Hochschule für Technik, Wirtschaft und Kultur Leipzig (FH)
Fachbereich Informatik, Mathematik und Naturwissenschaften

Diplomarbeit

Codegenerierung für variable Zielarchitekturen mit evolutionären Algorithmen

Holger Arnold

Eingereicht am 16. Oktober 2001

Betreut durch
Prof. Dr. Heinrich Krämer und
Prof. Dr. Bernd Engelmann

Inhaltsverzeichnis

Symbolverzeichnis

$G_P = (O_P \cup V_P, D_P)$	Programmgraph
$G_I = (I_I \cup V_I, D_I)$	Befehlsgraph
$P_I = (O_I \cup V_I, D_I)$	Befehlsmuster
$\omega(o) \in \Omega$	Operationstyp der Operation o aus der Menge der Operationstypen
$\delta(v) \in \Delta$	Datentyp des Wertes v aus der Menge der Datentypen
$\theta(v) \in \Theta$	Speicherklasse des Wertes v aus der Menge der Speicherklassen
$I(G_P)$	Menge der Eingabewerte des Programmgraphen G_P
$O(G_P)$	Menge der Ausgabewerte des Programmgraphen G_P
$I_A(v)$	Aktivitätsintervall des Wertes v
$A(t)$	Menge der zum Zeitpunkt t aktiven Werte
$M = (n_e, I_M, M_M, R_M)$	Maschinenbeschreibung
$R(v)$	Menge der Register, die dem Wert v zugeordnet werden können
$\sigma : O_P \times I_M \to \{0, 1\}$	Befehlsauswahl
$\rho : V_I^R \times R_M \to \{0, 1\}$	Registerzuordnung
$\gamma : I_I \times \mathbb{N} \to \{0, 1\}$	Befehlsplan
$l(\gamma)$	Länge des Befehlsplans γ
$D(i)$	Menge der Befehle, von denen der Befehl i direkt abhängt
$D^*(i)$	Menge der Befehle, von denen der Befehl i direkt oder indirekt abhängt
$G(i)$	Genom des Individuums i
$\|G\|$	Anzahl der Gene im Genom G
$o : S \to \mathbb{R}$	Zielfunktion, S: Menge der möglichen Lösungen
$f : I \to \mathbb{R}$	Bewertungsfunktion, I: Menge der möglichen Individuen
$f_r(i, P)$	relative Bewertung des Individuums i zur Population P

1 Einleitung

Leistungsfähige Multimedia-Anwendungen, ständig verfügbare Kommunikationsmittel und die Echtzeit-Steuerung von Prozessen spielen eine wichtige Rolle in unserem täglichen Leben. Medizintechnik, Fahrzeugbau, Sicherheitstechnik, Telekommunikation, Prozess-Steuerung und Unterhaltungselektronik sind Beispiele für Bereiche, in denen immer leistungsfähigere Systeme auf der Basis integrierter Schaltungen entwickelt werden. An die Komponenten dieser Systeme werden besondere Anforderungen gestellt: niedrige Kosten (und damit bei integrierten Schaltungen ein geringer Flächenbedarf) und geringer Stromverbrauch auf der einen Seite, hohe Leistungsfähigkeit auf der anderen Seite.

Zunehmend bilden leistungsfähige anwendungsspezifische Prozessoren die Basis für derartige Systeme. Allerdings kann jedes System nur so gut sein, wie die Werkzeuge, die zu seiner Entwicklung zur Verfügung stehen. Diese Diplomarbeit beschäftigt sich mit der Codegenerierung als letzter Stufe der Kette von Entwicklungswerkzeugen. Dabei werden insbesondere Prozessormodelle betrachtet, wie sie in den genannten Systemen zum Einsatz kommen.

1.1 Codegenerierung

Sofern eine Anwendung nicht direkt in der Assembler-Sprache des Zielprozessors entwickelt werden soll, benötigt man die Hilfe eines Compilers, der die verwendete Programmiersprache in die Maschinensprache des Prozessors übersetzt. Abbildung 1.1 (aus [3]) zeigt den grundlegenden Aufbau eines Compilers.

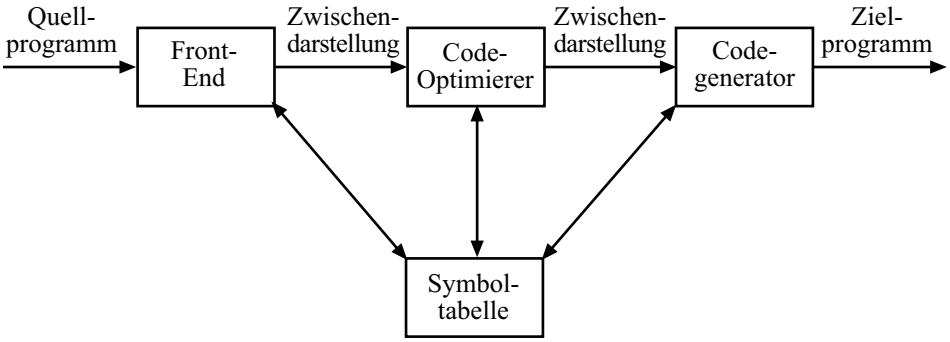


Abbildung 1.1: Aufbau eines Compilers

Aus dem eingegebenen Programmtext baut das Front-End des Compilers zunächst eine Zwischendarstellung auf. Dabei werden die im Programm verwendeten Bezeichner in die Symboltabelle[1] eingetragen. Die Zwischendarstellung und die Symboltabelle sind die Datenstrukturen, mit denen die weiteren Compilerstufen arbeiten. Ein Codeoptimierer kann versuchen, die Zwischendarstellung so zu verändern, dass diese ein dem eingegebenen Programm äquivalentes, aber hinsichtlich bestimmter Kriterien günstigeres Programm darstellt. Der Codegenerator ist schließlich dafür zuständig, aus der ihm gegebenen Zwischendarstellung ein Maschinenprogramm für den Zielprozessor zu erzeugen.

