Diplomarbeit

Vergleich von CLP und ILP basierten Optimierungsstrategien am Beispiel der Codegenerierung für DSPs.

Torsten Menne

Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

15. September 1999

Betreuer:

Dipl.–Inform. Steven Bashford
Prof. Dr. Peter Marwedel

Hiermit erkläre ich an Eides statt, daß ich für die Anfertigung dieser Arbeit keine anderen als die angegebenen Quellen verwendet habe. Ich versichere, sie noch keinem anderen Prüfungsamt vorgelegt zu haben.

Dortmund, 15. September 1999

Torsten Menne

Danksagung

Herrn Steven Bashford möchte ich für die Vergabe des interessanten und anspruchsvollen Themas danken. Danken möchte ich vor allem auch für seine hervorragende und intensive Betreuung. Bei Herrn Prof. Dr. Peter Marwedel möchte ich mich für seine Bereitschaft bedanken, diese Arbeit zu begutachten. Weiterhin bedanken möchte ich mich bei Daniel Kästner, der mir bei Fragen zu CPLEX und CIS viele wertvolle Hinweise gegeben hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Während der letzten Jahre haben sich digitale Signalprozessoren (*DSP*) als wichtigste Klasse von Prozessoren zur Implementierung eingebetteter Anwendungen in Konsumgütern etabliert. Sie übernehmen Aufgaben von der einfachen Steuerung von Waschmaschinen oder Mikrowellen, bis zur aufwendigen *real-time*-Codierung in digitalen portablen Funktelefonen (Handys) [Cas95]. Durch ihren Einsatz auf dem elektronischen Massenmarkt müssen sie einen Anforderungskonflikt zwischen niedrigen Preisen und hohen Leistungsanforderungen lösen. Die Kosten von DSPs werden niedrig gehalten durch die Minimierung der Chipfläche (geringere Ausschußraten), den weitestmöglichen Verzicht auf externen Speicher (Beschränkung auf im Chip integrierten Speicher (*on-chip memory*)) und einen geringen Stromverbrauch (Betriebskosten). Den hohen Leistungsanforderungen versuchen DSPs durch zusätzliche Hardware-Features nachzukommen. Es werden zum Beispiel Instruktionen zur parallelen Ausführung von Operationen und Speicherzugriffen oder die Hardwareunterstützung von Schleifen angeboten.

Ein Compiler für DSPs muß die besondere Architektur und damit den spezifischen Befehlssatz dieser Prozessoren berücksichtigen. Durch eine hohe Codequalität können bei DSPs sowohl die Hardwarekosten als auch die Betriebskosten reduziert werden. Durch sehr kompakte Programme kann zum Beispiel eine Reduzierung der Hardwarekosten durch den Verzicht auf externen Speicher erreicht werden. Eine Reduzierung der Betriebskosten ist zum Beispiel durch den Einsatz effizienter Programme möglich. So kann die Betriebszeit eines Handy verlängert werden, indem die Berechnungszeit für eine Codierung kleiner wird. Allerdings können existierende Hochsprachen-Compiler das Leistungspotential von DSPs nicht vollständig ausschöpfen [VZS95].

In der Vergangenheit sind deshalb wichtige, häufig aufgerufene Codestücke, wie innere Schleifen eines Programms, direkt in Assembler programmiert worden. Mit wachsender Funktionalität der DSPs und zunehmender Komplexität der Algorithmen gestaltet sich die Programmierung in Assembler jedoch zusehends aufwendiger, fehleranfälliger und teurer. Deshalb werden Möglichkeiten erforscht, mit denen sich traditionelle Compiler erweitern lassen, um die Programmierung für DSPs mit verbreiteten Hochsprachen, wie zum Beispiel C, zu ermöglichen.

Eine Phase eines Compilerdurchlaufs ist die Codegenerierung. Hier werden Teilaufgaben wie zum Beispiel die *Instruktionsanordnung* (*IS*) und die Registerallokation bearbeitet. Eine Klasse neuerer Verfahren löst derartige Aufgaben durch Modelle der *linearen Programmierung*. Durch die Lösung eines solchen *Linearen Programms* (*LP*) kann zum Beispiel eine, hinsichtlich der Anzahl der benötigten Instruktionen, optimale Anordnung berechnet werden.

Ein solches LP-Modell ist *Scheduling and allokation with Integer Linear Programming* (*SILP*) [Zha96]. Dieses Modell wurde in der Diplomarbeit von Daniel Kästner [Käs97] an einen realen Prozessor, den ADSP-2106x, angepaßt. Das angepaßte SILP-Modell kann sowohl Aufgaben der IS, als auch Aufgaben der IS unter Berücksichtigung von *RessourcenschrAnken* (*ISRA*) für ein Assembler-Programm als ein ganzzahliges lineares Gleichungssystem ausdrücken. Ein solches Gleichungssystem wird für ein konkretes Assemblerprogramm als *SILP-Programm* bezeichnet. Als Lösungsverfahren für SILP-Programme ist in [Käs97] *Integer Linear Programming* (*ILP*) verwendet worden. Es hat sich jedoch gezeigt, daß aufgrund der hohen Laufzeiten mit ILP optimale Lösungen nur für sehr kurze Codestücke berechnet werden können. Aus diesem Grund sind verschiedene approximative Verfahren mit deutlich besseren Laufzeiten bei hoher Codequalität entwickelt worden.

In dieser Arbeit wird ein anderes Lösungsverfahren verwendet: Die Constraint-logische Programmierung (*Constraint Logic Programming, CLP*). Mit diesem Verfahren sind in den letzten Jahren erstmals einige $\mathcal{NP}$-vollständige Probleme gelöst worden [vH89], für die, mit den bisherigen Verfahren, keine Lösung möglich war. Aus diesem Grund besteht ein Interesse an der Frage, ob dieses Lösungsverfahren für Aufgaben der Codegenerierung geeignet ist. Als Beschreibung der Aufgaben der Codegenerierung kann das SILP-Modell mit wenigen Ergänzungen für CLP verwendet werden. Zur Unterscheidung werden die Begriffe *SCLP-Modell*, für das an CLP angepaßte SILP-Modell, und *SCLP-Programm*, für die damit generierten CLP-Programme, eingeführt. Auf der Basis eines gemeinsamen Modells werden verschiedene optimale und suboptimale Lösungsstrategien entwickelt und mit ILP verglichen.

Zur Übersicht ist in Abbildung 1.1 der Zusammenhang der Begriffe, wie sie in dieser Arbeit verwendet werden, dargestellt. Es werden zwei Modelle zur Darstellung von IS und ISRA betrachtet, das SILP- und das SCLP-Modell. Letzteres ist eine an CLP angepaßte Version des SILP-Modells. Mit diesen Modellen werden für konkrete Assemblerprogramme SILP- bzw. SCLP-Programme generiert. Diese Programme werden mit den entsprechenden Lösungsverfahren ILP und CLP bearbeitet. Mit dem Begriff *Lösungsstrategie* werden Techniken zur Steuerung der Lösungssuche in beiden Verfahren bezeichnet. Die wichtigsten Lösungsstrategien in ILP sind der *Simplex-Algorithmus* und *Branch-and-Bound* Methoden. Diese Lösungsstrategien können in CLP ebenfalls verwendet werden; darüber hinaus kommen *Constraint* gesteuerte Lösungsstrategien zum Einsatz, die eine sehr flexible Anpassung an ein konkretes Problem ermöglichen, und die *Constraint Propagation* zur Reduzierung des *Suchraums*. Die genannten Modelle, Verfahren und Lösungsstrategien werden in den Kapiteln 3 und 4 im Detail erläutert.

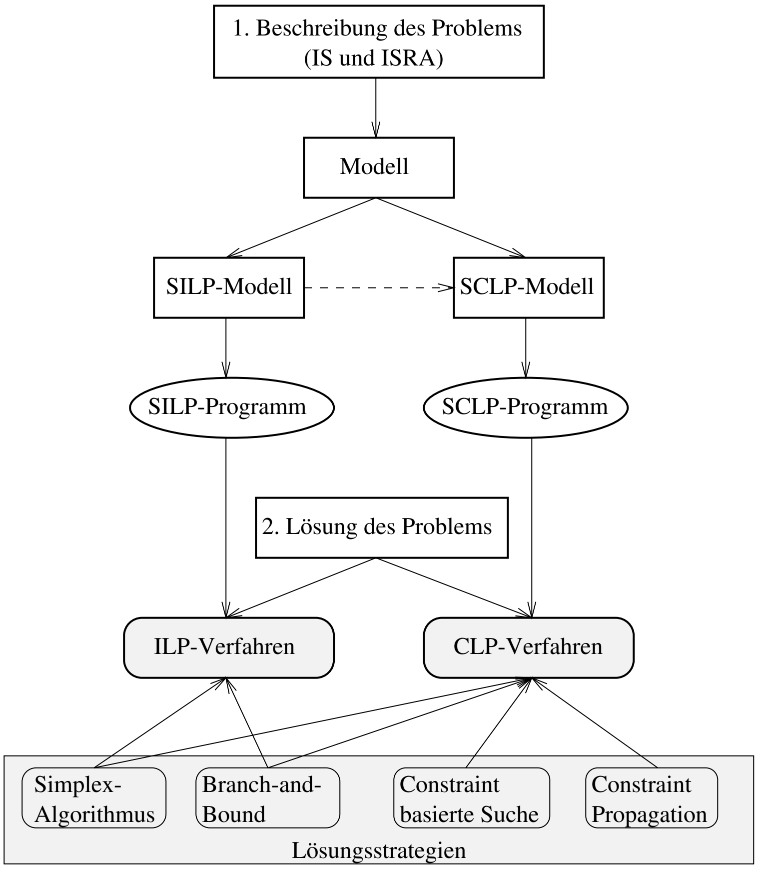


Abbildung 1.1: Zusammenhang der wichtigsten Begriffe

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist ein Vergleich von Optimierungsstrategien für ILP und CLP am Beispiel der Codegenerierung für digitale Signalprozessoren[I]. Für beide Verfahren werden Lösungsstrategien untersucht, mit denen die Berechnungszeiten und die Codequalität für eine Reihe von Benchmarks optimiert werden. Eine Lösungsstrategie wird in diesem Zusammenhang als effizient bezeichnet, wenn ihre Laufzeit möglichst gering ist. Unter dem Ausdruck *Programme mit hoher Codequalität* werden optimierte Programme mit möglichst wenigen Instruktionen verstanden. Durch die Verwendung eines gemeinsamen Modells (SILP) ist ein direkter Vergleich der Berechnungzeiten der beiden Verfahren möglich.

Dieses Ziel erfordert die Umsetzung einiger Teilziele:

1. Erzeugung von Referenzdaten für das ILP-Verfahren mit den in [Käs97] entwickelten Lösungsstrategien.
2. Abbildung des SILP-Modells auf das SCLP-Modell.

[I] genauer, am Beispiel des ADSP-2106x SHARC

3. Entwicklung von Lösungsstrategien für das CLP-Verfahren, mit denen die Berechnungszeiten und die Codequalität optimiert werden.
4. Vergleich und Bewertung der Ergebnisse.

1.3 Das SILP-SCLP-System

Das SILP-SCLP-System dient zur Umsetzung der gesetzten Ziele. Die hierzu verwendeten Programmkomponenten werden in Abschnitt 1.3.1 vorgestellt. Anhand einer Reihe von Benchmarks, die in Abschnitt 1.3.2 aufgelistet sind, werden die verschiedenen Lösungsstrategien getestet. Der dazu notwendige Datenfluß ist in Abschnitt 1.3.3 beschrieben und in Abbildung 1.2 dargestellt.

1.3.1 Programmkomponenten

Das SILP-SCLP-System besteht aus den folgenden Programmkomponenten:

- **CPLEX** ist eine kommerzielle Programmbibliothek zur Lösung von gemischt ganzzahligen linearen Programmen und wird zur Lösung von SILP-Programmen verwendet.
- **ECLiPSe** basiert auf der logischen Programmiersprache *Prolog* und bildet eine Plattform zur Programmierung in CLP. Mit ECLiPSe werden die Berechnungen der SCLP-Programme durchgeführt.
- **CIS** (*Comparativ Instruction Scheduler*) ist ein von Daniel Kästner [Käs97] und Marc Langenbach [Lan97] entwickeltes Tool und generiert unter anderem aus Assemblerprogrammen SILP-Programme.
- **SILP2SCLP** ist ein selbstentwickeltes Tool zur Konvertierung der SILP-Programme in SCLP-Programme.

Weitere Informationen zu diesen Komponenten befinden sich in Anhang B.

1.3.2 Benchmarks

Als Benchmarks werden Standardbeispiele aus dem Bereich der Codegenerierung für DSPs verwendet. Die erste Gruppe an Benchmarks (Gruppe A) umfaßt die von [Käs97] in Saarbrücken verwendeten Beispielprogramme. Damit ist ein direkter Vergleich mit den dort erarbeiteten Ergebnissen möglich. Das größte Programm der Gruppe A besteht aus 49 Instruktionen. Zur Erweiterung des Benchmarking werden weitere Beispielprogramme (Gruppe B) mit bis zu 93 Instruktionen betrachtet.

Einige Basisinformationen zu diesen Benchmarks finden sich in den Tabellen A.1 bis A.4 in Anhang A. Zudem ist dort zu jedem Benchmark eine kurze Beschreibung und eine Gegenüberstellung des Assemblerprogramms der seriellen Ausgangsbeschreibung und der optimierten parallelen Fassung aufgeführt.

Gruppe A	
FIR:	*finite impuls response* Filter
IIR:	*infinite impuls response* Filter
DFT:	Komplexe diskrete Fouriertransformation
Whetstone:	Teil des Whetstone Benchmarks
histo:	Bildhistogramm
conv:	Konvolutionsberechnung

Gruppe B	
Biquad_o:	*biquad_one_section* Filter
Complex_mul:	Multiplikation im Komplexen
lattice:	*lattice* Filter
n_comple:	*n_complex updates* Filter

Tabelle 1.1: Benchmarks der Gruppen A und B

1.3.3 Datenfluß

Das SILP-SCLP-System kann in vier Stufen aufgeteilt werden, die nacheinander durchlaufen werden. Dabei werden in den einzelnen Stufen Teilaufgaben bearbeitet, deren Grundlagen und Lösungen in den weiteren Kapiteln beschrieben sind. An dieser Stelle soll ein Überblick der einzelnen Arbeitsschritte gegeben werden, um damit den Arbeitsablauf der gesamten Arbeit in Zusammenhang zu stellen.

I. **Compiler**: In dieser Stufe findet die Übersetzung von C-Programmen durch einen Compiler in Assemblerprogramme für den ADSP-2106x statt.

II. **ILP**: In der zweiten Stufe werden aus den Assemblerprogrammen durch das Tool CIS SILP-Programme erzeugt. Die generierten Programme werden durch Einsatz von CPLEX gelöst. Dazu werden verschiedene Lösungsstrategien von CIS und CPLEX untersucht. Die SILP-Ergebnisse enthalten für alle Benchmarks die dafür notwendigen Berechnungszeiten und die jeweilige Anzahl an Instruktionen. Diese Ergebnisse dienen als Referenzdaten für das ILP-Verfahren.

III. **CLP**: In der dritten Stufe werden aus den SILP-Programmen durch SILP2SCLP SCLP-Programme erzeugt. Diese Programme werden in ECLiPSe eingelesen und dort mit selbstentwickelten Lösungsstrategien von CLP bearbeitet. Analog zu SILP enthalten die SCLP-Ergebnisse Anordnungs- und Laufzeitinformationen.

IV. **Vergleich und Bewertung** der Lösungsstrategien anhand der SILP- und SCLP-Ergebnisse.

In Abbildung 1.2 sind zur Übersicht die verwendeten Programmkomponenten und der Datenfluß der Assemblerdateien zwischen ihnen dargestellt.

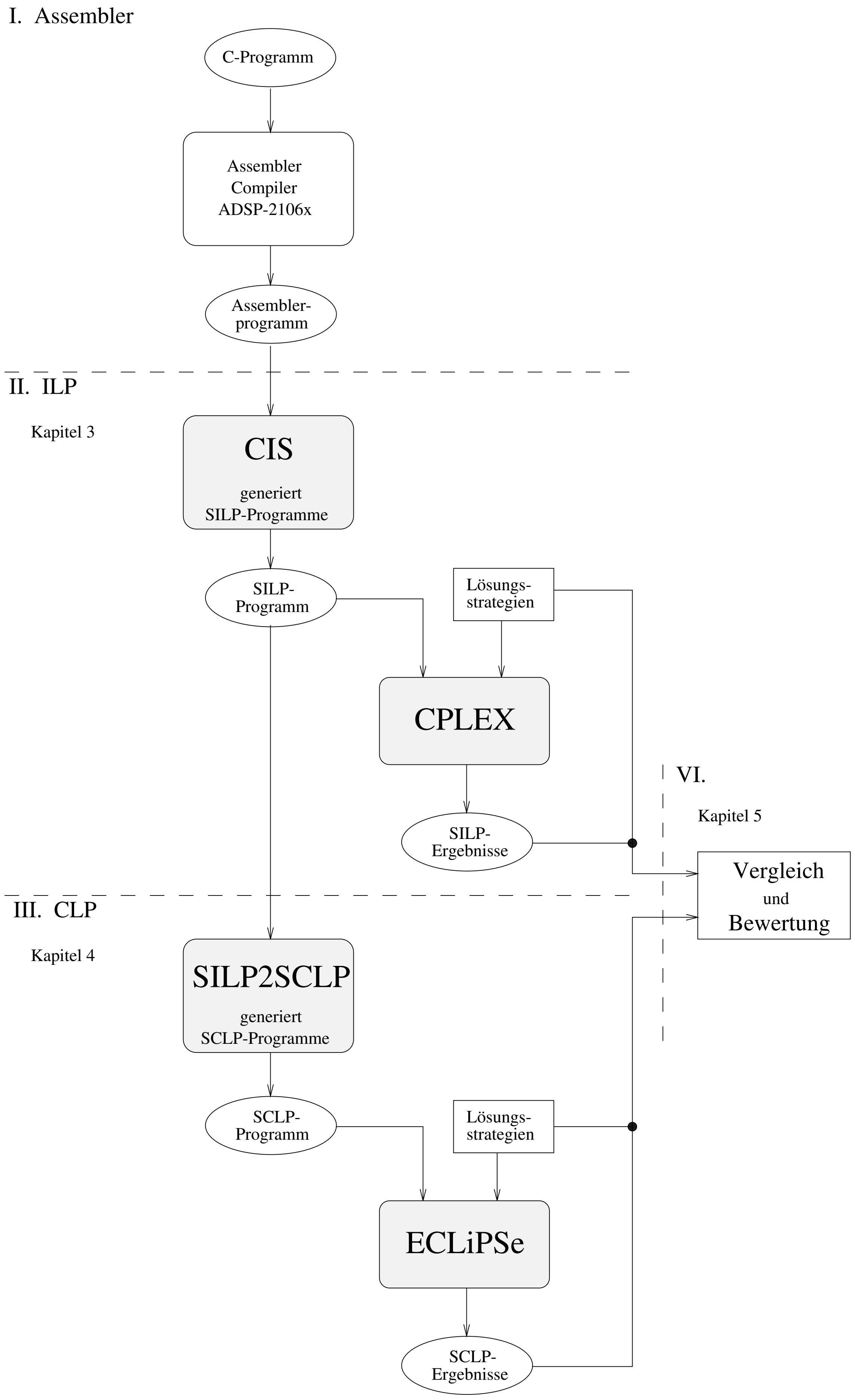


Abbildung 1.2: Das SILP-SCLP-System

1.4 Gliederung

Die Gliederung der Diplomarbeit lehnt sich an den in Abbildung 1.2 dargestellten Arbeitsablauf an. In Kapitel 2 wird zuerst ein Überblick über die Architektur des ADSP-2106x gegeben. Danach werden einige formale Grundlagen zu den Datenmodellen und der Codegenerierung geschaffen. Im Anschluß folgt eine Betrachtung der Aufgaben der Instruktionsanordnung und der Registerallokation. Kapitel 3 gibt eine Einführung in die lineare Programmierung und stellt Lösungsstrategien für LPs vor. Des weiteren werden zwei Modelle zur Beschreibung des ILP-Verfahrens vorgestellt und die Referenzdaten mit dem SILP-Modell aufgeführt. Kapitel 4 behandelt zunächst die Grundlagen Constraint-logischer Programmierung und eine Erläuterung der Abbildung von SILP auf SCLP. Im weiteren wird die Entwicklung und konkrete Implementierung der Lösungsstrategien vorgestellt. Dabei werden die Ergebnisse und damit die Vergleichsdaten für das CLP-Verfahren aufgeführt. Ein Vergleich der Verfahren und eine Analyse der Ergebnisse schließt sich in Kapitel 5 an. Nach einer Zusammenfassung und einem Ausblick in Kapitel 6 folgt ein Anhang mit genaueren Beschreibungen der Benchmarks und der verwendeten Programmkomponenten und Abkürzungen.

1.5 Notation

Zur besseren Darstellung und Strukturierung der Arbeit wird folgende Notation festgelegt:

- **Fachbegriffe** werden bei ihrer ersten Nennung *kursiv* gesetzt.
- **Abkürzungen** für Fachbegriffe werden bei der ersten Nennung *kursiv* in Klammern dahinter gesetzt. Eine Liste der Abkürzungen findet sich in Anhang C.
- **Definitionen** werden durch den Schlüsselbegriff „Definition“ begonnen und kapitelweise durchnumeriert. Der Definitionstext wird *kursiv* gesetzt.
- **Beispiele** werden durch den Schlüsselbegriff „Beispiel“ begonnen, kapitelweise durchnumeriert und durch das Sonderzeichen $\Box$ abgeschlossen.
- **Fußnoten** werden kapitelweise durchnumeriert und sind zur besseren Unterscheidung von Indizes und Exponenten in römischen Zahlen $(I, II, \ldots)$ angegeben.

Kapitel 2

Grundlagen der Codegenerierung

In diesem Kapitel wird eine Einführung in die Grundlagen der Codegenerierung in Hinblick auf die später zu lösenden Aufgaben IS und ISRA gegeben. Dazu wird in Abschnitt 2.1 zunächst die Architektur des ADSP-2106x betrachtet. In Abschnitt 2.2 folgt eine kurze Beschreibung der Phasen eines Compilers. Die Aufgaben einer dieser Phasen, der Codegenerierung, werden in Abschnitt 2.3 beschrieben. Abschnitt 2.4 enthält formale Grundlagen zur Darstellung von Programmen mittels Graphen. Mit diesen Beschreibungsmöglichkeiten werden in Abschnitt 2.5 IS und in Abschnitt 2.6 die Registerallokation detaillierter dargestellt und verschiedene Lösungsverfahren diskutiert. In Abschnitt 2.7 werden Vorteile und Probleme einer Phasenkopplung von Aufgaben, wie sie bei ISRA auftreten, betrachtet.

2.1 Architektur des ADSP-2106x

Der ADSP-2106x *SHARC* (*Super Harward ARchitecture Computer*) ist ein 32 Bit breiter digitaler Signalprozessor für multimediale Anwendungen, wie Sprach-, Ton- und Bildverarbeitung. Der Kernprozessor (*core processor*) besteht aus einem Datenregisterfile, drei Berechnungseinheiten im Rechenwerk, einer Kontrolleinheit, zwei Adreßgeneratoren, einem Zeitgeber (*timer*) und einem Instruktionscache. Die Berechnungseinheiten werden in Abschnitt 2.1.1 näher beschrieben; eine ausführliche Beschreibung der anderen Komponenten findet sich in [Lan97].

Auf dem Chip befinden sich drei Busse: der *PM*-Bus (*Program Memory*), der *DM*-Bus (*Data Memory*) und der I/O-Bus. Über den PM-Bus wird sowohl auf Instruktionen als auch auf Daten zugegriffen. Der Prozessor kann in einem Zyklus auf zwei Datenoperanden zugreifen, einen über den DM-Bus und einen über den PM-Bus. Zulässige Adressierungsmodi sind direktes und indirektes Adressieren; dabei sind sowohl *pre*- als auch *post increment* von Adreßregistern möglich. Zudem wird die Adressierung innerhalb zirkulärer Datenpuffer auf Hardwareebene unterstützt. Zirkuläre Puffer werden häufig bei der digitalen Signalverarbeitung eingesetzt, zum Beispiel in digitalen Filtern und Fouriertransformationen.

2.1.1 Rechenwerk

Es gibt drei unabhängige Berechnungseinheiten:

- **ALU**[I]
- **Multiplizierer** mit Festkommma-Akkumulator
- **Shifter**

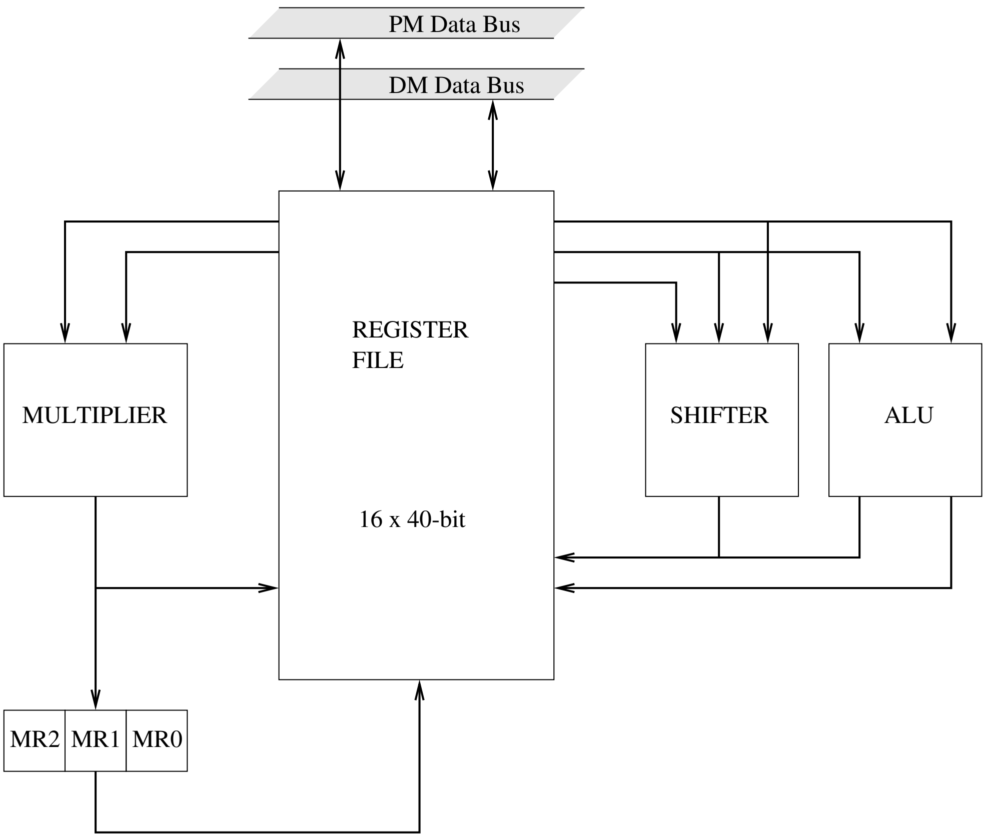


Abbildung 2.1: Die Berechnungseinheiten des ADSP-2106x

Die ALU führt arithmetische und logische Operationen sowohl für Fest-, als auch für Gleitkommazahlen aus. Der Multiplizierer berechnet Gleitkomma- und Festkommamultiplikationen ebenso wie kombinierte Multiplikationen und Additionen/Subtraktionen für Festkommazahlen. Der Shifter führt logische und arithmetische Shifts, sowie spezielle Bitmanipulationen auf 32-Bit-Operanden aus. Alle Instruktionen werden innerhalb eines Zyklus ausgeführt. ALU, Multiplizierer und Shifter sind über das Registerfile miteinander verbunden, so daß die Ausgabe einer beliebigen Einheit als Eingabe jeder beliebigen Berechnungseinheit im nächsten Zyklus verwendet werden kann. Am Ende jeder Operation werden verschiedene Bits in Kontroll- und Statusregister sowie weitere Flags gesetzt, die als Grundlage für nachfolgende bedingte Anweisungen dienen können.

[I] *Arithmetic Logic Unit*

2.1.2 Parallele Ausführung

Multifunktionsinstruktionen kombinieren die parallele Ausführung von Multiplizierer und ALU oder parallele Funktionen innerhalb der ALU. Bei einer parallelen Ausführung sind die Eingabeoperanden für Multiplizierer und ALU, wie in Abbildung 2.2 gezeigt, auf eine Menge von vier bestimmten Registern eingeschränkt; die Ausgabeoperanden können in jedes Register zurückgeschrieben werden. Das Registerfile ist dazu in vier Blöcke zu je vier Register aufgeteilt. Die Einschränkung auf bestimmte Register wird durch zusätzliche Bedingungen bei der Codegenerierung ausgedrückt.

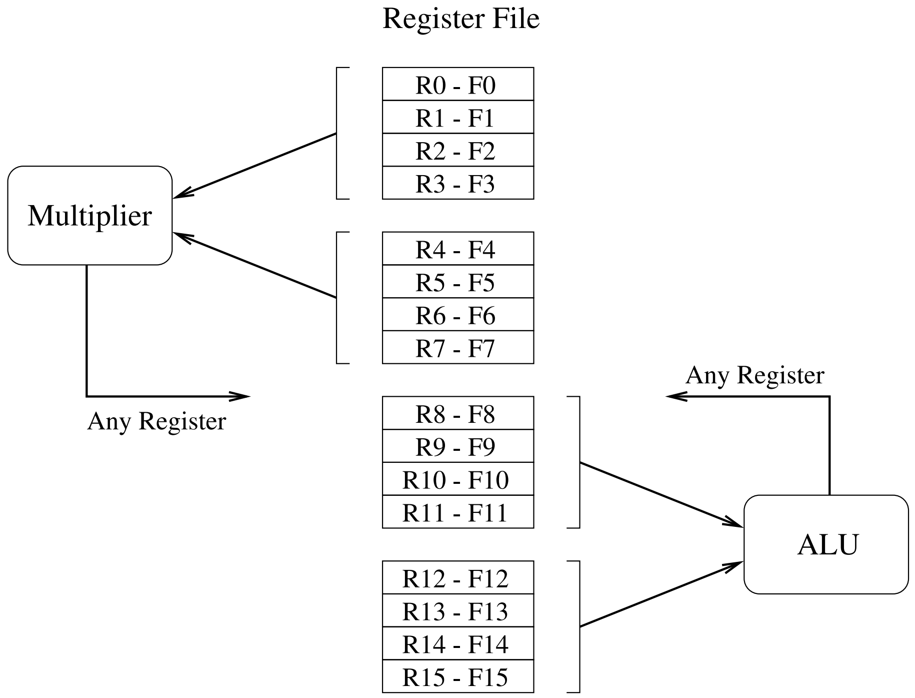


Abbildung 2.2: Eingaberegister bei Multifunktionsinstruktionen

Parallel zu einer Instruktion können zwei Operanden zwischen Speicher und Registerfile übertragen werden. Damit sind Datenübertragungen und arithmetische Operationen in derselben Instruktion möglich. In Beispiel 2.1 werden die Möglichkeiten der Parallelität demonstriert. Es ist ein Codeausschnitt angegeben, dessen Instruktionen nacheinander ausgeführt werden können.

Beispiel 2.1 Gegeben sei folgender Codeausschnitt:

```
1: r0  = dm(i4,m4);
2: r4  = pm(i12,m12);
3: r10 = r9+r13;
4: r2  = r0*r4;
```

In Instruktion 1 wird das Register $r0$ mit dem Inhalt der Speicherzelle beschrieben, die durch das Register $i4$ adressiert ist. Nach dem Speicherzugriff wird $i4$ um den in $m4$ enthaltenen Wert inkrementiert. In Instruktion 2 wird in ähnlicher Weise ein Datenzugriff auf den Programmspeicher PM ausgeführt. Die Instruktion 3 stellt eine Addition dar, deren Operandenregister zu einem früheren Zeitpunkt geladen wurden. In Instruktion 4 wird eine Multiplikation mit Operanden durchgeführt, die in Instruktion 1 und 2 in die Register $r0$ und $r4$ geladen wurden. Diese Instruktionen können auch zusammengefaßt und innerhalb eines Zyklus ausgeführt werden.

```
1: r0 = dm(i4,m4), r4 = pm(i12,m12), r10 = r9+r13, r2 = r0*r4;
```

□

Die parallele Ausführung von Instruktion 1, 2 und 4 ist ein Beispiel für die Möglichkeiten von Speicherzugriffen und Operationen auf den geladenen Operanden innerhalb von einem Zyklus. Die parallele Ausführung von Instruktion 3 und 4 demonstriert die gleichzeitige Bearbeitung von Multiplikation und Addition.

Anhand dieses Beispiels können auch Teilaufgaben der Registerallokation und Instruktionsanordnung gezeigt werden. So muß die Registerallokation dafür sorgen, daß die Operanden in geeignete Register geladen werden, damit durch die Instruktionsanordnung die Befehle derart kompakt angeordnet werden können.

Zur Notation: In Beispiel 2.1 sind die zu einer Instruktion zusammengefaßten Mikrooperationen durch Kommata getrennt, wohingegen Instruktionen, die in verschiedenen Zyklen ausgeführt werden, mittels Semikolon abgesetzt sind. Des weiteren wird die Konvention eingeführt, daß alle Instruktionen die in einem Zyklus ausgeführt werden, in einer Zeile des Assemblerprogramms zusammengefaßt sind. Instruktionen, die verschiedenen Zyklen zugeordnet sind, stehen somit in verschiedenen Zeilen des Assemblerprogramms.

2.1.3 Instruktionszyklus

Der ADSP-2106x führt Instruktionen innerhalb von drei Taktzyklen aus:

1. Im *fetch*-Zyklus wird die Instruktion aus dem Programmspeicher PM oder dem Instruktionscache abgerufen.
2. Im *decode*-Zyklus wird die Instruktion dekodiert.
3. Im *execute*-Zyklus wird die Instruktion ausgeführt, das heißt, die im Rahmen dieser Instruktion auszuführenden Operationen werden vollendet. In diesem Zyklus werden die in Abschnitt 2.1.2 eingeführten parallelen Ausführungsmöglichkeiten genutzt.

Die Zyklen sind in einer Pipeline (Befehlsfließband) angeordnet. Während eine Instruktion I_n abgerufen wird, wird bei linearem Programmfluß die Instruktion I_{n-1}, die im vorherigen Zyklus geladen wurde, dekodiert und die Instruktion I_{n-2}, die vor zwei Zyklen geladen wurde, ausgeführt. Somit beträgt der Durchsatz eine Instruktion pro Zyklus; jeder nicht-lineare Programmfluß (Sprünge) kann den Instruktionsdurchsatz jedoch einschränken.

2.2 Compiler

Ein Compiler transformiert ein in einer bestimmten Sprache (der Quellsprache) geschriebenes Programm in ein äquivalentes Programm einer anderen Sprache (der Ziel- meist Maschinensprache). In Abbildung 2.3 ist eine typische Zerlegung der Phasen eines Compilers dargestellt.

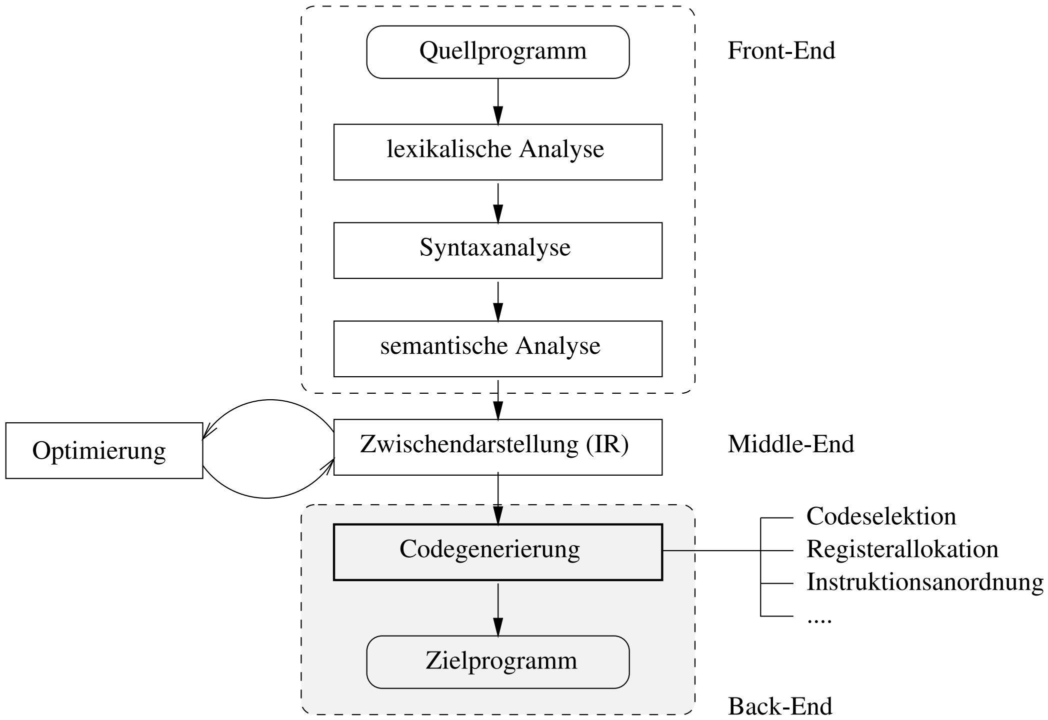


Abbildung 2.3: Phasen eines Compilers und Aufteilung in Front-, Middle- und Back-End

Als *Front-End* bezeichnet man die Phasen eines Compilers, die abhängig und spezifisch für die Quellsprache und weitgehend unabhängig von der Zielsprache sind. Hier finden grundsätzliche Analysen und Überprüfungen des Quellcodes statt. Danach betrachtet man eine Zwischenphase (*Middle-End*), in der Maschinen-unabhängige Optimierungen auf einer Zwischendarstellung (*Intermediate Representation*, *IR*) stattfinden. Dazu gehören unter anderem Transformationen zur Vereinfachung von Ausdrücken und zur Berechnung von gemeinsamen (Teil-)Ausdrücken. Das *Back-End* beinhaltet die Teile des Compilers, die sich auf die Zielsprache und damit auf die zugrundeliegende Hardware beziehen. Die wichtigste Aufgabe innerhalb des Back-Ends ist die Codegenerierung.

2.3 Aufgaben der Codegenerierung

Als Eingabe zur Codegenerierung dient eine IR des Quellprogramms, als Ausgabe wird ein semantisch äquivalentes Zielprogramm erzeugt. Ziel der Codegenerierung ist die Erzeugung eines Maschinenprogramms, mit dem das Leistungspotential der zugrundeliegenden Maschine möglichst gut ausgenutzt wird. Eine optimale Lösung kann durch die Komplexität des Problems im allgemeinen nicht berechnet werden. Von den untersuchten Aufgaben ist sowohl das Problem der Registerallokation als auch das Problem der Instruktionsanordnung $\mathcal{NP}$-vollständig [MRG79].

Die wichtigsten Teilaufgaben der Codegenerierung sind:

- **Codeselektion**: Ziel der Codeselektion ist es, Operationen, Ausdrücke und Anweisungen der IR möglichst gut auf Befehle (Instruktionen) des Befehlssatzes des Zielprozessors abzubilden. Die meisten Prozessoren bieten mehrere Befehlsfolgen als Übersetzung einer Quellsprachenanweisung an. Es gilt, eine bezüglich Ausführungszeit, Speicherplatzbedarf und/oder Befehlswortlänge kostengünstige Befehlsfolge auszuwählen.
- **Registerallokation**: Befehle, die Registeroperationen enthalten, sind meist kürzer und schneller als solche mit Speicheroperationen. Die Registerallokation hat die Aufgabe, Werte von Variablen und Ausdrücke der IR auf physikalische Register abzubilden. Dabei werden die beiden folgenden Teilprobleme unterschieden:
  - *Registervergabe*: Für jede Codezeile der IR wird die Menge der Variablen bestimmt, die Register belegen. Im allgemeinen übersteigt die Anzahl der gleichzeitig lebenden[II] Variablen die Anzahl der Register, so daß Werte zwischengespeichert werden müssen. Durch eine geschickte Auswahl der Variablen die in Registern gehalten und welche Werte zwischengespeichert werden, kann die Anzahl der Speicherzugriffe minimiert werden.
  - *Registerbindung*: Für die Variablen, die in Registern gehalten werden sollen, erfolgt hier die Auswahl eines bestimmten physikalischen Registers. Diese Aufgabe wird auch als Registerzuteilung oder Registerassignment bezeichnet.
- **Instruktionsanordnung**: Unter Instruktionsanordnung versteht man die Umordnung einer Instruktionssequenz, um effektiven Gebrauch von den Parallelverarbeitungsmöglichkeiten und dem Befehlsfließband[III] des Zielprozessors zu machen. Hierzu gehört zum Beispiel die Kompaktierung parallel ausführbarer Operationen in Multifunktionsinstruktionen beim ADSP-2106x.

Die Aufgaben der Codegenerierung sind nicht voneinander unabhängig, sondern beeinflussen sich gegenseitig. Wird zum Beispiel erst die Codeselektion und dann die Registerallokation durchgeführt, so werden möglicherweise mehr virtuelle Register eingesetzt als physikalische Register zur Verfügung stehen. In diesem Fall müssen Zwischenspeicherungen und Ladeoperationen (*Spillcode*) eingefügt werden. Umgekehrt kann der Codeselektor nach einer zuvor durchgeführten Registerallokation

[II] siehe Kapitel 2.14, Seite 24

[III] Ausnutzung des Pipelining in der sequentiellen Anordnung, um ungenutzte Fließbandtakte zu vermeiden.

möglicherweise eine günstigere Befehlssequenz nicht auswählen, weil dazu erforderliche Register fehlen.

Die Aufgaben der Codegenerierung sind außerdem von der zugrundeliegenden Hardware abhängig. So ist eine gute Registerallokation für *CISC*[IV]-Prozessoren wegen der geringen Anzahl verfügbarer Register schwierig. Da CISC-Prozessoren viele mögliche Befehlssequenzen für eine Instruktion mit eventuell sehr unterschiedlichen Kosten anbieten, ist eine gute Codeselektion basierend auf einem Kostenmaß wichtig [RW92]. Bei *RISC*[V]-Prozessoren, wie dem ADSP-2106x, ist aufgrund des einfacheren Befehlssatzes die Codeselektion leichter zu lösen als bei CISC-Prozessoren, so daß die Abhängigkeiten zwischen Codeselektion und Registerallokation an Bedeutung verlieren. Die Interaktion zwischen Instruktionsanordnung und Registerallokation nimmt hingegen an Bedeutung zu, da zur Instruktionsanordnung Register benötigt werden, um die Ausführung unabhängiger Operationen zu überlappen. Eine ausführliche Beschreibung der Instruktionsanordnung, der Registerallokation und der Möglichkeit ihrer Kopplung erfolgt in den Abschnitten 2.5 bis 2.7.

