DISSERTATION

Codegenerierung für Signalprozessoren mit Hilfe genetischer Algorithmen

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

o.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Mecklenbräuker
E389
Institut für Nachrichtentechnik und Hochfrequenztechnik

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

Dipl.-Ing. Stefan Fröhlich
Matrikel Nr. 9025800
Millergasse 17/19, A-1060 Wien

Wien, 27. April 2001

Kurzfassung

Mit der vorliegenden Arbeit wird ein automatischer Codegenerator vorgestellt, der Assemblercode für digitale Signalprozessoren erzeugt. Dieser Codegenerator basiert auf evolutionären Optimierungsstrategien und ist in der Lage, hochoptimierte Programme zu generieren.

In der digitalen Signalverarbeitung werden bevorzugt Gleichungssätze oder Blockschaltbilder verwendet, um die rechenintensiven Kernalgorithmen zu beschreiben. Dementsprechend wird diese Darstellungsform auch als Eingabe für den hier vorgestellten Codegenerator verwendet. Es handelt sich also nicht um einen klassischen Compiler für eine der bekannten Hochsprachen, sondern um eine Anwendung, die für eine spezielle Klasse von Problemen optimiert ist.

Der Codegenerator verbindet bereits bekannte Verfahren der automatischen Codegenerierung mit etablierten Suchstrategien aus anderen Bereichen, um insgesamt eine neue, bisher noch nicht erreichte Qualität der Ergebnisse zu erzielen.

Den Rahmen des Programmes bildet ein genetischer Optimierungsalgorithmus, dessen Verhalten über mehrere Parameter kontrollierbar ist. Mittels genetischer Operatoren (Selektion, Crossover und Mutation) wird versucht, Teilblöcke des zu erzeugenden Programmes in der richtigen Reihenfolge und mit den richtigen Randbedingungen zu plazieren. Der Code für diese Teilblöcke wird mit einer Kombination mehrerer bereits etablierter Optimierungsstrategien erzeugt. Dabei handelt es sich um Baumzerlegung und um die Befehlsauswahl mittels Trellisalgorithmen. Zusätzlich dazu werden Registerallokation sowie Kompaktierung des erzeugten sequentiellen Assemblercodes mit einem list scheduling Verfahren durchgeführt.

In der Arbeit wird gezeigt, daß die vorgeschlagene Kombination verschiedener Algorithmen geeignet ist, für gängige Aufgaben der digitalen Signalverarbeitung hochoptimierten, meist sogar optimalen Code zu erzeugen. An Hand eines konkreten Beispiels werden die einzelnen Parameter des genetischen Algorithmus detailliert betrachtet und Richtlinien für die Wahl geeigneter Werte gegeben. Weiters werden Untersuchungen über die notwendige Laufzeit in Abhängigkeit von der gewünschten Qualität der Ergebnisses durchgeführt. Abschließend wird an einigen typischen Aufgabenstellungen die Qualität des Programmes in Abhängigkeit von der Art und Größe des untersuchten Problems betrachtet. Es zeigt sich dabei, daß die meisten Aufgaben sehr gut für diese Art der Optimierung geeignet sind, es aber auch Ausnahmen hierzu gibt, die an Hand spezifischer Merkmale erkennbar sind.

Abstract

This thesis introduces an automatic code generator providing assembly code for digital signal processors. It is based on evolutionary optimization strategies and capable of generating highly optimized programs.

In the field of digital signal processing it is common practice to use a set of equations or even data flow graphs to describe the operation of the core algorithms. This is why the input files for the proposed compiler are also formulated as a set of equations. This compiler is therefore not a classical high level language compiler, but a specialized application to get very high quality solutions for a well defined set of problems.

The code generator combines well known optimization methods already used in the field of automatic code generation together with optimization strategies from other fields to achieve a new quality of results.

The framework of the compiler is built by a genetic algorithm, which may be controlled via a set of parameters. Genetic operators (selection, crossover and mutation) are used to place building blocks of the compiled program in the correct order using the correct boundary conditions. The code for this building blocks is generated by a combination of well known optimization strategies. These are the tree decomposition and the instruction selection, which is done with trellis algorithms. Additionally, register allocation and compaction of the sequential assembly code are done using a list scheduling procedure.

This thesis states that the proposed combination of optimization strategies is well suited for typical applications in the field of digital signal processing. Highly optimized code is generated for a number of standard problems, in many cases the solutions are optimal. By using a specific example, the different parameters of the genetic part are examined and guidelines for reasonable values are given. Furthermore the memory requirements and runtime performance are evaluated with respect to the desired quality of the solution. Finally a number of typical exercises is evaluated to see which problems are suited well for this compiler, but also to recognize possible weaknesses.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In der digitalen Signalverarbeitung werden hohe Anforderungen an die Ausführungsgeschwindigkeit von Algorithmen gestellt (Echtzeitabarbeitung), besonders in modernen Anwendungen wie beispielsweise in der Mobilkommunikation oder in Multimediasystemen.

Für die Implementierung der teilweise sehr rechenaufwendigen Algorithmen werden Spezialprozessoren mit darauf abgestimmter Architektur verwendet, sogenannte Signalprozessoren (DSPs). Diese zeichnen sich im allgemeinen durch einen im Vergleich zu CISC-Prozessoren sehr kleinen Befehlssatz, durch rasche arithmetische Einheiten und spezielle Adreßrecheneinheiten aus. Auch wenn der Geschwindigkeitsvorteil durch die Entwicklung der traditionellen CPUs zum Teil bereits wettgemacht worden ist, so sind DSPs durch ihren Kostenvorteil und den viel geringeren Energiebedarf nach wie vor aus industriellen Anwendungen nicht wegzudenken.

Die Programmierung von DSPs ist vergleichsweise schwierig, weil einerseits hohe Anforderungen an die Effizienz der implementierten Algorithmen gestellt werden und andererseits die Hardwarearchitekturen von Signalprozessoren spezielle Architekturmerkmale aufweisen, die durch handelsübliche Hochsprachencompiler nicht immer erfüllt werden können ([32]). Daher wird auch heute noch in der industriellen Praxis Assemblercode für Signalverarbeitungsaufgaben händisch optimiert, der Unterschied zu den Resultaten typischer C-Compiler liegt etwa bei einem Faktor 10.

Da das händische Programmieren zeitaufwendig und fehleranfällig, insgesamt also sehr teuer ist, sind verschiedene Ansätze entwickelt worden, hochoptimierende Compiler zu erzeugen. Bei dieser Aufgabe besteht das Hauptproblem darin, daß die gesuchte Lösung in einem extrem großen Lösungsraum eingebettet ist (Codegenerierung ist bereits für einfache Architekturen ein NP-vollständiges Problem, [2]), wodurch der Einsatz von Heuristiken notwendig wird. Ein optimales Ergebnis kann nur für kleine Teilbereiche, nicht aber für das gesamte Programm garantiert werden, beispielsweise durch das Verfahren aus [1], mit dem sogenannte expression trees in den kürzestmöglichen Maschinencode transformiert werden können.

Aus dieser Motivation heraus haben sich in der Vergangenheit verschiedene Algorithmen für die unterschiedlichen Teilaspekte der automatischen Codegenerierung etabliert ([26], [27], [29], [21]), die aber alle in sich geschlossene Systeme sind. Versucht man, mehrere solcher Programme nacheinander anzuwenden, so stellt man fest, daß ein sogenanntes Phasenkopplungsproblem auftritt, die Ergebnisse der einzelnen Optimierungsschritte sind voneinander abhängig. Dadurch wird der erzeugte Code suboptimal. Verfahren zur Minderung dieser Probleme werden in [18] und [3] vorgestellt, bieten aber nur eingeschränkte Optimierungsmöglichkeiten an (beispielsweise besteht eine Beschränkung der Wiederverwendbarkeit von Ergebniswerten in nachfolgenden Ausdrücken).

Andere Verfahren ([5], [17]) liefern vergleichbare Ergebnisse, nehmen jedoch keine Rücksicht auf die bei typischen DSPs mögliche Parallelisierbarkeit, sondern sind auf die Erzeugung von straight line code beschränkt. Eine nachfolgende Parallelisierung erzeugt erneut Phasenkopplungsprobleme.

In der vorliegenden Arbeit werden verschiedene etablierte Algorithmen für Teilaufgaben herangezogen, erweitert, an das Gesamtkonzept angepaßt und anschließend gemeinsam in den Kern eines evolutionären Algorithmus eingebaut. Durch dieses Einbetten wird versucht, das Phasenkopplungsproblem zu umgehen und eine globale Optimierung des erzeugten Codes zu erreichen.

1.1 Grundlegende Begriffe

1.1.1 Datenflußgraphen

Der in dieser Arbeit vorgestellte Compiler verwendet als Eingabe kein in einer Hochsprache (wie es z.B. die Programmiersprachen C oder FORTRAN wären) erstelltes Programm, sondern Gleichungen, mit denen das Problem algorithmisch beschrieben wird. So ein Gleichungssatz entspricht jeweils einem Datenflußgraphen (siehe auch Abbildung 1.1). Ein Datenflußgraph zeichnet sich dadurch aus, daß er die Abhängigkeit der einzelnen Operationen, sowie die Ein- und Ausgänge des Systems beschreibt, aber keine Festlegung über Details, wie die genaue Reihenfolge der Befehlsabarbeitung trifft. Dadurch ergibt sich ein extrem großer Lösungsraum mit einem entsprechend guten Potential zur Optimierung.

Nicht alle Algorithmen lassen sich vollständig als Datenflußgraph darstellen. Das ist insbesondere dann der Fall, wenn der Algorithmus Kontrollstrukturen, wie beispielsweise bedingte Verzweigungen enthält. In diesem Fall ist es aber möglich und sinnvoll, die Blöcke innerhalb der Kontrollstrukturen zu optimieren und anschließend in das konventionell erstellte Codegerüst einzubauen. Da der arithmetische Teil des Algorithmus in den meisten Fällen den weitaus größten Teil der Rechenzeit benötigt, hat dieses Vorgehen keinen merkenswerten Einfluß auf die Gesamtperformance des erzeugten Codes.

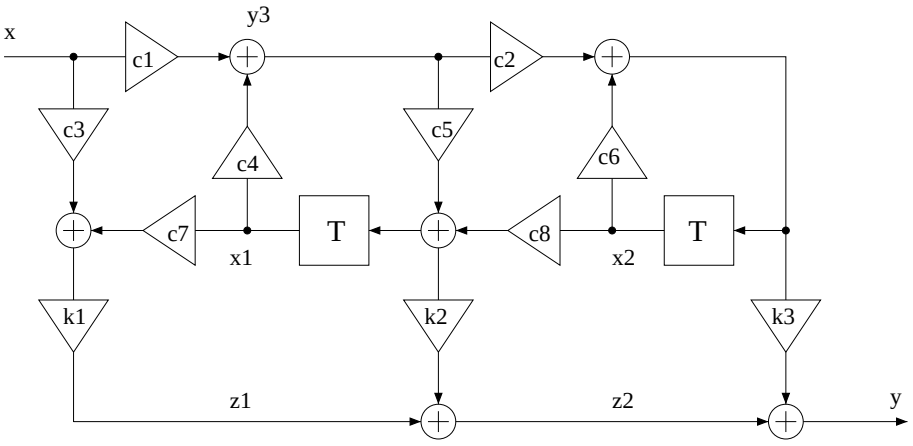

Abbildung 1.1: Datenflußgraph eines Lattice-Filters 2. Ordnung

1.1.2 Synchrone Datenflußgraphen

Ein synchroner Datenflußgraph zeichnet sich dadurch aus, daß die Anzahl der an einer Kante produzierten oder konsumierten Datenwerte konstant (aber nicht notwendigerweise gleich groß) ist. Immer, wenn am Eingang eines Knotens ausreichend viele Datenwerte zur Verfügung stehen, ist dieser Knoten berechenbar. Falls der erzeugende Knoten weniger Datenwerte je Aufruf generiert, als benötigt werden, um den konsumierenden Knoten ausführen zu können, so muß der erste Knoten im Algorithmus entsprechend öfter aufgerufen werden (der umgekehrte Fall verläuft analog dazu).

1.1.3 Homogene Datenflußgraphen

Ein homogener Datenflußgraph ist ein Spezialfall eines synchronen Datenflußgraphen, bei dem jeder Eingang exakt einen Datenwert je Aufruf konsumiert und jeder Ausgang exakt einen Datenwert je Aufruf produziert. Wenn ein homogener Datenflußgraph keine Schleifen ohne Verzögerungselement enthält (in diesem Fall wäre der Graph nicht ausführbar), so läßt sich immer eine statische Abarbeitungsreihenfolge angeben, in der jeder Knoten exakt ein Mal aufgerufen wird. Weiters kann man eine Obergrenze für die notwendigen Pufferspeicher angeben, die maximal um 1 größer ist, als die Anzahl der in einer einzelnen Schleife enthaltenen Verzögerungselemente.

1.1.4 Digitale Signalprozessoren

Diese Klasse von Prozessoren wurde speziell für die effiziente Bearbeitung von Algorithmen in der digitalen Signalverarbeitung entwickelt. Dabei handelt es sich sehr oft um zyklische Algorithmen, was bedeutet, daß am Ende des Pro-

grammablaufes entweder unmittelbar oder nach einem Interrupt wieder mit der Ausführung des exakt gleichen Programmes begonnen wird. Der Prozessor läuft also in einer Endlosschleife und kann damit beliebig lange Datenströme verarbeiten.

Digitale Signalprozessoren basieren fast ausschließlich auf Harvard-Architekturen, das bedeutet, daß sie mindestens zwei getrennte Speicherbänke für Programm- und Datenspeicher besitzen, oft können auch die Daten auf zwei verschiedene Speicherbänke verteilt werden, um einen Geschwindigkeitsgewinn zu erzielen. Außerdem sind die Recheneinheiten für die typisch zur Anwendung kommenden Algorithmen optimiert: die ALU ist in der Lage, sämtliche Berechnungen in einem Taktzyklus abzuarbeiten (hierfür ist meist eine eigene Einheit zur schnellen Multiplikation notwendig), es gibt zusammengesetzte Befehle wie die MAC-Instruktion, die bei General Purpose Prozessoren nicht vorkommen und oft ist auch die Möglichkeit vorhanden, Ergebnisse automatisch zu skalieren.

Weiters ist zur effizienten Kommunikation mit dem Hauptspeicher die Möglichkeit vorhanden, parallel zum Betrieb der ALU Daten aus dem Speicher zu holen, oder sie dort abzulegen. Je nach Prozessor können auf diese Art bei geschickter Programmierung bis zu drei Assemblerbefehle gleichzeitig abgearbeitet werden.

Praktisch alle Signalprozessoren haben neben der eigentlichen Recheneinheit, der ALU, auch noch ein oder zwei weitere, kleinere Recheneinheiten, die als Adreßrecheneinheiten bezeichnet werden. Mit ihnen ist ein sehr effizienter Zugriff auf den Hauptspeicher möglich, da sie Zeigervariablen parallel zum ablaufenden Programm berechnen können. Ein zeitaufwendiges Nachladen von Adreßzeigern kann daher bei entsprechend sorgfältiger Anordnung der Variablen im Speicher oftmals vermieden werden [29].

Kapitel 2

Automatische Codegenerierung

2.1 Übersicht

Die automatische Codegenerierung beschäftigt sich mit der Frage, wie man aus einem gegebenen Datenflußgraph möglichst effizienten Assemblercode für eine bestimmte Zielarchitektur erzeugen kann. In dieser Arbeit beschränken wir uns auf *homogene* Datenflußgraphen, was bedeutet, daß innerhalb eines Programmablaufes jeder Knoten exakt einmal ausgeführt wird. Bei der Codegenerierung müssen die folgenden Probleme bearbeitet werden:

- Auswahl von Auslagerungspunkten, Baumzerlegung
- Zuordnen von Variablen zu bestimmten Speicherbänken
- Befehlsauswahl der arithmetischen Befehle
- Festlegen der Abarbeitungsreihenfolge
- Auswahl der verwendeten Register
- Parallelisierung des sequentiellen Codes

Diese Schritte sind stark voneinander abhängig und müssen, um ein optimales Assemblerprogramm zu erhalten, gemeinsam optimiert werden. Optimalität bezieht sich dabei (sofern nicht ausdrücklich andere Kriterien vorgegeben sind) immer auf die Ausführungszeit des erzeugten Programmes.

Bereits die Festlegung der Abarbeitungsreihenfolge ist im Fall von beliebigen Datenflußgraphen ein NP-vollständiges Problem [11], es existiert also keine bekannte effiziente Lösungsmöglichkeit. Aus diesem Grund arbeitet die Codegenerierung nicht unmittelbar mit Datenflußgraphen, sondern zunächst mit Bäumen, die anschließend zum gesamten Graphen zusammengesetzt werden.

In den folgenden Abschnitten werden die einzelnen Problemstellungen, die in der automatischen Codegenerierung auftreten, zusammengefaßt und deren bestehende Lösungen kurz vorgestellt.

2.2 Zerlegung des Datenflußgraphen

Die Zerlegung in Teilbäume (meist kurz als Baumzerlegung bezeichnet) teilt einen (homogenen) Datenflußgraphen in einzelne Bäume (sogenannte Expression Trees) auf. Im allgemeinen ist es nicht möglich, einen Graphen mit einem einzigen Baum darzustellen, da insbesondere bei Verzweigungen (ein berechneter Wert wird an mehreren Stellen im Graphen für weitere arithmetische Operationen verwendet) eine Aufteilung in mehrere Bäume unvermeidbar ist.

Da der Trellisalgorithmus optimalen Code nur innerhalb eines Teilbaumes generiert, ist es ratsam, möglichst große Teilbäume zu erzeugen. Dafür gibt es verschiedene Ansätze.

2.2.1 Klassisches Zerlegungsverfahren

In der klassischen Methode wird bei jeder Verzweigung aufgetrennt und in Teilbäume gespalten. In Abbildung 2.1 findet sich ein Beispiel für ein Lattice-Filter 2.Ordnung. Man erkennt, daß in diesem Fall drei Teilbäume notwendig sind. Diese beeinflussen sich gegenseitig nicht und können daher getrennt voneinander berechnet werden. Es ist lediglich auf die Einhaltung der durch die gegenseitigen Abhängigkeiten bestimmte Reihenfolge zu achten, so muß beispielsweise der Baum, der den Wert z_1 berechnet vor dem Baum, der diesen Wert verwendet, abgearbeitet werden. Der Vorteil dieser Methode liegt darin, daß es sich um ein deterministisches Verfahren handelt und alle Freiheitsgrade innerhalb der Bäume erhalten bleiben. Nachteilig wirkt sich vor allem aus, daß bei Algorithmen, die stark verzweigt sind, relativ viele und kleine Bäume erzeugt werden. Dadurch ist der Gesamtcode, der sich aus den Teilcodes der einzelnen Bäume zusammensetzt, oft suboptimal.

2.2.2 Optimierte Zerlegung mit Randbedingungen

Um die im vorigen Abschnitt als problematisch erkannte hohe Anzahl an Teilbäumen zu verringern, wurde ein verbessertes Verfahren entwickelt, das mit Randbedingungen arbeitet. Die resultierenden Bäume werden oft auch als *constrained expression trees* bezeichnet [14]. Diese Methode trennt den Graphen bei Verzweigungen nicht in Teilbäume auf, sondern läßt eine willkürlich ausgewählte Kante des betreffenden Knotens im Graphen, während die andere Kante gelöscht und als innere Abhängigkeit vermerkt wird. Auf diese Art kann man die Anzahl der notwendigen Teilbäume deutlich reduzieren, man erhält größere Bäume, die besser optimierbar sind. Nachteilig wirkt sich aus, daß ein Baum nun im allgemeinen innere Abhängigkeiten enthält, die sich auf die Abarbeitungsreihenfolge der Knoten auswirken. Es stehen also nicht mehr alle Freiheitsgrade innerhalb

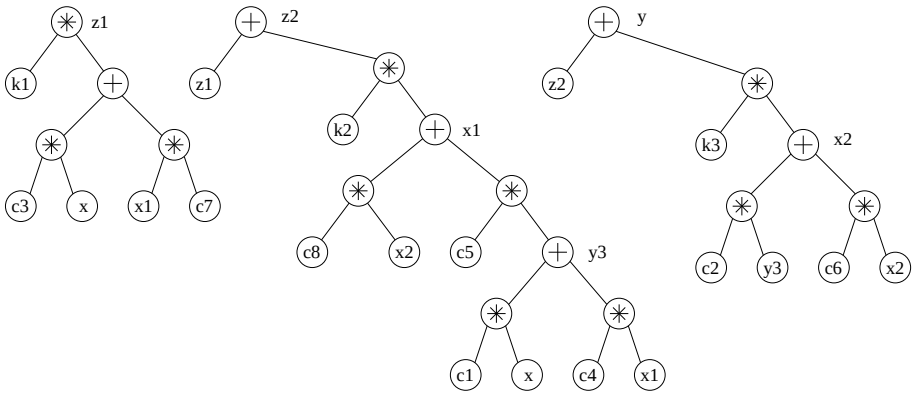

Abbildung 2.1: Teilbäume für ein Lattice-Filter 2. Ordnung

des Baumes zur Verfügung, wodurch suboptimaler Code entstehen kann. Um diese Fehlentscheidungen zu verhindern, kann man alle möglichen Entscheidungen durch Permutieren ausprobieren. Dadurch verliert man allerdings einen der Hauptvorteile des Trellisverfahrens, nämlich die linear mit der Codegröße wachsende Bearbeitungszeit.

Abbildung 2.2 zeigt die Baumzerlegung für das Lattice-Filter 2. Ordnung, das in Abbildung 1.1 dargestellt worden ist. Durch Verbinden der drei Teilbäume erhält man einen einzigen, großen Baum für den gesamten Graphen. Nachteilig wirken sich die beiden Abhängigkeiten aus, die durch das Verbinden der drei Bäume entstanden sind. Diese Abhängigkeiten stellen sicher, daß die Variablen $x1$, $x2$ und $y3$ zum Zeitpunkt ihrer Verwendung bereits berechnet worden sind.

2.3 Trellisdiagramme

Trellisdiagramme werden verwendet, um innerhalb eines abgeschlossenen Teilbaumes optimalen Code zu erzeugen. Sie beschreiben die Übergänge zwischen den einzelnen Zuständen des Prozessors, die beim Abarbeiten des Programmes durchlaufen werden. Der Zustand eines Prozessors ist eine vollständige Beschreibung der belegten Ressourcen, es wird also für jede Registerbank angegeben, wie viele Register belegt und wie viele noch verfügbar sind. Jede Instruktion kann durch einen Übergang von einem oder mehreren Quelldiagrammen zu einem Zieldiagramm beschrieben werden. Dabei sind für die Berechnung der Quelloperanden verwendete Register entsprechend als belegt zu kennzeichnen.

2.3.1 Statische Trellisdiagramme

Statische Trellisdiagramme enthalten für jeden Operanden alle möglichen Zustände, die sich durch Kombination der Belegung der verschiedenen Registerbänke ergeben, selbst dann, wenn eine Registerbank von einem Befehl gar nicht

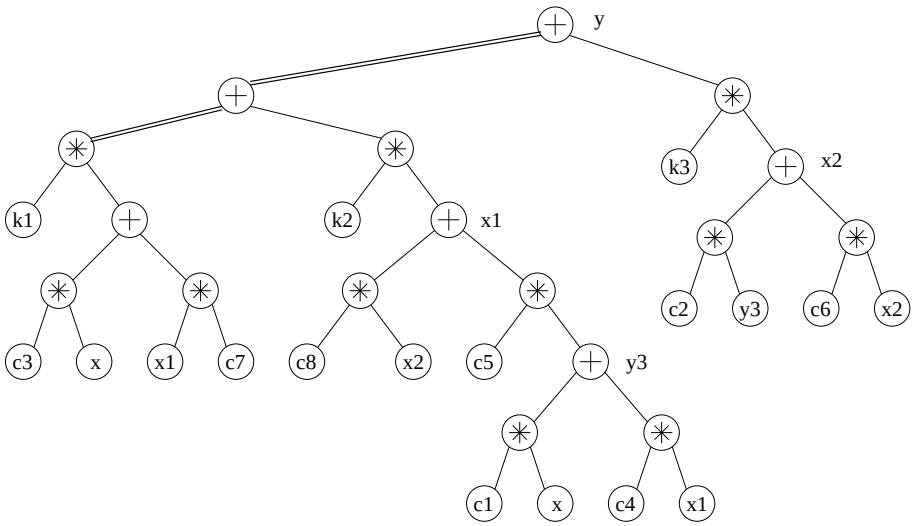

Abbildung 2.2: Baumzerlegung mit Randbedingungen

verwendet werden kann. Dadurch werden diese Diagramme speziell bei heterogenen Prozessorstrukturen sehr groß, belasten den verfügbaren Hauptspeicher des PCs bzw. der Workstation enorm und machen die Optimierung laufzeitmäßig ineffizient oder in Extremfällen sogar völlig unmöglich. Durch die starre Struktur ist die Implementierung von Mehrfachbefehlen (wie beispielsweise der bei Signalprozessoren häufig anzutreffende MAC-Instruktion) schwierig und nur über Umwege möglich. Der Vorteil statischer Trellisdiagramme liegt in der einfacheren Handhabung und der höheren Geschwindigkeit ihrer Abarbeitung. Dadurch, daß alle Diagramme die gleiche Struktur haben, können sie bereits vorab berechnet werden, während des Programmlaufes muß nur noch auf die bereitgestellten Schablonen zurückgegriffen werden. Durch die komplette Auflistung aller Zustände ist ebenfalls eine deutlich einfachere programmiertechnische Umsetzung gewährleistet, die die Verarbeitungsgeschwindigkeit erhöht.

In Abbildung 2.3 ist das statische Trellisdiagramm für eine Ladeanweisung eines sehr einfachen Prozessors dargestellt. Dieser Prozessor besitzt lediglich eine einzige Registerbank mit den zwei Registern $A0$ und $A1$, die gleichwertig in allen Befehlen verwendbar sind. Das Trellisdiagramm, das von unten nach oben zu lesen ist, beschreibt den Übergang vom Prozessorzustand *vor* der Befehlsausführung in den Zustand *nach* der Ausführung. Im Fall des Beispielprozessors kann sich der Operand entweder im Speicher befinden (Zustand m), oder in einem der Rechenregister. Hier ist zunächst gleichgültig, um welches der beiden Register es sich beim ablauffähigen Programm tatsächlich handeln wird (da die Register ja ohnehin beliebig gegeneinander austauschbar sind), hingegen ist die Information wichtig, ob das jeweils andere Register gerade einen Datenwert gespeichert hält oder frei verfügbar ist. Im Trellisdiagramm wird daher als Kennzeichnung

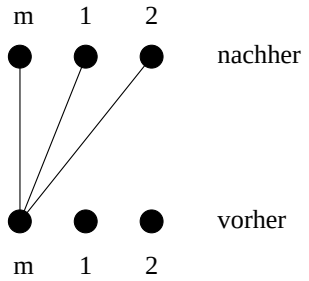

Abbildung 2.3: Trellisdiagramm *Laden*

beim Status die Anzahl der noch verfügbaren Register aufgetragen. In unserem Beispiel gibt es nur die Zustände *1* und *2*, da zumindestens eines der beiden Register frei sein muß, um den eben geladenen Wert aufnehmen zu können.

Der Quelloperand einer Ladeinstruktion befindet sich immer im Hauptspeicher, daher ist m der einzig gültige Ausgangszustand. Für das Trellisdiagramm ergeben sich daher 3 mögliche Zustandsübergänge mit den folgenden Bedeutungen:

- $m \rightarrow 2$
  Ein Datenwert wird aus dem Hauptspeicher in eines der beiden Register geladen. Dabei stehen beide der beiden Register zur Verfügung, eines enthält anschließend den Datenwert, das andere bleibt ungenützt. Der Kostenwert entspricht der Anzahl der für die Abarbeitung notwendigen Taktzyklen, bei den meisten Signalprozessoren wird dies 1 sein.

- $m \rightarrow 1$
  Ein Datenwert wird aus dem Hauptspeicher in eines der beiden Register geladen, wobei das andere Register mit einem Zwischenergebnis einer früheren Berechnung belegt ist. Auch hier ist der Kostenwert üblicherweise 1.

- $m \rightarrow m$
  Der Operand bleibt im Datenspeicher, beide Register können mit Zwischenergebnissen belegt sein, falls erforderlich. Es wird kein Assemblerbefehl ausgeführt, dieser Zustandsübergang verursacht daher auch *keinen* Beitrag in der Kostenfunktion.

Abbildung 2.4 zeigt das Trellisdiagramm für eine Speicheranweisung des gleichen Beispielprozessors. Es handelt sich um das genaue Gegenstück der Ladeanweisung, von jedem Zustand aus existiert eine Verbindung zum Zielzustand m, da das Ergebnis eines Speicherbefehles immer im Hauptspeicher steht.

Die Zustandsübergänge der Negation ($A = -A$) werden in Abbildung 2.5 gezeigt. Wie man sieht, wird der Operand aus einem Register geladen und auch wieder in ein Register geschrieben, daher führen keine Kanten von und zu den

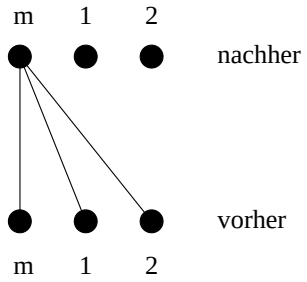

Abbildung 2.4: Trellisdiagramm *Speichern*

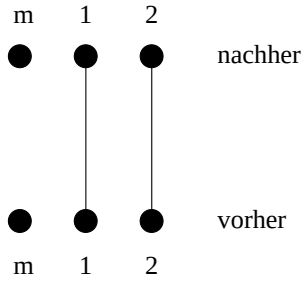

Abbildung 2.5: Trellisdiagramm *Negation*

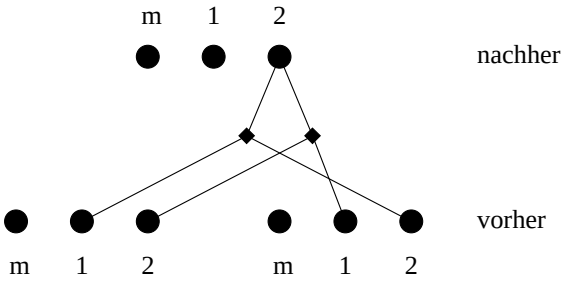

Abbildung 2.6: Trellisdiagramm *Addieren*

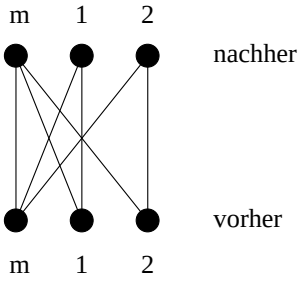

Abbildung 2.7: Trellisdiagramm *Datentransfer*

m-Zuständen. Die Anzahl der verfügbaren Register wird durch den Negationsbefehl nicht beeinflußt, deshalb wird jeder Quellzustand mit exakt dem gleichen Zielzustand verknüpft.

Die Addition, die in Abbildung 2.6 dargestellt wird, benötigt zwei Quelloperanden, daher wird ein binäres Trellisdiagramm verwendet, in dem für jeden der beiden Operanden ein entsprechender Zustand angegeben werden kann. Man erhält dadurch einzelne Zustandstripel, die gleichzeitig die Berechnungsreihenfolge der Operanden festlegen. Wird beispielsweise das linke Zustandstripel ausgewählt, so steht für den ersten Operanden nur ein freies Register zur Verfügung, während bei der Berechnung des zweiten Operanden noch auf beide Register zurückgegriffen werden kann. Daraus ergibt sich implizit, daß zunächst der zweite (rechte) Operand berechnet werden und als Zwischenergebnis im Register behalten werden muß. Anschließend erfolgt die Berechnung des ersten (linken) Operanden, für die jetzt tatsächlich nur noch ein freies Register zur Verfügung steht. Nach erfolgter Addition wird das gespeicherte Zwischenresultat nicht mehr benötigt, das Register daher freigegeben. Deshalb ist der Zielzustand auf alle Fälle 2.

Da man im allgemeinen nicht davon ausgehen kann, daß sich ein Trellisbaum ohne Zwischenspeichern von Ergebnissen im Hauptspeicher berechnen läßt, die exakten Positionen dieser Auslagerungsoperationen aber zunächst unbekannt

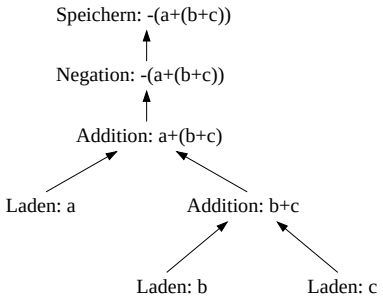

Abbildung 2.8: Expression tree

sind, werden zwischen den arithmetischen Befehlen jeweils 2 Transferdiagramme eingefügt. Transferdiagramme setzen sich aus einem Lade- und einem Speicherdiagramm zusammen und enthalten außerdem noch Verbindungen von jedem Zustand zu sich selber, die keinen Beitrag zur Kostenfunktion liefern und auch keinen Assemblerbefehl bedingen (NOP-Instruktionen, siehe Abbildung 2.7).

