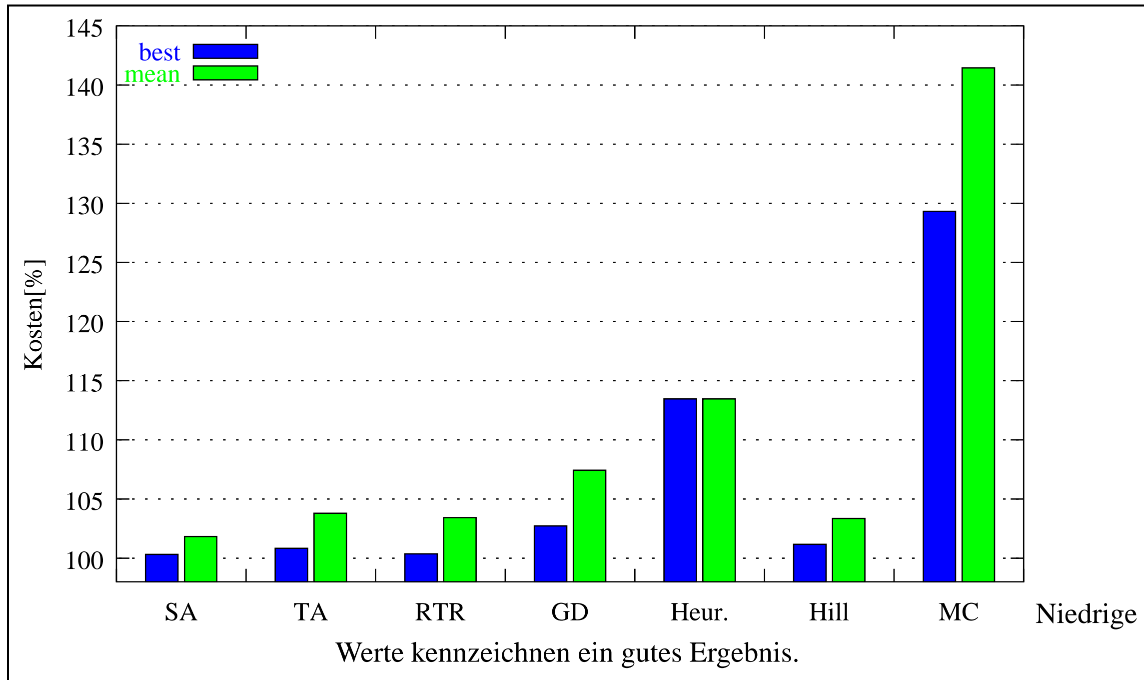


Abbildung 6.3: Kostenvergleich für Beispiel R

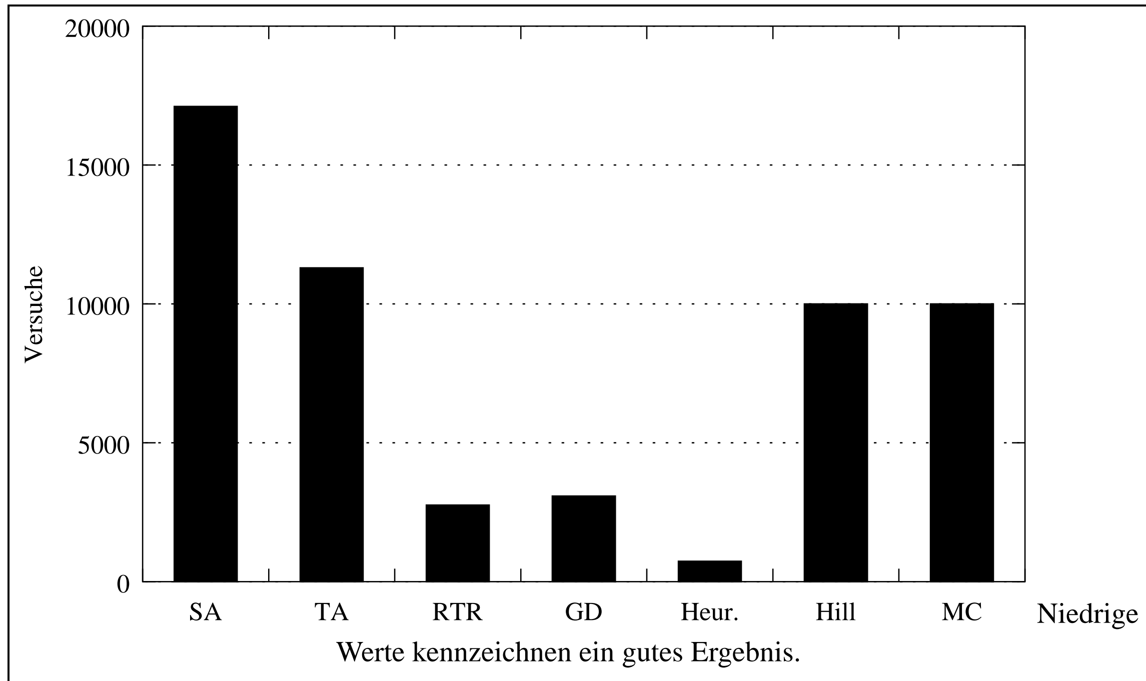
Record–To–Record–Traveling fast schon wie Hill Climbing, da nahezu nur noch Verbesserungen akzeptiert werden — entsprechend ähnlich sind die Ergebnisse (Abb. 6.3). Die Anwendung der Heuristik liefert zwar ein etwas schlechteres Ergebnis als für Beispiel E, benötigt dafür aber mit Abstand die wenigsten Versuche.

Im Vergleich mit dem Monte–Carlo–Verfahren zeigt sich wieder, daß eine rein zufällige Suche nicht den gewünschten Erfolg zeigt, und daß die zielgerichteten Komponenten der stochastischen Optimierungsverfahren wirken. Für die Genetischen Algorithmen kann bei diesem Beispiel keine Aussage getroffen werden; aufgrund der schlechten Ergebnisse mit Beispiel E wurde aus Aufwandsgründen auf weitere Experimente verzichtet.

6.1.3 Heuristikergebnis als Startpunkt

Für Beispiel E wurde weiterhin das Verhalten der stochastischen Optimierer untersucht, wenn das Ergebnis der Heuristik als Startpunkt verwendet wird. Für die Genetischen Algorithmen macht prinzipbedingt eine Startpunktvorgabe keinen Sinn, sie sind daher nicht berücksichtigt.

Die Parameter–Einstellungen wurden identisch zu Kapitel 6.1.1 verwendet. Die genauen Ergebnisse sind in Tabelle 6.3 dargestellt. Im Vergleich zu den Ergebnissen mit dem sehr schlechten Startpunkt „alle Tasks auf einem Rechner“ fällt auf, daß die zufällige



Niedrige Werte kennzeichnen ein gutes Ergebnis.

Abbildung 6.4: Aufwandsvergleich für Beispiel R

Verfahren	Parameter	Best-wert	Mittel-wert	Versuche
Simulated Annealing	$\alpha = 0.9$; $T_0 = 15000$; $c_{temp} = 50$; $t_{max} = 1000$	100%	100.1%	15533
Threshold Accepting	$\alpha = 0.9$; $T_0 = 15000$; $c_{temp} = 50$; $t_{max} = 1000$	100%	100.1%	6349
Record–To–Record–Traveling	$D = 100$; $t_{s,max} = 1000$	100%	100.7%	1470
Great Deluge	$D = 150$; $t_{s,max} = 1000$	100%	101.5%	1600
Monte Carlo	$t_{s,max} = 10000$	109.2%	109.2%	10000

Tabelle 6.3: Ergebnisvergleich für Beispiel E, mit Heuristik–Ergebnis als Startpunkt