2.4 Programmdarstellungen

Zur Darstellung der Aufgaben der Codegenerierung werden verschiedene Graphen verwendet. Dazu gehören neben Kontrollfluß- und Basisblockgraphen auch Daten- und Kontrollabhängigkeitsgraphen, die im folgenden erläutert werden. Die Notation ist an [RW92] und [Bas95] angelehnt; einige für den ADSP-2106x wichtige Bemerkungen sind aus [Käs97] entnommen.

2.4.1 Kontrollflußgraph

Definition 2.1 ***Kontrollflußgraph***

Der Kontrollflußgraph (control flow graph, cfg) einer Prozedur ist ein knotenmarkierter, kantengeordneter, gerichteter Graph $G_{cf} = (N, E, n_{start})$. *Zu jeder primitiven Anweisung* p *der Prozedur gibt es einen Knoten* n_p *in der Knotenmenge* N, *der mit dieser Anweisung markiert ist. Kanten in* E *zeigen an, welche Anweisungen nacheinander ausgeführt werden müssen. Sie werden mit* i *(immediate, unmittelbar),* t *(true, wahr) oder* f *(false, falsch) markiert.* n_{start} *markiert den Prozedureintrittspunkt.*

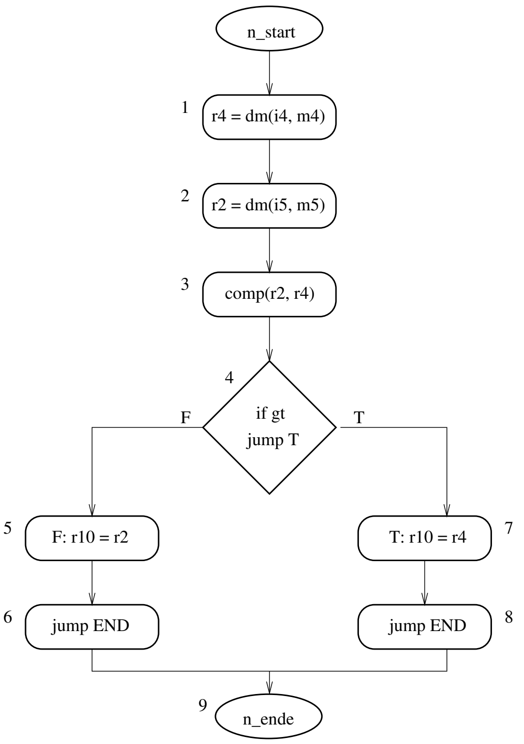
Knoten mit mehr als einem Vorgänger werden *Verschmelzungen*, Knoten mit mehr als einem Nachfolger *Verzweigungen* genannt. Im Kontrollflußgraphen sollte eine eindeutige Senke n_{ende}, das Programmende, existieren. Sind mehrere Senken vorhanden, werden diese mit einem neu hinzugefügten n_{ende}-Knoten verbunden.

In Abbildung 2.4 sind ein Beispielprogramm und der korrespondierende Kontrollflußgraph gegenübergestellt.

[IV] *Complex Instruction Set Computer*
[V] *Reduced Instruction Set Computer*

```
1:  r4 = dm(i4, m4);
2:  r2 = dm(i5, m5);
3:  comp(r2, r4);
4:  if gt jump T;
5:  F:  r10 = r2;
6:       jump END;
7:  T:  r10 = r4;
8:       jump END;
9:  END;
```

a) Programm



b) Kontrollflußgraph

Abbildung 2.4: Programmbeispiel und zugehöriger Kontrollflußgraph

2.4.2 Basisblockgraph

In einem Kontrollflußgraphen wie in Abbildung 2.4 gibt es Pfade, auf denen der Graph nicht aufspaltet oder zusammenläuft. Diese Sequenzen von primitiven Anweisungen werden häufig in sogenannten *Basisblöcken* zusammengefaßt und zusammen verarbeitet.

Definition 2.2 ***Basisblock***

Ein Basisblock in einem Kontrollflußgraphen ist ein Pfad maximaler Länge, der höchstens am Anfang eine Verschmelzung und höchstens am Ende eine Verzweigung hat.

Für jeden Basisblock in einer sequentiellen Programmiersprache gilt: Wird seine erste Anweisung ausgeführt, dann werden sequentiell die folgenden Anweisungen ausgeführt, falls nicht Laufzeitfehler, Ausnahmen (*exceptions*) o.ä. auftreten.

Definition 2.3 ***Basisblockgraph***

Der Basisblockgraph G_{bb} eines Kontrollflußgraphen G_{cf} entsteht aus G_{cf}, indem Basisblöcke zu einem Knoten zusammengefaßt werden. Kanten in G_{cf}, die zum ersten Knoten eines Basisblockes führen, führen in G_{bb} in den Knoten des Basisblockes hinein; Kanten, die in G_{cf} aus dem letzten Knoten eines Basisblockes herausführen, führen in G_{bb} aus dem Knoten des Basisblockes heraus.

In Abbildung 2.5 ist der Basisblockgraph des Kontrollflußgraphen aus Abbildung 2.4 dargestellt. Die Basisblöcke sind grau hervorgehoben, innerhalb der Knoten des Basisblockgraphen sind die dazugehörigen Instruktionen aufgeführt.

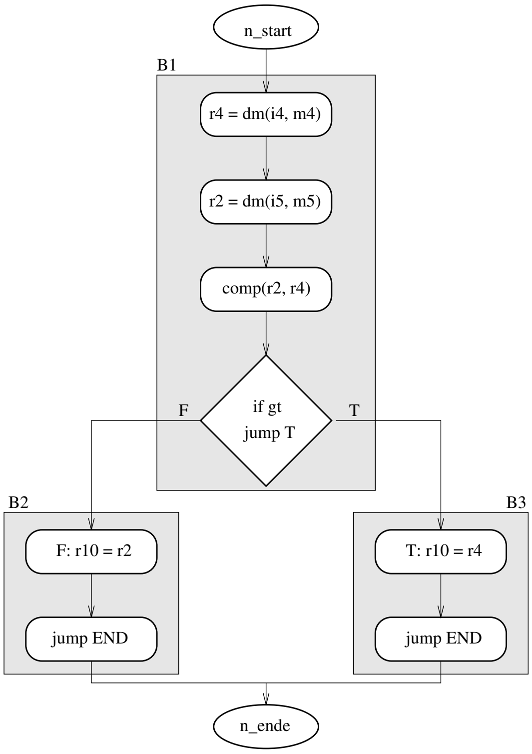


Abbildung 2.5: Der Basisblockgraph zu Abbildung 2.4

2.4.3 Datenabhängigkeitsgraph

Die Befehle innerhalb eines Basisblockes können unter Beachtung gewisser Einschränkungen neu angeordnet werden. Diese Einschränkungen ergeben sich durch die Datenabhängigkeiten zwischen den Befehlen. Abhängigkeiten bestehen in der Reihenfolge lesender oder schreibender Zugriffe auf Statusregister, Register- oder Speicherzellen. Schreibende Zugriffe werden als *Setzungen* (*definition*), lesende Zugriffe als *Benutzungen* (*uses*) bezeichnet. Eine *Ressource* ist zum Beispiel eine Speicher- bzw. Registerzelle, oder eine funktionale Einheit wie die ALU.

Definition 2.4 ***Datenabhängigkeitsgraph eines Basisblocks***

Es sei ein Basisblock eines Maschinenprogramms gegeben. Sein Datenabhängigkeitsgraph ist ein markierter, azyklischer, gerichteter Graph $G_D = (V_D, E_D)$, dessen Knoten mit den Befehlen des Basisblockes markiert sind. Es führt eine Kante vom Knoten eines Befehls a zum Knoten eines anderen Befehls b, wenn a vor b ausgeführt werden muß, d.h. wenn a in der Befehlsfolge vor b steht und wenn mindestens eine der folgenden Abhängigkeiten besteht:

1. *a eine Ressource setzt, b sie benutzt und der Weg von a nach b setzungsfrei ist (Setzungs-Benutzungs-Abhängigkeit, true dependence)*
2. *a eine Ressource benutzt, b sie setzt und der Weg von a nach b setzungsfrei ist (Benutzungs-Setzungs-Abhängigkeit, anti dependence)*
3. *a und b die gleiche Ressource setzen und der Weg von a nach b setzungs- und benutzungsfrei*[VI] *ist (Setzungs-Setzungs-Abhängigkeit, output dependence).*

Bezeichnet E_D^{true} *die Menge der Setzungs-Benutzungs-Abhängigkeiten,* E_D^{anti} *die Menge der Benutzungs-Setzungs-Abhängigkeiten und* E_D^{output} *die Menge der Setzungs-Setzungs-Abhängigkeiten, dann kann die Kantenmenge* E_D *des Datenabhängigkeitsgraphen geschrieben werden als:*

$$E_D \;=\; E_D^{true} \;\cup\; E_D^{anti} \;\cup\; E_D^{output}.$$

Beispiel 2.2 In Abbildung 2.6 definieren die Instruktionen 1 und 2 Register, die in Instruktion 3 als Operanden der Multiplikation verwendet werden. Damit besteht eine Setzungs-Benutzungs-Abhängigkeit (*true*) zwischen den Instruktionen 1 und 3 sowie zwischen 2 und 3. Instruktion 3 führt einen lesenden Zugriff, die nachfolgende Instruktion 4 einen schreibenden Zugriff auf Register $r0$ aus. Somit besteht eine Benutzungs-Setzungs-Abhängigkeit (*anti*) zwischen den Instruktionen 3 und 4. Da zwischen den Setzungen der Instruktionen 1 und 4 in Instruktion 3 eine Benutzung des Registers $r0$ erfolgt, liegt in diesem Fall keine Setzungs-Setzungs-Abhängigkeit vor. Die Instruktionen 4 und 5 beschreiben jedoch ohne dazwischenliegende Setzungen oder Benutzungen dasselbe Register, damit besteht zwischen diesen Instruktionen eine Setzungs-Setzungs-Abhängigkeit (*output*).

```
1: r0 = dm(i0, m0);
2: r1 = pm(i8, m8);
3: r2 = r0 * r1;
4: r0 = r3 + r4;
5: r0 = r5 + r6;
```

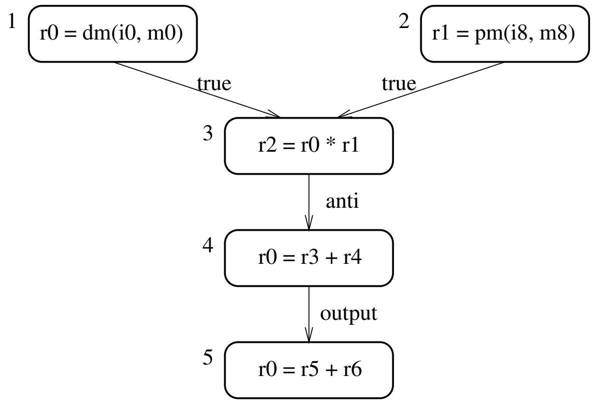


Abbildung 2.6: Programmbeispiel mit korrespondierendem Datenabhängigkeitsgraphen

□

[VI]Die Benutzungsfreiheit wird im allgemeinen nicht gefordert, ist aber Teil der Definition nach [RW92].

Der Datenabhängigkeitsgraph ist auf einzelne Basisblöcke begrenzt. Innerhalb eines Basisblockes sind jedoch die Möglichkeiten zur Parallelisierung in der Regel sehr beschränkt.

Definition 2.5 ***Abhängigkeitsgraph***

Erweitert man die Definition des Datenabhängigkeitsgraphen, so daß auch Abhängigkeiten zwischen verschiedenen Basisblöcken erfaßt werden, spricht man vom erweiterten oder verallgemeinerten Abhängigkeitsgraphen.

Der Abhängigkeitsgraph erweist sich jedoch bei globalen Analysen als unzureichend. Eine Problematik bei globaler Analyse wird in Beispiel 2.3 gezeigt.

Beispiel 2.3 In Abbildung 2.7 wird ein Beispiel für die Problematik bei globaler Betrachtung gezeigt. Allein durch die Datenabhängigkeiten kann die Verschiebung der Instruktion $r6 = r10 + r11$ aus Basisblock $B3$ in den Basisblock $B2$, also in den *false*-Teil, nicht ausgeschlossen werden, da zu dieser Instruktion keine Datenabhängigkeiten existieren. Eine solche Verschiebung hätte jedoch zur Folge, daß die Instruktion $r6 = r10 + r11$, abhängig vom Wert der Bedingung der *if*-Anweisung, vor oder nach der Anweisung $r10 = r2$ ausgeführt wird. Damit würde aber die Semantik des Eingabeprogramm verändert werden.

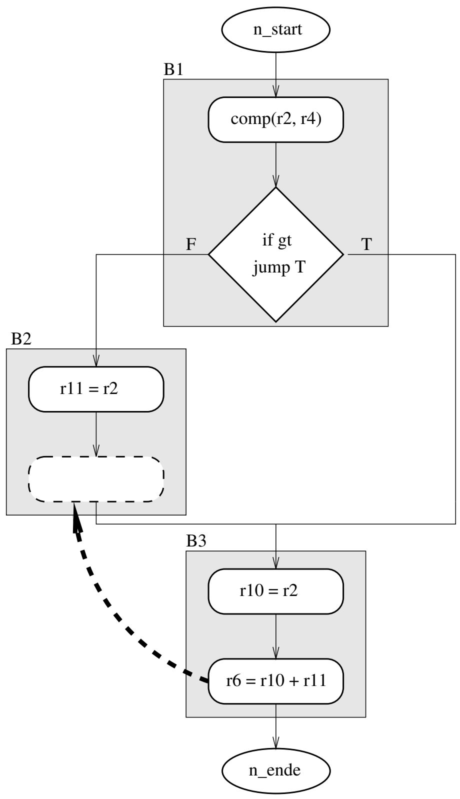


Abbildung 2.7: Problematik bei globaler Basisblockbetrachtung

□

2.4.4 Kontrollabhängigkeitsgraph

Daher ist neben dem Abhängigkeitsgraphen auch die Kontrollstruktur des Eingabeprogramms zu berücksichtigen. Die Kontrollabhängigkeiten werden durch einen weiteren Graphen modelliert, den *Kontrollabhängigkeitsgraphen*, zu dessen Definition noch einige Grundlagen geschaffen werden müssen.

Definition 2.6 ***Dominator***

Ein Knoten x dominiert einen Knoten y ($x \Delta_d y$), genau dann, wenn jeder Pfad im Kontrollflußgraph G_{cf} vom Eintrittspunkt n_{start} zu y den Knoten x enthält. x heißt Dominator von y. Jeder Knoten dominiert sich selbst.

Diese Definition besagt, daß bei Ausführung von y auch immer x ausgeführt wird, wenn Knoten x den Knoten y dominiert. Nach dieser Definition betrachtet man noch den Begriff des *unmittelbaren Dominators*.

Definition 2.7 ***unmittelbarer Dominator***

Die Menge der Dominatoren eines Knotens y bildet eine Kette. x ist genau dann unmittelbarer Dominator von y, wenn

- $x \Delta_d y$ *und*
- $\neg \exists z \ : x \Delta_d z \ \wedge \ z \Delta_d y \ \wedge \ z \neq x.$

Definition 2.8 ***Dominatorbaum***

Der Dominatorbaum eines Kontrollflußgraphen ist ein Baum, der die Knoten des Kontrollflußgraphen enthält. Seine Wurzel ist der Eintrittsknoten n_{start}. Es existiert eine Kante zwischen x und y genau dann, wenn x unmittelbarer Dominator von y ist.

In Analogie hierzu läßt sich auch die *Post-Dominanz* definieren.

Definition 2.9 ***Post-Dominator***

Ein Knoten x ist Post-Dominator von einem Knoten y ($x \Delta_p y$) genau dann, wenn jeder Pfad im Kontrollflußgraphen von y zum eindeutigen Endknoten n_{ende} den Knoten x enthält. Ein Knoten post-dominiert sich nie selbst.

Definition 2.10 ***unmittelbarer Post-Dominator***

Die Menge der Post-Dominatoren eines Knotens y bildet eine Kette. x ist unmittelbarer Post-Dominator von y, genau dann, wenn

- $x \Delta_p y$ *und*
- $\neg \exists z \ : y \Delta_p z \ \wedge \ z \Delta_p x \ \wedge \ z \neq y.$

Definition 2.11 ***Post-Dominatorbaum***

Der Post-Dominatorbaum eines Kontrollflußgraphen ist ein Baum, der die Knoten des Kontrollflußgraphen enthält. Seine Wurzel ist der Endeknoten n_{ende}. Es existiert eine Kante zwischen x und y genau dann, wenn x unmittelbarer Post-Dominator von y ist.

Mit Hilfe dieser Definitionen können die *Kontrollabhängigkeit* und der zugehörige Graph definiert werden.

Definition 2.12 ***Kontrollabhängigkeit***

Gegeben sei ein Kontrollflußgraph $G_{cf} = (N, E_{cf}, n_{start})$. Ein Knoten $x \in N$ hat genau dann Kontrollabhängigkeit über $y \in N$ ($x\ \delta_c^a\ y$), wenn gilt:

1. $(x, a) \in E_{cf}$
2. *y post-dominiert nicht x,* $\neg(x\ \Delta_p\ y)$
3. *Es existiert ein Pfad $p = x, a, \ldots, y$, so daß $\forall z$ mit $z \neq x$, $z \neq y$ gilt: $y\ \Delta_p\ z$*

Gilt $x\ \delta_c^a\ y$, dann sagt man auch, daß Knoten y kontrollabhängig von x ist.

Definition 2.13 ***Kontrollabhängigkeitsgraph***

Der Kontrollabhängigkeitsgraph G_{cd} des Kontrollflußgraphen $G_{cf} = (N, E_{cf}, n_{start})$ ist ein gerichteter Graph $G_{cd} = (N, E_{cd})$ mit Kantenmarkierung, so daß $(x, y, tf) \in E_{cd} \Leftrightarrow x\ \delta_c^a\ y$ und die Kante $(x, y) \in E_{cf}$ ist mit T bzw. F markiert.

Beispiel 2.4 Abbildung 2.8 zeigt ein Beispiel eines Kontrollflußgraphen; der dazugehörige Kontrollabhängigkeitsgraph ist in Abbildung 2.9 dargestellt. Knoten 1 hat Kontrollabhängigkeit über die Knoten 2 und 3, da diese nur ausgeführt werden, wenn der bedingte Ausdruck in Knoten 1 den Wert wahr annimmt. Knoten 3 ist nicht kontrollabhängig von 2, weil er Knoten 2 post-dominiert. Knoten 9 ist Post-Dominator jedes Knoten außer n_{ende} und sich selbst, daher ist er von keinem Knoten kontrollabhängig.

□

2.5 Instruktionsanordnung

Unter Algorithmen zur Instruktionsanordnung werden Verfahren verstanden, die durch Umordnung der Instruktionen der IR eines Programms dieses in Hinblick auf die Ausführungszeit verbessern. Die Verfahren lassen sich in zwei Klassen aufteilen: lokale und globale. Lokale Verfahren arbeiten auf einzelnen Basisblöcken, globale Verfahren betrachten das gesamte Programm bzw. komplette Pfade (*traces*) im Verlauf der Bearbeitung.

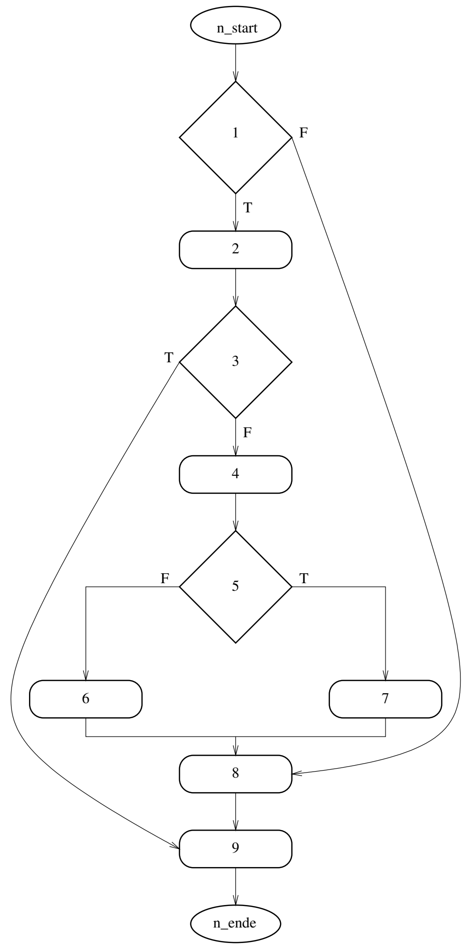


Abbildung 2.8: Der Kontrollflußgraph zu Beispiel 2.4

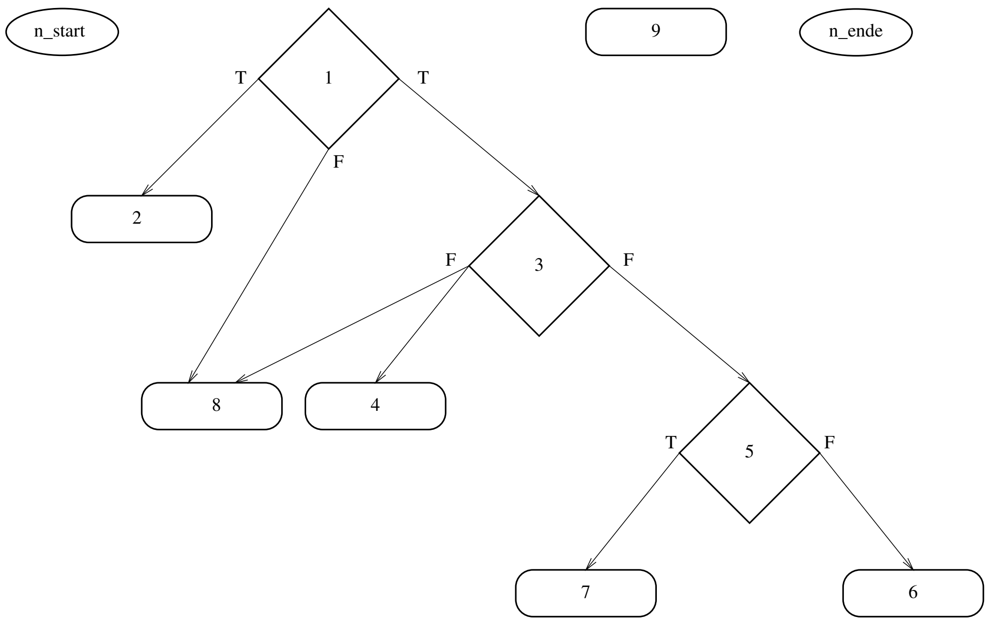


Abbildung 2.9: Der Kontrollabhängigkeitsgraph zu Beispiel 2.4

Übertragen auf die Graphen aus Abschnitt 2.4 entsprechen lokale Verfahren der Lösung einer Darstellung im Datenabhängigkeitsgraphen; bei globaler Betrachtung muß zusätzlich dafür gesorgt werden, daß die Kontrollabhängigkeiten beim Umordnungsprozeß gewahrt bleiben. Diese werden in einem Kontrollabhängigkeitsgraphen ausgedrückt.

Zur Lösung des Scheduling-Problems werden sowohl heuristische Verfahren, die approximative, suboptimale Lösungen finden, als auch exakte Verfahren, die eine optimale Lösung berechnen, verwendet. Das am häufigsten verwendete heuristische lokale Verfahren ist *List-Scheduling* [Gas89], für globale Instruktionsanordnung wird häufig *Trace-Scheduling* [Fis81] eingesetzt.

List-Scheduling beginnt mit einer leeren Liste von Instruktionen. Eine Mikrooperation[VII] wird in die letzte Instruktion der Liste eingefügt, wenn sie die drei folgenden Bedingungen erfüllt:

1. Sie ist ausführbereit, d.h. alle Vorgänger im Datenabhängigkeitsgraphen wurden bereits angeordnet.
2. Sie hat die höchste Priorität innerhalb der Menge aller ausführbereiten Mikrooperationen. Diese Priorität wird aufgrund von Gewichtungsheuristiken berechnet.
3. Die Einfügung in die letzte Instruktion ist zulässig, d.h. alle bereits enthaltenen Mikrooperationen können parallel zu der aktuell betrachteten ausgeführt werden.

Kann keine der ausführbereiten Mikrooperationen in die letzte Instruktion eingefügt werden, wird eine neue Instruktion an die Liste angehängt. Die Komplexität beträgt $\mathcal{O}(n^2)$ [Gas89].

Beim Trace-Scheduling werden an Stelle der Basisblöcke komplette Pfade im Kontrollflußgraphen betrachtet. Ein Pfad ist hierbei eine schleifenfreie Anordnung von Instruktionen, die aufgrund gegebener Daten nacheinander ausgeführt werden können. Die einzelnen Pfade in der Menge werden wie Basisblöcke behandelt und darauf List-Scheduling durchgeführt. Sprünge im Programm sowie Verschiebungen von Instruktionen über ursprüngliche Basisblockgrenzen führen gegebenenfalls zur Einfügung von Kompensationscode. Die einzelnen Basisblöcke/Pfade erhalten eine Priorität, die sich aus der Ausführungshäufigkeit ergibt. Diese Prioritäten werden zum Beispiel durch Probeläufe des Programm bestimmt. Basisblöcke mit der höchsten Priorität, also die am häufigsten durchlaufenen Pfade, werden zuerst angeordnet. Eine ausführliche Übersicht zu Verfahren der Instruktionsanordnung findet sich in [Bas95] und [Muc97].

Als exakte Verfahren zur Lösung des Scheduling-Problems können zum Beispiel ILP und CLP verwendet werden. Die meisten exakten Algorithmen verwenden ILP, um

[VII] Eine Mikrooperation ist zum Beispiel ein Speicherzugriff; Mikrooperationen können, je nach Hardware und Befehlssatz, zu Instruktionen zusammengefaßt werden.

optimale Lösungen zu berechnen. Die ILP-Verfahren sind durch ihren Berechnungsaufwand nur für kleine Codesequenzen geeignet [Käs97]. Ob sich CLP-Verfahren auch, oder besser, zur Berechnung von exakten Lösungen eignen, ist ein Thema dieser Diplomarbeit.

2.6 Registerallokation

Mit dem folgenden Verfahren zur Registerallokation werden global alle Register der Zielmaschine für alle Berechnungen innerhalb einer Prozedur vergeben. Die im Prinzip unbeschränkt vielen symbolischen Register sind den beschränkt vielen Registern der realen Maschine zuzuordnen. Dabei kann niemals ein reales Register zwei verschiedenen symbolischen Registern zugeteilt werden, wenn diese beide gleichzeitig *lebendig* sind und nicht sicher denselben Wert enthalten. Die nachfolgenden Definitionen basieren auf der Darstellung des Eingabeprogramms in Form eines Kontrollflußgraphen.

Definition 2.14 ***lebendig, Lebensspanne***

Ein symbolisches Register r *ist lebendig an einem Programmpunkt* p, *wenn es auf einem Programmpfad vom Eintrittsknoten der Prozedur nach* p *eine Setzung von* r *gibt und einen Pfad von* p *zu einer Benutzung von* r, *auf der* r *nicht gesetzt wird. Die Lebensspanne eines symbolischen Registers* r *ist die Menge der Programmpunkte, an denen* r *lebendig ist.*

Ein Register ist an einem Programmpunkt lebendig, wenn sein dortiger Inhalt noch benötigt werden kann. Der *Registerkollisionsgraph* drückt die Beschränkung für die Zuteilung von symbolischen an reale Register aus.

Definition 2.15 ***Kollision, Registerkollisionsgraph***

Zwei Lebensspannen von symbolischen Registern kollidieren, wenn eins von ihnen in der Lebensspanne des anderen gesetzt wird. Der Registerkollisionsgraph ist ein ungerichteter Graph. Seine Knoten sind Lebensspannen von symbolischen Registern, und es besteht eine Kante zwischen je zwei kollidierenden Lebensspannen.

Beispiel 2.5 In Abbildung 2.10 werden für ein Beispielprogramm (1) die Lebensspannen (2) und der Registerkollisionsgraph (3) dargestellt:

1. Beschreibt ein Beispielprogramm, in dem einige Register gesetzt und gelesen werden.

2. Gibt die Lebensspannen aller darin definierten Variablen an. So wird zum Beispiel das Register c in der dritten Instruktion gesetzt und in der fünften gelesen, so lange ist das Register c lebendig.

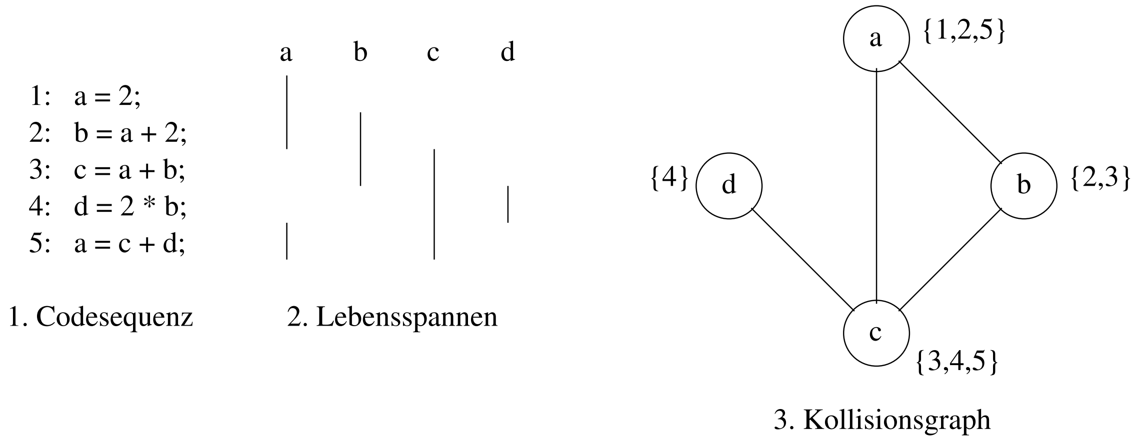


Abbildung 2.10: Programm zu Beispiel 2.5 mit Lebensspannen und Registerkollisionsgraph

3. Zeigt den Registerkollisionsgraph, an dessen Knoten die Lebensspannen der symbolischen Register aufgeführt sind (bei Knoten c die Instruktionen $\{3, 4, 5\}$). Eine Kante existiert zwischen zwei kollidierenden Lebensspannen wie zum Beispiel zwischen a und c, die beide in Instruktion 5 lebendig sind.

□

Eine globale Registerallokation muß, ebenso wie die globale Instruktionsanordnung, die Kontrollstruktur der zu übersetzenden Prozedur beachten. Es existieren hierzu verschiedene Verfahren, das bekannteste ist der Graphfärbealgorithmus [Bri92], bei dem sowohl Registerallokation als auch Registerzuteilung berechnet werden.

Das Registerallokationsproblem wird übersetzt in das Problem, den Registerkollisionsgraphen mit k Farben zu färben, wobei k die Anzahl der zur Verfügung stehenden realen Register ist. Die Farben entsprechen also realen Registern. Bei der Graphfärbung dürfen keine direkt verbundenen Knoten die gleiche Farbe zugeteilt bekommen. Für $k > 2$ ist das Problem, für beliebige Graphen zu entscheiden, ob sie k-färbbar sind, $\mathcal{NP}$-vollständig [MRG79]. Deshalb werden in der Praxis heuristische Methoden eingesetzt, um eine Färbung zu bestimmen.

Die Grundüberlegung für den Graphfärbealgorithmus ist folgende: Enthält der Graph G einen Knoten n mit Grad kleiner als k, so kann n mit Sicherheit eine Farbe zugeordnet werden, die von den Farben aller Nachbarn verschieden ist. Der Knoten n wird aus G entfernt, wodurch sich ein neuer Graph G' ergibt, der einen Knoten und einige Kanten weniger enthält.

Kann keine k-Färbung gefunden werden - was nicht bedeutet, daß der Graph nicht k-färbbar ist -, müssen einige Variablen zwischengespeichert werden (*Spilling*).

Sei x ein Knoten, dessen Nachbarn die Farben $\{1, \ldots, k\}$ haben, so daß keine Farbe mehr für x verbleibt. Dann gibt es drei Ansätze zur Lösung dieses Problems [Bas95]:

1. x wird im Hauptspeicher zwischengespeichert; alle Zugriffe auf x werden auf den Hauptspeicher abgebildet. Diese Methode kann für eine Load/Store-Architektur wie den zugrundeliegenden ADSP-2106x jedoch nicht angewendet werden.

2. Die Einführung von Zwischenspeicherungen (Spillcode) speichert eine Variable nach jeder Definition und lädt sie vor jeder Verwendung.

3. Die Lebenszeit wird in mehrere, disjunkte Lebenszeiten aufgespalten. Damit wird Spillcode nur an bestimmten Stellen einer Lebenszeit eingefügt und an bestimmten Stellen wieder geladen. Jede neue Lebenszeit stellt einen neuen Knoten im Registerkollisionsgraphen dar, wobei es gewöhnlich weniger Kollisionen gibt, als mit der ursprünglichen Lebenszeit.

Das Einfügen von Zwischenspeicherungen oder das Aufspalten von Lebensspannen führt zu einem neuen Kollisionsgraphen. Dabei kann der zweite Ansatz als Extremfall von Ansatz 3 angesehen werden. Das Aufspalten von Lebensspannen, d.h. die Bestimmung der aufzuspaltenden Lebensspannen und der Stellen der Aufspaltung ist jedoch sehr schwierig. Die Bestimmung von optimalen Lösungen ist für beide Probleme $\mathcal{NP}$-hart [MRG79].

Die Probleme der Aufspaltung von Lebensspannen bzw. Einfügung von Zwischenspeicherungen sind in eine ILP-Beschreibung nicht vollständig zu integrieren, da jedes aufgestellte lineare Programm sich immer auf eine fest vorgegebene Instruktionssequenz bezieht. Allerdings kann bei der Modellierung[VIII] in dem Fall, daß die vorhandenen Register nicht ausreichen, eine Empfehlung für eine Aufspaltung einer Lebensspanne gegeben werden. Aufgrund dieser Empfehlung kann eine umgebende Allokationsfunktion die vorgeschlagene Aufspaltung vornehmen und danach die Erzeugung eines neues ILPs für die geänderte Befehlsfolge durchführen. Die ermittelten Lösungen sind also optimal bezüglich des Resultates der Codeselektion und bezüglich der zuvor durchgeführten Verteilung von Variablen auf Register bzw. Hauptspeicher; Aufspaltungen von Lebensspannen und Einfügungen von Zwischencode werden nicht erfaßt.

2.7 Phasenkopplung

Bei der seriellen Bearbeitung von Phasen der Codegenerierung kann die zuerst durchgeführte Teilaufgabe der zweiten Beschränkungen auflegen, die zu einem ineffizienten Code führen können. Wird zum Beispiel die Registerallokation zuerst durchgeführt, kann sie die Überlappung von Operationen verhindern, indem sie die Operanden, aufgrund unnötiger Datenabhängigkeiten, die hier aber noch nicht erkannt werden können, in ungünstige Registerfiles schreibt. Wird die Instruktionsanordnung zuerst durchgeführt, kann sie die Anzahl der gleichzeitig lebenden Werte so stark erhöhen, daß viele dieser Werte im Hauptspeicher zwischengespeichert werden müssen. Dies führt zur Einfügung von Spillcode. Ein zusätzliches Problem beim ADSP-2106x ist,

[VIII] siehe Kapitel 3.2, Seite 37

daß beispielsweise Multiplikation und Addition parallel ausführt werden können, die Operanden dazu allerdings an bestimmte Registermengen gebunden werden müssen. Werden die Register nicht in Hinblick auf die Instruktionsanordnung vergeben, wird die parallele Ausführung verhindert. Diese Probleme werden mit dem Begriff Phasenanordnungsproblem benannt; eine Lösung ist die Verschränkung dieser Phasen (Phasenkopplung). Ziel der Phasenkopplung ist eine Verbesserung der Codequalität; zu Konflikten kommt es vor allem durch die unterschiedlichen Aspekte der Registervergabe. Bei der Modellierung dieser Aspekte müssen zusätzliche Abhängigkeiten formuliert werden, wodurch der Berechnungsaufwand für diese Optimierungsprobleme wesentlich größer wird. Eine ausführliche Betrachtung zur Phasenkopplung findet sich in [Bas95].

Kapitel 3

ILP

In Abschnitt 3.1 wird eine Einführung in die Theorie von linearer und ganzzahliger linearer Programmierung[I] gegeben. Es werden Lösungsstrategien für ILP-Verfahren mit dem Simplex-Algorithmus und Branch-and-Bound Methoden erläutert. In Abschnitt 3.2 werden zwei Modelle zur Codegenerierung mittels ILP, die in [Käs97] untersucht worden sind, vorgestellt: SILP und *Optimal Architecture Synthesis with Interface Constraints* (*OASIC*). Es wird begründet, warum für die weiteren Betrachtungen nur das SILP-Modell ausgewählt worden ist. Im Anschluß sind in Abschnitt 3.3 die Ergebnisse der Berechnungen mit CPLEX zur Lösung der SILP-Programme aufgeführt.

3.1 Theorie der linearen Programmierung

Die lineare Programmierung ist eine Methode, die ihren Ursprung im Bereich des *Operation Research* hat und immer häufiger zur Lösung von Optimierungsproblemen verwendet wird [WD92]. Typische Anwendungen sind die Verwaltung und der effiziente Einsatz knapper Ressourcen zur Steigerung der Produktivität.

3.1.1 Lineares Programm

Definition 3.1 ***Lineares Programm***

Eine Aufgabe

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &\leq b_2 \\ \vdots \quad &\quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &\leq b_m \end{aligned} \tag{3.1}$$

$$c_1x_1 + c_2x_2 + \ldots + c_nx_n = G \rightarrow Max \tag{3.2}$$

$$x_1, x_2, \ldots, x_n \geq 0 \tag{3.3}$$

[I] ganzzahliges LP $\doteq$ Integer LP $\doteq$ ILP *heißt Lineares Programm (LP) oder Lineares Optimierungsproblem. Neben den ($\geq$)-Beziehungen sind auch ($\leq$)-Beziehungen und Gleichungen möglich, statt der Maximierung kann auch die Minimierung Zielsetzung der Aufgabenstellung sein.*

Die x_j ($x \in \mathbb{R}, j \in \{1, \ldots, n\}$) heißen Entscheidungs- oder Strukturvariablen des Problems.

Die Beziehungen 3.1 heißen Nebenbedingungen oder Restriktionen, die sogenannte Zielfunktion ist in 3.2 gegeben und die Ungleichungen 3.3 heißen Nichtnegativitätsbedingungen.

Die b_i, c_i und a_{ij} (mit $b, c, a \in \mathbb{R}, i \in \{1, 2, \ldots, m\}$ und $j \in \{1, 2, \ldots, n\}$) sind konstante Werte und hängen von der konkret vorliegenden Problemstruktur ab [WD92].

Definition 3.2 ***zulässige, optimale Lösung***

1. *Ein Vektor $\vec{x} \in \mathbb{R}^n$, der die Bedingungen 3.1 und 3.3 erfüllt, heißt zulässige Lösung des Linearen Problems.*
2. *Die Menge der zulässigen Lösungen heißt zulässiger Bereich.*
3. *Eine zulässige Lösung, für welche die Zielfunktion maximiert (minimiert) wird, heißt optimale Lösung des Linearen Problems [WD92].*

Im weiteren werden noch zwei andere Schreibweisen für Lineare Programme verwendet, die Summen- und die Matrizenschreibweise:

Summenschreibweise

$$
\begin{array}{lcllcl}
\sum_{j=1}^{n} a_{ij} x_j & \leq & b_i, & i & = & 1, 2, \ldots, m \\
\sum_{j=1}^{n} c_j x_j & = & G \rightarrow Max & & & \\
x_j & \geq & 0, & j & = & 1, 2, \ldots, n
\end{array}
\tag{3.4}
$$

Matrizenschreibweise

$$
\begin{array}{lcl}
A\vec{x} & \leq & \vec{b} \\
\vec{c}^{\,T}\vec{x} & = & G \rightarrow Max \\
\vec{x} & \geq & 0
\end{array}
\tag{3.5}
$$

mit den Vektoren $\vec{x}, \vec{c}$ und $\vec{b}$:

$$
\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \vec{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}
$$

und der Koeffizientenmatrix A:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

Beispiel 3.1 In den verschiedenen Schreibweisen können Lineare Programme wie folgt dargestellt werden:

1. Als Gleichungssystem wie in den Gleichungen 3.1 bis 3.3:

$$\begin{array}{lrcl} Nebenbedingungen: & 4x_1 + 3x_2 & \leq & 600 \\ & 2x_1 + 2x_2 & \leq & 320 \\ & 3x_1 + 7x_2 & \leq & 840 \\ & & & \\ Zielfunktion: & 2x_1 + 3x_2 & = & G \rightarrow Max \\ Nichtnegativitätsbedingung: & x_1, x_2 & \geq & 0 \end{array}$$

2. In Summenschreibweise wie in Gleichung 3.4:

$$\begin{array}{lcll} \sum_{j=1}^{2} a_{ij} x_j & \leq & b_i, & i \in \{1,2,3\} \\ \sum_{j=1}^{2} c_j x_j & = & G \rightarrow Max & \\ x_j & \geq & 0, & j \in \{1,2\} \end{array}$$

$$\begin{array}{ll} mit & a_{11} = 4, a_{12} = 3, a_{21} = 2, a_{22} = 2, a_{31} = 3, a_{32} = 7 \\ & b_1 = 600, b_2 = 320, b_3 = 840 \\ & c_1 = 2, c_2 = 3 \end{array}$$

3. In Matrizenschreibweise wie in Gleichung 3.5:

Hier sind $\vec{b}, \vec{c}$ und $\vec{x}$ die Vektoren:

$$\vec{b} = \begin{pmatrix} 600 \\ 320 \\ 840 \end{pmatrix}, \quad \vec{c} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}, \quad \vec{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

und A die Koeffizientenmatrix:

$$A = \begin{pmatrix} 4 & 3 \\ 2 & 2 \\ 3 & 7 \end{pmatrix}$$

□

3.1.2 Lösung von Linearen Programmen mit dem Simplex-Algorithmus

Zur Lösung von Linearen Programmen wurden verschiedene Lösungsstrategien entwickelt, eine der bekanntesten ist der *Simplex-Algorithmus* [Dan66]. Die Kernidee des Simplex-Algorithmus ist, daß sich die Suche nach der optimalen Lösung des LPs auf endlich viele Punkte (Eckpunkte) des zulässigen Bereichs beschränken läßt [Neu75].