Zur Berechnung eines arithmetischen Ausdrucks werden die Trellisdiagramme für die einzelnen arithmetischen Operationen (unter Berücksichtigung der Transferdiagramme) untereinandergestellt. Untersucht man beispielsweise den Ausdruck $-(a + (b + c))$, so erhält man einen Baum, wie in Abbildung 2.8 dargestellt. Der Trellisbaum ergibt sich daraus, indem man die einzelnen Operationen durch ihre Trellisdiagramme ersetzt und an den notwendigen Stellen Movediagramme einfügt. Für den hier untersuchten Ausdruck und den in diesem Kapitel verwendeten Beispielprozessor ist ein Trellisbaum in Abbildung 2.9 dargestellt.

Auf diesen Trellisbaum wird nun der in [28] angegebene Optimierungsalgorithmus angewendet, der auf dem Prinzip der dynamischen Programmierung beruht und in der Übertragungstechnik auch unter dem Namen *Viterbialgorithmus* bekannt ist. Mit diesem Algorithmus ist es möglich, in linearem Zeitaufwand bezüglich der Anzahl der Knoten im Trellisbaum den *reduzierten Trellisbaum* zu erhalten, der noch genau jene Zweige beinhaltet, die die geringsten Gesamtkosten verursachen. Der entsprechende, für den Baum optimale Assemblercode kann durch Traversieren dieses reduzierten Baumes von den Blättern zur Wurzel hin ebenfalls mit linearem Zeitaufwand gewonnen werden.

Wesentlich ist, daß für *jeden* auf der angegebenen Architektur berechenbaren arithmetischen Ausdruck auch ein reduzierter Trellisbaum existiert. Wenn der Viterbialgorithmus ohne Ergebnis terminiert, dann ist der gesuchte Ausdruck auf der Zielhardware nicht berechenbar.

Im Fall heterogener Prozessorstrukturen wird das Konzept der Trellisdiagramme gegenüber der hier dargestellten Einführung erweitert. Es wird nicht mehr unmittelbar die Anzahl der zur Verfügung stehenden Register aufgetragen, sondern zunächst aus der Kombination aller Belegungsmöglichkeiten aller Registerbänke die Summer der Zustände ermittelt, in denen sich der Prozessor zu

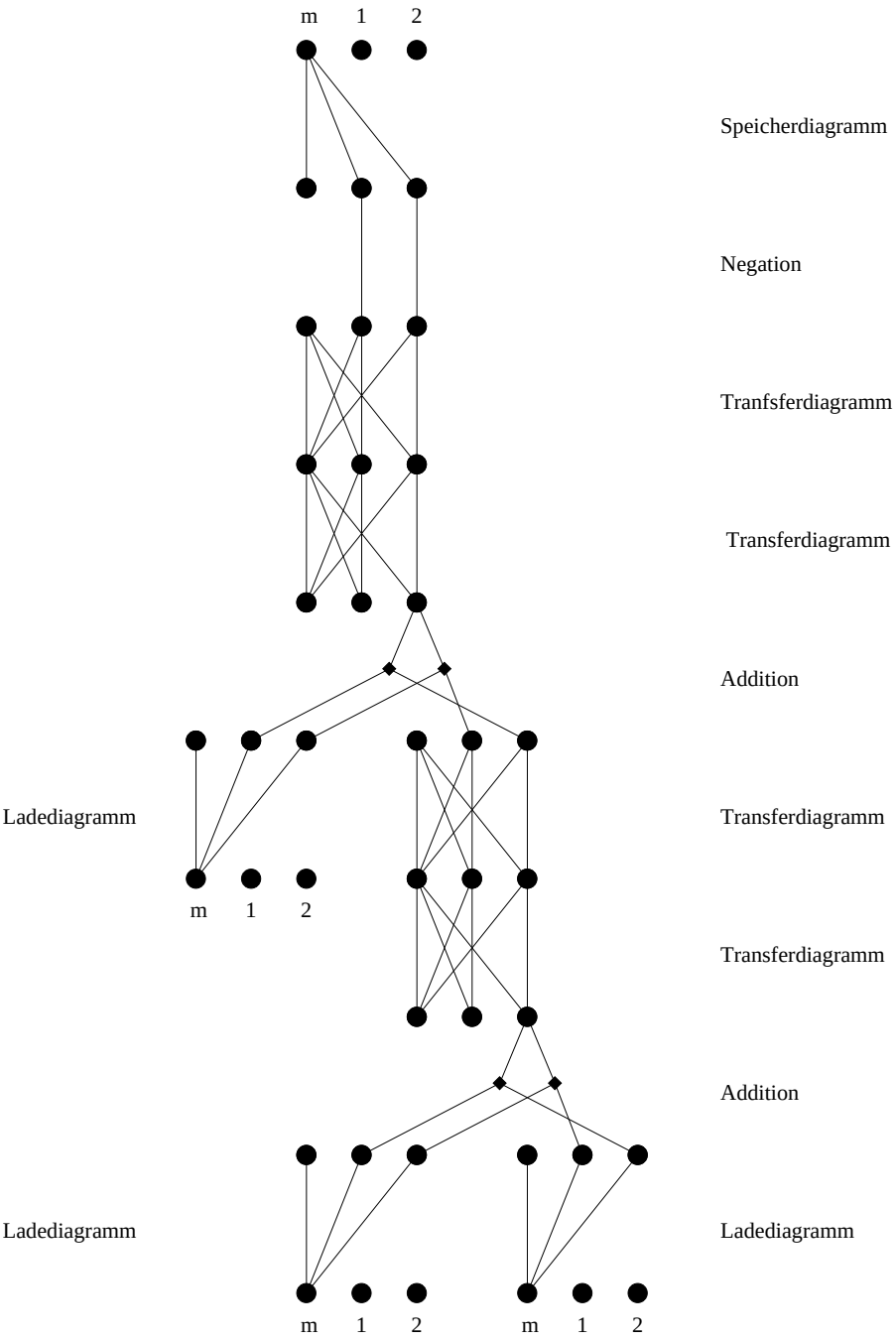

Abbildung 2.9: Trellisbaum

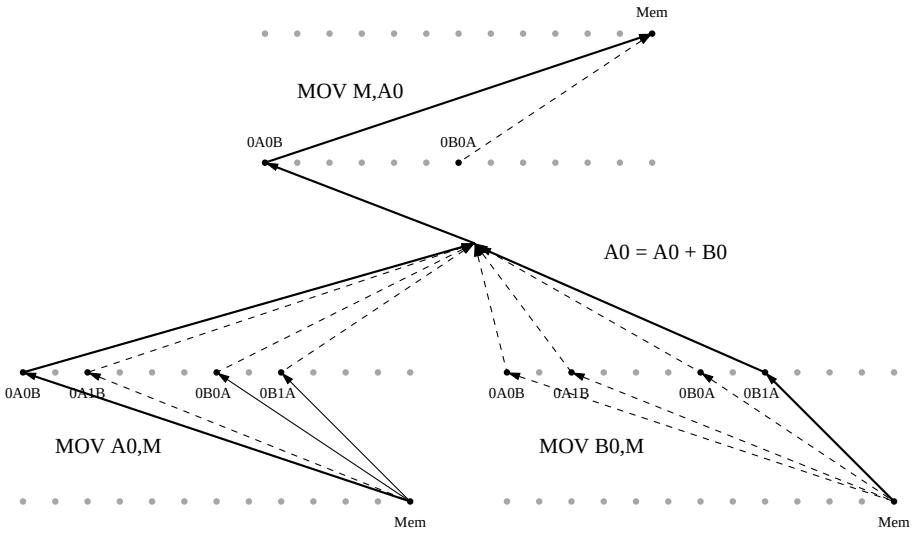

Abbildung 2.10: Dynamische Trellisdiagramme

einem beliebigen Zeitpunkt befinden kann. Diese Zustände werden dann analog zur Zahl der freien Register im homogenen Fall im Trellisdiagramm aufgetragen und auch ebenso verwendet ([15], [20]).

2.3.2 Dynamische Trellisdiagramme

Im Gegensatz zu den statischen Diagrammen werden dynamische Trellisdiagramme erst unmittelbar dann erzeugt, wenn sie auch tatsächlich im Baum benötigt werden. Weiters enthalten sie nicht mehr alle denkbaren Zustände, sondern nur noch diejenigen, von denen in der konkreten Realisierung auch Kanten wegführen. Dadurch kann das Trellisdiagramm für ein und denselben Befehl an zwei verschiedenen Stellen unterschiedlich aussehen, es ist aber in jedem Fall eine deutliche Reduktion der Zustandszahl möglich (das genaue Ausmaß hängt sehr stark von der Heterogenität der Zielarchitektur ab, siehe [8]. Weiters gewinnt man durch die dynamische Erzeugung an Flexibilität, im Gegensatz zu statischen Trellisdiagrammen können auch Mehrfachbefehle nachgebildet werden, so zum Beispiel die MAC-Operation (siehe [9]) oder Rundungs- bzw. Sättigungsoperationen. Es ist aber auch das Modellieren atypischer Befehle möglich, so kann beispielsweise ein Additionsbefehl verwendet werden, um eine Multiplikation mit 2 zu erreichen. Der Hauptnachteil dynamischer Trellisdiagramme ist die höhere benötigte Rechenzeit, die für den Gewinn an Flexibilität benötigt wird.

In Abbildung 2.10 wird ein Trellisbaum aus dynamischen Diagrammen für den Ausdruck $y = a + b$ auf einem fiktiven Prozessor mit 2 Registerbänken zu je 2 Registern gezeigt. Man erkennt deutlich, daß die Move-Diagramme in Abhän-

gigkeit vom Auftrittsort völlig unterschiedliche Charakteristik haben, auf alle Fälle nur sehr wenige aller möglichen Zustände tatsächlich verwendet werden.

2.4 Verbinden der Codefragmente

Nachdem für die Bäume mittels Trellis Diagrammen jeweils Assemblercode erzeugt worden ist, werden diese Codefragmente durch den Linker in der richtigen Reihenfolge zusammengefügt. Dabei wird versucht, Interfacevariablen in Registern zu behalten, wenn immer dies möglich ist. Nachteilig wirkt sich aus, daß der Trellisalgorithmus für diesen Optimierungsschritt nicht ausgelegt ist und daher bei der Vergabe von Registern auf mögliche Interfacevariablen keine Rücksicht nimmt. Die Optimierungsmöglichkeiten des Linkers sind aus diesem Grund beschränkt, der resultierende Code ist im allgemeinen suboptimal. Ein Feedback vom Linker zum Trellisalgorithmus, das für eine Verbesserung notwendig wäre, ist in den üblichen Compilern nicht vorgesehen.

2.5 Kompaktierung

Fast alle Signalprozessorarchitekturen unterstützen die parallele Ausführung mehrerer Befehle. Neben einer arithmetischen Operation können üblicherweise noch ein oder in bestimmten Fällen auch zwei Transferbefehle gleichzeitig durchgeführt werden. So ist es möglich, bereits die Quellregister für die nächste arithmetische Operation mit den entsprechenden Werten zu laden.

Bei der Kompaktierung des bis dahin rein sequentiellen Codes werden die Einzelbefehle auf ihre Parallelisierbarkeit und ihre gegenseitigen Datenabhängigkeiten hin untersucht und, wenn das möglich ist, neu angeordnet, so daß ein Maximum an Gleichzeitigkeit erzielt werden kann.

Für die Parallelisierung von Assemblerprogrammen existieren verschiedene Verfahren, wie der Critical Path Algorithmus, Branch and Bound Algorithmen oder List Scheduling [16]. Für die Kompaktierung im Bereich der digitalen Signalprozessoren ist *List Scheduling* allgemein anerkannt, da es bei linearer Ausführungszeit gute Ergebnisse liefert. Es handelt sich dabei um ein heuristisches Verfahren, das durch Weglassen von Zweigen mit geringer Erfolgswahrscheinlichkeit aus einem Branch-And-Bound Verfahren abgeleitet ist. Die für den in dieser Arbeit entwickelten Optimierungsalgorithmus wichtigen Prozessorarchitekturen weisen alle nur einen geringen Grad an Parallelität auf, für den List Scheduling hinreichend gute Ergebnisse liefert.

2.6 Phasenproblematik

Die erwähnten Optimierungsschritte (Befehlsauswahl und -anordnung, Linken und Kompaktieren) sind nicht voneinander unabhängig, sondern beeinflussen sich gegenseitig. Durch die sequentielle Abarbeitung findet ein Informationsfluß aber nur in eine Richtung statt, Fehlentscheidungen in den ersten Schritten

können anschließend nicht mehr korrigiert werden. Man spricht von einer Phasenabhängigkeit der verschiedenen Optimierungsstufen. Der erzeugte Code ist daher im allgemeinen suboptimal.

Kapitel 3

Genetische Algorithmen

3.1 Prinzip und Begriffsbestimmung

Genetische Algorithmen sind Optimierungsverfahren, die unabhängig von einer konkreten Problemstellung arbeiten. Sie benötigen, um die Optimierung durchführen zu können lediglich eine Information über die Parameter des Problems, welche zu optimieren sind und eine Auskunft über die Qualität der erzielten Lösung in Abhängigkeit der Eingangsparameter.

Es handelt sich bei genetischen Algorithmen um ein propabilistisches Verfahren. Das bedeutet, es gibt keine Garantie, jemals die optimale Lösung zu erreichen, dafür ist die Konvergenz des Verfahrens auch unter schlechten Randbedingungen gegeben. Das hat zur Folge, daß genetische Algorithmen besonders häufig dann zur Anwendung kommen, wenn Probleme mit extrem großen Lösungsräumen untersucht werden sollen, auf die bekannte Optimierungsverfahren bedingt durch den notwendigen Zeitaufwand nicht mehr angewendet werden können.

Ein weiterer Vorteil genetischer Algorithmen ist, daß sie nicht von einer mathematisch formulierbaren Zielfunktion abhängen und daher auch keine Eigenschaften bezüglich Stetigkeit oder Differenzierbarkeit verlangt werden.

Bei der Beschreibung genetischer Algorithmen werden unter anderem die folgenden Begriffe verwendet:

- Individuum
  Ein Individuum ist eine konkrete Lösung der gegebenen Aufgabenstellung, die sich durch einen wohldefinierten Satz an Parametern beschreiben läßt.
- Gen
  Das Gen enthält die charakteristische Information eines Individuums, es handelt also sich um die kodierte Darstellung der Parameter eines Individuums.
- Allel

Ein Allel ist ein konkreter Wert eines Gens, steht also für die charakteristische Information eines bestimmten Individuums.

- Fitness
  Die Fitness definiert die Qualität eines Individuums an Hand einer vorher festgelegten Skala. Je besser die Lösung, desto höher ist die Fitness. Üblicherweise definiert man die Fitness als nicht negativ und nach oben hin offen.

- Population
  Die Population ist die Summe aller Individuen. In jedem Zyklus des genetischen Algorithmus wird aus einer Population durch Selektion, Kombination und Mutation eine neue Population berechnet.

- Populationsgröße
  Die Populationsgröße gibt an, wieviele Individuen in der Population enthalten sind. Sie ist einer der entscheidenden Parameter für die Leistungsfähigkeit des Algorithmus.

- Hybrider genetischer Algorithmus
  Ein hybrider genetischer Algorithmus arbeitet nicht ausschließlich blind, also ohne Kenntnis des zu optimierenden Problems, sondern greift zusätzlich noch auf spezielles Fachwissen zurück. Ein hybrider genetischer Algorithmus kann beispielsweise einen Mutationsoperator verwenden, der gezielt Mutationen einer bestimmten, als günstig bekannten Richtung bevorzugt. Im Rahmen dieser Arbeit wird ein genetischer Algorithmus verwendet, der nicht direkt die Fitness der Individuen bewertet. Statt dessen werden noch bekannte, heuristische Optimierungsverfahren zwischengeschaltet und erst deren Ergebnis bestimmt die Fitnessfunktion. Die Optimierung der einzelnen Parameter erfolgt also indirekt über den Zwischenschritt des heuristischen Verfahrens.

3.2 Selektion

Unter Selektion versteht man die meist zufallsgesteuerte Auswahl von Individuen aus der Gesamtpopulation, um sie anschließend der Reproduktion zuzuführen. Die Selektion erfolgt typisch in Abhängigkeit der Fitness, bessere Individuen werden bevorzugt ausgewählt.

Eine Standardmethode zur Auswahl ist die Roulettemethode ([13], siehe Abbildung 3.1). Dabei wird jedem Element i der Gesamtpopulation P ein Gewicht w_i zugeordnet. Die Summe aller Gewichte $\sum_{I=1}^{n} w_i$ wird zu 1 normiert, das einzelne Gewicht w_i entspricht daher der Wahrscheinlichkeit, daß ein bestimmtes Individuum innerhalb eines Auswahlvorgangs selektiert wird.

Zur Festlegung der Gewichte existieren ebenfalls etablierte Verfahren. Die einfachste Möglichkeit ist, direkt die Fitnesswerte zu verwenden und durch die aufsummierte Fitness $f_{\text{ges}} = \sum_{i=1}^{n} f_i$ der Population zu dividieren. Die damit erzielten Ergebnisse sind jedoch nicht überzeugend, weil in Abhängigkeit von der

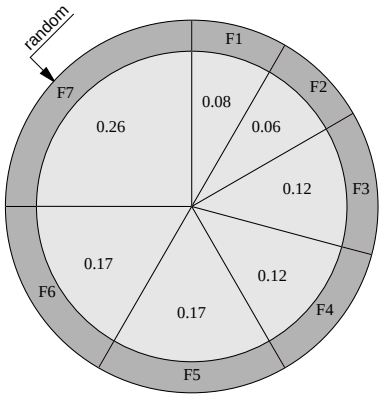

Abbildung 3.1: Selektion nach der Roulettemethode

Verteilung der Fitnesswerte eine stärkere oder eine schwächere Selektion durchgeführt wird. Beispielsweise kommt es zu Beginn der Optimierung, wenn die Individuen noch breit im Lösungsraum verteilt sind, zu einer sehr starken Unterdrückung der schlechteren Individuen, während gegen Ende des Vorganges, wenn die meisten Individuen in der Nähe des Optimums zu finden sind, kaum mehr eine Selektion stattfindet und daher die Optimierung nicht mehr zielgerichtet verläuft. Aus diesem Grund kann auf eine Skalierung der Population vor dem Selektionsprozeß nicht verzichtet werden.

Allgemein üblich ist eine automatische Skalierung, die jede Population auf einen konstanten, vordefinierten Wertebereich abbildet. Dabei wird oft ein festes Verhältnis zwischen der Fitness eines durchschnittlichen Individuums und der Fitness des besten Individuums als Kriterium verwendet. Dieser Faktor wird als Selektionsdruck bezeichnet und gibt an, wieviel Mal öfter das beste Individuum im Durchschnitt für die Nachfolgepopulation ausgewählt werden wird.

In Abhängigkeit von der konkreten Aufgabenstellung kommen auch komplexere Skalierungsverfahren zur Anwendung, die beispielsweise die Ergebnisse der letzten n berechneten Generationen berücksichtigen und damit die Trägheit des Algorithmus beeinflussen. Auch bei den Selektionsmethoden gibt es neben der Roulettemethode weitere Verfahren. Dazu zählen die *Rank Selection*, bei der anstatt des Fitnesswertes nur die Position innerhalb der Gesamtpopulation bewertet wird und die *Tournament Selection*, bei der gar keine Fitnesswerte verwendet werden, sondern nur ein Vergleichsoperator zwischen jeweils zwei oder mehreren Individuen benötigt wird. Die letzte Methode wird vor allem dann bevorzugt, wenn sich die Qualität einer Lösung nicht in einer einzigen Zahl ausdrücken läßt.

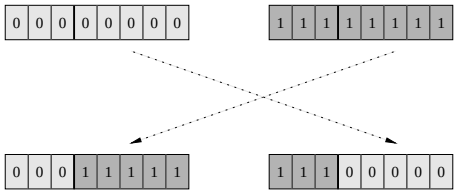

Abbildung 3.2: 1-point Crossover

3.3 Crossover, Rekombination

Mittels der Rekombination werden aus zwei im allgemeinen zufällig ausgewählten Individuen neue Individuen abgeleitet. Zu diesem Zweck wird ein eigener Crossover Operator benötigt. Eines der am weitesten verbreiteten Verfahren ist das sogenannte *1-point crossover*. Dabei wird die als Bitstring codierte Information zweier Individuen jeweils an der gleichen Stelle (dem *crossover point*) geteilt, die Teilstücke werden dann gegenseitig ausgetauscht und zu zwei neuen Chromosomen zusammengesetzt. Die Wahl des crossover points erfolgt durch einen Zufallsprozeß. Eine Veranschaulichung dieses Operators wird in Abbildung 3.2 gezeigt.

Eine Erweiterung des 1-point Crossover Verfahrens sind das 2-point bzw. das Multipoint-Crossover Verfahren, bei denen ebenfalls das Chromosom in Teilstücke getrennt wird, allerdings erfolgt die Auftrennung nicht nur an einem, sondern an zwei bzw. n verschiedenen Stellen. Dadurch wird einerseits eine größere Mischung des vorhandenen Erbguts erreicht, auf der anderen Seite gehen allerdings vorhandene Informationen auch leichter wieder verloren, so daß man generell mit einer schwächeren Konvergenz rechnen muß.

Für kombinatorische Probleme ist die Auswahl eines Crossover Operators generell schwieriger, da nicht einfach Bitmuster ausgetauscht werden können, wie im vorigen Absatz beschrieben. Statt dessen enthalten die Chromosomen Angaben über die Position der einzelnen Elemente einer vorgegebenen Menge. Da diese nur jeweils ein Mal innerhalb des Chromosoms auftreten dürfen, müssen zusätzliche Anforderungen an den Crossover Operator gestellt werden. In der Literatur werden unter anderem order crossover ([4]), cycle crossover ([19]) und partially mapped crossover ([12]) erwähnt. In [7] finden sich noch weitere, deutlich komplexere Verfahren, mit denen sich in vielen Anwendungen raschere Konvergenz erzielen läßt. Beim vorliegenden Problem handelt es sich allerdings nicht nur um eine rein kombinatorische Aufgabe, sondern es treten zusätzlich noch Randbedingungen in Form von Abhängigkeiten zwischen den einzelnen Elementen. Die meisten der etablierten Verfahren können eine korrekte Reihenfolge der Elemente nicht sicherstellen, dies ist lediglich beim order crossover der Fall, das deswegen auch die Basis für den hier verwendeten Algorithmus bildet.

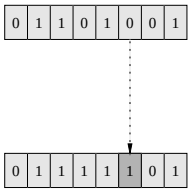

Abbildung 3.3: Mutationsoperator

3.4 Mutation

Im Rahmen der Mutation wird das Erbgut eines Individuums zufällig verändert. Durch dieses Verfahren wird sichergestellt, daß immer wieder neues Erbgut, bzw. Erbgut, daß im Lauf der Generationen verloren gegangen ist, in die Population eingebracht wird. Die Mutation kann beispielsweise durch Ändern eines beliebigen Bits innerhalb des Chromosoms erfolgen (siehe Abbildung 3.3). Grundsätzlich muß bei der Konstruktion von Mutationsoperatoren beachtet werden, daß am Individuum nur kleine Änderungen in Bezug auf dessen Fitnessfunktion vorgenommen werden, um die Konvergenz des Prozesses nicht zu gefährden.

3.5 Parameter

Die in den vorigen Abschnitten beschriebenen Verfahren lassen sich jeweils durch bestimmte Parameter steuern, durch die man Einfluß auf die Effizienz und die Qualität der Optimierung des genetischen Algorithmus gewinnen kann. Dabei handelt es sich um

- den *Selektionsdruck*
  Der Selektionsdruck bestimmt, um wieviel höher die Überlebenschancen für das beste Individuum einer Population sind, als für ein durchschnittliches Individuum. Die Wahl des richtigen Selektionsdrucks ist für eine gute Gesamtperformance außerordentlich wichtig. Bei zu hohem Druck werden gute Individuen überproportional bevorzugt, alternative Lösungen haben daher kaum Chancen auf Verbreitung, der Algorithmus tendiert zum Konvergieren in lokalen Maxima. Zu niedriger Selektionsdruck verhindert, daß sich gute Lösungen gegenüber schlechteren durchsetzen. Der genetische Algorithmus degeneriert auf diese Weise zu einer Zufallssuche und konvergiert gar nicht mehr, oder nur sehr langsam.

- die *Crossoverwahrscheinlichkeit*
  Die Crossoverwahrscheinlichkeit beschreibt, wie häufig nach der Selektion eine Rekombination durchgeführt wird. In allen anderen Fällen werden die selektierten Individuen unverändert in die nachfolgende Population übernommen.

- die *Mutationswahrscheinlichkeit*
  Mit der Mutationswahrscheinlichkeit wird festgelegt, wie oft beliebige Bits innerhalb des Chromosoms verändert werden. Bei einer zu niedrigen Mutationsrate besteht die Gefahr der Verarmung des Erbmaterials, bei hohen Mutationsraten entartet der genetische Algorithmus zu einer gerichteten Zufallssuche.

- den *Elitismus*
  Von Elitismus spricht man, wenn das beste, bzw. die n besten Individuen einer Generation unabhängig vom Selektions- und vom Rekombinationsprozeß in die nächste Generation übernommen werden. Damit wird sichergestellt, daß einmal erzielte gute Lösungen nicht wieder zufällig verloren gehen können. Nachteilig wirkt sich hingegen aus, daß dadurch eine höhere Konvergenz zu lokalen Maxima besteht, der beispielsweise durch eine größere Mutationswahrscheinlichkeit begegnet werden kann.

In theoretischen Untersuchungen von genetischen Algorithmen wird an verschiedenen Stellen die Frage nach der Notwendigkeit von Crossover Operatoren (bzw. alternativ dazu nach der Notwendigkeit von Mutationsoperatoren) aufgeworfen. Während bei einfachen Aufgaben (die sich vollständig durch Bitstrings beschreiben lassen und deren Lösungsraum keine ungültigen Bereiche enthält) eine Optimierung alleine durch Mutation unter Umständen zielführend ist, genügt das bei so komplexen Chromosomen wie in der gegebenen Aufgabenstellung nicht mehr. In [22] und [23] wird gezeigt, daß Crossover Operatoren besser geeignet sind, building blocks höherer Ordnung zu erzeugen, als das mit Mutationsoperatoren der Fall ist. Das wird auch durch die im Kapitel 5 dokumentierten Testläufe bestätigt, die mit Crossover wesentlich schneller und auch tiefere Sättigungsniveaus erreichen, als ohne.

Kapitel 4

Hybrider Optimierer

4.1 Ansatz und Motivation

Die vorliegende Arbeit stellt eine Lösung vor, mit der das bisher unvermeidbare Phasenkopplungsproblem bei der automatischen Codegenerierung weitestgehend ausgeschaltet werden kann. Zu diesem Zweck werden die verschiedenen Phasen in einem gemeinsamen, übergreifenden Optimierungsschritt zusammengefaßt. Man erhält dadurch einen extrem großen Suchraum (im Fall eines Lattice-Filters zweiter Ordnung beispielsweise etwa 10^{22}, siehe Kapitel 6.5), der mit konventionellen Optimierungsstrategien nicht mehr zu bewältigen ist.

Auf Grund positiver Erfahrungen im Bereich des Memory Layout Assignment [29] wird in dieser Arbeit auf evolutionäre Suchstrategien zurückgegriffen. Konkret wird ein genetischer Algorithmus dazu verwendet, im Zusammenspiel mit konventionellen Strategien den erzeugten Assemblercode global zu optimieren. Dies geschieht durch Festlegen der Randbedingungen, unter denen dann die einzelnen Phasen lokal berechnet werden. Die Teilalgorithmen sind ihrerseits sehr schnell (sie arbeiten jeweils mit linearer Komplexität bezüglich der Größe des Datenflußgraphen), dadurch ist eine Verwendung innerhalb des genetischen Algorithmus sinnvoll möglich.

Da die Einzelalgorithmen die evolutionäre Optimierung durch Expertenwissen unterstützen, handelt es sich insgesamt um einen hybriden genetischen Optimierer.

Es ist grundsätzlich auch möglich, auf die konventionellen Suchstrategien zu verzichten, indem man den Datenflußgraphen in seine elementaren Bestandteile zerlegt. Die einzelnen Bäume müssen dann nicht mehr mit dem Trellisalgorithmus bearbeitet werden, weil die Lösung unmittelbar aus den Randbedingungen folgt. Man erhält auf diese weise einen rein genetischen Optimierer. In der Praxis zeigt sich jedoch, daß ein solches Vorgehen den Suchraum massiv vergrößert, so daß der genetische Algorithmus nicht mehr in der Lage ist, zuverlässig hochoptimierte Lösungen zu finden (siehe beispielsweise die Ergebnisse in Kapitel 6.2.2).

$$
\begin{aligned}
z_2 &= c_1 * (x + x_0) \\
z_3 &= x + z_2 \\
z_7 &= c_2 * (z_3 + x_1) \\
z_6 &= z_2 + x_0 \\
x_0 &= z_7 + x_1 \\
x_1 &= z_3 + z_7 \\
y &= c_3 * z_6 + c_4 * x_0 + c_5 * x_1
\end{aligned}
$$

Abbildung 4.1: Systemgleichungen entsprechend der Baumzerlegung eines Lattice-Filters zweiter Ordnung

4.2 Baumzerlegung

Da der Trellisalgorithmus in der Lage ist, für jeweils einen Baum unter den vorgeschriebenen Randbedingungen den optimalen Assemblercode mit linearem Aufwand bezogen auf die Anzahl der Knoten innerhalb eines Baumes zu erzeugen, ist es sinnvoll, größtmögliche Bäume zu verwenden. Dabei darf jedoch keine Einschränkung an den darüber liegenden genetischen Algorithmus gestellt werden, da jede Reduktion der Freiheitsgrade die Qualität der Optimierung mindert. Aus diesem Grund muß beim Erstellen der Bäume immer dann eine Auftrennung erfolgen, wenn ein Ergebnis an mehreren Stellen weiter verwendet wird. Nur dadurch ist es dem genetischen Algorithmus möglich, die Optimierung der Ein- und Auslagerungsvorgänge zu steuern. Aus der selben Überlegung heraus wird auch vorausgesetzt, daß innerhalb eines Baumes kein Auslagerungsvorgang notwendig ist. Übergroße Bäume werden daher geteilt, die Einzelteile mit dem Trellisalgorithmus optimiert und durch den genetischen Algorithmus in der geeignetesten Reihenfolge wieder zusammengefügt. Ein Beispiel für eine Baumzerlegung wird in Abbildung 4.1 gegeben. Es handelt sich dabei um die Systemgleichungen für das Lattice-Filter zweiter Ordnung, dessen Datenflußgraph bereits in Abbildung 1.1 skizziert worden ist. Jede der angegebenen Gleichungen entspricht genau einem Baum.

4.3 Chromosom

Um ein Individuum eindeutig bestimmen zu können, wird sowohl Information über die Abarbeitungsreihenfolge der Bäume benötigt, als auch über die Art und den Ort des Zugriffs auf die Bauminterfacevariablen. Aus der Summe dieser Informationen wird das Chromosom gebildet. Es handelt sich hier um eine für genetische Algorithmen vergleichsweise komplexe Codierung, da sehr unterschiedliche Daten gemeinsam erfaßt werden müssen.

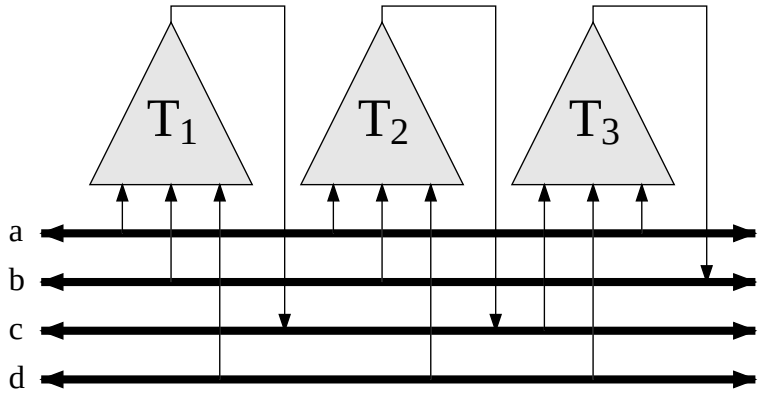

Abbildung 4.2: Interface zwischen Bäumen und dem Gleichungssystem

4.3.1 Abarbeitungsreihenfolge

Durch die Angabe eines Gleichungssystems werden dem Algorithmus die Anzahl und der Aufbau der einzelnen Bäume vorgeschrieben, es bleiben jedoch Freiheitsgrade in der Reihenfolge der Abarbeitung vorhanden. Grundsätzlich muß jedoch darauf geachtet werden, daß durch eine neue Reihenfolge keine baumübergreifenden Abhängigkeiten verletzt werden. Die Ordnung, die durch die Reihenfolge der Gleichungen vorgegeben wird, ist auf alle Fälle gültig. Davon ausgehend kann man durch fortgesetztes Vertauschen zweier benachbarter Bäume alle weiteren gültigen Schedules erreichen, wenn man dabei die folgende Bedingung erfüllt: zwei benachbarte Bäume sind exakt dann vertauschbar, wenn die Zielvariable des einen Baumes nicht im jeweils anderen Baum als Quellvariable verwendet wird.