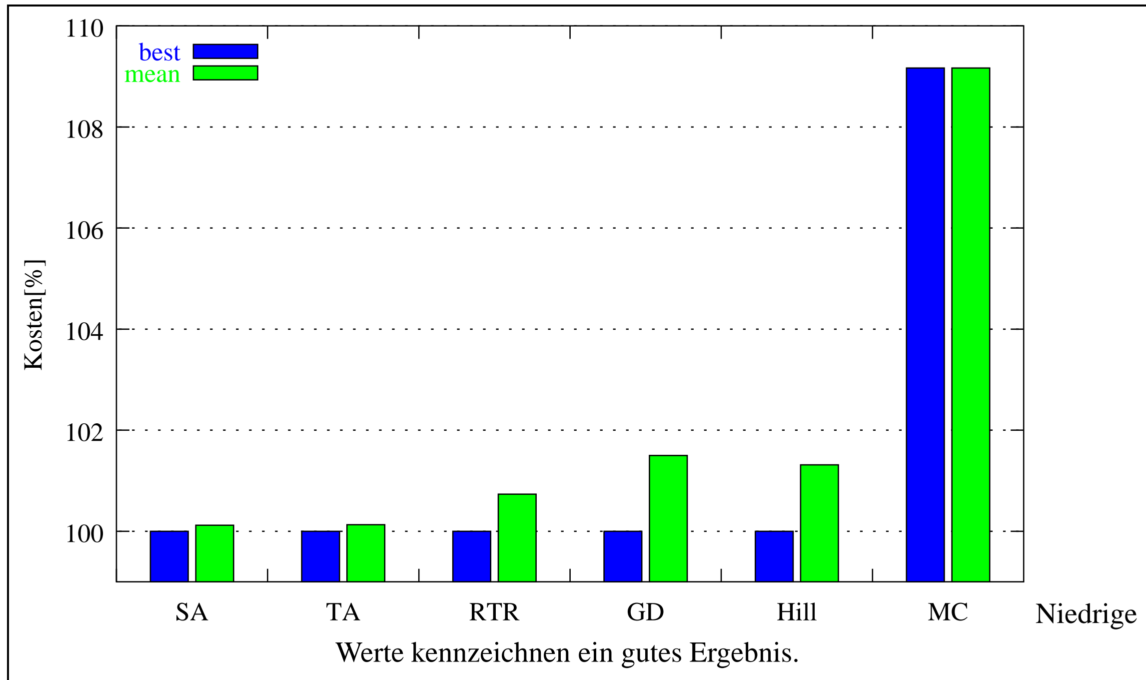


Abbildung 6.5: Kostenvergleich für Beispiel E, Heuristik–Ergebnis als Startpunkt

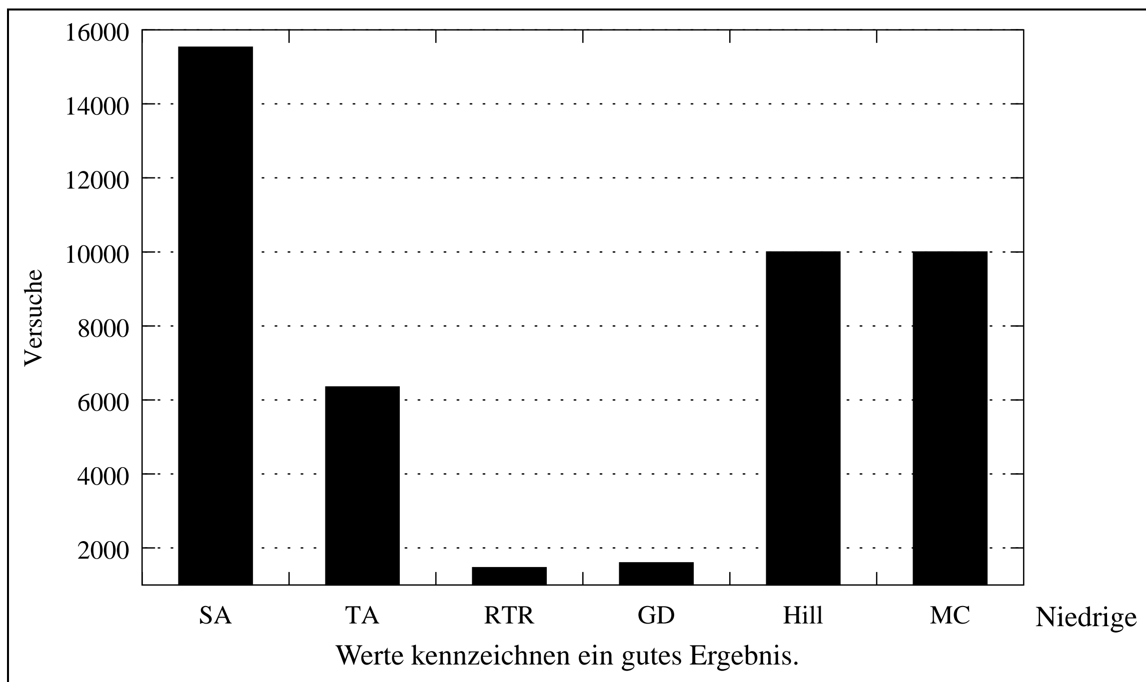
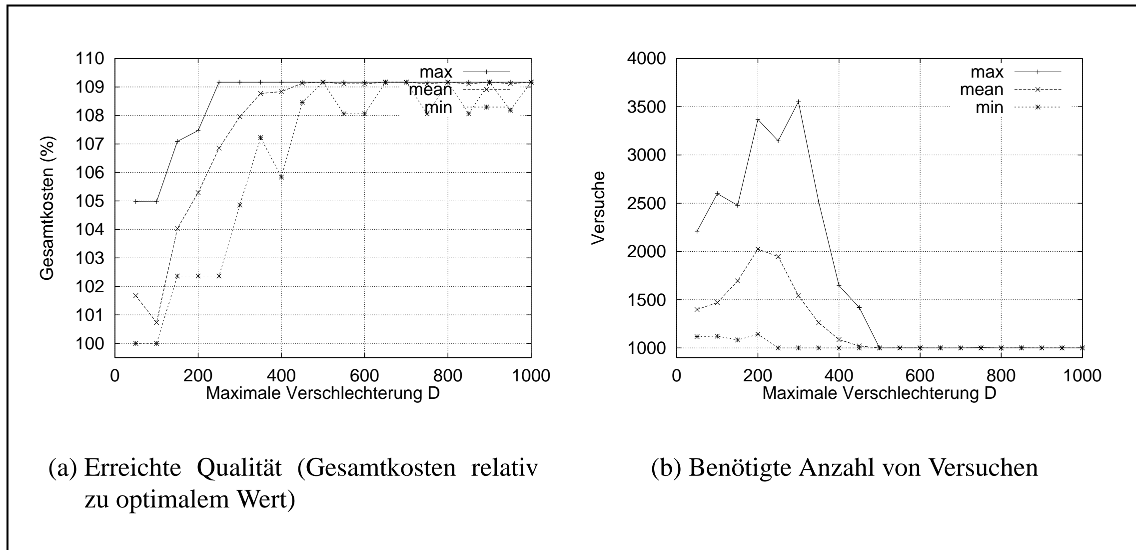


Abbildung 6.6: Aufwandsvergleich für Beispiel E, Heuristik–Ergebnis als Startpunkt

Suche mit Monte Carlo keine Verbesserung gegenüber dem Heuristik–Wert mehr erzielt (Abbildung 6.5).



(a) Erreichte Qualität (Gesamtkosten relativ zu optimalem Wert)

(b) Benötigte Anzahl von Versuchen

Abbildung 6.7: RTR: Einfluß von Abweichung D in Beispiel E mit Startpunkt aus Heuristik

Auch Simulated Annealing kann keinen Gewinn aus dem „guten“ Startpunkt ziehen: Das Ergebnis war ohnehin nicht mehr zu verbessern, und die Anzahl der nötigen Versuche steigt sogar in diesem Fall noch an. Für Threshold Accepting wird dagegen eine Verringerung der Versuchszahl erreicht (Abbildung 6.6). Beide Verfahren würden hier bessere Ergebnisse liefern, wenn die Anfangstemperatur T_0 entsprechend dem verbesserten Startpunkt verringert würde. Durch die unveränderte Einstellung wird die zielgerichtete Komponente zu spät aktiviert.

Für Record–To–Record–Traveling und Great Deluge ergeben sich deutliche Verbesserungen der mittleren Optimierungsleistung; das Ergebnis eines Optimierer–Laufes wird also verläßlicher. Gleichzeitig sinkt die Zahl der aufgewendeten Versuche stark; sie wird jetzt wesentlich von der Abbruchbedingung $t_{s,max}$ geprägt. Dies wird auch in Abbildung 6.7 deutlich: Verglichen mit Abbildung 4.21 auf Seite 51 zeigt sich, daß Record–To–Record–Traveling für $D > 500$ nahezu keine Verbesserung mehr erzielen kann; das Verhalten ist hier analog zum Monte–Carlo–Verfahren, und die Optimierung endet nach $t_{s,max} = 1000$. Für $D = 50$ verschlechtert sich das Verhalten jetzt nicht mehr; durch den mit der Heuristik ermittelten Startpunkt stagniert das Verfahren nicht mehr in lokalen Minima, sondern findet — im Verhalten jetzt ähnlich wie Hill–Climbing — häufig das Optimum.

6.2 Möglichkeiten der Verbesserung

Aus den Ergebnissen der Experimente lassen sich einige Hinweise zu Verbesserungsmöglichkeiten entnehmen. Für die Genetischen Algorithmen erwies sich die gewählte Realisierung als nicht erfolgreich zur Lösung des gestellten Problems. Zwar war der Startwert in den Experimenten für die Genetischen Algorithmen denkbar ungünstig, da alle Individuen initial die gleiche Konfiguration „alle Tasks auf einem Rechner“ nutzten, aber einige Tests mit einer rein zufälligen Verteilung auf mehrere Rechner, d. h. mit einer größeren „Genvielfalt“ zu Beginn, brachten keine wesentliche Verbesserung. Dennoch ist dieser Startpunkt sinnvoller, insbesondere bei kleineren Mutationsraten.

Ein größeres Problem ist die lokale Selektion innerhalb getrennter Populationen, die wegen der einfacheren Parallelisierbarkeit gewählt wurde. Hier empfiehlt es sich, selbst wenn damit ein höherer Kommunikationsaufwand bei der Implementierung des Optimierers in parallele Prozesse in Kauf genommen werden muß, zu *einer* Population mit nicht überlappenden Generationen und globaler Selektion überzugehen, da hier die zielgerichtete Komponente wesentlich stärker zum Tragen kommt.

Auch die Crossover–Operation sollte so geändert werden, daß Strings bei einfachem Crossover nur an einer Stelle aufgebrochen werden. Zusätzlich erscheint eine Implementierung von Operatoren sinnvoll, die die Reihenfolge der Parameter innerhalb eines Strings verändern (Inversion, Partially Matched Crossover). Dadurch besteht die Chance, daß die Parameter gekoppelter Tasks (z. B. durch Kommunikation oder Restriktionen) benachbart im String abgelegt und gemeinsam vererbt werden.