Diese Idee läßt sich anschaulich wie folgt beschreiben: Die Nebenbedingungen eines LP definieren ein Polyeder[II] in $I\!R$. Das *Simplex-Theorem* besagt, daß mindestens eine der Ecken diese Polyeders eine optimale Lösung des LPs darstellt. Um eine solche Ecke zu finden, wählt man eine beliebige Ecke des Polyeders als Startpunkt und wechselt zu einer benachbarten Ecke, wenn diese einen besseren Wert bezüglich der Zielfunktion ergibt. Dieses Vorgehen wiederholt man so lange, bis die Ecke erreicht ist, die der optimalen Lösung entspricht.

Beispiel 3.2 Für den zweidimensonalen Fall ist ein möglicher Berechnungsverlauf in Abbildung 3.1 gezeigt. Dabei stellt P den Zulässigkeitsbereich des LPs dar und Z die Zielfunktion. Als Startecke wird $X(Lösung\ 1)$ berechnet; danach wird über die Punkte $X(Lösung\ 2)$ und $X(Lösung\ 3)$ die optimale Lösung $X(Optimum)$ erreicht.

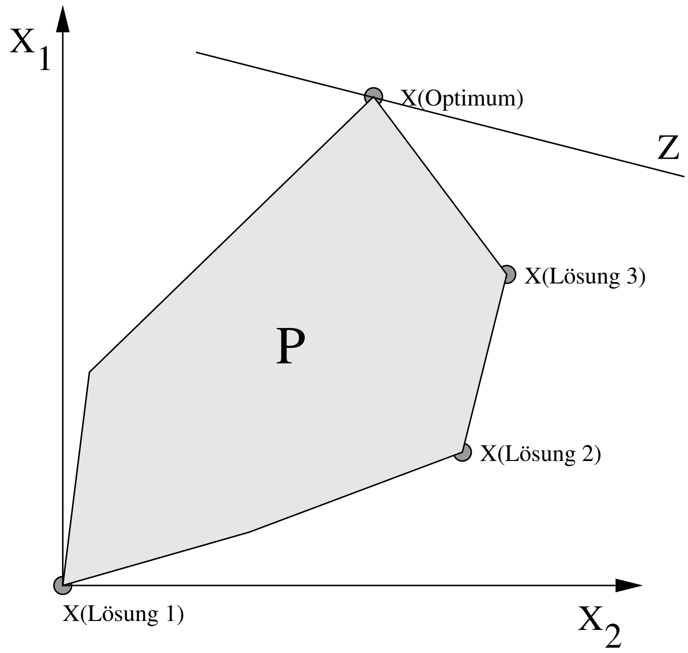


Abbildung 3.1: Möglicher Berechnungsverlauf des Simplex-Algorithmus

□

[II]Ein Polyeder ist ein Körper, der von Ebenen begrenzt wird [Bro89]. Im zweidimensonalen Fall wird er von Geraden, für $I\!R^n$, $n > 3$, von Hyperebenen begrenzt.

Als *Primaler* und *Dualer* Simplex-Algorithmus findet dieses Verfahren Anwendung im CPLEX-Solver. In diesem Zusammenhang wird das Problem 3.5 auch als *primales* Problem bezeichnet. Das dazugehörige sogenannte *duale* Problem, kurz das *Duale*, wird wie folgt definiert.

Dualproblem in Summenschreibweise

$$\begin{array}{lclcl} \sum_{i=1}^{m} a_{ij} u_i & \geq & c_j, & j & = & 1, 2, \ldots, n \\ \sum_{i=1}^{m} b_i u_i & = & K \rightarrow Min & & & \\ u_i & \leq & 0, & i & = & 1, 2, \ldots, m \end{array} \tag{3.6}$$

Dualproblem in Matrizenschreibweise

$$\begin{array}{lcl} A^T \vec{u} & \geq & \vec{c} \\ \vec{b}^T \vec{u} & = & K \rightarrow Min \\ \vec{u} & \leq & 0 \end{array} \tag{3.7}$$

Jedes LP läßt sich auf diese Weise dualisieren. Die Beziehung ist symmetrisch, d.h. das duale Programm des Dualen ergibt wieder das Primale. Wichtig ist hier die praktische Bedeutung, denn die rechentechnischen Schwierigkeiten nehmen beim Simplex-Algorithmus stärker mit der Anzahl der Nebenbedingungen (Primal), als mit der Anzahl Variablen (Dual) zu [Tah71]. Aufgrund der Dualität kann die jeweils günstigste Darstellung gewählt werden.

Beispiel 3.3 Das Ausgangsbeispiel 3.1 (Primalproblem) lautet:

$$\begin{array}{lcl} 4x_1 + 3x_2 & \leq & 600 \\ 2x_1 + 2x_2 & \leq & 320 \\ 3x_1 + 7x_2 & \leq & 840 \\ & & \\ 2x_1 + 3x_2 & = & G \rightarrow Max \\ x_1, x_2 & \geq & 0 \end{array}$$

Das dazugehörige Dualproblem mit weniger Nebenbedingungen, aber mehr Variablen lautet:

$$\begin{array}{lcl} 4u_1 + 2u_2 + 3u_3 & \geq & 2 \\ 3u_1 + 2u_2 + 7u_3 & \geq & 3 \\ & & \\ 600u_1 + 320u_2 + 840u_3 & = & K \rightarrow Min \\ u_1, u_2, u_3 & \leq & 0 \end{array}$$

□

Die Dualität wird außerdem für Aussagen über die möglichen Lösungen eines LPs verwendet. Für ein Paar dualer Linearer Programme P und D gelten folgende Aussagen [Had74]:

1. Ein LP besitzt genau dann eine optimale Lösung, wenn sowohl P als auch D (mindestens) eine zulässige Lösung besitzen. Die optimalen Zielfunktionswerte von P und D stimmen dabei überein.
2. Ist P nicht nach unten beschränkt, ist D unlösbar.
3. Ist D nicht nach oben beschränkt, ist P unlösbar.

3.1.3 Theorie der ganzzahligen linearen Programmierung

Ein Sonderfall der linearen Programmierung ist die ganzzahlige lineare Programmierung, bei der alle auftretenden Variablen aus $\mathbb{Z}$ sind. Zwischen ILP und LP besteht der enge Zusammenhang, daß es zu jedem ganzzahligen linearen Problem ein Problem der linearen Programmierung gibt, das dieselbe Lösung besitzt.

$$\begin{array}{lcl} A\vec{x} & \leq & \vec{b} \\ \vec{c}^{\,T}\vec{x} & = & G \rightarrow Max \\ \vec{x} & \geq & 0 \end{array} \tag{3.8}$$

mit

$$\vec{x} \in \mathbb{Z}^n, \vec{b} \in \mathbb{Z}^m, \vec{c} \in \mathbb{Z}^n, A \in \mathbb{Z}^{m \times n}.$$

Durch die Beschränkung auf ganze Zahlen wird die Bestimmung einer optimalen Lösung im allgemeinen schwieriger; das Problem der ganzzahligen linearen Programmierung ist $\mathcal{NP}$-vollständig [MRG79]. Einer der Gründe ist, daß eine gefundene optimale reelle Lösung nicht einfach durch Rundung auf den nächsten ganzzahligen Wert gefunden werden kann, denn dadurch können sich suboptimale oder sogar ungültige Lösungen ergeben. Zur Lösung dieser Probleme in endlich vielen Iterationsschritten werden hier insbesondere *Branch-and-Bound*-Methoden [GLN88] eingesetzt.

3.1.4 Branch-and-Bound

Der Simplex-Algorithmus ist eine Lösungsstrategie, mit der eine optimale Lösung eines LP bestimmt werden kann, zur Ermittlung einer optimalen Lösung für ILP können Branch-and-Bound Methoden [GLN88] verwendet werden.

Das Prinzip der Branch-and-Bound Methode besteht darin, die Menge aller zulässigen Lösungen in disjunkte *Teilmengen* aufzuspalten (*Branching*) und für den Zielfunktionswert in jeder Teilmenge im Fall der Minimierung eine untere, im Fall der Maximierung eine obere *Schranke* anzugeben (*Bounding*). Falls für eine Teilmenge der Wert dieser Schranke größer bzw. kleiner, d.h. schlechter ist als der Zielfunktionswert einer bereits gefundenen Lösung, braucht die betreffende Teilmenge nicht weiter untersucht werden, da dann auch alle Lösungen dieser Teilmenge im Sinne des Optimierungsziels schlechter sind als die schon bekannte Vergleichslösung [WD92].

Für die Effektivität dieser Methode ist es wichtig, daß eine gute *Vergleichslösung* möglichst früh vorliegt, und daß die Schranken den Optimalwerten für die jeweiligen Teilmengen möglichst nahekommen. Eine Vergleichslösung kann zum Beispiel durch ein heuristisches Verfahren gewonnen werden. Die Schranken können durch eine *Relaxation* der Teilprobleme ermittelt werden. Bei einer Relaxation werden gewisse Bedingungen, zum Beispiel die Ganzzahligkeitsforderung bei ILP, weggelassen. Dadurch entstehen Teilprobleme, die effizient lösbar sind und deren Optimalwerte als Schranken benutzt werden können, da sie mit Sicherheit nicht schlechter sondern mindestens gleich gut sind, wie die Optimalwerte der nicht relaxierten Teilprobleme [WD92].

Zur Veranschaulichung wird im folgenden ein ganzzahliges lineares Problem mittels Branch-and-Bound gelöst. Zur Darstellung wird ein *Entscheidungsbaum* verwendet. Als Relaxation wird auf die Ganzzahligkeitsbedingung verzichtet, d.h. für Teilprobleme können Variablen Werte aus dem Reellen annehmen. Damit entstehen LPs, deren Lösung eine obere Schranke (Maximierungsproblem) für den Zielfunktionswert des entsprechenden ILPs darstellt. Zur Lösung der entsprechenden LPs kann wieder der Simplex-Algorithmus verwendet werden.

Definition 3.3 ***Entscheidungsbaum***

Gegeben sei ein knoten- und kantenmarkierter Entscheidungsbaum $E_B(E, Z)$ mit E, den Entscheidungsvariablen des ILP und Z, dem Zielfunktionswert. Die Knoten von E_B sind mit den Werten der Entscheidungsvariablen und des Zielwertes markiert. Seine Wurzel wird mit der Vergleichslösung markiert, in der alle Variablen relaxiert sind. Die Kanten von E_B entstehen durch die Aufteilung in disjunkte Teilmengen einer Variablen und werden mit diesen zusätzlichen Bedingungen markiert.

Beispiel 3.4 Es sei das folgende ganzzahlige lineare Problem ($x_1,\ x_2,\ x_3 \in \mathbb{Z}$) gegeben:

$$\begin{array}{rcrcrcl} x_1 & + & 3x_2 & + & x_3 & \leq & 25 \\ 2x_1 & + & 2x_2 & + & 5x_3 & \leq & 27 \\ 8x_1 & + & 12x_2 & + & 10x_3 & = & G \rightarrow Max \end{array}$$

$$x_1,\ x_2,\ x_3 \geq 0$$

Der Entscheidungsbaum zu diesem Beispiel ist in in Abbildung 3.2 dargestellt. Die Numerierung der Knoten gibt die Reihenfolge an, in der die Linearen Programme gelöst werden. Eine Vergleichslösung wird in Knoten 1 gefunden und damit eine obere Schranke für das gegebene Problem. Das Teilproblem 7 liefert hier die erste ganzzahlige Vergleichslösung, die sich nach der Lösung des Teilproblems 8 als optimal herausstellt, da der gefundene Zielfunktionswert $G = 128$ nicht schlechter ist, als die Schranken der Teilprobleme 2, 6 und 8 und das Teilproblem 5 keine zulässigen Lösungen aufweist. Eine weitere Aufspaltung von Teilproblem 2 nach der Variablen x_2 und von 6 und 8 nach der Variablen x_3 ist somit nicht mehr nötig.

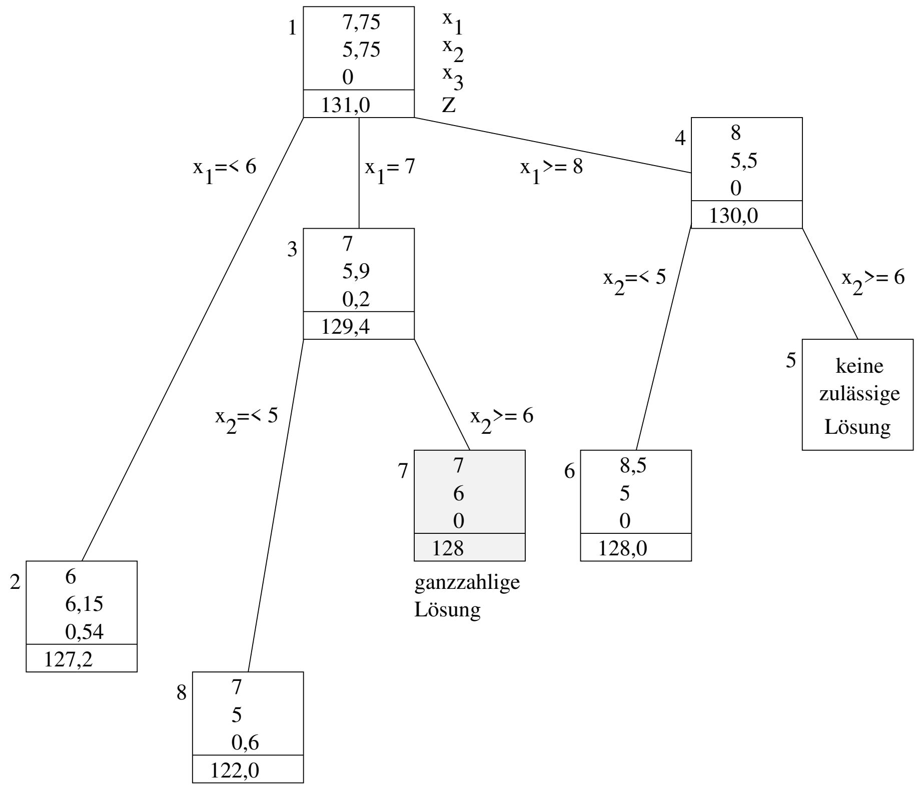


Abbildung 3.2: Branch-an-Bound Entscheidungsbaum zu Beispiel 3.4

□

Die Aufspaltung der Menge der zulässigen Lösungen wird in Beispiel 3.4 dadurch erreicht, daß der Wertebereich der einzelnen Variablen x_i in mindestens zwei disjunkte Intervalle aufgeteilt wird. Diese Teilung wird so vorgenommen, daß die nichtganzzahligen Lösungen der Teilprobleme schrittweise ausgeschlossen werden. Der Entscheidungsbaum in Abbildung 3.2 zeigt an seinen Kanten diese Zusatzbedingungen und in den Knoten die Werte der Variablen, die sich als Lösung des LPs mit der Folge der Zusatzbedingungen ergeben. Außerdem sind die Knoten mit dem Zielfunktionswert, als Schranke für alle ganzzahligen Lösungen des entsprechenden Zweiges, markiert. Für die Reihenfolge der Behandlung der einzelnen Variablen und der Intervallaufteilung werden heuristische Regeln verwendet. Beim vorliegenden Lösungsweg wurde die gegebene Reihenfolge der Variablen beibehalten und bei der Bestimmung der Intervalle angenommen, daß die ganzzahligen Lösungswerte in der Nähe der nichtganzzahligen Lösung liegen.

Die Branch-and-Bound Methoden lassen sich in *vollständige* Verfahren (hier wird im Laufe des Algorithmus der komplette Entscheidungsbaum erzeugt) und *heuristische* Verfahren (es werden nur bestimmte Verzweigungen weiterverfolgt) unterteilen. Während vollständige Branch-and-Bound Methoden optimale Lösungen bei exponentieller Laufzeit finden, werden mit heuristischen Methoden in der Regel nur suboptimale Ergebnisse erzielt. Dabei können jedoch durch günstige Wahl der zugrundeliegenden Heuristiken polynomielle Laufzeiten erreicht werden [DL80].

Wie eine vollständige Branch-and-Bound Methode eine optimale Lösung garantieren kann, ist in Beispiel 3.4 demonstriert. Eine obere Schranke für das Maximierungsproblem stellt die Vergleichslösung (Knoten 1) dar. Die Folgeknoten können nur kleinere obere Schranken besitzen. Die oberen Schranken aller Folgeknoten werden berechnet, d.h. der Entscheidungsbaum wird mit einer Breitensuche (*breadth first search*) durchlaufen. Wenn eine erste ILP-Lösung gefunden wird (Knoten 7) und alle weiteren Knoten auf der gleichen Baumebene keine kleinere obere Schranke (Knoten 2, 6 und 8) oder keine zulässige Lösung (Knoten 5) besitzen, ist diese ILP-Lösung eine optimale Lösung.

Für die Berechnung einer optimalen Lösung der SILP-Programme verwendet der CPLEX-Solver eine vollständige Branch-and-Bound Methode. Aus diesem Grund wird hierbei ein kompletter Entscheidungsbaum aufgebaut, der einen sehr großen Speicherplatz benötigt, da alle Folgeknoten gespeichert werden müssen. Dies kann bei größeren Benchmarks zu dem Problem führen, daß eine Berechnung aufgrund von Arbeitsspeichermangel abbricht[III].

3.2 Modelle zu IS und ISRA

In [Käs97] sind zwei Modelle zu IS und ISRA untersucht worden: SILP und OASIC. Die Modelle sind im Rahmen der Architektursynthese entwickelt worden. Das Ziel der Architektursynthese besteht darin, für eine gegebene Codesequenz entweder bezüglich einer bestimmten Kostengrenze die schnellste Architektur, oder bezüglich einer vorgegebenen Leistungsgrenze die kostengünstigste Architektur zu realisieren [CHG92]. Diese Problemstellung hat einen engen Bezug zur Instruktionsanordnung. In der Architektursynthese ist die Codesequenz gegeben, und durch die Auswahl funktionaler Einheiten kann eine kostengünstige Architektur entwickelt werden. In der Codegenerierung ist die Hardware vorgegeben und durch geschickte Auswahl und Anordnung der Instruktionen wird versucht eine hohe Codequalität zu erreichen.

3.2.1 SILP

In diesem Abschnitt wird eine Basisformulierung des SILP-Modells angegeben. Eine genauere Beschreibung kann [Zha96] entnommen werden. Die genauen Anpassungen des Modells an den ADSP-2106x werden hier nicht behandelt, sie sind in [Käs97] ausführlich beschrieben. Mit dem SILP-Modell wird entweder die reine Instruktionsanordnung (IS) oder eine um Ressourcenschranken erweiterte IS beschrieben (ISRA). Mit dem Begriff Ressourcenschranken wird eine Reduktion des Registerallokationsproblems auf ein Registerverteilungsproblem bezeichnet. Eine ausführliche Beschreibung, welche Aspekte der Registerallokation durch ISRA betrachtet werden, findet sich in [Käs97].

Mit dem Begriff Ressourcentyp k werden in SILP funktionale Einheiten, wie zum Beispiel ALU und Multiplizierer, benannt. Eine Ressource ist eine Instanz dieses

[III] siehe Kapitel 3.3.1, Seite 45

Typen, also zum Beispiel eine ALU. Da von allen funktionalen Einheiten nur eine Instanz auf dem ADSP-2106x vorhanden ist, kann das Modell später entsprechend vereinfacht werden.

Grundlage des SILP-Modells ist der Abhängigkeitsgraph $G_D = (V_D^k, E_D^k)$. V_D^k ist die Menge aller Knoten des Abhängigkeitsgraphen, die zu Instruktionen gehören, welche auf Ressourcentyp k ausgeführt werden können. Der Abhängigkeitsgraph dient als Ausgangspunkt des sogenannten *Ressourcenflußgraphen* G_F. Er beschreibt die Programmausführung als einen Fluß der vorhandenen Hardwareressourcen durch die Instruktionen des Programmes. Der Ressourcenflußgraph wird algebraisch durch ein Gleichungssystem dargestellt, in dem alle Bedingungen (Kontrollfluß-, Datenabhängigkeiten) durch ein System von Ungleichungen ausgedrückt werden. Dieses Gleichungssystem ist das Ziel dieses Modells, damit können die genannten Aufgaben als ganzzahliges lineares Problem (SILP-Programm) beschrieben werden.

Zunächst werden einige wichtige Größen und Bezeichnungen aufgelistet, um einen Überblick über die verwendete Terminologie zu geben:

- Die Variable t_i gibt die relative Position einer Mikrooperation[IV] i innerhalb der Instruktionen der optimierten Codesequenz an.
- Die Variable M_{Steps} beschreibt das Maximum an nötigen Kontrollschritten. Der Wert entspricht der Anzahl der benötigten Instruktionszyklen und ist Ziel der Minimierung.
- Die Variable w_j beschreibt die Ausführungszeit der Instruktion $j \in V_D$.
- Die Anzahl der zur Verfügung stehenden Ressourcen vom Typ $k \in V_K$ beträgt R_k (für den ADSP-2106x gilt: $R_k = 1, \forall k$).
- Ein Ressourcentyp $k \in V_K$ wird durch zwei Knoten k_Q und k_S in G_F repräsentiert, wobei k_Q als Quelle und k_S als Senke in dem zu definierenden Flußnetzwerk fungiert.

Definition 3.4 ***Ressourcenflußgraph***

Der Ressourcenflußgraph ist ein gerichteter Graph $G_F = (V_F, E_F)$ mit

$$V_F = \bigcup_{k \in V_K} V_F^k \quad \textit{und} \quad E_F = \bigcup_{k \in V_K} E_F^k$$

Dabei ist

$$V_F^k = V_D^k \cup \{k_Q, k_S\} = \{u \in V_D \mid (u,k) \in E_R\} \cup \{k_Q, k_S\}$$

und

[IV] siehe Kapitel 2.1.2, Seite 12

$$
\begin{aligned}
E_F^k &= \{(i,j) \mid i,j \in V_D^k \wedge j \neg\prec i \wedge i \neq j\} \\
&\cup \{(k_Q, j) \mid (k,j) \in E_R\} \\
&\cup \{(j, k_S) \mid (k,j) \in E_R\}
\end{aligned}
$$

mit einer Vorrangrelation $\prec \subset V_D \times V_D$

$$i \prec j \Leftrightarrow i \stackrel{*}{\longrightarrow}_{G_D}{}^{\mathrm{V}} j$$

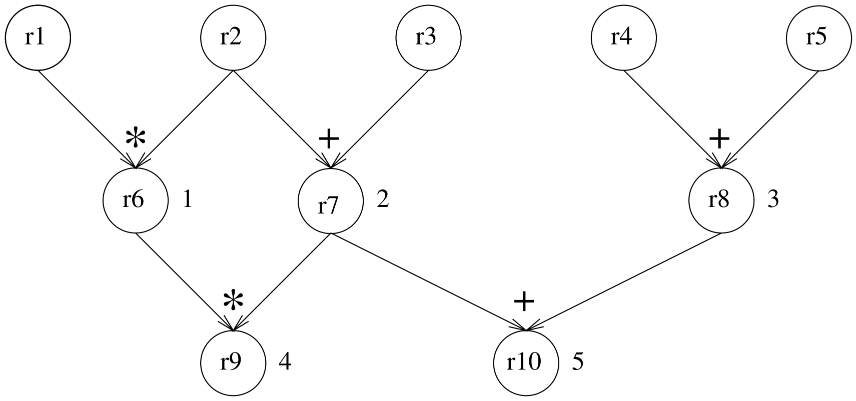
Es gilt $i \prec j$ *genau dann, wenn Operation* i *direkt oder indirekt von Operation* j *abhängt.*

Jeder Kante $(i,j) \in E_F^k$ *wird eine Flußvariable* $x_{ij}^k \in \{0,1\}$ *zugeordnet. Eine Hardwarekomponente* k *wird genau dann über die Kante* (i,j) *von Knoten* i *nach Knoten* j *befördert, wenn* $x_{ij}^k = 1$ *gilt [Käs97].*

Jede Kante $(a,b) \in E_F^k$ bezeichnet einen möglichen Fluß der Ressourcen des Typen $k \in V_k$ von a nach b. Sind a und b beides Knoten des Abhängigkeitsgraphen, dann wird eine Kante (a,b) nur in die Menge E_F^k aufgenommen, wenn sie nicht gegen eine Vorrangrelation $a \prec b$ verstößt.

Beispiel 3.5 In Abbildung 3.3 ist a) ein Programmbeispiel mit fünf Instruktionen und b) der korrespondierende Datenflußgraph dargestellt.

```
1: r6  = r1 * r2;
2: r7  = r2 + r3;
3: r8  = r4 + r5;
4: r9  = r6 * r7;
5: r10 = r7 + r8;
```



a) Programm b) Datenflußgraph

Abbildung 3.3: Programm und Datenflußgraph zu Beispiel 3.5

Abbildung 3.4 zeigt den dazugehörigen Ressourcenflußgraphen. Mit *Flußnetzwerk* wird der Teil des Ressourcenflußgraphen bezeichnet, der zu einem Ressourcentypen gehört. In diesem Beispiel werden nur die Ressourcen ALU und Multiplizierer verwendet und entsprechend zwei Flußnetzwerke dargestellt. Die Ressourcenknoten A_q

[V] $\stackrel{*}{\longrightarrow}_{G_D}$ bedeutet transitiver Abschluß über den Graphen G_D

und A_s bzw. M_q und M_s bezeichnen Quellen und Senken für die Ressourcen ALU bzw. Multiplizierer. Die Instruktionen des Beispielprogramms sind in Abhängigkeit ihrer Operationen als Knoten in die Flußnetzwerke eingetragen. Die Pfeile zwischen ihnen symbolisieren eine mögliche Ausführungsreihenfolge. Insgesamt werden so alle möglichen Wege durch den Ressourcenflußgraphen abgebildet. In einer Lösung, das heißt einer Anordnung dieser Instruktionen in einer Codesequenz, müssen alle Daten- und Kontrollflußabhängigkeiten gewahrt bleiben, und es ergibt sich ein eindeutiger Pfad durch jedes Flußnetzwerk.

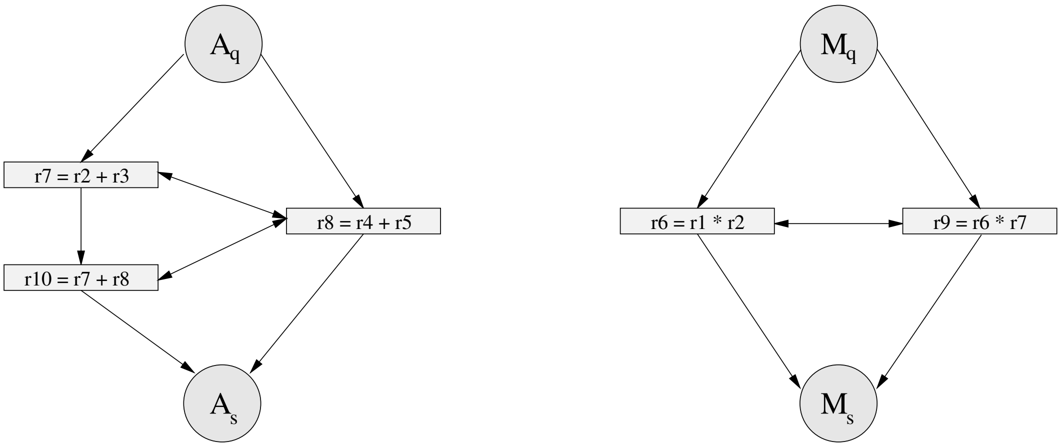


Abbildung 3.4: Der Ressourcenflußgraph zu Beispiel 3.5

Zur Illustration ist in Abbildung 3.5 ein möglicher Fluß durch den Ressourcenflußgraphen dargestellt. Bei einem Fluß durch das Flußnetzwerk für den Multiplizierer „fließt" eine Ressource Multiplizierer und führt an den Knoten erst die Instruktion 1 und dann 4 aus.

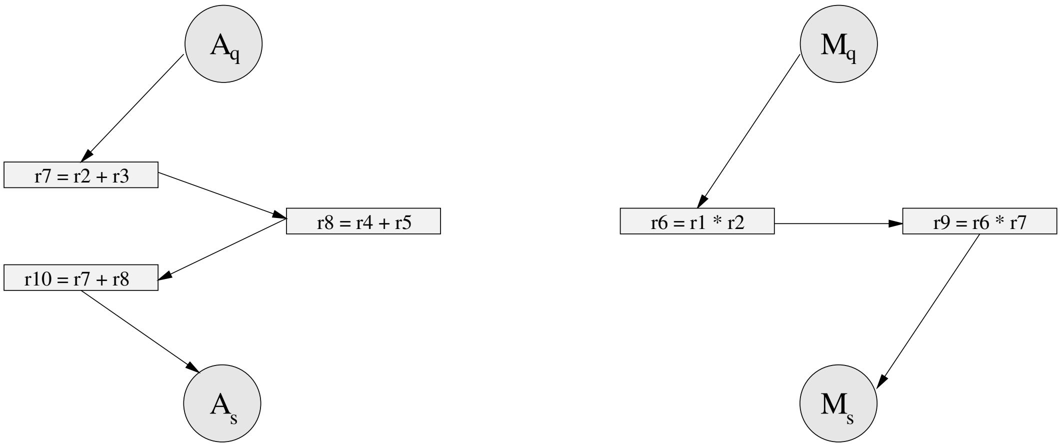


Abbildung 3.5: Ein möglicher Weg durch den Ressourcenflußgraphen zu Beispiel 3.5

Die Kontrollfluß- und Datenabhängigkeiten werden durch Nebenbedingungen des LP ausgedrückt. Eine Datenabhängigkeit besteht zum Beispiel zwischen Instruktion 3 und 5, dies wird durch eine Vorrangbedingung ausgedrückt. Außerdem verfügt der

ADSP-2106x über nur eine Multipliziereinheit, also muß durch Ressourcenschranken die gleichzeitige Ausführung zweier Multiplikationen verboten werden.

□

Im folgenden wird zunächst eine Zielfunktion angegeben und danach einige Nebenbedingungen formuliert, mit denen der Ressourcenflußgraph und die Daten- und Kontrollflußabhängigkeiten in algebraischen Gleichungen ausgedrückt werden. Das SILP-Modell wird dadurch nicht vollständig wiedergegeben, jedoch soll durch diese Übersicht nur eine Vorstellung der Modellierung mit SILP vermittelt werden.

Zielfunktion

Die Anzahl S der benötigten Kontrollschritte soll minimiert werden.

$$S = \min M_{steps} \tag{3.9}$$

Flußerhaltung

Die Flußerhaltung formuliert Bedingungen, mit denen ein korrekter Fluß durch die Instruktionen erreicht wird. Dieser Fluß wäre unter anderem nicht korrekt, wenn mehr als eine Instruktion zur gleichen Zeit auf einer Ressource ausgeführt wird. Den Fluß, der in einen Knoten $j \in V_D$ eintritt stellt Φ_j^k dar, und Ψ_j^k den Fluß, der den Knoten j verläßt (Gleichung 3.10). Der einen Knoten betretende positive Nettofluß muß denselben Wert haben wie der den Knoten verlassende positive Nettofluß. Diese Flußerhaltung wird durch Gleichung 3.11 ausgedrückt. Eine weitere Bedingung dafür, daß jede Operation genau einmal auf einer Ressource ausgeführt werden darf (Ausführungsbeschränkung), wird durch Gleichung 3.12 ausgedrückt.

$$\Phi_j^k = \sum_{(j,k)\in E_R^k} x_{ij}^k \;; \qquad \Psi_j^k = \sum_{(j,k)\in E_R^k} x_{ji}^k \tag{3.10}$$

$$\Phi_j^k - \Psi_j^k = 0 \quad \forall\, j \in V_D,\ \forall\, k \in V_k \;:\; (j,k) \in E_R \tag{3.11}$$

$$\sum_{k\in V_K:(j,k)\in E_R} \Phi_j^k = 1 \quad \forall\, j \in V_D \tag{3.12}$$

Beispiel 3.6 Aus dem in Beispiel 3.5 angegebenen Programm werden ausschnittsweise die Multiplikationen (Instruktion 1 und 4) betrachtet. Die Indizes für zum Beispiel x_{M1}^M bedeuten, daß hiermit eine Kante des Ressourcenflußgraphen für den Ressourcentypen Multiplizierer M von Quelle M nach Instruktion 1 abgebildet wird. Entsprechend bedeutet x_{41}^M eine Kante zwischen Instruktion 4 und 1 für den Ressourcentypen Multiplizierer. Zur Modellierung der Flußerhaltung (Gleichung 3.11) ergeben sich folgende Gleichungen:

$$x^M_{M1} + x^M_{41} - x^M_{14} - x^M_{1M} = 0$$

$$x^M_{M4} + x^M_{14} - x^M_{41} - x^M_{4M} = 0$$

Die Gleichungen zur Realisierung der Ausführungsbeschränkung (Gleichung 3.12) lauten wie folgt:

$$x^M_{M1} + x^M_{41} = 1$$

$$x^M_{M4} + x^M_{14} = 1$$

□

Vorrangbedingung

Des weiteren müssen die durch den Abhängigkeitsgraphen gegebenen Vorrangbeziehungen modelliert werden. Die zu erstellenden Ungleichungen hängen dabei vom Typen der Abhängigkeiten ab. Besteht zwischen den Instruktionen i und j eine Setzungs-Setzungs-Abhängigkeit oder eine Setzungs-Benutzungs-Abhängigkeit[VI], ist die Ungleichung

$$t_j - t_i \geq w_i \tag{3.13}$$

zu erzeugen. Liegt hingegen eine Benutzungs-Setzungs-Abhängigkeit vor, so genügt es, zu gewährleisten, daß Instruktion j niemals vor Instruktion i ausgeführt wird.

$$t_j - t_i \geq 1 \tag{3.14}$$

Beispiel 3.7 Die fünf Instruktionen aus Beispielprogramm 3.5 werden auf die Variablen t_1 bis t_5 abgebildet. Es ergeben sich zur Darstellung der Vorrangbedingungen für die Benutzungs-Setzungs-Abhängigkeit die folgenden Ungleichungen:

$$\begin{array}{rcl} t_4 - t_1 & \geq & 1 \\ t_4 - t_2 & \geq & 1 \\ t_5 - t_2 & \geq & 1 \\ t_5 - t_3 & \geq & 1 \end{array}$$

□

Weitere Nebenbedingungen sind zum Beispiel die *Ressourcenschranken*, wobei die Anzahl der Instanzen eines Ressourcentypen nicht überschritten werden darf, und die *Lebenszeit*, mit der die Lebensspanne[VII] eines Wertes in einem Register beschrieben wird.

[VI]siehe Kapitel 2.4, Seite 18
[VII]siehe Kapitel 2.14, Seite 24

3.2.2 OASIC

Das zweite in [Käs97] untersuchte Modell ist OASIC [CHG92]. In OASIC werden die Bedingungen und Abhängigkeiten ähnlich wie in dem SILP-Modell durch ein System von Ungleichungen beschrieben. Die Formulierung durch OASIC weist einige Nachteile gegenüber der Formulierung in SILP auf. Ein erster Nachteil zeigt sich in den Größen der erzeugten Dateien. In den Tabellen 3.1 und 3.2 sind einige ausgewählte Benchmarks und die Größe der von CIS erzeugten Problemdateien gegenübergestellt. In der rechten Spalte ist das auf ganze Zahlen aufgerundete Verhältnis der Dateigrößen aufgeführt.

Benchmark	SILP [Byte]	OASIC [Byte]	SILP : OASIC
IIR	4250	206135	1 : 49
Whetstone	15625	162385	1 : 10
histo	30372	174288	1 : 59

Tabelle 3.1: Dateigrößen für IS mit Benchmarks der Gruppe A

Benchmark	SILP [Byte]	OASIC [Byte]	SILP : OASIC
FIR	60357	2835883	1 : 47
IIR	91515	7154872	1 : 78
DFT	301662	46341471	1 : 154

Tabelle 3.2: Dateigrößen für ISRA mit Benchmarks der Gruppe A

An dem Beispiel des Benchmark *DFT* mit einer Dateigröße von über 46 MB wird das Problem bei größeren Programmen deutlich. Die von CIS erzeugten Dateien werden so groß, daß die Bearbeitung mit CIS aufgrund von Speichermangel abbricht. Aufgrund dieses Nachteils ist es nicht möglich gewesen, gerade die interessanten, großen LP für das OASIC-Modell zu erzeugen.

Der entscheidende Grund für die Wahl von SILP ist jedoch das bessere Laufzeitverhalten. Ziel dieser Diplomarbeit ist ein Vergleich der beiden Verfahren ILP und CLP. Damit bietet sich als Vergleichsbasis das schnellere der beiden untersuchten Modelle an. Eine auszugsweise Gegenüberstellung der in [Käs97] aufgeführten Berechnungszeiten belegt die wesentlich größeren Laufzeiten für OASIC gegenüber SILP in den Tabellen 3.3 und 3.4. In den rechten Spalten der Tabellen sind die aufgerundeten Verhältnisse der Berechnungszeiten aufgeführt.

Benchmark	SILP	OASIC	SILP	:	OASIC
IIR	0,19s	5,45s	1	:	24
DFT	47,42s	12600,00s	1	:	266
Whetstone	7440,00s	1110,00s	7	:	1

Tabelle 3.3: Laufzeitergebnisse für IS mit Benchmarks der Gruppe A

Benchmark	SILP	OASIC	SILP : OASIC
FIR	1,69s	251,00s	1 : 149
DFT	>24h	>24h	-

Tabelle 3.4: Laufzeitergebnisse für ISRA mit Benchmarks der Gruppe A

Aufgrund dieser Betrachtungen wird für den Vergleich der beiden Verfahren ILP und CLP das SILP-Modell zugrundegelegt.

3.3 SILP Referenzdaten

Für den Vergleich der Berechnungszeiten von ILP und CLP werden in diesem Abschnitt die Referenzdaten für das SILP-Modell aufgeführt. Nachdem in Abschnitt 3.2.1 eine Beschreibung des Modells vorgestellt wurde, werden in diesem Abschnitt möglichst effiziente Lösungsstrategien untersucht. Es gibt zwei Möglichkeiten, die Berechnungszeiten der Lösungssuche in ILP zu verbessern:

1. **Lösungsstrategien in CPLEX**

   Eine Möglichkeit liegt in der Wahl effizienter Lösungsstrategien in CPLEX [ILO98]. Eine Lösungsstrategie wird über eine Reihe von Parametern in CPLEX eingestellt. Es wird zunächst eine Parametereinstellung gesucht, mit der eine möglichst effiziente Lösungssuche gelingt. Zu diesem Zweck wird eine Teilmenge der Benchmarks untersucht, und eine Lösungsstrategie entwickelt, die dann zur Generierung der Referenzdaten für alle Benchmarks verwendet wird.

2. **Approximative Lösungsstrategien für SILP**

   Die Berechnung einer optimalen Anordnung der Instruktionen ist mit einem hohen Berechnungsaufwand verbunden. Eine weitere Möglichkeit zur Reduzierung dieses Rechenaufwands bietet sich durch den Verzicht auf die Garantie der Optimalität einer Lösung. Auf diese Weise konnten in [Käs97] folgende approximative Lösungsstrategien entwickelt werden, die ein wesentlich besseres Laufzeitverhalten aufweisen:

   - Approximation durch Rundung
   - schrittweise Approximation
   - isolierte Flußanalyse
   - *Schrittweise Approximation der Isolierten Flußanalyse* (*SAIF*)

Zur Umsetzung der approximativen Lösungsstrategien werden Relaxationen[VIII] verwendet. Es werden zunächst Teillösungen berechnet, in denen eine Reihe der Variablen reelle Werte annehmen können. Aus den Teillösungen wird dann schrittweise

[VIII] siehe Kapitel 3.1.4, Seite 35

eine ganzzahlige Lösung für das ILP berechnet. Von den genannten Lösungsstrategien zeigt die schrittweise Approximation der isolierten Flußanalyse sowohl das beste Laufzeitverhalten als auch die geringsten Abweichungen vom Optimum der Instruktionsanzahl [Käs97]. Aus diesem Grund wird diese Lösungsstrategie für den Vergleich mit den approximativen Lösungsstrategien in CLP verwendet.

In Abschnitt 3.3.1 wird zunächst eine effiziente Lösungsstrategie in CPLEX (Möglichkeit 1) durch Testreihen mit Benchmarks der Gruppe A für eine optimale Anordnung der Instruktionen ermittelt. Mit dieser Lösungsstrategie werden in Abschnitt 3.3.2 die Laufzeiten für die Berechnungen der optimalen Anordnungen der Gruppe B durchgeführt. Abschnitt 3.3.3 führt die Ergebnisse für Möglichkeit 2, die suboptimale Lösung der Optimierungsprobleme, auf.

3.3.1 Berechnungen mit verschiedenen Parametereinstellungen

Eine Reihe der Parameter wird von CPLEX automatisch an das zu lösende Problem angepaßt, für die meisten anderen Parameter sind Default-Einstellungen aufgrund von Erfahrungswerten vorgegeben. Es finden sich dennoch einige Parameter, die mit benutzerdefinierten Einstellungen entweder zu kürzeren Berechnungszeiten oder zu einem geringeren Arbeitsspeicherbedarf führen. Ohne diese Einstellungen wächst der Arbeitsspeicherverbrauch bei einigen Berechnungen auf mehrere hundert Megabyte an. Bei Testreihen mit einer Teilmenge der Benchmarks hat sich gezeigt, daß für die hier betrachteten großen Optimierungsprobleme zwei Parameter besonders großen Einfluß haben:

- ***presolve: off***

  Mit dieser Einstellung wird standardmäßig von CPLEX eine nicht genauer spezifizierte Reduktion des Problems berechnet. Für die hier betrachteten Probleme hat dies allerdings im allgemeinen zu einer wesentlich längeren Laufzeit geführt, deshalb wird presolve deaktiviert.

- ***strong branching: on***

  Durch diese Einstellung wird beim Branch-and-Bound Algorithmus in CPLEX die Auswahl der nächsten Variablen gesteuert. Durch die Aktivierung von strong branching wird für alle Folgeknoten über eine Heuristik eine Kostenabschätzung berechnet und der kostengünstigste Folgeknoten ausgewählt. Zwar wird auf diese Weise die Berechnungszeit an den einzelnen Knoten länger, jedoch ist der dabei entstehende Entscheidungsbaum, und damit der Arbeitsspeicherbedarf, deutlich geringer.