Diese Arbeit beschäftigt sich nur mit der letzten Stufe eines Compilers, dem Codegenerator. Insbesondere beschäftigen wir uns mit Codegeneratoren für Prozessorenarchitekturen, die von Standardcompilern nur mangelhaft unterstützt werden: Prozessoren mit Befehlsparallelismus und komplexen Befehlen. Beispiele für derartige Architekturen finden wir in digitalen Signalprozessoren (DSPs), Multimediaprozessoren mit speziellen Befehlssätzen zur Verarbeitung von Mediendaten und anderen anwendungsspezifischen Prozessoren (ASIPs[2]). Einen guten Überblick über den Entwurf und die Implementierung von Compilern für Standardprozessoren bietet [3]. Darin werden alle Stufen eines Compilers behandelt und verschiedene Verfahren zu ihrer Implementierung angegeben.

Die Compiler, die für gängige DSPs erhältlich sind, unterstützen deren spezielle Fähigkeiten nur unzureichend. Der von ihnen erzeugte Code ist teilweise von so schlechter Qualität, dass den Systementwicklern oft nur die Möglichkeit bleibt, vollständig in Assembler zu entwickeln, mit allen bekannten Konsequenzen hinsichtlich Fehleranfälligkeit und Portierbarkeit. Die Gründe für die schlechte Codequalität sind die Prozessormodelle auf denen diese Compiler basieren und die daraus abgeleiteten Verfahren. Wir werden darauf in Kapitel 2 noch zu sprechen kommen.

Ein anderes Problem ist, dass es für viele Prozessoren überhaupt keine Compiler gibt. Insbesondere anwendungsspezifische Prozessoren, die dem Systementwickler als reine Prozessormodelle, sogenannte Cores, geliefert werden, sind davon betroffen. Die Einbettung in ein System aus Komponenten, die dem Prozessorentwickler unbekannt sind, verhindert hier die Entwicklung optimierender Compiler.

1.2 Evolutionäre Algorithmen

Bereits in den 60er Jahren wurden verschiedenste Verfahren vorgeschlagen, die Prinzipien der biologischen Evolution — Selektion, Mutation und Rekombination — mit computertechnischen Verfahren nutzbar zu machen. Ursprünglich

[1] Die Bezeichnung Symboltabelle hat historische Gründe. Gemeint ist jede Datenstruktur, die symbolische Namen mit Zusatzinformationen über Datentypen, Gültigkeitsbereiche etc. verbindet.

[2] ASIP: Application-Specific Instruction Set Processor

waren diese Verfahren als Mittel zur Simulation biologischer Prozesse gedacht.[3] Bald wurde jedoch deutlich, dass die Prinzipien der Evolution auch zur Lösung technischer Optimierungsprobleme mit Erfolg eingesetzt werden können. Inzwischen wurden evolutionäre Algorithmen bei vielen schwierigen Optimierungsproblemen angewendet und haben sich als stabile Lösungsmethode erwiesen.

Die Verfahren, die sich der Evolutionsprinzipien im weitesten Sinne zur Optimierung bedienen, basieren auf drei grundlegenden Ansätzen: den genetischen Algorithmen nach Holland [13] und Goldberg [8], den evolutionären Strategien nach Rechenberg [23] und der genetischen Programmierung nach Koza [14]. Wir werden die Vor- und Nachteile der verschiedenen Ansätze an dieser Stelle nicht diskutieren. In [12] wird versucht, die Eigenschaften von genetischen Algorithmen und Evolutionsstrategien durch ihre Anwendung auf eine Reihe von Testfunktionen zu vergleichen. Wir werden in dieser Arbeit ein Verfahren einsetzen, dass sich sich an den datenstrukturbasierten genetischen Algorithmen orientiert, wie sie zum Beispiel in [16] beschrieben sind.

1.3 Ziele und Überblick

Diese Diplomarbeit verfolgt zwei Ziele:

1. Es soll ein Verfahren zur Codegenerierung entwickelt werden, das von einem konkreten Zielprozessor unabhängig ist. Die Beschreibung der Zielarchitektur soll Teil der Eingabedaten des Codegenerators sein. Bei der Umstellung auf eine andere Zielarchitektur soll keine Modifikation oder Neuübersetzung des Codegenerators erforderlich sein. Trotz der Unabhängigkeit von einer konkreten Prozessorarchitektur soll der von diesem Codegenerator erzeugte Code die speziellen Eigenschaften der Zielarchitektur möglichst gut ausnutzen. Das betrifft insbesondere den Befehlsparallelismus und komplexe Befehle spezieller Einheiten des Zielprozessors.

2. Es soll ermittelt werden, ob und in welcher Weise evolutionäre Algorithmen dazu geeignet sind, die bei der Codegenerierung entstehenden Optimierungsprobleme zu lösen.