Bei allen Verfahren wäre für den Anwender eine automatische Bestimmung der Optimierer–Parameter sehr hilfreich. Für Simulated Annealing und Threshold Accepting kann eine sinnvolle Anfangstemperatur T_0 leicht wie in der Literatur vorgeschlagen aus einigen zufällig gewählten Konfigurationen ermittelt werden. Auch eine Ableitung aus dem mit der Heuristik ermittelten Wert, z. B. die doppelten dort bestimmten Kosten, ist ohne großen Initialaufwand möglich. Ähnlich können sinnvolle Werte für die Absenkungsparameter D von Record–to–Record–Traveling und Great Deluge aus der Kostendifferenz zufälliger Testkonfigurationen bestimmt werden.

Da die Kostenkurven, insbesondere in den ersten Phasen der Optimierung bei einem schlechten Startwert, sehr stark fallen und danach abflachen, sollten die Abkühlungs– und Absenkungs–Parameter (α bzw. D) dynamisch dem Optimierungsfortschritt angepaßt werden (vgl. Abbildung 4.36(a) auf Seite 70), um einen schnelleren Übergang in zielgerichtete Phasen zu erreichen (vgl. Kapitel 4.5.2).

Ebenso sollte das Abbruchkriterium dynamisch angepaßt werden, indem ab der Ermittlung eines neuen Bestwertes nur noch maximal z. B. 10% der bis dahin benötigten Anzahl von Versuchen investiert werden, ohne daß ein neuer Bestwert gefunden wird. Eine gewisse Anzahl von Versuchen (z. B. 1000) muß allerdings auf jeden Fall durchgeführt

werden.

Ob eine generelle Verwendung der Heuristik–Ergebnisse als Startwert sinnvoll ist, kann aus der beschränkten Anzahl von Experimenten nicht allgemeingültig beantwortet werden. Auf jeden Fall aber kann das Heuristikergebnis als Vergleichsmaßstab herangezogen werden, insbesondere da es mit relativ wenig Aufwand berechnet werden kann.

Bei Einbeziehung der Kommunikation könnte es sich erweisen, daß die heuristische Festsetzung von Zwischen–Deadlines (vgl. Kapitel 4.1.3.2) zu suboptimalen Ergebnissen führt. In diesem Fall bietet sich an, diese Deadlines durch einen zwischengeschalteten Optimierungsschritt festzulegen. Auch hierfür kann ein stochastisches Verfahren zum Einsatz kommen.

7 Zusammenfassung

Der Entwurf verteilter Realzeitsysteme ist überaus komplex und manuell nur sehr aufwendig optimierbar. Auf der einen Seite gilt es, die Kosten durch geschickte Auswahl der notwendigen Komponenten möglichst gering zu halten, auf der anderen Seite muß der Echtzeitbetrieb, also die schritthaltende Verarbeitung, dabei immer gewährleistet sein.

In dieser Arbeit wird ein System für die automatische Auslegung kostengünstiger verteilter Realzeitsysteme vorgestellt. Dieses System benötigt als Eingaben die formale Beschreibung der zur Verfügung stehenden Typen von Hardwarekomponenten, die nach Verarbeitungseinheiten (Rechnerknoten) und Kommunikationskomponenten differenziert werden, sowie die formale Beschreibung der Softwarestruktur und die formale Beschreibung der (dynamischen) Lastanforderung (Ereignisstrom). Daraus wird eine möglichst kostengünstige Hardwarekonfiguration und die Abbildung der Softwarestruktur auf diese Hardwarekonfiguration (Taskallokation) ermittelt.

Aufgrund der vielen Parameter (z. B. Rechneranzahl, Rechnerleistung) wird zur Ermittlung des Lösungsraumes ein (stochastisches) Optimierungsverfahren verwendet, welches eine Hardwarekonfiguration mit der zugehörigen Taskverteilung auf die Hardwarekomponenten vorschlägt. Die gewählte Konstellation wird bezüglich ihrer Realzeiteigenschaften verifiziert und bewertet. Aufgrund der Bewertung schlägt das Optimierungsverfahren eine erneute Konstellation vor. Der Vorgang wird so oft wiederholt, bis das kostengünstigste System ermittelt werden konnte.

Das Verfahren wurde für verschiedene stochastische Optimierungsalgorithmen prototypisch in ein Optimierungstool implementiert und auf beispielhafte Systemmodelle angewendet: Die beste Eignung bezüglich erreichtem Ergebnis und notwendigem Aufwand weisen danach die Algorithmen *Great Deluge* und *Treshold Accepting* auf. *Genetische Algorithmen* erscheinen ungeeigneter, da sie sich nur sehr schwierig an die gegebene Problematik anpassen lassen. Zur Adaptierung der stochastischen Algorithmen an das untersuchte Problem werden die problemspezifischen Funktionen zur Startpunktbestimmung, zur Änderung einer Konfiguration und zur Ermittlung einer kostenorientierten Bewertungszahl entwickelt.

Zur Ergänzung der stochastischen Verfahren wird eine deterministische Heuristik vorgeschlagen, die zu einer sinnvollen, wenn auch suboptimalen Lösung führt. Diese heu-

ristisch ermittelte Konfiguration kann als Startwert zur weiteren Optimierung und als schnell zu ermittelnder Vergleichswert herangezogen werden.

Bedingt durch die übernommene Komponente zum Echtzeitnachweis ist die realisierte Variante des Verfahrens noch auf dedizierte Einsatzbereiche eingeschränkt. So wird gegenwärtig davon ausgegangen, daß Tasks statisch auf die einzelnen Rechnerknoten alloziert sind, daß die einzelnen Rechner das *Earliest Deadline First* Scheduling verwenden und zur Kommunikation ein echtzeitfähiges TDMA–Verfahren eingesetzt wird. Das Verfahren ist aber unter Einbindung eines angepaßten Verfahrens zur Realzeitanalyse leicht auf andere Systemumgebungen anpaßbar.

Die Notwendigkeit, formale Methoden beim Systementwurf zu verwenden, steigt in Zukunft, insbesondere auch aus der Forderung nach hoher Verläßlichkeit eingebetteter Systeme. Damit einher geht die Möglichkeit, den Systementwurf als solchen zu automatisieren und nicht nur sicherer, sondern auch kostengünstiger zu gestalten. Allerdings sind dazu weitere Arbeiten notwendig, um formale Methoden für den Anwender einfacher — z. B. durch entsprechende Frontends — nutzbar zu machen.

Die vorliegende Arbeit zeigt, daß eine automatische Systemauslegung für verteilte Echtzeitsysteme unter Verwendung stochastischer Verfahren möglich und sinnvoll ist.

A Implementierte Algorithmen

In diesem Kapitel werden die Algorithmen der verwendeten Optimierungsverfahren in der Ausprägung wiedergegeben, wie sie tatsächlich in dem prototypischen Optimierungswerkzeug *EZA* (für *E*cht*Z*eit*A*nalyse– und Optimierungs–Tool) realisiert wurden [Her94, Mus95, Bou95, Wet94, Pet95]. Da die Algorithmen in allgemeiner Form bereits in Kapitel 4 erläutert wurden, wird hier nur auf die Änderungen eingegangen.

A.1 Simulated Annealing

Die implementierte Version von Simulated Annealing (Abbildung A.1) unterscheidet sich nur geringfügig vom allgemeinen Algorithmus aus Abbildung 4.9 auf Seite 40. Gegenüber dem Original wird vor allem die beste Konfiguration gespeichert und als Ergebnis zurückgegeben, da Simulated Annealing nicht notwendigerweise mit der besten der getesteten Varianten terminiert.

Die Startkonfiguration X_0 und Anfangstemperatur T_0 werden nicht automatisch bestimmt, sondern müssen für *EZA* manuell spezifiziert werden. Die gewählten Werte sind jeweils bei den Untersuchungen in Kapitel 4.2.2 angegeben.

Wenn die Anzahl c akzeptierter Änderungen einen Grenzwert c_{temp} überschreitet, wird die innere Schleife abgebrochen („Temperaturausgleich“), wobei die Anzahl t *aller versuchten* Änderungen auf $t_{\max}$ begrenzt wird. $t_{\max}$ sollte daher wesentlich größer als c_{temp} gewählt werden.

Als Abkühlfunktion wird die in [KGV83] vorgeschlagene Multiplikation mit einer Konstanten α verwendet (siehe Seite 41). Der Algorithmus terminiert, sobald auf einem Temperaturniveau keine Änderung mehr akzeptiert wurde.