Es gibt eine Fülle weiterer Einstellungsmöglichkeiten, allerdings haben sich bei den Testreihen damit keine wesentlichen Reduzierungen der Laufzeiten erreichen lassen.

Verwendete Computer:

Die Berechnungen von [Käs97] an der Universität Saarbrücken wurden auf einem Compute Server Ultra 2/2x200 mit 1024 MB Hauptspeicher unter dem Betriebssystem Solaris 2.5.1 mit CPLEX in der Version 4.0 ausgeführt. Die Berechnungszeiten dieser Diplomarbeit an der Universität Dortmund wurden auf einem PC mit einem AMD K6/2 300 MHz Prozessor mit 64 MB Hauptspeicher unter dem Betriebssystem Linux 2.0.36 mit der Version 6.0 von CPLEX ermittelt. Zur Vereinfachung der Notation werden die Städtenamen Saarbrücken und Dortmund als Synonym für die Diplomarbeiten an den entsprechenden Universitäten verwendet.

In den Tabellen 3.5 und 3.6 sind die Ergebnisse der Berechnungen der Benchmarks der Gruppe A durch CPLEX mit verschiedenen Parametereinstellungen aufgeführt. In den jeweiligen Spalten stehen die Testergebnisse mit den Default-Einstellungen von CPLEX und mit den oben beschriebenen Parametereinstellungen für presolve und strong branching. Eine weitere Möglichkeit liegt in der Kombination dieser beiden Einstellungen (rechte Spalte). Einige Berechnungen konnten wegen Arbeitsspeichermangels nicht zu Ende geführt werden, da der von CPLEX aufgebaute Entscheidungsbaum mit der Größe des betrachteten Problems sehr stark anwächst. Bei den in Klammern aufgenommenen Zeitwerten ist die Bearbeitung wegen Speicherplatzmangels trotz einer Vergrößerung des Swap-Space auf mehr als 600 Megabyte abgebrochen worden.

Benchmark	Default	presolve: off (I)	strong branching: on (II)	Kombination: I + II
FIR	0,03s	0,11s	0,03s	0,06s
IIR	0,05s	0,05s	0,10s	0,14s
DFT	273,29s	114,59s	144,92s	164,57s
Whetstone	1005,37s	802,84s	3659,43s	1197,46s
histo	(5816,69s)	(6309,98s)	(54959,44s)	48509,17s
conv	2888,42s	2857,86s	4101,93s	3770,11s

Tabelle 3.5: CPLEX Ergebnisse mit verschiedenen Parametereinstellungen für IS mit Benchmarks der Gruppe A

Benchmark	Default	presolve: off (I)	strong branching: on (II)	Kombination: I + II
FIR	0,23s	15,40s	1,04s	118,07s
IIR	16208,31s	20907,27s	>24h	11924,12s
DFT	(22682,11s)	(24683,71s)	>24h	64063,91s
Whetstone	>24h	>24h	>24h	>24h
histo	>24h	>24h	>24h	>24h
conv	>24h	>24h	>24h	>24h

Tabelle 3.6: CPLEX Ergebnisse mit verschiedenen Parametereinstellungen für ISRA mit Benchmarks der Gruppe A

Die Auswertung der obigen Tabellen ergibt, daß mit der Kombination der beiden Parameter (I + II) zum einen für die größte Anzahl an Benchmarks eine Berechnung innerhalb von 24h möglich ist. Zum anderen sind bei größeren Benchmarks (mehr als 20 Instruktionen) die Berechnungszeiten kürzer. Diese Parametereinstellung wird, um einen besseren Vergleich zu ermöglichen, für die weiteren Berechnungen verwendet, obwohl sich vereinzelt mit anderen Einstellungen wesentlich kürzere Laufzeiten ergeben. Die Ergebnisse mit der Kombination der beiden Parametereinstellungen sind damit gleichzeitig die Referenzdaten des SILP-Modells für eine optimale Anordnung der Instruktionen der Benchmarks der Gruppe A.

3.3.2 Ergebnisse für optimale Lösungen

In Tabelle 3.7 stehen die Ergebnisse der Berechnungen einer optimalen Anordnung der Instruktionen für die Beispielprogramme des erweiterten Benchmarking (Gruppe B). Bei diesen Benchmarks werden weitere und teilweise wesentlich größere Beispielprogramme (mit über 80 Instruktionen) optimiert, daher konnten für die meisten Benchmarks keine Lösungen innerhalb von 24h berechnet werden. Von den ersten beiden Benchmarks wird für IS eine Lösung in ca. 5,5h bzw. 11h berechnet, für ISRA ist nur die Lösung von biquad_o innerhalb von 24h möglich; das Ergebnis für biquad_o liegt dabei nach fast 12h vor.

Benchmark	IS	ISRA
biquad_o	20114,70s	42953,22s
complex_mul	40911,70s	>24h
lattice	>24h	>24h
n_comple	>24h	>24h

Tabelle 3.7: CPLEX Ergebnisse für Benchmarks der Gruppe B

3.3.3 Ergebnisse für suboptimale Lösungen

Bei der Berechnung einer optimalen Lösung wird zunächst das Beispielprogramm von CIS bearbeitet und eine Datei generiert, die in CPLEX eingelesen wird. Daraufhin kann der Optimierungsprozeß in CPLEX gestartet werden. Auf diese Weise sind die Laufzeiten für die optimalen Lösungen auf einem PC in Dortmund ermittelt worden und können damit direkt mit den Ergebnissen der Berechnungen mit ECLiPSe verglichen werden.

Für die Berechnung der approximativen Lösungsstrategien ist eine direkte Kommunikation zwischen CIS und CPLEX erforderlich. Über ein Kommunikationsprotokoll wird von CIS ausgehend eine Relaxation mit CPLEX berechnet und dieses Ergebnis als Grundlage für weitere Bearbeitungsschritte an CIS übergeben. Als Endergebnis wird auf diesem Weg eine ganzzahlige Lösung für das ILP berechnet. Die erforderliche Kommunikation zwischen CIS und CPLEX konnte in Dortmund nicht initiiert werden. Da aus diesem Grund keine Referenzdaten auf dem PC in Dortmund ermittelt werden konnten, werden für den Vergleich mit den approximativen Lösungsstrategien in CLP die Berechnungszeiten von SAIF aus [Käs97] verwendet.

Mögliche Gründe für das Fehlschlagen der Kommunikation liegen in den unterschiedlichen Betriebssystemen, mit ihren verschiedenen Laufzeitbibliotheken, und dem Versionswechsel von CPLEX. Es ist zu vermuten, daß die Schnittstellenbeschreibung von CPLEX von dem Versionswechsel betroffen ist. Dieses Kommunikationproblem konnte auch in direkter Zusammenarbeit mit Daniel Kästner nicht behoben werden.

Um dennoch eine Abschätzung für die Vergleichbarkeit der unterschiedlichen Berechnungen zu geben, sind in in den folgenden Tabellen 3.8 und 3.9 die Ergebnisse für die optimale Anordnung der Instruktionen aus Dortmund den Berechnungszeiten aus Saarbrücken gegenübergestellt. Die Berechnungszeiten sind teilweise in Dortmund und teilweise in Saarbrücken kürzer. Neben der unterschiedlichen Hardware (PC und UNIX-Workstation) spielt hierbei möglicherweise ein weiteres Mal der Versionswechsel von CPLEX 4.0 (Saarbrücken) auf 6.0 (Dortmund) eine Rolle. Trotzdem sind die Ergebnisse insofern vergleichbar, als das sie in ähnlichen „Größenordnungen“ liegen. So liegen für IS die Laufzeiten der Benchmarks IIR und DFT in vergleichbaren Sekundenbereichen und die Laufzeiten der Benchmarks Whetstone und conv in vergleichbaren Stundenbereichen. Für ISRA werden Lösungen für den Benchmarks FIR sowohl in Saarbrücken, als auch in Dortmund innerhalb von Sekunden ermittelt; für genauere Aussagen liegen aus Saarbrücken zu wenige Ergebnisse vor.

Benchmark	Saarbrücken	Dortmund
FIR	-	0,06s
IIR	0,19s	0,14s
DFT	47,42s	164,57s
Whetstone	7440,00s	1197,46s
histo	>24h	48509,17s
conv	3660,00s	3770,11s

Tabelle 3.8: Vergleich der CPLEX Ergebnisse von Saarbrücken und Dortmund für IS mit Benchmarks der Gruppe A

Benchmark	Saarbrücken	Dortmund
FIR	1,69s	118,07s
IIR	>24h	11924,12s
DFT	>24h	64063,91s
Whetstone	-	>24h
histo	-	>24h
conv	-	>24h

Tabelle 3.9: Vergleich der CPLEX Ergebnisse von Saarbrücken und Dortmund für ISRA mit Benchmarks der Gruppe A

In den Tabellen 3.10 und 3.11 sind die Ergebnisse aus [Käs97] für die Lösungsstrategie SAIF angegeben. In der rechten Spalte sind jeweils die mit der Approximation berechneten und die optimale Anzahl an Instruktionen gegenübergestellt. Mit dieser Approximation werden für IS alle Benchmarks in einem Zeitraum von einigen Sekunden bis zu etwa einer Stunde gelöst. Nur für den Benchmark Whetstone wird dabei eine suboptimale Lösung mit einer Abweichung von einer Instruktion berechnet. Für ISRA werden für die Benchmarks FIR, IIR und DFT optimale Lösungen berechnet, die weiteren Berechnungen sind in [Käs97] aufgrund zu hoher Laufzeiten und zu großem Arbeitsspeicherbedarf nicht durchgeführt worden.

Benchmark	Saarbrücken	Anzahl Instruktionen	
		approximativ	optimal
FIR	-	-	8
IIR	0,27s	7	7
DFT	42,58s	14	14
Whetstone	85,96s	21	20
histo	3720,00s	31	31
conv	53,66s	17	17

Tabelle 3.10: Ergebnisse aus [Käs97] mit der schrittweisen Approximation der isolierten Flußanalyse für IS mit Benchmarks der Gruppe A

Benchmark	Saarbrücken	Anzahl Instruktionen	
		approximativ	optimal
FIR	19,58s	8	8
IIR	86,72s	7	7
DFT	560,00s	14	14
Whetstone	-	-	-
histo	-	-	-
conv	-	-	-

Tabelle 3.11: Ergebnisse aus [Käs97] mit der schrittweisen Approximation der isolierten Flußanalyse für ISRA mit Benchmarks der Gruppe A

Um einen direkten Vergleich der Qualität der Lösungsstrategien zu ermöglichen, wird eine mittlere Abweichung vom Optimum (M_O) nach folgender Formel berechnet:

$$M_O = \left(\frac{\left(\sum_{i=1}^{n} \frac{b_i}{o_i} - n \right) * 100}{n} \right) \tag{3.15}$$

mit M_O : *mittlere Abweichnung vom Optimum in Prozent*
n : *Anzahl der Benchmarks*
b_i : *berechnete Anzahl an Instruktionen eines Benchmarks i*
o_i : *optimale Anzahl an Instruktionen eines Benchmarks i*

Für die schrittweise Approximation der isolierten Flußanalyse ergibt sich für IS eine mittlere Abweichung vom Optimum $M_O = 1,0$ % und für ISRA $M_O = 0$ %.

Kapitel 4

CLP

In diesem Kapitel wird in Abschnitt 4.1 eine Einführung in die Grundlagen Constraint-logischer Programmierung gegeben. Es werden Begriffe und allgemeine Lösungsstrategien eines CLP-Systems vorgestellt. Im Anschluß werden konkrete Möglichkeiten zur Steuerung der Lösungssuche in CLP betrachtet. Abschnitt 4.2 stellt das CLP-Modell SCLP vor, mit dem das ILP-Modell SILP in einem CLP-Modell dargestellt wird. Damit werden die Optimierungsaufgaben, die im letzten Kapitel mit SILP dargestellt und mit CPLEX berechnet worden sind, in SCLP transformiert und mit ECLiPSe berechnet. Abschnitt 4.3 beschreibt die konkrete Implementierung, die Entwicklung der Lösungsstrategien und die Referenzberechnungen für CLP.

4.1 Grundlagen Constraint-logischer Programmierung

Die Constraint-logischen Programmiersprachen entstanden Mitte der achtziger Jahre aus der Fusion zweier Programmierkonzepte: Constraintlösen und Logikprogrammierung. Mit ECLiPSe wurde Anfang der neunziger Jahre ein CLP-System geschaffen, das eine logische Programmiersprache (Prolog) um Constraint-Konzepte zu einer CLP-Sprache verbindet. Andere CLP-Systeme sind unter anderem *Constraint Handling in Prolog (CHIP)* [SA91] und *Prolog III* [Pro91].

Unter einer konkreten CLP-Sprache in ECLiPSe versteht man die Erweiterung des CLP-Systems um eine Bibliothek mit Lösungsmechanismen, *Constraints* und *Domänen*. Eine dieser Bibliotheken ist die *Finite Domain*. Die dazugehörige CLP-Sprache heißt *CLP(Finite Domain)* (*CLP-FD*). Sie arbeitet auf endlichen Bereichen (zum Beispiel einem Teilbereich der Ganzen Zahlen) und dient als Programmiersprache zur Beschreibung der Probleme und der Lösungsstrategien in dieser Arbeit.

Eine Übersicht der wichtigsten Begriffe wird im folgenden gegeben; eine detailierte Beschreibung erfolgt in den weiteren Abschnitten.

- **Beschreibung**: Ein *Constraint Satisfaction Problem* (*CSP*) beschreibt ein Problem in CLP. Es besteht aus:
  - Variablen
  - Domänen
  - Constraints
- **Techniken**: Einige Techniken zur Steuerung der Suche und Reduktion des *Suchraums* werden vom CLP-System zur Verfügung gestellt:
  - *Backtracking*
  - *Constraint Propagation* (*CP*)
- **Darstellung**: Zur Illustration der Suche wird ein Suchbaum verwendet.
- **Suche**: Bei der Lösungssuche wird unterschieden zwischen:
  - Suche nach einer Lösung, mit einem *Labeling-Prädikat*.
  - Suche nach einer minimalen Lösung, mit dem Prädikat *minimize*.

4.1.1 Constraint

Definition 4.1 ***Domäne***

Eine Domäne D ist eine beliebige Menge. Mit D_X wird die Domäne D der Variablen X bezeichnet. Mit $D_{X_1,\ldots,X_n}$ wird die Domäne D einer Variablenmenge $V = \{X_1, \ldots, X_n\}$ benannt. Zur Vereinfachung der Schreibweise wird auch D_V für die Domäne einer Variablenmenge V verwendet.

Beispiel 4.1 Gegeben sei eine Variable X, dann bedeutet $D_X \in I\!R$, daß X Werte aus dem Reellen annehmen kann. Ein Beispiel für eine endliche Domäne einer Variablenmenge $V = \{X, Y\}$ ist $D_V = \{1, \ldots, 10\}$.

□

Definition 4.2 ***Constraint***

Es sei $V = \{X_1, \ldots, X_n\}$ eine endliche Menge von Variablen, welche einen Wert aus den Domänen $D_1, \ldots, D_n$ annehmen dürfen. Ein Constraint $c(X_{i1}, \ldots, X_{ik})$ zwischen k Variablen aus V beschreibt eine Teilmenge des kartesischen Produktes $D_{i1} \times D_{ik}$.

Beispiel 4.2 Gegeben sei die Variablenmenge $V = \{X, Y\}$ mit $D_V = \{1, 2, 3\}$. Dann sei ein 2-stelliger Constraint folgendermaßen definiert:

$$c(X, Y) : X < Y.$$

Als kartesisches Produkt wird der Constraint $c(X, Y)$ durch die Menge $\{(1, 2), (1, 3), (2, 3)\}$ ausgedrückt. Da die erste Schreibweise deutlich kompakter ist, wird sie im folgenden verwendet.

□

4.1.2 Constraint Satisfaction Problem

Definition 4.3 ***Constraint Satisfaction Problem***

Ein Constraint Satisfaction Problem $CSP = (V, D_V, C)$ *besteht aus einer Menge Variablen* V*, ihren Domänen* D_V *und einer Menge Constraints* $C = \{c_1, \ldots, c_m\}$*.*

Definition 4.4 ***Lösung eines CSP***

Es sei ein $CSP = (V, D_V, C)$ *gegeben. Eine CSP-Lösung ist eine Belegung aller Variablen* $X \in V$ *mit Werten ihrer Domänen* D_X*, so daß gegen keinen der Constraints* C *verstoßen wird. Man sagt auch, eine solche Belegung erfüllt alle Constraints* $c \in C$*.*

Beispiel 4.3 Es sei wie in Beispiel 4.2 ein CSP gegeben mit:

$$V = \{X, Y\},\ D_V = \{1, 2, 3\},\ C = \{(X < Y)\}^{\mathrm{I}}.$$

Die möglichen CSP-Lösungen sind:

$$(X = 1,\ Y = 2),\ (X = 1,\ Y = 3)\ \mathit{und}\ (X = 2,\ Y = 3).$$

□

4.1.3 Suchbaum

Zur graphischen Darstellung der Lösungssuche für ein CSP werden *Suchbäume* eingeführt. Ein Suchbaum bildet eine Struktur über den Suchraum eines CSP.

Definition 4.5 ***Suchraum***

Ein Suchraum $S_R = (V, D_V)$ *ist eine Menge, die sich aus den möglichen Belegungen der Variablen aus* V *mit Werten aus* D_V *ergibt. Die Größe eines Suchraums ist [JS99]:*

$$\text{Größe Suchraum} = \text{Größe Domäne}^{\text{Anzahl Variablen}} = |D_V|^{|V|} \tag{4.1}$$

Beispiel 4.4 Die Menge der Variablen sei $V = \{X, Y\}$, die Domäne $D_V = \{1, 2, 3, 4\}$. Damit enthält der Suchraum $S_R = 4^2 = 16$ Elemente. In Abbildung 4.1 ist der Suchraum dargestellt durch eine Menge von Knoten, die mit den möglichen Belegungen der Variablen markiert sind.

□

[I] Auf die explizite Angabe der Variablen eines Constraints wird zur Vereinfachung der Schreibweise im weiteren verzichtet.

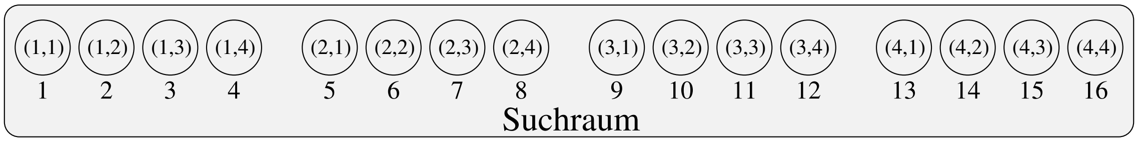


Abbildung 4.1: Suchraum zu Beispiel 4.4

Definition 4.6 ***Suchbaum***

Ein Suchbaum ist ein kantenmarkierter Baum $S_B(V, D_V)$. Eine Kante ist eine Belegung einer Variablen $X \in V$ mit einem Wert aus ihrer Domäne D_X. Ein Pfad von der Wurzel bis zu einem Blatt entspricht einer eindeutigen Belegung aller Variablen aus V mit Elementen aus D_V. Ein Knoten K ist durch die Belegung der Variablen, auf dem Pfad von der Wurzel bis zum Knoten, gekennzeichnet.

Sei n *die Anzahl der Variablen V und* m *die Größe der Domäne D_V. Die Höhe des Baumes ergibt sich aus der Anzahl der Variablen: $Höhe(S_B) = n$. Die Anzahl möglicher Pfade und damit die Anzahl der Blätter ergibt sich aus den möglichen Kombinationen einer Belegung für V mit Elementen aus D_V: $Anzahl\ Blätter(S_B) = m^n$.*

Durch einen Suchbaum S_B wird ein Suchraum S_R strukturiert. Der Zusammenhang zwischen S_B und S_R besteht darin, daß die Blätter des Suchbaums $S_B(V, D_V)$ den Elementen des Suchraums $S_R = (V, D_V)$ entsprechen.

Beispiel 4.5 Es sei ein CSP gegeben mit:

$$V = \{X, Y\},\ D_V = \{1, 2, 3\},\ C = \{\}.$$

Dann können Suchraum und Suchbaum zu diesem CSP wie in Abbildung 4.2 dargestellt werden. Die Höhe des Suchbaums entspricht der Anzahl der Variablen und ist gleich zwei. Der Suchraum besitzt $|D_V|^{|V|} = 3^2 = 9$ Elemente, entsprechend der neun Blätter des Suchbaums.

□

Die Struktur (*Ordnung*) eines Suchbaums entsteht zum einen durch die Reihenfolge, in der die Variablen betrachtet werden, und zum anderen durch die Reihenfolge, mit der die Variablen V eines CSP mit Elementen aus D_V belegt werden. Zwei unterschiedlich strukturierte Suchbäume eines CSPs sind in den Abbildungen 4.2 und 4.3 dargestellt.

Beispiel 4.6 Es sei wie in Beispiel 4.5 ein CSP gegeben mit:

$$V = \{X, Y\},\ D_V = \{1, 2, 3\},\ C = \{\}.$$

Dann ist es möglich den dazugehörigen Suchbaum wie in Abbildung 4.3 darzustellen. In diesem Fall werden die Variablen X und Y abwechselnd betrachtet. Als Belegung für die Variablen werden die jeweils noch unbetrachteten Elemente ihrer Domänen gewählt.

□

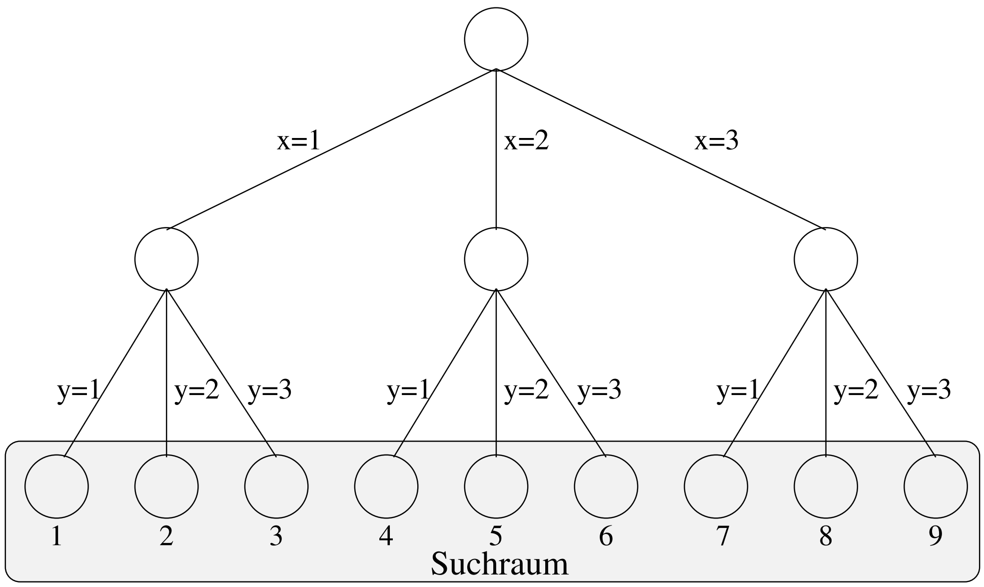


Abbildung 4.2: Suchbaum zu Beispiel 4.5

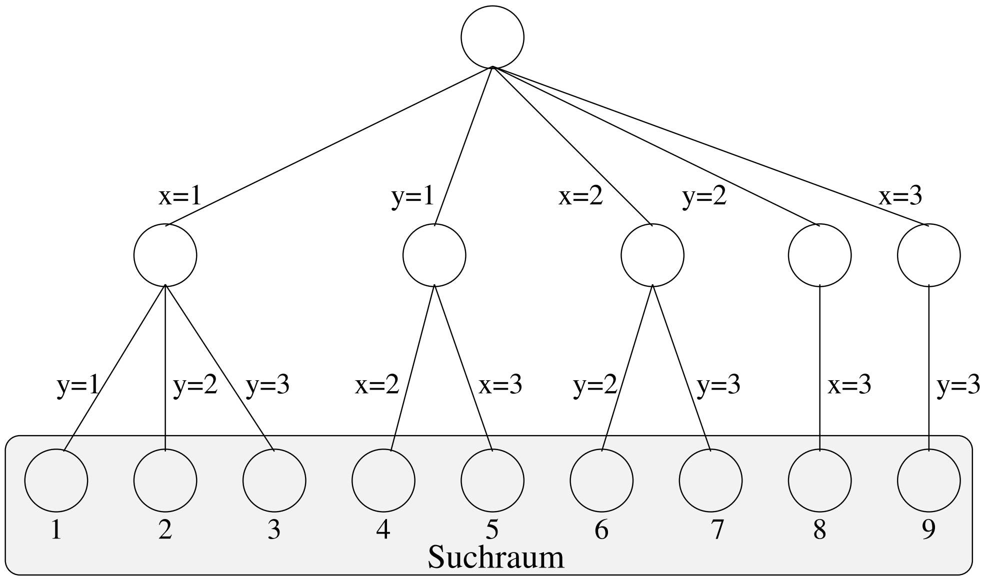


Abbildung 4.3: Suchbaum zu Beispiel 4.6

Definition 4.7 ***Lösungssuche***

Die Suche nach einer möglichen Lösung eines CSP wird als Lösungssuche bezeichnet.

Zur Darstellung der Lösungssuche für ein CSP wird ein Suchbaum verwendet, dessen Knoten durch die Menge der Constraints C_{Suche} gekennzeichnet sind. In der Wurzel des Suchbaums entspricht C_{Suche} der Menge C des CSP. Wenn eine Kante mit einer Variablenbelegung $X_i = d_i, d_i \in D_X$ markiert wird, wird ein zusätzlicher Constraint $X_i = d_i$ in die Menge C_{Suche} eingefügt. Einerseits kennzeichnet diese Einfügung einen Knoten des Suchbaums eindeutig, andererseits dient sie zur Steuerung der Lösungssuche.

Die grundlegende Lösungssuche in ECLiPSe kann mit Tiefensuche (*depth-first-search*) in einem Suchbaum beschrieben werden. Dabei wird ein *Backtrack*-Algorithmus[II] verwendet. Bei der Lösungssuche können zwei Fälle auftreten:

1. Es wird ein Blatt erreicht, damit ist eine CSP-Lösung gefunden, die der Belegung der Variablen auf dem Pfad von der Wurzel zum Blatt entspricht.
2. Es wird ein Knoten K_j erreicht. In diesem Fall gibt es zwei Möglichkeiten:
   a) Durch die Belegung einer Variablen X_i werden keine Constraints verletzt. Die Lösungssuche wird mit der Nachfolgervariablen X_{i+1} fortgesetzt und wieder unterschieden, ob ein Blatt (1) oder ein Knoten (2) erreicht wird.
   b) Durch die Belegung einer Variablen X_i wird, von dem Knoten K_j ausgehend, ein Constraint der Menge C_{Suche} in K_j verletzt. In diesem Fall wird über Backtracking die nächste Möglichkeit gesucht. Dies kann entweder eine andere Belegung der Variablen X_i sein, wenn die Domäne D_i der Variablen noch weitere, bis dahin nicht betrachtete Elemente besitzt, oder eine andere Belegung der Vorgängervariablen $X_{i-1}, \ldots, X_1$ sein. Dieser Schritt wird solange wiederholt, bis entweder Möglichkeit (1) oder (2a) eintritt, oder bis der gesamte Suchbaum untersucht ist. In diesem Fall ist keine Lösung für das CSP möglich.

Der Ablauf bei dieser Lösungssuche ist in Beispiel 4.7 anhand eines Suchbaums illustriert.

Beispiel 4.7 Es sei ein CSP gegeben mit:

$$V = \{X, Y\},\ D_V = \{1, 2\},\ C = \{(X + Y > 3)\}.$$

Ein resultierender Suchbaum ist in Abbildung 4.4 dargestellt.

In Abbildung 4.4 sind die einzelnen Schritte der Lösungssuche durch eine alphabetische Markierung in den Knoten gekennzeichnet. Im zweiten Lösungsschritt (Knoten b) wird zum Beispiel der Constraint $(1 + 1 > 3)$ verletzt; erst im letzten möglichen Lösungsschritt (Knoten f) wird mit $(X = 2,\ Y = 2)$ eine CSP-Lösung gefunden. Der Pfad, auf dem im Suchbaum die Lösung erreicht wird, ist **fett** hervorgehoben.

[II] Ein Backtrack-Algorithmus zeichnet sich dadurch aus, daß er bei Erreichen einer Sackgasse soweit zurück geht, bis er einen neuen Weg verfolgen kann.

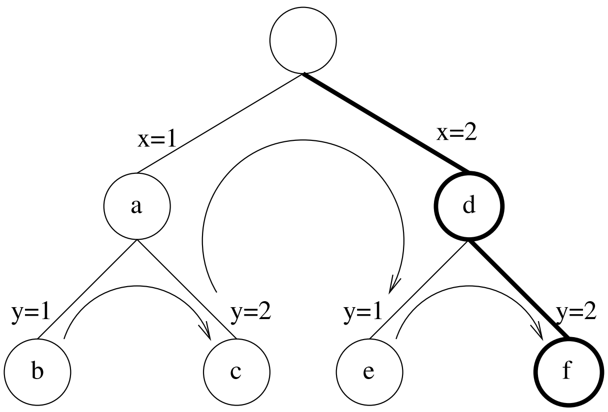


Abbildung 4.4: Suchbaum zu Beispiel 4.7

□

Dieser Suchbaum dient nur zur Illustration. Das CLP-System arbeitet aus Effizienz- und Speicherplatzgründen immer nur auf einem Teilbereich und nicht auf dem kompletten Suchbaum. Durch die Lösungsuche werden nur die Teile des Suchbaums generiert, die wirklich betrachtet werden, bis eine Lösung gefunden ist. Denkbar wäre es, den Suchbaum vor der Lösungssuche komplett zu generieren. Da der Suchbaum exponentiell mit der Anzahl der Variablen wächst, ist eine derartige Darstellung für eine größere Anzahl Variablen nur mit hohem Rechenaufwand und Speicherplatz erreichbar. Durch die Ordnung des Suchbaums ist es ausreichend, einen *Stack* mit den in C_{Suche} eingefügten Constraints zu verwalten. Aus diesen Informationen können, in Zusammenhang mit den Domänen der Variablen, die noch unbetrachteten Möglichkeiten und damit der nächste Lösungsschritt bestimmt werden.

Beispiel 4.8 Es sei das CSP aus Beispiel 4.7 gegeben. Zur Darstellung des Stacks wird eine Liste L verwendet, deren erstes Element das zuletzt eingefügte ist. In Knoten b ist $L = [(Y = 1), (X = 1), (X + Y > 3)]$. Da an dieser Stelle keine andere Variable belegt werden kann, wird über Backtracking eine andere Belegung für die zuletzt eingefügte Variable Y gesucht, die sich aus der Domäne D_Y ergibt. Für Y kann noch $(Y = 2)$ gewählt werden, damit wird Knoten c mit $L = [(Y = 2), (X = 1), (X + Y > 3)]$ generiert. Da auch in diesem Schritt noch keine Lösung für das CSP gefunden ist, wird erneut Backtracking verwendet. In diesem Fall gibt es keine weitere Möglichkeit für eine andere Belegung von Y, also wird die Vorgängervariable X in der Liste betrachtet. Die Domäne D_X bietet noch die Möglichkeit $(X = 2)$, damit wird Knoten d mit $L = [(X = 2), (X + Y > 3)]$ generiert. Die nächsten Lösungsschritte ergeben sich analog.

□

An Beispiel 4.7 können zwei Faktoren, welche die Effizienz einer Lösungsuche in einem CLP-System verbessern können, verdeutlicht werden. Die Suche wäre effizienter, wenn es einen anderen Durchlauf durch den Suchbaum geben würde, mit dem die CSP-Lösung schneller erreicht wird. Diese Möglichkeit wird in Abschnitt 4.1.6 betrachtet. Eine andere Methode könnte den Suchraum reduzieren, indem sie Variablenkombinationen, mit denen Constraints verletzt werden, erkennt. Ein derartiges Verfahren ist Thema des folgenden Abschnitts.

4.1.4 Constraint Propagation

Definition 4.8 ***Vollständige Konsistenztechnik***

Es sei ein $CSP = (V, D_V, C)$ *gegeben. Eine Konsistenztechnik heißt vollständig, wenn sie in der Lage ist, für eine beliebige Menge Constraints* C *eines* CSP *zu entscheiden, ob sie erfüllt werden kann.*

Beispiel 4.9 Es sei ein CSP gegeben mit:

$$V = \{X, Y\},\ D_V = \{1, 2, 3\},\ C = \{(X < Y), (X > 2)\}.$$

Eine Überprüfung der Konsistenz zeigt, daß die Constraints dieses CSP nicht erfüllbar sind. Der Constraint $c_2 : X > 2$ kann nur durch $(X = 3)$ erfüllt werden, damit ist der Constraint $c_1 : X < Y$ durch kein Element für Y aus der Menge D_V erfüllbar.

□

Mögliche Konsistenztechniken sind ein Theorembeweiser [MS98] oder der Simplex-Algorithmus[III]. Der Test, ob eine Menge von Constraints vollständig konsistent ist, ist im allgemeinen $\mathcal{NP}$-vollständig [Bie95]. Wegen der hohen Laufzeiten wurden im CLP-System Verfahren implementiert, die zumindestens für bestimmte Klassen von Constraints oder bestimmte Wertebereiche (zum Beispiel endliche Bereiche bei der Finite Domain) effizient arbeiten. Ein effizientes Verfahren, das jedoch keine vollständige Konsistenz garantieren kann, wird von CLP-FD verwendet: die Constraint Propagation (CP).

Durch das CLP-System wird bei der Finite Domain bei jedem Einfügen eines Constraints CP durchgeführt. Durch die Bearbeitung mit CP wird eine Anpassung der Domänen D_V eines $CSP = (V, D_V, C)$ in Abhängigkeit der Constraints C vorgenommen. Dadurch wird der Suchraum eines CSP stark eingeschränkt. Der einfachste Fall mit einem Constraint ist in Beispiel 4.10 dargestellt.

Beispiel 4.10 Es sei wie in Beispiel 4.2 ein CSP gegeben mit:

$$\begin{array}{lcl} V &=& \{X, Y\}, \\ D_X &=& \{1, 2, 3\}, \\ D_Y &=& \{1, 2, 3\}, \\ C &=& \{(X < Y)\}. \end{array}$$

[III] siehe Kapitel 3.1.2, Seite 32

Die Domänen werden durch CP soweit eingeschränkt, daß sie keine Elemente enthalten, die nicht zu einer Lösung gehören können. Zum Beispiel kann $(X = 3)$ zu keiner Lösung gehören, weil es kein Element in D_Y gibt, mit dem der Constraint erfüllt wird. Damit wird:

$$\begin{aligned} D_X &= \{1, 2\}, \\ D_Y &= \{2, 3\}, \end{aligned}$$

denn auch mit $(Y = 1)$ kann der Constraint nicht erfüllt werden. Der durch den Constraint reduzierte Suchbaum ist in Abbildung 4.5 dargestellt. Der Suchbaum konnte um die gestrichelten Teile reduziert werden, und der Suchraum enthält nur noch vier statt vorher neun Elemente.

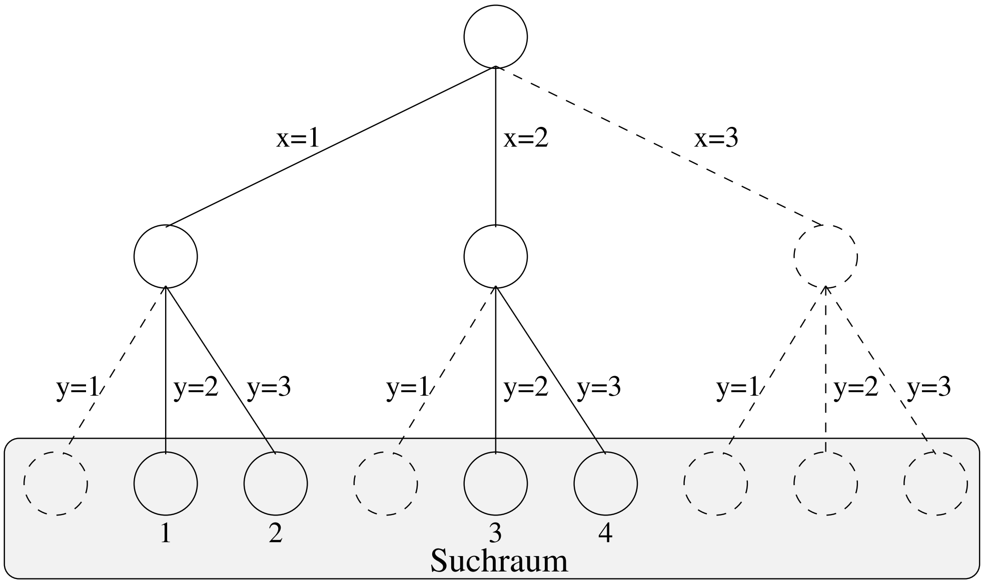


Abbildung 4.5: Reduzierter Suchbaum zu Beispiel 4.10

□

Daß CP keine vollständige Konsistenztechnik ist, demonstriert Beispiel 4.11.

Beispiel 4.11 Es sei ein CSP gegeben mit:

$$\begin{aligned} V &= \{X, Y, Z\}, \\ D_V &= \{1, 2\}, \\ C &= \{(X \neq Y), (Y \neq Z), (Z \neq X)\}. \end{aligned}$$

Durch den Constraint $c_1 : X \neq Y$ können die Domänen D_X und D_Y nicht eingeschränkt werden, denn für $(X = 1)$ muß in der Domäne D_Y das Element $(Y = 2)$ verbleiben, und für $(X = 2)$ das Element $(Y = 1)$. Analog gilt dies auch für die Constraints c_2 und c_3.

Damit sind die Domänen nach der Bearbeitung durch CP unverändert und es wird, anders als in Beispiel 4.9, nicht erkannt, daß es keine Lösung für dieses CSP geben kann.

□

Bei einem Aufruf von CP durch das CLP-System werden zuerst die Constraints der Menge $C = \{c_1, \ldots, c_n\}$ in eine Liste $L = [l_1, \ldots, l_n]$ abgebildet[IV]. Das jeweils erste Element der Liste (l_1) wird entnommen und die Domänen der Variablen angepaßt. Es kann zum Beispiel der Fall auftreten, daß bei der Konsistenzprüfung von l_2 die Domäne einer der Variablen von l_1 geändert wird. In diesem Fall wird l_1 an das Ende der Liste L eingefügt und ein weiteres Mal ausgewertet. Allgemein werden Constraints, die Variablen enthalten, deren Domänen von CP verändert worden sind, an das Ende der Liste L eingefügt, sofern sie nicht bereits in der Liste enthalten sind. Das Verfahren bricht ab, wenn die Liste L leer ist.

In Abbildung 4.6 ist das CP-Verfahren schematisch dargestellt. Bei einem Aufruf von CP werden die folgenden Arbeitsschritte durchlaufen:

1. Initialisierung: Abbildung von $C = \{c_1, \ldots, c_n\} \rightarrow L = [l_1, \ldots, l_n]$
2. Fallunterscheidung:
   a) leere Liste: Ende der Bearbeitung
   b) nicht leere Liste: Weiterbearbeitung in Schritt 3
3. Entnahme des Constraints $l_1(X_1, \ldots, X_k)$ aus L
4. Reduzierung der Domänen der Variablen $V = \{X_1, \ldots, X_k\}$
5. Einfügen von Constraints an das Ende der Liste L, wenn sie
   a) Variablen aus V enthalten, deren Domänen von CP modifiziert worden sind, und
   b) nicht in L enthalten sind
6. Erneute Bearbeitung ab Schritt 2

Die Arbeitsweise von CP bei mehr als einem Constraint wird in Beispiel 4.12 gezeigt.

Beispiel 4.12 Gegeben sei ein $CSP = (V, D_V, C)$:

$$\begin{array}{lcl} V & = & \{X, Y\}, \\ D_X & = & \{1,2,3,4\}, \\ D_Y & = & \{1,2,3,4\}, \\ C & = & \{(X < Y), (X > 1)\}. \end{array}$$

Bei der Bearbeitung durch CP werden als erstes die Constraints des CSP ausgewertet:

$$\begin{array}{lclcl} \mathit{Auswertung\ von}\ c_1 : X < Y & \Rightarrow & D_X & = & \{1,2,3\}, \\ & & D_Y & = & \{2,3,4\}, \\ & & & & \\ \mathit{Auswertung\ von}\ c_2 : X > 1 & \Rightarrow & D_X & = & \{2,3\}, \\ & & D_Y & & \{2,3,4\}. \end{array}$$

[IV] Die Abbildung auf eine Liste wird zur besseren Darstellung verwendet und gibt nicht die reale Datenstruktur wieder.

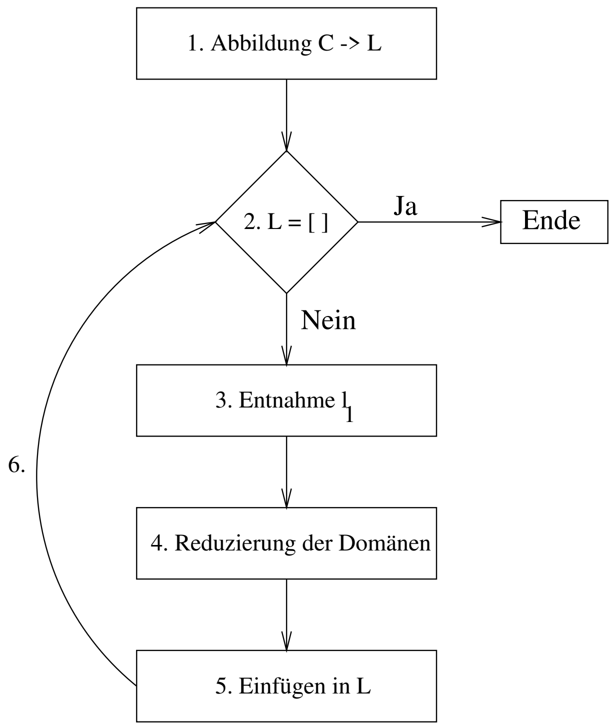


Abbildung 4.6: Schematische Darstellung der Constraint Propagation

Bei der Bearbeitung von c_2 ist die Domäne D_X verändert worden, deshalb werden alle weiteren Constraints neu bearbeitet, die X enthalten. In diesem Beispiel wird dadurch c_1 erneut ausgewertet und damit die Domäne D_Y weiter eingeschränkt:

$$\begin{array}{lrcl} \textit{erneute Auswertung von } c_1 & \Rightarrow D_X & = & \{2,3\}, \\ & D_Y & = & \{3,4\}. \end{array}$$

Ein resultierender Suchbaum ist in Abbildung 4.7 dargestellt. Er enthält statt der ursprünglich 16 Blätter nur noch vier.