Zum Aufbau der Diplomarbeit: Zunächst werden wir uns in Kapitel 2 mit den Grundlagen der Codegenerierung beschäftigen, die zum Verständnis der Arbeit erforderlich sind. Es wird beschrieben, wie zu übersetzende Programme dargestellt werden und wie die Beschreibung der Zielarchitektur erfolgt. Außerdem werden die bei der Codegenerierung zu lösenden Probleme definiert. Der evolutionäre Algorithmus, der die Basis für unseren Codegenerator bildet, ist das Thema von Kapitel 3. Es werden die allgemeinen Konzepte der evolutionären Algorithmen vorgestellt, die wir in unserem Codegenerator einsetzen. Kapitel 4 beschreibt anschließend detailliert die Implementierung des Codegenerators und

[3] Eines der ersten Beispiele dafür ist [29].

insbesondere die Anwendung der in den vorangehenden Kapiteln beschriebenen Konzepte. Die eingesetzten evolutionären Operatoren spielen dabei eine besondere Rolle. In Kapitel 5 werden die Experimente beschrieben, mit denen der entwickelte Codegenerator untersucht wurde. Aus den Ergebnissen werden entsprechende Schlussfolgerungen gezogen. Die gewonnenen Erkenntnisse werden in Kapitel 6 zusammengefasst. Darin wird der Beitrag dieser Diplomarbeit herausgestellt und abschließend mögliche Ansatzpunkte für weitere Untersuchungen genannt.

2 Grundlagen der Codegenerierung

Zum Verständnis dieser Arbeit sind einige Grundlagen der Codegenerierung erforderlich, die in diesem Kapitel behandelt werden. Zunächst wird es um die Darstellung des zu übersetzenden Programms gehen. Unser Verfahren soll unabhängig von einer konkreten Zielarchitektur arbeiten. Dem Codegenerator muss deshalb beim Übersetzungsvorgang ein Modell der Zielarchitektur als Eingabe zur Verfügung stehen. In diesem Kapitel wird beschrieben, wie ein solches Architekturmodell formuliert werden kann. Danach werden die Probleme definiert, die bei der Codegenerierung zu lösen sind. Abschließend werden zwei grundlegende Klassen von Codegeneratoren vorgestellt (die wir klassische und nichtklassische nennen) und einige ihrer Eigenschaften beschrieben.

2.1 Die Darstellung des Programms

Das zu übersetzende Quellprogramm wird dem Codegenerator in Form einer Zwischendarstellung übergeben, die vom Compiler-Front-End erzeugt wird. Informationen über die in der Zwischendarstellung verwendeten Namen werden in einer Symboltabelle bereitgestellt. Für die Form der Zwischendarstellung gibt es verschiedene Alternativen, wir wählen eine Darstellung mit einem *Kontroll-/Datenfluss-Graphen*. Das ist ein Graph, dessen Kanten den Kontrollfluss des Programms darstellen und dessen Knoten die Datenflussgraphen der Grundblöcke des Programms sind. Ein *Grundblock* (engl. basic block, [3]) ist eine Folge fortlaufender Anweisungen, in die der Kontrollfluss nur am Anfang eintritt und die er nur am Ende verlässt. Abbildung 2.1 zeigt den Kontrollfluss-Teil eines Kontroll-/Datenfluss-Graphen mit den Grundblöcken B_1, B_2, B_3 und B_4. Der Block B_1 ist, da er mehr als eine ausgehende Kante besitzt, ein Verzweigungsblock. An den Verzweigungskanten sind die Bedingungen angegeben, unter denen der Kontrollfluss diese Kanten auswählt (c und $\neg c$).

Wir nehmen an, dass der Kontroll-/Datenfluss-Graph bereits von einem Codeoptimierer durch verschiedene Transformationen so weit optimiert wurde, wie es ohne detaillierte Informationen über die Zielarchitektur möglich ist. Desweiteren nehmen wir an, dass die Operationen des Programms bereits in eine für die gewählte Zielarchitektur passende Form gebracht wurden. Dazu gehört zum Beispiel das Zerlegen von Operationen, die über die Verarbeitungsbreite des Prozessors hinausgehen, in mehrere Teiloperationen, oder das Ersetzen von Operationen, die auf dem Zielprozessor nicht implementiert sind. Dafür sind

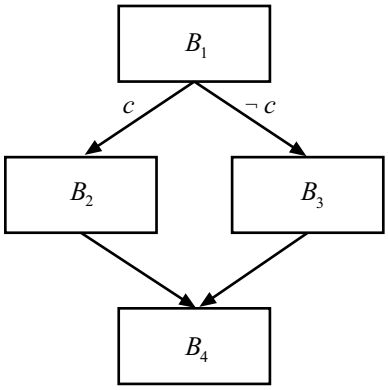


Abbildung 2.1: Kontrollfluss-Teil eines Kontroll-/Datenfluss-Graphen

nur wenige Informationen über die Zielarchitektur erforderlich, zum Beispiel die Wortlänge des Zielprozessors und die vorhandenen Operationen.

Da der Kontroll-/Datenfluss-Graph also keinen weiteren Transformationen unterworfen werden muss, und da unser Augenmerk auf der Optimierung des Datenflusses eines Programms liegt, scheint es vernünftig zu sein, die Grundblöcke des Programms getrennt zu übersetzen. Wir werden daher im folgenden stets einen einzelnen zu übersetzenden Grundblock betrachten und wenden uns nun der Datenstruktur zu, mit der ein solcher Grundblock dargestellt wird.

2.1.1 Der Programmgraph

Ein zu übersetzender Grundblock wird in unserem Codegenerator durch einen Programmgraphen dargestellt, der die Datenfluss-Informationen dieses Grundblocks enthält.