Das Chromosom (bzw. der Teil davon, der für die Abarbeitungsreihenfolge zuständig ist) wird durch die Liste der Bäume in der Reihenfolge ihrer Ausführung definiert. Das ist eine für solche Probleme übliche und zweckmäßige Codierung (siehe z.B. [30]).

4.3.2 Interfacevariablen

Wie in Abbildung 4.2 gezeigt wird, ist jeder Baum mit dem gesamten Gleichungssystem über 1 Ausgangsvariable und n Eingangsvariable verbunden. Abbildung 4.3 greift einen beliebigen Baum in vergrößerter Darstellung heraus. Man sieht, daß für jede Interfacevariable die Zugriffsart vorgegeben wird. Dabei sind die folgenden drei Begriffe zu unterscheiden:

- Die Quelle bzw. das Ziel der Interfacevariablen. Jede Variable kann sich entweder im Speicher oder in einem Register gehalten werden. Handelt es sich um eine Eingangsvariable, entspricht dies den Zuständen *load* (für Speicherzugriffe) und *use* (für Registerzugriffe), handelt es sich um die

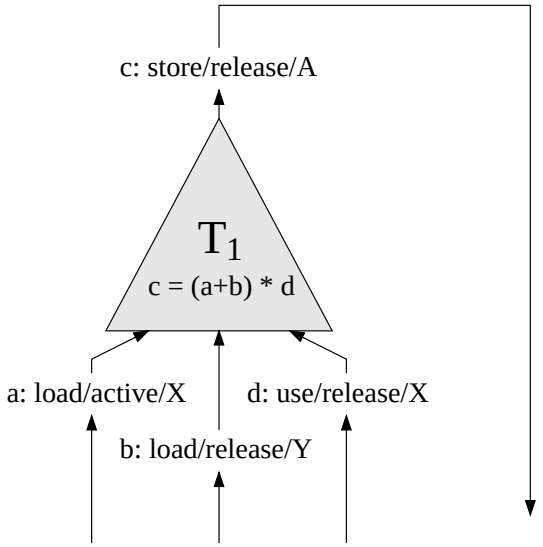

Abbildung 4.3: Beschreibung der Interfacevariablen

Zielvariable, so ist zwischen *store* (für Speicherzugriffe) und *keep* (für Registerzugriffe) zu unterscheiden.

- Der Angabe, ob der Wert auch nach Verwendung noch im Register gehalten werden soll, oder ob das Register anschließend zur freien Verfügung steht. Der erste Fall wird mit dem Schlüsselwort *active* gekennzeichnet, der zweite Fall mit *release*.

- Der Angabe, welche Registerbank verwendet werden soll. Beispielsweise werden für die Prozessorfamilie 56k von Motorola die Bezeichnungen A, X und Y verwendet.

Welches Register innerhalb der gewählten Registerbank verwendet wird, ist nicht wesentlich, da die Register alle gleichwertig sind. Die genaue Festlegung des verwendeten Registers wird erst zu einem späteren Zeitpunkt getroffen.

Durch Kombination der drei relevanten Attribute können alle Interfacevariablen eindeutig festgelegt werden. In Abbildung 4.4 sind die sinnvollen Kombinationen zusammen mit einer kurzen Beschreibung angeführt (die Angabe der Registerbank entfällt in der Tabelle, weil aus der Sicht des Algorithmus alle Registerbänke gleichwertig sind).

Abgesehen von den Zugriffsarten auf die Interfacevariablen, werden noch die folgenden Informationen zur Codegenerierung innerhalb eines Baumes benötigt:

- Speicherbänke: der genetische Algorithmus optimiert die Belegung der verfügbaren Speicherbänke, insbesondere unter dem Aspekt der besseren Par-

Name		Beschreibung
load	active	laden aus dem Hauptspeicher und behalten im Register
load	release	laden aus dem Hauptspeicher und anschließend freigeben
use	active	Registerinhalt verwenden und weiterhin reserviert lassen
use	release	Registerinhalt verwenden und anschließend freigeben
store	active	Variable speichern und im Register aktiv lassen
store	release	Variable speichern und das Register freigeben
keep	active	Ergebnis nicht speichern, sondern im Register halten

Abbildung 4.4: Zugriffsarten auf Interfacevariable

allelisierbarkeit des erzeugten Zwischencodes. Da der Zugriff auf den Speicher von der verwendeten Bank abhängig sein kann, ist diese Angabe auch innerhalb des Trellisalgorithmus notwendig.

- gesperrte Register: da Teilergebnisse auch baumübergreifend in Registern gehalten werden können, muß der Trellisalgorithmus die Information erhalten, welche Registerbänke für die Berechnung zur Verfügung stehen. In der Praxis bestehen Registerbänke meist aus mehr als nur einem einzelnen Register, es wird daher nicht die Belegung, sondern die Anzahl der jeweils in der Bank noch verfügbaren Register ausgewertet. Es ist allerdings nicht notwendig. Diese Information im Chromosom zu kodieren, da sie sich aus den Interfacevariablen der anderen Bäume eindeutig berechnen läßt. Die Registerbelegung für den ersten Baum kann vom Anwender vorgegeben werden, um bestimmte Register für das Programm zu sperren. Diese Information ist jedoch global für alle Individuen gleich und wird daher ebenfalls nicht im Chromosom kodiert.

4.3.3 Matrixdarstellung

Um eine kompaktere Schreibweise der Randbedingungen zu ermöglichen, kann man die Zugriffsarten auch in Matrixform darstellen. Dabei sind die Zugriffsmatrix und der Speichervektor zu unterscheiden. Die Zugriffsmatrix gibt an, mit welcher Zugriffsart eine Variable zu einem bestimmten Zeitpunkt angesprochen und welche Registerbank dafür verwendet wird, während der Speichervektor bei einer allfälligen Auslagerung in den Hauptspeicher festlegt, welche Speicherbank verwendet wird. In den Zeilen dieser beiden Konstrukte werden jeweils die vorhandenen Interfacevariablen aufgetragen. Die Spalten der Zugriffsmatrix werden durch die Teilbäume des Gleichungssystems in der Reihenfolge ihrer Berechnung gebildet. In der Matrix sind die Elemente nur dann besetzt, wenn innerhalb des betreffenden Teilbaumes auf die entsprechende Variable zugegriffen wird. Kommt eine Variable innerhalb eines Baumes nicht vor, so bleiben die entsprechenden Elemente der Matrix leer.

Abbildung 4.5 zeigt ein Beispiel für die Zustandsmatrix und den Speichervektor für das Lattice-Filter zweiter Ordnung aus Abbildung 1.1. Die Konstanten c_1

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	Bank
z_2	sa	ua		ur				X
z_3		sa	ur			lr		X
z_7			sr		la	ur		Y
z_6				sr			lr	Y
x_0	lr			lr	sa		ur	X
x_1			la		ur	sa	ur	Y
y							sr	Y

Abbildung 4.5: Zustandsmatrix und Speichervektor eines Lattice-Filters

bis c_5 sind dabei zur Erhöhung der Übersichtlichkeit nicht mit angeführt. In der Zustandsmatrix kann man erkennen, daß alle Schreibzugriffe auf der Hauptdiagonale liegen, was durch entsprechende Anordnung der Variablen erreicht wird. Diese Anordnung hat den Vorteil, daß sich eine klare Trennung zwischen den Ladeoperationen oberhalb und unterhalb der Hauptdiagonale ergibt. Die Zugriffe oberhalb der Hauptdiagonale erfolgen nach der Berechnung des entsprechenden Wertes, es handelt sich also um Zwischenergebnisse, die an einer oder mehreren Stellen verwendet werden. Im Gegensatz dazu werden die Zugriffe unterhalb der Hauptdiagonale ausgeführt, *bevor* das entsprechende Ergebnis berechnet worden ist. In diesem Fall wird der Wert der letzten Berechnung verwendet (die Programme werden ja grundsätzlich zyklisch ausgeführt), was im Datenflußdiagramm dem Einsatz eines Verzögerungselementes entspricht. Im konkreten Beispiel treten Zugriffe unterhalb der Hauptdiagonale nur für die Variablen x_0 und x_1 auf (mit jeweils zwei Zugriffen), was auch mit den zwei Verzögerungselementen und deren Beschaltung in Abbildung 1.1 korrespondiert.

4.4 Genetische Operatoren

4.4.1 Selektion

Als Ausgangsbasis für den Fitnesswert wird die Ausführungszeit t des erzeugten Codes herangezogen (ebenso wäre auch die Codegröße oder eine Kombination beider Kriterien möglich; zur Zeit sind diese Kriterien aber gleichwertig, da alle Befehle mit gleicher Ausführungszeit bewertet werden). Ein niedrigerer Zahlenwert für t bedeutet besseren Code. Da die üblichen Selektionsmechanismen allerdings größere Fitnesswerte bevorzugen, müssen die Ergebnisse vor der Selektion entsprechend umskaliert werden. Als Basiswert für die Fitness wird der Wert $f = t_{max} + t_{min} - t$ verwendet, der im gleichen Wertebereich zu liegen kommt (t_{max} entspricht dabei f_{min} und umgekehrt).

Zusätzlich zu diesem unskalierten Fitnesswert werden im Rahmen der vorliegenden Arbeit noch zwei verschiedene Methoden zur Skalierung vorgestellt, nämlich die lineare und die exponentielle Skalierung, die eine raschere und bessere Konvergenz des Verfahrens bewirken sollen.

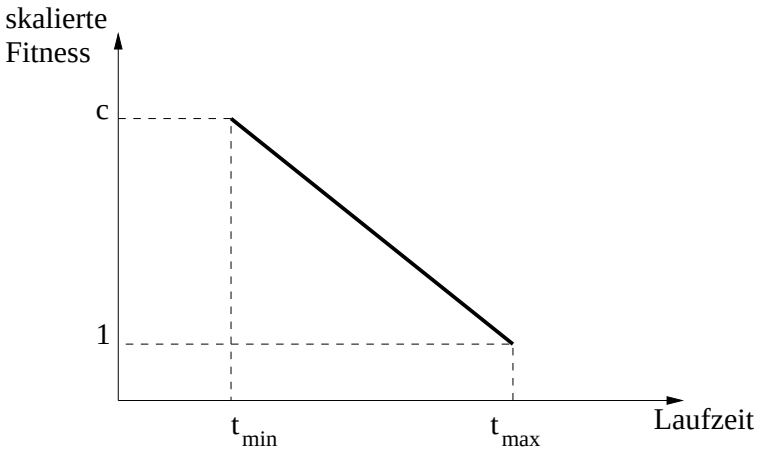

Abbildung 4.6: Lineare Skalierung

Lineare Skalierung

Für die gegebene Aufgabenstellung treten typischerweise ähnliche Fitnesswerte ohne große Streuung auf, die relativ weit von Null entfernt sind. Eine Skalierung wie beispielsweise in [13] vorgeschlagen ist daher nicht zielführend, da in der Mehrzahl der Fälle die reguläre Skalierung nicht durchführbar wäre und auf das Ersatzverfahren zurückgegriffen werden müßte, das keine Wahl des Selektionsdrucks mehr ermöglicht. Gerade der Einfluß des Selektionsdrucks stellt aber einen wichtigen Punkt in der Untersuchung der verwendeten Algorithmen dar.

Aus diesem Grund wird im Rahmen dieser Arbeit ein anderes, einfacheres Bewertungsverfahren verwendet: der Fitnessbereich der jeweiligen Population wird durch eine lineare Umskalierung auf den Bereich von 1 bis c abgebildet. Der Parameter c wird in weiterer Folge als Selektionsdruck bezeichnet und gibt an, wieviel wahrscheinlicher das beste Individuum einer Population zur Reproduktion ausgewählt wird, als dies beim schlechtesten Individuum der Fall sein wird. In Abbildung 4.6 wird das Prinzip der linearen Skalierung graphisch veranschaulicht.

Der Nachteil dieser Methode gegenüber der ursprünglich in [13] vorgeschlagenen besteht darin, daß es im Fall eines überdurchschnittlich guten Individuums zu vorzeitiger Fokussierung auf ein lokales Optimum kommen kann, weil die anderen Individuen durch die starke Skalierung zurückgedrängt würden. Die Referenzmethode umgeht diese Schwierigkeit durch Verwendung der mittleren Fitness, die sich durch ein einzelnes Superindividuum nur geringfügig ändert. Bei der gegenständlichen Problematik sind aber die Fitnesswerte durch die minimale Programmgröße nach oben hin begrenzt, außerdem kommt es durch die Natur der zu optimierenden Funktion zu keinen großen Sprüngen in der Bewertung, Superindividuen sind daher nicht zu erwarten. Dagegen steht der Vorteil der verwendeten Methode, den Selektionsdruck im Bereich 1 bis ∞ einstellen

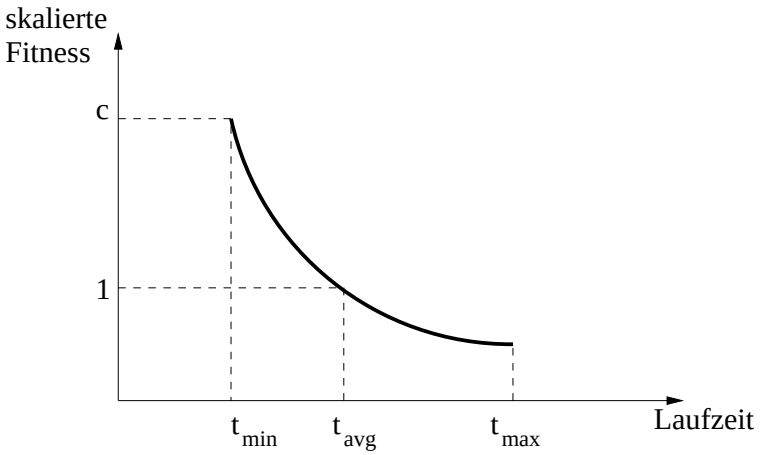

Abbildung 4.7: Exponentielle Skalierung

zu können.

Exponentielle Skalierung

Bei der exponentiellen Skalierung (auch als *Power Law Skalierung* bekannt) werden die natürlichen Fitnesswerte f_i mit einem Potenzfaktor k gewichtet, so daß man als skalierte Fitness die Werte f_i^k erhält. Ein Beispiel für exponentielle Skalierung wird in Abbildung 4.7 gezeigt.

Um einen gleichmäßigen Selektionsdruck zu erreichen, werden die Fitnesswerte zuvor mit der durchschnittlichen Fitness der Population normiert. Im Anschluß daran wird k so berechnet, daß das beste Individuum die skalierte Fitness c, also exakt den Selektionsdruck aufweist. k läßt sich durch die Beziehung

$$k = \frac{\ln c}{\ln \frac{f_{min}}{f_{avg}}}$$

ausdrücken, die skalierten Fitnesswerte g_i errechnen sich dann zu

$$g_i = (\frac{f_i}{f_{avg}})^k.$$

Im Gegensatz zur linearen Skalierung ist hier die Verwendung der durschnittlichen Fitness möglich, da unter keinen Umständen negative skalierte Fitnesswerte auftreten können.

4.4.2 Crossover

Die Aufgabe des Crossover-Operators ist es, zwei verschiedene Individuen zu einem neuen zu kombinieren. Da kein Bitstring als Chromosom vorhanden ist,

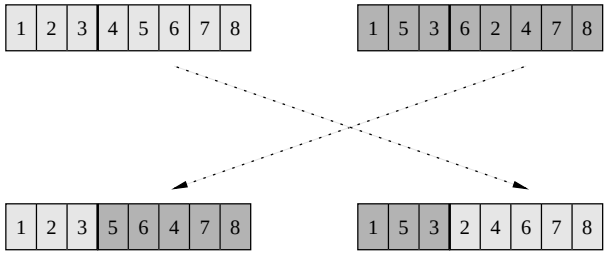

Abbildung 4.8: Crossover Operator für die Abarbeitungsreihenfolge der Teilbäume, detaillierte Erläuterung im Text

kann der klassische one-point Operator nicht angewendet werden, sondern es ist notwendig, eine eigene, an die Aufgabenstellung angepaßte Funktion zu entwickeln.

Die charakteristischen Informationen jedes Individuums lassen sich in drei voneinander unabhängige Bereiche aufteilen, nämlich die Allokation des Hauptspeichers, die Abarbeitungsreihenfolge der Bäume und die Zugriffsarten auf die Bauminterfacevariablen.

Beim Zugriff auf den Hauptspeicher können keine ungültigen Zustände auftreten, daher wird für diesen Teilbereich ein one-point Crossover-Operator implementiert.

Bei der Abarbeitungsreihenfolge wird ein ähnliches Verfahren verwendet, allerdings muß hier darauf geachtet werden, daß nur gültige Permutationen erzeugt werden dürfen. Dies wird durch das folgende Verfahren sichergestellt, das auch in Abbildung 4.8 dargestellt ist und auf order crossover ([4]) basiert:

- Zunächst wird der Aufteilungspunkt für den Operator zufällig gewählt (beispielsweise die Position n innerhalb des Schedules)
- Vom ersten Individuum werden die ersten n Bäume in unveränderter Abarbeitungsreihenfolge übernommen
- Die restlichen Bäume werden in der Reihenfolge angefügt, in der sie innerhalb des Schedules des zweiten Individuums auftreten

Im Beispiel von Abbildung 4.8 werden die ersten drei Positionen im Schedule der beiden Eltern beibehalten. Die restlichen fünf Plätze werden in der Reihenfolge des Auftretens im Schedule des jeweils anderen Elternteils vergeben. Für das links dargestellte Individuum sind die Bäume 1, 2 und 3 unverändert vom linken Elternteil übernommen, die restlichen Bäume (4, 5, 6, 7 und 8) werden in der Reihenfolge angeführt, in der sie im rechten Elternteil auftreten, im konkreten Beispiel ist das 5, 6, 4, 7 und 8. Analog wird das Schedule für das andere abgeleitete Individuum erstellt.

Die Zugriffsarten für die Bauminterfacevariablen werden zunächst unverändert von demjenigen Individuum übernommen, aus dem auch die Position des

entsprechenden Baumes innerhalb des Schedules abgeleitet worden ist. Dadurch wird sichergestellt, daß die Zugriffe für benachbarte Bäume auch aus dem gleichen Individuum stammen, grobe Unstetigkeiten können auf diese Weise vermieden werden. Dieser Operator ist stakr an die gegebene Aufgabenstellung angepaßt und gehört nicht zu den klassischen Crossover Operatoren. Die Zugriffsmatrix ist in der Regel allerdings *nicht* gültig, insbesondere an der durch den Crossover-Operator entstandenen Grenze treten Inkonsistenzen auf, aber auch durch die Änderung der Abarbeitungsreihenfolge. Daher muß das Individuum im Anschluß an den Crossover durch eine Reparaturfunktion wieder in einen gültigen Zustand übergeführt werden.

Reparaturfunktion

Die Reparaturfunktion stellt sicher, daß die Zugriffsmatrix eines Individuums in einen konsistenten Zustand gebracht wird. Zu diesem Zweck werden alle Teilbäume für jede Zugriffsvariable der Reihe nach abgearbeitet und die Zugriffsart sowie die ausgewählte Registerbank korrigiert, wenn das erforderlich ist. Dabei wird immer der spätere Zugriff an den früheren angepaßt. Eine Ausnahme dafür ist nur bei der Schleifenoptimierung (siehe Kapitel 4.6) notwendig, da sichergestellt werden muß, daß die verwendeten Register am Anfang und am Ende des Schedules identisch sind. Daher werden bei aktivierter Schleifenoptimierung zunächst die Registerbänke schleifenübergreifend angepaßt und erst danach die Zugriffsarten festgelegt.

4.4.3 Mutation

Für den Algorithmus kommen verschiedene Mutationsoperatoren zur Anwendung, die mit unterschiedlicher Gewichtung ausgewählt werden. Die Gewichtung nimmt Rücksicht darauf, daß die einzelnen Operatoren unterschiedlich starke Auswirkungen auf das Individuum haben.

Im Gegensatz zum Crossover ist es bei der Mutation möglich, durch gezielte Änderungen immer die Konsistenz des Individuums zu bewahren. Der Aufruf einer Reparaturfunktion kann daher entfallen.

Auslagern von Variablen

Dieser Operator verschiebt eine Variable aus einer Speicherbank in ein Register und umgekehrt. Die Änderung erfolgt jeweils in der kleinstmöglichen Einheit, also zwischen zwei aufeinanderfolgenden Zugriffen. Sowohl die Variable, die verändert wird, als auch der betroffene Zugriff werden gleichverteilt zufällig ausgewählt. In Abhängigkeit der Zugriffsart sind unterschiedliche Änderungen notwendig:

- *load/release*: Die Variable wurde bisher nach dem Laden wieder verworfen. Nach der Mutation bleibt sie im entsprechenden Register enthalten, der Zustand wird daher auf *load/active* geändert.

- *load/active*: Die Variable wurde bisher im Register behalten, nach der Mutation ist dies nicht mehr der Fall. Der Zustand wird daher auf *load/release* geändert.

- *use/active*: Die Variable wurde bisher im Register behalten. Nach der Mutation wird das Register unmittelbar nach dem Zugriff freigegeben, der Zustand wird daher auf *use/release* geändert.

- *use/release*: Das Register wurde bisher nach dem Zugriff auf die Variable freigegeben. Nach der Mutation bleibt die Variable erhalten, der Zustand muß deshalb auf *use/active* geändert werden.

- *store/release*: Bisher wurde die Variable in den Speicher geschrieben und das entsprechende Register freigegeben. Nach der Mutation wird der Inhalt des Registers für den nächsten Zugriff aufgehoben, der Zustand ändert sich daher auf *store/active*.

- *keep/active, store/active*: Die Variable wurde nach dem Abspeichern im Register behalten, nach der Mutation hingegen wird das Register unmittelbar nach dem Speicherzugriff freigegeben. Der neue Zustand muß daher *store/release* lauten.

Auch beim nachfolgenden Zugriff auf diese Variable sind Anpassungen notwendig, da sich der Zugriff von Speicher auf Register bzw. umgekehrt ändert. Im Detail ergeben sich die folgenden Vorschriften:

- *load/release*: Die Variable wurde bisher aus dem Speicher geholt, steht aber nach der Mutation in einem Register. Es ist daher die Änderung des Zustands auf *use/release* notwendig, sowie die Anpassung der verwendeten Registerbank auf diejenige, die bereits beim ersten Zugriff benutzt wurde.

- *load/active*: Hier wurde die Variable ebenfalls aus dem Speicher geholt, allerdings im Gegensatz zum letzten Punkt anschließend nicht unmittelbar freigegeben. Daher muß der neue Zustand auf *use/active* gesetzt werden und die Anpassung der verwendeten Registerbank ist bei allen folgenden Zuständen notwendig, bis durch das Auftreten von *use/release* das Register wieder freigegeben wird.

- *use/active, use/release*: Die Variable wurde bisher durch ein Register übergeben, nach der Mutation muß auf den Speicher zugegriffen werden. Der Zustand ist daher auf *load/active* bzw. *load/release* zu ändern. Anpassungen bei den verwendeten Registern sind hier nicht notwendig.

Im Fall, daß es sich um einen schreibenden Zugriff handelt (*store/release*, *keep/active* oder *store/active*) ist keine Änderung notwendig. Nach Abschluß der Änderungen wird geprüft, ob überhaupt noch ein Speicherzugriff auf die gewählte Variable notwendig ist. Wenn dies nicht der Fall ist, so kann (falls es sich um einen errechneten Wert handelt), der Schreibzugriff von *store/active* auf

	T_1	T_2	T_3	T_4	T_5	T_6	T_7
z_2	sa	ua		ur			
z_3		sa	ur			lr	
z_7			sr		la	ur	
z_6				sr			lr
x_0	lr			lr	sa		ur
x_1			la		ur	sa	ur
y							sr

	T_1	T_2	T_3	T_4	T_5	T_6	T_7
z_2	sa	ua		ur			
z_3		sa	ur			lr	
z_7			**sa**		**ua**	ur	
z_6				sr			lr
x_0	lr			lr	sa		ur
x_1			la		ur	sa	ur
y							sr

Abbildung 4.9: Mutationsoperator: Änderung der Zugriffsart

keep/active umgestellt werden. In diesem Fall wird die Variable ausschließlich in Registern gehalten.

Ein Beispiel für die Änderung der Zugriffsart zeigt Abbildung 4.9. Dabei wurde der Zugriff beim Speichern von Variable z_7 in Baum T_3 von *store/release* auf *store/active* geändert und als Konsequenz daraus auch der Zugriff in Baum T_5 von *load/active* auf *use/active* adaptiert.

Wahl der Speicherbank

Dieser Operator verändert die Speicherbank, in der eine (zufällig ausgewählte) Variable gehalten wird. Durch günstige Verteilung auf die verfügbaren Speicherbänke kann bei manchen Prozessoren in der Kompaktierung besserer Code erzeugt werden.

Da bei den meisten Aufgabenstellungen nicht alle Variablen in beliebige Speicherbänke geschrieben werden dürfen (beispielsweise findet man in vielen Systemen Speicherbänke, die nur lesbar und daher lediglich für Konstante geeignet sind), ist die Möglichkeit vorgesehen, zusammen mit dem Gleichungssystem auch für beliebige Variablen eine Speicherbank fest vorzuschreiben.

Registerallokation

Dieser Operator ändert die zugewiesene Registerbank für eine Variable innerhalb eines Fragments. Die Variable, das betroffene Fragment und die neue Registerbank werden gleichverteilt zufällig aus allen zur Verfügung stehenden Ressourcen ausgewählt. Ein Fragment im Sinn dieses Operators beginnt mit dem Laden einer Variable aus dem Speicher bzw. mit dem Berechnen des Ergebnisses und endet, sobald die Registerbank freigegeben wird (ein Zugriff mit dem Status *use/release*). Die neue Registerbank wird allen Zugriffen innerhalb dieses Fragmentes zugewiesen, die Art der Zustände selber bleibt jedoch unverändert.

Abarbeitungsreihenfolge

Dieser Operator vertauscht die Abarbeitungsreihenfolge zweier Bäume. Grundsätzlich werden dabei immer nur zwei benachbarte Bäume vertauscht, um die

	T_1	T_2	T_3	T_4	T_5	T_6	T_7
z_2	sa	ua		ur			
z_3		sa	ur			lr	
z_7			sr		la	ur	
z_6				sr			lr
x_0	lr			lr	sa		ur
x_1			la		ur	sa	ur
y							sr

T_1	T_2	T_4	T_3	T_5	T_6	T_7
sa	ua	**ur**				
	sa		**ur**		lr	
			sr	la	ur	
		sr				lr
lr		**lr**		sa		ur
			la	ur	sa	ur
						sr

Abbildung 4.10: Mutationsoperator: Änderung der Baumreihenfolge

Auswirkungen auf die Registerbelegung möglichst gering zu halten. Die Auswahl der Bäume erfolgt gleichverteilt zufällig.

Ein Austausch ist nur dann zulässig, wenn zwischen den beiden Bäumen keine Datenabhängigkeiten bestehen. Eine Abhängigkeit ergibt sich durch jeweils einen Lese- und einen Schreibzugriff auf die gleiche Variable in den beiden Bäumen. Falls in beiden Bäumen lesend auf die gleiche Variable zugegriffen wird, ist ein Austausch erlaubt, es muß allerdings eine Korrektur der Zugriffsarten nach der folgenden Tabelle vorgenommen werden, in Abhängigkeit von der Zugriffsart bei dem Baum, der von der ersten an die zweite Stelle gewandert ist:

- *load/release*: Im ursprünglich nachfolgenden Baum können als Zustand nur *load/release* oder *load/active* auftreten. Im ersten Fall sind keine Änderungen notwendig, im zweiten Fall müssen die beiden Zugriffsarten gegeneinander vertauscht werden und die Registerbank bei dem nach hinten verschobenen Baum entsprechend angepaßt werden.

- *load/active, use/active*: Im ursprünglich nachfolgenden Baum können als Zustand nur *use/active* oder *use/release* auftreten. Es ist ausreichend, die beiden Zugriffsarten gegeneinander zu vertauschen.

- *use/release*: Im ursprünglich nachfolgenden Baum können als Zustand nur *load/release* oder *load/active* auftreten. Es ist notwendig, die beiden Zugriffsarten auszutauschen und die verwendeten Registerbänke für beide Zugriffe entsprechend anzupassen.

In Abbildung 4.10 ist ein Beispiel für eine Änderung der Baumreihenfolge dargestellt. Die Abarbeitungsreihenfolge der Bäume T_3 und T_4 wurde vertauscht, Anpassungen bei den Variablen waren im konkreten Fall nicht notwendig.

4.5 Bauminterne Optimierung

Für die Optimierung von Trellisbäumen existieren bekannte Algorithmen ([28], [14]), die allerdings nicht die Möglichkeit bieten, Randbedingungen vorzuschrei-

ben, beispielsweise die Tatsache, daß ein bestimmter Operand in einem bestimmten Register erwartet wird. Da vom genetischen Algorithmus in dieser Arbeit allerdings die Registerbänke für sämtliche Interfacevariablen zusammen mit der Zugriffsart vorgeschrieben werden, ist es notwendig, den Trellisalgorithmus zu modifizieren, um diese Randbedingungen berücksichtigen zu können.

Falls Interfacevariablen baumübergreifend in Registern gehalten werden, so stehen diese Register für alle dazwischenliegenden Teilbäume nicht mehr zur Verfügung. Dies kann im Trellisalgorithmus berücksichtigt werden, indem als Ausgangspunkt der Zustand mit der entsprechend verringerten Anzahl an freien Registern gewählt wird, es ist aber keine Änderung am Algorithmus selbst notwendig.

Werden hingegen die Eingangsvariablen für einen Teilbaum nicht (wie in [28] beschrieben) aus dem Hauptspeicher gelesen und im Anschluß daran wieder verworfen, so sind in Abhängigkeit der Zugriffsart entsprechende Anpassungen am Algorithmus notwendig. Generell gilt, daß die Abarbeitung der Trellisbäume bei der Wurzel beginnt, als Ausgangszustand wird also der *nach* der Berechnung des Baumes gültige Zustand verwendet.

- *load/active*: Die Variable wird ab dem Zeitpunkt des Zugriffes bis zum Ende der Abarbeitung (und noch darüber hinaus, was aber für den Trellisalgorithmus irrelevant ist) in einem Register gehalten, das daher für den Trellisalgorithmus nicht mehr zur Verfügung steht. Zu Beginn der Abarbeitung wird daher mit der reduzierten Registeranzahl gearbeitet. Kommt bei einer Verzweigung die Ladeoperation in dem Zweig zu liegen, der als zweiter bearbeitet wird, so steht für die Operationen im anderen Zweig ein Register mehr zur Verfügung (da die Ladeoperation erst im Anschluß daran ausgeführt werden kann). Daher ist der Zustand für die Berechnung des Teilbaumes entsprechend anzupassen.

- *load/release*: Bei dieser Zugriffsart handelt es sich um den klassischen Trellisalgorithmus, es sind keine Anpassungen notwendig

- *use/active*: Bei dieser Zugriffsart wird das Zwischenergebnis während der gesamten Auswertung des Baumes im Register gehalten. Es ist daher lediglich eine Änderung des Anfangszustandes erforderlich, da konstant ein Register weniger zur Verfügung steht. Bei der Erzeugung des Codes muß darauf geachtet werden, daß für diese Variable kein Ladebefehl benötigt wird.

- *use/release*: Die Variable befindet sich zunächst in einem Register, das aber unmittelbar nach dem Zugriff freigegeben wird und wieder für Zwischenergebnisse zur Verfügung steht. Aus der Sicht des Trellisalgorithmus bedeutet das, daß das Register zunächst als frei betrachtet wird (da die Abarbeitung bei der Wurzel des Baumes beginnt). Kommt in einer Verzweigung der Zugriff in dem Zweig zu liegen, der als zweiter bearbeitet wird, so steht für den anderen Zweig ein Register weniger zur Verfügung,

da dieses zur Zwischenspeicherung des Operanden benötigt wird. Der Zustand für die Berechnung des Teilbaumes ist daher entsprechend anzupassen, indem das betreffende Register als belegt gekennzeichnet wird.