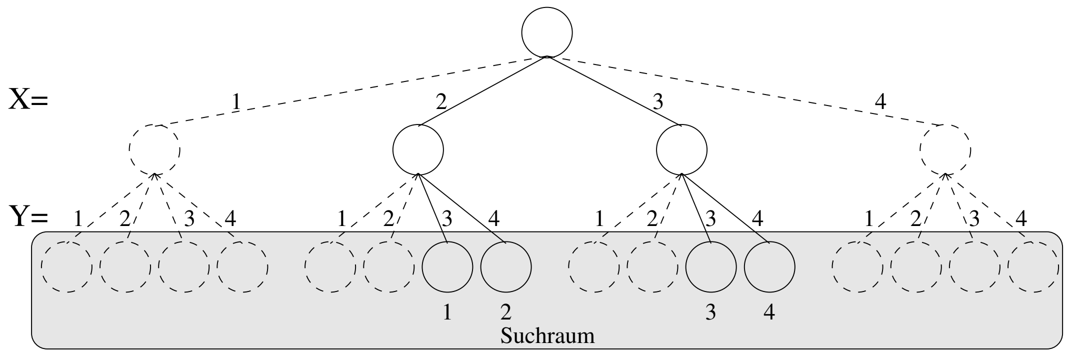


Abbildung 4.7: Reduzierter Suchbaum zu Beispiel 4.12

□

Eine wichtige Eigenschaft von CP ist *Monotonie*; sie garantiert, daß dieses Verfahren nach endlich vielen Schritte abbricht. Domänen können durch CP nur monoton kleiner werden.

$$D' \subseteq D, \; mit \; D \xrightarrow{CP} D' \tag{4.2}$$

Ohne diese Eigenschaft könnte sich ein unendlicher Zyklus ergeben, wie Beispiel 4.13 zeigt.

Beispiel 4.13 Gegeben sei ein CSP mit:

$$\begin{array}{ll} V & = \{X, Y\}, \\ D_V & = \{1, 2\}, \\ C & = \{(X > Y), (X > 0)\}. \end{array}$$

CP ohne Monotonie:

1. Durch $c_1 : X > Y$ wird $D_X = \{2\}, \; D_Y = \{1\}$.
2. Durch $c_2 : X > 0$ könnte, wenn CP nicht monoton wäre, die Domäne von X wieder so groß wie im CSP angegeben werden, also $D_X = \{1, 2\}$. Dadurch würde der Constraint c_1, der ebenfalls die Variable X enthält, erneut auswertet (Schritt 1). Dies hätte zur Folge, daß auch Schritt 2 ein weiteres Mal ausgeführt wird usw. $\Rightarrow$ unendlicher Zyklus.

CP mit Monotonie:

1. Durch c_1 wird $D_X = \{2\}, \; D_Y = \{1\}$.
2. Durch c_2 werden die Domänen nicht verändert, und das Verfahren bricht ab.

□

4.1.5 ECLiPSe Notation

Zur Darstellung der Verfahren und Lösungsstrategien der nächsten Abschnitte werden einige Vereinbarungen zur Notation getroffen. Die Notation wird eng an ECLiPSe angelehnt, ist aber zur einfacheren Darstellung modifiziert. Ziel der Notation ist eine leicht nachvollziehbare Darstellung der Lösungsstrategien. Auf eine Einführung in die logische Programmierung wird an dieser Stelle verzichtet und auf die umfangreiche Literatur zu diesem Thema, wie zum Beispiel [Bra88] oder [HKB88], verwiesen.

Der Begriff *Prädikat* stammt aus der Logikprogrammierung und wird dort wesentlich abstrakter gefaßt, als es für die Darstellung in dieser Arbeit notwendig ist. Aus diesem Grund wird der Begriff in dieser Arbeit in einer eingeschränkten Form verwendet. Ein Prädikat bezeichnet ein programmiersprachliches Konstrukt in CLP, daß mit einer Prozedur in einer imperativen Programmiersprache vergleichbar ist.

Anders ausgedrückt kann ein CLP-Programm verschiedene Unterprogramme enthalten, die als Prädikate bezeichnet werden.

In den Programmbibliotheken von ECLiPSe ([AA97], [PB97], [Mei96]) sind eine Reihe von Prädikaten für die unterschiedlichsten Aufgaben implementiert, weitere können vom Benutzer definiert werden. Einige der Prädikate von ECLiPSe werden in den nächsten Abschnitten dieses Kapitels vorgestellt und verwendet. Ein benutzerdefiniertes Prädikat beschreibt in Beispiel 4.14 ein CSP als CLP-Programm.

Beispiel 4.14 Gegeben sei ein CSP mit:

$$V = \{X, Y\},\ D_V = \{1, 2, 3\},\ C = \{(X < Y)\}.$$

Ein benutzerdefiniertes Prädikat zu diesem CSP kann wie folgt lauten:[V]

$$\begin{array}{rl}
\mathit{beispiel}(X,Y) :- & \%\ \mathit{Prädikatname : beispiel,\ Variablen : X, Y} \\
V = [X,Y], & \%\ \mathit{definiert\ die\ Variablenmenge\ V\ als\ Liste} \\
V :: [1..3], & \%\ \mathit{definiert\ die\ Domäne\ zu\ V} \\
X < Y. & \%\ \mathit{drückt\ den\ Constraint\ aus}
\end{array}$$

□

4.1.6 Labeling

Mit CP kann eine Reduzierung des Suchraums eines CSPs erreicht werden, aber keine vollständige Reduktion auf die Lösungen des CSP. Um eine Lösung des CSP zu berechnen, ist eine Suche notwendig. Die Methoden, mit denen eine Suche gesteuert wird, werden *Labeling-Strategien* genannt.

Definition 4.9 ***Labeling-Strategie, -Prädikat***

Mit dem Begriff Labeling-Strategie wird eine Methode zur Lösungssuche bezeichnet. Ziel dieser Strategie ist die effiziente Suche nach einer Lösung für das CSP. Ein Labeling-Prädikat ist die Programm-technische Umsetzung für eine Labeling-Strategie in ECLiPSe.

Die Labeling-Strategien modifizieren die in Abschnitt 4.1.3 beschriebene allgemeine Lösungssuche für ein CSP durch zwei Faktoren:

1. Die Auswahl einer Variablen X_i aus V, oder

2. Die Auswahl eines Elementes d_i aus einer Domäne D.

Eine Labeling-Strategie ist in CLP-FD immer mit einem Aufruf von CP verbunden. Die Auswahl einer Variablen X_i und ihrer Belegung d_i wird durch einen Constraint $X_i = d_i$ ausgedrückt, der zu der Menge C hinzugefügt wird. Zur Überprüfung der

[V]Das Sonderzeichen % kennzeichnet einen Programmkommentar

Konsistenz und Anpassung der Domänen wird anschließend eine Bearbeitung mit CP durchgeführt.

Die Auswirkungen auf die Lösungssuche durch die Auswahl eines Elementes aus einer Domäne 2 werden in Beispiel 4.15 an zwei verschiedenen benutzerdefinierten Labeling-Prädikaten demonstriert: *labelingmindomain(V)* und *labelingmaxdomain(V)*. Labeling-Prädikate, welche die Auswahl der Variablen beeinflussen (1), werden in Abschnitt 4.3.1 betrachtet.

Bei einer Lösungssuche mit dem Labeling-Prädikat labelingmindomain(V) wird aus einer Variablenliste $V = [X_1, \ldots, X_n]$ die erste Variable X_1 ausgewählt. Diese Variable wird mit dem ersten Element der Domäne $D_{X_1} = [d_{X_1}, \ldots, d_{X_m}]$ belegt und ein Constraint $X_1 = d_{X_1}$ in C eingefügt. Mit CP wird die Konsistenz der Constraints geprüft und mögliche Reduzierungen der Domänen vorgenommen. Hierbei können zwei Fälle unterschieden werden:

1. Die Konsistenz der Domänen bleibt gewahrt. In diesem Fall wird die nächste Variable der Liste durch lablingmindomain ausgewählt, mit einem Wert belegt und eine Konsistenzprüfung mit CP durchgeführt.
2. Mindestens ein Constraint kann nicht erfüllt werden. In diesem Fall wird der zuletzt eingefügte Constraint entfernt und, wie in Abschnitt 4.1.3 (Seite 56) beschrieben, Backtracking durchgeführt. Variablen- und Werteauswahl beim Backtracking werden durch labelingmindomain gesteuert.

Das Prädikat labelingmaxdomain(V) wählt ebenfalls die erste Variable X_1 der Variablenliste V aus, belegt sie aber mit dem letzten Wert ihrer Domäne $D_{X_1} = [d_{X_1}, \ldots, d_{X_m}]$. Es wird ein Constraint $X_1 = d_{X_m}$ in C eingefügt. Die weiteren Schritte verlaufen bei diesem und bei allen weiteren Labeling-Prädikaten, wie oben beschrieben. Zur Illustration wird die Lösungssuche mit diesen beiden Prädikaten in Beispiel 4.15 durch Suchbäume dargestellt.

CLP-Programme, die sowohl ein CSP beschreiben, als auch eine Lösungsstrategie in Form eines Labeling-Prädikates enthalten, werden im folgenden *solve* genannt. Zur Unterscheidung von zwei Labeling-Prädikaten werden auch *solveA* und *solveB* als Namen verwendet.

Beispiel 4.15 Es sei ein CSP gegeben mit:

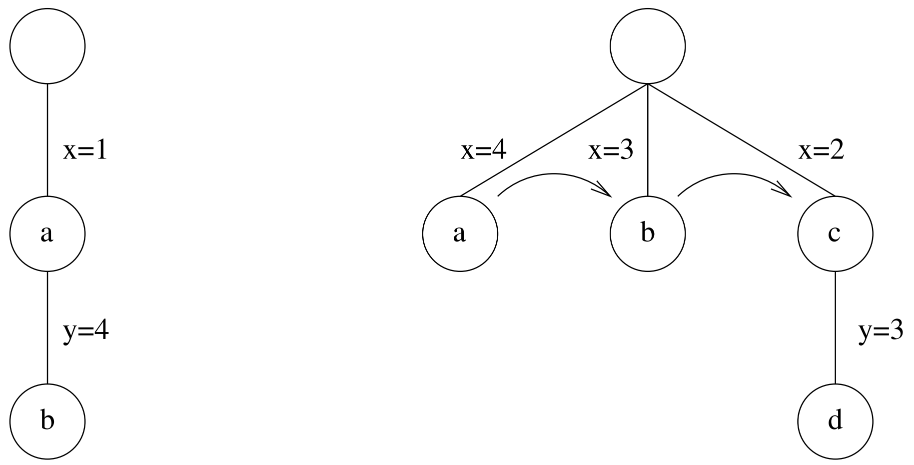
$$\begin{aligned} V &= \{X, Y\}, \\ D_X &= \{1, 2, 3, 4\}, \\ D_Y &= \{1, 2, 3, 4\}, \\ C &= \{(X + Y = 5), (X < Y)\}. \end{aligned}$$

Das CSP und die Labeling-Prädikate A) labelingmindomain und B) labelingmaxdomain werden durch die Programme solveA und solveB in Abbildung 4.8 beschrieben.

Durch das CLP-System werden bei der Initialisierung der Programme die Domänen durch CP angepaßt. Es ergeben sich die folgenden Domänen:

$$\begin{aligned} D_X &= \{1, 2, 3\}, \\ D_Y &= \{2, 3, 4\}. \end{aligned}$$

Der Ablauf der Suche in A) und B) ist durch zwei Suchbäume dargestellt. Die Knoten werden in alphabetischer Reihenfolge erreicht.



A) labelingmindomain

B) labelingmaxdomain

```
solveA(X,Y):-
        V=[X,Y],
        V::[1,2,3,4],
        X + Y = 5,
        X < Y,
        labelingmindomain(V).
```

```
solveB(X,Y):-
        V=[X,Y],
        V::[1,2,3,4],
        X + Y = 5,
        X < Y,
        labelingmaxdomain(V).
```

Abbildung 4.8: Suchbäume und Programme zu Beispiel 4.15

Mit der Labeling-Strategie A) sind zwei Schritte zu einer Lösung des CSP notwendig. Im Gegensatz dazu benötigt die Labeling-Strategie B) vier Schritte. Die Arbeitsweise der Prädikate kann dadurch veranschaulicht werden, daß zum Beispiel für Programm A) zuerst die Variable X ausgewählt wird und mit $(X = 1)$ belegt wird. In Knoten a stehen deshalb die folgenden Constraints in der Liste: $L = [(X = 1), (X < Y), (X + Y = 5)]$. Durch die Constraints wird die Domäne von Y auf $D_Y = \{4\}$ eingeschränkt. Im zweiten Schritt wird die einzig mögliche Belegung für Y gewählt und in Knoten b eine Lösung des CSP mit $(X = 1, Y = 4)$ erreicht.

Von der Labeling-Strategie B) wird ebenfalls die Variable X zuerst ausgewählt, aber mit dem größten Wert ihrer Domäne belegt. In Programm B) wird deshalb folgende Variablenbelegung gewählt: $(X = 4)$. Daraus ergibt sich die folgende Liste von Constraint in Knoten a: $L = [(X = 4), (X < Y), (X + Y = 5)]$. Diese Constraints können nicht erfüllt werden und über Backtracking (in der Abbildung durch einen Pfeil symbolisiert) wird der Knoten b erreicht. Die Constraints in Knoten b ($L = [(X = 3), (X < Y), (X + Y = 5)]$) können ebenfalls nicht erfüllt werden. Durch weiteres Backtracking wird in Knoten c der Constraint $X = 2$ in die Liste eingefügt ($L = [(X = 2), (X < Y), (X + Y = 5)]$) und die Domäne von Y auf $D_Y = \{3\}$ eingeschränkt. Mit dieser einzig möglichen Belegung für Y wird in Knoten d eine weitere Lösung des CSP mit $(X = 2, Y = 3)$ erreicht.

□

In Beispiel 4.15 werden durch verschiedene Labeling-Prädikate nicht nur unterschiedlich viele Schritte in der Lösungssuche benötigt, sondern auch zwei unterschiedliche CSP-Lösungen berechnet. Diese Eigenschaften werden in den folgenden Abschnitten bei der Entwicklung von Lösungsstrategien ausgenutzt.

4.1.7 Minimize

Mit den Labeling-Strategien kann die Effizienz der Lösungssuche für ein CSP verbessert werden. Eine CSP-Lösung ist die Belegung aller Variablen mit Werten aus ihren Domänen. An eine solche Lösung wird nun die zusätzliche Anforderung gestellt, daß sie einen Zielwert minimiert. Für die Suche nach einer minimalen Belegung eines Zielwertes wird ein Prädikat minimize der Finite Domain benutzt. Diesem Prädikat wird im einfachsten Fall[VI] ein Labeling-Prädikat, mit einer Variablenmenge V, und eine zu minimierende Zielvariable Z übergeben. Mit dem Labeling-Prädikat wird, wie bereits gezeigt, zuerst eine CSP-Lösung bestimmt. In einer CSP-Lösung sind alle Variablen, also auch Z, eindeutig mit Werten aus ihren Domänen belegt. Wenn Z_1 die Belegung der Variablen Z in der ersten Lösung ausdrückt, wird ein Constraint $Z < Z_1$ in die Menge C eingefügt und die weitere Lösungssuche über Backtracking erneut angestoßen. Es können drei Fälle unterschieden werden:

1. Eine bessere Lösung, d.h. ein kleinerer Zielwert Z_i wird für Z gefunden, dann wird der Constraint $Z < Z_i$ eingefügt, und die Lösungssuche über Backtracking weitergeführt.

2. Der Constraint $Z < Z_i$ wird verletzt, d.h. in dem Teilbaum können keine besseren Lösungen für Z gefunden werden, dann wird die Suche in diesem Teilbaum abgebrochen, der Constraint $Z < Z_i$ aus C entfernt und Backtracking durchgeführt.

3. Es gibt keine weiteren Möglichkeiten zur Lösungssuche, dann wird die letzte (und damit kleinste) Lösung für das CSP ausgegeben.

Beispiel 4.16 Es sei ein CSP gegeben mit:

$$\begin{array}{ll} V & = \{X, Y\}, \\ D_V & = \{1, 2, 3\}, \\ C & = \{c_1, c_2\}, \\ c_1: & X < Y, \\ c_2: & Z = X - Y + 3. \end{array}$$

Ein entsprechendes CLP-Programm lautet:

$$\begin{array}{lll} solve(X, Y, Z) :- & & \%\ Prädikatname \\ \quad V & = [X, Y], & \%\ Variablendefinition \\ \quad V & :: [1, 2, 3], & \%\ Domänendefinition \\ \quad X & < Y, & \%\ Constraint\ c_1 \\ \quad Z & = X - Y + 3, & \%\ Constraint\ c_2 \\ \quad \multicolumn{2}{l}{minimize(labelingmindomain(V), Z).} & \%\ Such-\ und\ Mini- \\ & & \%\ mierungsprädikat \end{array}$$

[VI] Eine Erweiterung dieses Prädikats wird in Abschnitt 4.3.7 benutzt.

Die Domänen werden bei der Initialisierung des Programms mit Hilfe von CP eingeschränkt. Damit wird:

$$\begin{aligned} D_X &= \{1,2\}, \\ D_Y &= \{2,3\}, \end{aligned}$$

In Abbildung 4.9 ist der Suchbaum zu diesem Programm dargestellt. Seine Knoten sind wieder in alphabetischer Reihenfolge entsprechend ihres Erreichens markiert. Durch CP konnte der Suchbaum um die gestrichelten Teile reduziert werden. In Knoten b wird eine erste CSP-Lösung gefunden ($X = 1,\ Y = 2,\ Z_1 = 2$). Durch Backtracking wird in Knoten c eine kleinere Lösung für den Zielwert erreicht ($X = 1,\ Y = 3,\ Z_2 = 1$) . In Knoten d wird durch die Wahl von ($X = 2$) die Domäne von Y mit CP reduziert auf $D_Y = \{2\}$, damit liegt auch ($Z_3 = 2$) fest. Da dieser Wert schlechter als Z_2 ist und keine weiteren Möglichkeiten im Suchbaum bestehen, wird als minimale Lösung ($X = 1,\ Y = 2,\ Z = 1$) ausgegeben.

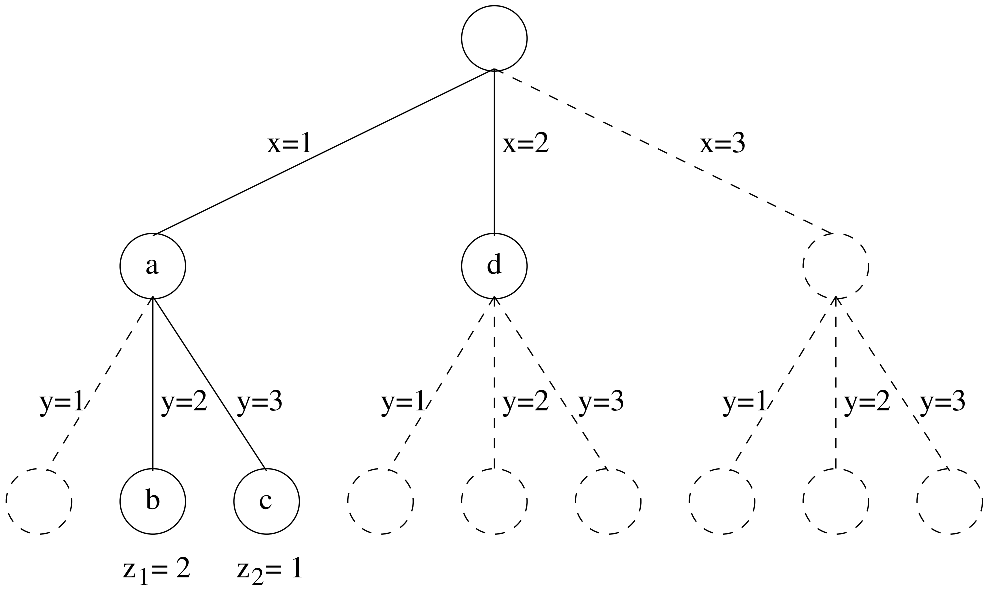


Abbildung 4.9: Suchbaum zu Beispiel 4.16

□

4.2 Übergang von SILP nach SCLP

In diesem Abschnitt wird ein Modell zur Beschreibung des SILP-Modells in CLP formuliert. Dieses CLP-Modell wird mit SCLP bezeichnet. Formal ist der Übergang von SILP nach SCLP eine Abbildung $SILP \rightarrow SCLP$. Ein SCLP-Programm besteht aus einem $CSP_{SCLP} = (V, D_V, C)$ und einer Lösungsstrategie. Das CSP_{SCLP} wird aus den Daten eines SILP-Programmes in drei Schritten erzeugt:

1. **Generierung der Menge der Constraints C-SCLP**

   Das SILP-Modell zur Lösung des Problems der Instruktionsanordnung unter Berücksichtigung von Ressourcenschranken wird durch eine Zielfunktion und eine Reihe von Nebenbedingungen[VII] beschrieben. Bei der Generierung der SILP-Programme durch CIS werden die Nebenbedingungen durch eine Menge von Ungleichungen ausgedrückt. Die Zielfunktion ist dahingehend in die Nebenbedingungen integriert, daß sie durch eine spezielle Variable S ausgedrückt wird, die minimiert werden soll.

Beispiel 4.17 Seien t_X und t_Y zwei Variablen, die den Ausführungszeitpunkt von zwei Instruktionen X und Y bezeichnen. Dann kann durch die Ungleichung (Gleichung 3.14, Seite 42)*:*

$$t_X - t_Y \geq 1$$

eine Ausführungsreihenfolge ausgedrückt werden, die besagt, daß Instruktion X vor Instruktion Y ausgeführt werden muß.

□

Die Ungleichungen in einem SILP-Programm können für ein SCLP-Programm als Constraints interpretiert werden.

Beispiel 4.18 Ein Constraint*:*

$$T_X - T_Y \ \geq 1,$$

würde die gleiche Ausführungsreihenfolge wie in Beispiel 4.17 festlegen.

□

Die Ungleichungen in einem SILP-Programm können durch eine Transformation auf Constraints in einem SCLP-Programm abgebildet werden. Da die Formulierung einer Ungleichung in SILP einem CLP Constraint sehr ähnlich ist, reicht für diese Transformation eine einfache Zeichenersetzung, die von dem Tool SILP2SCLP geleistet wird.

Beispiel 4.19 In der linken Spalte sind einige Codezeilen eines SILP-Programms aufgeführt; ihre CLP Pendanten stehen rechts gegenüber.

SILP Ungleichung		*CLP Constraint*
$t2 - t1 >= 1$	$\rightarrow$	$T2 - T1 \ \# >= 1,$
$p15\_16 + q15\_16 = 1$	$\rightarrow$	$P15\_16 + Q15\_16 \ \# = 1,$
$t3 - tE0 <= 0$	$\rightarrow$	$T3 - TE0 \ \# <= 0,$

Zur Notation: In ECLiPSe müssen Variablen groß geschrieben werden, einzelne Constraints werden durch Kommata getrennt. Mit dem Sonderzeichen # wird dem CLP-System mitgeteilt, daß diese Constraints Anweisungen der CLP-FD sind.

□

[VII] siehe Kapitel 3.2.1, Seite 41

2. **Generierung der Variablenlisten VI, VB, VL und VE**

   Zur Lösung eines CSP müssen die Variablen des SILP-Programms in Listen zusammengefaßt werden. Dazu werden von SILP2SCLP Listen zusammengehöriger Variablen erstellt. Aus diesen Listen werden vier Variablenlisten generiert; diese Verteilung ist für die späteren Optimierungen wichtig. Eine Sonderstellung nimmt die Variable S für den Zielwert ein, sie wird in keine der Listen eingetragen.

   a) **VI** enthält die Liste der Variablen t_i [VIII], welche die relative Position einer Mikrooperation innerhalb der Instruktionen beschreiben.
   b) **VB** enthält die Liste der Basisblockvariablen. Sie beschreiben Start- und Endtaktzyklus eines Basisblocks. Diese Variablen werden in dem von [Käs97] angepaßten SILP-Modell zusätzlich generiert, um eine Zuordnung von Mikrooperationen zu Basisblöcken abzubilden.
   c) **VL** enthält die Liste der Variablen, mit denen die Lebenszeit[IX] eines Wertes in einem Register beschrieben wird.
   d) **VE** enthält die Liste der Entscheidungsvariablen. In dieser Liste werden alle Variablen zusammengefaßt, die nicht in einer der anderen Listen enthalten sind. Der Name resultiert aus der Haupteigenschaft dieser Variablen, Entscheidungen zum Beispiel zur Registerbelegung auszudrücken, indem entsprechende Variablen kennzeichnen, ob ein Register bereits mit einem Wert belegt ist oder nicht. Die Entscheidungsvariablen drücken unter anderem die Kanten des Ressourcenflußgraphen[X] aus und sind Variablen der Flußgleichungen (Gleichung 3.11 und 3.12, Seite 41).

Beispiel 4.20 Aus den Constraints aus Beispiel 4.19

$$\begin{array}{l} T2 - T1 \ \# >= 1, \\ P15\_16 + Q15\_16 \ \# = 1, \\ T3 - TE0 \ \# <= 0, \end{array}$$

werden folgende Variablenlisten generiert:

$$\begin{array}{ll} VI & = [T1, T2, T3], \\ VB & = [TE0], \\ VL & = [\ ], \\ VE & = [P15\_16, Q15\_16], \end{array}$$

□

3. **Generierung der Domänen DZ und DE**

   Der Suchraum eines CSP hängt von der Anzahl der Variablen und der Größe der Domänen ab[XI]. Zur Beschreibung des SCLP-Modells werden deshalb die

[VIII] siehe Kapitel 3.2.1, Seite 38
[IX] siehe Kapitel 3.2.1, Seite 42
[X] siehe Kapitel 3.4, Seite 38
[XI] siehe Kapitel 4.1, Seite 53

Domänen der Variablenlisten möglichst klein gewählt und durch SILP2SCLP in das Modell eingefügt. Für die Variablenlisten VI,VB und VL wird die Domäne $DZ = \{1, \ldots, M\}$ verwendet, M ist dabei die Anzahl an Instruktionen in der seriellen Ausführung. Die Variablen in diesen Listen müssen Werte aus DZ annehmen, im anderen Fall würden Nebenbedingungen des SILP-Modells[XII] verletzt werden. Anschaulich ausgedrückt würde ein größerer Wert für den Ausführungszeitpunkt einer Instruktion, als die maximale Anzahl an Instruktionen in der seriellen Ausführung keine Verbesserung darstellen. Aus ähnlichen Gründen wird eine zweite Domäne $DE = \{0, 1\}$ für die Entscheidungsvariablen eingefügt.

Damit ist der erste Bearbeitungschritt des Tools SILP2SCLP abgeschlossen: Aus einem SILP-Programm wird durch SILP2SCLP ein CSP_{SCLP} erzeugt. Dieses CSP_{SCLP} beschreibt die Aufgabe eines SCLP-Programms. Im zweiten Bearbeitungsschritt wird die Lösungsstrategie für das SCLP-Programm eingefügt und damit eine Datei generiert, die direkt in ECLiPSe eingelesen und gelöst werden kann. Die verschiedenen Lösungsstrategien sind Thema des folgenden Abschnitts.

4.3 Implementierung und Testergebnisse

In diesem Abschnitt werden Strategien zur Lösung der Optimierungsaufgaben entwickelt und ihre Effizienz mittels Testreihen mit den sechs Benchmarks der Gruppe A überprüft. Aus der Gruppe A ergeben sich zwölf Optimierungsaufgaben (je sechs für IS und ISRA), deren Berechnungsaufwand durch zwei Größen grob gekennzeichnet werden kann:

1. Eine Größe ist der zusätzliche Berechnungsaufwand bei der Phasenkopplung[XIII] in ISRA. Die Berücksichtigung der Ressourcenschranken wird über eine Reihe zusätzlicher Nebenbedingungen formuliert. Dadurch erhöht sich die Anzahl der Constraints um das drei- bis fünffache und die Anzahl der Entscheidungsvariablen um das fünf- bis elffache gegenüber IS (siehe Anhang A). Da die Anzahl der Variablen als Exponent in die Größe des Suchraums einfließt (Gleichung 4.1: Größe Suchraum $= |D_V|^{|V|}$), steigt der Berechnungsaufwand für das gleiche Beispielprogramm von IS auf ISRA stark an.

2. Eine zweite Größe ist die Anzahl der Instruktionen. Durch sie kann der Berechnungsaufwand verschiedener Beispielprogramme eingeschätzt werden. Der Berechnungsaufwand kann wie folgt charakterisiert werden:

   - kleine Benchmarks: bis ca. 20 Instruktionen
   - mittelgroße Benchmarks: zwischen 20 und 30 Instruktionen
   - große Benchmarks: zwischen 30 und 50 Instruktionen
   - sehr große Benchmarks: über 50 Instruktionen (für Gruppe B relevant)

[XII] siehe Kapitel 3.2.1, Seite 41
[XIII] siehe Kapitel 3.2, Seite 37

Hinsichtlich dieser Größen werden die Optimierungsaufgaben für die Gruppe A in Tabelle 4.1 in sechs Teilgruppen unterteilt, um die Effizienz einer Lösungsstrategie direkt aus den innerhalb von 24h berechenbaren Teilgruppen abzuleiten.

Gruppe	Beschreibung
IS-A1	IS für FIR und IIR (< 20 Instruktionen)
IS-A2	IS für DFT und Whetstone (20 bis 30 Instruktionen)
IS-A3	IS für histo und conv (> 30 Instruktionen)
ISRA-A1	ISRA für FIR und IIR (< 20 Instruktionen)
ISRA-A2	ISRA für DFT und Whetstone (20 bis 30 Instruktionen)
ISRA-A3	ISRA für histo und conv (> 30 Instruktionen)

Tabelle 4.1: Gruppenzuordnung der Optimierungsaufgaben

Eine Lösungsstrategie ist in erster Linie durch die Auswahl von Variablen gekennzeichnet. Durch die Auswahl der Variablen wird die Lösungssuche, und damit die Berechnungszeit, entscheidend beeinflußt.

Die Auswahl der Variablen erfolgt in CLP-Programmen im Normalfall über die Labeling-Prädikate. In Abschnitt 4.3.1 werden Labeling-Prädikate vorgestellt, die im Rahmen der Diplomarbeit entwickelt und untersucht worden sind. Durch die flexiblen Möglichkeiten der Constraint-logischen Programmierung konnten weitere, an das Optimierungsproblem angepaßte, Lösungsstrategien zur Variablenauswahl und damit zur Lösungssuche entwickelt werden.

Um einen Überblick über die Entwicklung zu geben, werden im folgenden die Lösungsstrategien kurz skizziert. Eine detailierte Beschreibung erfolgt in den Abschnitten 4.3.2 bis 4.3.7.

- Strategie I, die *Standardvariante*: Alle Variablen werden in der Lösungssuche betrachtet. Die Gruppen IS-A1 und IS-A2 sind innerhalb von 24h berechenbar.
- Strategie II, die *Variante der geteilten Variablenmengen* (*VGV*): Die Lösung der Optimierungsaufgabe wird in zwei Schritten erreicht:
  - **Lösungssuche** auf einer Teilmenge der Variablen, und
  - **Verifikation** der Restmenge der Variablen.

  Mit dieser Strategie können die Gruppen IS-A1, IS-A2, ISRA-A1 und eine der Optimierungsaufgaben aus ISRA-A2 innerhalb von 24h gelöst werden.
- Strategie III, die *Variante der geteilten Instruktionsmenge* (*VGI*): Die Lösungssuche auf einer Teilmenge der Variablen aus Strategie II wird auf eine einzelne Variable eingeschränkt. Es können, bis auf einen Benchmark der Gruppe ISRA-A3, alle Optimierungsaufgaben innerhalb weniger Sekunden berechnet werden. Jedoch kann die Optimalität der Lösung nicht garantiert werden; bei 25% der Benchmarks wird eine Instruktion mehr als das Minimum benötigt.

- Strategie IV, die *Variante der M-fachen Lösungssuche* (*VML*); eine Weiterentwicklung der Strategie III: In der Lösungssuche wird weiterhin VGI mit einer Variablen aus der Instruktionsmenge verwendet, jedoch wird mit VGI nacheinander für alle Variablen aus der Menge der Instruktionen eine Lösung berechnet. Auf diese Weise können für alle Optimierungsaufgaben (bis auf einen Benchmark der Gruppe ISRA-A3) optimale Lösungen in wenigen Minuten berechnet werden. Durch den M-fachen Aufruf der Lösungssuche mit verschiedenen Variablen berechnet VML für die untersuchten Benchmarks jeweils mehrere optimale Lösungen. Allerdings kann auch für diese Lösungsstrategie die Optimalität der Lösung nicht garantiert werden.

- Strategie V, die *Lösungsvariante für VML* (*VML+*): Sie basiert auf Strategie IV, jedoch werden durch ein alternatives Labeling-Prädikat (*labelingown*) im Schritt der Lösungssuche kleine Verbesserungen (im Prozentbereich) der Laufzeiten erreicht.

In Abschnitt 4.3.8 sind die Ergebnisse mit der Lösungsvariante für VML (Strategie V) für Programme des erweiterten Benchmarking dargestellt. In Abschnitt 4.3.9 werden zwei weitere Programmbibliotheken von ECLiPSe betrachtet.

4.3.1 Labeling-Prädikate

Mit Labeling-Prädikaten wird, wie in Abschnitt 4.1.6 (Seite 63) beschrieben, die Lösungssuche zum einen durch die Auswahl einer Variablen, und zum anderen durch die Wahl einer Belegung aus der Domäne der Variablen, gesteuert.

Aus der Programmbibliothek der Finite Domain stammt das Prädikat *labeling*. Dieses Prädikat entspricht dem in Abschnitt 4.1.6 verwendeten Prädikat labelingmindomain. Weitere Labeling-Prädikate sind im Rahmen der Arbeit entwickelt worden. Mit ihnen werden alternative Möglichkeiten zur Auswahl einer Variablen untersucht:

- **labeling**: Das Standard Labeling-Prädikat; aus einer Variablenliste wird das erste Element ausgewählt.

- **labelingdel**: Die Variable mit der kleinsten Domäne (den wenigsten Elementen) wird als erstes gewählt.

- **labelingdelc**: Eine Erweiterung von labelingdel; hierbei wird bei Variablen mit gleich großen Domänen zusätzlich die Anzahl der zugehörigen Constraints betrachtet. Bei gleicher Größe der Domänen wird zuerst die Variable mit den meisten Abhängigkeiten gewählt.

- **labelingmin**: Bei dieser Variante wird die Variable als erstes gewählt, die den kleinsten unteren Domänenwert aufweist. Auf diese Weise können oft diejenigen Variablen als erstes belegt werden, die in der optimierten Anordnung am Anfang des Programmes stehen.

- **lablingsplit**: Diese Variante teilt die Variablenmenge in zwei möglichst gleich große Teilmengen und gibt das mittlere Element aus.

- **labelingown**: Eine Erweiterung des Standard Labeling-Prädikats, bei der labeling um einen *Bound Check* erweitert wird. Durch den Bound Check wird in jedem Lösungsschritt eine Kostenanalyse durchgeführt. Auf diese Weise kann teilweise die Suche in einem Zweig (Teilbaum) eher abgebrochen werden.

Ziel der Optimierungsaufgabe ist eine Minimierung der Anzahl benötigter Instruktionszyklen. In einer minimalen Lösung sind die Variablen, mit denen der Ausführungszeitpunkt einer Instruktion beschrieben wird (VI, VB und VL), eher mit den unteren als mit den oberen Werten ihrer Domäne (DZ) belegt, da die Instruktionen in der resultierenden Codesequenz früher angeordnet werden, als in der seriellen Ausgangsreihenfolge. Aus diesem Grund wird in allen Labeling-Prädikaten bei der Auswahl einer Belegung das jeweils erste (kleinste) Element einer Domäne gewählt.

In Abschnitt 4.3.4 werden diese Labeling-Prädikate in die Lösungsstrategie VGV (Strategie II) eingesetzt und getestet.

4.3.2 Standardvariante

In Abschnitt 4.2 ist beschrieben, wie aus einem SILP-Programm eine Beschreibung des Optimierungsproblems für CLP erzeugt wird. Diese Beschreibung (CSP_{SCLP}) besteht aus der Menge der Constraints C-SCLP, den Variablenlisten VI, VB, VL und VE, und den Domänen DZ und DE. Zur Lösung des Optimierungsproblems werden SCLP-Programme verwendet, die aus der Beschreibung des Problems durch CSP_{SCLP} und einer Lösungsstrategie bestehen. Ziel der Optimierung ist eine minimale Anzahl der benötigten Instruktionszyklen. Diese Anzahl wird durch eine Zielvariable S[XIV] ausgedrückt.

In einem ersten Lösungsansatz werden alle Variablenlisten in der Liste $VG = [VI@VB@VL@VE]$[XV] zusammengefaßt und gleichzeitig betrachtet. Zur Lösungssuche wird das Standard Labeling-Prädikat labeling verwendet. M bezeichnet die Anzahl der Instruktionen in der seriellen Ausführung.

Strategie I *Standardvariante*

$$
\begin{array}{ll}
solve(VG, S) :- & \%\ Prädikatname : solve, \\
 & \%\ Variablen : Liste\ VG,\ \ Zielvariable\ S \\
\quad C\!-\!SCLP, & \%\ Menge\ der\ Constraints \\
\quad VG = [VI@VB@VL@VE], & \%\ Variablenliste \\
\quad VI :: [1..M], & \%\ Domäne\ zu\ VI \\
\quad VB :: [1..M], & \%\ Domäne\ zu\ VB \\
\quad VL :: [1..M], & \%\ Domäne\ zu\ VL \\
\quad VE :: [0,1], & \%\ Domäne\ zu\ VE \\
\quad S :: [1..M], & \%\ Domäne\ der\ Zielvariablen \\
 & \\
\quad minimize(labeling(VG), S). & \%\ Such-\ und\ Minimierungsprädikat
\end{array}
$$

[XIV] Maximale Anzahl der benötigten Instruktionszyklen

[XV] Das Sonderzeichen @ beschreibt die Verkettung von zwei Listen.

Das SCLP-Programm solve besteht aus der Beschreibung des CSP_{SCLP} durch die Menge der Constraints C-SCLP, der Variablenliste VG und den Domänen VI, VB, VL und VE. Um den Suchraum weiter einzuschränken, wird die Domäne der Zielvariablen S angegeben. Durch das Suchprädikat labeling werden Lösungen für das CSP berechnet. Die Suche nach einer minimalen Lösung wird über das Minimierungsprädikat und die Zielvariable S gesteuert.

Die Laufzeitergebnisse mit dieser Strategie für die Benchmarks der Gruppe A sind in Tabelle 4.2 zusammengefaßt.

Benchmark	IS	ISRA
FIR	0,06s	> 24h
IIR	0,07s	> 24h
DFT	0,67s	> 24h
Whetstone	268,98s	> 24h
histo	> 24h	> 24h
conv	> 24h	> 24h

Tabelle 4.2: Laufzeitergebnisse mit der Standardvariante (Strategie I)

Mit diesem Lösungsansatz werden die Optimierungsaufgaben der Gruppen IS-A1 und IS-A2 in wenigen Minuten gelöst, jedoch kann keine Optimierungsaufgabe für ISRA innerhalb von 24h berechnet werden.

4.3.3 Variante der geteilten Variablenmengen (VGV)

Einem ersten wichtigen Schritt zur Reduzierung des Suchraums liegt die Überlegung zugrunde, daß von den Variablen zunächst nur eine Teilmenge betrachtet wird. Nachdem eine Lösung für diese Teilmenge gefunden ist (Lösungssuche), werden die restlichen Variablen betrachtet (Verifikation). Wenn es für diese Variablen eine gültige Belegung gibt, ist eine Gesamtlösung gefunden, im anderen Fall wird durch Backtracking die Lösungssuche erneut gestartet.

Die Zerlegung in eine Teilmenge zur Lösungssuche und eine Restmenge zur Verifikation ist aus folgender Überlegung heraus möglich: Gegeben ist das $CSP_{SCLP} = (V, D_V, C)$. Eine minimale Lösung für das CSP_{SCLP} ist eine Anordnung der Instruktionen mit den Eigenschaften, daß die Anzahl der Instruktionszyklen minimal ist, alle Variablen belegt sind und alle Constraints erfüllt werden. Entscheidend ist hierbei, daß sich die minimale Lösungssuche auf die Variablen der Variablenmenge VI beschränken läßt. Für die Belegung der weiteren Variablen muß nur eine beliebige gültige, aber keine minimale Lösung gefunden werden. Aus diesem Grund kann sich die Verifikation auf eine Lösungssuche mit dem Prädikat labeling beschränken und abbrechen, sobald eine Lösung gefunden ist. In der Standardvariante (Strategie I) wird in diesem Fall die Lösungssuche fortgesetzt, da eine minimale Belegung für die Zielvariable S über alle Variablen des CSP berechnet wird.

Die möglichen Anordnungen der Instruktionen können in einem reduzierten $CSP_{VI} = (VI, D_{VI}, C)$, das auf der Variablenmenge VI beruht, dargestellt werden. Eine Lösung für CSP_{VI} ist eine minimale Anordnung der Instruktionen und kann durch einen Teilpfad im Suchbaum zu CSP_{SCLP} abgebildet werden. Zur Verifikation dieser Belegung der Variablen in VI muß ein Pfad im Suchbaum, von diesem Teilpfad ausgehend, zu einer Lösung führen. Wenn ein derartiger Pfad existiert, ist eine Lösung für das CSP_{SCLP} gefunden. Existiert kein derartiger Pfad (für mindestens eine Variable kann keine Belegung berechnet werden, so daß alle Constraints erfüllt sind), wird über Backtracking die Lösungssuche in CSP_{VI} fortgesetzt.

Diese Idee wird in Beispiel 4.21 dargestellt.