Ein *Programmgraph* $G_P = (O_P \cup V_P, D_P)$ ist ein gerichteter azyklischer Graph mit den Knotenmengen O_P und V_P und der Kantenmenge D_P. O_P ist die Menge der Operationen des Programmgraphen, V_P ist die Menge der Werte. $D_P \subset (O_P \times V_P) \cup (V_P \times O_P)$ ist die Menge der direkten Datenabhängigkeiten zwischen Operationen und Werten. Abbildung 2.2 zeigt den Programmgraphen eines Grundblocks, der die Anweisung[1] $s = s + (*ap++) * (*bp++)$ realisiert, wie sie beispielsweise bei der Berechnung eines Skalarprodukts auftreten könnte (ap und bp sind Zeiger auf die Elemente der zu multiplizierenden Vektoren, s enthält das Teilskalarprodukt, die Konstante C entspricht der Zahl der Bytes, die ein Adresswert auf der Zielarchitektur belegt).

Jeder Operation o des Programmgraphen ist ein *Operationstyp* $\omega(o)$ aus der Menge der Operationstypen Ω zugeordnet. Die verschiedenen Operationstypen sind in Tabelle 2.1 aufgeführt.

[1] Programmbeispiele werden, falls nicht explizit etwas anderes erwähnt wird, in der Syntax der Programmiersprache C angegeben.

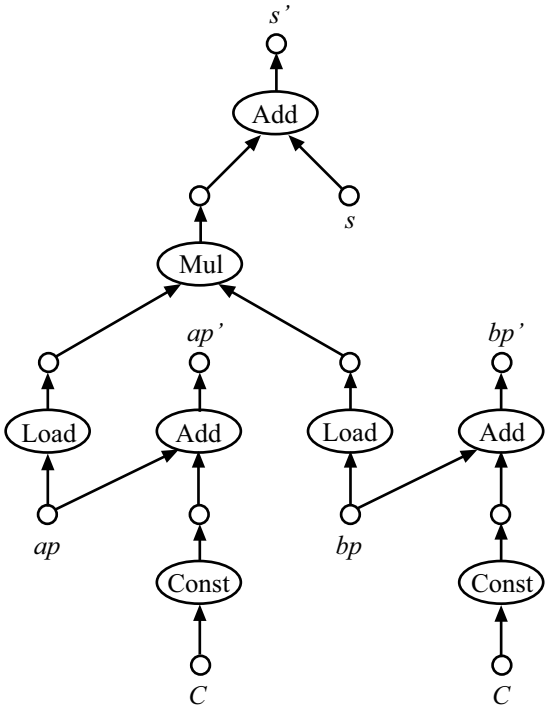


Abbildung 2.2: Beispiel für einen Programmgraphen

Jedem Wert v des Programmgraphen ist ein *Datentyp* $\delta(v)$ aus der Menge der Datentypen Δ zugeordnet. Ein Datentyp A heißt in einem Datentyp B *enthalten* und wir schreiben $A \subseteq B$, wenn sich alle Werte, die durch einen Wert vom Datentyp A darstellbar sind, auch durch einen Wert vom Datentyp B darstellen lassen. Der Datentyp eines Wertes entspricht dem Typ, den dieser Wert im zu übersetzenden Programm besitzt. Zu beachten ist, dass das Compiler-Front-End bereits alle Datenstrukturen auf die Basisdatentypen reduziert. Außerdem ist es wichtig daran zu denken, dass der Datentyp eines Wertes keine Informationen über die tatsächliche Darstellung dieses Typs auf dem Zielprozessor enthält. Man könnte daher auch von logischen Datentypen sprechen. Die verschiedenen Datentypen sind in Tabelle 2.2 aufgelistet.

Jeder Wert v des Programmgraphen gehört zu einer bestimmten *Speicherklasse* $\theta(v)$ aus der Menge der Speicherklassen Θ. Die Speicherklasse eines Wertes gibt an, ob es sich bei Zugriffen auf diesen Wert um einen Speicher- oder Registerzugriff handelt, oder ob der Wert direkt als Konstante in das Programm einzufügen ist. Die Speicherklassen sind in Tabelle 2.3 aufgeführt.

Eine Teilmenge der Werte ist als Menge der *Eingabewerte* $I(G_P)$ des Programmgraphen, eine andere Teilmenge der Werte als Menge der *Ausgabewerte* $O(G_P)$ des Programmgraphen gekennzeichnet.

Ein wichtiger Begriff ist der Begriff der Aktivität eines Wertes. Sei t_p der

Add	Addition
Addc	Addition mit Überlauf
Sub	Subtraktion
Subb	Subtraktion mit Unterlauf
Mul	Multiplikation
Div	Division
Mod	Modulo
Neg	Negation
And	bitweises Und
Or	bitweises Oder
Xor	bitweises ausschließendes Oder
Not	bitweise Inversion
Shl	bitweises links-Schieben
Shr	bitweises rechts-Schieben
Const	Zugriff auf eine Konstante
Conv	Datentyp-Konvertierung
Move	Kopieren eines Wertes ohne Speicherzugriff
Load	Lesen eines Wertes aus dem Speicher
Store	Schreiben eines Wertes in den Speicher

Tabelle 2.1: Operationstypen

Char	Zeichen
Short	kurze Ganzzahl mit Vorzeichen
Int	mittlere Ganzzahl mit Vorzeichen
Long	lange Ganzzahl mit Vorzeichen
UShort	kurze Ganzzahl ohne Vorzeichen
UInt	mittlere Ganzzahl ohne Vorzeichen
ULong	lange Ganzzahl ohne Vorzeichen
Address	Speicheraddresse ohne Vorzeichen
Bit	einzelnes Bit
Float	Fließkommazahl einfacher Genauigkeit
Double	Fließkommazahl doppelter Genauigkeit