Die bei der Ausgangsvariablen verwendete Zugriffsart erfordert innerhalb der Baumbearbeitung keine Anpassungen. Bei den Zugriffsarten *store/active* und *keep/active* ist es allerdings notwendig, den Ausgangszustand für die Bearbeitung des nachfolgenden Baumes anzupassen, da das entsprechende Register dann nicht mehr zur Verfügung steht.

Bei aktivierter Schleifenoptimierung müssen beim Ausgangszustand für den ersten Baum alle schleifenübergreifenden Variablen entsprechend berücksichtigt werden.

4.6 Schleifenoptimierung

In vielen Fällen werden die implementierten Algorithmen nicht nur ein einziges Mal durchlaufen, sondern innerhalb einer Schleife abgearbeitet, die entweder sehr oft, oder in manchen Fällen endlos abgearbeitet wird. In diesen Anwendungsbereichen sind die Initialisierungskosten gegenüber den Kosten innerhalb der Schleife vernachlässigbar, wodurch sich andere Optimierungskriterien ergeben.

Im Abschnitt 4.4.3 wurden die verschiedenen Möglichkeiten dargestellt, um auf Variable zuzugreifen. Enthält ein Datenflußgraph Verzögerungselemente, so hat das zur Folge, daß die Variable zuerst gelesen (und für Berechnungen verwendet) wird und erst anschließend der neu berechnete Wert zum Abspeichern zur Verfügung steht. In diesem Fall ist der Wert der Variablen für den nächsten Programmdurchlauf von Bedeutung, er wird daher in den Hauptspeicher geschrieben (das entspricht den Zugriffsarten *store/release* oder *store/active*).

Wenn der Algorithmus innerhalb einer Schleife ausgeführt wird, stehen die Register des Prozessors auch schleifenübergreifend zur Verfügung. In diesem Fall darf daher auch das Attribut *keep/active* verwendet werden, wodurch ein Assemblerbefehl eingespart werden kann. Durch diesen Schritt ist die Schleifenoptimierung unmittelbar an den genetischen Algorithmus angekoppelt, eine Phasenproblematik kann daher grundsätzlich vermieden werden.

Durch die Schleifenoptimierung sind auch bestimmte Modifikationen in den anderen Programmteilen notwendig, die jeweils an entsprechender Stelle beschrieben werden.

Im Rahmen dieser Arbeit wird lediglich die Lebensdauer von Variablen über die Schleifengrenzen hinweg optimiert. Durch die dadurch vermeidbaren Lese- und Schreibzugriffe auf den Hauptspeicher können Programmschritte eingespart werden. Grundsätzlich ist eine ähnliche Erweiterung auch bei der Parallelisierung möglich, indem man die letzten Befehle des Programmes an den Anfang stellt, falls sich dadurch Einsparungen (durch Ausnützen von Parallelbefehlen) ergeben. Da der Gewinn unabhängig von der Programmlänge lediglich bei einer, in ganz seltenen Fällen bei zwei Instruktionen liegt, wurde in dieser Arbeit auf

die Implementierung dieses Optimierungsschrittes verzichtet. Es bietet sich hier allerdings eine Möglichkeit zur weiteren (wenn auch geringfügigen) Leistungssteigerung an.

4.7 Registerallokation

Mit der Registerallokation erfolgt die tatsächliche Zuteilung eines konkreten Registers für jeden Operanden, während in den bisherigen Abschnitten lediglich die Registerbank festgelegt worden ist.

Ist die Schleifenoptimierung abgeschaltet, so kann die Registerzuweisung auf first-come-first-serve Basis erfolgen und jeder Operand wird in dem verfügbaren Register mit der niedrigsten Nummer untergebracht. Es ist dabei lediglich notwendig, eine Tabelle mit dem aktuellen Belegungsstand aller Register zu führen. Der Trellisalgorithmus garantiert, daß immer gültige Zuordnungsschemata gefunden werden.

Im Gegensatz dazu erfordert die Schleifenoptimierung ein aufwendigeres Verfahren. Es kann mit dem vorher beschriebenen Algorithmus nicht garantiert werden, daß für schleifenübergreifende Variablen zu Beginn und am Ende der Abarbeitung das gleiche Register verfügbar ist. Diese Forderung ist jedoch notwendig für ein gültiges Programm.

Die Allokation erfolgt in zwei Stufen. Im ersten Teil werden die Lebensdauern aller in Registern gespeicherten Operanden ermittelt und in einer Tabelle festgehalten. Dabei werden auch Lebensdauern, die über die Schleifengrenzen hinweg gehen, korrekt erkannt. Weiters werden Variable, auf die ausschließlich lesend zugegriffen wird, mit einer Lebensdauer von unendlich versehen (das entsprechende Register darf nie überschrieben werden).

Im zweiten Durchlauf wird für jeden Operanden ein freies Register ermittelt, im Gegensatz zum vorher geschilderten first-come-first-serve Verfahren ist hier aber bereits vorweg die Lebensdauer bekannt. Eine Erkennung von Zugriffskonflikten auf Grund einer schleifenübergreifenden Lebensdauer ist also möglich.

Es können Lebensdauertabellen angegeben werden, für die keine gültige Registerbelegung gefunden werden kann (solche Belegungen entstehen durch sich zyklisch überlappende Lebensdauern innerhalb der gleichen Registerbank. Für Schleifenabarbeitung ist es notwendig, in jedem Zyklus die gleichen Register zuzuweisen. Das kann aber durch hohe Auslastung der Registerbank unmöglich gemacht werden). Da diese Fälle zu einem früheren Zeitpunkt nicht erkennbar sind, werden sie im Rahmen der Registerallokation abgefangen, das Individuum wird als ungültig gekennzeichnet und an den genetischen Algorithmus zurückgegeben. Die Selektionsmechanismen innerhalb des genetischen Algorithmus verhindern, daß sich die entsprechenden Lösungen ausbreiten.

4.8 Kompaktierung

Die Parallelisierung des erzeugten Assemblercodes ist ein wesentlicher Schritt innerhalb der Optimierung. Da bei den handelsüblichen Prozessoren meist bis zu drei Befehle parallel zueinander ausgeführt werden können, ist im Optimalfall eine Reduktion der Programmgröße und der Ausführungszeit auf ein Drittel des Ausgangswertes möglich. In der Praxis wird dieser Wert jedoch nie erreicht, weil beispielsweise bei der verwendeten Hardware zu jedem Zeitpunkt nur eine arithmetische Operation ausgeführt werden kann. Beispiele dafür finden sich in Kapitel 6 dieser Arbeit.

Das Optimierungspotential durch den Kompaktierungsschritt wird von verschiedenen Gesichtspunkten beeinflußt:

- *Wahl der Speicherbänke*: Bei fast allen Architekturen ist es so, daß Variable nur dann gleichzeitig aus dem Speicher geladen, bzw. in diesen zurückgeschrieben werden können, wenn sie in unterschiedlichen Speicherbänken abgelegt sind. Dadurch kommt der Wahl der Speicherbank eine zentrale Bedeutung im Hinblick auf die Optimierbarkeit des Codes zu, sie ist aber weitgehend unabhängig von den vorherigen Optimierungsschritten.

- *Wahl der Registerbänke*: Bei verschiedenen Lade- bzw. Speicherbefehlen ist es notwendig, eine zur Speicherbank passende Registerbank auszuwählen, um das Potential der Assemblersprache voll ausnützen zu können. Die Wahl der Registerbank ist daher ebenfalls wesentlich für die Qualität der Parallelisierung, sie beeinflußt allerdings auch stark die bauminterne Optimierung.

- *Abarbeitungsreihenfolge der Trellisbäume*: Durch die Wahl der Baumreihenfolge werden Datenabhängigkeiten verändert. Beispielsweise kann ein Ergebnis bereits früher in ein Register geladen werden, wenn in der Zwischenzeit nicht mehr darauf zugegriffen wird. Unter Umständen ist auf diese Art das zusätzliche Ausnützen von Parallelitäten möglich. Dieser Aspekt ist von Bedeutung für die Kompaktierung, er wirkt sich aber auch auf die Anzahl der verfügbaren Register innerhalb der Bäume aus und stellt daher teilweise starke Einschränkungen an die bauminterne Optimierung.

Alle der genannten Aspekte werden durch den übergreifenden genetischen Algorithmus kontrolliert, daher ist eine globale Optimierung unter Berücksichtigung der vorhandenen Abhängigkeiten zur bauminternen Optimierung möglich.

4.8.1 Datenabhängigkeiten

Beim Kompaktieren kann der Fall auftreten, daß zwei Befehle, die gleichzeitig ausgeführt werden sollen, auf das selbe Register zugreifen. Hier hängt es von der verwendeten Hardware und von der Art des Zugriffs ab, ob die Parallelisierung zulässig ist, oder nicht. Bei den meisten gängigen Prozessoren sind die

```
1: mpy X0,X0,A
2: move A,Y1
3: mpy X1,Y1,B
```

Abbildung 4.11: Starke Datenabhängigkeit bezüglich A bzw. $Y1$

Datenregister als sogenannte Dual-Port-Register ausgeführt. Das bedeutet, daß innerhalb eines Befehls sowohl der alte Registerinhalt gelesen, als auch das Register mit einem neuen Inhalt beschrieben werden kann, was bestimmte Formen der Parallelisierung ermöglicht. Konkret muß man zwischen den folgenden vier Fällen unterscheiden:

Starke Datenabhängigkeit

Eine solche Datenabhängigkeit liegt vor, wenn ein Rechenwert mit einer Assemblerinstruktion in ein Register geschrieben und von einer anderen Instruktion aus dem gleichen Register wieder ausgelesen werden soll. Obwohl grundsätzlich Schreib- und Lesezugriff auf das gleiche Register im gleichen Taktzyklus gestattet sind, ist das in diesem Fall nicht zulässig, da die Leseoperation stets den *alten* Registerinhalt ausliest, das gewünschte Ergebnis somit aber erst im nachfolgenden Taktschritt zur Verfügung steht.

Abbildung 4.11 zeigt zwei Beispiele für eine starke Datenabhängigkeit in der Assemblersprache des Motorola DSP 56k. Obwohl es möglich ist, einen mov-Befehl parallel zu einem mpy-Befehl auszuführen, darf der Transferbefehl in diesem konkreten Beispiel weder zeitgleich mit der einen, noch mit der anderen Multiplikation ausgeführt werden. Im ersten Fall würde das Register $Y1$ mit dem falschen - weil alten - Inhalt von A geladen, im zweiten Fall würde der falsche - weil alte - Inhalt von $Y1$ für die Berechnung der Multiplikation verwendet werden.

Schwache Datenabhängigkeit

Von einer schwachen Datenabhängigkeit wird gesprochen, wenn ein Rechenwert in ein Register geschrieben wird, in dem noch ein älteres Ergebnis zwischengespeichert ist, das vorher ausgelagert oder in einer arithmetischen Operation verarbeitet werden muß. Hier ist der gleichzeitige Zugriff von Lese- und Schreiboperation erlaubt und aus Effizienzgründen sogar wünschenswert, da die Schreiboperation den vorigen Wert erst *nach* dem Auslesen zerstört.

Abbildung 4.12 zeigt ein Beispiel für eine schwache Datenabhängigkeit zwischen dem zweiten und dem dritten Befehl. Diese beiden Befehle können parallel ausgeführt werden, weil der alte Inhalt des Registers A gesichert wird, bevor er von der Multiplikation mit dem neuen Ergebnis überschrieben wird. Der Wegfall des Schlüsselwortes *move* im Zuge der Parallelisierung ist durch die Assemblersyntax von Motorola bedingt und hat keine algorithmische Bedeutung, ebenso ist die Tatsache, daß der Befehl zur Sicherung von Register A *hinter* der Multi-

Sequentiell:

```
1: mpy X0,Y0,A
2: move A,X0
3: mpy X1,Y1,A
```

Parallel:

```
1: mpy X0,Y0,A
2: mpy X1,Y1,A␣A,X0
```

Abbildung 4.12: Schwache Datenabhängigkeit bezüglich A

plikation geschrieben wird, bedeutungslos, da beide Befehle vollkommen parallel ausgeführt werden.

Paralleles Lesen

Das gleichzeitige Lesen eines Datenwertes durch mehrere parallel ausgeführte Befehle ist erlaubt und stellt daher *keine* Datenabhängigkeit dar. Die betroffenen Befehle können sowohl parallel als auch in beliebiger Reihenfolge sequentiell ausgeführt werden.

Paralleles Schreiben

Gleichzeitiges Schreiben von zwei Assemblerbefehlen in ein und dasselbe Register ist unzulässig. Für den Kompaktierungsalgorithmus spielt diese Einschränkung allerdings keine Rolle, da in einem normalisierten Programm ohnehin jeder berechnete Wert auch weiter verwendet wird, wodurch eine Leseoperation vor der nächsten Schreiboperation erzwungen wird.

4.8.2 List scheduling

Unter Berücksichtigung der Randbedingungen, die durch die oben beschriebenen Abhängigkeiten auftreten, kann der List Schedule Algorithmus geeignet modifiziert werden. Im ersten Schritt ist aus dem sequentiellen Code ein Abhängigkeitsgraph zu erstellen, der dann in einem zweiten Schritt in parallelen Code übergeführt wird.

Zur Erstellung des Abhängigkeitsgraphen sind für jeden sequentiellen Befehl die folgenden Schritte durchzuführen. In der Implementierung erfolgt die Suche nach Abhängigkeiten idealerweise getrennt von der Bestimmung der Gewichtungsfaktoren, da für letztere dann bereits auf die vorhandenen Ergebnisse zurückgegriffen werden kann. Als begleitendes Beispiel wird Motorola Assemblercode für den Ausdruck $d = b + c * (a + b)$ erzeugt. Der sequentielle Code ist in Abbildung 4.13 angegeben. Dieses Beispiel dient nur zur Illustration des angepaßten List Scheduling Algorithmus und ist nicht die optimale Lösung für den gegebenen Ausdruck.

1. Suchen aller starken Abhängigkeiten, das sind alle Instruktionen, mit denen Quelloperanden des untersuchten Befehls berechnet werden. Tatsäch-

```
1: move X:a,Y0
2: move X:b,A
3: add Y0,A
4: move A,X0
5: move X:c,Y1
6: mpy X0,Y1,A
7: move A,Y1
8: move X:b,B
9: add Y1,B
10: move B,Y:d
```

Abbildung 4.13: Sequentieller Assemblercode für $d = b + c * (a + b)$

lich besteht natürlich auch eine starke Abhängigkeit zu allen vorangehenden Instruktionen, die das gleiche Register verwenden. Für eine korrekte Parallelisierung ist jedoch nur die jeweils nächste Abhängigkeit von Bedeutung.

Die starken Abhängigkeiten des Beispielausdrucks sind im linken Teil von Abbildung 4.14 angeführt.

2. Suchen aller schwachen Abhängigkeiten, das sind alle vorangehenden Instruktionen, die andere Variablen aus einem Register lesen, das vom untersuchten Befehl verwendet wird. Tatsächlich reduzieren sich die relevanten schwachen Abhängigkeiten auf den letzten vorangegangenen Lesezugriff des Zielregisters des gerade untersuchten Befehls, da in den Quellregistern bereits der gewünschte Operand vorhanden sein muß. Es besteht dort daher eine starke Abhängigkeit zu einer vorherigen Instruktion, die auf alle Fälle strenger ist, als es die schwache Abhängigkeit sein könnte.

   Im verwendeten Beispiel existiert nur eine einzige schwache Abhängigkeit (siehe den mittleren Teil von Abbildung 4.14). Das ist nicht typisch für den erzeugten Code und liegt an der Kürze des gegebenen Beispiels. Schwache Abhängigkeiten treten typischerweise bei der Wiederverwendung bereits früher belegter Register auf.

3. Bestimmen des Gewichtungsfaktors des untersuchten Befehls. Das Gewicht errechnet sich aus der Anzahl der Befehle, die von der gerade betrachteten Instruktion stark oder schwach abhängig sind. Befehle, die im Abhängigkeitsgraphen doppelt aufscheinen, werden auch doppelt in die Gewichtung einbezogen. Die Gewichtungsfaktoren des Beispielausdrucks sind im rechten Teil der Abbildung 4.14 dargestellt.

Durch diese Schritte erhält man einen gewichteten Abhängigkeitsgraphen, der anschließend dazu verwendet wird, einen optimierten parallelen Code zu erzeugen. Dazu wird wie folgt vorgegangen:

starke Abhängigk.:		schwache Abhängigk.:		Gewichtungsfakt.:	
1:	-	1:	-	1:	9
2:	-	2:	-	2:	9
3:	1,2	3:	-	3:	8
4:	3	4:	-	4:	7
5:	-	5:	-	5:	4
6:	4,5	6:	-	6:	3
7:	6	7:	4	7:	2
8:	-	8:	-	8:	2
9:	7,8	9:	-	9:	1
10:	9	10:	-	10:	0

Abbildung 4.14: Abhängigkeiten und Gewichtungsfaktoren für den Assemblercode aus Abbildung 4.13

1. Suchen aller ausführbaren Anweisungen:
   Ausführbar sind alle Anweisungen, die keine *starke* Abhängigkeit zu noch nicht ausgeführten Befehlen besitzen. Schwache Abhängigkeiten sind zu diesem Zeitpunkt noch erlaubt. Im ersten Durchlauf sind das für den Beispielcode die Befehle 1, 2, 5 und 8.

2. Sortieren der gefundenen Anweisungen:
   Die ausführbaren Anweisungen werden nach absteigendem Gewicht sortiert, so daß der Befehl, von dem die meisten weiteren Befehle abhängen, an erster Stelle steht. Im konkreten Beispiel bleibt die Reihenfolge durch das Sortieren unverändert.

3. Verwenden der Instruktion mit dem größten Gewicht:
   Dieser Befehl wird als Grundlage für den zu bildenden Mehrfachbefehl herangezogen. Für den Beispielausdruck ist das der Befehl 1.

4. Prüfen der weiteren Befehle auf Parallelisierbarkeit:
   Damit einer der weiteren ausführbaren Befehle tatsächlich parallelisiert werden kann, darf eine allfällige schwache Abhängigkeit nur noch zu Befehlen bestehen, die bereits im zu bildenden Mehrfachbefehl enthalten sind. Außerdem müssen die beteiligten Instruktionen in der Hardwarebeschreibung als gemeinsam ausführbar gekennzeichnet sein. Da im gegebenen Beispiel alle ausführbaren Anweisungen aus der gleichen Speicherbank lesen, ist eine Parallelisierung nicht möglich. Würde bei einem der drei verbleibenden Befehle die andere Speicherbank genutzt, so könnte die entsprechende Instruktion für einen Mehrfachbefehl genutzt werden.

5. Parallelisieren:
   Sind die Voraussetzungen aus Punkt 4 erfüllt, wird die Instruktion zum aktuellen Mehrfachbefehl hinzugefügt, andernfalls verbleibt sie in der Liste der ausführbaren Befehle. In beiden Fällen wird mit Punkt 4 beim

```
1:  move X:a,Y0
2:  move X:b,A
3:  add Y0,A      X:c,Y1
4:  move A,X0
5:  mpy X0,Y1,A   X:b,B
6:  move A,Y1
7:  add Y1,B
8:  move B,Y:d
```

Abbildung 4.15: Paralleler Assemblercode für $d = b + c * (a + b)$

nächsten als ausführbar gekennzeichneten Befehl fortgesetzt.

6. Sind keine weiteren ausführbaren Befehle mehr vorhanden, so wird die Bearbeitung des aktuellen Mehrfachbefehls abgeschlossen. Dieser wird in den endgültigen Programmcode aufgenommen, ein neuer, leerer Mehrfachbefehl wird angelegt und der Algorithmus mit Punkt 1 fortgesetzt.

7. Diese Schritte werden so lange wiederholt, bis alle Einfachbefehle abgearbeitet worden sind. Die Menge aller erzeugten Mehrfachbefehle ergibt nun in der Reihenfolge ihrer Erstellung das kompaktierte Programm. Der parallele Code für den hier verwendeten Beispielausdruck ist in Abbildung 4.15 dargestellt. Gegenüber dem sequentiellen Code aus Abbildung 4.13 konnten 2 Befehlsschritte durch Parallelisierung eingespart werden.

Kapitel 5

Ergebnisse

Zur Untersuchung des Verhaltens des entwickelten Algorithmus ist es notwendig, verschiedene Tests mit unterschiedlichen Parameterwerten durchzuführen. Die im folgenden angeführten Beispiele sind alle für den DSP 56k von Motorola berechnet worden und verwenden als Gleichungssystem ein Zustandsraumfilter zweiter Ordnung. Details zu dieser Aufgabenstellung sind neben weiteren Beispielen im Kapitel 6.2 zu finden.

Alle in diesem Kapitel durchgeführten Simulationen verwenden die folgenden Parameter, etwaige Abweichungen werden jeweils explizit angeführt:

- Populationsgröße: 500
- Crossoverwahrscheinlichkeit: 0.9
- Mutationswahrscheinlichkeit: 0.9
- Skalierung: linear
- Selektionsdruck: 50
- Elitismus: aktiviert

In den folgenden Abbildungen wird häufig die Qualität des erzeugen Codes in Abhängigkeit verschiedener Parameter dargestellt. Um einen sinnvollen Wertebereich darstellen zu können, aber trotzdem irreführende Nullpunktsunterdrückung zu vermeiden, wird dabei die Codequalität auf die bestmögliche, mit dem Compiler erzielbare Lösung (die für das Versuchsbeispiel bekannt ist, nämlich 10 Instruktionen) bezogen. Der dargestellte Wert gibt in diesem Fall die Anzahl der über die optimale Lösung hinaus benötigten Assemblerinstruktionen an.

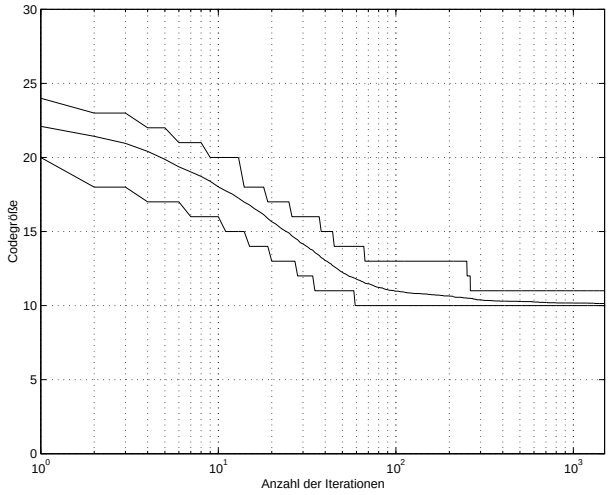

Abbildung 5.1: Konvergenzverhalten

5.1 Einfluß der genetischen Parameter

5.1.1 Iterationsanzahl und Konvergenz

Die Anzahl der durchgeführten Iterationen ist eigentlich kein Parameter des genetischen Algorithmus, sondern eine Konsequenz aus der gewünschten Qualität des Ergebnisses und der Konvergenzgeschwindigkeit des Optimierungsvorganges. Je besser die Lösung sein soll, bzw. je langsamer der Algorithmus konvergiert, desto mehr Iterationen sind notwendig, um zum gewünschten Resultat zu kommen.

Abbildung 5.1 zeigt typische Konvergenzdaten, die aus 100 Compilerläufen (mit identischen Parametern) gewonnen wurden. Es werden dabei für jede Iterationszahl das beste, das schlechteste, sowie das durchschnittliche Ergebnis dargestellt. Man erkennt, daß zu Beginn des Optimierungsvorganges einige Iterationen benötigt werden, bis sich die zufällig initialisierte Population bereinigt hat. Im Anschluß daran folgt ein linearer Verlauf der Konvergenzkurve, der bedingt durch die logarithmische Skalierung einer sich exponentiell verlangsamenden Verbesserung des Ergebnisses entspricht. Nach einer gewissen Zeit sättigt die Optimierung und zeigt auch bei massiver Vergrößerung der Iterationszahl nur noch marginale Verbesserungen des Ergebnisses. Dabei ist zu beachten, daß der Übergang vom exponentiell abnehmenden Verlauf zur Sättigung relativ langsam erfolgt, da bedingt durch die Mutationen stets noch zufällige Verbesserungen des Ergebnisses erzielt werden, allerdings mit deutlich geringerer Geschwindigkeit.

Zu beachten ist, daß der beste bzw. der schlechteste Optimierungslauf (abgesehen von den ersten Iterationsschritten, die hauptsächlich durch die zufällige Initialisierung bestimmt werden) in einem relativ engen Abstand zur gemit-

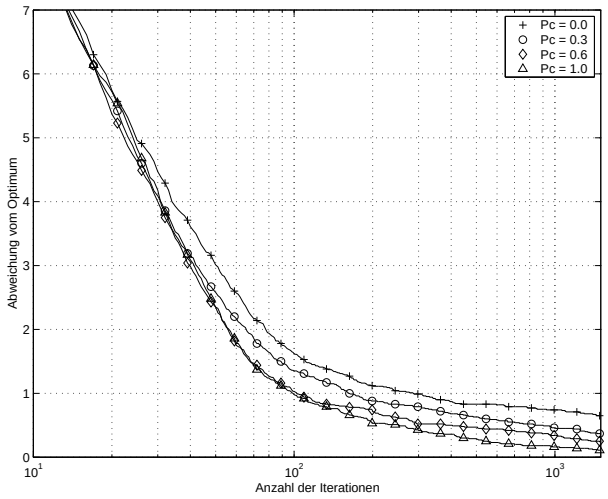

Abbildung 5.2: Konvergenzverhalten bei unterschiedlicher Crossoverrate

telten Kurve liegen. Dieses Verhalten ist nicht nur auf das gezeigte Beispiel beschränkt, sondern typisch für den Algorithmus. Man kann daher mit einer relativ hohen Sicherheit aus einem einzigen Compilerlauf auf die Charakteristik des untersuchten Problems schließen, insbesondere auf die Anzahl der notwendigen Iterationen.

Die Steigung, Lage des Sättigungspunktes und Qualität des erreichten Endwertes hängen von den Parametern des genetischen Algorithmus ab, die in den folgenden Abschnitten näher untersucht werden.

5.1.2 Crossoverwahrscheinlichkeit

Der Crossover Operator ist der Kernbestandteil des genetischen Teiles der vorliegenden Arbeit. Er bewirkt eine gute Durchmischung des Erbmaterials von einer Generation zur nächsten. Dementsprechend ist auch die Wahl einer geeigneten Crossoverwahrscheinlichkeit von zentraler Bedeutung. Wird diese Wahrscheinlichkeit zu klein gewählt, so degeneriert der Algorithmus (durch die Kombination aus Mutationsoperator und Selektion) zu einer Zufallssuche. Auch mit diesen Parametern konvergiert der Algorithmus, allerdings deutlich langsamer und mit schlechterem Endergebnis. Bei sehr hohen Werten für die Crossoverrate besteht dagegen die Gefahr, daß bewährte Chromosomen zu oft zerstört werden und die Population daher nicht gegen das globale Optimum konvergiert.

In Abbildung 5.2 sind einige Beispiele für unterschiedliche Crossoverraten dargestellt, durch die die Betrachtungen des vorigen Abschnittes illustriert werden. Bei zunehmender Crossoverwahrscheinlichkeit verbessert sich demnach das erreichte Ergebnis. Deutlicher wird dieser Effekt in Abbildung 5.3 dargestellt, hier ist die Qualität des Ergebnisses direkt in Abhängigkeit von der Crossoverra-

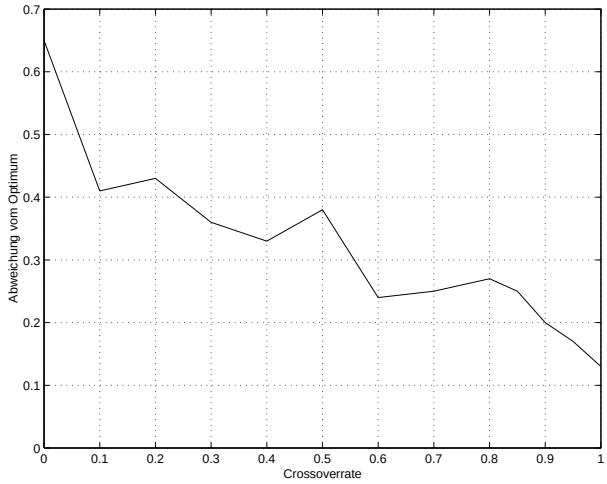

Abbildung 5.3: Codequalität bei unterschiedlicher Crossoverrate

te aufgetragen. Bei kleinen Werten für p_c werden deutlich schlechtere Resultate erzielt, ab etwa $p_c = 0.6$ kommt es jedoch zu einer Sättigung, lediglich mit extrem hohen Crossoverraten kann noch einmal eine leichte Verbesserung erzielt werden. Diese Werte unterliegen einer relativ starken Streuung, man kann daher die Ergebnisse für $p_c = 0.6$ bis $p_c = 0.9$ als etwa konstant betrachten. Die hohe Qualität der Ergebnisse auch bei hohen Crossoverraten erklärt sich dadurch, daß in der Population identische Individuen zulässig sind. Besonders dann, wenn der Algorithmus das Sättigungsniveau erreicht, besteht ein vergleichsweise großer Anteil der Population aus identischen Chromosomen, welche dann auch durch den Crossover Operator nicht geändert werden (und die effektive Crossoverrate entsprechend reduzieren).

5.1.3 Mutationsrate

Durch die Mutationsrate wird verhindert, daß bestimmte Genkombinationen im Lauf der Optimierung aus der Population verschwinden und dadurch das Finden des globalen Optimums verhindern. Üblicherweise wird die Mutationsrate relativ gering (etwa in der Größenordnung einiger Prozent) gewählt. Bei der vorliegenden Aufgabe ist es wegen der Ausführung des Crossover Operators jedoch notwendig, mit deutlich höheren Mutationsraten zu arbeiten. Der Crossover Operator entspricht bei der Auswahl der Speicherbänke vollkommen, bei der Bestimmung der Abarbeitungsreihenfolge annähernd dem one point Crossover. Bei der Auswahl der Zugriffsmethoden hingegen ist es nicht möglich, die beiden Individuen ohne Seiteneffekte zu kreuzen. Daher wird auch ein eigener Reparaturalgorithmus benötigt, der entstandene Inkonsistenzen behebt (siehe Kapitel 4.4.2. Auf Grund dieser Abweichungen wird keine so gute Durchmi-

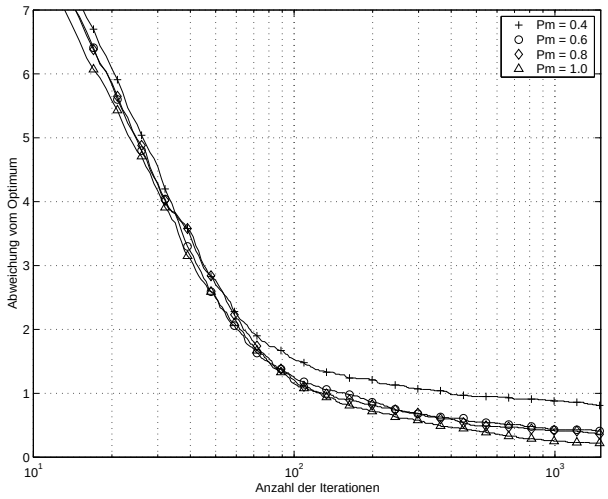

Abbildung 5.4: Konvergenzverhalten bei unterschiedlicher Mutationsrate

schung des Erbmaterials erreicht, wie es für die Optimierung notwendig wäre, weiters besteht das Risiko, bestimmte Genkombinationen durch den Reparaturalgorithmus zu unterdrücken. Daher wird mit einer hohen Mutationsrate in dem Bereich der Zugriffsart versucht, diese negativen Effekte zu kompensieren.

In Abbildung 5.4 werden die Konvergenzkurven verschiedener Testläufe mit unterschiedlichen Mutationsraten verglichen. Man erkennt, daß bei zu geringen Mutationsraten lediglich lokale Minima erreicht werden, da bestimmte Genkombinationen im Erbgut verlorengehen.

Die Grafik in Abbildung 5.5 zeigt, daß ein optimales Resultat ab einer Mutationsrate von etwa $p_m = 0.9$ erreicht wird. Noch höhere Mutationsraten liefern im Rahmen der Testgenauigkeit vergleichbare Ergebnisse, tragen aber zur einer höheren Rechenzeit bei (siehe Kapitel 5.3.2) und sind daher nicht sinnvoll.