Beispiel 4.21 Es sei ein CSP gegeben mit:

$$VI = \{X, Y\},\ VE = \{E1, E2, E3, E4\},\ D_{VI} = \{1, 2\},\ D_{VE} = \{0, 1\},\ C = \{\}.$$

Für dieses CSP gibt es $2^6 = 64$ Lösungsmöglichkeiten. Diese Möglichkeiten können durch obige Überlegungen eingeschränkt werden, da zunächst nur die Kombinationen von X und Y von Interesse sind. Bei der Lösungssuche werden die vier denkbaren Möglichkeiten der Kombination von X und Y $((1,1),(1,2),(2,1),(2,2))$ untersucht und eine minimale (zum Beispiel $(1,1)$) gefunden.

Für diese Möglichkeit erfolgt im zweiten Schritt die Verifikation. Dazu werden die Variablen in VE betrachtet, für die es $2^4 = 16$ Kombinationsmöglichkeiten gibt. Im *worst case* müssen, wie bei der Standardvariante, alle Kombinationen für VI mit allen Kombinationen für VE getestet werden. Im *average case* ist die Lösungssuche im ersten Schritt sehr schnell, und durch die Constraints des CSP ist diese Lösungssuche bereits im Vorfeld soweit eingeschränkt (CP), daß Verifikation gelingt.

Zur Illustration dieser Idee ist in Abbildung 4.10 ein Suchbaum dargestellt. Die eigentliche minimale Lösungssuche beschränkt sich auf die Variablen X und Y; in den durch Dreiecke symbolisierten Bereichen findet die Verifikation der Entscheidungsvariablen VE statt. Ein möglicher Pfad zu einer Lösung für das CSP ist **fett** dargestellt.

Im Gegensatz dazu muß in der Standardvariante (Strategie I) die Lösungssuche auf dem gesamten Suchbaum durchgeführt werden.

□

Durch die Zerlegung wird der Suchraum für die Lösungssuche auf die Menge der Variablen VI reduziert, wodurch eine Lösung des Teilproblem für mehr Benchmarks berechnet werden kann als mit der Standardvariante (Strategie I). Die Belegung der Variablen VI wird im nächsten Bearbeitungsschritt verifiziert. Bei der Verifikation ist es für das Laufzeitverhalten dieser Lösungsstrategie wichtig, einen Fehler in der Belegung von VI so früh wie möglich zu erkennen und in diesem Fall die Lösungssuche in VI fortzusetzen. Nach einer Vielzahl von Versuchsreihen mit einigen der Benchmarks hat sich gezeigt, daß es günstig ist, zuerst die Variablenmengen VB und VL zu betrachten. Die Reihenfolge der Entscheidungvariablen VE ist danach beliebig.

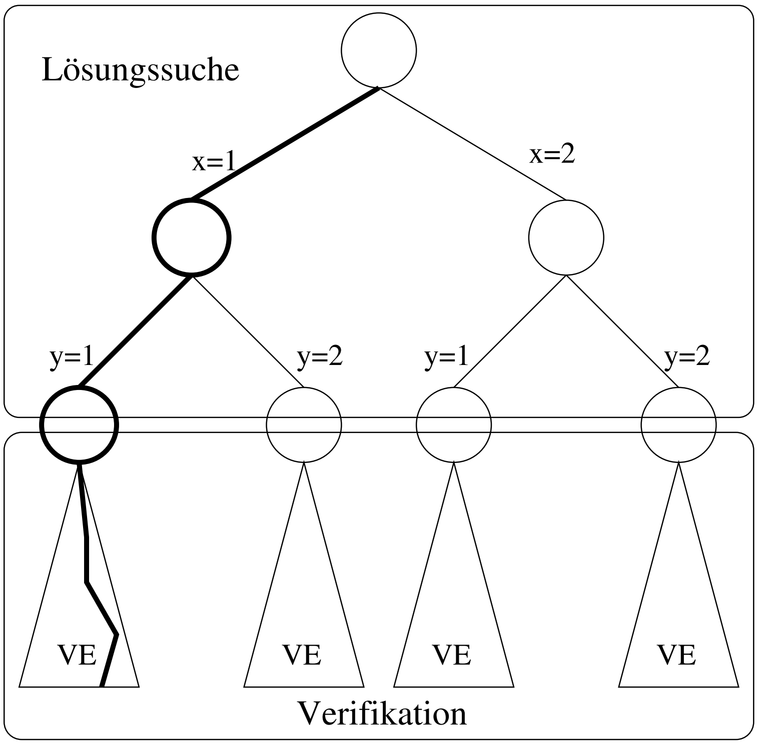


Abbildung 4.10: Suchbaum zu Beispiel 4.21

Aus diesen Überlegungen ergibt sich die Variante der geteilten Variablenmengen:

Strategie II *Variante der geteilten Variablenmengen*

$$
\begin{array}{ll}
\multicolumn{2}{l}{solve(VG, S) : -} \\
\quad C{-}SCLP, & \\
\quad VG = [VI@VB@VL@VE], & \\
\quad VI :: [1..M], & \\
\quad VB :: [1..M], & \\
\quad VL :: [1..M], & \\
\quad VE :: [0, 1], & \\
\quad S :: [1..M], & \\
& \\
\quad minimize(labeling(VI), S), & \%\ minimale\ Lösungssuche\ in\ VI \\
\quad labeling(VB), & \%\ Verifikation\ von\ VB \\
\quad labeling(VL), & \%\ Verifikation\ von\ VL \\
\quad labeling(VE). & \%\ Verifikation\ der \\
& \%\ Entscheidungsvariablen
\end{array}
$$

Durch den Aufruf *minimize(labeling(VI),S)* wird die minimale Lösungssuche auf die Variablenliste VI eingeschränkt. Die Verifikation der Variablenlisten VB, VL und VE findet durch die Labeling-Prädikate *labeling(VB), labeling(VL)* und *labeling(VE)* statt.

Die Laufzeitergebnisse in Tabelle 4.3 demonstrieren die durch den Einsatz dieser Strategie erzielten Verbesserungen. Die Berechnungzeiten der bereits mit der Standardvariante (Strategie I) gelösten Optimierungsprobleme IS-A1 und IS-A2 verbessern sich nur geringfügig. Durch die Aufteilung der Variablenmengen können jedoch zusätzlich auch ISRA-A1 und ein Optimierungsproblem aus ISRA-A2 innerhalb weniger Sekunden berechnet werden.

Benchmark	IS	ISRA
FIR	0,03s	0,54s
IIR	0,05s	0,86s
DFT	0,64s	13,55s
Whetstone	266,06s	>24h
histo	>24h	>24h
conv	>24h	>24h

Tabelle 4.3: Laufzeitergebnisse mit der Variante der geteilten Variablenmengen

Mit dieser Variante können kleine und mittelgroße Optimierungsaufgaben innerhalb weniger Minuten berechnet werden; große Optimierungsaufgaben werden allerdings erst in Tagen oder Wochen berechnet. Als Beispiel für den Berechnungsaufwand der Optimierungsaufgabe bei großen Problemen wurde die Berechnung für IS für den Benchmark histo nicht nach 24h abgebrochen, sondern fortgesetzt. Das Ergebnis der Optimierung lag erst nach über 16 Tagen vor.

Beweisidee: In der Standardvariante (Strategie I) ist die Optimalität einer Lösung dadurch sichergestellt, daß alle Variablen des Problems in der Lösungssuche betrachtet werden. In diesem Fall wird die Korrektheit durch das CLP-Verfahren garantiert. In VGV (Strategie II) werden im ersten Schritt, in der Lösungssuche, nur die Variablen aus VI untersucht. Alle möglichen Lösungen (zur Anordnung der Instruktionen) müssen aber in den Variationen der Anordnungen der Instruktionen enthalten sein. Durch die anschließende Verifikation wird sichergestellt, daß alle Variablen belegt sind und alle Constraints erfüllt werden.

4.3.4 Testergebnisse mit anderen Labeling-Prädikaten

In diesem Abschnitt werden die alternativen Labeling-Prädikate aus Abschnitt 4.3.1 getestet. Die Testreihen werden aus zwei Gründen auf der Basis von VGV (Strategie II) durchgeführt: Zum einen können mit dieser Strategie, im Gegensatz zu einer Testreihe mit der Standardvariante (Strategie I), auch Benchmarks für ISRA berechnet werden. Zum anderen ist diese Lösungsstrategie hinsichtlich der zugrundeliegenden Idee mit den approximativen Lösungsstrategien vergleichbar. Damit sind die Erkenntnisse aus diesen Testreihen auf die suboptimalen Lösungsstrategien übertragbar.

Zur Untersuchung wird das Standard Labeling-Prädikat labeling im Prädikat minimize gegen alternative Prädikate ausgetauscht, da im Schritt der minimalen Lösungssuche der größte Einfluß auf die Effizienz der Lösungsstrategie genommen werden kann. In den Tabellen 4.4 bis 4.6 sind die Berechnungszeiten einiger Probleme mit den verschiedenen Labeling-Prädikaten aufgeführt. DFT wird zur Untersuchung ausgewählt, da er der größte Benchmark ist, für den sowohl eine Lösung für IS als auch für ISRA berechnet werden kann. Der Benchmark Whetstone wird als größtes für IS zu lösendes Optimierungsproblem ausgewählt.

Labeling-Prädikat	Berechnungszeit
labeling	0,64s
labelingdel	>2h
labelingdelc	34,63s
labelingmin	1,13s
labelingsplit	1610,19s
labelingown	0,90s

Tabelle 4.4: Laufzeitergebnisse für IS des Benchmarks DFT mit verschiedenen Labeling-Prädikaten

Labeling-Prozedur	Berechnungszeit
labeling	13,55s
labelingdel	>2h
labelingdelc	430,48s
labelingmin	18,11s
labelingsplit	2897,70s
labelingown	12,78s

Tabelle 4.5: Laufzeitergebnisse für ISRA des Benchmarks DFT mit verschiedenen Labeling-Prädikaten

Labeling-Prozedur	Berechnungszeit
labeling	268,98s
labelingdel	>2h
labelingdelc	>2h
labelingmin	217,13s
labelingsplit	>2h
labelingown	264,98s

Tabelle 4.6: Laufzeitergebnisse für IS des Benchmarks Whetstone mit verschiedenen Labeling-Prädikaten

Die Testergebnisse der alternativen Labeling-Prädikaten labelingmin und labelingown zeigen teilweise eine geringe Verbesserung gegenüber dem Standard Labeling-Prädikat labeling. Ein Grund für die nur geringen Verbesserungen der Berechnungszeiten liegt in der Sortierung der Variablenliste VI: Die Variablen liegen in der Reihenfolge der Instruktionen der seriellen Ausführung vor. Bei der Optimierung bleibt diese Sortierung häufig erhalten; die Instruktionen werden parallel, jedoch nur selten in einer anderen Reihenfolge im Programm angeordnet. Dadurch wird bereits mit dem Standard Labeling-Prädikat labeling eine günstige Reihenfolge bei der Betrachtung der Variablen erreicht. Aufgrund ihrer Laufzeiten werden die Labeling-Prädikate labelingmin und labelingown in der Lösungsvariante für VML noch weiter untersucht.

Auffallend ist der Laufzeitunterschied der Labeling-Prädikate labelingdel und labelingdelc für den Benchmark DFT. Der Unterschied der beiden Strategien bezüglich der Variablenauswahl liegt in der Anzahl der mit einer Variablen verbundenen Constraints. Für diesen Benchmark führt dies bei labelingdelc zu einer starken Reduzierung der Laufzeit gegenüber labelingdel. Dieses Verhalten hängt offensichtlich vom betrachteten Benchmark ab, da für Whetstone ein entsprechendes Verhalten nicht erkennbar ist. Aufgrund der hohen Laufzeiten werden die Labeling-Prädikate labelingdel, labelingdelc und labelingsplit in den weiteren Lösungsstrategien nicht weiter untersucht.

4.3.5 Variante der geteilten Instruktionsmenge (VGI)

In den folgenden Lösungsstrategien VGI, VML und VML+ wird auf die Garantie verzichtet, eine optimale Lösung zu berechnen. Auf diese Weise können die Berechnungszeiten sehr stark reduziert werden. Es werden dennoch Lösungen berechnet, die für die untersuchten Benchmarks nur selten vom Optimum abweichen. Selbst in diesen wenigen Fällen wird nur eine Instruktion mehr als das Optimum benötigt.

In der Variante der geteilten Variablenmengen (Strategie II) ist eine Reduzierung der Berechnungszeiten durch eine Reduzierung der Variablenanzahl im Lösungsschritt erreicht worden. Mit einer weiteren Verfeinerung dieser Idee können fast alle Optimierungsaufgaben in wenigen Sekunden berechnet werden.

In der Variante der geteilten Instruktionsmenge (Strategie III) wird die Idee der Einschränkung des Suchraums im Lösungsschritt dadurch fortgesetzt, daß die Menge der Variablen VI ebenfalls aufgeteilt wird. In Strategie III wird im Lösungsschritt nur noch eine Variable aus VI betrachtet und eine gefundene Lösung durch Belegung der restlichen Variablen verifiziert.

Die Kernidee von VGI wird in Beispiel 4.22 anhand eines Suchbaums demonstriert.

Beispiel 4.22 In Abbildung 4.11 ist ein Beispiel mit drei Variablen dargestellt. Die Domäne aller Variablen ist $D = \{1, 2\}$. Aus der Variablenliste $V = [X, Y, Z]$ wird im Lösungsschritt die erste Variable aus V, die Variable X betrachtet. Es ergeben sich zwei Möglichkeiten zur Belegung für X: $(X = 1)$ und $(X = 2)$.

Die Zielfunktion lautet wie folgt: $S = X + Y + (2 * Z)$. Damit ergibt sich als Domäne von S für die Belegung $(X = 1)$: $D_S = \{1, \ldots, 7\}$ und entsprechend für $(X = 2)$: $D_S = \{2, \ldots, 8\}$. Aus den möglichen Belegungen von X wird über das minimize Prädikat diejenige Belegung gesucht, für die S den kleinsten unteren Wert besitzt, in diesem Fall also $X = 1$. Das minimize Prädikat schränkt auf diese Weise die Domäne von S nur so weit wie nötig ein. Damit kann in weiteren Schritten eine Belegung für die restlichen Variablen berechnet werden.

Der nächste Bearbeitungsschritt ist die Verifikation. Es wird eine Belegung für die Variablen gesucht, mit der alle Constraints erfüllt werden. In den durch Dreiecke symbolisierten Bereichen wird die Verifikation für eine Menge nicht näher spezifizierter Entscheidungsvariablen durchgeführt.

Eine erste Lösung ($S1 = 6$) wird auf dem **fett** markierten Pfad erreicht. An dieser Stelle wird die Optimalität einer Lösung in Strategie III verfehlt. Eine minimale Lösung ($S2 = 5$) könnte, wie dargestellt, in einem anderen Teilbaum unter ($X = 1$) liegen. In dem mit Verifikation bezeichneten Teil des Suchbaums findet jedoch nur eine Lösungssuche statt, d.h. wenn es einen Pfad gibt, auf dem alle Variablen mit gültigen Werten aus ihren Domänen belegt werden, wird die Lösungssuche beendet. In diesem Fall bedeutet dies, daß zwar für die Variablen (X,Y,Z) eine gültige Lösung ($X = 1, Y = 1, Z = 2$) mit ($S = 6$), jedoch nicht die minimale Lösung ($X = 1, Y = 2, Z = 1$) mit ($S = 5$) berechnet wird. Aus diesem Grund kann für die Variante der geteilten Instruktionsmenge (Strategie III) nicht mehr garantiert werden, daß eine optimale Lösung berechnet wird.

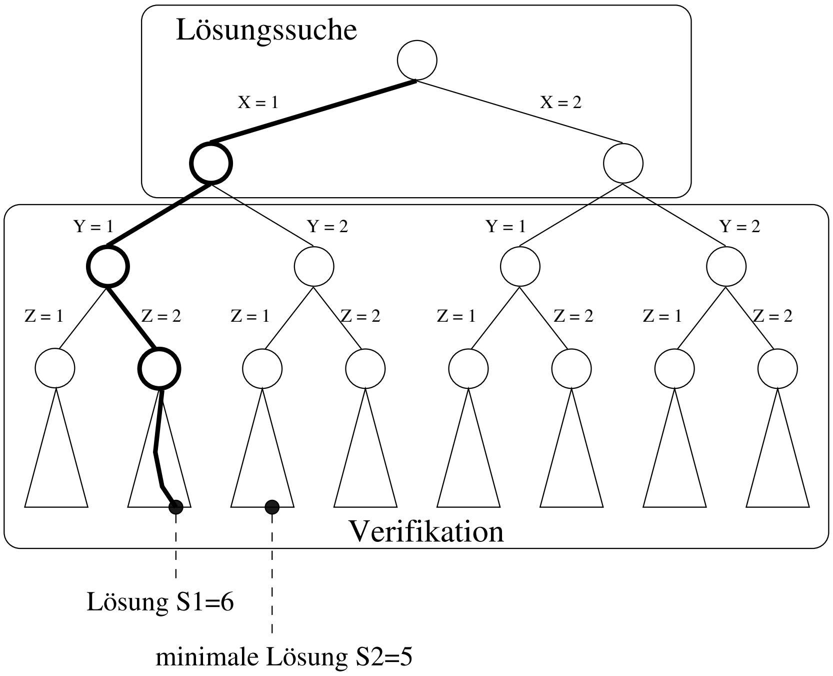


Abbildung 4.11: Ein Suchbaum zu Strategie III

In Vergleich dazu wird in VGV (Strategie II) eine minimale Lösung über alle Variablen aus VI berechnet. Deshalb würde mit VGV ein Backtracking zu dem rechten Teilbaum unter $X = 1$ und damit zu einer minimalen Lösung ($S2$) führen.

□

Zur Umsetzung dieser Idee wird ein Prädikat *split* verwendet, mit dem die Variablenliste VI in eine Variablenliste mit dem ersten Element der Liste $VI1$ und eine Restliste $VIRest$ aufgeteilt wird.

Eine weitere Verbesserung der Berechnungszeiten wird dadurch erreicht, daß eine Sortierung der Variablenliste VI durchgeführt wird. Es hat sich als günstig erwiesen, zuerst diejenigen Instruktionen und damit die mit den Instruktionen verbundenen Variablen aus VI zu betrachten, die in der optimierten Lösung am Anfang des Programms stehen. Diese Instruktionen sind unter den Variablen zu finden, deren Domänen den kleinsten unteren Wert besitzen; alle weiteren Instruktionen können frühestens danach in das Programm eingefügt werden. Aus diesem Grund wird mit einem Prädikat *sort* die Liste der Variablen VI nach ihren kleinsten Domänenwerten sortiert[XVI].

Strategie III *Variante der geteilten Instruktionsmenge*

$solve(VG, S) :-$
$\quad C\!-\!SCLP,$
$\quad VG = [VI@VB@VL@VE],$
$\quad VI :: [1..M],$
$\quad VB :: [1..M],$
$\quad VL :: [1..M],$
$\quad VE :: [0, 1],$
$\quad S :: [1..M],$

$\quad sort(VI, VIsort)$
$\quad \%\ sortiert\ die\ Liste\ VI\ und\ gibt\ VIsort\ aus$

$\quad split(VIsort, VI1, VIsortRest),$
$\quad \%\ Aufspaltung\ von\ VIsort\ in\ VI1\ mit\ der\ ersten\ Variablen$
$\quad \%\ und\ VIsortRest\ mit\ den\ restlichen\ Variablen$

$\quad minimize(labeling(VI1), S), \qquad \%\ minimale\ Lösungssuche\ in\ VI1$
$\quad labeling(VIsortRest), \qquad \%\ Verifikation\ von\ VIRest$
$\quad labeling(VB),$
$\quad labeling(VL),$
$\quad labeling(VE).$

In Tabelle 4.7 sind die Ergebnisse der Lösungsstrategie III aufgeführt. Für drei der Benchmarks wird eine Lösung berechnet, die um eine Instruktion vom Optimum abweicht. Die entsprechenden Berechnungszeiten sind durch Klammern gekennzeichnet. Diese drei Benchmarks sind ein Beispiel dafür, daß mit VGI die Optimalität der Lösung nicht garantiert ist. Für die anderen Benchmarks wird jeweils eine optimale Lösung berechnet.

Die Ergebnisse in Tabelle 4.7 zeigen die Effizienz dieser Lösungsstrategie; bis auf den Benchmark conv können für ISRA alle Optimierungsaufgaben in wenigen Sekunden

[XVI] Die Betrachtung der Variablen mit dem kleinsten unteren Domänenwert ist vergleichbar mit dem Ansatz in anderen Lösungsverfahren, zuerst die Instruktionen mit kleinstem *as soon as possible* (*asap*)-Wert ins Programm einzuordnen.

Benchmark	IS	ISRA
FIR	0,03s	0,40s
IIR	0,03s	0,65s
DFT	(0,08s)	(1,13s)
Whetstone	(0,13s)	11,32s
histo	0,49s	7,29s
conv	0,20s	>24h

Tabelle 4.7: Laufzeitergebnisse mit der Variante der geteilten Instruktionsmenge

berechnet werden. Für IS werden sogar alle Benchmarks in weniger als einer halben Sekunde berechnet.

Für VGI ergibt sich für IS eine mittlere Abweichung vom Optimum (Gleichung 3.15, Seite 49) $M_O = 2,02$ % und und für ISRA $M_O = 1,43$ %.

4.3.6 Variante der M-fachen Lösungssuche (VML)

Die Kernidee in VGI (Strategie III) besteht darin, daß in der minimalen Lösungssuche nur noch eine Variable betrachtet wird. VGI zeigt ein sehr gutes Laufzeitverhalten, jedoch ist der Suchbaum unzulässig eingeschränkt und es werden teilweise suboptimale Lösungen berechnet. In der Variante der M-fachen Lösungssuche (Strategie IV) wird die Effizienz von VGI ausgenutzt und die Strategie III dahingehend erweitert, daß die minimale Lösungssuche nacheinander für jede Variable aus VI mit VGI durchgeführt wird. Auf diese Weise kann zwar die Optimalität weiterhin nicht garantiert werden, dennoch werden für alle untersuchten Benchmarks minimale Lösungen berechnet.

Der Suchbaum wird, wie in Abbildung 4.12 dargestellt, in M-Teilbäume (entsprechend der Anzahl der Variablen VI) zerlegt. Ein derartiger Teilbaum entspricht einem Suchbaum in Strategie III. Von VML werden diese Teilbäume nacheinander untersucht und auf diese Weise im Schritt der Lösungssuche die vollständige Liste der Variablen aus VI betrachtet. Indem für die einzelnen minimalen Lösungssuchen in den Teilbäumen VI_j ($j \in \{1, \ldots, M\}$) jeweils nur eine Variable betrachtet wird, kann das gute Laufzeitverhalten aus Strategie III ausgenutzt werden. Gleichzeitig werden durch die verschiedenen Teilbäume weitere Kombinationen der Belegungen der Variablen VI erreicht. Eine derartige Kombination kennzeichnet einen der Knoten, unter denen die Entscheidungsvariablen verifiziert werden (siehe Abbildung 4.11). Können weitere dieser Knoten erreicht werden, steigt die Wahrscheinlichkeit einen Knoten zu erreichen, der auf dem Pfad zu einer minimalen Lösung liegt. Dadurch, daß in VML verschiedene Variablen in der minimalen Lösungssuche betrachtet werden, ergeben sich verschiedene Belegungen für diese Variablen und damit auch unterschiedliche Kombinationen der Belegungen von VI.

Auf diese Weise können im average case mehrere minimale Lösungen berechnet werden.

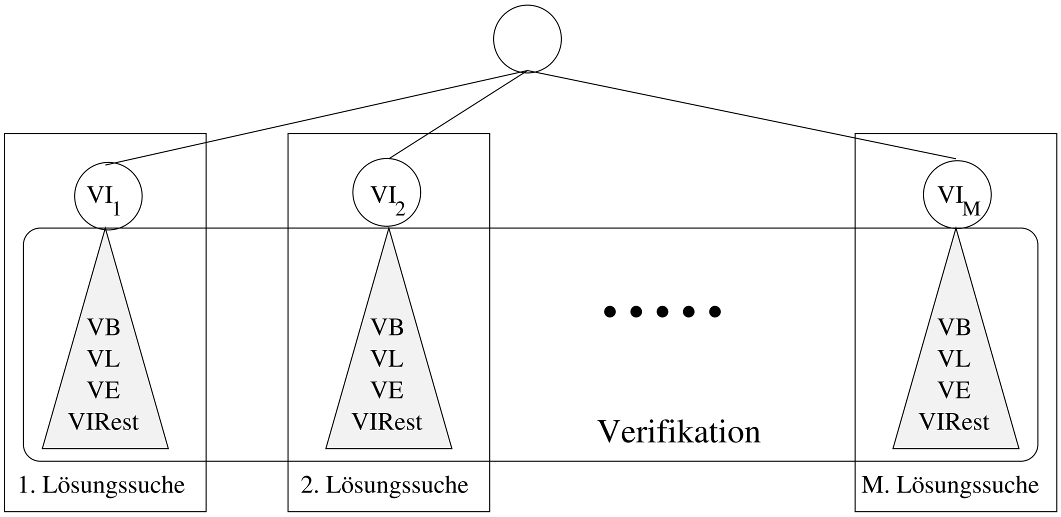


Abbildung 4.12: Allgemeiner Suchbaum mit Teilbäumen

Zum M-fachen Aufruf von VGI wird in VML ein Prädikat *solveges* verwendet, das nacheinander alle Variablen aus $VI = [VI_1, \ldots, VI_M]$ in den Lösungsschritt von VGI einsetzt und die Lösungssuche startet. Eine Lösung für den Zielwert S wird als Startwert der nächsten Lösungssuche übergeben. Auf diese Weise kann der Suchraum für die folgenden Teilbäume weiter eingeschränkt werden. Im folgenden ist eine funktionale Beschreibung von solveges angegeben:

$$
\begin{array}{ll}
solveges(VG, S) :- & \\
\quad FOR\ I = 1\ TO\ M\ \{ & \%\ M-facher\ Aufruf \\
\quad\quad solve(S, I, VG, Sneu); & \%\ des\ Prädikats\ solve \\
\quad\quad IF\ Sneu < S\ THEN\ S := Sneu & \%\ eine\ kleinere\ Lösung\ für\ S\ wird \\
 & \%\ neuer\ Startwert\ oder\ Ergebnis \\
\quad\quad \} &
\end{array}
$$

Durch ein weiteres Prädikat *choice* wird in dem SCLP-Programm solve zum einen aus der Liste der Variablen VI die jeweils aktuell zu untersuchende Variable VII bestimmt, zum anderen werden die restlichen Variablen in $VIRest$ ausgegeben. Es ersetzt das Prädikat split aus Strategie III.

Um eine Lösung als Startwert für die Lösungssuche im folgenden Teilbaum zu verwenden, wird ein erweitertes minimize Prädikat der Finite Domain verwendet. Diesem Prädikat wird eine gefundene Lösung $Salt$ als Startwert (obere Grenze) von S für die neue Suche übergeben; auf diese Weise kann die Lösungssuche bei einer „ungünstigen“ Variablen VI_j ($j \in \{1, \ldots, M\}$) schnell abgebrochen werden.

Strategie IV *Variante der M-fachen Lösungssuche*

$solve(Salt, I, VG, S) : -$
$C\!-\!SCLP,$
$VG = [VI@VB@VL@VE],$
$VI :: [1..M],$
$VB :: [1..M],$
$VL :: [1..M],$
$VE :: [0,1],$
$S :: [1..M],$
$sort(VI, VIsort)$

$choice(VIsort, I, VII, VIrest),$
$\%\ Aufspaltung\ von\ VIsort\ in\ VII\ mit\ einer\ Variablen$
$\%\ und\ VIrest\ mit\ den\ restlichen\ Variablen\ aus\ VIsort$

$minimize(labeling(VII), S, Salt),\quad \%\ Lösungssuche\ in\ VII$

$labeling(VIrest),\quad \%\ Verifikation\ von\ VIrest$
$labeling(VB),$
$labeling(VL),$
$labeling(VE).$

In Tabelle 4.8 sind die Berechnungszeiten mit dieser Lösungsstrategie aufgeführt.

Benchmark	IS	ISRA
FIR	0,33s	4,44s
IIR	0,51s	10,50s
DFT	4,14s	54,01s
Whetstone	17,79s	300,80s
histo	19,23s	200,76s
conv	46,51s	>24h

Tabelle 4.8: Laufzeitergebnisse mit der Variante der M-fachen Lösungssuche

Der zusätzliche Berechnungsaufwand in VML (Strategie IV) durch den M-fachen Aufruf von VGI führt dazu, daß die Laufzeitergebnisse im Vergleich zu VGI (Strategie III) relativ gesehen deutlich schlechter werden. Zum Beispiel unterscheidet sich für IS die Berechnungszeit von VGI und VML für conv um den Faktor 232. Die Optimierungsaufgaben werden für VML im Minuten- und nicht im Sekundenbereich wie bei VGI gelöst. Absolut gesehen sind jedoch die Berechnungen mit VML sehr effizient, da bis auf ISRA für den Benchmark conv alle Optimierungsaufgaben innerhalb von ca. fünf Minuten gelöst werden können.

Die wichtigste Vorteile von VML sind zum einen, daß für alle Benchmarks optimale Lösungen berechnet werden. Damit ist die mittlere Abweichung vom Optimum M_O (Gleichung 3.15, Seite 49) geringer als bei VGI und beträgt für IS und ISRA bei VML: $M_O = O\ \%$.

Zum anderen werden durch die M-fache Lösungssuche nicht nur eine, sondern direkt mehrere minimale Lösungen berechnet. Je nach Größe des Benchmarks werden mindestens eine, im Durchschnitt jedoch fünf bis acht, zumeist optimale, Anordnungen ermittelt. Aus diesen verschiedenen Lösungen kann in weiteren Optimierungen in einem größeren Programmkontext eine „günstige“ verwendet werden.

4.3.7 Lösungsvariante für VML (VML+)

Die Berechnungszeiten können in der Lösungsvariante für VML (*VML+*) (Strategie V) durch Anwendung der Erkenntnisse aus den Versuchen mit alternativen Labeling-Prädikaten (Abschnitt 4.1.6) teilweise noch etwas verbessert werden. In Testreihen mit einigen Benchmarks der Gruppe A hat sich gezeigt, daß für den ersten Lösungsschritt, der Suche nach einer Lösung für eine Instruktionsvariable (VII), und für die ersten Verifikationen für die Menge der restlichen Variablen aus VI ($VIrest$), das Labeling-Prädikat labelingown am effizientesten ist. Für die Variablenmengen VB, VL und VE sind mit den alternativen Labeling-Strategien keine Laufzeitverbesserungen möglich.

Damit ergibt sich als Lösungsvariante für VML:

Strategie V *Lösungsvariante für VML*

$$
\begin{array}{ll}
\multicolumn{2}{l}{solve(Salt, I, VG, S) :-} \\
\quad C{-}SCLP, & \\
\quad VG = [VI@VB@VL@VE], & \\
\quad VI :: [1..M], & \\
\quad VB :: [1..M], & \\
\quad VL :: [1..M], & \\
\quad VE :: [0, 1], & \\
\quad S :: [1..M], & \\
\quad sort(VI, VIsort) & \\
\quad choice(VIsort, I, VII, VIrest), & \\
\quad minimize(labelingown(VII), S, Salt), & \%\ \textit{Lösungssuche mit labelingown} \\
\quad labelingown(VIrest), & \%\ Verifikation\ von\ VIrest\ mit\ labelingown \\
\quad labeling(VB), & \\
\quad labeling(VL), & \\
\quad labeling(VE). &
\end{array}
$$

In Tabelle 4.9 sind die Berechnungszeiten für die Benchmarks der Gruppe A mit VML+ (Strategie V) aufgeführt. Die Laufzeiten sind geringfügig besser als mit VML. Über die Labeling-Prädikate kann in dieser Lösungsstrategie nur wenig Einfluß auf die Variablenauswahl und damit auf die Lösungssuche genommen werden. Deshalb sind auch nur geringe Verbesserungen in den Berechnungszeiten möglich.

Mit der Lösungsstrategie VML+ können, bis auf ISRA für den Benchmark conv, alle aufgeführten Optimierungsprobleme in weniger als fünf Minuten gelöst werden. Es werden jeweils optimale Lösungen berechnet, damit ist auch für VML+: $M_O = 0\ \%$.

Benchmark	IS	ISRA
FIR	0,32s	4,42s
IIR	0,55s	10,85s
DFT	4,22s	50,50s
Whetstone	18,20s	296,48s
histo	20,74s	194,30s
conv	45,73s	>24h

Tabelle 4.9: Laufzeitergebnisse mit VML+ (Strategie V)

4.3.8 Testergebnisse des erweiterten Benchmarking

In den folgenden Tabellen 4.10 bis 4.12 sind die Laufzeitergebnisse für die Benchmarks der Gruppe B unter Verwendung der Lösungsstrategien VGV, VGI und VLM+[XVII] aufgeführt. Bei der Untersuchung der suboptimalen Lösungsstrategien (Strategie III bis V) hat sich gezeigt, daß für fast alle Benchmarks aus Gruppe A eine Lösung in wenigen Minuten berechnet werden kann. Da jedoch auch diese Lösungsstrategien ein exponentiell anwachsendes Laufzeitverhalten besitzen, stellt sich die Frage, bis zu welcher Programmgröße eine effiziente Berechnung möglich ist. Diese Grenze konnte für IS nicht ermittelt werden. Durch einen nicht genauer zu lokalisierenden Fehler in einem der verwendeten Tools konnten keine Benchmarks mit mehr als 99 Instruktionen fehlerfrei generiert werden. Es liegt die Vermutung nahe, daß beim Wechsel auf dreistellige Werte für die Instruktionen entweder ein Zähler überläuft oder Variablenbezeichnungen nicht mehr eindeutig sind. Aus diesem Grund ist mit 93 Instruktionen n_comple der größte untersuchte Benchmark. Die Grenze einer effizienten Berechnung für ISRA liegt bei den Benchmarks aus Gruppe A bei etwa 50 Instruktionen, da für den Benchmarks conv mit 49 Instruktionen keine Lösung berechnet werden konnte. Eine Bestätigung für diese Grenze zeigt sich bei den Berechnungen mit den Benchmarks der Gruppe B, auch hier können die sehr großen Benchmarks für ISRA nicht gelöst werden.

Zur Vollständigkeit sind in Tabelle 4.10 die Berechnungszeiten für die exakte Lösungsstrategie VGV (Strategie II) angegeben. Aufgrund der Größe der Beispielprogramme können nur für die ersten beiden Benchmarks biquad_o und complex_mul optimale Lösungen innerhalb von 24h berechnet werden. Für IS werden die ersten beiden Benchmarks in etwa einer halben Stunde bzw. 14 Stunden berechnet. Für ISRA ist nur für einen Benchmark (biquad_o) eine Lösung innerhalb von 24h möglich.

Benchmark	IS	ISRA
biquad_o	2249,43s	4363,01s
complex_mul	51879.19s	>24h
lattice	>24h	>24h
n_comple	>24h	>24h

Tabelle 4.10: Laufzeiten mit VGV (Strategie II) für Benchmarks der Gruppe B

[XVII] VML wird als Vorgänger von VML+ hier nicht weiter untersucht.

In der folgenden Tabelle 4.11 sind die Laufzeiten für die suboptimale Lösungsstrategie VGI (Strategie III) aufgeführt. Für IS werden mit dieser Strategie für alle Benchmarks innerhalb von zwei Sekunden Lösungen berechnet. Für die ersten beiden Benchmarks biquad_o und complex_mul ist dabei die Anzahl der Instruktionen minimal. Da für die Benchmarks lattice und n_comple keine optimale Anordnung ermittelt werden konnte, ist auch keine genaue Aussage über die Codequalität der berechneten approximativen Lösungen möglich. Jedoch werden für lattice mit 64 Instruktionen und n_comple mit 85 Instruktionen die gleichen Werte wie mit VML+ ermittelt. Da bei den bisherigen Benchmarks mit VML+ immer eine optimale Lösung berechnet worden ist, ist zu vermuten, daß auch die Lösungen für lattice und n_comple minimal sind. Unter dieser Annahme kann die Aussage getroffen werden, daß VGI für die Benchmarks der Gruppe B jeweils minimale Lösungen berechnet. Für ISRA werden optimale Lösungen für die ersten beiden Benchmarks mit 27 (biquad_o) bzw. 32 (complex_mul) Instruktionen in einer Sekunde bzw. in wenigen Minuten berechnet. Da für die beiden letzten Benchmarks lattice und n_comple mit 87 bzw. 93 Instruktionen keine Lösungen innerhalb von 24h berechnet werden, bestätigt sich die Annahme, daß die Grenze einer Lösungsberechnung für ISRA bei etwa 50 Instruktionen liegt.

Benchmark	IS	ISRA
biquad_o	0,18s	0,57s
complex_mul	0,46s	314,50s
lattice	1,50s	>24h
n_comple	1,91s	>24h

Tabelle 4.11: Laufzeiten mit VGI (Strategie III) für Benchmarks der Gruppe B

Mit der Lösungsstrategie VML+ (Strategie V) zeichnet sich für die Gruppe B ein vergleichbares Bild wie für Gruppe A ab. Die Lösungsstrategien VGI und VML+ unterscheiden sich durch eine etwas höhere Laufzeit für VML+ voneinander, jedoch kann jeweils für die selben Benchmarks eine Lösung berechnet werden. In Tabelle 4.12 ist zu erkennen, daß für IS Lösungen für alle Benchmarks in weniger als fünf Minuten berechnet werden. Im Gegensatz dazu wird, bei gleicher Codequalität, eine Berechnung mit VGI in weniger als zwei Sekunden durchgeführt. Von den Optimierungsaufgaben für ISRA können ebenfalls die ersten beiden Benchmarks berechnet werden. Für die beiden letzten sehr großen Benchmarks wird keine minimale Lösung innerhalb von 24h ermittelt.

Benchmark	IS	ISRA
biquad_o	5,15s	15,56s
complex_mul	11,12s	3194,01s
lattice	256,47s	>24h
n_comple	109,68s	>24h

Tabelle 4.12: Laufzeiten mit VML+ (Strategie V) für Benchmarks der Gruppe B

4.3.9 Berechnungen mit anderen Bibliotheken

ECLiPSe stellt eine Reihe von Programmbibliotheken mit Prädikaten zur Verfügung, die für unterschiedliche Aufgaben geeignet sind. Die bisherigen Berechnungen sind mit Prädikaten aus der Finite Domain durchgeführt worden. In diesem Abschnitt werden einige Ergebnisse mit weiteren Bibliotheken vorgestellt.

EPLEX

Die Bibliothek *EPLEX* stellt eine Verbindung zwischen ECLiPSe und einem externen Simplex-Solver, wie zum Beispiel CPLEX, her. Über ein Interface können Constraints von ECLiPSe zur Lösung an einen CPLEX-Solver übergeben werden. Über ein Prädikat *optimize* kann, ähnlich dem Prädikat minimize in der Finite Domain, eine optimale Lösung berechnet werden.

Zur Berechnung mit EPLEX müssen nur wenige Änderungen am SCLP-Modell vorgenommen werden. Ein SCLP-Programm mit Strategie I dient als Grundlage. Darin wird das minimize gegen das optimize Prädikat ausgetauscht und einige Anpassungen (Zeichenersetzungen) bei den Constraints vorgenommen. Bei der Bearbeitung mit EPLEX werden die Daten nicht an einen externen Solver übergeben; die entsprechenden Lösungsstrategien sind in ECLiPSe implementiert. Zur Benutzung der Funktionalität eines CPLEX-Solvers wird eine Prüfung auf eine gültige CPLEX-Lizenz im System durchgeführt. Bei Erfolg können die entsprechenden Funktionen des ECLiPSe-Systems genutzt werden.

Die Berechnungszeiten mit der EPLEX-Bibliothek sind mit denen der reinen CPLEX Bearbeitung vergleichbar. Auch mit weiteren angebotenen Prädikaten zur Steuerung der Suche konnten nur geringe Verbesserungen erzielt werden. Aus diesen Gründen wurde die Bibliothek EPLEX nicht für die Berechnungen in dieser Arbeit verwendet.

FDPLEX

Eine weitere Bibliothek ist die *FDPLEX*. Sie erweitert die Finite Domain Bibliothek um einen Simplex-Solver. Eine Reduzierung des Suchraums wird vom Finite Domain Solver durch CP geleistet, die minimale Lösungssuche wird über einen Simplex-Solver gesteuert.

Zur Berechnung der SCLP-Programme mit FDPLEX sind auch hier nur einige Zeichenersetzungen in den Constraints nötig. Das Prädikat zur minimalen Lösungssuche heißt in der FDPLEX ebenfalls minimize. Einige Ergebnisse der Berechnungen sind in Tabelle 4.13 dargestellt. Mit Strategie V sind auch für diese Möglichkeit die kürzesten Laufzeiten möglich, allerdings sind sie wesentlich größer (um den Faktor drei und mehr) als mit der reinen Finite Domain.

Benchmark	IS	ISRA
IIR	2,01s	25,27s
Whetstone	43,54s	889,23s
histo	101,65s	1023,34s

Tabelle 4.13: Laufzeitergebnisse mit EPLEX

In der Bibliothek werden noch weitere Prädikate zur Steuerung der Suche angeboten, Testreihen mit einigen Benchmarks haben allerdings keine weiteren Verbesserungen ergeben. Deshalb dienen als Referenzdaten des Vergleichs von ILP und CLP die Berechnungszeiten der Finite Domain.

Kapitel 5

Vergleich von ILP und CLP

In diesem Kapitel werden die Lösungsstrategien und die ermittelten Ergebnisse für das ILP-Verfahren aus Kapitel 3 und das CLP-Verfahren aus Kapitel 4 gegenübergestellt und verglichen. In Abschnitt 5.1 werden zunächst die wichtigsten Eigenschaften der beiden Verfahren zusammengefaßt. In den Abschnitten 5.2 und 5.3 werden die Ergebnisse der exakten und approximativen Lösungsstrategien betrachtet und einige Besonderheiten der betrachteten Benchmarks erläutert. Ein Vergleich mit konventionellen, graphbasierten Algorithmen schließt sich in Abschnitt 5.4 an. Eine Gesamtbewertung der Verfahren wird in Abschnitt 5.5 vorgenommen; hier werden Möglichkeiten und Beschränkungen der Codegenerierung mit CLP herausgestellt.

5.1 Gegenüberstellung der Verfahren

Die beiden in dieser Arbeit vorgestellten Verfahren ILP und CLP können mit vergleichbaren Modellen Aufgaben der Codegenerierung darstellen. Als gemeinsames Modell wird SILP verwendet. Durch einfache Konvertierungen kann ein SILP-Programm, das ein Optimierungsproblem für ILP beschreibt, in ein SCLP-Programm transformiert werden. Die wesentlichen Unterschiede zwischen den Verfahren ergeben sich aus den verschiedenen Lösungsstrategien.