Tabelle 2.2: Datentypen

Register	ein Wert, der in einem Register gespeichert werden soll
Memory	ein Wert, auf den direkt im Speicher zugegriffen wird
Constant	ein konstanter Wert, der direkt in das Programm eingefügt wird (es handelt sich also um ein Literal)

Tabelle 2.3: Speicherklassen

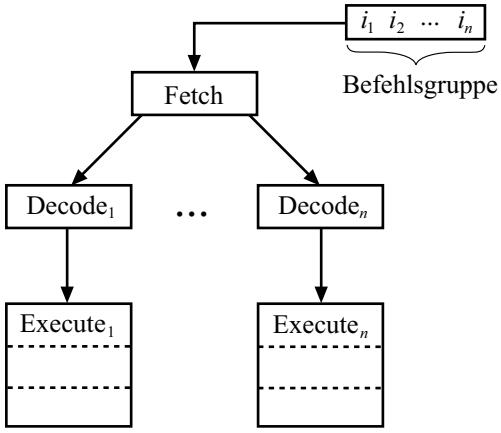


Abbildung 2.3: Das verwendete Prozessormodell

Zeitschritt, in dem der Wert v berechnet wird und t_l der Zeitschritt, in dem der Wert v zum letzten Mal als Operand verwendet wird. Dann heißt das halboffene Intervall $I_A(v) = [t_p, t_l)$ das *Aktivitätsintervall* (engl. lifetime interval) des Wertes v. Innerhalb dieses Zeitintervalls heißt der Wert v *aktiv* (engl. live). Die Menge der zu einem Zeitschritt t aktiven Werte bezeichnen wir mit $A(t)$:

$$A(t) = \{v | t \in I_A(v)\} \tag{2.1}$$

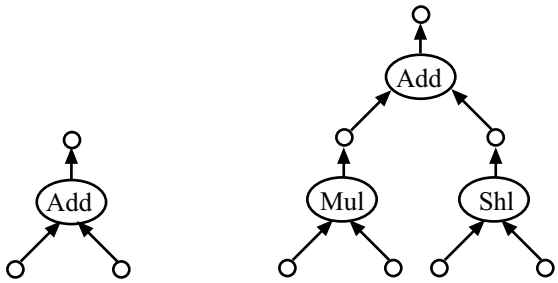
2.2 Das Prozessormodell

Unser Codegenerator soll bezüglich des Zielprozessors frei konfigurierbar sein. Trotzdem müssen wir einige Annahmen über die Zielarchitektur treffen, um eine Basis zu haben, auf der wir arbeiten können. Abbildung 2.3 skizziert das von uns zugrunde gelegte Prozessormodell.

Die *Maschinenbeschreibung* $M = (n_e, I_M, M_M, R_M)$ eines Prozessors besteht aus der Anzahl der parallelen Ausführungspfade n_e, dem Befehlssatz des Prozessors I_M, der Menge der Speicher M_M und der Menge der Register R_M.

Die Fetch-Stufe unseres Prozessormodells holt in jedem Zyklus parallel n_e Befehle. Jeder Ausführungspfad besitzt eine eigene Decodiereinheit und eigene Ausführungseinheiten. Wir orientieren uns mit unserem Prozessormodell an VLIW[2]-Architekturen, wie sie bei moderneren DSPs und ASIPs zunehmend eingesetzt werden. Diese Architekturen bilden aus jeweils n_e aufeinanderfolgenden Befehlen eine Befehlsgruppe und führen die Befehle einer Gruppe parallel aus. Jede Stelle einer Befehlsgruppe kann einen Befehl einer bestimmten Ausführungseinheit des Prozessors aufnehmen. Da die Gruppierung der Befehle explizit im Maschinencode enthalten ist, muss ein Compiler für eine derartige Architektur die parallele Ausführung der Befehle genau planen.

[2]VLIW: Very Long Instruction Word



(a) einfacher Befehl (b) komplexerer Befehl

Abbildung 2.4: Beispiele für Befehle

Wir werden zunächst annehmen, dass die Ausführung eines Befehls genau einen Taktzyklus in Anspruch nimmt. Außerdem werden wir Pipeline-Konflikte vorerst ignorieren. Diese beiden Vereinfachungen sind allerdings nicht prinzipieller Natur.

2.2.1 Darstellung des Befehlssatzes

Der *Befehlssatz* I_M eines Prozessors ist die Menge der Befehle, die dieser Prozessor ausführen kann. Die Semantik eines Befehls wird durch einen Ausdrucksbaum, das Befehlsmuster angegeben.

Ein *Befehlsmuster* $P_I = (O_I \cup V_I, D_I)$ ist ein Baum mit den Knotenmengen O_I und V_I und der Kantenmenge D_I. O_I ist die Menge der Operationen des Befehlsmusters, V_I ist die Menge der Werte. $D_I \subset (O_I \times V_I) \cup (V_I \times O_I)$ ist die Menge der direkten Datenabhängigkeiten zwischen Operationen und Werten. Abbildung 2.4 zeigt die Befehlsmuster zweier Befehle.