Zusammengefaßt ergeben sich damit folgende Optimierer–Parameter:

Wähle Startkonfiguration $X_{\text{best}} = X = X_0$
Wähle Anfangstemperatur $T = T_0$
Wähle nötige Anzahl akzeptierter Änderungen c_{temp} bis „Temperaturausgleich“
Wähle maximale Anzahl Versuche $t_{\max}$ bis „Temperaturausgleich“ ($t_{\max} \gg c_{\text{temp}}$)
repeat
  $t = c = 0$
  repeat
    $X_{\text{test}} = \text{ChangeOf}(X)$
    $\Delta C = \text{Cost}(X_{\text{test}}) - \text{Cost}(X)$
    if $\Delta C \leq 0$ **then**
      $X = X_{\text{test}}$
      $c = c + 1$
      if $\text{Cost}(X) < \text{Cost}(X_{\text{best}})$ **then**
        $X_{\text{best}} = X$
      end if
    else
      $r = \text{random}()$ (gleichverteilt; $0 < r < 1$)
      if $r < e^{-\Delta C/T}$ **then**
        $X = X_{\text{test}}$
        $c = c + 1$
      end if
    end if
    $t = t + 1$
  until $c \geq c_{\text{temp}}$ **or** $t \geq t_{\max}$ („Temperaturausgleich“)
  $T = \text{CoolDown}(T)$
until $c \equiv 0$ („keine Verbesserung“)
return X_{best}

Abbildung A.1: „Simulated Annealing“, implementierte Version

Parameter	Anmerkung
T_0	Anfangstemperatur
c_{temp}	Anzahl akzeptierter Änderungen bis „Temperaturausgleich“ angenommen wird; begrenzt durch t_{max}
t_{max}	maximale Anzahl von Änderungsversuchen auf einem Temperaturniveau; $t_{max} \gg c_{temp}$
α	Abkühlungsfaktor; $0 < \alpha < 1$

A.2 Threshold Accepting

„Threshold Accepting“ wurde bis auf die für diesen Algorithmus andere Akzeptanzbedingung (siehe Kapitel 4.3.1 auf Seite 46) genauso wie Simulated Annealing (Abbildung A.1) implementiert. Alternativ zur Abkühlfunktion von Simulated Annealing kann auch eine feste Schwellwert–Sequenz vorgegeben werden. Eine feste Anzahl n von Versuchen für die innere Schleife, wie sie in [DS90] vorgeschlagen wird, kann durch die Wahl der Parameter $c_{temp} = t_{max} = n$ erreicht werden.

A.3 Record–to–Record–Traveling

Abbildung A.2 zeigt den implementierten Algorithmus von „Record–to–Record–Traveling“. Der Algorithmus entspricht genau dem Original in Abbildung 4.20 auf Seite 50; lediglich das Abbruchkriterium ist genauer spezifiziert: Die Optimierung endet, sobald über $t_{s,max}$ Versuche keine Kostenverbesserung erreicht wurde, oder wenn mehr als t_{max} Versuche durchgeführt wurden.

In der folgenden Tabelle sind die einstellbaren Parameter von Record–to–Record–Traveling zusammengefaßt:

Parameter	Anmerkung
D	maximale Abweichung vom bisherigen Bestwert
$t_{s,max}$	maximale Anzahl von Änderungsversuchen ohne Kostenverbesserung
t_{max}	maximale Anzahl von Änderungsversuchen; $t_{max} \gg t_{s,max}$

A.4 Great Deluge Algorithm

Die Implementierung des „Great Deluge Algorithm“ ist in Abbildung A.3 dargestellt. Wieder wird die beste gefundene Konfiguration X_{best} gespeichert, da der Algorithmus

```
Wähle Startkonfiguration X_R = X = X_0
Wähle maximale Abweichung D vom Bestwert
Wähle maximale Anzahl Versuche t_s,max ohne Verbesserung
Wähle maximale Anzahl Versuche t_max    (t_max ≫ t_s,max)
t = t_s = 0
repeat
   X_test = ChangeOf(X)
   t_s = t_s + 1
   ΔC = Cost(X_test) − Cost(X)
   if ΔC < D then
      X = X_test
      if ΔC < 0 then
         X_R = X_test
         t_s = 0
      end if
   end if
   t = t + 1
until t_s ≥ t_s,max or t ≥ t_max
return X_R
```

Abbildung A.2: „Record–to–Record–Traveling“, implementierte Version

mit einer um D schlechteren Konfiguration als X_{best} terminieren kann. Die Abbruchbedingungen sind ähnlich zum vorherigen Algorithmus, wobei $t_{\text{s,max}}$ hier die maximale Anzahl von Versuchen ohne Senkung des Niveaus angibt.

Die Niveau–Senkung wird dynamisch als Maximum aus dem vom Anwender gewählten Wert D und der mit α_g gewichteten Kostenverbesserung gebildet. In den Experimenten wurde keine Variation von α_g untersucht ($\alpha_g = 0$).

Die folgende Tabelle zeigt die Parameter dieses Optimierungsverfahrens:

Parameter	Anmerkung
D	fixer Kostenwert für Niveausenkung
α_g	Faktor für Niveausenkung relativ zur Kostendifferenz; $0 \leq \alpha_g \leq 1$
$t_{\text{s,max}}$	maximale Anzahl von Änderungsversuchen ohne Niveausenkung
t_{max}	maximale Anzahl von Änderungsversuchen; $t_{\text{max}} \gg t_{\text{s,max}}$

Wähle Startkonfiguration $X_{\text{best}} = X = X_0$
Wähle Verringerung D
Wähle maximale Anzahl Versuche $t_{\text{s,max}}$ ohne Niveausenkung
Wähle maximale Anzahl Versuche t_{max} $\quad (t_{\text{max}} \gg t_{\text{s,max}})$
$t = t_s = 0$
$N = \text{Cost}(X_0)$
repeat
  $X_{\text{test}} = \text{ChangeOf}(X)$
  $t_s = t_s + 1$
  $\Delta C = \text{Cost}(X_{\text{test}}) - N$
  if $\Delta C < 0$ **then**
    $X = X_{\text{test}}$
    $D_{\text{dyn}} = -\alpha_g * \Delta C$
    $N = N - \max(D, D_{\text{dyn}})$
    $t_s = 0$
    if $\text{Cost}(X) < \text{Cost}(X_{\text{best}})$ **then**
      $X_{\text{best}} = X$
    end if
  end if
  $t = t + 1$
until $t_s \geq t_{\text{s,max}}$ **or** $t \geq t_{\text{max}}$
return X_{best}

Abbildung A.3: „Great Deluge Algorithm“, implementierte Version

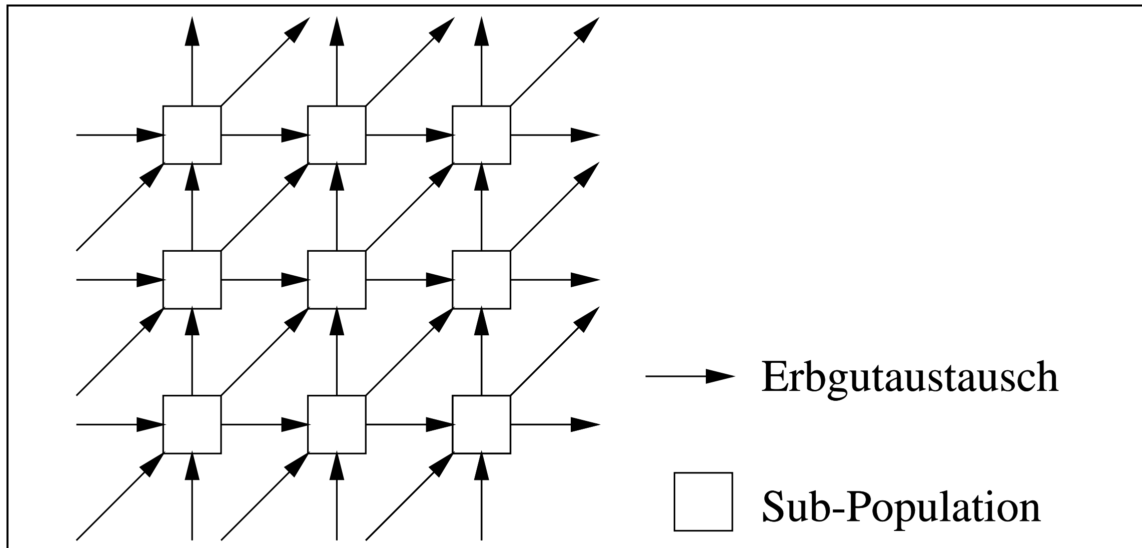


Abbildung A.4: Populationsstruktur

A.5 Genetische Algorithmen

Wie in Kapitel 4.6 bereits erwähnt wurde, wurde zur einfacheren Implementierung des Optimierungsverfahrens in parallele UNIX–Prozesse ein Ansatz gewählt, bei dem Reproduktion und Selektion in lokal Populationen erfolgen (vgl. Kapitel 4.6.5 auf Seite 67). Hierzu sollen im folgenden noch einige Details verdeutlicht werden.

A.5.1 Populationsstruktur

Ein einzelner UNIX–Prozeß repräsentiert eine lokale Population, in der *ein* Eltern–Individuum Erbgut mit den Individuen der anderen UNIX–Prozesse austauscht und daraus in Rekombination mit dem eigenen Erbgut *mehrere* Nachkommen erzeugt (Abbildung 4.32 auf Seite 62).

Aus jedem der empfangenen Erbgut–Strings werden durch Rekombination mit dem eigenen Erbgut (per Optimierer–Parameter einstellbar) ein oder zwei Nachkommen erzeugt (Abbildung 4.33 auf Seite 66).

Für die Kommunikation der Eltern–Individuen untereinander wurde eine quadratische Struktur gewählt, bei der von jedem Prozeß das Erbgut unidirektional zu drei Nachbarn geschickt wird (Abbildung A.4). Weiterhin wurde eine Variante implementiert, bei der jeder Prozeß sein Erbgut an jeden anderen Prozeß sendet.

Die Anzahl n_N aller Nachkommen ist also durch die Anzahl der Eltern–Individuen n_E (d. h. Anzahl der parallelen Prozesse), durch die Zahl der Kommunikationspartner n_K

(hier 3 oder $n_E - 1$) sowie durch die Zahl der Nachkommen je Crossover–Operation n_C (hier 1 oder 2) festgelegt:

$$n_N = n_E * n_K * n_C$$

Da bei der lokalen Selektion auch das Erbgut des lokalen Elternteils mit einbezogen wird (überlappende Generationen), ergibt sich die insgesamt im Selektionsprozeß berücksichtigte Population n_P zu

$$n_P = n_E + n_N$$

In den Experimenten wird die Population G_{pop} durch das Tripel (Eltern, Nachkommen, Gesamtpopulation) beschrieben. Bei 16 Prozessen, unidirektionaler Kommunikation (d. h. 3 Kommunikationspartnern) und einem Nachkommen pro Crossover ergibt dies also $G_{\mathrm{pop}} = (16, 6, 112)$, bei 8 Prozessen, vollständiger Kommunikation und zwei Nachkommen je Crossover $G_{\mathrm{pop}} = (8, 14, 120)$.