5.1.4 Selektion

Im Rahmen der Selektion ist eine Beeinflussung durch die Parameter möglich, die die Skalierungsmechanismen betreffen, wie sie in Kapitel 4.4.1 beschrieben werden. Es handelt sich dabei einerseits um den Selektionsdruck, um den Elitismus, der entweder ein- oder ausgeschaltet werden kann, sowie um die Art der Skalierung (im Rahmen dieser Arbeit werden lineare und logarithmische Skalierung betrachtet). In den folgenden Abschnitten werden diese Parameter jeweils getrennt voneinander untersucht, abschließend wird eine kurze Zusammenfassung gegeben, um auch auf gegenseitige Beeinflussungen einzugehen.

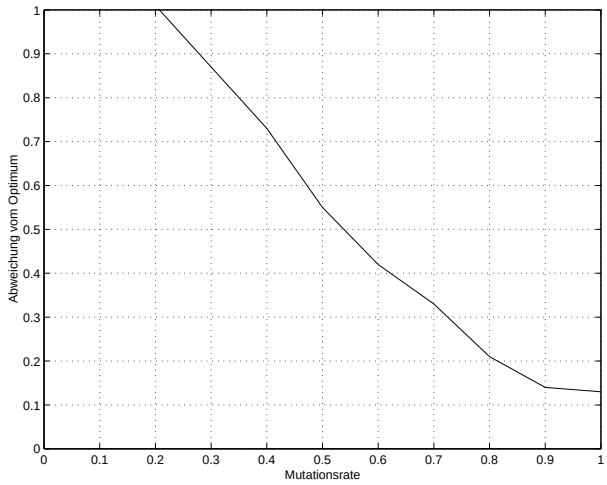

Abbildung 5.5: Codequalität bei unterschiedlicher Mutationsrate

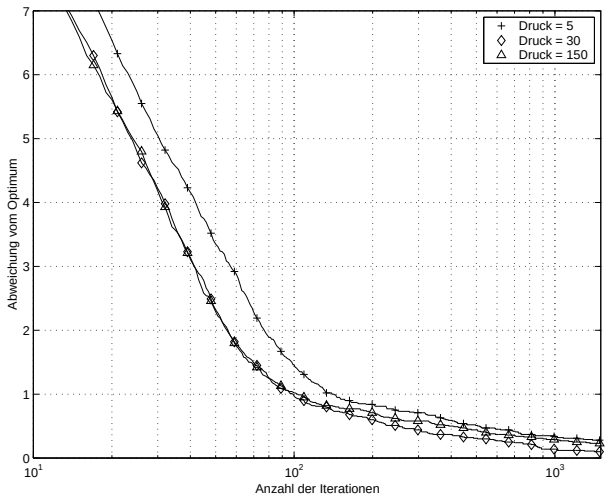

Abbildung 5.6: Konvergenzverhalten bei unterschiedlichem Selektionsdruck

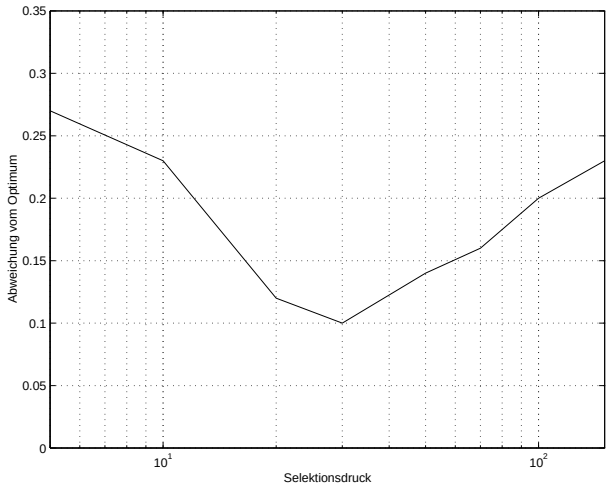

Abbildung 5.7: Codequalität bei unterschiedlichem Selektionsdruck

Selektionsdruck

Wie bereits im Kapitel 3.5 erklärt, wird mit dem Selektionsdruck die Qualität und Geschwindigkeit der Konvergenz des Verfahrens eingestellt. In Abbildung 5.6 sind die Konvergenzkurven für die Werte 5, 30 und 150 des Selektionsdrucks dargestellt. Man erkennt deutlich, daß bei sehr geringem Druck das Verfahren von Beginn an schlechter konvergiert, da weniger fitte Lösungen nicht ausreichend unterdrückt werden. Die anderen beiden Testläufe verhalten sich während der ersten etwa 100 Iterationen annähernd gleich, bei zu hohem Selektionsdruck kommt es aber häufiger zum Auffinden eines lokalen Minimums, das (weil alternative, geringfügig schlechtere Lösungen stark unterdrückt werden) nicht mehr verlassen werden kann.

Eine Aufstellung der durchschnittlichen Ergebnisse nach 1500 Iterationen in Abhängigkeit des Selektionsdrucks wird in Abbildung 5.7 gezeigt. Es ist erkennbar, daß es ein Optimum für die Wahl des Selektionsdrucks gibt, das etwa bei $p = 30$ liegt. Der Verlauf des Minimums ist relativ flach, die Wahl des Selektionsdrucks entsprechend unkritisch.

Elitismus

Die Ergebnisse im vorigen Abschnitt wurden mit aktiviertem Elitismus erzielt, was bedeutet, daß das jeweils beste Individuum einer Population in die neue Population übernommen wird, unabhängig vom zufallsgesteuerten Selektionsmechanismus. Es ist naheliegend zu glauben, daß die Optimierung mit aktiviertem Elitismus zu besseren Ergebnissen führt, da andernfalls bereits erzielte gute Lösungen wieder verloren gehen könnten. Die folgenden Simulationen zeigen allerdings ein genau gegenteiliges Verhalten.

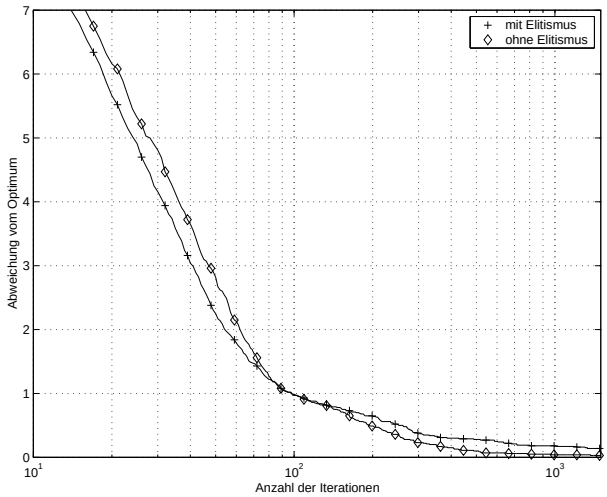

Abbildung 5.8: Einfluß von Elitismus auf das Konvergenzverhalten

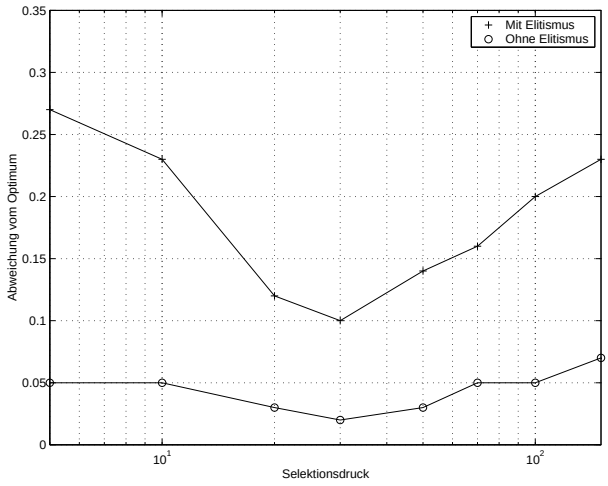

Abbildung 5.9: Einfluß von Elitismus bei unterschiedlichem Selektionsdruck

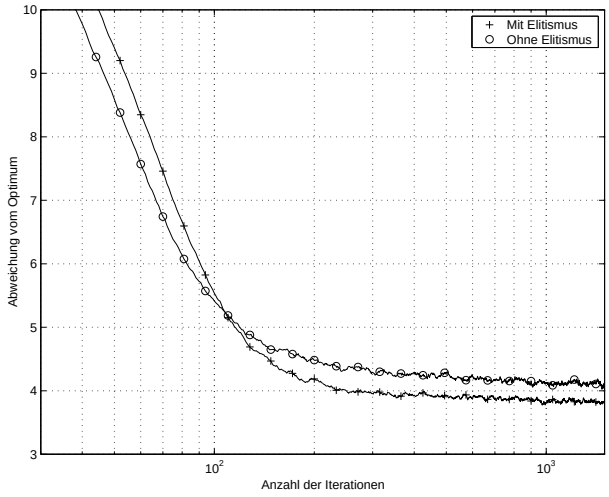

Abbildung 5.10: Vergleich der durchschnittlichen Population mit bzw. ohne Elitismus

In Abbildung 5.8 sind Konvergenzkurven für ein- bzw. ausgeschalteten Elitismus bei ansonsten identischen Simulationsparametern dargestellt. Man erkennt, daß die Konvergenz mit Elitismus rascher erzielt wird, allerdings auf einem höheren Endwert sättigt. Dieses Resultat ist, wie in Abbildung 5.9 gezeigt wird, unabhängig vom verwendeten Selektionsdruck.

Eine nähere Untersuchung der Individuen innerhalb einer Population zeigt die Ursachen für dieses Konvergenzverhalten auf: im Zuge der normalen Crossover- und Mutationsoperationen treten immer wieder Individuen auf, deren Fitnesswert deutlich besser als der Populationsdurchschnitt ist. Bei Verwendung von Elitismus ist für diese Individuen das Überleben innerhalb der Population unter allen Umständen gesichert, sie erzeugen daher überproportional viele Nachkommen, die durchschnittliche Fitness der Population steigt innerhalb weniger Zyklen stärker, als sonst. Dadurch werden Lösungen, die sonst noch akzeptable Überlebenschancen gehabt hätten, zurückgedrängt und ihr Erbmaterial geht verloren. Ohne Elitismus ist die beste Fitness innerhalb der Population zwar nicht stetig (es kommt immer wieder zu Rückschritten), aber die durchschnittliche Fitness verläuft wesentlich glatter und stetiger. Dadurch ist insgesamt die Chance, das globale Optimum zu erkennen, größer.

Die Konvergenz der Population läßt sich an Abbildung 5.10 erkennen, das die durchschnittliche Fitness der Population mit und ohne Elitismus darstellt, wieder über 100 Testläufe gemittelt. Im Vergleich zu Abbildung 5.8 zeigen sich keine auffälligen Unterschiede, abgesehen davon, daß die durchschnittliche Fitness natürlich stets höher liegt als die optimale Fitness.

Betrachtet man anstatt der gemittelten Kurve das Konvergenzverhalten eines einzelnen Testlaufes, so kann man mit Elitismus erhebliche Abweichungen

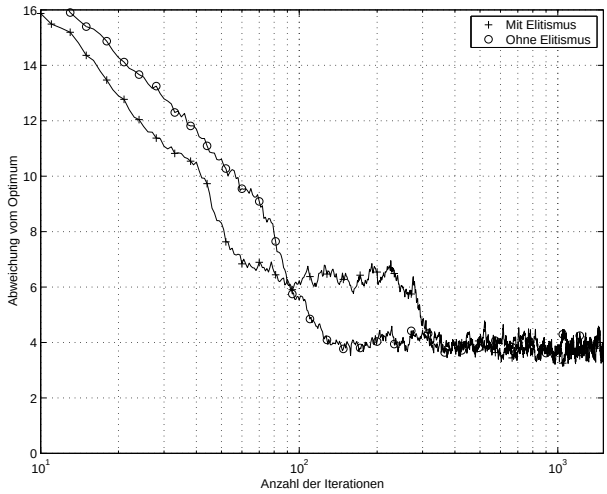

Abbildung 5.11: Vergleich jeweils eines Populationsmittelwertes mit bzw. ohne Elitismus

vom Durchschnittsverhalten feststellen. Man erkennt den wesentlich steileren Verlauf der Simulation mit Elitismus, die allerdings bereits bei etwa 80 Iterationen und 6 Taktzyklen über dem Optimum sättigt. Ohne Elitismus tritt die Sättigung erst bei ungefähr 150 Iterationen auf, liegt aber um mehr als 2 Taktzyklen unter dem Sättigungsniveau der anderen Simulation. Zu beachten ist der Sprung in der durchschnittlichen Fitness, der mit Elitismus nach etwa 300 Simulationsschritten stattfindet. An dieser Stelle wurde durch ständige Mutationen zufällig eine optimale Lösung gefunden, die im weiteren Verlauf die gesamte Population entsprechend verbessert hat. Diese Zufallssuche ist auch verantwortlich dafür, daß mit Elitismus auch nach Sättigung noch eine (schwache) Verbesserung des Ergebnisses stattfindet. Allerdings sind die Sprünge für einen einzelnen Simulationslauf nicht prognostizierbar und finden teilweise auch gar nicht statt, weshalb das Gesamtergebnis dann auch entsprechend schlechter ausfällt.

Lineare und logarithmische Skalierung

Sowohl bei linearer als auch bei logarithmischer Skalierung läßt sich über den Selektionsdruck eine Anpassung an die Problemstellung erreichen. Im Gegensatz zur linearen Skalierung, die einen proportionalen Zusammenhang zwischen Fitness und Überlebenswahrscheinlichkeit herstellt, werden durch die logarithmische Skalierung die fitteren Individuen überdurchschnittlich bevorzugt.

In Abbildung 5.12 werden je eine charakteristische Konvergenzkurve mit linearer bzw. mit logarithmischer Skalierung gegenübergestellt. Man erkennt, daß durch die logarithmische Skalierung zu Beginn des Optimierungsvorganges die besseren Individuen stärker bevorzugt werden und daher eine wesentlich

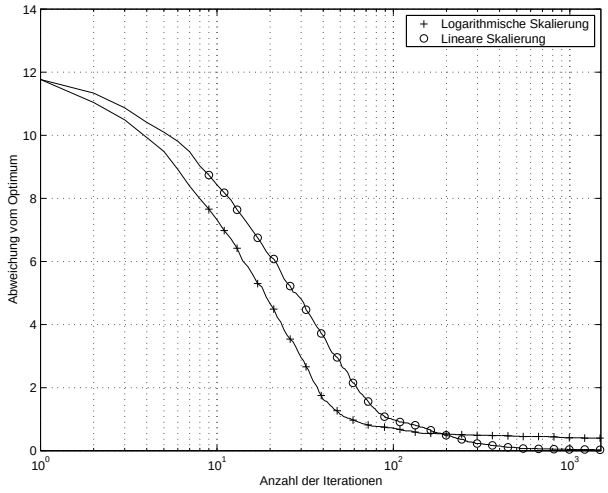

Abbildung 5.12: Vergleich zwischen typischen Konvergenzkurven für lineare und logarithmische Skalierung

raschere Konvergenz erfolgt. Im Gegenzug wird dafür allerdings oft ein lokales Minimum erreicht, daß eben wegen dieser relativ starken Bevorzugung nicht mehr verlassen werden kann. Aus diesem Grund ist auch das Endergebnis mit logarithmischer Skalierung schlechter als im linearen Fall.

Zusammenfassung

Die Abbildungen 5.9 und 5.13 zeigen einen Vergleich zwischen linearer und logarithmischer Skalierung, jeweils mit und ohne Elitismus, für verschiedene Werte des Selektionsdrucks. Man erkennt, daß die Ergebnisse mit Elitismus für relevante Werte des Selektionsdrucks stets besser sind als ohne Elitismus (also unabhängig von der Art der Skalierung und weitgehend unabhängig vom Selektionsdruck). Lediglich bei sehr geringem Selektionsdruck kommt es ohne Elitismus nur noch sehr schwer zu Konvergenz und daher auch zu vergleichsweise schlechteren Ergebnissen. Dieses Verhalten wird besonders bei der logarithmischen Skalierung deutlich sichtbar (siehe Abbildung 5.13), ist aber für die praktische Anwendung nicht relevant, da in diesem Skalierungsbereich grundsätzlich keine guten Ergebnisse mehr erzielbar sind.

Im Vergleich der verschiedenen Verfahren ist des weiteren zu erkennen, daß die logarithmische Skalierung grundsätzlich schlechter abschneidet als die lineare Skalierung. Ein direkter Vergleich ist hier auf Grund der anderen Berechnungsmethode für den Selektionsdruck leider nicht möglich, es ist aber erkennbar, daß das Optimum bei logarithmischer Skalierung deutlich schmäler und daher bei einer neuen Problemstellung schwieriger einzugrenzen ist. Das Optimum bei linearer Skalierung ist deutlich weniger scharf abgegrenzt, eine Änderung des

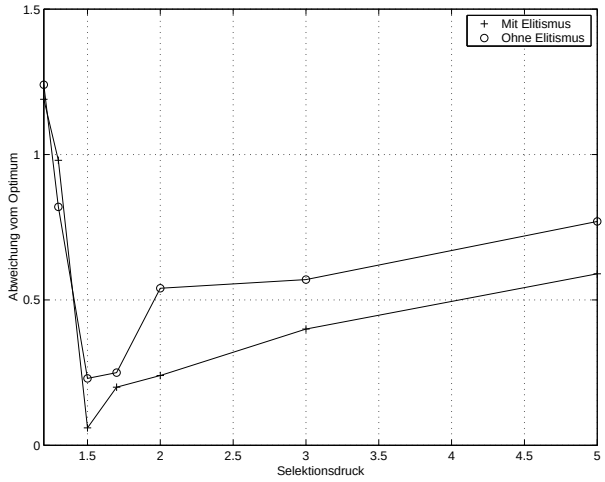

Abbildung 5.13: Vergleich zwischen Ergebnissen mit und ohne Elitismus bei logarithmischer Skalierung

Selektionsdrucks um einen Faktor 2 wirkt sich noch kaum auf die Qualität aus (bei einer Abweichung um einen solchen Faktor liegt das Ergebnis noch immer unter dem Optimum, das mit logarithmischer Skalierung erzielbar wäre).

5.1.5 Populationsgröße

Die Anzahl der Individuen innerhalb einer Population ist ein sehr wesentlicher Parameter für die Effizienz des Algorithmus. Es wird durch eine größere Population bei einer bestimmten Anzahl an Iterationen ein besseres Ergebnis erzielt, so daß grundsätzlich eine möglichst große Population wünschenswert ist. Dem gegenüber steht aber ein (linear mit der Populationsgröße) steigender Rechenaufwand und damit eine entsprechende höhere Ausführungszeit. Es ist daher sinnvoll, eine differenziertere Betrachtung durchzuführen. Die größte Effizienz wird an dem Punkt erreicht, wo mit der geringsten Rechenzeit das optimale Ergebnis gefunden werden kann. Dazu ist es notwendig, die entsprechende Kombination aus Populationsgröße und Iterationszahl zu finden.

In Abbildung 5.14 ist das Ergebnis der Optimierung über der Anzahl der durchgeführten Iterationen für verschiedene Populationsgrößen dargestellt. Man erkennt, daß der Knick, bei dem die Sättigung des Optimierungsvorganges beginnt, weitgehend unabhängig von der Anzahl der Individuen ist und stets bei etwa 80 Iterationen liegt. Entsprechend unterscheiden sich auch die Sättigungsniveaus der einzelnen Kurven, es ist daher nicht möglich, mit kleinen Populationen eine vergleichbare Qualität des Ergebnisses zu erreichen wie mit größeren, selbst wenn man entsprechend mehr Iterationen durchführt. Für ein gewünschtes Qualitätsniveau existiert daher eine Mindestpopulationsgröße, ab der dieses

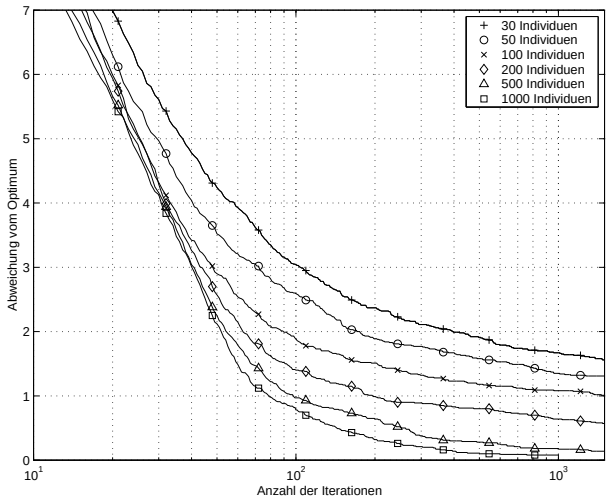

Abbildung 5.14: Konvergenzverhalten in Abhängigkeit der Iterationsanzahl für verschiedene Populationsgrößen

überhaupt erreicht werden kann.

Abbildung 5.15 zeigt die Qualität der erhaltenen Ergebnisse in Abhängigkeit der verwendeten Populationsgröße für verschiedene Iterationszahlen. Es ist erkennbar, daß mit steigender Population auch die Ergebnisse besser werden. Allerdings existiert eine Schranke, über der keine nennenswerte Steigerung mehr beobachtet werden kann. Insbesondere wird oberhalb einer bestimmten Kombination aus Iterationszahl und Populationsgröße immer die optimale Lösung gefunden, eine weitere Erhöhung dieser Parameter vergrößert daher nur noch den notwendigen Rechenaufwand.

Abbildung 5.16 verbindet die Darstellungen aus den Abbildungen 5.14 und 5.15, indem auf der Ordinate das Produkt aus Populationsgröße und Iterationszahl aufgetragen wird, was der Anzahl der insgesamt berechneten Individuen entspricht, also der Rechenzeit direkt proportional ist. Aus dieser Abbildung kann man erkennen, daß jeweils eine der Versuchsreihen für eine gegebene Anzahl an zu berechnenden Individuen das beste Ergebnis liefert, im Unterschied zu Abbildung 5.14 (dort war es immer die Versuchsreihe mit der größten Population) variiert hier die optimale Populationsgröße allerdings.

Es gibt also für jede gewünschte Programmlaufzeit (bzw. ebenso für jede erzielbare Codequalität) eine optimale Populationsgröße. Für hochwertige Lösungen sind große Populationen erforderlich, will man jedoch rasch eine Lösung möglicherweise geringerer Qualität zur Verfügung haben, sind kleinere Populationen sinnvoller.

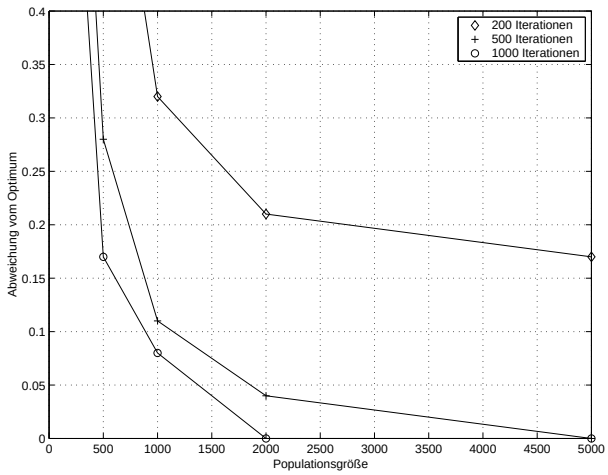

Abbildung 5.15: Abhängigkeit der Codegröße von der Populationsgröße

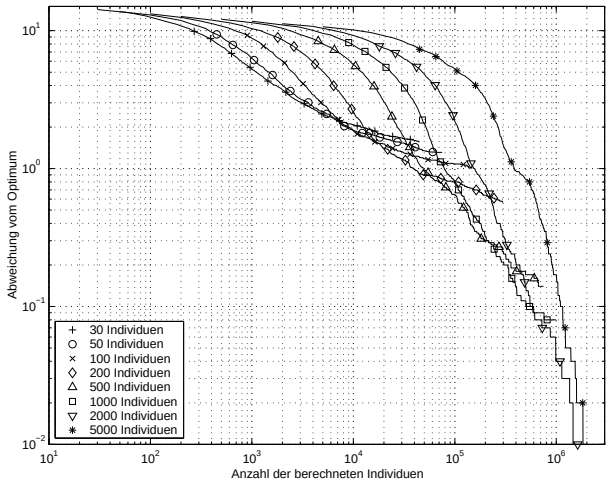

Abbildung 5.16: Konvergenzverhalten in Abhängigkeit der Anzahl der berechneten Individuen für verschiedene Populationsgrößen

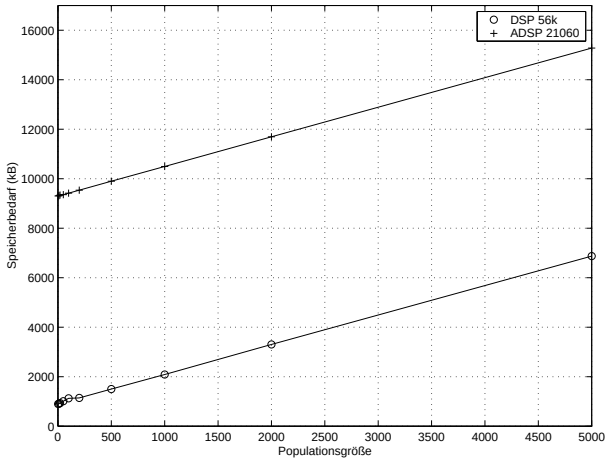

Abbildung 5.17: Speicherverbrauch in Abhängigkeit von der Populationsgröße

5.2 Speicherbedarf

Der benötigte Hauptspeicher wird einerseits für konstante Daten und andererseits für Populationsdaten verwendet. Für beide Bereiche ergeben sich unterschiedliche Abhängigkeiten von mehreren Parametern.

Eine Darstellung des Zusammenhanges zwischen Populationsgröße und Speicherverbrauch wird in Abbildung 5.17 für zwei verschiedene Hardwarearchitekturen gezeigt. Man erkennt, daß sich der Bedarf an Hauptspeicher aus zwei verschiedenen Komponenten zusammensetzt, einem konstanten Anteil, der von der Prozessorarchitektur abhängig ist und einem variablen Anteil, der direkt proportional zur Anzahl der Individuen innerhalb der verwendeten Population ist.

Zusammenfassend kann man feststellen, daß der Speicherbedarf unproblematisch ist. Auch für komplexe (heterogene) Prozessorarchitekturen und große Populationen findet man mit modernen Rechnern problemlos das Auslangen.

5.2.1 Konstante Daten

Die Komplexität der Zielarchitektur wirkt sich vor allem auf die Größe der Trellisdiagramme aus, besonders die Anzahl der verschiedenen Registerbänke ist hier wesentlich, da diese in exponentieller Abhängigkeit zum Speicherbedarf steht. Für traditionelle Prozessorarchitekturen wie z.B. die DSP 56k Serie von Motorola ist der Speicherbedarf praktisch vernachlässigbar (etwa 900 kB), während bei eher heterogenen Architekturen wie z.B. der Shark-Familie von Analog Devices bereits deutlich mehr (annähernd 9 MB) Hauptspeicher notwendig ist.

Die Größe des untersuchten Graphen bewirkt ein Speicherwachstum proportional zur Anzahl der vorhandenen Knoten.

5.2.2 Populationsdaten

Die Anzahl der Bäume im zu bearbeitenden Graphen bewirkt ein etwa lineares Wachstum an Speicherbedarf (exakt zeigt sich eine lineare Abhängigkeit zur Anzahl der Bauminterfacevariablen, die im Mittel aber proportional zur Anzahl der Bäume ist).

Die Anzahl der in der Population enthaltenen Individuen wirkt sich strikt linear auf den Speicherverbrauch aus, wie auch aus der Grafik 5.17 entnommen werden kann.

Die Anzahl der berechneten Iterationen hat keinen Einfluß auf den Speicherbedarf. Ebenfalls ohne Einfluß auf den notwendigen Hauptspeicher sind alle genetischen Parameter.

5.3 Laufzeitverhalten

Die Laufzeit steigt linear mit der Größe des Graphen, mit der Anzahl der berechneten Individuen und der Iterationszahl. Es ist zu beachten, daß für einen größeren Graphen meist auch mehr Iterationen und/oder mehr Individuen für eine Lösung gleicher Qualität notwendig sind, daher steigt der gesamte Rechenaufwand für die Lösung mehr als linear, der Anstieg ist aber stark von der jeweiligen Aufgabenstellung abhängig und kann nicht allgemein angegeben werden.

Die Komplexität der untersuchten Prozessorarchitektur und damit die Größe der verwendeten Trellisdiagramme geht als konstanter Faktor in die Laufzeit ein. Heterogene Architekturen benötigen ein Vielfaches der Laufzeit, um Lösungen ähnlicher Güte zu erhalten, wie das bei homogenen Architekturen bereits mit wesentlich weniger Aufwand erzielbar ist. Das Verhältnis der Laufzeiten bei einem ADSP 2106 und einem DSP der 56k Familie von Motorola beträgt in etwa den Faktor 20.

5.3.1 Abhängigkeit von der Qualität der erzielten Lösung

Die Rechenzeit, die zur Ermittlung einer Lösung benötigt wird, ist von der Qualität eben dieser Lösung abhängig. Dieser zunächst etwas überraschende Zusammenhang läßt sich durch die Natur des verwendeten Trellisalgorithmus erklären. Gute Lösungen zeichnen sich durch kompakteren Code aus, insbesondere also durch weniger benötigte Transferoperationen. Befindet sich an der Spitze eines Trellisbaumes eine Speicheroperation, so kann die oberste (zeitlich gesehen letzte) arithmetische Operation jedes beliebige Register als Zielregister verwenden, wodurch sich die Anzahl der zu berechnenden Teilbäume entsprechend erhöht. Ist jedoch das Zielregister vom genetischen Algorithmus bereits fest vorgegeben, so reduzieren sich die Freiheitsgrade für den Trellisalgorithmus deutlich, für das erste Diagramm kommt nur noch ein einziger Anfangszustand in Frage.

In Abbildung 5.18 wird der Zusammenhang zwischen Qualität der Lösung (Anzahl der Programmzeilen) und der dafür benötigten Rechenzeit (in Sekunden) graphisch dargestellt. Es zeigt sich ein etwa linearer Zusammenhang, die

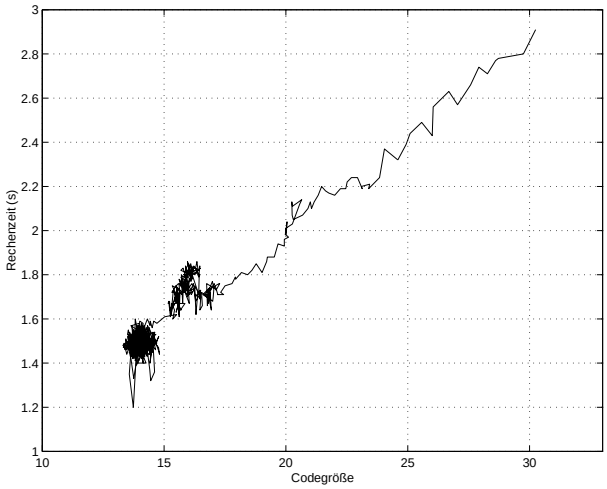

Abbildung 5.18: Rechenzeit in Abhängigkeit von der Qualität der erzielten Lösung

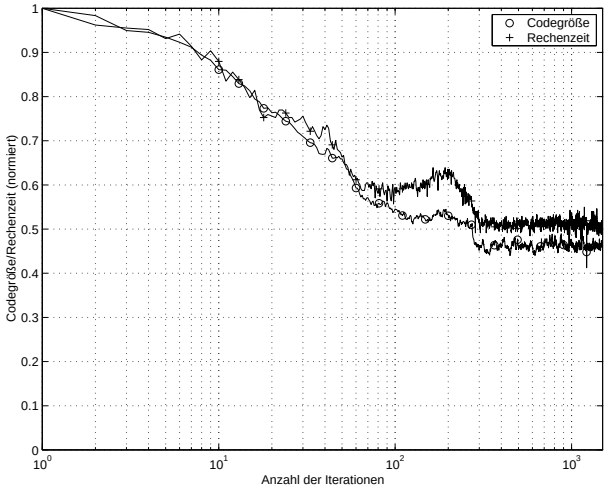

Abbildung 5.19: Vergleich zwischen Rechenzeit je Individuum und Qualität der erzielten Lösung

Unregelmäßigkeiten bei kompakterem Code sind unter anderem darauf zurückzuführen, daß etwa 90% der Datenwerte in diesem Bereich liegen.