Die Werte, die keine Eingangskanten besitzen, bezeichnen wir als Menge der *Eingabewerte* $I(P_I)$, den Wert, von dem keine Kanten ausgehen (den Wurzelknoten des Ausdrucksbaumes), als *Ausgabewert* $v_o(P_I)$ des Befehlsmusters.

Die Beschreibung eines Befehls enthält außer dem Befehlsmuster noch für jeden Registerwert[3] v des Befehlsmusters die Menge der Register $R(v)$, die diesem Wert zugeordnet werden können, sowie die Codierung des Befehls. Weitere Informationen, zum Beispiel die Assemblersyntax, können ebenfalls Teil der Beschreibung eines Befehls sein.

Jedem Befehl ist genau eine Ausführungseinheit zugeordnet. Das bedeutet, dass zwei Befehle mit gleicher Semantik, die auf verschiedenen Einheiten ausgeführt werden, zwei verschiedene Befehle darstellen. Nur wenn zwei Befehle verschiedenen Einheiten zugeordnet sind, können sie prinzipiell parallel ausgeführt werden. Ressourcenbeschränkungen, zum Beispiel hinsichtlich der verfügbaren Register, können diese parallele Ausführbarkeit jedoch einschränken.

[3] Registerwerte sind Werte mit der Speicherklasse *Register*, siehe dazu Abschnitt 2.1.1.

2.2.2 Darstellung der Speicher- und Registerstruktur

In unserem Prozessormodell hat ein Prozessor Zugriff auf eine Menge von *Speichern*, die wir mit M_M bezeichnen. Die Adressbereiche dieser Speicher dürfen sich nicht überlappen. Unterschieden werden die Speicher im allgemeinen durch Decodierung einiger Bits der Speicheradresse am Adressbus des Prozessors.

Die Speicherstruktur ist bei der Frage von Bedeutung, wie die Daten des zu übersetzenden Programms auf die verschiedenen Speicher verteilt werden. Die Optimierung der Speicherzugriffe ist besonders bei DSPs ein wichtiger Aspekt, der zum Beispiel in [4] und [27] ausführlich betrachtet wird. In [30] wird ein auf Simulated Annealing basierendes Verfahren dafür vorgestellt und mit anderen Verfahren verglichen. Wir wollen dieses Problem jedoch nicht betrachten und setzen im weiteren voraus, dass den Daten des zu übersetzenden Programms bereits Speicherplätze zugewiesen wurden.

Neben den Speichern besitzt ein Prozessor in unserem Modell eine Menge von *Registern* R_M. Jedem Register ist ein Datentyp zugeordnet, der bestimmt, welche Werte dieses Register aufnehmen kann. Teilmengen der Register können zu *Registerklassen* zusammengefasst werden. Wir nehmen an, dass jedes Register in einem Zeitschritt nur von einer Ausführungseinheit beschrieben werden kann, die Anzahl der Leseoperationen aber nicht begrenzt ist.

2.3 Problemdefinition

Wir haben jetzt definiert, wie das zu übersetzende Programm bzw. ein einzelner Grundblock des Programms dargestellt wird und wie man die Zielarchitektur modelliert, für die Code generiert werden soll. Als nächstes werden wir die Probleme definieren, die bei der Codegenerierung zu lösen sind. Diese Arbeit beschäftigt sich mit folgenden drei Codegenerierungsproblemen: der Auswahl der Befehle, der Zuordnung von Werten zu Registern und der Ablaufplanung der Befehle. Das sind die Teilaufgaben, die im wesentlichen auf der Ebene von Grundblöcken gelöst werden können. Aufgaben höherer Ebenen, wie zum Beispiel die Optimierung der Speicherzuweisung an Variablen, sind nicht Thema dieser Arbeit.

2.3.1 Befehlsauswahl

Die abstrakten Operationen des Programmgraphen müssen letztendlich durch konkrete Befehle des Zielprozessors ausgeführt werden. Die Aufgabe der Befehlsauswahl besteht darin, für jede Operation des Programmgraphen einen Befehl aus dem Befehlssatz des Prozessors auszuwählen, der diese Operation ausführt. Mehrere Operationen können dabei in einem Befehl zusammengefasst werden.

Formal verstehen wir unter einer *Befehlsauswahl* für einen Programmgraphen $G_P = (O_P \cup V_P, D_P)$ und eine Maschinenbeschreibung $M = (n_e, I_M, M_M, R_M)$

eine Funktion $\sigma : O_P \times I_M \rightarrow \{0, 1\}$ mit

$$\sigma(o, i) = \begin{cases} 1 & \text{wenn der Operation } o \text{ der Befehl } i \text{ zugeordnet ist,} \\ 0 & \text{sonst.} \end{cases} \tag{2.2}$$

Eine Befehlsauswahl heißt *gültig*, wenn jeder Operation genau ein Befehl zugeordnet ist, wenn also gilt:

$$\forall o \in O_P : \sum_{i \in I_M} \sigma(o, i) = 1. \tag{2.3}$$

Die Suche nach einer Abbildung von Operationen auf Befehle kann als Suche nach einer Überdeckung des Programmgraphen mit Befehlsmustern des Zielprozessors aufgefasst werden. Ein Befehlsmuster P_i überdeckt einen Teilgraphen G_j des Programmgraphen G_p, wenn P_i isomorph[4] zu G_j ist und die jeweils überdeckten Operationen und Werte verträglich sind. Eine Operation o_P des Programms ist genau dann mit einer Operation o_I eines Befehlsmusters verträglich (wir schreiben $o_P \sim o_I$), wenn beide den gleichen Operationstyp besitzen:

$$o_P \sim o_I \Leftrightarrow \omega(o_P) = \omega(o_I) \tag{2.4}$$

Ein Wert v_P des Programms ist genau dann mit einem Wert v_I eines Befehlsmusters verträglich (wir schreiben $v_P \sim v_I$), wenn beide die gleiche Speicherklasse besitzen und der Datentyp des Programmwerts im Datentyp des Befehlsmusterwerts enthalten ist:

$$v_P \sim v_I \Leftrightarrow \theta(v_P) = \theta(v_I) \wedge \delta(v_P) \subseteq \delta(v_I) \tag{2.5}$$