A.5.2 Initiale Population

Die initiale Population wird entsprechend Kapitel 4.6.3.3 zufällig ermittelt. In der Implementierung ist hierbei die Anzahl der initial belegbaren Rechner einstellbar. Wird hier nur ein Rechner erlaubt, ergibt sich ein Start mit identischem Erbgutsatz bei allen Individuen.

A.5.3 Fitness und Selektion

Zusätzlich zu der in Kapitel 4.6.4.3 vorgeschlagenen Abbildung der Kostenfunktion auf die Fitnessfunktion durch Kehrwertbildung wurde zusätzlich eine Differenzbildung zu einem maximalen Kostenwert implementiert:

$$f_i = K_{\max} - K_i$$

Neben der Selektion nach der Roulette–Methode ist in der implementierten Version auch die Auswahl des besten Strings in einer Subpopulation möglich.

A.6 Vergleichsverfahren

Die im folgenden beschriebenen einfachen Optimierungsverfahren werden in Kapitel 6.1 zur Bewertung der zielgerichteten stochastischen Verfahren verwendet.

A.6.1 Monte–Carlo

Dieses Verfahren tastet den Parameterraum an zufällig gewählten Punkten ab, indem die momentane Konfiguration — ausgehend von der gewählten Startkonfiguration (siehe Kapitel 4.1.1) — mit Hilfe der Änderungsfunktion aus Kapitel 4.1.3 modifiziert wird; diese geänderte Konfiguration wird wiederum als Ausgangspunkt für die nächste Iteration benutzt. Die jeweils beste Konfiguration wird gespeichert und wird nach Durchführung von t_{max} Versuchen als Ergebnis zurückgegeben (Abbildung A.5).

Wähle Startkonfiguration $X_{best} = X = X_0$
for $i = 1$ to t_{max} **do**
  $X = \text{ChangeOf}(X)$
  if $\text{Cost}(X) < \text{Cost}(X_{best})$ **then**
    $X_{best} = X$
  end if
end for
return X_{best}

Abbildung A.5: Algorithmus „Monte–Carlo“

Im Gegensatz zu den Simulated–Annealing–Varianten und den genetischen Algorithmen wird also eine geänderte Konfiguration unabhängig von den Kosten als Ausgangspunkt für die nächste Iteration akzeptiert; es sind somit beliebige Verbesserungen oder Verschlechterungen des Kostenwertes möglich.

Der einzige Parameter dieses Verfahrens, die Anzahl der Versuche t_{max}, wird so eingestellt, daß ungefähr die gleiche Anzahl von Versuchen wie bei den „intelligenteren“ Verfahren durchgeführt wird. Durch Vergleich der durchschnittlichen Qualität der mit „Monte–Carlo“ bzw. den anderen Verfahren gefundenen Lösungen kann so festgestellt werden, wie sich die „zielgerichteten“ Auswahlverfahren der anderen Optimierungsmethoden auswirken.

A.6.2 Hill–Climbing

Auch dieses Verfahren unterscheidet sich nur durch die Akzeptanzbedingung einer Konfiguration für den nächsten Iterationsschritt von den anderen Optimierungsverfahren: Eine geänderte Konfiguration wird nur dann akzeptiert, wenn sich eine Kostenverbesserung ergibt[1]. Dadurch bleibt dieses Verfahren im ersten gefundenen lokalen Optimum hängen. Der Algorithmus ist in Abbildung A.6 dargestellt.

[1] Der Name „Hill–Climbing“ ist im Hinblick auf eine Maximierungsaufgabe gewählt; bei der hier vorliegenden Optimierungsaufgabe wäre also „Hill–Descending“ zutreffender.

```
Wähle Startkonfiguration X_best = X_0
for i = 1 to t_max do
   X_test = ChangeOf(X_best)
   if Cost(X_test) < Cost(X_best) then
      X_best = X_test
   end if
end for
return X_best
```

Abbildung A.6: Algorithmus „Hill–Climbing“

Auch hier wird mit dem Parameter t_{max} ungefähr die gleiche Anzahl von Versuchen eingestellt, die auch die anderen Optimierungsverfahren benötigen. Dies erlaubt die Beurteilung deren Fähigkeit, lokale Optima (durch die Akzeptanz auch schlechterer Lösungen) zu verlassen: Sie müssen näher an das globale Optimum gelangen und daher bessere Ergebnisse als Hill–Climbing erreichen.

Das Hill–Climbing–Verfahren ist als Sonderfall z. B. in Record–To–Record–Traveling enthalten, wenn die Parameter $D = 0$ und $t_{s,max} = t_{max}$ gewählt werden.

B Verwendete Beispiele

B.1 Systemmodell

Für die in Kapitel 4.2 ff. vorgestellten Untersuchungen wurden die in Tabelle B.1 aufgeführten Rechnertypen zur Auswahl vorgegeben. Die Kostenkurve entspricht Abbildung 4.2 auf Seite 25. Basis dieser Aufstellung ist eine Erhebung der Kosten von PC–Mainboards inklusive Prozessor von Juni 1994. Eine Wiederholung der Kostenanalyse Mitte 1996 zeigte eine qualitativ ähnliche Kostenkurve mit einer der Weiterentwicklung der Prozessoren entsprechenden Leistungsverschiebung.

Fixkosten für Gehäuse, Stromversorgung, Speicher oder Installation wurden nicht eingerechnet. Die Untersuchungen entsprechen daher in der Realität am ehesten einem System mit Backplane und zusteckbaren Slot–CPU–Boards.

Der Rechner mit bestem Preis–/Leistungs–Verhältnis ist in diesem Beispiel der Typ C10 (Abbildung 4.3 auf Seite 25). Je nach Höhe der hinzuzurechnenden Fixkosten verschiebt sich das beste Preis–/Leistungs–Verhältnis hin zu leistungsfähigeren Typen.

Anmerkung: TODO: Anpassen auf geändertes Bild; Kommunikationssystem

B.2 Taskmodell E

Das Modell E besteht aus 100 identischen, periodischen Tasks. In jeder Task sind 3100 Befehle abzuarbeiten; die Deadline beträgt 1ms. Die Tasks werden mit einer Periode von 1ms aktiviert. Jede Task benötigt damit eine Rechenleistung von 3,1 MIPS. Abbildung B.1 zeigt die grafische Darstellung einer solchen Task mit einer auf 10 MIPS bezogenen Rechenzeit.

Wegen des sehr einfachen Systems ist direkt nachvollziehbar, daß die kostengünstigste Realisierung aus 32 Rechnern vom Typ C10 (bestes Preis–/Leistungsverhältnis) mit jeweils 3 Tasks (Auslastung 93%) und 2 Rechnern vom Typ C9 mit jeweils 2 Tasks besteht (Auslastung 68,9%). Der minimale Wert für die Hardwarekosten beträgt somit 5548;

Name	Rechenleistung (SpecInt92)	Kosten DM	realer Prozessor / Taktrate [MHz]
C90	90	2896	P5 / 90
C65	65	2446	P5 / 66
C58	58	2196	P5 / 60
C51	51	1616	i486 DX/4 / 100
C32	32	846	i486 DX/2 / 66
C30	30	757	i486 DX / 50
C26	26	602	i486 DX/2 / 50
C18	18	502	i486 DX / 33
C14	14	324	i486 SX / 25
C10	10	164	i386 DX / 40
C9	9	150	i386 DX / 33
C2	2	70	i286 DX / 10

Tabelle B.1: Rechnertypen der Beispiel–Modelle

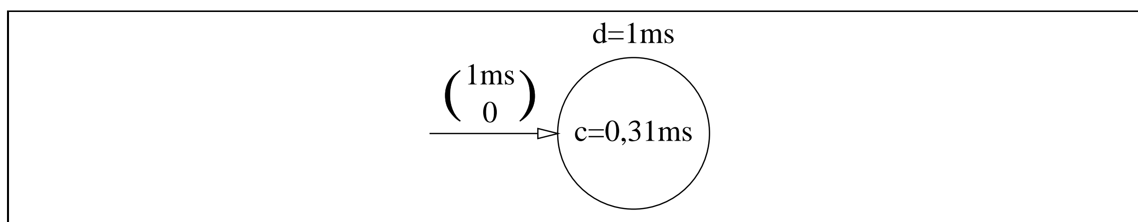


Abbildung B.1: Task in Beispiel E

als Gesamtwert für die Kostenfunktion ergibt sich mit Einrechnung der Beträge für die Rechnerlasten im optimalen Fall 5588,34.

B.3 Taskmodell R

Es handelt sich hier um ein komplexeres Modell, das an das realitätsnahe Anwendungsbeispiel aus [Gre93, Anhang C] angelehnt ist, wobei die auf Seite 20 genannten Einschränkungen gelten und wie in Modell E noch keine Kommunikation vorhanden ist. Zur Erhöhung der Komplexität wurde die Anzahl der benötigten Befehle für jede Task verdoppelt. Weiterhin ist das gesamte Modell 4fach redundant ausgelegt, d. h. jede Task existiert in vier Replikaten, die alle auf verschiedene Rechner alloziert werden müssen.