Abbildung 5.19 zeigt den Verlauf von Codegröße und Rechenzeit in Abhängigkeit der Anzahl an Iterationen. Zur besseren Vergleichbarkeit der beiden Kurven sind die Anfangswerte jeweils auf 1 normiert. Man sieht, daß sich Laufzeit und Optimierungsgrad etwa 100 Iterationen fast deckungsgleich verhalten. Danach bleibt der Geschwindigkeitszuwachs etwas hinter der fortschreitenden Optimierung zurück. Diese Änderung im Verhalten wird durch die verschiedenen Stadien der Optimierung erzeugt. Zu Beginn werden überwiegend Änderungen an der Registerkonfiguration vorgenommen, Parallelisierung tritt kaum auf, da noch viele gegenseitige Abhängigkeiten bestehen. Sobald diese erste Phase der Optimierung abgeschlossen ist, machen sich zusätzlich Effekte bemerkbar, die durch den Kompaktierungsschritt erzielt werden. An dieser Stelle werden kaum mehr Ein- bzw. Auslagerungen von Datenwerten hinzugefügt oder entfernt, sondern primär Register gegenseitig ausgetauscht, um weitere Parallelisierungen zu ermöglichen. Dadurch kommt es aber auch zu keinen zusätzlichen Laufzeitgewinnen mehr.

Das in Abbildung 5.18 gezeigte Verhalten ist grundsätzlich bei verschiedenen Problemstellungen zu beobachten, in Abhängigkeit von der Aufgabe kann aber die Registeroptimierung oder die Codekompaktierung stärkeren Einfluß auf das Endergebnis haben. Dementsprechend ergeben sich dann auch größere oder kleinere Effekte bei den Änderungen der Programmlaufzeit.

5.3.2 Abhängigkeit von der Crossover- bzw. der Mutationsrate

Bei niedriger Crossover- bzw. Mutationsrate werden zunehmend weniger Individuen den genetischen Operatoren unterworfen. Wenn ein Individuum unverändert aus der vorhergehenden Population übernommen wird, so ist es nicht notwendig, den Fitnesswert erneut zu berechnen, statt dessen kann der bereits bekannte Wert übernommen werden. Bei einer Crossoverrate p_v und einer Mutationsrate p_m ist der zu erwartende Anteil der unveränderten Individuen

$$p = (1 - p_v)(1 - p_m).$$

Die Berechnung jeder Generation wird um den entsprechenden Faktor beschleunigt, es ist aber zu beachten, daß niedrigere Crossover- und Mutationsraten auch zu einer schlechteren Gesamtperformance führen. Entsprechend dem vorigen Punkt kommt es dadurch wieder zu einer Erhöhung der Laufzeit, durch die die hier beschriebenen Effekte teilweise kompensiert werden. Da außerdem die Qualität des Ergebnisses sinkt, ist eine praktische Verwendbarkeit zur Laufzeitreduktion nicht gegeben.

5.3.3 Abhängigkeit von der Größe des Gleichungssystems

Alle Algorithmen, die innerhalb des genetischen Algorithmus zur Anwendung kommen, zeichnen sich grundsätzlich durch lineare Abhängigkeit der Laufzeit

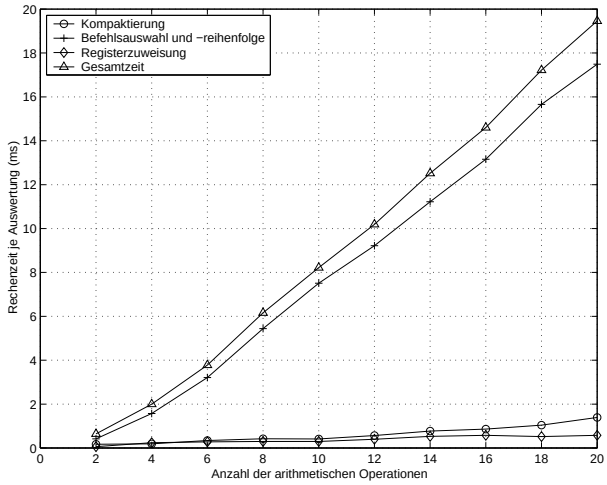

Abbildung 5.20: Rechenzeit in Abhängigkeit von der Größe des Gleichungssystems

von der Größe des zu bearbeitenden Gleichungssystems aus. Dadurch wird eine Anwendung im Zusammenspiel mit dem genetischen Algorithmus überhaupt erst möglich, da eine hohe Anzahl an Individuen berechnet werden muß.

In Abbildung 5.20 wird die Entwicklung der Laufzeit in Abhängigkeit der Größe des untersuchten Graphen für die wesentlichsten Komponenten des Algorithmus dargestellt. Als Graph wurde dabei ein FIR-Filter zunehmender Ordnung verwendet, die Anzahl der arithmetischen Operationen beträgt jeweils das doppelte der Ordnung.

Man erkennt, daß der Hauptanteil der Rechenleistung für die Festlegung der Befehle und deren Reihenfolge benötigt wird, dies entspricht dem Trellisalgorithmus. Ein deutlich geringerer Anteil an Laufzeit wird für die Registerzuweisung und die Kompaktierung verwendet. Alle anderen Programmteile, insbesondere die genetischen Operatoren Selektion, Mutation und Crossover benötigen keine erwähnenswerte Rechenleistung (diese Anteile sind in der Gesamtzeit aus Abbildung 5.20 bereits enthalten).

Kapitel 6

Fallbeispiele

In den folgenden Abschnitten wird der implementierte Algorithmus mit unterschiedlichen Beispielen aus der Praxis getestet. Dabei werden sowohl besonders gut, als auch besonders schlecht geeignete Datenflußgraphen untersucht, um die typischen Einsatzgebiete dieses Verfahrens zu verdeutlichen. Für jeden Testfall werden die Systemgleichungen und der Datenflußgraph spezifiziert. Weiters wird der beste erzielte parallele Code für zyklische Ausführung angegeben (also mit den Randbedingungen, unter denen die Schleifenoptimierung aus Kapitel 4.6 verwendet werden kann), sowie ein Diagramm mit dem Konvergenzverhalten gezeigt. Bei einzelnen Beispielen sind darüber hinaus noch weitere Ergebnisse dargestellt, um bestimmte beachtenswerte Effekte zu zeigen.

Zur Berechnung des Lösungsraumes wird die verwendete Hardwarebeschreibung herangezogen, die auf der Architektur eines Motorola DSPs der 65k Serie basiert. Die Architektur besteht aus 3 voneinander unabhängigen Registerbänken. Jeder Quelloperand kann einen von zwei möglichen Zuständen (*active* oder *release*) annehmen (die Auswahl zwischen *load* und *use* wird durch die Vorgeschichte bestimmt und vergrößert daher nicht den Lösungsraum), dadurch ergeben sich insgesamt 6 Kombinationsmöglichkeiten je Quelloperand. Für Zieloperanden gibt es 3 Varianten (*store/active*, *store/release* und *keep/active*), aus denen sich mit den 3 Registerbänken der untersuchten Architektur insgesamt 9 Kombinationsmöglichkeiten ergeben. Außerdem wird der Lösungsraum noch durch die unterschiedlichen Baumabarbeitungsreihenfolgen aufgespannt. Man erhält daher für einen Gleichungssatz mit k Quelloperanden, l Zieloperanden und m möglichen Permutationen einen Lösungsraum p von

$$p = m \cdot 6^k \cdot 9^l.$$

Die einzelnen Beispiele wurden jeweils mit den gleichen Parametern für den genetischen Algorithmus ausgewertet, um eine gute Vergleichbarkeit zu gewährleisten. Die Parameter sind für eine gute Konvergenz beim Zustandsraumfilter zweiter Ordnung mit einem Lösungsraum von etwa 10^{21} ausgelegt. Es wird ausdrücklich darauf hingewiesen, daß bei den Punkten mit größeren Lösungsräumen

durch Vergrößern der Population bessere Resultate erzielt werden können. Die verwendeten Werte sind im Detail:

- Populationsgröße: 500
- Crossoverwahrscheinlichkeit: 0.9
- Mutationswahrscheinlichkeit: 0.9
- Skalierung: linear
- Selektionsdruck: 50
- Elitismus: nicht aktiviert
- Schleifenoptimierung: aktiviert

Um eine Vergleichsbasis für die erhaltenen Ergebnisse zur Verfügung zu stellen, werden bei jedem Beispiel die Werte angeführt, die mit dem von Motorola zur Verfügung gestellten C-Compiler erzielbar sind. Diese wurden durch Übersetzen der jeweiligen Gleichungssysteme unter Ausnützung aller vorhandenen Optimierungsmöglichkeiten erhalten. Da der Compiler im Gegensatz zu dem hier vorgestellten Algorithmus auch Adreßrechnung berücksichtigt, wurden vor der Auswertung die entsprechenden Instruktionen aus dem Code entfernt, um eine faire Vergleichsbasis zu haben. Ebenso wurden Befehle, die der Normierung von Ergebniswerten auf die in C üblichen Bitbreiten dienen nicht berücksichtigt.

Auch unter Ausnützung aller Optimierungsmöglichkeiten unterstützt der C-Compiler nur eine Speicherbank (wahlweise X- oder Y-Speicher). Daher wird zusätzlich zum Optimalwert auch noch das Ergebnis angeführt, das der vorgestellte Algorithmus liefert, wenn ihm nur eine Speicherbank zur Verfügung steht und die Schleifenoptimierung (die unter C nicht möglich ist) deaktiviert ist. Damit ist ein direkter Vergleich der Performance unter den selben Randbedingungen möglich.

Um einen Vergleich mit handoptimiertem Code zu ermöglichen, wurden alle Aufgaben einem erfahrenen DSP Programmierer vorgelegt, der sie entsprechend den vorgegebenen Randbedingungen (keine Adreßarithmetik) nachprogrammiert hat. Alle händisch ermittelten Lösungen haben die kürzest mögliche Länge (die sich aus der Anzahl der arithmetischen Operationen unter Berücksichtigung von MAC-Befehlen ergibt), die dafür benötigte Zeit reicht von 20 Minuten bis zu 1.5 Stunden. Von den 7 Aufgaben wurden 5 auf Anhieb richtig gelöst, in zwei Fällen war eine Überarbeitung notwendig, da die erste Lösung nicht korrekt gewesen ist (die angegebenen Arbeitszeiten beziehen sich immer nur auf den Aufwand für die korrekte Lösung).

6.1 Transponiertes IIR-Filter 2. Ordnung

6.1.1 Datenflußgraph und Systemgleichungen

$$y = w1 + x * b_0$$

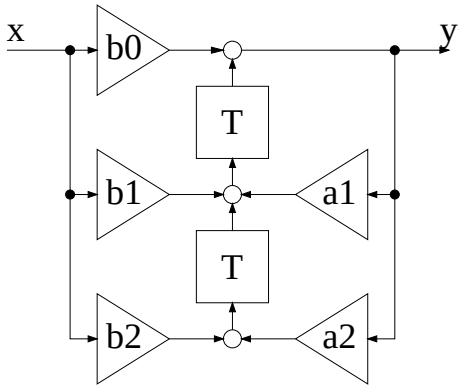

Abbildung 6.1: Transponiertes IIR-Filter zweiter Ordnung, Datenflußgraph

$$w_1 = w2 + x * b_1 + y * a_1$$

$$w_2 = x * b_2 + y * a_2$$

Das transponierte IIR-Filter (in der Literatur of auch als *direct form II* beschrieben), ist eine alternative Realisierungsform des klassischen IIR-Filters (siehe Kapitel 6.3). Dieses Filter kann durch 3 Gleichungen beschrieben werden, die in einer einzigen (nämlich genau der angegebenen) Reihenfolge ausführbar sind. Es gibt 3 Ausgangs- und 12 Eingangsvariable, daraus ergibt sich die Größe des Lösungsraumes zu $P = 1.6 \cdot 10^{12}$.

6.1.2 Ergebnisse

Paralleler Code

Das kürzeste gefundene Programm benötigt 6 Taktzyklen zur Ausführung, sowie einmalig 2 Taktzyklen zur Initialisierung. Eine der möglichen Lösungen ist:

```
Initialisierung:

1: move Y:w1,A
2: move X:w2,B

Hauptprogramm:

1: move X:x,X0    Y:b0,Y0
2: mac X0,Y0,A    Y:b1,Y1
3: mac X0,Y1,B    X:a1,X1  A,Y0
4: mac Y0,X1,B    X:a2,X1  Y0,Y:y
5: mpy X1,Y0,B    Y:b2,Y0  B,A
6: mac X0,Y0,B
```

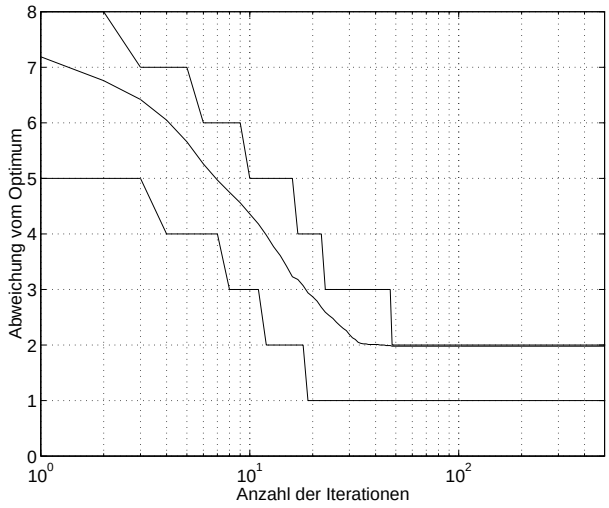

Abbildung 6.2: Transponiertes IIR-Filter, Konvergenzkurve

Bei diesem, wie auch bei allen weiteren Beispielen gilt, daß eine große Zahl äquivalenter Lösungen existieren, die sich teilweise nur durch Permutation der verwendeten Register unterscheiden. Die angegebenen Programmstücke sind willkürlich aus der Menge der erhaltenen Lösungen ausgewählt und gegenüber anderen möglichen Lösungen mit der gleichen Codegröße nicht ausgezeichnet.

Im Rahmen der verwendeten Verfahren handelt es sich bei dem angegebenen Codestück um die optimale Lösung. Eine weitere Verkürzung um einen Taktzyklus wäre möglich, wenn man den ersten Move-Befehl ans Ende der Schleife verlegt, was eine Schleifenoptimierung auf der Ebene des List Scheduling Verfahrens erfordern würde.

In Abbildung 6.2 ist die Konvergenzkurve für die zu Beginn des Kapitels angegebenen Parameter dargestellt. Das gezeigte Optimum bezieht sich dabei auf das besten für den Compiler (durch die Art der verwendeten Algorithmen) erreichbaren Wert. Man erkennt eine vergleichsweise schlechte Konvergenz, die Mehrzahl aller Versuche weicht um eine Instruktion vom Optimum ab. Dies ist vor allem durch die besonders kompakte Lösung verursacht, da im Ergebnis mit einer Ausnahme jede Möglichkeit zur Parallelisierung auch ausgenutzt werden muß. Bei Algorithmen mit einem höheren Anteil arithmetischer Instruktionen ergibt sich (wie auch die nachfolgenden Ergebnisse zeigen) deutlich bessere Konvergenz.

Vergleichswerte

Der vom C-Compiler erzeugte Code benötigt zur Berechnung des transpo-

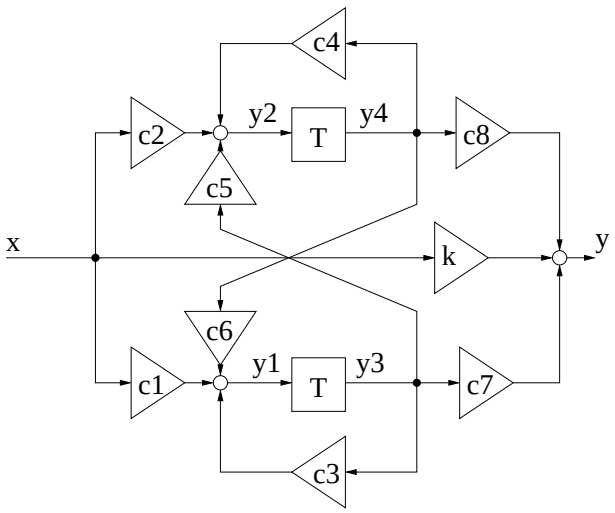

Abbildung 6.3: Zustandsraumfilter zweiter Ordnung, Datenflußgraph

nierten IIR-Filters 18 Taktzyklen, das 3-fache des hybriden Optimierers. Bei Verwendung von lediglich einer Speicherbank und ohne Schleifenoptimierung erreicht der hier vorgestellte Algorithmus eine Lösung mit 11 Taktzyklen, ist also immer noch um einen Faktor 1.6 besser als der Hochsprachencompiler.

Die handcodierte Lösung besteht aus 5 Instruktionen (und ist identisch zur oben angeführten Lösung, wenn man den ersten MOVE Befehl parallel zur letzten MAC-Instruktion ausführt). Es wurden 1.5 Stunden benötigt, um diese Lösung mit der Hand zu finden, während beim hybriden Compiler die Sättigung nach etwa einer Minute erreicht wurde.

6.2 Zustandsraumfilter 2. Ordnung

6.2.1 Datenflußgraph und Systemgleichungen

$$y = y_3 * c_7 + y_4 * c_8 + x * k$$

$$y_2 = y_3 * c_5 + y_4 * c_4 + x * c_2$$

$$y_t = y_3 * c_3 + y_4 * c_6 + x * c_1$$

$$y_3 = y_t$$

$$y_4 = y_2$$

Das Gleichungssystem besteht aus 5 Gleichungen, die in 12 verschiedenen Abarbeitungsreihenfolgen darstellbar sind. Es sind 5 Ausgangs- und 20 Eingangsvariable vorhanden. Der Größe des Lösungsraumes errechnet sich daraus zu $2.6 \cdot 10^{21}$.

6.2.2 Ergebnisse

Paralleler Code

Das kürzeste gefundene Programm für ein Zustandsraumfilter zweiter Ordnung besteht aus zehn Befehlszeilen zuzüglich zwei Befehlen für die Initalisierung. Eine der möglichen Lösungen ist:

```
Initialisierung:

1: move X:y3,A
2: move Y:y4,Y0

Hauptprogramm:

1: move A,X0      Y:c7,Y1
2: mpy X0,Y1,B    X:c8,X0  Y:k,Y1
3: mac Y0,X0,B    X:x,X0
4: mac X0,Y1,B    A,X0     Y:c5,Y1
5: mpy X0,Y1,B    X:c4,X0  B,Y:y
6: mac Y0,X0,B    X:x,X0   Y:c2,Y1
7: mac X0,Y1,B    A,X0     Y:c3,Y1
8: mpy X0,Y1,A    X:x,X0   Y:c6,Y1
9: mac Y0,Y1,A    Y:c1,Y0
10: mac X0,Y0,A   B,Y0
```

Die Aufgabe besteht aus insgesamt 15 arithmetischen Operationen, von denen sich maximal sechs zu Mehrfachbefehlen (mit MAC-Operationen) zusammenfassen lassen. Neun ist daher eine untere Schranke für die Mindestanzahl an Programmschritten. Durch schleifenübergreifendes Parallelisieren kann auch tatsächlich eine entsprechende Lösung gefunden werden, die somit noch einen Taktzyklus kürzer ist.

Die Konvergenzkurve für dieses Filter wird in Abbildung 6.4 gezeigt. Anders als in Abbildung 6.2 ist hier sehr gute (aber vergleichbar rasche) Konvergenz erkennbar. Der Grund dafür liegt im höheren Anteil arithmetischer Operationen beim Zustandsraumfilter.

Zerlegung in elementare Bäume

Zerlegt man das Gleichungssystem in seine elementaren arithmetischen Operationen (dadurch arbeitet der genetische Algorithmus ausschließlich mit Bäumen minimaler Größe, der Trellisalgorithmus wird nicht mehr verwendet), so erhält man für das Zustandsraumfilter zweiter Ordnung 15 Gleichungen mit je einer arithmetischen Operation. Die Aufgabenstellung enthält daher 15 Ausgangs- und 30 Eingangsvariablen in $4 \cdot 10^6$ möglichen Permutationen, wodurch sich der

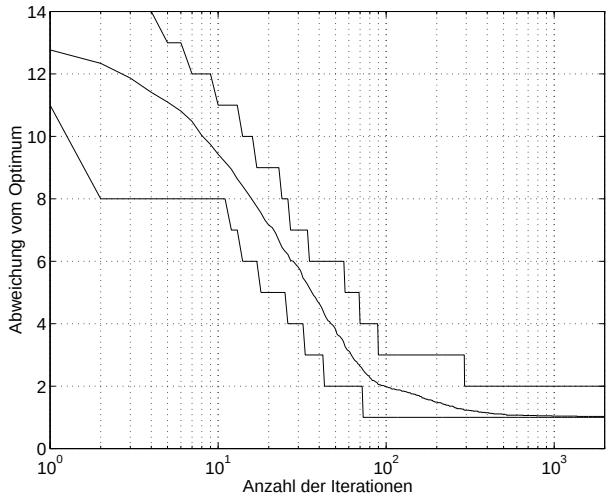

Abbildung 6.4: Zustandsraumfilter, Konvergenzkurve

Suchraum auf $2 \cdot 10^{44}$ vergrößert. Dennoch konvergiert das Programm in vergleichbarer Zeit zur Version des Gleichungssystems mit komplexen Teilbäumen, es wird aber eine deutlich schlechtere Lösung gefunden, weil das Ausnützen von MAC-Befehlen mit den verwendeten Algorithmen nicht baumübergreifend möglich ist. Für die 15 arithmetischen Operationen werden insgesamt 17 Programmschritte benötigt, je einer davon zum Laden der Anfangs- und zum Speichern der Endwerte.

```
Initialisierung:

1: move Y:y3,A

Hauptprogramm:

1: move A,X0      Y:c7,Y0
2: mpy X0,Y0,A    X:y4,X0  Y:c8,Y0
3: mpy X0,Y0,B    X:x,X1   Y:y3,Y1
4: add B,A        Y:k,X0
5: mpy X1,X0,A    X:c2,X0  A,Y0
6: add Y0,A       X:x,X1   Y:c5,Y0
7: mpy X1,X0,B    X:y4,X0  A,Y:y
8: mpy Y1,Y0,A    B,X:t7   Y:c4,Y0
9: mpy X0,Y0,B    Y:y3,X1
10: add B,A       X:c3,X0  Y:c6,Y1
11: mpy X1,X0,B   X:y4,X0
12: mpy X0,Y1,B   X:x,X1   B,Y0
```

```
13: add Y0,B       X:t7,Y1
14: add Y1,A       Y:c1,X0
15: mpy X1,X0,A    A,X:y4
16: add B,A
17: move A,Y:y3
```

Abbildung 6.5 zeigt die Konvergenzkurve für diese Berechnung. Man erkennt, daß die Sättigung weit entfernt vom Optimum erfolgt, die Gründe dafür sind im vorigen Absatz angeführt. Bemerkenswert ist jedoch die trotz des deutlich größeren Suchraumes wesentlich raschere und bessere Konvergenz auf den durch den Algorithmus vorgegebenen bestmöglichen Wert. Die Ursache dafür liegt in der großen Zahl an gleichwertigen Lösungen. Durch das erzwungene Aufteilen der MAC-Befehle auf separate Addition und Multiplikation werden mehr Möglichkeiten geschaffen, parallele Lade- bzw. Speicherbefehle ohne zusätzliche Kosten zu verwenden. Es sind dadurch zahlreiche Lösungen gleich gut, obwohl sie eine sehr unterschiedliche Anzahl an ausgelagerten Variablen verwenden. Die Anzahl der bestmöglichen Lösungen steigt daher in diesem Fall überproportional im Verhältnis zum Suchraum an und erleichtert die Suche.

Zerlegung in elementare Bäume unter Berücksichtigung von MAC-Operationen

Um die Möglichkeiten der MAC-Befehle auszunützen, wurden die entsprechenden Gleichungen aus dem System an Elementargleichungen zusammengefaßt, so daß nun 9 Gleichungen mit 9 Ausgangs- und 24 Eingangsvariablen vorhanden sind, die auf 144 mögliche Reihenfolgen ausgeführt werden können. Der Suchraum hat daher die Größe $2.6 \cdot 10^{29}$. Um diese Reduktion durchzuführen, ist allerdings Wissen um die Beschaffenheit der Zielhardware notwendig, da sich nur in diesem Fall geeignete Gleichungen zusammenfassen lassen. Das kürzeste gefundene Programm hat in diesem Fall 10 Befehlsschritte und ist daher äquivalent zu der Lösung, die bereits mit dem kompakten Gleichungssystem gefunden wurde. Auf eine Wiedergabe wird deswegen verzichtet. Anhand von Abbildung 6.6 kann man jedoch feststellen, daß die Konvergenz wesentlich schlechter ist, als mit dem ursprünglichen Gleichungssystem. Eine Zerlegung in Elementarbäume ist also auch dann nicht ratsam, wenn man die möglichen Optimierungen händisch vornimmt.

Vergleichswerte

Der vom C-Compiler erzeugte Code für das Zustandsraumfilter zweiter Ordnung hat eine Länge von 24 Instruktionen, das ist um einen Faktor 2.4 mehr im Vergleich zur besten gefundenen Lösung. Bei Beschränkung auf eine Speicherbank und Verzicht auf die Schleifenoptimierung errechnet der genetische Algorithmus Assemblercode mit einer Ausführungszeit von 17 Taktzyklen, was immer noch einer Einsparung von 29% entspricht.

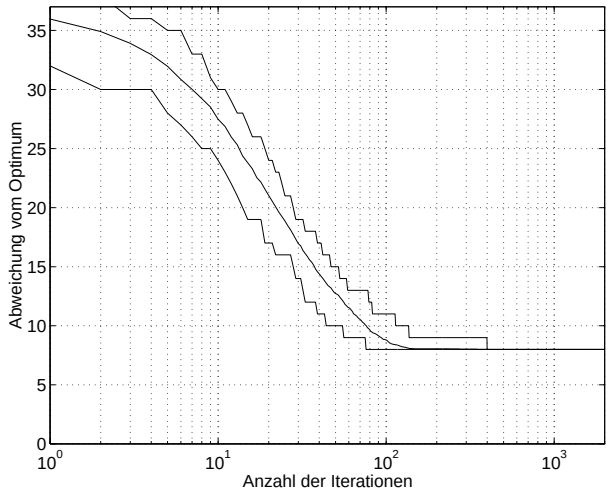

Abbildung 6.5: Zustandsraumfilter bei Zerlegung in Elementarbäume, Konvergenzkurve

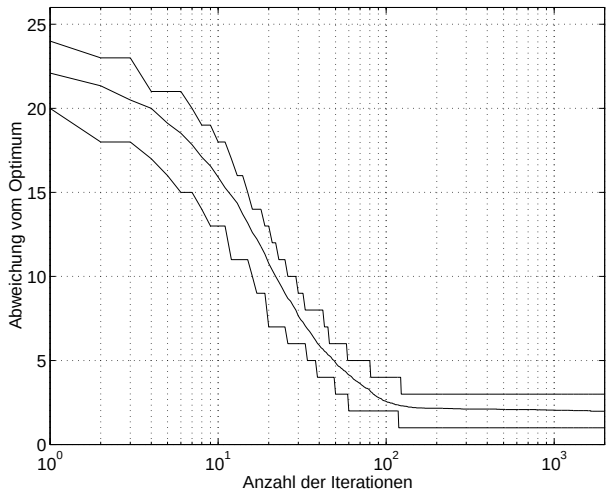

Abbildung 6.6: Zustandsraumfilter bei Zerlegung in Elementarbäume unter Berücksichtigung von MAC-Operationen, Konvergenzkurve

Die handcodierte Lösung verwendet 9 Befehlszeilen, entspricht dem theoretischen Optimum und ist um eine Instruktion kürzer, als die vom genetischen Algorithmus gefundene Lösung (unter optimalen Bedingungen, also mit komplexen Teilbäumen). Es wurden etwa 30 Minuten für diese handoptimierte Lösung benötigt, während der hybride Compiler die Sättigung in weniger als 10 Minuten erreichte.

6.3 IIR-Filter 6. Ordnung

6.3.1 Systemgleichungen

$$y = x + v_1 * a_1 + v_2 * a_2 + v_3 * a_3 + w_1 * b_1 + w_2 * b_2 + w_3 * b_3$$

$$v_3 = v_2$$

$$w_3 = w_2$$

$$v_2 = v_1$$

$$w_2 = w_1$$

$$v_1 = x$$

$$w_1 = y$$

Das hier verwendete Gleichungssystem (in der Literatur oft auch als *direct form I* bezeichnet) stellt die kompakteste Möglichkeit dar, mit einem endlichen Eingangssignal eine unendliche Impulsantwort zu erreichen und ist somit der Prototyp der Klasse der IIR-Filter. Dieses Gleichungssystem besteht aus 7 Gleichungen, die sich in 20 verschiedenen Reihenfolgen ausführen lassen. Es sind 7 Ausgangs- und 19 Eingangsvariable vorhanden, wodurch sich ein Lösungsraum von $5.8 \cdot 10^{22}$ ergibt. Ebenso wie beim FIR-Filter wird auch hier in der Praxis durch Verwendung der Adreßregister und den damit möglichen arithmetischen Operationen eine zusätzliche Verkürzung des Algorithmus erzielt, die mit den Methoden dieses Programmes nicht erreicht werden kann.

6.3.2 Ergebnisse

Paralleler Code

```
Initialisierung:

1: move Y:v1,A
2: move X:v2,X0
3: move Y:w3,Y0
```

```
Hauptprogramm:

1: move X:a1,X1   A,Y1
2: mpy X1,Y1,B    X:v3,X1  Y:x,Y1
3: add Y1,B       X0,X:v3  Y:a2,Y1
4: mac X0,Y1,B    A,X0     Y:a3,Y1
5: mac X1,Y1,B    X:w1,X1  Y:b1,Y1
6: mac X1,Y1,B    X:w2,X1  Y:b2,Y1
7: mac X1,Y1,B    X:w1,A   Y:b3,Y1
8: mac Y0,Y1,B    X:w2,Y0
9: move A,X:w2    Y:x,A
10: move A,X:w1   B,Y:y
```

Die Aufgabe besteht aus insgesamt 12 arithmetischen Operationen, die sich durch Ausnützen von MAC-Instruktionen auf 6 Assemblerbefehle reduzieren lassen. Unter Ausnutzung einer vorhandenen Moduloarithmetik der Adreßregister kann man das Gleichungssystem in 8 Befehlszeilen implementieren, da je eine Instruktion zum Laden des Eingangswertes bzw. zum Speichern des Ergebnisses benötigt wird.

Der hier gezeigte Code ist durch die genannten Einschränkungen länger, es werden jedoch für die insgesamt 6 Umspeichervorgänge nur 2 zusätzliche Befehlszeilen benötigt, da die meisten dieser Operationen durch die schleifenübergreifende Automatik registerintern abgewickelt werden können (für v_1, v_2 und w_3 wird kein eigener Speicherplatz im RAM benötigt, wenn man von den Initialisierungswerten absieht, die im praktischen Anwendungsfall meist 0 sind).

Die Konvergenzkurve, die in Abbildung 6.7 dargestellt ist, zeigt eine im Vergleich zu den anderen Fallbeispielen schlechte Konvergenz. Die Ursache dafür ist die stark unterschiedliche Komplexität der einzelnen Gleichungen. Das System besteht aus vergleichsweise vielen Gleichungen, von denen aber alle bis auf die erste aus einer trivialen Zuweisung bestehen. Dementsprechend wird ein erheblicher Teil der Optimierung auf die Bestimmung der bestmöglichen Reihenfolge von Operationen verwendet, die letztendlich ohnedies durch die Optimierung eingespart werden können.

Vergleichswerte

Mit dem C-Compiler von Motorola wird für dieses Filter ausführbarer Code in der Länge von 29 Instruktionen erzeugt, das ist das 2.9-fache des mit dem genetischen Algorithmus ermittelten besten Ergebnisses. Wird das hier vorgestellte auf eine einzige Speicherbank eingeschränkt und die Schleifenoptimierung deaktiviert, so beträgt die beste Lösung immer noch lediglich 21 Taktzyklen, liegt also um 28 Prozent unter dem vom C-Compiler erreichten Wert.

Der handoptimierte Code benötigt ebenfalls 10 Befehlszeilen, es wurde jedoch eine weitere Lösung gefunden, bei der zwei Zyklen des Programmes in

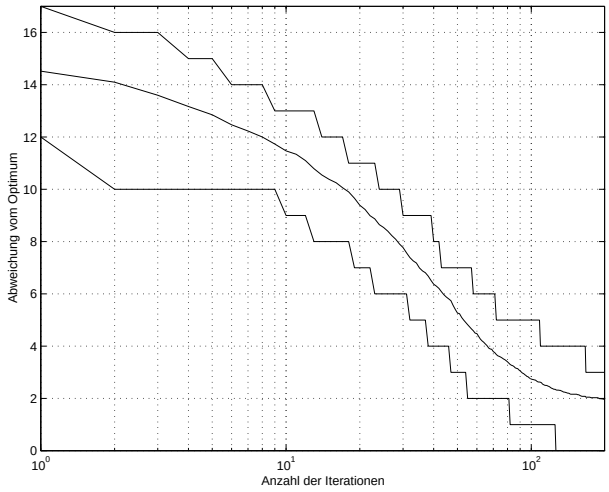

Abbildung 6.7: IIR-Filter 6. Ordnung, Konvergenzkurve

insgesamt 18 Zeilen untergebracht werden konnten. Der Aufwand dafür war in etwa 1 Stunde, im Gegensatz zu etwa 20 Minuten für den hier vorgestellten Compiler.