Abbildung 2.5 skizziert, wie ein Teilgraph des Programmgraphen durch ein Befehlsmuster überdeckt wird. Eine Überdeckung des Programmgraphen G_p ist genau dann gültig, wenn jede Operation aus G_p von genau einem Befehlsmuster überdeckt wird.

Aus dem mit Befehlen überdeckten Programmgraphen wird anschließend der zu dieser Befehlsauswahl gehörende Befehlsgraph aufgebaut.

Ein *Befehlsgraph* $G_I = (I_I \cup V_I, D_I)$ ist ein gerichteter azyklischer Graph mit den Knotenmengen I_I und V_I und der Kantenmenge D_I. I_I ist die Menge der Befehle des Befehlsgraphen, V_I die Menge der Werte. $D_I \subset (I_I \times V_I) \cup (V_I \times I_I)$ ist die Menge der direkten Datenabhängigkeiten zwischen Befehlen und Werten. Im Befehlsgraphen werden die Operationen des Programmgraphen durch die ausgewählten Befehle I_I ersetzt. Jeder Befehl vereinigt die von ihm überdeckten Operationen. Die Werte, die zwischen den Befehlen transportiert werden, werden in die Wertemenge V_I übernommen. Werte, die nicht von einem Eingabe- oder Ausgabewert des Befehls überdeckt werden, treten dagegen im Befehlsgraphen nicht auf. Abbildung 2.6 skizziert das Vorgehen.

[4] Der Begriff der Isomorphie von Graphen und weitere Begriffen aus der Graphentheorie werden zum Beispiel in [26] definiert und erläutert.

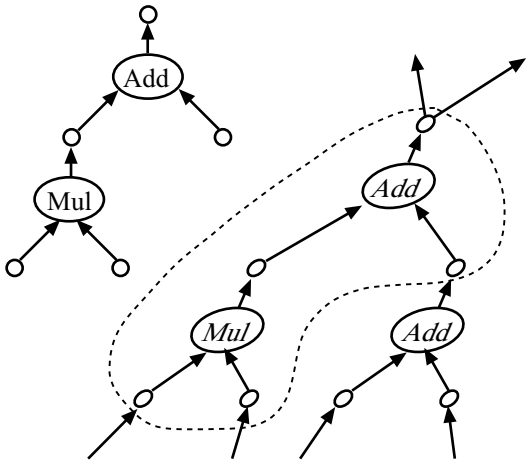


Abbildung 2.5: Überdeckung eines Teilgraphen des Programmgraphen durch ein Befehlsmuster

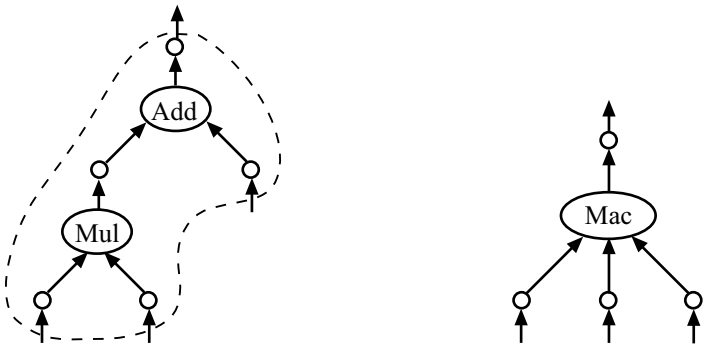


Abbildung 2.6: mit MAC-Befehl (Multiply-Accumulate) überdeckter Programmteilgraph (links) und zugehöriger Befehlsgraph (rechts)

2.3.2 Registerzuordnung

Die Zwischenergebnisse der Operationen werden im allgemeinen in Registern gespeichert. Ob ein Wert in einem Register gespeichert werden soll, wird durch seine Speicherklasse bestimmt. Die Aufgabe der Registerzuordnung besteht darin, den Registerwerten des Befehlsgraphen jeweils ein Register des Zielprozessors zuzuweisen.

Eine *Registerzuordnung* für einen Befehlsgraphen $G_I = (I_I \cup V_I, D_I)$ mit Registerwerten V_I^R und eine Maschinenbeschreibung $M = (n_e, I_M, M_M, R_M)$ ist eine Funktion $\rho : V_I^R \times R_M \to \{0, 1\}$ mit

$$\rho(v, r) = \begin{cases} 1 & \text{wenn dem Wert } v \text{ das Register } r \text{ zugeordnet ist,} \\ 0 & \text{sonst.} \end{cases} \tag{2.6}$$