Das 1fach ausgelegte Modell besteht aus 27 Tasks, das gesamte Modell enthält also 108 Tasks. Eine Zusammenstellung der Taskparameter für ein 1faches Modell befindet sich in Tabelle B.2.

Die sich überschneidenden Semaphorgruppen ergeben folgende Gruppen von Tasks, die auf denselben Rechner alloziert werden müssen:

Gruppe	Tasks	Semaphore
g_1	T1, T2, T8	S1, S2
g_2	T9, T10, T11, T15	S3, S4, S10
g_3	T13, T14	S5
g_4	T17, T18	S6
g_5	T19, T22, T24	S7
g_6	T25, T26, T27	S8, S9

Die Gruppen existieren jeweils 4fach für die vier Replikate der Tasks, insgesamt ergeben sich 24 Gruppen mit dieser Restriktion. Diese Gruppen werden bei der Taskallokation entsprechend Kapitel 4.1.3.1 immer gemeinsam plaziert.

Durch die vielfältigen Restriktionen und die komplexe Anregung läßt sich der optimale Kostenwert für dieses Modell nicht mehr ohne weiteres berechnen.

Der minimale Wert aller Optimierungsläufe beträgt für die Gesamtkosten 3324,65 und für die reinen Hardwarekosten 3050. Dabei handelt es sich um ein System mit 8 Rechnern (1 mal C9, 3 mal C10 und 4 mal C26).

Task	RA [Befehle]	DL [ms]	kritische Bereiche	ES (Zyklus, Offset) [ms]	RL [MIPS]
T1	5.000	2	S1,S2	(40,0)	2.5
T2	4.200	40	S1	(40,0)	0.11
T3	80.000	40		(40,0)	2
T4	80.000	40		(40,0)	2
T5	80.000	40		(40,0)	2
T6	80.000	40		(40,0)	2
T7	300.000	80		(80,0)	3.75
T8	6.000	20	S2	2(7200,0),(7200,3600)	0.6
T9	15.000	100	S4	2(7200,0),(7200,3600)	0.3
T10	7.800	20	S3,S4,S10	(80,0)	0.39
T11	3.000	20	S3	2(7200,0),(7200,3600)	0.3
T12	1.500.000	200		2(7200,0),(7200,3600)	15
T13	2.000	220	S5	(3600,0)	0.01
T14	100.000	400	S5	(3600,0)	0.25
T15	3.000	60	S10	(250,0)	0.05
T16	78.000	40		(40,0)	1.95
T17	8.000	40	S6	(40,0)	0.2
T18	3.000	100	S6	(100,0)	0.03
T19	5.000	100	S7	(100,0)	0.05
T20	300.000	100		(100,0)	3
T21	200.000	200		(100,0)	1
T22	300.000	300	S7	(300,0)	1
T23	4.000	20		(3600,0)	0.2
T24	300.000	120	S7	(3600,0)	2.5
T25	2.000	50	S8	(3600,0)	0.04
T26	3.000	67	S8,S9	(67,0)	0.04
T27	3.000	70	S9	(250,0)	0.04

RA=Rechenarbeit, DL=Deadline, ES=Ereignisstrom, RL=erforderliche Rechenleistung

Tabelle B.2: Task–Parameter Modell R

Abkürzungen

CPU	Central Processing Unit
DMA	Direct Memory Access
EDF	Earliest Deadline First
EZA	Prototypische Implementierung des Optimierungsverfahrens in das *E*cht*Z*eit*A*nalyse– und Optimierungs–Tool
GA	Genetische Algorithmen
GD	Great Deluge Algorithm
MSC	Message Sequence Chart
RTR	Record–To–Record–Traveling
RZAF	Rechenzeit–Anforderungsfunktion
SA	Simulated Annealing
SDL	Specification and Description Language
TA	Threshold Accepting
TDMA	Time Division Multiple Access

Glossar

Allokation: Verteilung von → Tasks auf → Rechnerknoten

Antwortzeit: → Reaktionszeit

Block–DMA: Direct Memory Access, bei dem der Hauptprozessor während der Übertragung eines ganzen Speicherblocks zu bzw. von einem Peripheriegerät keinen Speicherzugriff hat.

Crossing Over: In der Biologie verwendeter Begriff für das bei Genetischen Algorithmen häufiger verwendete → Crossover (vgl. S. 60).

Crossover: Operation der Genetischen Algorithmen: Neubildung des Erbguts der Nachkommen aus den → Strings der Eltern (Kapitel 4.6.4.1 Seite 65).

Cycle–Stealing–DMA: Direct Memory Access, bei dem der Hauptprozessor während der Übertragung eines ganzen Speicherblocks zu bzw. von einem Peripheriegerät beim Speicherzugriff gebremst wird, da einzelne Zyklen für DMA benutzt werden.

Deadline: Zeitpunkt, zu dem die → maximal zulässige Reaktionszeit abläuft.

Earliest Deadline First–Scheduling: Zuteilungsverfahren für Rechenprozesse nach frühesten geforderten Antwortzeiten

Ereignisstrom: Beschreibung der Auftrittszeitpunkte unterscheidbarer Ereignisse durch Ereignistupel bestehend aus Zyklus z_i und Intervall a_i beschreiben (aus [Gre93]; vgl. Seite 15).

Fitness: Numerischer Wert, der bei Genetischen Algorithmen die Güte eines Erbinformations–Strings wiedergibt. Ein höherer Wert bedeutet eine höhere Qualität.

NP–vollständige Probleme: Klasse von Problemen, bei denen keine analytische Lösung möglich ist, deren Aufwand nur polynomial mit der Komplexität des Problems wächst.

Partitionierung: Verteilung der Funktionen eines komplexen Systems auf einzelne Funktionsträger, insbesondere auch Aufteilung in Hardware– und Software–

Module sowie Verteilung der Hardware–Module auf einzelne Baugruppen und der Software–Module (Tasks) auf Rechner.

Präzedenzsystem: System von Tasks, in dem durch Kommunikationsbeziehungen Abhängigkeiten in der Reihenfolge der Bearbeitung entstehen (vgl. Abbildung 3.4 auf Seite 14).

Rechenzeit–Anforderungsfunktion: maximale Rechenzeitsumme für Anforderungen, die innerhalb eines Intervalls angefordert werden und beendet sein müssen [Gre93].

Reaktionszeit: Unter Reaktionszeit t_R versteht man die Zeit, innerhalb derer ein Rechenprozeß seine Ausgabe an den technischen Prozeß ausgeführt hat. Der Bezugszeitpunkt der Reaktionszeit ist dabei das Eintreffen des Ereignisses. Damit ist die Reaktionszeit die Summe aus Wartezeit plus Verarbeitungszeit des Rechenprozesses. Eine Wartezeit ergibt sich beispielsweise durch höherpriorisierte Tasks (oder Interrupts).

maximale Reaktionszeit: Die maximale Reaktionszeit t_{Rmax} ergibt sich aufgrund der Auslegung des Rechensystems und ist die Zeit, die der Rechner maximal (im → Worst Case) zur Beantwortung einer Anfrage (Ereignis) benötigt.

maximal zulässige Reaktionszeit: Die maximal zulässige Reaktionszeit (Deadline) ist durch den technischen Prozeß vorgegeben, der das Ereignis ausgelöst hat. Innerhalb der maximalen Reaktionszeit muß der Rechenprozeß seine Ausgabe an den technischen Prozeß durchgeführt haben, damit die Echtzeitbedingung erfüllt ist ($t_R \leq t_{\mathrm{Rzul}}$.

Realzeitnachweis: Analytischer Nachweis, daß alle Antwortzeiten eines Realzeitsystems mit harten Realzeitbedingungen in jedem Fall (also auch im Worst Case) innerhalb der geforderten Deadlines liegen.

SpecInt92: Benchmark der SPEC von 1992 zum Vergleich der Integerleistung von Mikroprozessoren.

String: Repräsentation des „Erbguts“ bei Genetischen Algorithmen, analog zu Chromosomen in der Biologie.

Task: Rechenprozeß

Technischer Prozeß: Ein Prozeß ist nach [DIN81] die “Umformung und/oder Transport von Materie, Energie und/oder Information”. Man spricht von einem technischen Prozeß, wenn Materie und/oder Energie umgeformt und/oder transportiert wird.

Unabhängige Task: Task ohne Kommunikationsbeziehungen und Restriktionen.

Virtueller Rechner: Nicht verfügbares Rechenelement, das in der Optimierung temporär verwendet wird, wenn kein für die momentan getestete Konfiguration ausreichend leistungsfähiges Rechenelement vorhanden ist. Die Rechenleistung des virtuellen Rechners wird auf den benötigten Wert gesetzt, die Kosten bestimmen sich in Abhängigkeit der geforderten Rechenleistung und liegen wesentlich über den Kosten des leistungsfähigsten realen Rechenelements (siehe Kap. 4.1.2.2 auf Seite 27).

Worst–Case–Laufzeit: Längste Laufzeit einer Task, die „unter schlimmsten Bedingungen“ (Worst Case) benötig wird.