6.4 FIR-Filter 6. Ordnung

6.4.1 Systemgleichungen

$$t_6 = x_6 * c_6$$

$$x_6 = x_5$$

$$t_5 = x_5 * c_5 + t_6$$

$$x_5 = x_4$$

$$t_4 = x_4 * c_4 + t_5$$

$$x_4 = x_3$$

$$t_3 = x_3 * c_3 + t_4$$

$$x_3 = x_2$$

$$t_2 = x_2 * c_2 + t_3$$

$$x_2 = x_1$$

$$t_1 = x_1 * c_1 + t_2$$

$$x_1 = x$$

$$y = x * c_0 + t_1$$

Das hier beschriebene Gleichungssystem realisiert das einfachste aller FIR-Filter, das sogenannte Kammfilter. Das Gleichungssystem besteht aus 13 Gleichungen, die sich in 77 unterschiedlichen Permutationen ausführen lassen. Es existieren 13 Ausgangs- und 26 Eingangsvariablen. Daraus ergibt sich ein Suchraum von $P = 3 \cdot 10^{34}$. Dieses Beispiel ist für die Bearbeitung durch den entwickelten Algorithmus extrem ungünstig. Es enthält sehr viele Teilbäume geringer Größe. Dadurch, sowie durch die Kette von Verzögerungselementen müssen viele Variable berücksichtigt werden, es entsteht ein extrem großer Suchraum, obwohl das Problem an sich nicht übermäßig komplex ist.

In der Praxis verwendet die Implementierung eines FIR-Filters die Adreßrecheneinheiten des DSPs, um die Kopieroperationen zu vermeiden. Das ist für einen Programmgenerator, der jeder Variablen eine bestimmte Speicherzelle zuweist, grundsätzlich nicht möglich, dementsprechend ist auch der erzeugte Code vergleichsweise ineffizient.

6.4.2 Ergebnisse

Paralleler Code

Das kürzeste gefundene Programm für ein FIR-Filter 6. Ordnung besteht aus zwei Zeilen Initialisierungscode und 10 Zeilen Assemblercode für den Hauptteil des Programms. Wie im vorigen Absatz erwähnt, ist bei händischer Codierung durch Ausnützen der Adreßrecheneinheiten eine Reduktion der Transferbefehle und dadurch eine weitere Verkürzung des Programmes möglich. Für eine feste Zuordnung der Verzögerungselemente zu bestimmten Speicherplätzen stellt die gefundene Lösung jedoch das Optimum dar. Sie zeichnet sich durch einen extrem hohen Grad an Parallelisierung aus, die sechs notwendigen Umspeicheroperationen führen lediglich zu zwei zusätzlichen Befehlszeilen. Die in Abbildung 6.8 dargestellte Konvergenzkurve zeigt, daß durch den großen Lösungsraum das Auffinden der bestmöglichen Lösung schwierig ist. Die durchschnittliche Lösung mit dem angegebenen Parametersatz ist um 1.6 Instruktionen schlechter als die beste gefundene.

```
Initialisierung:

1: X:x5,X0
2: X:x2,Y0

Hauptprogramm:

1: move X:x6,X1    Y:c6,Y1
2: mpy X1,Y1,A     X0,X:x6  Y:c5,Y1
```

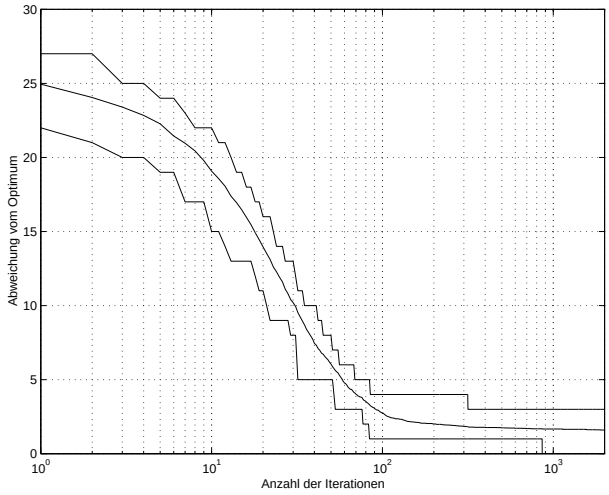

Abbildung 6.8: FIR-Filter 6. Ordnung, Konvergenzkurve

```
3: mac X0,Y1,A     X:x4,X0  Y:c4,Y1
4: mpy X0,Y1,B     X:c3,X1  Y:x3,Y1
5: add A,B         Y0,Y:x3
6: mac Y1,X1,B     X:c2,X1  Y:x,Y1
7: mac Y0,X1,B     X:c1,X1  Y:x1,Y0
8: mac Y0,X1,B     X:c0,X1  Y:x3,A
9: mac Y1,X1,B     A,X:x4   Y:x,Y1
10: move B,X:y     Y1,Y:x1
```

Vergleichswerte

Der vom C-Compiler erzeugte Code benötigt zur Berechnung der angegebenen Filtergleichungen 37 Taktzyklen, das 3.7-fache vom hier entwickelten Algorithmus. Bei Verwendung einer einzigen Speicherbank und ohne Schleifenoptimierung erreicht der hybride Optimierer eine Lösung mit 24 Taktzyklen, ist also immer noch um einen Faktor 1.5 besser.

Der handoptimierte Code für das FIR-Filter benötigt ebenso wie das Ergebnis des hier vorgestellten Compilers 10 Befehlszeilen. Bedingt durch die regelmäßige Struktur der Gleichungen konnte der handoptimierte Code relativ rasch geschrieben werden, beide Verfahren benötigten etwa 15 Minuten zum Finden der optimalen Lösung.

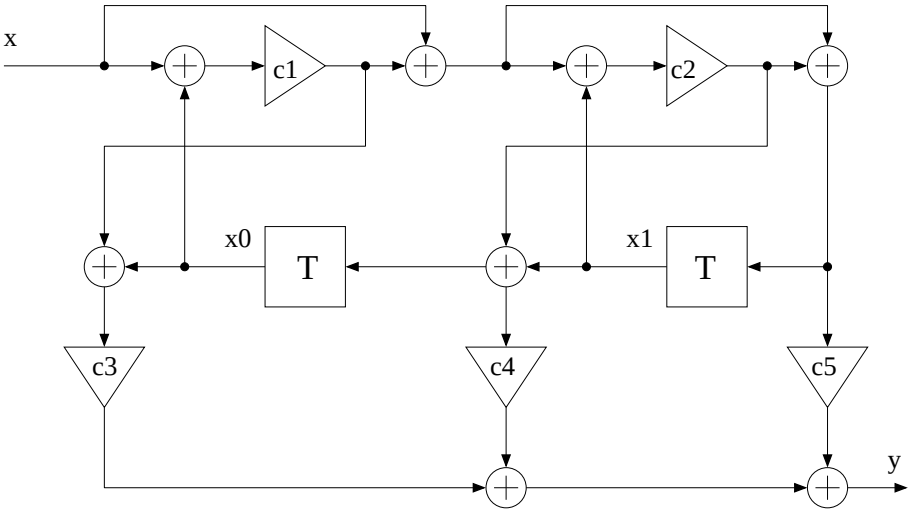

Abbildung 6.9: Lattice-Filter zweiter Ordnung, Datenflußgraph

6.5 Lattice Filter 2. Ordnung

6.5.1 Datenflußgraph und Systemgleichungen

$$z_2 = c_1 * x + c_1 * x_0$$

$$z_3 = x + z_2$$

$$z_7 = c_2 * z_3 + c_2 * x_1$$

$$z_6 = z_2 + x_0$$

$$x_0 = z_7 + x_1$$

$$x_1 = z_3 + z_7$$

$$y = c_3 * z_6 + c_4 * x_0 + c_5 * x_1$$

Filter mit Lattice-Struktur werden unter anderem im Bereich der Sprachanalyse und -synthese eingesetzt, um den menschlichen Vokaltrakt zu simulieren ([24]). Das (rekursive) Lattice-Filter zweiter Ordnung besteht aus 7 Gleichungen mit insgesamt 7 Ausgangs- und 20 Eingangsvariablen. Es existieren 3 gültige Abarbeitungsreihenfolgen. Daraus ergibt sich für den genetischen Algorithmus ein Lösungsraum $P = 3 \cdot 6^{20} \cdot 9^7 = 5 \cdot 10^{22}$.

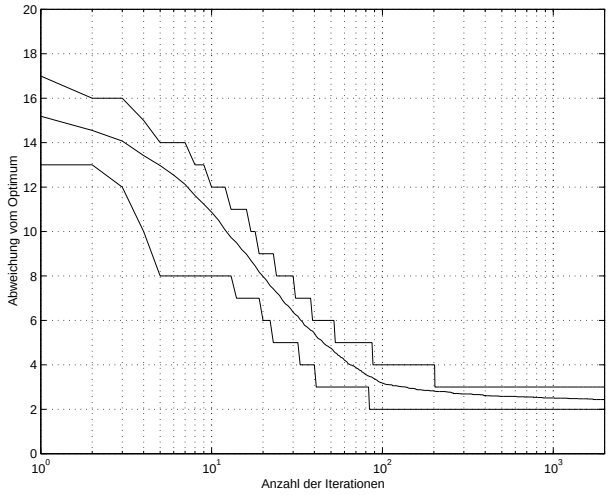

Abbildung 6.10: Lattice-Filter zweiter Ordnung, Konvergenzkurve

6.5.2 Ergebnisse

Paralleler Code

Das kürzeste gefundene Programm für ein Latticefilter zweiter Ordnung besteht aus 13 Befehlszeilen. Es ist kein zusätzlicher Code für die Initialisierung notwendig. Eine der möglichen Lösungen ist:

```
1: move X:x0,X0   Y:c1,Y0
2: mpy X0,Y0,A    X:x,X1    Y:c1,Y0
3: mac Y0,X1,A    X:x,B     Y:x1,Y1
4: add X0,A       X:c2,X1   A,Y0
5: add Y0,B       A,X0      Y:x1,A
6: mpy X1,Y1,B    X:c2,X1   B,Y0
7: mac X1,Y0,B    Y:c3,X1
8: add Y0,B       B,Y1
9: add Y1,A       Y:c4,Y0
10: mpy X1,X0,A   X:c5,X1   A,Y1
11: mac Y0,Y1,A   B,X0      B,Y:x1
12: mac X1,X0,A   Y1,X:x0
13: move A,Y:y
```

Die Aufgabe besteht aus insgesamt 13 arithmetischen Operationen, davon sind 2 durch MAC-Befehle zu Mehrfachoperationen zusammenfaßbar. Für das

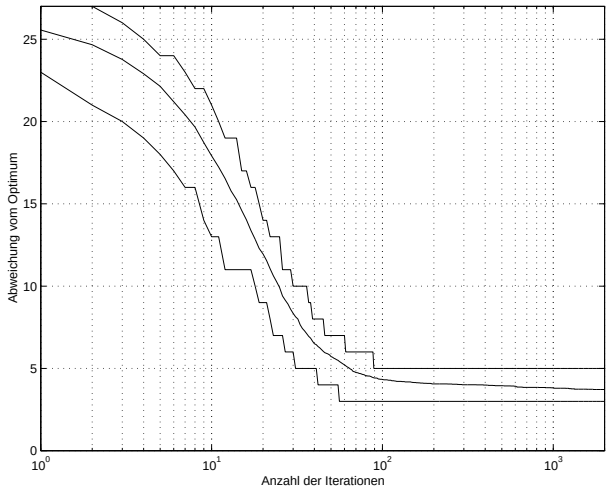

Abbildung 6.11: Lattice-Filter zweiter Ordnung, Konvergenzkurve bei Zerlegung in Elementarbäume

Speichern des Ergebnisses y und dem Laden des neuen Eingangswertes x wird jeweils ein Taktzyklus benötigt. Somit ist dieses Programm die beste mit dem vorgestellten Algorithmus erreichbare Lösung.

Zerlegung in elementare Bäume

Zerlegt man die Gleichungen des Lattice-Filters in einzelne arithmetische Operationen (mit Ausnahme der möglichen Zusammenfassung in MAC-Befehle), so erhält man 11 Gleichungen mit 11 Ausgangs- und 24 Eingangsvariablen. Durch die 9 unterschiedlichen gültigen Abarbeitungsreihenfolgen ergibt sich der Lösungsraum zu $P = 9 \cdot 6^{24} \cdot 9^{11} = 1.3 \cdot 10^{30}$. In Abbildung 6.11 wird die Konvergenzkurve für diese Simulation gezeigt. Man erkennt einen insgesamt ungünstigeren Verlauf, als beim kompakten Gleichungssatz (siehe Abbildung 6.10), es treten bedingt durch den größeren Lösungsraum sowohl eine höhere Streuung, als auch ein schlechterer Konvergenzwert auf.

Vergleichswerte

Der vom C-Compiler erzeugte Code benötigt zur Berechnung des Lattice-Filters 40 Taktzyklen, das 2.8-fache vom hier entwickelten Algorithmus. Bei Verwendung einer einzigen Speicherbank und ohne Schleifenoptimierung erreicht der hybride Optimierer eine Lösung mit 16 Taktzyklen, ist also immer noch um einen Faktor 2.5 besser.

Beim handoptimierten Code wurden die einleitende und abschließende Ladeoperation in die Schleifenoptimierung mit einbezogen. Dadurch konnte die mi-

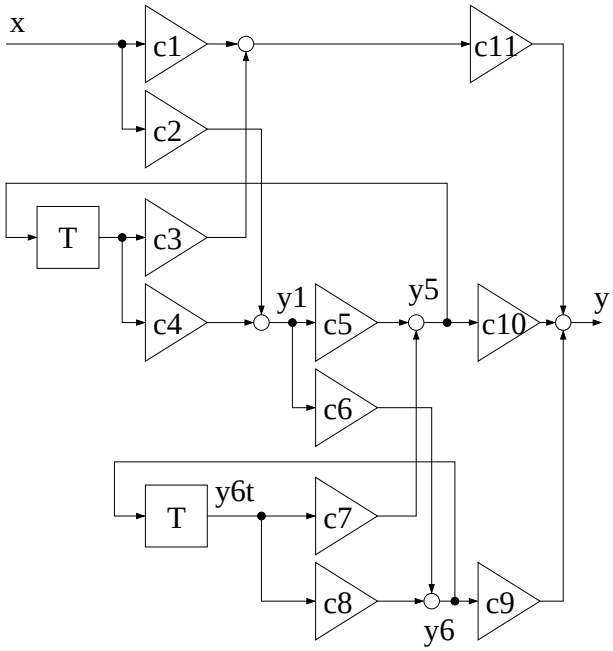

Abbildung 6.12: Ladder-Filter zweiter Ordnung, Datenflußgraph

nimal mögliche Programmlänge von 11 Befehlen erreicht werden. Das Ermitteln der händischen Lösung benötigte 1.5 Stunden, während der hybride Optimierer nach etwa 30 Minuten Konvergenz erreicht.

6.6 Ladder-Filter

6.6.1 Datenflußgraph und Systemgleichungen

$$y_1 = y_4 * c_4 + x * c_2$$

$$y_5 = y_{6t} * c_7 + y_1 * c_5$$

$$y_6 = y_{6t} * c_8 + y_1 * c_6$$

$$y_{6t} = y_6$$

$$y = (y_4 * c_3 + x * c_1) * c_{11} + y_5 * c_{10} + y_6 * c_9$$

$$y_4 = y_5$$

Ladder-Filter sind bereits seit langem in der Analogtechnik bekannt und werden dort vor allem wegen ihrer Robustheit gegenüber Schwankungen der Parameter geschätzt. Durch Umwandeln der Filtergleichungen erhält man eine zeitdiskrete Form dieser Filter ([25]). Ein Ladder-Filter zweiter Ordnung ist durch 6

Systemgleichungen beschreibbar, die in insgesamt 6 verschiedenen Reihenfolgen ausgeführt werden können. Es existieren 6 Ausgangs- und 23 Eingangsvariablen, wodurch man eine Größe des Lösungsraumes von $2.5 \cdot 10^{24}$ erhält.

6.6.2 Ergebnisse

Paralleler Code

```
Initialisierung:

1: move X:y4,X0

Hauptprogramm:

1: move Y:c4,X1
2: mpy X0,X1,A     X:x,X0     Y:c2,Y0
3: mac X0,Y0,A     X:y6,X0    Y:c7,Y1
4: mpy X0,Y1,A     X:c5,X1    A,Y0
5: mac Y0,X1,A     X:x,X1     Y:c8,Y1
6: mpy X0,Y1,B     X:c10,X0   Y:c6,Y1
7: mac Y0,Y1,B     A,X:y5     A,Y1
8: mpy X0,Y1,A     X:c1,X0    B,Y0
9: mpy X1,X0,B     X:y4,X0    Y:c3,Y1
10: mac X0,Y1,B    Y0,X:y6
11: move B,X0      Y:c11,Y1
12: mac X0,Y1,A    X:c9,X0
13: mac Y0,X0,A    X:y5,X0
14: move X0,X:y4   A,Y:y
```

Das Gleichungssystem des Ladder-Filters enthält 17 arithmetische Operationen, die sich durch Ausnützen von MAC-Instruktionen auf 11 Befehlszeilen reduzieren lassen. Die beste gefundene Lösung benötigt zusätzlich einen Transferbefehl zum Speichern des Ergebnisses, sowie einen weiteren nicht vermeidbaren Transferbefehl zum Umkopieren eines Zwischenergebnisses. Es kann nicht ausgeschlossen werden, daß der Ladebefehl in der allerersten Zeile durch eine andere Lösung noch eliminiert werden kann, jedoch ist keine solche Lösung bekannt. Das gefundene Programm weicht damit maximal eine Instruktion von der besten erzielbaren Lösung ab. Der Konvergenzverlauf für dieses Filter wird in Abbildung 6.13 gezeigt.

Vergleichswerte

Der vom C-Compiler erzeugte Code hat eine Länge von 49 Taktzyklen und ist damit um einen Faktor 3.5 länger als die beste gefundene Lösung. Bei Beschränkung auf eine externe Speicherbank und Verzicht auf die Schleifenoptimierung

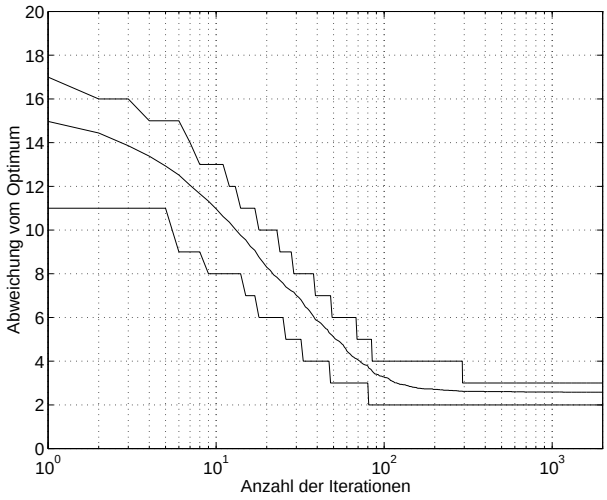

Abbildung 6.13: Ladder Filter, Konvergenzkurve

hat die beste, mit dem hybriden Optimierer gefundene Lösung eine Länge von 21 Instruktionen, was immer noch einer Verbesserung um einen Faktor 2.3 im Vergleich zur Lösung des C-Compilers entspricht.

Die handoptimierte Lösung erreicht durch Schleifenoptimierung des ersten bzw. letzten Ladebefehls eine Länge von 12 Taktzyklen. Der Ladebefehl innerhalb des Programmes konnte ebenfalls vermieden werden, allerdings nur auf Kosten einer zusätzlichen arithmetischen Instruktion, die durch Ausmultiplizieren der Klammer in der letzten Gleichung entsteht. Auf die Gesamtkosten des Programmes wirkt sich diese Umformung daher nicht aus. Während der hybride Optimierer nach etwa 7 Minuten Sättigung erreicht hat, war für die händische Lösung ungefähr eine Stunde notwendig.

6.7 Generalized Impedance Converter

6.7.1 Datenflußgraph und Systemgleichungen

$$y_{4a} = y_4 + x * c_2$$

$$y_{3a} = y_3 + x * c_1$$

$$y_9 = y_{3a} * c_3 + y_{4a} * c_4$$

$$y = y_{3a} + y_{4a} + y_9$$

$$y_4 = y_{3a} + x * c_2 + y_9$$

$$y_3 = y_{4a} + x * c_1 + y_9$$

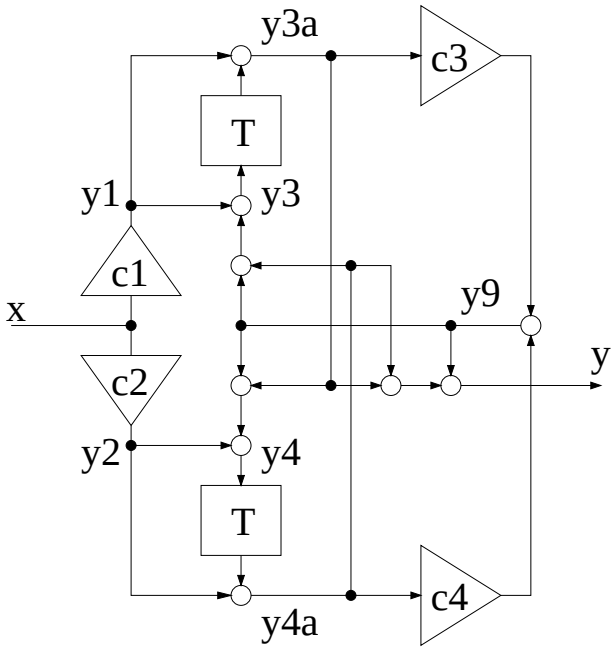

Abbildung 6.14: General Impedance Converter, Datenflußgraph

General Impedance Converter (in der Fachliteratur oft auch als *Gyratoren* bezeichnet) wurden häufig in der Analogtechnik eingesetzt. Durch Transformation in den Digitalbereich erhält man das angeschriebene Gleichungssystem. Digitale Impedanzwandler wurden früher häufiger verwendet (siehe zum Beispiel [10]), teilweise auch als elementarer Bestandteil zur Realisation komplexerer Filter ([6]).

Der allgemeine Impedanzwandler wird durch einen Gleichungssatz von 6 Gleichungen beschrieben, die sich in 12 verschiedenen Permutationen anschreiben lassen. Es sind 8 Ausgangs- und 21 Eingangsvariablen vorhanden, woraus sich die Größe des Suchraumes zu $1.1 \cdot 10^{25}$ errechnet.

6.7.2 Ergebnisse

Paralleler Code

```
Initialisierung:

1: move X:y4,A
2: move Y:y3,B
3: move X:c1,X0

Hauptprogramm:

1: move X:c2,X1   Y:x,Y0
2: mac Y0,X1,A    Y:c4,X1
3: mac Y0,X0,B    A,X:y4a  A,Y0
4: mpy X1,Y0,A    X:y4a,B  B,Y1
5: add Y1,B       X:c3,X1  Y1,Y:y3a
6: mac Y1,X1,A    X:c2,X1  Y:x,Y0
7: add A,B        A,X:y9   Y:y3a,A
8: mac Y0,X1,A    X:y9,B   B,Y:y
9: add B,A        X:y4a,B  Y:x,Y1
10: mac Y1,X0,B   X:y9,Y0
11: add Y0,B
```

Die beste gefundene Lösung benötigt 11 Taktzyklen. Das Gleichungssystem besteht aus 15 arithmetischen Operationen, die durch Verwendung von MAC-Instruktionen auf 10 reduziert werden können. Eine weitere Assemblerinstruktion wird für das Laden des Eingangswertes benötigt, die hier gezeigte Lösung stellt daher das Optimum für den genetischen Algorithmus dar. Eine weitere Befehlszeile läßt sich trivial einsparen, indem man die Ladebefehle der ersten Instruktion parallel zur Addition im letzten Befehl ausführt.

Die Konvergenzkurve für den General Impedance Converter ist in Abbildung 6.15 dargestellt. Man erkennt im Gegensatz zu den bisherigen Beispielen

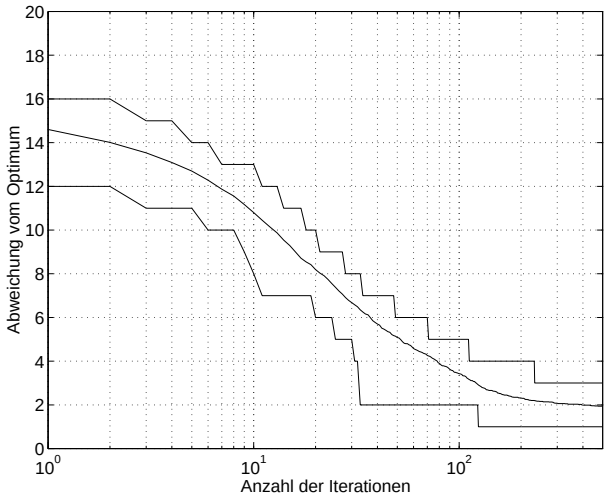

Abbildung 6.15: General Impedance Converter, Konvergenzkurve

eine deutlich raschere Konvergenz. Mit den gewählten Parametern werden deutlich weniger als 1000 Iterationen benötigt, um Sättigung auf einem qualitativ hochwertigen Niveau zu erzielen.

Vergleichswerte

Der C-Compiler erzeugt für das Gleichungssystem des General Impedance Converters eine Lösung mit insgesamt 50 Befehlszeilen. Das ist ein Faktor 3.8 im Vergleich zum genetisch erzeugten Code. Bei Abschalten der Schleifenoptimierung und Verzicht auf die Verwendung einer zweiten externen Speicherbank hat die beste gefundene Lösung die Länge 16, das ist ein Faktor 3.1 im Vergleich zum C-Compiler.

Die handoptimierte Lösung ist de facto ident zur automatisch gefundenen Version, durch Verschieben des Ladebefehls für den Eingangswert allerdings um eine Instruktion kürzer. Der Aufwand zum Finden der Lösung betrug für die händisch ermittelte Lösung etwa 30, für die automatisch erzeugte Version etwa 25 Minuten.

6.7.3 Zusammenfassung

Der hier vorgestellte Algorithmus ist in der Lage, hochoptimierten Code zu entwickeln. Für alle untersuchten Aufgabenstellungen wurde die bestmögliche Lösung gefunden, die (unabhängig von der Problemstellung) um maximal zwei Assemblerinstruktionen von der optimalen Lösung abweicht. Die dazu notwendige Rechenzeit lag selbst auf einem durchschnittlich schnellen Rechner stets

unter der Zeit, die ein erfahrener Assembler-Programmierer für die Lösung der selben Aufgabe benötigte.

Vergleicht man den erzeugten Code mit den handoptimierten Lösungen, so ist bis auf die schleifenübergreifende Parallelisierung kaum ein Unterschied erkennbar. In einigen Fällen wurden sogar idente Lösungen erzeugt.

Im Vergleich zu einem typischen C-Compiler zeigt sich eine massive Überlegenheit des genetischen Hybrids, sowohl bei den Leistungsmerkmalen, als auch bei der Qualität der Ergebnisse. Während der C-Compiler nur eine Speicherbank unterstützen kann und alleine dadurch bereits auf ein erhebliches Optimierungspotential verzichtet, kann der hier entwickelte Algorithmus mit allen verfügbaren (bei üblichen Prozessoren sind das zwei) Speicherbänken arbeiten. Ebenfalls ist bei den C-Compilern kein Ansatz für schleifenoptimierende Registerbelegung zu erkennen, wodurch weitere (geringere) Verluste entstehen. Selbst unter gleichen Voraussetzungen (also bei Annahme nur einer zulässigen Speicherbank und Abschaltung der schleifenübergreifenden Registerallokation) ist der hier dargestellte Algorithmus noch immer deutlich effektiver, als der C-Compiler. Die Ursache liegt darin, daß der Compiler alle Möglichkeiten, die die Sprache C bietet berücksichtigen muß. Dadurch werden oftmals Kopien von Variablenwerten angelegt, die tatsächlich dann gar nicht benötigt werden. Offenbar sind die Optimierungsroutinen nicht ausgereift genug, um diese überflüssigen Instruktionen zu erkennen und zu entfernen.

Die Programmlaufzeit des C-Compilers ist deutlich geringer, als die des genetischen Hybrides. Dem kann allerdings entgegengehalten werden, daß in fast allen Fällen auch die (zufällig gefundenen) Initiallösungen für die verwendete Population bereits besser waren, als das Endergebnis des C-Compilers (das läßt sich unschwer aus den Konvergenzkurven der vorhergehenden Kapitel erkennen). Eine qualitative Verbesserung ist also durchaus auch mit vergleichbarer Rechenleistung möglich.

Abbildung 6.16 zeigt eine zusammenfassende Gegenüberstellung aller untersuchten Gleichungssysteme für den handoptimierten (und gleichzeitig optimalen) Code, den vom genetischen Hybrid erzeugten Code (sowohl mit bestmöglicher Optimierung, als auch unter den bereits erwähnten eingeschränkten Bedingungen) und die Ergebnisse des C-Compilers von Motorola. Man erkennt, daß die optimale Lösung und das Ergebnis des hier vorgestellten Algorithmus stets knapp beisammen liegen, während die Resultate des C-Compilers deutlich schlechter ausfallen.

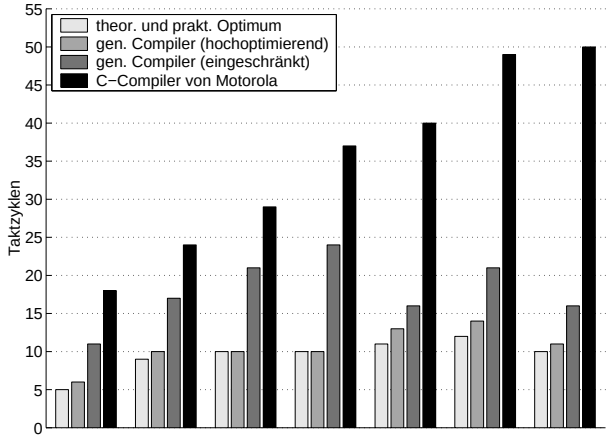

Abbildung 6.16: Gegenüberstellung typischer Ergebnisse für handoptimierten Code, hybriden Optimierer und C-Compiler

Kapitel 7

Retargierbarkeit

Der Codegenerator bezieht seine Informationen über die Architektur des verwendeten Signalprozessors, sowie die Syntax der Assemblersprache aus einer separaten Hardwarebeschreibung, die in einer einfach strukturierten Sprache formuliert ist (siehe Anhang A). Dadurch ist grundsätzlich Unabhängigkeit von einer speziellen Architektur gewährleistet. Die Entwicklung erfolgte jedoch mit besonderer Berücksichtigung der Prozessoren der Firma Motorola, so daß nicht alle am Markt erhältlichen Prozessorfamilien gleich gut für eine Optimierung mit dem in dieser Arbeit verwendeten Ansatz geeignet sind. Nachteilig wirkt sich das vor allem auf die Architekturen der Firma Analog Devices aus. Diese Prozessoren besitzen eine hohe Anzahl an Registern, die bei der Befehlsauswahl absolut gleichwertig sind, wo jedoch für die Parallelisierung bestimmte Einschränkungen eingehalten werden müssen. Da in dieser Arbeit diese beiden Schritte sequentiell durchgeführt werden (und im Rahmen der Befehlsauswahl kein Unterschied zwischen den betroffenen Registern gemacht werden kann), gibt es keine Möglichkeit, zwischen parallelisierbaren und nicht parallelisierbaren Befehlen zu unterscheiden. Der resultierende Code ist daher von geringerer Qualität.

Abhilfe ist möglich, indem die Gleichungssysteme in ihrer elementaren Form angegeben werden. Dann nämlich wird die Entscheidung über die Registerbelegung vom Trellisalgorithmus zum genetischen Algorithmus verlagert (da alle Register automatisch an den Baumgrenzen liegen), wo eine Unterscheidung sehr wohl möglich ist. Dieser Schritt erhöht zwar den Lösungsraum und damit die notwendige Laufzeit, sorgt aber für deutlich bessere Ergebnisse.

Im folgenden werden einige Codebeispiele für den ADSP 21060 von Analog Devices gegeben. Dieser Prozessor verfügt über insgesamt 16 Datenregister, deutlich mehr als die CPUs der Motorola-Familie. Für die folgenden Beispiele wurden alle Register als verwendbar gekennzeichnet, eine Annahme, die in der Praxis oft nicht zulässig ist, da verschiedene Register von übergeordneten Kontrollprogrammen benötigt werden.