ILP

Die minimale Lösungssuche eines ILP-Problems mit CPLEX beruht auf einer Branch-and-Bound Methode, die anhand eines Entscheidungsbaums dargestellt werden kann. Ein derartiger Entscheidungsbaum muß vollständig berechnet und gespeichert werden, was zu einem großen Arbeitsspeicherbedarf dieses Verfahrens führt. Außerdem ist es notwendig, für alle Knoten einer Baumebene eine Lösung und damit eine untere Schranke zu berechnen. Die Berechnung von Teilproblemen (einzelne Knoten des Entscheidungsbaums) wird mit einem Simplex-Algorithmus durchgeführt. Eine einzelne Berechnung erfolgt in der Regel, durch das gute Laufzeitverhalten des Simplex-Algorithmus, sehr schnell, jedoch ist die Gesamtlaufzeit durch die Vielzahl der notwendigen Berechnungen sehr hoch. Insgesamt hängt der Berechnungsaufwand sowohl von der Anzahl der Variablen als auch von den Nebenbedingungen ab. Mit steigender Anzahl der Variablen wächst der Entscheidungsbaum und damit die Anzahl der zu berechnenden Teilprobleme; mit der Anzahl der Nebenbedingungen steigt die Laufzeit des Simplex-Algorithmus.

Die Lösungssuche und die Effizienz der Berechnungen des ILP-Verfahrens können vor allem durch eine geeignete Formulierung des ILP-Modells verbessert werden. Die eigentliche Lösungsstrategie mit CPLEX ist hingegen in weiten Teilen festgelegt und kann nur in geringen Maßen über Parametereinstellungen modifiziert werden.

CLP

Die Lösung eines CLP-Problems mit ECLiPSe beruht zum einen auf einer effizienten Methode zur Reduzierung des Suchraums (Constraint Propagation) und einer Suche, die durch Backtracking gesteuert wird. Zum anderen kann die Suche nach einer minimalen Lösung mit Hilfe von Labeling-Strategien flexibel an spezielle Aufgaben angepaßt werden. Mit diesen Möglichkeiten kann eine effiziente Lösungsstrategie für ein CLP-Modell entwickelt werden.

Der Ablauf einer minimalen Lösungssuche kann anhand eines Suchbaums dargestellt werden. In einem vollständigen Suchbaum werden alle Kombinationen der möglichen Variablenbelegungen durch Blätter dargestellt. Die Größe eines Suchbaums ist ein entscheidendes Merkmal für den Berechnungsaufwand des CLP-Verfahrens. Sie ergibt sich aus der Anzahl der Variablen und der Größe der Domänen. Für die Berechnung einer minimalen Lösung wird jedoch nicht der vollständige Suchbaum berechnet, sondern die Lösungssuche durch eine geeignete Lösungsstrategie auf Teilbäume des Suchbaums eingeschränkt. Diese Teilbäume werden durch CP weiter stark verkleinert. Eine größere Anzahl an Constraints führt häufig bei der Bearbeitung durch CP zu einer weiteren Reduzierung des Suchbaums und ist damit im Gegensatz zum ILP-Verfahren eher von Vorteil. Durch diese Reduzierungen muß in der minimalen Lösungssuche nur ein geringer Teil eines vollständigen Suchbaums berechnet werden. Der Arbeitsspeicherbedarf des CLP-Verfahrens ist im Vergleich zur Branch-and-Bound Methode in CPLEX deutlich geringer, denn durch die Struktur (Ordnung) des Suchbaums reicht eine Beschreibung der aktuellen Position im Suchbaum in Zusammenhang mit einigen Daten über bereits erreichte Teilziele zur Steuerung der Lösungssuche aus.

Mit CLP wäre eine eigenständige und vielleicht auch effizientere Beschreibung der Aufgaben der Codegenerierung auf der Basis eines neuentwickelten Modells möglich gewesen. Dieser Ansatz wurde hier nicht weiter verfolgt, da der direkte Vergleich der beiden Verfahren im Vordergrund stand und die Formulierung eines solchen Modells den Rahmen der Diplomarbeit gesprengt hätte. Aus diesen Gründen wurde für beide Verfahren SILP als Modell verwendet und nach Möglichkeiten gesucht, die Berechnungszeiten durch effiziente Lösungsstrategien zu reduzieren.

5.2 Ergebnisse mit exakten Lösungsstrategien

In den Tabellen 5.1 bis 5.4 sind zur Übersicht die Laufzeitergebnisse mit den effizientesten exakten Lösungsstrategien für SILP und SCLP gegenübergestellt. Für SILP sind hierzu die entsprechenden Parameter[I] in CPLEX eingestellt; für SCLP wird

[I] siehe Kapitel 3.3.1, Seite 45

VGV[II] (Strategie II) verwendet. Zusätzlich ist die Anzahl der benötigten Instruktionen in der seriellen und der optimierten parallelen Programmdarstellung angegeben. In der rechten Spalte ist jeweils das aufgerundete Verhältnis der Berechnungszeiten dargestellt.

Benchmark	Anzahl Instruktionen		SILP	SCLP	SCLP : SILP
	seriell	parallel			
FIR	18	8	0,06s	0,03s	1 : 2
IIR	20	7	0,14s	0,05s	1 : 3
DFT	26	20	164,57s	0,65s	1 : 253
Whetstone	26	20	1197,46s	266,06s	1 : 5
histo	43	31	48509,17s	>24h	-
conv	49	17	3770,11s	>24h	-

Tabelle 5.1: Vergleich der Laufzeiten für exakte Lösungen mit SILP und SCLP für IS mit Benchmarks der Gruppe A

In Tabelle 5.1 ist zu erkennen, daß die Berechnung von kleinen bis mittelgroßen Benchmarks mit CLP deutlich schneller als mit ILP ist. Für die großen Benchmarks histo und conv steigt jedoch der Berechnungsaufwand mit CLP sehr stark an, so daß im Gegensatz zu ILP keine Lösungen innerhalb von 24h berechnet werden. Auffallend sind die Berechnungszeiten der Benchmarks DFT und Whetstone. Bei gleicher Instruktionsanzahl nehmen die Laufzeiten bei Whetstone für SILP um ca. das Siebenfache, für SCLP sogar um mehr als das 400fache gegenüber DFT zu. Der Grund für diesen Anstieg ist in der Anzahl der Parallelisierungsverbote bei Whetstone zu sehen. Das Eingabeprogramm enthält viele Instruktionen, deren Parallelisierung explizit untersagt werden muß. Dadurch müssen eine Reihe zusätzlicher Entscheidungsvariablen eingeführt werden, die den Berechnungsaufwand erheblich erhöhen [Käs97]. Gleichzeitig sinkt in diesem Fall die Anzahl der Gleichungen (Constraints). Für SCLP führt dies dazu, daß die Berechnungszeit im Verhältnis wesentlich stärker ansteigt als für SILP. Dieser Benchmark ist ein Beispiel dafür, daß für die Effizienz einer Lösung in CLP das Verhältnis von Variablen zu Constraints in dem Sinne wichtig ist, als daß mit den Constraints der Suchraum der Variablen eingeschränkt werden kann.

Des weiteren fällt für SILP der Unterschied der Berechnungszeiten von conv und histo auf. Der Benchmark conv wird, obwohl er mehr Instruktionen als histo enthält, von SILP in einer wesentlich kürzeren Zeit berechnet. Dieser Unterschied ergibt sich aus der Struktur der Programme. In conv wird nur ein Basisblock ohne Schleifen betrachtet, bei histo tragen zwei Schleifen und vier Basisblöcke zur Komplexität bei. Dadurch werden für conv im SILP-Programm weniger der Variablen benötigt, die von CPLEX in den ersten Lösungsschritten berechnet werden. Auf diese Weise ergibt sich eine günstige Reihenfolge der Variablen im Entscheidungsbaum, die schnell zu einer minimalen Lösung führt.

In Tabelle 5.2 sind die Ergebnisse für exakte Lösungen der Benchmarks der Gruppe A für ISRA von SILP und SCLP gegenübergestellt.

[II] siehe Kapitel 4.3.3, Seite 74

Benchmark	Anzahl Instruktionen seriell	parallel	SILP	SCLP	SCLP : SILP
FIR	18	8	118,07s	0,54s	1 : 219
IIR	20	7	11924,12s	0,86s	1 : 13865
DFT	26	20	64063,91s	13,55s	1 : 4728
Whetstone	26	-	>24h	>24h	-
histo	43	-	>24h	>24h	-
conv	49	-	>24h	>24h	-

Tabelle 5.2: Vergleich der Laufzeiten für exakte Lösungen mit SILP und SCLP für ISRA mit Benchmarks der Gruppe A

Für ISRA kann mit beiden Verfahren nur noch für die beiden kleinen und einen der mittelgroßen Benchmarks eine optimale Lösung berechnet werden. Daß die minimale Anzahl der Instruktionen für IS und ISRA dabei jeweils übereinstimmt, ist eine Eigenschaft des SILP-Modells[III]. Die Laufzeiten für das CLP-Verfahren sind für die ersten drei Benchmarks wesentlich geringer als für ILP. Die größte Differenz ergibt sich beim Benchmarks IIR mit einem Laufzeitunterschied von fast 1 : 14 000. Für die drei weiteren Benchmarks kann mit keinem der beiden Verfahren eine exakte Lösung innerhalb von 24h berechnet werden.

In den Tabellen 5.3 und 5.4 sind die Laufzeitergebnisse für die Programme des erweiterten Benchmarking (Gruppe B) gegenübergestellt. Die ersten beiden Benchmarks biquad_o und complex_mul sind mit 27 und 32 Instruktionen etwa mittelgroß bis groß, während die sehr großen Benchmarks lattice und n_comple mit 87 und 93 Instruktionen fast doppelt soviele Anweisungen besitzen wie das größte Codestück in [Käs97].

Benchmark	Anzahl Instruktionen seriell	parallel	SILP	SCLP	SCLP : SILP
biquad_o	27	22	20114,70s	2249,43s	1 : 9
complex_mul	32	26	40911,70s	51879,90s	1 : 1
lattice	87	-	>24h	>24h	-
n_comple	93	-	>24h	>24h	-

Tabelle 5.3: Vergleich der Laufzeiten für exakte Lösungen mit SILP und SCLP für IS mit Benchmarks der Gruppe B

Für IS können mit SILP für die beiden ersten Benchmarks Lösungen in ca. 5,5h bzw. ca. 11h berechnet werden. Für die beiden sehr großen Benchmarks kann hingegen keine Lösung innerhalb von 24h ermittelt werden. Mit SCLP wird eine minimale Lösung für den Benchmark biquad_o etwa neunmal schneller als mit SILP berechnet. Die Berechnung für complex_mul wird mit beiden Verfahren in annähernd der gleichen Zeit durchgeführt; für lattice und n_comple kann auch mit SCLP keine optimale Anordnung innerhalb von 24h berechnet werden.

[III] siehe Kapitel 3.2, Seite 37

Benchmark	Anzahl Instruktionen seriell	parallel	SILP	SCLP	SCLP : SILP
biquad_o	27	22	42953,22s	4363,01s	1 : 10
complex_mul	32	-	>24h	>24h	-
lattice	87	-	>24h	>24h	-
n_comple	93	-	>24h	>24h	-

Tabelle 5.4: Vergleich der Laufzeiten für exakte Lösungen mit SILP und SCLP für ISRA mit Benchmarks der Gruppe B

Eine exakte Lösung für ISRA kann, wie in Tabelle 5.4 zu erkennen ist, mit beiden Verfahren nur für den kleinsten Benchmark biquad_o der Gruppe B berechnet werden. In diesem Fall wird eine Lösung mit CLP etwa zehnmal so schnell wie mit ILP berechnet. Für die weiteren Benchmarks der Gruppe B wird keine optimale Anordnung innerhalb von 24h ermittelt.

5.3 Ergebnisse mit approximativen Lösungsstrategien

Die Laufzeiten der exakten Lösungsstrategien sind auf dem selben Computer (PC in Dortmund) ermittelt worden und damit direkt vergleichbar. Die Berechnungen mit der approximativen Lösungsstrategie SAIF[IV] aus [Käs97] konnten in Dortmund nicht durchgeführt werden, da eine dafür notwendige Kommunikation zwischen CIS und CPLEX nicht initiiert werden konnte[V]. Es ist anzunehmen, daß der Grund hierfür in dem Versionswechsel von CPLEX 4.0 (Saarbrücken) auf CPLEX 6.0 (Dortmund) und den damit verbundenen Änderungen in der Schnittstellenbeschreibung liegt. Da auch in Zusammenarbeit mit Daniel Kästner dieses Problem nicht behoben werden konnte, werden für die Approximation SAIF die Ergebnisse aus [Käs97] verwendet.

Ein Vergleich der Berechnungszeiten aus [Käs97] mit den in Dortmund durchgeführten Berechnungen für die approximativen Lösungsstrategien ist insofern möglich, als daß für ILP ein Vergleich der Laufzeiten für die exakten Lösungsstrategien (siehe Tabelle 3.8 und 3.9, Seite 48) zeigt, daß die Berechnungszeiten zumindestens in vergleichbaren Zeitbereichen liegen.

In den folgenden Tabellen 5.5 und 5.6 sind die Berechnungszeiten mit der effizientesten Approximation aus [Käs97] SAIF den Laufzeiten mit VGI (Strategie III) und VML+[VI] (Strategie V) gegenübergestellt. In den beiden rechten Spalten sind jeweils die aufgerundeten Verhältnisse der Berechnungszeiten dargestellt.

Die Berechnungszeiten mit den approximativen Lösungsstrategien sind deutlich geringer als mit den exakten Lösungsstrategien. Wie in Tabelle 5.5 zu erkennen ist, können alle Benchmarks aus IS in wenigen Sekunden bis knapp einer Stunde (SAIF)

[IV] siehe Kapitel 2, Seite 44
[V] siehe Kapitel 3.3.3, Seite 47
[VI] siehe Kapitel 4.3.5, Seite 79 und Kapitel 4.3.7, Seite 85

Benchmark	SAIF	VGI	VML+	VGI : SAIF	VML+ : SAIF
FIR	-	0,03s	0,32s	-	-
IIR	0,27s	0,03s	0,55s	1 : 9	2 : 1
DFT	42,58s	0,08s	4,22s	1 : 53	1 : 10
Whetstone	85,86s	0,13s	18,20s	1 : 660	1 : 5
histo	3720,00s	0,49s	20,74s	1 : 7592	1 : 180
conv	53,66s	0,20s	45,73s	1 : 268	1 : 1

Tabelle 5.5: Vergleich der Laufzeiten für approximative Lösungen für IS mit Benchmarks der Gruppe A

berechnet werden. Alle Optimierungsprobleme werden mit VGI in weniger als einer halben Sekunde gelöst. Auch wenn aus den oben genannten Gründen ein direkter Vergleich der Laufzeiten nicht möglich ist, so werden die Berechnungen mit VGI dennoch deutlich (bis zu mehrere tausendmal) schneller als mit SAIF durchgeführt. Die Laufzeiten steigen zwar für VML+ gegenüber VGI relativ gesehen stark an, dennoch können mit VML+ Lösungen für alle Benchmarks innerhalb von einer Minute ermittelt werden.

Benchmark	SAIF	VGI	VML+	VGI : SAIF	VML+ : SAIF
FIR	19,58s	0,40s	4,44s	1 : 49	1 : 4
IIR	86,72s	0,65s	10,50s	1 : 133	1 : 8
DFT	560,00s	1,13s	54,01s	1 : 496	1 : 10
Whetstone	-	13,32s	300,80s	-	-
histo	-	7,29s	200,76s	-	-
conv	-	-	-	-	-

Tabelle 5.6: Vergleich der Laufzeiten für approximative Lösungen für ISRA mit Benchmarks der Gruppe A

Der beim Wechsel von IS auf ISRA stark ansteigende Berechnungsaufwand zeigt sich in Tabelle 5.6 auch bei den approximativen Lösungsstrategien deutlich. Mit SAIF werden Lösungen für die ersten drei Benchmarks in wenigen Sekunden bis Minuten berechnet, für die letzten drei steigt sowohl der Berechnungsaufwand, als auch der Arbeitsspeicherbedarf so stark an, daß in [Käs97] keine Lösungen berechnet wurden. Sowohl mit VGI, als auch mit VML+, kann bis auf conv für alle Benchmarks eine Lösung in wenigen Sekunden bis knapp fünf Minuten berechnet werden. Auch bei den approximativen Lösungsstrategien zeigt sich ein starker Anstieg der Berechnungszeiten von DFT auf Whetstone. Für ISRA ist die Berechnung für Whetstone sogar länger als für den wesentlich größeren Benchmark histo. Diese Ergebnisse sind ein Beleg dafür, daß mit zunehmender Anzahl der Variablen die Berechnungszeiten für CLP stark ansteigen. Gleichzeitig können eine „geringe“ Anzahl an Constraints zu einem weiteren Anstieg führen, während eine „große“ Anzahl an Constraints zu einer Reduzierung des Suchraums und damit der Berechnungszeit beitragen. In Tabelle 5.7 sind zur Verdeutlichung dieser Eigenschaften die Berechnungszeiten mit

VGI und die Variablen- bzw. Constraintanzahl[VII] von einigen Benchmarks für ISRA aufgeführt.

Benchmark	Laufzeit	Variablen	Constraints
DFT	1,13s	1209	2151
Whetstone	13,32s	1623	1408
histo	7,29s	2799	3619
conv	>24h	6619	17272

Tabelle 5.7: Vergleich der Benchmarks DFT, Whetstone, histo und conv

Ein zentrales Kennzeichen für der Berechnungsaufwand ist die Anzahl der Variablen. Die Ergebnisse deuten auf einen exponentiellen Zusammenhang zwischen der Anzahl der Variablen und den entsprechenden Berechnungszeiten hin. Infolge dessen steigt die Laufzeit bei conv so stark an, daß eine Lösung innerhalb von 24h nicht berechnet werden kann. Der Einfluß der Constraints wird bei den Benchmarks Whetstone und histo besonders deutlich. Obwohl in histo fast doppelt so viele Variablen betrachtet werden, sinkt die Laufzeit gegenüber Whetstone. Es ist anzunehmen, daß durch die große Menge an Constraints für histo über CP eine starke Reduzierung des Suchraums erfolgt.

In den Tabellen 5.8 und 5.9 ist die Anzahl der benötigten Instruktionen für die approximativen Lösungsstrategien aufgeführt. Über die mittlere Abweichung vom Optimum M_O (Gleichung 3.15, Seite 49) kann eine Aussage über die Codequalität der Lösungsstrategien getroffen werden.

Benchmark	SAIF	VGI	VML+	Optimum
FIR	-	8	8	8
IIR	7	7	7	7
DFT	14	15	14	14
Whetstone	21	21	20	20
histo	31	31	31	31
conv	17	17	17	17
mittlere Abweichung vom Optimum M_O	0,83%[VIII]	2,02%	0%	-

Tabelle 5.8: Vergleich der Anzahl der Instruktionen für approximative Lösungen für IS mit Benchmarks der Gruppe A

Mit allen drei Approximationen wird eine sehr gute Codequalität erreicht. Nur für wenige Benchmarks wird eine suboptimale Lösung berechnet, und selbst in diesen Fällen wird nur eine Instruktion mehr als notwendig verwendet. Mit VML+ wird für alle Benchmarks eine minimale Anordnung berechnet; die mittlere Abweichung bei den beiden anderen Approximationen liegt maximal bei etwa 2%.

[VII] siehe Tabelle A.2, Seite 108

[VIII] Der Wert wurde unter der Annahme berechnet, daß für den Benchmark FIR eine optimale Lösung berechnet werden kann, auch wenn dieses Ergebnis in [Käs97] nicht angegeben ist.

Benchmark	SAIF	VGI	VML+	Optimum
FIR	8	8	8	8
IIR	7	7	7	7
DFT	14	15	14	14
Whetstone	-	20	20	20
histo	-	31	31	31
conv	-	-	-	-
mittlere Abweichung vom Optimum M_O	0%	1,43%	0%	-

Tabelle 5.9: Vergleich der Anzahl der Instruktionen für approximative Lösungen für ISRA mit Benchmarks der Gruppe A

Da aus den genannten Gründen die Laufzeiten für die Beispielprogramme des erweiterten Benchmarking für die approximative Lösungsstrategie aus [Käs97] nicht ermittelt werden konnten, entfällt an dieser Stelle der Vergleich der approximativen Lösungsstrategien für die Benchmarks der Gruppe B. Die in Dortmund ermittelten Ergebnisse mit Gruppe B gestatten jedoch, eine genauere Aussage über die Lösungsstrategien VGI und VML+ vorzunehmen. Die entsprechenden Ergebnisse sind in den Tabellen 4.11 und 4.12 (Seite 87) aufgeführt. Die guten Eigenschaften der beiden Strategien bezüglich der Laufzeiten und der erreichten Codequalität bestätigen sich auch bei den Benchmarks der Gruppe B. Für IS können mit VGI sogar Beispielprogramme mit bis zu 93 Instruktionen in weniger als zwei Sekunden gelöst werden; VML+ löst die entsprechenden Benchmarks in weniger als fünf Minuten. Für ISRA bestätigt sich mit beiden Lösungsstrategien die Grenze von 50 Instruktionen, über die hinaus eine effiziente Bearbeitung der Programme nicht mehr möglich ist. Die Codequalität kann nur für zwei der vier Beispielprogramme beurteilt werden, da nur für diese beiden eine optimale Lösung berechnet werden konnte. In diesen beiden Fällen werden mit beiden Lösungsstrategien jeweils optimale Lösungen berechnet.

5.4 Vergleich mit klassischen Verfahren zur Instruktionsanordnung

Das in dieser Arbeit verwendete Tool CIS entstand in der Zusammenarbeit von Daniel Kästner und Mark Langenbach. In der Arbeit von Daniel Kästner [Käs97] wurden die damit generierten Optimierungsaufgaben mit dem ILP-Verfahren gelöst. In der Diplomarbeit von Mark Langenbach [Lan97] werden ausgewählte klassische Verfahren zur Instruktionsanordnung untersucht und mit den Ergebnissen in [Käs97] verglichen. Dabei wurden die Untersuchungen ebenfalls mit den Benchmarks der Gruppe A durchgeführt, wodurch ein Vergleich mit den Ergebnissen in dieser Arbeit möglich ist. Die Verfahren beruhen auf im Laufe der Zeit entwickelten, graphbasierten Algorithmen und finden in vielen Compilern Anwendung [Lan97]. Die Verfahren sind nicht exakt, das heißt sie berechnen nur suboptimale Lösungen, jedoch benötigen sie stets nur eine Laufzeit von weniger als einer Sekunde[IX]. Die Qualität der damit gefundenen Lösungen kann aus Tabelle 5.10 entnommen werden; die mittlere Abweichung der Anzahl der Instruktionen vom Optimum beträgt 11,89% [Lan97].

Benchmark	*list scheduling*				*critical path*	optimal
	first fit	longest remain	max depend	highest level first		
FIR	10	10	10	10	10	8
IIR	9	9	15	9	11	7
DFT	16	16	16	16	16	14
Whetstone	22	22	25	21	23	20
histo	31	31	31	31	31	31
conv	17	17	18	18	18	17

Tabelle 5.10: Ergebnisse mit konventionellen, graphbasierten Algorithmen für IS von Benchmarks der Gruppe A

Mit der Variante der geteilten Instruktionsmenge (Strategie III) können alle Benchmarks der Gruppe A für IS in weniger als einer Sekunde berechnet werden. Daher bietet sich VGI für einen Vergleich mit den klassischen Verfahren zur Instruktionsanordnung an, da beide Verfahren suboptimale Ergebnisse in weniger als einer Sekunde berechnen. In Tabelle 5.11 ist jeweils die Anzahl der benötigten Instruktionen für die in [Lan97] untersuchten, graphbasierten Algorithmen und für VGI (Strategie III) gegenübergestellt. Als Lösung für die graphbasierten Algorithmen wird jeweils das kleinste Ergebnis für einen Benchmark aus Tabelle 5.10 verwendet. In der rechten Spalte der Tabelle ist das jeweilige Optimum für einen Benchmark aufgeführt.

Benchmark	graphbasierte Algorithmen	VGI (Strategie III)	optimal
FIR	10	8	8
IIR	9	7	7
DFT	16	15	14
Whetstone	21	21	20
histo	31	31	31
conv	17	17	17
mittlere Abweichung vom Optimum M_O	12,14%	2,02%	-

Tabelle 5.11: Vergleich der Anzahl der Instruktionen von graphbasierten Algorithmen mit VGI (Strategie III) für IS mit Benchmarks der Gruppe A

Mittels der graphbasierten Algorithmen werden für die großen Optimierungsaufgaben optimale Lösungen berechnet. Für die kleinen und mittelgroßen Benchmarks beträgt die Abweichung vom Optimum im Mittel fast zwei Instruktionen. Mit den

[IX] Unter Verwendung des selben Computers wie in [Käs97].

besten Ergebnissen aus Tabelle 5.10 ergibt sich eine mittlere Abweichung vom Optimum $M_O = 12,14$ % (siehe Tabelle 5.11). Mit VGI wird für vier von sechs Benchmarks eine optimale Lösung berechnet. In den beiden anderen Fällen beträgt die Abweichung vom Optimum nur eine Instruktion, damit ist $M_O = 2,03$ % für VGI. Da die Laufzeiten der beiden Verfahren (für diese Benchmarks) vergleichbar sind, mit VGI jedoch eine deutlich geringere Abweichung vom Optimum erreicht wird, könnte VGI eine Alternative zu konventionellen, graphbasierten Algorithmen sein. Durch den exponentiellen Charakter der Optimierungsaufgaben ist jedoch zu erwarten, daß eine Berechnung mit VGI nur bis zu einer gewissen Programmgröße möglich ist. Diese Programmgröße konnte jedoch aufgrund der genannten Probleme nicht ermittelt werden.

5.5 Gesamtauswertung

Der Vergleich der exakten Lösungsstrategien zeigt, daß zwar für IS mit SILP eine Lösung für die meisten Benchmarks berechnet wird, jedoch die Laufzeiten für Programme mit mehr als 20 Instruktionen stark ansteigen. Bis zu einer Größe von etwa 50 Instruktionen ist eine Berechnung in mehreren Stunden möglich; bei noch größeren Programmen sind für die Berechnungen mehr als 24h erforderlich. Die Berechnung mit SCLP ist für Programme mit bis zu 30 Instruktionen effizienter als mit SILP. Für größere Programme können mit SCLP innerhalb von 24h keine exakten Lösungen ermittelt werden. Für ISRA können sowohl mit SILP und als auch mit SCLP nur noch Optimierungsaufgaben bis zu etwa 30 Instruktionen gelöst werden. Diese Lösungen werden jedoch von SCLP in Sekunden, von SILP hingegen erst in Stunden berechnet. Insgesamt betrachtet ist die Berechnung einer garantiert optimalen Lösung mit beiden Verfahren nur für sehr kleine Programme effizient.

Aus diesem Grund wurden die Möglichkeiten von approximativen Lösungstrategien mit den Anforderungen einer hohen Codequalität bei effizienter Laufzeit untersucht. Für SCLP sind zwei geeignete Lösungsstrategien entwickelt worden: VGI und VML+. Zur Bewertung wurden sie mit der effizientesten Lösungsstrategie (SAIF) für SILP aus [Käs97] verglichen.

Mit SAIF können für IS fast alle Benchmarks in wenigen Minuten berechnet werden, für ISRA ist eine Lösung der Optimierungsaufgaben nur noch für Programme mit weniger als 30 Instruktionen möglich. Die erreichte Codequalität ist sehr gut; für IS wird eine mittlere Abweichung vom Optimum von nur $M_O = 0,83$ % berechnet, für ISRA ist $M_O = 0$ %. Im Gegensatz dazu kann mit VGI für IS für alle Benchmarks eine Lösung in weniger als einer halben Sekunde berechnet werden. Für ISRA wird, bis auf einen Benchmark, eine Lösung in wenigen Sekunden ermittelt. Die Codequalität ist ebenfalls sehr gut und beträgt für IS $M_O = 2,02$ % und für ISRA $M_O = 1,43$ %. Mit VML+ wird die beste Codequalität der drei approximativen Lösungsstrategien erreicht. Die mittlere Abweichung vom Optimum ist sowohl für IS als auch für ISRA $M_O = 0$ %. Für VML+ sind etwas höhere Laufzeiten notwendig als für VGI, die Berechnungszeiten sind dennoch deutlich kleiner als für SAIF.

Der Vergleich der Laufzeiten der exakten Lösungsstrategien mit den Ergebnissen für approximative, graphbasierte Algorithmen aus [Lan97] zeigt, daß eine optimale

Anordnung der Instruktionen weder von ILP noch von CLP in einer vergleichbaren Laufzeit berechnet werden kann. Die Untersuchungen von ILP in [Käs97] haben aber bereits gezeigt, daß mit approximativen Lösungstrategien Verbesserungen in der Codequalität bei akzeptablen Laufzeiten gegenüber den graphbasierten Algorithmen erzielt werden können. Mit CLP sind weitere Verbesserungen sowohl in der Laufzeit als auch in der Codequalität erreicht worden. Mit VGI ist sogar eine Lösungsstrategie für CLP entwickelt worden, die für die untersuchten Benchmarks eine vergleichbare Laufzeit wie die graphbasierten Algorithmen aufweist, dabei aber eine deutlich bessere Codequalität zeigt: $M_O = 12,14$ % für die graphbasierten Algorithmen gegenüber $M_O = 2,02$ % für VGI.

Aus diesen Ergebnissen ergeben sich verschiedene Möglichkeiten, CLP in der Codegenerierung für DSPs einzusetzen. Mit VGI ist eine Lösungsstrategie entwickelt worden, die bis zu einer gewissen Programmgröße[X] eine Alternative zu klassischen Verfahren der Instruktionsanordnung auf der Basis von approximativen, graphbasierten Algorithmen darstellen könnte. Eine weitere erfolgversprechende Möglichkeit ist eine Integration der CLP-Optimierung in graphbasierte Algorithmen. Durch eine Überprüfung des Programms könnten hier Codestücke, die für eine Optimierung mit CLP geeignet sind (zum Beispiel innere Schleifen), erkannt werden. Für diese Codestücke kann dann eine entsprechende Optimierung mit CLP gestartet werden. Als zusätzlicher Vorteil von CLP können durch den Einsatz von VML+ mehrere, zumeist optimale, Anordnungen der Instruktionen berechnet werden, von denen in weiteren Optimierungen des Gesamtprogramms die beste ausgewählt werden kann.

Weitere Verbesserungen der Laufzeiten für CLP sind durch einen Wechsel des Modells zu erwarten. Das auf ILP zugeschnittene SILP-Modell wurde nur unwesentlich an CLP angepaßt. Durch eine andere Modellierung könnten spezielle Eigenschaften und Features von CLP besser ausgenutzt werden. Hierzu gehören zum Beispiel komplexe Constraints (*cumulative constraints*) über mehrere Bedingungen oder andere Bibliotheken wie *REPAIR*, in der bei der Verletzung eines Constraints eine Lösung gesucht wird, um diesen Constraint möglicherweise doch zu erfüllen (*repair contraints*).

[X] Diese Programmgröße konnte aufgrund der Begrenzung eines Tools auf Programme bis zu maximal 99 Instruktionen nicht ermittelt werden.

Kapitel 6

Zusammenfassung und Ausblick

Das Ziel der Diplomarbeit ist ein Vergleich von Optimierungsstrategien für ILP und CLP am Beispiel der Codegenerierung für DSPs. Als Modell zur Beschreibung von Aufgaben der Codegenerierung ist das SILP-Modell verwendet worden, mit dem die für die Instruktionsanordnung relevanten Berechnungsvorgänge eines realen Prozessors vollständig und korrekt in Form von ganzzahligen linearen Programmen modelliert werden. Zunächst sind die Referenzdaten für das ILP-Verfahren auf der Basis dieses Modells mit den in [Käs97] entwickelten Lösungsstrategien ermittelt worden. Im nächsten Schritt ist eine Abbildung dieses Modells auf CLP notwendig gewesen. Dazu wurden durch das neuentwickelte Tool SILP2SCLP Optimierungsaufgaben des SILP-Modells, die für die Berechnung mit ILP generiert wurden, in eine Darstellung überführt, die mit CLP bearbeitet werden kann. Auf der Basis dieses gemeinsamen Modells sind Lösungsstrategien für CLP entwickelt worden, mit denen eine effiziente Berechnung der Optimierungsaufgaben in CLP möglich ist. Mit den ermittelten Ergebnissen für das CLP-Verfahren konnte dann der Vergleich von exakten und suboptimalen Lösungsstrategien für ILP und CLP durchgeführt werden. Zusätzlich ergab sich die Möglichkeit, einen Vergleich mit klassischen Verfahren zur Instruktionsanordnung auf der Basis des gleichen Modells und der gleichen Benchmarks zu führen.

Der Vergleich der exakten Lösungsstrategien hat gezeigt, daß mit beiden Verfahren eine garantiert optimale Anordnung nur für kleine Programme mit bis zu etwa 30 Instruktionen berechnet werden kann. Für diese Berechnungen werden für das CLP-Verfahren einige Sekunden, für das ILP-Verfahren einige Stunden benötigt. Allerdings können mit ILP innerhalb von 24h einige Benchmarks mehr als mit CLP gelöst werden. Aufgrund der hohen Laufzeiten der exakten Lösungsstrategien sind für beide Verfahren approximative Strategien entwickelt worden, mit dem Ziel, eine möglichst gute Codequalität in effizienter Laufzeit zu berechnen. Mit den approximativen Lösungsstrategien werden mit ILP Programme für das Problem der Instruktionsanordnung mit bis zu 50 Instruktionen in wenigen Minuten bis zu einer Stunde berechnet. Es wird eine sehr hohe Codequalität mit einer mittleren Abweichung vom Optimum $M_O = 0,83$ % erreicht. Mit der approximativen Lösungsstrategie VGI können mit CLP Optimierungsaufgaben mit bis fast 100 Instruktionen in zwei Sekunden gelöst werden. Größere Programme konnten aufgrund der Beschränkung eines der Tools nicht untersucht werden. Die Codequalität ist mit $M_O = 2,02$ % geringfügig schlechter als mit der suboptimalen Lösungsstrategie in

ILP. Eine optimale Codequalität von $M_O = 0$ % konnte für die untersuchten Benchmarks mit der Lösungsstrategie VML+ für CLP erreicht werden. Die Berechnungszeiten sind für diese Strategie etwas schlechter als mit VGI, jedoch deutlich besser als in ILP. Ein großer Vorteil von VML+ ist die Berechnung mehrerer, zumeist minimaler Anordnungen. Im Vergleich zu den Ergebnissen mit klassischen Verfahren zur Instruktionsanordnung aus [Lan97] zeigt sich, daß mit VGI in vergleichbarer Laufzeit eine wesentlich höhere Codequalität erreicht wird. Mit den approximativen, graphbasierten Algorithmen wird eine mittlere Abweichung vom Optimum $M_O = 11,89$ %, mit VGI $M_O = 2,02$ % erreicht. Allerdings ist zu vermuten, daß die Laufzeiten mit VGI ab einer gewissen Programmgröße sehr stark ansteigen und eine Lösung mit dieser Lösungsstrategie verhindern. Ein weiterer Vorteil des CLP-Verfahrens gegenüber ILP ist der wesentlich geringere Arbeitsspeicherbedarf. In ILP wird die Lösungssuche über einen Entscheidungsbaum gesteuert, dessen Repräsentation im Speicher so groß wird, daß für große Benchmarks eine Berechnung mit ILP nicht möglich ist. Im Gegensatz dazu wird die Lösungssuche in CLP über zusätzliche Constraints geführt, die nur zu einem geringen Anstieg des Arbeitsspeicherbedarfs führen[I].

Ein völlig eigenständiges Verfahren zur Instruktionsanordnung und Registerallokation auf der Basis eines ILP-Modells ist aus Komplexitätsgründen nicht erfolgversprechend. Durch den Einsatz von CLP konnten zwar die Grenzen der Einsetzbarkeit gegenüber ILP deutlich verbessert werden. Für das Problem der Instruktionsanordnung werden für Programme mit bis zu 100 Instruktionen Lösungen in wenigen Sekunden berechnet. Jedoch ist zu erwarten, daß bei größeren Programmen die Laufzeiten exponentiell ansteigen und eine Lösung mit CLP mit den bislang entwickelten Lösungsstrategien an ihre Grenzen stößt. Allerdings werden bei der Codegenerierung für eingebettete Systeme auch höhere Compilerlaufzeiten durch solche Optimierungen in Kauf genommen. Zum einen werden diese Optimierungen nicht bereits in der Programmentwicklung durchgeführt, sondern erst bei Fertigstellung. Zum anderen sind die damit verbundenen Kosteneinsparungen gerade für eingebettete Systeme wirtschaftlich sehr wichtig.

Das CLP-Verfahren ist gut geeignet bei der Anwendung auf Codesequenzen, wie etwa Funktionen oder innere Schleifen, die im Bereich der Codegenerierung für DSPs häufig auftreten. Diese Codestücke sind in der Regel klein und können mit CLP innerhalb von einigen Sekunden bis wenigen Minuten berechnet werden.

Aus den genannten Gründen bietet sich eine Verbindung mit einem globalen heuristischem Verfahren an, wie den konventionellen, graphbasierten Algorithmen, die in vielen optimierenden Compilern Verwendung finden. Dort könnten für geeignete, häufig aufgerufene Codestücke Optimierungen mit hoher Codequalität in effizienter Laufzeit berechnet und damit die Ausführungsgeschwindigkeit des erzeugten Programms deutlich gesteigert werden. Dabei eröffnet die hier erarbeitete Lösungsstrategie VML+ weitere Optimierungsoptionen über die Wahlmöglichkeit zwischen mehreren, zumeist optimalen Lösungen. Damit ist die Wahl einer geeigneten Lösung möglich, die durch eine Verzahnung mit anderen Codestücken möglicherweise zu einer weiteren Reduzierung der Instruktionsanzahl beiträgt.

[I] Alle Berechnungen mit CLP in dieser Arbeit konnten vollständig im Hauptspeicher von 64 Megabyte durchgeführt werden.

Es sind noch bessere Ergebnisse mit CLP denkbar. In dieser Arbeit wurde, um einen direkten Vergleich mit ILP zu ermöglichen, das SILP-Modell zur Beschreibung von Aufgaben der Codegenerierung verwendet. Bei der direkten Abbildung dieser Aufgaben in ein CLP-Modell könnte eine effizientere Beschreibung gefunden werden, um die Möglichkeiten von CLP bereits bei der Modellierung besser auszunutzen. Interessant wäre ein derartiges Modell auch dahingehend, daß die Beschreibung eines Problems in CLP leicht um weitere Aspekte erweitert werden kann. Damit könnte dieses Modell zum Beispiel an verschiedene Prozessoren angepaßt werden, indem Hardware-spezifische Bedingungen gesondert beschrieben werden. Ein solches Modell könnte über Constraints ausdrücken, welche Hardwareressourcen zur Verfügung stehen und wäre damit leicht an verschiedene Prozessoren anzupassen. Diese Möglichkeiten gewinnen im Hinblick auf die Vielzahl unterschiedlicher DSPs und ihrer Weiterentwicklung an Bedeutung.

Anhang A

Benchmarks

Dieser Anhang enthält Informationen über die untersuchten Beispielprogramme. In den Tabellen A.1 bis A.4 sind zur Beschreibung der Benchmarks die Anzahl der Instruktionen in der seriellen (nicht optimierten) Ausführung und die Anzahl der Variablen und Constraints aufgeführt. Zusammen mit der Größe der von CIS generierten *LP*-Dateien ergibt sich ein Bild von der Komplexität des betrachteten Problems. Die Beispielprogramme der Gruppe A haben etwa 20 bis 50 Instruktionen, die Beispielprogramme der Gruppe B haben bis zu 93 Instruktionen. Jede Mikrooperation des ADSP-2106x kann in einem Taktzyklus ausgeführt werden, damit ist in der seriellen Fassung die Anzahl der Instruktionen gleich der Anzahl der benötigten Instruktionszyklen (Takte).

Auf den weiteren Seiten steht für jeden Benchmark nach einer kurzen Beschreibung des betrachteten Problems in der linken Spalte jeweils die Instruktionsfolge, die als Eingabe des Optimierungsprozesses verwendet wird. In der rechten Spalte ist eine mit CLP berechnete optimale Lösung für IS dargestellt. Zu einer Instruktion zusammengefaßte Mikrooperationen werden durch Kommata getrennt; Instruktionen werden durch Strichpunkte abgesetzt. Alle Assemblercodesequenzen werden eingeleitet und abgeschlossen durch Assemblerdirektiven, die Beginn bzw. Ende eines Codesegments spezifizieren. Die Direktiven stellen keine Instruktionen dar und werden daher bei der Ermittlung der Instruktionsanzahl nicht berücksichtigt. Eine weitere Beschreibung der Beispielprogramme der Gruppe (A) findet sich in [Käs97].