Variablenverzeichnis

In der Tabelle verwendete Abkürzungen:
(ÄF) Änderungsfunktion, (SA) Simulated Annealing, (TA) Threshold Accepting, (GD) Great Deluge Algorithm, (RT) Record–to–Record–Traveling, (GA) Genetische Algorithmen

Variable	Bedeutung	Seite
ΔC	Kostenunterschied	40, 46, 50
D	(RT) maximale Abweichung (Deviation) vom bisherigen Bestwert (GD) fixer Kostenwert für Niveausenkung	50, 52, 54–56, 101, 102, 107
E_i	Ereignis (Anforderung) i innerhalb eines Ereignisstroms	14, 15
G_{break}	(GA) Anzahl von Generationen ohne Kostenverbesserung, nach denen abgebrochen wird	68, 69, 105
G_{pop}	(GA) Populationsgröße; Notation: (Eltern–Exemplare, Nachkommen, Gesamtpopulation)	68, 105
K_{HW}	Hardwarekosten eines Rechners (reale und virtuell)	24, 26, 33
K_{K}	Kostenwert für Kommunikationselemente	31
K_{Last}	Lastabhängiger Kostenwert eines Rechners	24, 28
K_{R}	Rechnerkosten	24, 30
K_{Restrikt}	Kostenwert zur Bewertung von Restriktionsverletzungen	24, 30
K_{ges}	Gesamtkosten eines Rechnersystems	31, 33
K_i	(GA) Kostenwert eines Individuums i	105
$K_{\text{r,real}}$	Reale Hardware–Kosten des Rechners r	24, 26
$K_{\text{r,virt}}$	Virtuelle Hardware–Kosten eines nicht existierenden Rechners r in Abhängigkeit von der geforderten Rechenleistung P	26, 27, 31
K_{max}	(GA) Maximaler Kostenwert für Fitnessberechnung aus Differenz	105
N	(GD) Kostenwert: Momentanes Optimierungs–Niveau	54
N_0	(GD) Kostenwert: Anfangsniveau	54
N_r	Anzahl der Restriktionsverletzungen	30, 33
P	Rechenleistung	27, 33
P_{max}	Rechenleistung des leistungsstärksten verfügbaren Rechners	27, 33
P_{bPL}	Rechenleistung des Rechnertyps R_{bPL}	75
R	Rechnertyp	29
R^+	Rechnertyp mit nächsthöherer Rechenleistung	29
R^-	Rechnertyp mit nächstniedrigerer Rechenleistung	29
R_{bPL}	Rechnertyp mit bestem Preis–/Leistungsverhältnis	75
T	(SA) „Temperatur“–Wert (TA) Verteuerungsschwelle (Threshold) zur Annahme geänderter Lösungen	40, 46, 47
T_0	(SA, TA) Anfangstemperatur	41, 43, 46, 101
X	Lösungsvektor (Punkt im Parameterraum)	40, 46, 50, 54
X_0	Startpunkt im Parameterraum	40, 46, 50, 54
X_R	(RT) Bisher beste („recorded“) Lösung	50
X_{test}	Zu testender, geänderter Lösungsvektor	40, 46, 50, 54
α	(SA, TA) Abkühlungsfaktor; $0 < \alpha < 1$	41, 43, 47, 101

Variable	Bedeutung	Seite
a_b	Faktor zur Einbeziehung der Kostendifferenz zum nächst*b*illigeren Rechnertyp R^-	29, 33
a_i	Intervallzeit eines Tupels i im Ereignisstrom	15
a_k	Faktor zur Bewertung der Kommunikationskosten ($a_k > 0$)	31, 33
a_l	Faktor zur Gewichtung der lastabhängigen Kosten K_{Last}	28, 29
a_r	Faktor zur Bewertung der Restriktionsverletzungen	30, 33
a_t	Faktor zur Einbeziehung der Kostendifferenz zum nächst*t*eureren Rechnertyp R^+	29, 33
a_v	Faktor für Kostensprung bei virtuellen Rechnern oder Kommunikationselementen; $a_v > 1$	27, 31, 33
α_g	(GD) Faktor für Niveausenkung relativ zur Kostendifferenz; $0 \leq \alpha_g \leq 1$	102
c_{temp}	(SA, TA) Anzahl akzeptierter Änderungen bis „Temperaturausgleich“ angenommen wird; begrenzt durch t_{max}	41, 43, 49, 101
e_k	(ÄF) Parameter zur Bestimmung von p_k (Exponent für n_t)	35
e_v	Exponent für Kostenanstieg bei virtuellen Rechnern oder Kommunikationselementen; $e_v > 1$	27, 31, 33
f_i	(GA) Fitnesswert eines Individuums i	105
k	Index für Kommunikationselemente	33
k_t	(ÄF) Parameter zur Änderung von $t_{a,i}$ und t_p	38
l_k	Leistungsfaktor (Auslastungswert) eines Kommunikationselementes k im Worst–Case	30, 31, 33
l_r	Leistungsfaktor (Auslastungswert) eines Rechners r, $l_r = t_v/t_{R\text{zul}}$ im Worst–Case	22, 23, 28, 29, 31, 33
l_{bPL}	Relative Worst–Case–Auslastung eines Rechners vom Typ R_{bPL}	75
n_t	Anzahl der Tasks im System	35
p_k	(ÄF) Wahrscheinlichkeit zur Entfernung eines nicht–leeren Rechners: $p_k(n_t) = p_{k_0} * 1/{n_t}^{e_k}$	35
p_{k_0}	(ÄF) Parameter zur Bestimmung von p_k (Grundwahrscheinlichkeit)	35
p_n	(ÄF) Wahrscheinlichkeit zur Erzeugung eines *n*euen Rechners	35
r	Index für Rechner	22, 33
$t_{R\text{zul}}$	Maximal zulässige Reaktionszeit (Deadline) einer Task	14, 22, 37
t_S	Spielraum	37
$t_{a,i}$	Zugriffszeit (access time) des Teilnehmers i auf das TDMA–Kommunikationssystem	13, 38, 39
t_{max}	(RT, GD) maximale Anzahl von Änderungsversuchen; $t_{\text{max}} \gg t_{\text{s,max}}$ (SA, TA) maximale Anzahl von Änderungsversuchen auf einem Temperaturniveau; $t_{\text{max}} \gg c_{\text{temp}}$	41, 43, 50, 101, 102, 106, 107
t_p	Gesamtzyklus (Periodenzeit) des TDMA–Bussystems	38, 39
$t_{p,i}$	Periodenzeit des Teilnehmers i für den Zugriff auf das TDMA–Kommunikationssystem	13
$t_{\text{s,max}}$	maximale Anzahl von Änderungsversuchen ohne Kostenverbesserung (RT) bzw. ohne Niveausenkung (GD)	50, 52, 55, 57, 101, 102, 107

Variable	Bedeutung	Seite
t_v	Verarbeitungszeit einer Task	14, 22
z_i	Zykluszeit eines Tupels i im Ereignisstrom	15

Literaturverzeichnis

[ABD$^+$95] N. C. Audsley, A. Burns, R. I. Davis, K. Tindell, und A. J. Wellings. Fixed priority preemptive scheduling: An historical perspective. *Real–Time Systems*, 8(2/3):173–198, März 1995. 2.1

[Abl87] Paul Ablay. Optimieren mit Evolutionsstrategien. *Spektrum der Wissenschaft*, Seiten 104–115, Juli 1987. 2.2

[AD95] P. Altenbernd und C. Ditze. Allocation of Periodic Hard Real–Time Tasks. In *Proc. of the 1995 IFIP/IFAC Workshop on Real–Time Programming*, 1995. . 2.2

[ATB$^+$91] N. C. Audsely, K. Tindell, A. Burns, M. F. Richardson, und A. J. Wellings. Hard Real–Time Scheduling: The Deadline Monotonic Approach. In *Proceedings of the 8th IEEE Workshop on Real–Time Operating Systems and Software*, Seiten 127–132, 1991. 2.1

[BB97] I. Bate und A. Burns. Schedulability analysis of fixed priority real–time systems with offsets. In *Proceedings of the Ninth Euromicro Workshop on Real–Time Systems*, Seiten 153–160, Toledo, Juni 1997. 2.1

[Bou95] Oliver Bouchard. Entwicklung und Implementierung von Optimierungsverfahren zum automatischen Entwurf von Realzeitsystemen. Diplomarbeit, Lehrstuhl für Prozeßrechner, TU München, 1995. A

[Bur93] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. Technischer Bericht YCS214, Department of Computer Science, University of York, 1993. 2.1

[CDM92] Alberto Colorni, Marco Dorigo, und Vittorio Maniezzo. Genetic Algorithms: A New Approach to the Timetable Problem. *NATO ASI Series*, F 82:235–239, 1992. 2.2, 4.1.3, 4.6.3

[DHHM95] Marc Diefenbruch, Elke Heck, Jörg Hintelmann, und Bruno Müller–Clostermann. Performance Evaluation of SDL Systems Adjunct by Queueing Models. In *SDL'95 With MSC in CASE, Proceedings of the Seventh SDL Forum*, Seiten 231–242, Oslo, Norway, September 1995. 2.1

[DIN81] *DIN 66201, Prozeßrechensysteme*. Beuth Verlag, Berlin, 1981. 7

[DS90] Gunter Dueck und Tobias Scheuer. Threshold Accepting: A General Purpose Optimization Algorithm Appearing Superior to Simulated Annealing. *Journal of Computational Physics*, 90(1):161–175, 1990. 2.2, 4.3.1, 4.3.1, A.2

[DSW93] Gunter Dueck, Tobias Scheuer, und Hans-Martin Wallmeier. Toleranzschwelle und Sintflut: neue Ideen zur Optimierung. *Spektrum der Wissenschaft*, Seiten 42–51, März 1993. 2.2, 4.3.1

[DTV90] *dtv Lexikon in 20 Bänden*. Deutscher Taschenbuch Verlag, München, 1990. 4.6.1