7.1 Lattice Filter 2. Ordnung

```
Initialisierung:

1: F8 = (PM: x0)
2: F0 = (DM: x1)

Hauptprogramm:

1: F5 = (DM: c2)   F12 = (PM: x)
2: F4 = F8 + F12   F13 = (PM: c1)
3: F13 = F4 * F13  F1 = (DM: c3)
4: F9 = F12 + F13
5: F0 = F0 + F9
6: F12 = F0 * F5   F4 = F8 + F13   F5 = (PM: c5)
7: F10 = F1 * F4   F0 = F9 + F12   F8 = (DM: x1)
8: F9 = F0 * F5    F8 = F8 + F12   F5 = (PM: c4)
9: F5 = F5 * F8    (DM: x1) = F0
10: F9 = F5 + F9
11: F1 = F10 + F9
12: (DM: y) = F1
```

Der erhaltene Code ist mit 12 Befehlszeilen kürzer als der für den Prozessor von Motorola. Der Grund dafür ist, daß der ADSP 21060 in der Lage ist, Multiplikation und Addition auch dann gleichzeitig auszuführen, wenn es sich *nicht* um eine MAC-Operation handelt. Dadurch sind weitere Einsparungen möglich. Das Ergebnis ist die mit den verwendeten Algorithmen bestmögliche Lösung. Es werden nur zwei Multiplikationen ohne parallele Addition ausgeführt. In beiden Fällen sind alle folgenden Additionen direkt vom Ergebnis der Multiplikation abhängig und können daher keinesfalls vorgezogen werden.

7.2 Zustandsraumfilter 2. Ordnung

```
Initialisierung:

1: F0 = (PM: y3)
2: F1 = (DM: y4)

Hauptprogramm:

1: F2 = (DM: x)     F4 = (PM: c5)
2: F8 = F0 * F4     F4 = (DM: c8)
```

```
3: F9 = F1 * F4      F4 = (DM: c4)
4: F12 = F1 * F4     F4 = (PM: c7)
5: F13 = F0 * F4     F12 = F8 + F12    F4 = (DM: k)
6: F14 = F2 * F4     F9 = F9 + F13     F4 = (PM: c2)
7: F9 = F2 * F4      F3 = F9 + F14     F4 = (DM: c6)
8: F10 = F1 * F4     F1 = F9 + F12     F4 = (DM: c3)
9: F13 = F0 * F4     F4 = (PM: c1)
10: F14 = F2 * F4    F10 = F10 + F13   (DM: y) = F3
11: F0 = F10 + F14
```

Auch hier ist der erhaltene Code kürzer, als für den Motorola-Prozessor. Der Grund liegt ebenfalls in der möglichen Zusammenfassung von Multiplikationen und Additionen, die nicht auf einem gemeinsamen Zwischenergebnis beruhen. Mit den verwendeten Algorithmen ist diese Lösung die kürzest mögliche. Zu den 9 Multiplikationen tritt noch die Addition von Zeile 11, die direkt von der vorhergehenden Multiplikation abhängig ist. Eine weitere Instruktion wird zu Beginn jedes Zyklus zum Laden der Eingangswerte benötigt.

Kapitel 8

Zusammenfassung, offene Probleme und Ausblick

Im Rahmen dieser Arbeit wurde ein Codegenerator für digitale Signalprozessoren entwickelt, der durch ein hybrides Design die Vorteile konventioneller Optimierungsstrategien der automatischen Codegenerierung mit der Flexibilität genetischer Algorithmen kombiniert.

Die wesentlichsten Leistungen der Arbeit sind das Anpassen und Verbinden von existierenden Algorithmen, die bisher aber separat angewendet wurden, zu einem Gesamtkonzept. Darüber hinaus wurden für den genetischen Algorithmus leistungsfähige Operatoren entwickelt. Besonders hervorzuheben ist der Crossover Operator, da alle bisher für die Codegenerierung von DSP-Programmen existierenden evolutionären Ansätze auf Mutationen beschränkt gewesen sind.

Das bereits bisher für heterogene Prozessorarchitekturen verwendete Trellisverfahren wurde adaptiert, um den erzeugten Code an vorgegebene Randbedingungen anpassen zu können. Die Randbedingungen werden durch den genetischen Algorithmus ausgewählt und durch geeignet gewählte Crossover- und Mutationsoperatoren laufend optimiert. Neben dem Trellisverfahren befinden sich noch ein Algorithmus zur Schleifenoptimierung, sowie ein List Scheduling Verfahren zur Parallelisierung des Assemblercodes innerhalb des genetischen Algorithmus. Durch diese Kombination können Phasenprobleme, die sich bei konventionellen Compilerstrukturen negativ auf die Performance auswirken, vermieden werden.

Die Wirkungsweise des hybriden Compilers wird durch mehrere Parameter beeinflußt und kann daher gut an die jeweiligen Problemstellungen angepaßt werden. Dabei handelt es sich um die Populationsgröße, den Selektionsdruck, das zur Anwendung gelangende Skalierungsverfahren für die Fitnessberechnung sowie um die Mutations- bzw. die Crossoverwahrscheinlichkeiten.

Die Beschreibung der Zielhardware erfolgt über eine einfache Beschreibungssprache, die es ermöglicht, gängige DSP-Architekturen ohne Pipelinestrukturen darzustellen. Damit ist eine Einschränkung auf bereits etwas ältere Prozessorfa-

milien verbunden. Allerdings finden gerade diese Prozessoren (vorwiegend wegen der geringen Kosten und ihres geringen Energieverbrauchs) in der Industrie nach wie vor sehr häufig Verwendung. Die in dieser Arbeit vorgestellten Testfälle sind durchaus als repräsentativ zu betrachten. Für eine Erweiterung auf komplexere Architekturen inklusive Pipelinestrukturen wäre zumindestens der Austausch der Subroutinen zur Befehlsauswahl notwendig, da sich derartige Befehlssätze mit Trellisdiagrammen nicht mehr beschreiben lassen. Das Rahmenprogramm zur genetischen Optimierung, die genetischen Operatoren, sowie die Logik zur Ansteuerung der einzelnen Teilalgorithmen könnte möglicherweise beibehalten oder adaptiert werden. Tatsächlich gibt es auch für VLIW-Architekturen bereits Ansätze, die sich mit genetischer Optimierung befassen ([31]).

Insbesondere für die DSP-56k Serie von Motorola wurden umfangreiche Testläufe durchgeführt, die den Einfluß der genetischen Parameter, sowie die Leistungsfähigkeit des entwickelten Programmpakets darstellen. Im Vergleich zu klassischen Hochsprachencompilern werden deutlich bessere Ergebnisse erzielt (bezogen auf die Qualität des erzeugten Codes), zusätzlich können auch die Architekturmerkmale besser ausgenützt werden (insbesondere die Parallelisierung von Instruktionen).

Für eine Weiterentwicklung des hier vorgestellten Programmes bieten sich mehrere Möglichkeiten an. Die Schleifenoptimierung beschränkt sich zur Zeit auf den sequentiellen Code, eine Erweiterung auf den Kompaktierungsalgorithmus wäre denkbar und würde bei mehreren Problemstellungen Gewinne von ein bis zwei weitere Taktzyklen erzielen. Des weiteren wäre eine Einbeziehung der Adreßberechnung und die Erzeugung eines Speicherlayouts für die verwendeten Variablen denkbar, da im Moment die Adreßberechnung separat durchgeführt werden müßte und daher an dieser Stelle weiterhin Phasenkopplungsprobleme auftreten können.

Anhang A

Hardware-beschreibungssprache

Zur Beschreibung der Zielarchitektur wird eine eigens entwickelte Hardwarebeschreibungssprache verwendet, die auf die absolut notwendigen Bestandteile reduziert ist.

A.1 Grammatik

⟨*hardware*⟩ ::= ⟨*registers*⟩ ⟨*memory*⟩ ⟨*parallel*⟩ ⟨*instructionset*⟩

⟨*registers*⟩ ::= '`regbanks`' '`{`' ⟨*registerbank*⟩ '`}`' '`;`'

⟨*registerbank*⟩ ::= ⟨*registerbank*⟩ '`,`' ⟨*registerbank*⟩
| ⟨*bank*⟩ '`(`' ⟨*registerlist*⟩ '`)`'
| '`virtual`' ⟨*bank*⟩

⟨*registerlist*⟩ ::= ⟨*register*⟩
| ⟨*registerlist*⟩ '`,`' ⟨*register*⟩

⟨*register*⟩ ::= ⟨*identifier*⟩

⟨*memory*⟩ ::= '`membanks`' '`{`' ⟨*memorylist*⟩ '`}`' '`;`'

⟨*memorylist*⟩ ::= ⟨*memorybank*⟩
| ⟨*memorylist*⟩ '`,`' ⟨*memorybank*⟩

⟨*memorybank*⟩ ::= ⟨*identifier*⟩

⟨*parallel*⟩ ::= '`parallel`' '`{`' ⟨*parallellist*⟩ '`}`' '`;`'

⟨*parallellist*⟩ ::= ⟨*empty*⟩
  | ⟨*parallellist*⟩ ⟨*multiinstruction*⟩ ';'

⟨*multiinstruction*⟩ ::= ⟨*singleinstruction*⟩
  | ⟨*multiinstruction*⟩ ',' ⟨*singleinstruction*⟩

⟨*instructionset*⟩ ::= ⟨*instruction*⟩ | ⟨*instructionset*⟩ ⟨*instruction*⟩

⟨*instruction*⟩ ::= `instruction` '{' ⟨*instructionline*⟩ '}' ';'

⟨*instructionline*⟩ ::= ⟨*instructionitem*⟩ ';'
  | ⟨*instructionline*⟩ ⟨*instructionitem*⟩ ';'

⟨*instructionitem*⟩ ::= ⟨*instructionitem*⟩ `type` '=' ⟨*type*⟩
  | `target` '=' ⟨*register*⟩
  | `source` '=' ⟨*registerlist*⟩
  | `cost` '=' ⟨*integer*⟩
  | `parallel` '=' ⟨*integer*⟩
  | `name` '=' ⟨*string*⟩

⟨*type*⟩ :== ⟨*identifier*⟩

⟨*integer*⟩ ::= [1-9] | ⟨*integer*⟩ [0-9]

⟨*string*⟩ ::= '"' ⟨*char*⟩ '"'

⟨*identifier*⟩ ::= ⟨*alphachar*⟩ | ⟨*identifier*⟩ ⟨*char*⟩

⟨*alphachar*⟩ ::= [a-zA-z_]

⟨*char*⟩ ::= ⟨*alphachar*⟩ | [0-9$%]

A.2 Schlüsselwörter

A.2.1 Befehlstypen

Für den Typ *type* können die folgenden Schlüsselwörter verwendet werden:

- *add*: Addition, benötigt 2 Operanden, kommutativ
- *subtract*: Subtraktion, benötigt 2 Operanden, nicht kommutativ
- *multiply*: Multiplikation, benötigt 2 Operanden, kommutativ
- *divide*: Division, benötigt 2 Operanden, nicht kommutativ
- *negate*: Negation, benötigt 1 Operanden
- *abs*: Absolutbetrag, benötigt 1 Operanden

- *move*: Transferbefehl für Register-Register Transfers. Hier müssen alle möglichen Kombinationen für Transfers zwischen Registerbänken angeführt werden. Falls Transfers innerhalb der gleichen Registerbank möglich sind, dann müssen auch diese mit dem entsprechenden Kostenwert angegeben werden. Die daraus resultierenden NOP-Instruktionen (Transfers von einem Register zu sich selbst) werden vom Programm automatisch erkannt und mit Kostenwert 0 versehen.
- *leaf*: Blatt im Trellisbaum. Für jede Registerbank, die einen Quelloperanden aufnehmen kann, ist eine eigene *instruction*-Sequenz vom Typ *leaf* anzuführen, die nur aus den Attributen *type* und *target* besteht. Im allgemeinen wird das bei allen Registerbänken mit Ausnahme virtueller Register der Fall sein.
- *load*: Ladebefehl, benötigt 1 Operanden, der einer Speicherbank entsprechen muß.
- *store*: Speicherbefehl, benötigt 1 Operanden. Das Ziel muß einer Speicherbank entsprechen.

A.2.2 Quellregister

Die Quellregister müssen in der gleichen Reihenfolge angegeben werden, wie das die zugehörige arithmetische Operation vorschreibt, insbesondere bei den nicht kommutativen Operationen. Ist ein Quellregister mit dem Zielregister identisch, so muß es unbedingt an erster Stelle angeführt werden. Soll hingegen ein Quellregister zwar aus der gleichen Bank genommen werden, wie das Zielregister, aber mit einem anderen Register realisiert werden, so ist das Quellregister an zweiter Stelle anzuführen. Letzteres ist nur bei binären Befehlen möglich.

A.2.3 Assemblersyntax

Um in die Assemblersyntax die tatsächlich verwendeten Register einbinden zu können, müssen im *name*-Parameter entsprechende Platzhalter verwendet werden. Dabei stehen die Bezeichner '`$t$`' für das Zielregister und '`$s1$`', '`$s2$`' bzw. '`$s3$`' für maximal drei Quellregister zur Verfügung. Die Zuordnung der Quelloperanden zu den Platzhaltern erfolgt in der gleichen Reihenfolge, wie sie im Parameter *source* angegeben worden sind, mit Ausnahme von Mehrfachbefehlen. Bei diesen ersetzt der erste Quelloperand des eingebundenen Befehls das virtuelle Register, alle weiteren Quelloperanden belegen in aufsteigender Reihenfolge die bisher noch nicht vergebenen Platzhalter. Der Syntaxstring des eingebundenen Befehls wird nicht ausgewertet.

A.2.4 Kostenwert

Der Kostenwert eines Befehls wird als Optimierungskriterium verwendet. Üblicherweise handelt es sich dabei um die Ausführungszeit, es können aber auch

andere Werte eingetragen werden, wenn andere Optimierungskriterien gefordert werden, beispielsweise die Codelänge. Bei NOP-Befehlen ist ein Kostenwert von 0 anzugeben, bei Mehrfachbefehlen sind die gesamten Kosten beim übergeordneten Befehl anzuschreiben, die Kosten des eingebundenen Befehls müssen ebenfalls auf 0 gesetzt werden.

Bei parallel ausgeführten Befehlen wird zur Kostenberechnung immer der größte Kostenwert aller beteiligten Befehle herangezogen.

A.2.5 Parallelität

Um die Unterstützung von Mehrfachbefehlen ausnützen zu können, ist es notwendig, die einzelnen Teilbefehle in Gruppen einzuteilen. Durch Zusammenfassen verschiedener Gruppen werden dann die Mehrfachbefehle gebildet. Jeder Befehl darf nur genau einer Gruppe angehören, eine Gruppe darf aber zur Bildung verschiedener Mehrfachbefehle herangezogen werden. Die Reihenfolge der Gruppen in der Sektion '`parallel`' muß exakt derjenigen in der Assemblersyntax entsprechen. Die Numerierung der Gruppen beginnt mit 1, eine Gruppe mit der Kennung 0 wird intern für die Markierung von nicht parallelisierbaren Befehlen verwendet.

Anhang B

Gleichungseingabe

Für die Beschreibung der Aufgabenstellung an den Compiler wurde eine möglichst einfache, aber den typischen Problemstellungen angepaßte Syntax entwickelt. Die Syntax orientiert sich an algebraischer Notation, es werden alle Grundrechnungsarten, Klammer- und Vorrangregeln wie gewohnt unterstützt. Zusätzlich existieren Erweiterungen, um bestimmte Variablen festen Speicherbänken zuweisen zu können, bzw. um ein Abspeichern im Hauptspeicher, oder ein Lesen aus dem Hauptspeicher zu erzwingen.

B.1 Grammatik

⟨*input*⟩ ::= ⟨*line*⟩ | ⟨*input*⟩ ⟨*line*⟩

⟨*line*⟩ ::= ⟨*target*⟩ '`:=`' ⟨*equation*⟩ '`;`'
  | '`define`' ⟨*identifier*⟩ '`as`' ⟨*membank*⟩ '`;`'
  | '`volatile`' ⟨*identifier*⟩ '`;`'
  | '`reserve`' '`(`' ⟨*registerlist*⟩ '`)`' '`;`'

⟨*target*⟩ ::= ⟨*identifier*⟩
  | '`*`' ⟨*identifier*⟩

⟨*membank*⟩ ::= ⟨*identifier*⟩

⟨*registerlist*⟩ ::= ⟨*identifier*⟩
  | ⟨*registerlist*⟩ '`,`' ⟨*identifier*⟩

⟨*equation*⟩ ::= ⟨*identifier*⟩
  | ⟨*equation*⟩ '`+`' ⟨*equation*⟩
  | ⟨*equation*⟩ '`-`' ⟨*equation*⟩
  | ⟨*equation*⟩ '`*`' ⟨*equation*⟩
  | ⟨*equation*⟩ '`/`' ⟨*equation*⟩

| '-' ⟨*equation*⟩
| '(' ⟨*equation*⟩ ')'

⟨*identifier*⟩ ::= ⟨*alphachar*⟩ | ⟨*identifier*⟩ ⟨*char*⟩

⟨*alphachar*⟩ ::= [a-zA-z_]

⟨*char*⟩ ::= ⟨*alphachar*⟩ | [0-9$%]

B.2 Einschränkungen und Hinweise

Bei der Eingabe der Gleichungen ist es notwendig, neben der korrekten Syntax auch einige Richtlinien zu befolgen, um ein zufriedenstellendes Ergebnis zu erhalten.

- Um einer Variablen eine bestimmte Speicherbank fest zuzuweisen, ist das Schlüsselwort '`define`' vorgesehen. Beispielsweise kann mit der Anweisung '`define y as X;`' die Variable y der Speicherbank X fest zugeordnet werden. Die Zuweisung kann an beliebiger Stelle in der Eingabedatei erfolgen.
- Um das Auslagern einer Variablen in den Hauptspeicher zu erzwingen, ist bei der entsprechenden Gleichung ein '`*`' vorzusetzen. Wird das nicht getan, so behält der Compiler die Variable, falls dies machbar und effizient ist, ausschließlich in Registern.
- Um das Lesen einer Variablen aus dem Hauptspeicher zu erzwingen, ist das Schlüsselwort '`volatile`' vorgesehen. Beispielsweise kann mit der Anweisung '`volatile x;`' festgelegt werden, daß die Variable x bei jedem Durchlauf neu aus dem Speicher gelesen werden muß. Andernfalls versucht der Compiler, Variablen nach Möglichkeit auch über Schleifendurchläufe in Registern zu behalten. Das ist im Allgemeinen bei Eingangsvariablen nicht erwünscht.
- Der Algorithmus unterstützt zur Zeit noch keine automatische Baumzerlegung. Da innerhalb der Teilbäume nicht ausgelagert wird, ist es notwendig, daß jede Gleichung für sich mit Hilfe der vorhandenen Register berechnet werden kann. Falls in der Aufgabenstellung komplexere Gleichungen enthalten sind, so müssen diese an geeigneter Stelle zerlegt werden, andernfalls terminiert der Algorithmus mit einem entsprechenden Hinweis und es kann kein Code generiert werden.
- Falls es notwendig ist, bestimmte Register zu sperren, so kann dies mit dem Schlüsselwort '`reserve`' geschehen. Beispielsweise bewirkt die Anweisung '`reserve (X0,Y1)`', daß die Register $X0$ und $Y1$ der Zielarchitektur nicht verwendet werden. Reservierte Register, die auf der Zielplattform nicht existieren, haben keine Auswirkung auf das Ergebnis. Wird die Menge der verfügbaren Register zu stark eingeschränkt, kann es sein, daß keine gültige Lösung mehr gefunden wird.

- Aus programmiertechnischen Gründen sind bestimmte semantische Konstrukte nicht zulässig:
  - Jeder Variablen darf nicht öfter als ein Mal ein Wert zugewiesen werden: diese Einschränkung ist für eine effiziente Arbeitsweise des Compilers erforderlich, behindert aber die Eingabemöglichkeiten kaum. Falls ein Gleichungssatz einen Variablennamen für mehrere Zuweisungen benutzt, so müssen ab der zweiten Zuweisung temporäre Namen verwendet werden.
  - Eine Variable darf innerhalb einer Gleichung nicht öfter als ein Mal vorkommen: auch diese Einschränkung ist aus Effizienzgründen notwendig. Es ist weder zulässig, eine Variable zwei Mal als Eingangsvariable zu verwenden, noch dürfen Gleichungen der Form $x1 := x1 + c$ angeschrieben werden. Auch in diesem Fall kann das Problem aber leicht durch Einführen einer Hilfsvariablen gelöst werden.
- Mehrfachbefehle, wie zum Beispiel MAC-Instruktionen werden zur Zeit nur dann erkannt und optimiert, wenn sie innerhalb eines Baumes auftreten. Bei der Gleichungseingabe sollte daher darauf geachtet werden, daß potentiell optimierbare Strukturen nicht auf zwei verschiedene Bäume aufgeteilt werden.

Anhang C

Programmaufruf

C.1 Kommandozeilenparameter

C.1.1 -h: help

Ein erläuternder Hilfetext, der die einzelnen Kommandozeilenparameter und deren Argumente beschreibt, wird auf der Konsole ausgegeben. Wenn '-h' verwendet wird, werden alle weiteren Kommandozeilenparameter ignoriert.

C.1.2 -d: debug

Das Programm gibt zusätzliche Statusinformationen aus, die bei der Suche nach Programmierfehlern hilfreich sein können. Achtung: diese Option ist nur für die interne Fehlersuche gedacht!

C.1.3 -g #: graph

Der Name der Datei mit dem Datenflußgraphen wird als Argument übergeben.

C.1.4 -t #: target

Die Datei, die die Zielhardware definiert, wird als Argument übergeben.

C.1.5 -i #: iterations

Mit dieser Option kann die Anzahl der Generationen festgelegt werden, die maximal berechnet werden. Bei Verwendung der Option *-b* kann es auch schon vorher zu einem Programmabbruch kommen.

C.1.6 -m #: mutation propability

Diese Option dient zum Einstellen der Mutationswahrscheinlichkeit.

C.1.7 -v #: crossover propability

Diese Option dient zum Einstellen der Crossoverwahrscheinlichkeit.

C.1.8 -p #: selection pressure

Mit diesem Parameter wird der Selektionsdruck festgelegt. Die genaue Bedeutung des Zahlenwerts hängt vom verwendeten Skalierungsverfahren ab.

C.1.9 -e: elitism

Bei Angabe dieser Option wird im Selektionsverfahren Elitismus verwendet, das heißt die jeweils zwei besten Individuen einer Population werden auf alle Fälle in die nächste Generation übernommen. Auf Grund der in Kapitel 5.1.4 gezeigten Ergebnisse wird die Verwendung dieser Option nicht empfohlen.

C.1.10 -s #: scaling

Legt das Skalierungsverfahren fest. Zur Auswahl stehen:

- '1': lineare Skalierung
- '2': exponentielle Skalierung
- '3': keine Skalierung

Auf Grund der in Kapitel 5.1.4 dargestellten Ergebnisse wird die Verwendung von linearer Skalierung ausdrücklich empfohlen.

C.1.11 -l: loop optimization

Diese Option aktiviert die Schleifenoptimierung. Das erzeugte Programm ist optimiert unter der Voraussetzung, daß es wiederholt innerhalb einer Schleife ausgeführt wird. Zusätzlich wird bei Verwendung dieser Option Initialisierungscode erstellt, der notwendig ist, um schleifenübergreifend belegte Register vor dem ersten Durchlauf korrekt zu initialisieren.

C.1.12 -c: compaction mode

Mit diesem Parameter wird der Kompaktierungsmodus aktiviert, der entsprechend der Hardwarebeschreibung versucht, den sequentiellen Zwischencode zu parallelisieren. Die Verwendung dieser Option wird empfohlen, da es außer zu Test- bzw. Forschungszwecken keinen Grund gibt, auf die Parallelisierung zu verzichten.

C.1.13 -n #: number of individuals

Über diesen Schalter kann die Anzahl der Individuen innerhalb der Population festgelegt werden. Der Wert bestimmt maßgeblich die Qualität des erzielten Ergebnisses, Hinweise über sinnvolle Werte sind im Abschnitt 5.1.5 angeführt.

C.1.14 -b #: break at fitness

Wenn ein Abbruchkriterium für die Optimierung bekannt ist, beispielsweise eine untere Schranke für die Codelänge oder eine obere Schranke für die Ausführungszeit einer bestimmten Anwendung, so kann mit diesem Schalter die entsprechende Abbruchbedingung festgelegt werden. Sobald die beste gefundene Lösung den vorgegebenen Wert erreicht oder unterschreitet, wird die Optimierung beendet und die gefundenen Lösung ausgegeben. Sofern eine enge untere Schranke für die Codegröße bekannt ist, wird die Verwendung dieser Option empfohlen.

C.1.15 -r #: initialize random number generator

Das Programm verwendet zur Berechnung von Wahrscheinlichkeiten einen internen Zufallsgenerator, der beim Programmstart initialisiert wird. Die Initialisierung basiert unter anderem auf der Systemzeit und ist daher bei jedem Programmstart anders. Damit ist es möglich, statistische Aussagen über das Programmverhalten zu treffen. Zur Fehlersuche ist es jedoch zweckmäßig, einen bestimmten Programmablauf beliebig oft reproduzieren zu können. Daher kann mit dieser Option der Initialisierungswert für den Zufallszahlgenerator vorgegeben werden.

Abbildungsverzeichnis

Literaturverzeichnis

[1] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[2] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *Journal of the ACM*, 24(1):146–160, January 1977.

[3] G. Araujo and S. Malik. Using register-transfer paths in code generation for heterogeneous memory-register architectures. In *Proc. 33rd ACM/IEEE Design Automation Conf.*, Las Vegas, June 1996.

[4] L. Davis. Applying adaptive algorithms to epistatic domains. In *Proc. 9th Int. Joint Conf. on Artificial Intelligence*, pages 162–164, Los Angeles, August 1985.

[5] M. A. Ertl. Optimal code selection in dags. In *Proc. ACM Int. Symp. on Principles of Programming Languages*, 1999.

[6] C. Eswaran, V. Ganapathy, and A. Antoniou. On the sensitivity of gic-based wave digital filters. *IEEE Trans. on Circuits and Systems*, CAS-29(9):639–642, September 1982.

[7] B. Fox and M. McMahon. *Genetic Operators for Sequencing Problems*, volume 1 of *Foundations of Genetic Algorithms*. Morgan Kaufmann Publishers, Inc., 1991.

[8] S. Fröhlich, M. Gotschlich, U. Krebelder, and B. Wess. Dynamic trellis diagrams for optimized DSP code generation. In *Proc. IEEE Int. Symp. on Circuits and Systems*, volume 3, pages 492–495, Orlando, May/June 1999.

[9] S. Fröhlich and B. Wess. Optimizing complex machine instructions with dynamic trellis diagrams. In *Proc. Int. Conf. on Signal Processing Applications and Technology*, Dallas, October 2000.

[10] V. Ganapathy, C. Eswaran, and A. Antoniou. Improved gic digital filter structures. *IEEE Trans. on Circuits and Systems*, CAS-30(1):49–54, January 1983.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979.

[12] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley.

[13] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, 1989.

[14] M. Gotschlich and B. Wess. Automatic generation of constrained expression trees for global optimized DSP assembly code. In *Proc. 7th Int. Conf. on Signal Processing Applications & Technology*, volume 1, pages 732–736, Boston, October 1996.

[15] W. Kreuzer, M. Gotschlich, A. Helm, and B. Wess. Übersetzung skalarer Datenflußgraphen in Assemblerprogramme für heterogene Registerarchitekturen. In *Proc. DSP Deutschland*, pages 37–48, Munich, October 1996.

[16] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, September 1980.

[17] R. Leupers. Register allocation for common subexpressions in DSP data paths. In *Proc. Asia Pacific Design Automation Conference*, Yokohama, January 2000.

[18] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Code optimization techniques for embedded DSP microprocessors. In *Proc. 32nd ACM/IEEE Design Automation Conf.*, 1995.

[19] I. Oliver, D. Smith, and J. Holland. A study of permutation crossover operators on the travelling salesman problem. In *Genetic Algorithms and their Applications: Proceedings of the Second International Conference*, pages 224–230. L. Erlbaum, 1987.

[20] J. L. Pino, T. M. Parks, and E. A. Lee. Automatic code generation for heterogeneous multiprocessors. In *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, volume 2, pages 445–448, Adelaide, April 1994.

[21] M. Rim and R. Jain. RECALLS II: a new list scheduling algorithm. In *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, volume 2, pages 461–464, Adelaide, April 1994.

[22] W. Spears. *Crossover or Mutation?*, volume 3 of *Foundations of Genetic Algorithms.* Morgan Kaufman Publishers, Inc., 1993.

[23] W. Spears. *Evolutionary algorithms: the role of mutation and recombination.* Natural computing series. Springer-Verlag, 2000.

[24] R. Strum and D. Kirk. *First Principles of Discrete Systems and Digital Signal Processing.* Addison-Wesley, 1988.

[25] F. Taylor. *Digital Filter Design Handbook.* Marcel Dekker, Inc., 1983.

[26] B. Wess. On the optimal code generation for signal flow graph computation. In *Proc. IEEE Int. Symp. on Circuits and Systems*, volume 1, pages 444–447, New Orleans, May 1990.

[27] B. Wess. Automatic instruction code generation based on trellis diagrams. In *Proc. IEEE Int. Symp. on Circuits and Systems*, volume 2, pages 645–648, San Diego, May 1992.

[28] B. Wess. *Übersetzung von Signalflußgraphen in optimierte Programme für integrierte Signalprozessoren.* PhD thesis, TU-Wien, 1993.

[29] B. Wess, S. Fröhlich, and M. Gotschlich. Optimizing data address computation overhead in DSP programs: analysis and evaluation. In *Proc. 9th Int. Conf. on Signal Processing Applications & Technology*, volume 1, pages 575–579, Toronto, September 1998.

[30] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salesman: the genetic edge recombination operator. In *Proc. 3rd Int. Conf. on Genetic Algorithms*, pages 133–140, Fairfax, June 1989.

[31] T. Zeitlhofer and B. Wess. Code optimization for the Carmel DSP-core. to appear in Proc. 10th Int. Conf. on Signal Processing Applications & Technology, November 1999.

[32] V. Zivojnovic, S. Ritz, and H. Meyr. Retiming of DSP programs for optimum vectorization. In *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, volume 2, pages 465–468, Adelaide, April 1994.

Lebenslauf

Name:	Stefan Fröhlich
Geburtsdatum:	25. Februar 1971
Geburtsort:	Wien
Vater:	Dr. Anton Fröhlich, EDV-Organisator
Mutter:	Elisabeth Fröhlich, Human Resources Managerin

Ausbildung

1977 - 1981	Volksschule, Wien XIX
1981 - 1985	Gymnasium, Wien XIX
1985 - 1990	HTL, TGM Wien XX, Fachrichtung Nachrichtentechnik
1990 - 1996	Studium der Elektrotechnik/Nachrichtentechnik an der Technischen Universität Wien
1996	Diplomprüfung
seit 1996	Doktoratsstudium der technischen Wissenschaften, Elektrotechnik

Berufslaufbahn

1995 - 1996	Tutor am Institut für Nachrichtentechnik und Hochfrequenztechnik, TU Wien
1996 - 1997	wissenschaftlicher Mitarbeiter am Institut für Nachrichtentechnik und Hochfrequenztechnik, TU Wien
1997 - 2001	Universitätsassistent am Institut für Nachrichtentechnik und Hochfrequenztechnik, TU Wien

Wissenschaftliche Veröffentlichungen

W. Kreuzer, S. Fröhlich, M. Gotschlich, A. Helm und B. Wess, Übersetzung von Datenflußgraphen in optimierte Assemblerprogramme für Signalprozessoren in *e&i*, Jänner 1998, Volume 115, Seiten 41-47

B. Wess, S. Fröhlich and M. Gotschlich, Optimizing data address computation overhead in DSP programs: analysis and evaluation in *Proc. 9th Int. Conf. on Signal Processing Applications & Technology*, Toronto, September 1998

B. Wess and S. Fröhlich, DSP data memory layouts optimized for interme-

diate address pointer updates in *Proc. IEEE Asia Pacific Conf. on Circuits and Systems*, Chiangmai, November 1998

S. Fröhlich, M. Gotschlich, U. Krebelder and B. Wess, Dynamic Trellis Diagrams for Optimized DSP Code Generation in *Proc. IEEE Int. Symp. on Circuits and Systems*, Orlando, Juni 1999

S. Fröhlich and B. Wess, Optimizing Complex Machine Instructions with Dynamic Trellis Diagrams in *Proc. Int. Conf. on Signal Processing Applications and Technology*, Dallas, Oktober 2000

S. Fröhlich and B. Wess, Evolutionary Code Generation for Architectures with Multiple Data-Memory Banks in 4th International Workshop on Software and Compilers for Embedded Systems, St. Goar, März 2001