Benchmark	Instruktionen	Variablen	Constraints	Größe LP [Byte]
FIR	18	71	155	3783
IIR	20	82	166	4250
DFT	26	189	410	15157
Whetstone	26	218	387	15265
histo	43	440	722	30373
conv	49	447	916	38952

Tabelle A.1: IS für Benchmarks der Gruppe A

Benchmark	Instruktionen	Variablen	Constraints	Größe LP [Byte]
FIR	18	678	425	60357
IIR	20	957	613	91515
DFT	26	1209	2151	301662
Whetstone	26	1634	1408	206419
histo	43	2799	3619	545183
conv	49	6619	17272	2794031

Tabelle A.2: ISRA für Benchmarks der Gruppe A

Benchmark	Instruktionen	Variablen	Constraints	Größe LP [Byte]
biquad_o	27	366	724	32099
complex_mul	32	346	526	22808
lattice	87	1894	1446	91700
n_comple	93	2463	1332	97342

Tabelle A.3: IS für Benchmarks der Gruppe B

Benchmark	Instruktionen	Variablen	Constraints	Größe LP [Byte]
biquad_o	27	1245	1569	177908
complex_mul	32	1260	1021	119322
lattice	87	10374	4717	918737
n_comple	93	10623	3337	750023

Tabelle A.4: ISRA für Benchmarks der Gruppe B

FIR-Filter

FIR-Filter (*Finite Impulse Response* Filter) zählen zu den am häufigsten implementierten Anwendungen der digitalen Bildverarbeitung. Sie berechnen im wesentlichen ein gewichtetes Mittel einer Folge von Eingabedaten und lassen sich durch folgende Gleichung beschreiben:

$$y[n] = \sum_{i=0}^{M} h_i x[n-i]$$

Dabei heißt M die Ordnung des Filters, die $h_i \in I\!R$ $(0 \leq i \leq M)$ werden als Filterkoeffizienten bezeichnet; $y[n]$ ist das Ausgangssignal und $x[n]$ das Eingangssignal zum Zeitpunkt n.

```
.SEGMENT /PM pm_code;
.GLOBAL fir;
.GLOBAL fir_init;
.EXTERN coefs;
.EXTERN dline;

fir: dm(i0,m0)=f0;
     f0=dm(i0,m0);
     f4=pm(i8,m8);
        f8=f0*f4;
        f0=dm(i0,m0);
        f4=pm(i8,m8);
        f12=f0*f4;
        f0=dm(i0,m0);
        f4=pm(i8,m8);
        lcntr=r1, do macs until lce;
                f8=f8+f12;
                f12=f0*f4;
                f0=dm(i0,m0);
        macs:
                f4=pm(i8,m8);

        f8=f8+f12;
        f12=f0*f4;
        f0=f8+f12;
        rts;

.ENDSEG;
```

```
.SEGMENT /PM pm_code;
.GLOBAL fir;
.GLOBAL fir_init;
.EXTERN coefs;
.EXTERN dline;

fir: dm(i0,m0)=f0,f4=pm(i8,m8);
     f0=dm(i0,m0);
     f0=dm(i0,m0),f4=pm(i8,m8),f8=f0*f4;
     f0=dm(i0,m0),f4=pm(i8,m8),f12=f0*f4;
     lcntr=r1, do macs until lce;
        f0=dm(i0,m0),f4=pm(i8,m8),f8=f8+f12,f12=f0*f4;
     macs:
        f8=f8+f12,f12=f0*f4;
     rts, f0=f8+f12;

.ENDSEG;
```

IIR-Filter

Aus der Berechnung eines IIR-Filters *Infinite Impuls Response Filter* wurde ein Basisblock ausgewählt um eine vollständige, von externen Registerallokatoren unabhängige, Registerzuteilung durchzuführen. Dieses Codestück wird in [Käs97] auch als *cascade* bezeichnet.

```
.SEGMENT /PM    pm_code;

cascaded_biquad:
                f12=0;
                f8=0;
                f2=dm(i0,m1);
                f4=pm(i8,m1);
                f8=f8+f12;
                f12=f2*f4;
                f3=dm(i0,m1);
                f4=pm(i8,m8);
                f8=f8+f12;
                f12=f3*f4;
                dm(i1,m1)=f3;
                f4=pm(i8,m8);
                f8=f8+f12;
                f12=f2*f4;
                f2=dm(i0,m1);
                f4=pm(i8,m8);
                dm(i1,m1)=f8;
                f8=f8+f12;
                f12=f3*f4;
                f4=pm(i8,m8);
.ENDSEG;
```

```
.SEGMENT /PM    pm_code;

cascaded_biquad:
                f12=0;
                f8=0;
                f2=dm(i0,m1),f4=pm(i8,m1),f8=f8+f12;
                f12=f2*f4,f3=dm(i0,m1),f4=pm(i8,m8);
                f8=f8+f12,f12=f3*f4,dm(i1,m1)=f3,f4=pm(i8,m8);
                f8=f8+f12,f12=f2*f4,f2=dm(i0,m1),f4=pm(i8,m8);
                dm(i1,m1)=f8,f8=f8+f12,f12=f3*f4,f4=pm(i8,m8);
.ENDSEG;
```

DFT-Transformation

Dieses Beispielprogramm berechnet die komplexe diskrete Fouriertransformation (DFT) der Daten einer Eingabedatei. Sie läßt sich wie folgt beschreiben:

$$c_k = \sum_{j=0}^{n-1} y_j w_n^{jk} \qquad k \in \{0, 1, \ldots, n-1\}$$

Dabei stellen die c_k die Ausgabewerte für $0 \leq k \leq n-1$ dar, die y_i sind Eingabewerte zum Zeitpunkt j. Die komplexe Größe w_n stellt eine n-te Einheitswurzel dar: $w_n = e^{-\frac{2\pi}{n}}$.

```
.SEGMENT/PM     pm_code;

dft:
    I10=0;
    L10=0;
    F15=0.0;
    LCNTR=64, DO outer UNTIL LCE;
      F8=PASS F15;
      M8=I10;
      F9=PASS F15;
      F0=DM(I0,M1);
      F5=PM(I9,M8);
      F12=F0*F5;
      F4=PM(I8,M8);
      LCNTR=63, DO inner UNTIL LCE;
        F13=F0*F4;
        F9=F9+F12;
        F0=DM(I0,M1);
        F5=PM(I9,M8);
        F12=F0*F5;
        F8=F8-F13;
inner:
      F4=PM(I8,M8);
      F13=F0*F4;
      F9=F9+F12;
      F8=F8-F13;
      DM(I2,M1)=F9;
      MODIFY(I10,M9);
outer:
      DM(I1,M1)=F8;
      RTS;

.ENDSEG;
```

```
.SEGMENT/PM     pm_code;

dft:
       I10=0;
       F15=0.0;
       LCNTR=64, DO outer UNTIL LCE;
         F8=PASS F15,M8=I10;
         F9=PASS F15,F0=DM(I0,M1),F5=PM(I9,M8);
         F12=F0*F5,F4=PM(I8,M8);
         LCNTR=63, DO inner UNTIL LCE;
           F13=F0*F4,F9=F9+F12,F0=DM(I0,M1),F5=PM(I9,M8);
inner:
        F12=F0*F5,F8=F8-F13,F4=PM(I8,M8);
        F13=F0*F4,F9=F9+F12,MODIFY(I10,M9);
        F8=F8-F13,DM(I2,M1)=F9;
outer:
        DM(I1,M1)=F8;
       L10=0;
       RTS;

.ENDSEG;
```

Whetstone-Funktion

In diesem Beispielprogramm wird der Code für eine im Rahmen des bekannten Whetstone-Benchmarks verwendete Funktion optimiert.

```
.segment /pm seg_pmco;
.file "whet.c";
.endseg;
.segment /dm seg_dmda;
.gcc_compiled;
.extern _pa;
.extern _atanf;
.extern _sinf;
.extern _cosf;
.extern _p3;
.extern _p0;
.extern _sqrtf;
.extern _expf;
.extern _logf;
.endseg;
.segment /pm seg_pmco;
.global _p3;

_p3:    modify(i7,-3);
        f13=f8;
        f1=dm(100);
        f4=f4+f13;
        f5=dm(200);
        f8=f1*f4;
        f2=recips f5,i4=r12;
        f4=f8+f13;
        f13=f1*f4;
        f9= 0x40000000;
        f12=f2*f5;
        f4=f8+f13;
        f4=f2*f4;
        f2=f9-f12;
        f12=f2*f12;
        f4=f2*f4;
        f2=f9-f12;
        f12=f2*f12;
        f4=f2*f4;
        f2=f9-f12;
        f4=f2*f4;
        f2=f2-f12;
        f2=f2*f4;
        f2=f2+f4;
        dm(i4,m5)=f2;
.endseg;
```

```
.segment /pm seg_pmco;
.file "whet.c";
.endseg;
.segment /dm seg_dmda;
.gcc_compiled;
.extern _pa;
.extern _atanf;
.extern _sinf;
.extern _cosf;
.extern _p3;
.extern _p0;
.extern _sqrtf;
.extern _expf;
.extern _logf;
.endseg;
.segment /pm seg_pmco;
.global _p3;

_p3:    f13=f8;
        f1=dm(100);
        f5=dm(200);
        f4=f4+f13,f2=recips f5,i4=r12;
        f8=f1*f4;
        f4=f8+f13;
        f13=f1*f4,f4=f8+f13;
        f9= 0x40000000;
        f12=f2*f5;
        f4=f2*f4;
        f2=f9-f12,f12=f2*f12;
        f4=f2*f4;
        f2=f9-f12,f12=f2*f12;
        f4=f2*f4;
        f2=f9-f12;
        f4=f2*f4;
        f2=f2-f12;
        f2=f2*f4;
        f2=f2+f4;
        dm(i4,m5)=f2;
        modify(i7,-3);

.endseg;
```

Histo Bildhistogramm

Ein Bildhistogramm ist ein Feld, das die Anzahl der Vorkommen jeder Graustufe in einem Bild angibt. Kann ein Pixel eines typischen monochromatischen Bildes 256 verschiedene Werte annehmen, sind für das Histogramm entsprechend 256 Feldelemente erforderlich. In jedem Element wird die Anzahl der Vorkommen der entsprechenden Graustufe gespeichert.

```
.SEGMENT /pm    pm_code;
.GLOBAL _histo;

_histo:     r3 = l1;
            r3 = lshift r3 by -1;
            r14=i8;
            r3 = r3 - 1;
            r15=i9;
            r0=dm(i0,m0);
            lcntr = r3, do hloop until lce;
                r0=r0+r14;
                r2=dm(i0,m0);
                r2=r2+r15;
                i8=r0;
                i9=r2;
                r1=pm(i8,m15);
                r1=r1+1;
                r3=pm(i9,m15);
                r3=r3+1;
                pm(i8,m15)=r1;
                r0=dm(i0,m0);
hloop:
        pm(i9,m15)=r3;
        r0=r0+r14;
        r2=dm(i0,m0);
        r2=r2+r15;
        i8=r0;
        i9=r2;
        r1=pm(i8,m15);
        r1=r1+1;
        r3=pm(i9,m15);
        r3=r3+1;
        pm(i8,m15)=r1;
        pm(i9,m15)=r3;
        i8=b8;
        i9=b9;
        r2=l1;
        r2=r2-1;
        r0=pm(i8,m8);
        r1=pm(i9,m8);
        lcntr=r2, do combine until lce;
                r0=pm(i8,m8);
                r2=r0+r1;
                dm(i1,m0)=r2;
combine:
        r1=pm(i9,m8);
        rts (db);
        r2=r0+r1;
        dm(i1,m0)=r2;

.ENDSEG;
```

```
.SEGMENT /pm    pm_code;
.GLOBAL _histo;

_histo:  r3 = l1;
         r3 = lshift r3 by -1,r14=i8;
         r3 = r3 - 1,r15=i9;
         r0=dm(i0,m0);
         lcntr = r3, do hloop until lce;
           r0=r0+r14,r2=dm(i0,m0);
           r2=r2+r15,i8=r0;
           i9=r2;
           r1=pm(i8,m15),r0=dm(i0,m0);
           r1=r1+1,r3=pm(i9,m15);
           r3=r3+1,pm(i8,m15)=r1;
hloop:
         pm(i9,m15)=r3;
         r0=r0+r14,r2=dm(i0,m0);
         r2=r2+r15,i8=r0;
         i9=r2;
         r1=pm(i8,m15);
         r1=r1+1;r3=pm(i9,m15);
         r3=r3+1,pm(i8,m15)=r1;
         pm(i9,m15)=r3;
         i8=b8;
         i9=b9;
         r2=l1;
         r2=r2-1,r0=pm(i8,m8);
         r1=pm(i9,m8);
         lcntr=r2, do combine until lce;
           r0=pm(i8,m8);
           r2=r0+r1;
combine:
         dm(i1,m0)=r2;r1=pm(i9,m8);
         rts (db);
         r2=r0+r1;
         dm(i1,m0)=r2;

.ENDSEG;
```

Konvolutionsberechnung

Das Beispielprogramm conv stammt aus dem Bereich der Bildverarbeitung; es wird eine Funktion berechnet, die zur Bildrekonstruktion eingesetzt wird. Die am meisten eingesetzte Bildrekonstruktionsmethode ist die Konvolution. Sie rekonstruiert ein Bild in zwei Schritten: Konvolution der Projektionsdaten mit einer gegebenen Funktion und Rückprojektion der transformierten Projektionsdaten. Die Konvolution ist eine Funktion

$$[\Phi * \Psi](v) = \int_{-\infty}^{\infty} \Phi(u)\ \Psi(v-u)\ du, \quad mit \ \ \Phi, \Psi \in I\!R$$

Um die Berechnungszeit einzuschränken, wurde in [Käs97] aus dem resultierenden Assemblerprogramm mit 124 Anweisungen ein Basisblock mit 49 Instruktionen extrahiert.

```
.SEGMENT /PM pm_code;
.GLOBAL _conv;
_conv:  f8=f9+f12;
        f12=f2*f6;
        f13=f1*f6;
        f8=f8+f12;
        f7=dm(i0,m1);
        f9=f9+f13;
        f13=f2*f7;
        f4=dm(i0,m0);
        f0=pm(i8,m8);
        f12=f0*f4;
        f9=f9+f13;
        f5=dm(i0,m0);
        f1=pm(i8,m8);
        f13=f0*f5;
        f8=f8+f12;
        f6=dm(i0,m0);
        f12=f1*f5;
        f9=f9+f13;
        f2=pm(i8,m8);
        f8=f8+f12;
        f12=f2*f6;
        f13=f1*f6;
        f8=f8+f12;
        f7=dm(i0,m1);
        f9=f9+f13;
        f13=f2*f7;
        f4=dm(i0,m1);
        f0=pm(i8,m8);
        f12=f0*f4;
        f9=f9+f13;
        f5=dm(i0,m0);
        f1=pm(i8,m8);
        f13=f0*f5;
        f8=f8+f12;
        f6=dm(i0,m0);
        f12=f1*f5;
        f9=f9+f13;
        f2=pm(i8,m8);
        f8=f8+f12;
        f12=f2*f6;
        f7=dm(i0,m2);
        f13=f1*f6;
        f8=f8+f12;
        f9=f9+f13;
        f13=f2*f7;
        modify(i0,m3);
        f9=f9+f13;
        dm(i1,m0)=f8;
        dm(i1,m0)=f9;
.ENDSEG;
```

```
.SEGMENT /pm    pm_code;
.GLOBAL _conv;
_conv:  f8=f9+f12,f12=f2*f6,f7=dm(i0,m1),f0=pm(i8,m8);
        f13=f1*f6,f8=f8+f12,f4=dm(i0,m0),f1=pm(i8,m8);
        f9=f9+f13,f13=f2*f7,f5=dm(i0,m0),f2=pm(i8,m8);
        f12=f0*f4,f9=f9+f13,f6=dm(i0,m0);
        f13=f0*f5,f8=f8+f12,f7=dm(i0,m1),f0=pm(i8,m8);
        f12=f1*f5,f9=f9+f13,f4=dm(i0,m1);
        f8=f8+f12,f12=f2*f6,f5=dm(i0,m0);
        f13=f1*f6,f8=f8+f12,f1=pm(i8,m8),f6=dm(i0,m0);
        f9=f9+f13,f13=f2*f7,f2=pm(i8,m8),f7=dm(i0,m2);
        f12=f0*f4,f9=f9+f13,modify(i0,m3);
        f13=f0*f5,f8=f8+f12;
        f12=f1*f5,f9=f9+f13;
        f8=f8+f12,f12=f2*f6;
        f13=f1*f6,f8=f8+f12;
        f9=f9+f13,f13=f2*f7,dm(i1,m0)=f8;
        f9=f9+f13;
        dm(i1,m0)=f9;

.ENDSEG;
```

Biquad_one

In dem Beispielprogramm Biquad_one_section wird ein Filter mit folgender mathematischer Beschreibung abgebildet:

$$\begin{array}{rclclcl} w(n) & = & x(n) & - & a_1 * w_{n-1} & - & a_2 * w_{n-2} \\ y(n) & = & b_0 * w_n & + & b_1 * w_{n-1} & + & b_2 * w_{n-2} \end{array}$$

```
.SEGMENT /PM pm_code;
.GLOBAL _biquad_o;

_biquad_0:  modify(i7,-2);
            r12=r2*r4;
            r2=r8-r12;
            dm(-3,i6)=r2;
            r12=r2*r4;
            r8=dm(-3,i6);
            r2=r8-r12;
            dm(-3,i6)=r2;
            r4=dm(-3,i6);
            r2=r2*r4;
            dm(-2,i6)=r2;
            r12=r2*r4;
            r8=dm(-2,i6);
            r2=r8+r12;
            dm(-2,i6)=r2;
            r12=r2*r4;
            r8=dm(-2,i6);
            r2=r8+r12;
            dm(-2,i6)=r2;
            r2=dm(-3,i6);
            dm(i7,m7)=r2;
            dm(i7,m7)=pc;
            r0=dm(-2,i6);
            jump(pc, _L$2);
    _L$2:   i12=dm(-1,i6);
            i7=i6;
            i6=dm(0,i6);

.ENDSEG;
```

```
SEGMENT /pm    pm_code;
.GLOBAL _biquad_o;

_biquad_0:  modify(i7,-2);
            r12=r2*r4,jump(pc, _L$2);
            i12=dm(-1,i6);jump(pc, _L$2);
            r2=r8-r12;
            dm(-3,i6)=r2,r12=r2*r4;
            r8=dm(-3,i6);
            r2=r8-r12,i7=i6;
            i6=dm(0,i6);
            r2=r8-r12;
            dm(-3,i6)=r2;
            r4=dm(-3,i6);
            r2=r2*r4;
            dm(-2,i6)=r2,r12=r2*r4;
            r8=dm(-2,i6);
            r2=r8+r12;
            dm(-2,i6)=r2,r12=r2*r4;
            r8=dm(-2,i6);
            r2=r8+r12;
            dm(-2,i6)=r2;
            r2=dm(-3,i6);
            dm(i7,m7)=r2;
            dm(i7,m7)=pc;
            r0=dm(-2,i6);

.ENDSEG;
```

Multiplikation im Komplexen

In diesem Beispielprogramm aus der Benchmark Sammlung zum DSP-Kernel (DSP Stone) wird die Multiplikation von zwei komplexen Zahlen (X, Y) berechnet. Jede komplexe Zahl $a = (\alpha, \beta)$ besteht aus Realteil $\alpha = (\alpha, 0) = Re(a)$ und Imaginärteil $i\beta = (0, \beta) = Im(a)$. Die Multiplikation zweier komplexer Zahlen ist wie folgt definiert:

$$X * Y = (\alpha_1, \beta_1) * (\alpha_2, \beta_2) = (\alpha_1 * \alpha_2 - \beta_1 * \beta_2, \alpha_1 * \beta_2 + \alpha_2 * \beta_1)$$

```
.SEGMENT /PM pm_code;
.GLOBAL _complex_mul;

_complex_mul:  r2=_ci_9;
               dm(i7,m7)=r2;
               r2=_cr_8;
               dm(i7,m7)=r2;
               r2=_bi_7;
               dm(i7,m7)=r2;
               r12=_br_6;
               r8=_ai_5;
               r4=_ar_4;
               dm(i7,m7)=r2;
               dm(i7,m7)=pc;
               modify(i7,3);
               r8=r2*r4;
               r12=r2*r4;
               r2=r8-r12;
               r8=r2*r4;
               r12=r2*r4;
               r2=r8+r12;
               r2=_ci_9;
               dm(i7,m7)=r2;
               r2=_cr_8;
               dm(i7,m7)=r2;
               r2=_bi_7;
               dm(i7,m7)=r2;
               r12=_br_6;
               r8=_ai_5;
               r4=_ar_4;
               dm(i7,m7)=r2;
               dm(i7,m7)=pc;
               modify(i7,3);
   _L$2:       i12=dm(-1,i6);
               i7=i6;
               i6=dm(0,i6);

.ENDSEG;
```

```
SEGMENT /pm    pm_code;
.GLOBAL _complex_mul;

_complex_mul:   r12=_br_6;
                r2=_ci_9;
                dm(i7,m7)=r2,r8=_ai_5;
                r4=_ar_4;
                r2=_cr_8;
                dm(i7,m7)=r2,r2=_bi_7;
                dm(i7,m7)=r2,r8=r2*r4;
                dm(i7,m7)=r2,r12=r2*r4;
                dm(i7,m7)=pc,r2=r8-r12;
                r8=r2*r4;
                r12=r2*r4;
                r4=_ar_4;
                modify(i7,3);
                r2=r8+r12;
                r12=_br_6;
                r8=_ai_5;
                r2=_ci_9;
                dm(i7,m7)=r2,r2=_cr_8;
                dm(i7,m7)=r2,r2=_bi_7;
                dm(i7,m7)=r2;
                dm(i7,m7)=r2;
                dm(i7,m7)=pc;
                modify(i7,3);
                i12=dm(-1,i6);
                i7=i6;
                i6=dm(0,i6);

.ENDSEG;
```

Lattice

Dieses Beispielprogramm gehört nicht zu den Standardbeispielen aus dem DSP Stone. Es wird ein Lattice-Filter zweiter Ordnung berechnet. Aus dem Assemblerprogramm wurde ein einzelner Basisblocks mit 83 Anweisungen ausgewählt.

```
.SEGMENT /PM pm_code;
.GLOBAL _lattice;

_lattice:   r2=dm(-3,i6);
            r4=8;
            comp(r2,r4);
            i3=i6;
            modify(i3,-1083);
            r2=i3;
            r4=dm(-3,i6);
            r8=120;
            r4=r4*r8;
            r2=r2+r4;
            r4=dm(-2,i6);
            r1=r2+r4;
            i4=r1;
            i3=i6;
            modify(i3,-1083);
            r2=i3;
            r4=dm(-3,i6);
            r8=r4-1;
            r12=120;
            r4=r8*r12;
            r2=r2+r4;
            r4=dm(-2,i6);
            r1=r2+r4;
            i0=r1;
            r2=dm(-3,i6);
            m4=r2;
            modify(i1,m4);
            i3=i6;
            modify(i3,-2163);
            r2=i3;
            r4=dm(-3,i6);
            r8=r4-1;
            r12=120;
            r4=r8*r12;
            r2=r2+r4;
            r4=dm(-2,i6);
            r8=r4-1;
            r1=r2+r8;
            i2=r1;
            r2=dm(i1,m5);
            r4=dm(i2,m5);
            r12=r2*r4;
            r8=dm(i0,m5);
            r2=r8+r12;
            dm(i4,m5)=r2;
            i3=i6;
            modify(i3,-2163);
            r2=i3;
            r4=dm(-3,i6);
            r8=120;
            r4=r4*r8;
            r2=r2+r4;
            r4=dm(-2,i6);
            r1=r2+r4;
            i4=r1;
```

```
SEGMENT /pm    pm_code;
.GLOBAL _lattice;

_lattice:       r4=8;
                r2=dm(-3,i6);
                r8=120;
                r12=120;
                comp(r2,r4),i3=i6;
                r4=dm(-3,i6);
                modify(i3,-1083);
                r2=i3,r4=r4*r8;
                r2=r2+r4,r4=dm(-2,i6);
                r1=r2+r4,i3=i6;
                r4=dm(-3,i6);
                i4=r1,r8=r4-1;
                modify(i3,-1083);
                r2=i3,r4=r8*r12;
                r12=120;
                r2=r2+r4,r4=dm(-2,i6);
                r1=r2+r4,r2=dm(-3,i6);
                i0=r1;
                m4=r2;
                i3=i6;
                r4=dm(-3,i6);
                modify(i1,m4),r8=r4-1;
                modify(i3,-2163);
                r2=i3,r4=r8*r12;
                r2=r2+r4,r4=dm(-2,i6);
                r8=r4-1,i3=i6;
                r1=r2+r8,r2=dm(i1,m5);
                r8=dm(i0,m5);
                modify(i3,-2163);
                i2=r1;
                r4=dm(i2,m5);
                r12=r2*r4;
                r2=r8+r12;
                r8=120;
                r12=120;
                dm(i4,m5)=r2;
                r2=i3;
                r4=dm(-3,i6);
                r4=r4*r8;
                r2=r2+r4,r4=dm(-2,i6);
                r1=r2+r4,i3=i6;
                r4=dm(-3,i6);
                i4=r1,r8=r4-1;
                modify(i3,-2163);
                r2=i3,r4=r8*r12;
                r12=120;
                r2=r2+r4,r4=dm(-2,i6);
                r8=r4-1;
                r1=r2+r8,r2=dm(-3,i6);
                i0=r1;
                m4=r2;
                i3=i6;
                r4=dm(-3,i6);
                modify(i1,m4),r8=r4-1;
                modify(i3,-1083);
```

```
        i3=i6;
        modify(i3,-2163);
        r2=i3;
        r4=dm(-3,i6);
        r8=r4-1;
        r12=120;
        r4=r8*r12;
        r2=r2+r4;
        r4=dm(-2,i6);
        r8=r4-1;
        r1=r2+r8;
        i0=r1;
        r2=dm(-3,i6);
        m4=r2;
        modify(i1,m4);
        i3=i6;
        modify(i3,-1083);
        r2=i3;
        r4=dm(-3,i6);
        r8=r4-1;
        r12=120;
        r4=r8*r12;
        r2=r2+r4;
        r4=dm(-2,i6);
        r1=r2+r4;
        i2=r1;
        r2=dm(i1,m5);
        r4=dm(i2,m5);
        r12=r2*r4;
        r8=dm(i0,m5);
        r2=r8+r12;
        dm(i4,m5)=r2;

.ENDSEG;
```

```
                r2=i3,r4=r8*r12;
                r2=r2+r4,r8=dm(i0,m5);
                r4=dm(-2,i6);
                r1=r2+r4,r2=dm(i1,m5);
                i2=r1;
                r4=dm(i2,m5);
                r12=r2*r4;
                r2=r8+r12;
                dm(i4,m5)=r2;

.ENDSEG;
```

N_complex updates

Das Beispielprogramm n_comple berechnet einen *n_complex updates* Filter. Es wird das Produkt von zwei komplexen Zahlen A und B berechnet und zu einer komplexen Zahl C addiert.

$$D(i) = A(i) * B(i) + C(i)$$

mit $A(i)$, $B(i)$, $C(i)$ und $D(i)$ komplexe Zahlen und $i = 1, \ldots, N$.

```
.SEGMENT /PM pm_code;
.GLOBAL _n_comple;

_n_complex:   r2=dm(-6,i6);
              r4=15;
              comp(r2,r4);
              i4=dm(-5,i6);
              i0=dm(-4,i6);
              r1=i0;
              r5=1;
              r2=r1+r5;
              dm(-4,i6)=r2;
              i1=dm(-2,i6);
              r1=i1;
              r5=1;
              r2=r1+r5;
              dm(-2,i6)=r2;
              i2=dm(-3,i6);
              r1=i2;
              r5=1;
              r2=r1+r5;
              dm(-3,i6)=r2;
              r2=dm(i1,m5);
              r4=dm(i2,m5);
              r12=r2*r4;
              r8=dm(i0,m5);
              r2=r8+r12;
              dm(i4,m5)=r2;
              r2=dm(-5,i6);
              r1=1;
              r4=r2+r1;
              dm(-5,i6)=r4;
              i4=r2;
              i0=dm(-2,i6);
              r2=dm(-3,i6);
              r5=1;
              r4=r2-r5;
              dm(-3,i6)=r4;
              r4=dm(i0,m5);
              i3=r2;
              r2=dm(i3,m5);
              r12=r4*r2;
              r8=dm(i4,m5);
              r2=r8-r12;
              dm(i4,m5)=r2;
              i4=dm(-5,i6);
              i0=dm(-4,i6);
              r1=i0;
              r5=1;
              r2=r1+r5;
              dm(-4,i6)=r2;
              r2=dm(-2,i6);
              r1=1;
```

```
SEGMENT /pm    pm_code;
.GLOBAL _n_comple;

_n_complex:      r4=15;
                 r2=dm(-6,i6);
                 r5=1;
                 comp(r2,r4);
                 i4=dm(-5,i6);
                 i0=dm(-4,i6);
                 r1=i0;
                 r2=r1+r5;
                 r5=1;
                 dm(-4,i6)=r2;
                 i1=dm(-2,i6);
                 r1=i1;
                 r2=r1+r5;
                 r1=1;
                 dm(-2,i6)=r2;
                 i2=dm(-3,i6);
                 r1=i2;
                 r2=r1+r5;
                 r5=1;
                 r5=1;
                 dm(-3,i6)=r2;
                 r2=dm(i1,m5);
                 r4=dm(i2,m5);
                 r12=r2*r4,r8=dm(i0,m5);
                 r2=r8+r12;
                 dm(i4,m5)=r2;
                 r2=dm(-5,i6);
                 r4=r2+r1,i4=r2;
                 dm(-5,i6)=r4;
                 i0=dm(-2,i6);
                 r2=dm(-3,i6);
                 r4=r2-r5,i3=r2;
                 r5=1;
                 dm(-3,i6)=r4;
                 r4=dm(i0,m5);
                 r2=dm(i3,m5);
                 r12=r4*r2,r8=dm(i4,m5);
                 r2=r8-r12;
                 dm(i4,m5)=r2;
                 i4=dm(-5,i6);
                 i0=dm(-4,i6);
                 r1=i0;
                 r2=r1+r5;
                 r1=1;
                 dm(-4,i6)=r2;
                 r2=dm(-2,i6);
                 r4=r2-r1,i3=r2;
                 r1=1;
                 dm(-2,i6)=r4;
                 i1=dm(-3,i6);
```

```
                r4=r2-r1;
                dm(-2,i6)=r4;
                i1=dm(-3,i6);
                r5=i1;
                r1=1;
                r4=r5+r1;
                dm(-3,i6)=r4;
                i3=r2;
                r2=dm(i3,m5);
                r4=dm(i1,m5);
                r12=r2*r4;
                r8=dm(i0,m5);
                r2=r8+r12;
                dm(i4,m5)=r2;
                r2=dm(-5,i6);
                r1=1;
                r4=r2+r1;
                dm(-5,i6)=r4;
                i4=r2;
                i0=dm(-2,i6);
                r5=i0;
                r1=1;
                r2=r5+r1;
                dm(-2,i6)=r2;
                i1=dm(-3,i6);
                r5=i1;
                r1=1;
                r2=r5+r1;
                dm(-3,i6)=r2;
                r2=dm(i0,m5);
                r4=dm(i1,m5);
                r12=r2*r4;
                r8=dm(i4,m5);
                r2=r8+r12;
                dm(i4,m5)=r2;
   _L$8:        r2=dm(-6,i6);
                r4=r2+1;
                r2=r4;
                dm(-6,i6)=r2;
                r2=dm(-2,i6);
                r4=r2+1;
                r2=r4;
                dm(-2,i6)=r2;

.ENDSEG;
```

```
                r5=i1;
                r4=r5+r1;
                r1=1;
                dm(-3,i6)=r4;
                r2=dm(i3,m5);
                r4=dm(i1,m5);
                r12=r2*r4,r8=dm(i0,m5);
                r2=r8+r12;
                dm(i4,m5)=r2;
                r2=dm(-5,i6);
                r4=r2+r1,i4=r2;
                dm(-5,i6)=r4;
                r1=1;
                i0=dm(-2,i6);
                r5=i0;
                r2=r5+r1;
                dm(-2,i6)=r2;
                r1=1;
                i1=dm(-3,i6);
                r5=i1;
                r2=r5+r1;
                dm(-3,i6)=r2;
                r2=dm(i0,m5);
                r4=dm(i1,m5);
                r12=r2*r4,r8=dm(i4,m5);
                r2=r8+r12;
                dm(i4,m5)=r2;
   _L$8:        r2=dm(-6,i6);
                r4=r2+1;
                r2=r4;
                dm(-6,i6)=r2;
                r2=dm(-2,i6);
                r4=r2+1;
                r2=r4;
                dm(-2,i6)=r2;

.ENDSEG;
```

Anhang B

Tools

CIS

CIS (*Comparativ Instruktion Scheduler*) ist ein von Daniel Kästner [Käs97] und Marc Langenbach [Lan97] entwickeltes Tool und realisiert sowohl Verfahren der linearen Programmierung als auch klassische Algorithmen zur Lösung der Instruktionsanordnung- und Registerallokationsproblematik. Die Implementierung liegt als ausführbare Datei vor, die gewünschten Berechnungsverfahren lassen sich mittels Kommandozeilenoperationen auswählen. Als Ausgabe werden ganzzahlige lineare Programme im sogenannten $LP-$Format erzeugt, das im wesentlichen der algebraischen Darstellung des Optimierungsproblems entspricht. Im Rahmen dieser Diplomarbeit werden mit CIS Beispielprogramme für die optimale Lösung des Modellierungsansatz SILP erzeugt.

CPLEX

CPLEX ist eine Programmbibliothek zur Lösung von gemischt ganzzahligen linearen Programmen. Die von CIS erzeugten Dateien werden eingelesen und über Parametereinstellungen die Suchstrategien und Randbedingungen festgelegt. Als Ergebnis wird die Anzahl der benötigten Takte und die Berechnungszeit ausgegeben, über weitere Anfragen sind Informationen über die Variablenbelegung der gefundenen Lösung zugänglich. Zur Lösung von linearen Programmen wird der Simplexalgorithmus [Neu75] verwendet. Enthält das Programm ganzzahlige Variablen, wird eine Branch-and-Bound [GLN88] Methode eingesetzt, die vom Benutzer beeinflußt werden kann.

SILP2SCLP

Das Konvertierungsprogramm SILP2SCLP ist in Perl (*Practical Extraction and Report Language*) [Sch96] geschrieben. Es liest beim Aufruf das Arbeitsverzeichnis aus und gibt eine Liste aller Dateien mit der Endung “lp” aus. Damit kann das zu konvertierende Programm ausgewählt werden. Danach wird die Bearbeitung in drei Schritten durchgeführt. Zuerst erfolgt eine Zerlegung der Quelldatei in mehrere Einzeldateien, dabei werden die verschiedenen Constraintklassen in eigene Dateien kopiert um den nächsten Schritt zu erleichtern. Im zweiten Schritt wird die eigentliche Konvertierung durchgeführt, dazu müssen eine Reihe von Anpasssungen in der Syntax vorgenommen werden. Außerdem werden Listen aller vorkommenden Variablen erzeugt. Diese Informationen werden wiederum in Dateien zwischengespeichert. Im dritten und letzten Schritt werden die einzelnen Dateien wieder zusammengesetzt, zusätzlich wird eine ECPLiPSe Umgebung geschaffen und die nötigen Bibliotheken, Hilfsprogramme und die Lösungsstrategie V eingefügt. Die Ausgabedatei trägt die Endung "pl" und kann direkt mit ECLiPSe bearbeitet werden. Die Hilfsdateien werden anschließend wieder gelöscht.

ECLiPSe

ECLiPSe (*ECRC Constraint Logic Parallel System*) wurde am ECRC (*European Computer-Industry Research Centre*) in München entwickelt; die weitere Entwicklung und der Support erfolgt seit 1996 am IC-Parc (*Centre for Planning and Resource Control at the Imperial College in London*). ECLiPSe basiert auf der logischen Programmiersprache Prolog und bildet eine Plattform zur Programmierung in CLP. Es werden Techniken zur mathematischen und stochastischen Lösung von Aufgaben angeboten. Mit ECLiPSe werden unterschiedliche Aufgaben im Bereich der Planung, des Scheduling und der Ressourcenallokation gelöst.

Anhang C

Abkürzungen

ALU Arithmetic Logic Unit

CIS Comperativ Instruction Scheduler

CISC Complex Instruction Set Computer

CLP Constraint Logic Programming

CLP-FD Constraint Logic Programming Finite Domain

CP Constraint Propagation

CSP Constraint Satisfaction Problem

DMA Direct Memory Access

DSP Digitaler Signalprozessor

ECLiPSe ECRC Constraint Logic Parallel System

ILP Integer Linear Programming

IR Intermediate Representation, Zwischendarstellung

IS Instruktionsanordnung

ISRA Instruktionsanordnung unter Berücksichtigung von Ressourcenschranken

LP Linear Programm, lineares Programm

OASIC Optimal Architecture Synthesis with Interface Constraints

RISC Reduced Instruction Set Computer

SAIF schrittweise Approximation der isolierten Flußanalyse

SCLP Kombination aus SILP und CLP

SILP Scheduling and allokation with Integer Linear Programming

VGI Variante der geteilten Instruktionsmenge

VGV Variante der geteilten Variablenmengen

VML Variante der mehrfachen Lösungssuche

Literaturverzeichnis

[AA97] u.a. Abderrahamane Aggoun. *ECLiPSe, User Manual, Release 4.1.* International Computers Limited and IC-Parc, 1997.

[Bas95] Steven Bashford. *Code Generation Techniques for Irregular Architectures.* Report No. 596, Lehrstuhl Informatik XII, University of Dortmund, 1995.

[Bie95] Ulrich Bieker. *Retargierbare Compilierung von Selbsttestprogrammen digitaler Prozessoren mittels Constraint-logischer Programmierung.* Dissertation, Universität Dortmund, 1995.

[Bra88] Ivan Bratko. *PROLOG Programmierung für künstliche Intelligenz.* Addison-Wesley, 1988.

[Bra94] Thomas S. Braisier. *FRIGG: An New Approach to Combining Register Assignment and Instruktion Scheduling.* Michigan Technological University, 1994.

[Bri92] Preston Briggs. *Register Allocating via Graph Coloring.* PhD thesis, Rice University, Huston, Texas, 1992.

[Bro89] Semendjajew Bronstein. *Taschenbuch der Mathematik.* Verlag Harri Deutsch, Thun, 1989.

[Cas95] G. Castelli. *The Seemilingly Unlimited Market for Mikrocontroller-Based Embedded Systems.* IEEE Micro, 15(5):6-8, 1995.

[CHG92] M. I. Elmasry C. H. Gebotys. *Optimal VLSI Architectural Synthesis.* Kluwer Academic, 1992.

[Dan66] G. B. Dantzig. *Lineare Programmierung und Erweiterungen.* Berlin - Heidelberg - New York, 1966.

[DL80] Bruce Shriver und Patrick W. Mallet David Landskov, Scott Davidson. *Local Microcode Compaction Techniques.* ACM Computing Surveys, 12(3): 261-294, 1980.

[Fis81] J. A. Fisher. *Trace scheduling: A technique for global microcode compaction.* IEEE Transactions on Computers, C-30(7):478-490, 1981.

[Gas89] F. Gasperoni. *Compilation Techniques for VLIW-Architectures.* Technischer Report, Courant Institute of Mathematical Science, New York University, 1989.

[GLN88] L. A. Wolsey G. L. Nehmhauser. *Integer and Combinatorial Optimization*. John Wiley and Sons, 1988.

[Güt92] Ralf Hartmut Güting. *Datenstrukturen und Algorithmen*. B.G. Teubner Stuttgart, 1992.

[Had74] G. Hadley. *Linear Programming*. Amsterdam - London, 1974.

[HKB88] S. Schmitgen H. Kleine Büning. *Prolog*. B. G. Teubner Stuttgart, 1988.

[ILO98] Inc ILOG. *CPLEX 6.0, Installation and Use Notes*. ILOG, 889 Alder Avenue, Suite 200, Incline Village, USA, 1998.

[JLH94] David A. Patterson John L. Henneyse. *Rechnerarchitektur*. Vieweg, 1994.

[JS99] Mark Wallace Joachim Schimpf, Kish Shen. *Tutorial on Search in ECLiPSe*. Imperial College London, 1999.

[Käs97] Daniel Kästner. *Instruktionsanordnung und Registerallokation auf der Basis ganzahliger linearer Programmierung für den digitalen Signalprozessor ADSP-2106x*. Diplomarbeit, Universität des Saarlandes, 1997.

[Lan97] Marc Langenbach. *Instruktionsanordnung unter Verwendung graphbasierter Algorithmen für den digitalen Signalprozessor ADSP-2106x*. Diplomarbeit, Universität des Saarlandes, 1997.

[Lan98] Birger Landwehr. *ILP-basierte Mikroarchitektur-Syntese mit komplexen Bausteinbibliotheken*. Doktorarbeit, VDI Verlag, 1998.

[Mei96] Micha Meier. *ProTcXl 2.1 User Manual*. European Computer Industry Research Centre, Arabellastr. 17, 81925 Munich, Germany, 1996.

[MRG79] D. S. Johnson M. R. Garey. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. Freeman and Company, New York, 1979.

[MS98] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1997.

[Neu75] K. Neumann. *Operations Research Verfahren, Band I*. Carl Hanser Verlag, 1975.

[PB97] u.a. Pascal Brisset. *ECLiPSe, User Manual, Release 4.1*. International Computers Limited and IC-Parc, 1997.

[PR97] Gustav Pomberger Peter Rechenberger. *Informatik Handbuch*. Carl Hanser Verlag München Wien, 1997.

[Pro91] PrologIA. *Prolog III Version 1.3, Referenz Manual*. Parc Technologique de Luminy - Case 919, 13288 Marseille Cedex, France, 1991.

[RW92] Dieter Maurer Reinhard Wilhelm. *Übersetzerbau; Theorie, Konstruktion, Generierung.* Springer Verlag, 1992.

[SA91] COSYTEC SA. *CHIP Users Guide, Version 1.1.* Parc Club Orsay Universite, 4, rue Jean Rostand, 91893 Orsay Cedex, France, 1991.

[Sch96] Randal L. Schwartz. *Einführung in Perl.* OReilly, 1996.

[Tah71] A. H. Taha. *Operations Research.* New York, 1971.

[Tei97] Jürgen Teich. *Digitale Hardware/Software-Systeme.* Springer Verlag, 1997.

[TF97] Slim Abennadher Thom Frühwirth. *Constraint-Programmierung.* Springer Verlag, Berlin Heidelberg New York, 1997.

[TO96] P. Widmayer T. Ottmann. *Algorithmen und Datenstrukturen.* Spektrum, Akademischer Verlag, 1996.

[u.a93] Thom Frühwirth u.a. Constrain logic programming - an informal introduction. Technical report, European Computer Industry Research Centre, 1993.

[vH89] Pascal van Hentenryck. *Constraint Satisfaction in logic Programming.* Massachusetts Institut of Technology, 1989.

[VZS95] M. Williams V. Zivojnovic, H. Schraut and R. Schoenen. *DSPs, GPPs and Multimedia Applications - An Evaluation Using DSPstone.* Proceedings of the International Conferenz on Signal Processing Applications and Technology, Seiten 1779 -1783, DSP Associates, 1995.

[Wal95] Mark Wallace. Constraint programming. Technical report, William Penney Laboraty,Imperial College,LONDON SW7 2AZ, 1995.

[WD92] Klaus Kleibohm Walter Dürr. *Operations Research, Lineare Modelle und ihre Anwendungen.* Carl Hanser Verlag München Wien, 1992.

[WNS97] Mark Wallace, Stefano Novello, and Joachim Schimpf. Eclipse: A platform for constraint logic programming. Technical report, William Penney Laboraty, Imperial College, LONDON SW7 2AZ, 1997.

[Zha96] L. Zhang. *Scheduling and Allocating with Integer Linear Programming.* Doktorarbeit, Technische Universität des Saarlandes, 1996.

Index