[Dud94] *Schülerduden Die Biologie*. Dudenverlag, Mannheim, Leipzig, Wien, Zürich, 1994. 4.6.1, 4.6.1, 4.6.1, 4.6.1

[Due93] Gunter Dueck. New Optimization Heuristics: The Great Deluge Algorithm and the Record–to–Record Travel. *Journal of Computational Physics*, 104(1):86–92, 1993. 2.2, 4.4.1, 4.4.1, 4.5, 4.5.1

[Fär99] Georg Färber. *Prozeßrechentechnik. Manuskript zur Vorlesung*. Lehrstuhl für Realzeit–Computersysteme, TU München, München, 1999. 3.4.2

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison–Wesley, Januar 1989. 2.2, 4.6.2, 4.6.3, 4.6.4.1, 4.6.4.2, 4.6.4.3, 4.6.5

[Gre93] Klaus Gresser. *Echtzeitnachweis ereignisgesteuerter Realzeitsysteme*. Fortschrittsberichte VDI, Reihe 10, Nr. 268. VDI–Verlag, Düsseldorf, 1993. Dissertation, Lehrstuhl für Prozeßrechner, TU München. 1, 2.1, 3.3, 3.4, 3.3, 3.3, 3.5, 3.3, 3.4.2, 3.6, 3.6, 4.1.2.1, 1, 3, B.3, 7, 7

[Gre95] Klaus Gresser. Modellierung des Zeitverhaltens ereignisdiskreter Systeme zum Nachweis der Echtzeitfähigkeit. *at – Automatisierungstechnik*, 43(8):368–373, 1995. 2.1

[Hei94] Jochen Heistermann. *Genetische Algorithmen*. Teubner–Texte zur Informatik, Nr. 9. Teubner, Stuttgart, Leipzig, 1994. 2.2

[Her94] Thomas Herbig. Algorithmen zur Analyse und Bewertung verteilter Echtzeitsysteme. Diplomarbeit, Lehrstuhl für Prozeßrechner, TU München, 1994. A

[Hol73] J.H. Holland. Genetic algorithms and the optimal allocation of trials. *SIAM Journal on Computing*, 2(2):88–105, 1973. . 2.2

[Hol75] J.H. Holland. *Adaption in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, 1975. . 2.2

[Hul92] Martin Hulin. *Evolutionsstrategien zur Schaltungspartitionierung*. Dissertation, TU München, 1992. 2.2, 2.2

[Ing89] Lester Ingber. Very fast simulated re–annealing. *Mathl. Comput. Modelling*, 12:967–973, 1989. 2.2

[Ing93] Lester Ingber. Adaptive Simulated Annealing (ASA). *[ftp.caltech.edu: /pub /ingber/asa.Z]*, 1993. 2.2

[IR92] Lester Ingber und Bruce Rosen. Genetic Algorithms and Very Fast Simulated Reannealing: A Comparison. *Mathematical and Computer Modelling*, 16(11):87–110, 1992. 2.2

[ITU94] ITU–T. *ITU–T Recommendation Z.100: CCITT Specification and Description Language (SDL)*, Juni 1994. 2.1

[KGV83] S. Kirkpatrick, C. D. Gelatt, Jr., und M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, Mai 1983. 2.2, 4.2.1, 4.2.1, 4.2.1, 4.2.1, A.1

[Kop86] Hermann Kopetz. Scheduling in Distributed Real Time Systems. In *Proceedings of the Advanced Seminar on Real-Time Local Area Networks*, Seiten 105–126, INRIA, Rocquencourt, France, 1986. 2.1

[Len76] Widukind Lenz. *Medizinische Genetik*. Georg Thieme Verlag, Stuttgart, 3. Auflage, 1976. 4.6.1, 4.6.1

[LL73] C. L. Liu und James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard–Real–Time Environment. *Journal of the ACM*, 20(1):46–61, 1973. 2.1

[LMM98] S. Lauzac, R. Melhem, und D. Mossé. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *Proceedings of the 10th Euromicro Workshop on Real–Time Systems*, Berlin, Juni 1998. 2.1

[LW82] J. Y. T. Leung und J. Whitehead. On the complexity of fixed–priority scheduling of periodic real–time tasks. *Performance Evaluation*, 2(4):237–250, Dezember 1982. 2.1

[MA98] Y. Manabe und S. Aoyagi. A feasibility decision algorithm for rate monotonic and deadline monotonic scheduling. *Real–Time Systems*, 14(2):171–181, März 1998. 2.1

[MRTT53] N. Metropolis, A. W. Rosenbluth, A. H. Teller, und E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953. 4.2.1

[Mus95] Oliver Muskat. Benutzerschnittstelle für ein Tool zur Optimierung verteilter Echtzeitsysteme. Diplomarbeit, Lehrstuhl für Prozeßrechner, TU München, 1995. A

[Mut82] C. Muth. Einführung in die Evolutionsstrategie. *Regelungstechnik*, 30:297–303, 1982. TODO: besorgen! 2.2

[Ott94] Thomas Otto. Reiselust. Travelling Salesman — eine neue Strategie für eine alte Aufgabe. *c't*, Seiten 188–194, Januar 1994. 2.2

[Pet95] Stefan Petters. Genetische Algorithmen zur Entwurfsoptimierung von Realzeitsystemen. Diplomarbeit, Lehrstuhl für Prozeßrechner, TU München, 1995. A

[Rec73] Ingo Rechenberg. *Evolutionsstrategie; Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Problemata. Frommann–Holzboog, Stuttgart, 1973. 2.2, 4.1.3.4

[Sch76] Hans Paul Schwefel. *Numerische Optimierung von Computermodellen mittels der Evolutionsstrategie*. Birkhäuser Verlag, Basel und Stuttgart, 1976. 2.2

[Sch93] Markus Schwarz. *Ein massiv paralleles Rechnersystem für die Emulation künstlicher neuronaler Netze und genetischer Algorithmen mit Anwendungen in der Bildmustererkennung*. Fortschrittsberichte VDI, Reihe 10, Nr. 236. VDI–Verlag, Düsseldorf, 1993. Dissertation. 2.2, 4.6.4.3

[Sec88] Carl Sechen. *VLSI Placement and Global Routing Using Simulated Annealing*. The Kluwer international series in engineering and computer science, Nr. 54. Kluwer Academic Publishers, Boston, 1988. 4.2.1, 4.2.1, 4.2.1

[SHF94] Eberhard Schöneburg, Frank Heinzmann, und Sven Feddersen. *Genetische Algorithmen und Evolutionsstrategien: Eine Einführung in Theorie und Praxis der simulierten Evolution*. Addison–Wesley, Bonn, Paris, Reading Mass., 1994. 2.2

[TB91] E-G. Talbi und P. Bessière. A Parallel Genetic Algorithm for the Graph Partitioning Problem. *[ftp.imag.fr: /pub/SYMPA/talbi.ACM91.e.ps.Z]*, 1991. 2.2, 4.6.5

[TBW92] K. W. Tindell, A. Burns, und A. J. Wellings. Allocating Hard Real–Time Tasks: An NP–Hard Problem Made Easy. *Journal of Real–Time Systems*, 4(2):145–165, 1992. 2.1, 2.2, 4.2.1, 4.2.1

[Tin94] K. Tindell. Adding time–offsets to schedulability analysis. Technischer Bericht YCS221, Department of Computer Science, University of York, 1994. 2.1

[Tri94] Michael Triller. *Verbesserung des Echtzeitverhaltens von Mehrrechnersystemen durch Prozeßmigration*. Fortschrittsberichte VDI, Reihe 10, Nr. 279. VDI–Verlag, Düsseldorf, 1994. Dissertation, Lehrstuhl für Prozeßrechner, TU München. 2.1

[Wal94] Dr. Norbert Waleschkowski. Genetische Algorithmen zur Lösung komplexer Optimierungsaufgaben. *Marktreport 1994 — Intelligente Software–Technologien*, Seiten 72–77, 1994. 2.2, 4.6.4.2

[Wan99] Shuhua Wang. *Specification, Allocation and Schedulability–Analysis for Fixed–Priority Hard Real–Time Systems*. Dissertation, Lehrstuhl für Realzeit–Computersysteme, TU München, 1999. 2.1

[Wet94] Michael Wetzel. Modellierung eines verteilten Realzeitsystems und Analyse eines Feldbusses zum Nachweis der Echtzeitfähigkeit am Beispiel eines Hubschrauber–Trainers. Diplomarbeit, Lehrstuhl für Prozeßrechner, TU München, 1994. A

[WF98] Shuhua Wang und Georg Färber. Schedulability Analysis for Communicating Tasks on a Multiprocessor System. In *Proc. 23th IFAC/IFIP Workshop on Real Time Programming (WRTP'98)*, Seiten 25–30, Shantou, China, Juni 23–25 1998. IFAC/IFIP. 2.1

[WF99] Shuhua Wang und Georg Färber. On the Schedulability Analysis for Distributed Real–Time Systems. In *Proc. Joint 24th IFAC/IFIP Workshop on Real Time Programming and Third International Workshop on Active and Real–Time Database Systems (WRTP'99, ARTDB'99)*, Saarland, Germany, May 30 – June 2 1999. IFAC/IFIP. 2.1

[WLL88] D. F. Wong, H. W. Leong, und C. L. Liu. *Simulated Annealing for VLSI Design*. Kluwer Academic Publishers, Boston, 1988. 4.2.1

[ZA95] Dieter Zöbel und Wolfgang Albrecht. *Echtzeitsysteme. Grundlagen und Techniken*. International Thomson Publishing, Bonn, Albany, Attenkirchen, 1. Auflage, 1995. 1