Damit eine Registerzuordnung *gültig* ist, müssen mehrere Bedingungen erfüllt sein. Jedem Registerwert aus V_I muss genau ein Register zugeordnet sein:

$$\forall v \in V_I^R : \sum_{r \in R_M} \rho(v, r) = 1 \tag{2.7}$$

Die Menge der einem Wert v zuweisbaren Register bezeichnen wir mit $R(v)$. Jedem Registerwert v aus V_I darf nur ein Register aus dieser Menge zugewiesen werden:

$$\rho(v, r) = 1 \Rightarrow r \in R(v) \tag{2.8}$$

Zu einem Zeitpunkt t kann in einem Register natürlich nur ein Wert gespeichert sein. Die Aktivitätsintervalle der Werte, die einem Register zugeordnet sind, dürfen sich also nicht überlappen:

$$\forall t \in \mathbb{N}, \forall r \in R_M : \sum_{v \in A(t)} \rho(v, r) \leq 1 \tag{2.9}$$

Die Registerzuordnung für einen Wert muss mit dem Befehl, der diesen Wert produziert, und mit den Befehlen, die diesen Wert als Operanden konsumieren, verträglich sein. Abbildung 2.7 illustriert das Problem: Befehl i_1 erlaubt für das Ergebnis die Registermenge $R_1 = \{A, B, X, Y\}$, Befehl i_2 für den rechten Operanden die Registermenge $R_2 = \{A, B\}$ und Befehl i_3 für den linken Operanden die Registermenge $R_3 = \{A, X\}$. Somit muss dem Wert v das Register A zugeordnet werden, damit die Registerzuordnung mit den Befehlen verträglich ist.

Wenn für einen Wert keine Registerzuordnung existiert, die mit den angrenzenden Befehlen verträglich ist, kann diesem Wert kein Register zugeordnet werden. Vor der Registerzuordnung muss daher sichergestellt sein, dass dieser Fall nicht eintreten kann. Das geschieht durch Einfügen von entsprechenden Kopierbefehlen in den Befehlsgraphen.

Für die Registervergabe können zusätzliche Bedingungen, zum Beispiel durch das Compiler-Front-End, vorgegeben werden: *Registervorgaben* und *zyklische*

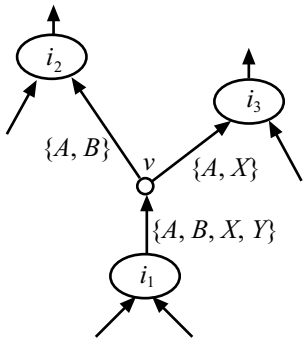


Abbildung 2.7: Registerzuordnung

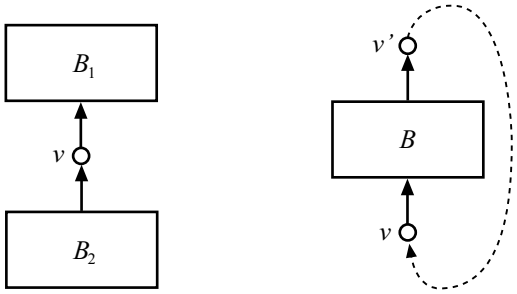


Abbildung 2.8: Registervorgabe (links) und zyklische Abhängigkeit (rechts)

Abhängigkeiten. Registervorgaben werden eingesetzt, wenn Werte über die Grenzen eines Grundblocks hinaus aktiv sind. In diesem Fall ist es sicher sinnvoll, diesen Werten in allen Grundblöcken, in denen diese Werte aktiv sind, die gleichen Register zuzuordnen. Bei der Registerzuordnung müssen diese Vorgaben beachtet werden. Zyklische Abhängigkeiten spielen eine Rolle, wenn der zu übersetzende Grundblock der Rumpf einer Schleife ist. Werte die über den Rand einer Schleife hinaus aktiv sind (zum Beispiel Zählvariablen der Schleife), tauchen nämlich an den Schleifengrenzen doppelt auf, und zwar jeweils am Ende der aktuellen und am Anfang der nächsten Iteration. Diesen doppelten Werten sollte nach Möglichkeit das gleiche Register zugeordnet werden, um unnötige Umspeicherungen zu vermeiden.

Abbildung 2.8 stellt diese Situationen dar. Links ist der Wert v in den beiden Grundblöcken B_1 und B_2 aktiv und sollte deshalb in beiden Blöcken dem gleichen Register zugeordnet werden. Diese Zuordnung erfolgt auf einer äußeren Ebene und wird bei der Übersetzung von B_1 und B_2 als Registervorgabe weitergegeben. In der rechten Abbildung bildet der Block B den Rumpf einer Schleife und der Wert v ist zwischen zwei Iterationen aktiv. Der Wert v' am Ende der Iteration ist gleich dem Wert v zu Beginn der nächsten Iteration. Zwischen v' und v besteht deshalb eine zyklische Abhängigkeit und den beiden Werten sollte

das gleiche Register zugeordnet werden.

2.3.3 Befehlsplanung

Letztendlich muss noch bestimmt werden, in welcher Reihenfolge die ausgewählten Befehle auszuführen sind. Aufgabe der Befehlsplanung ist es, für jeden Befehl des Befehlsgraphen den Zeitschritt zu bestimmen, zu dem seine Ausführung beginnt.

Ein *Befehlsplan* für einen Befehlsgraphen $G_I = (I_I \cup V_I, D_I)$ ist eine Funktion $\gamma : I_I \times \mathbb{N} \to \{0, 1\}$ mit

$$\gamma(i,t) = \begin{cases} 1 & \text{wenn der Befehl } i \text{ für den Zeitschritt } t \text{ geplant ist,} \\ 0 & \text{sonst.} \end{cases} \tag{2.10}$$

Als Länge $l(\gamma)$ eines Befehlsplanes γ bezeichnen wir den Wert

$$l(\gamma) = \max_{\substack{t \in \mathbb{N} \\ i \in I_I \\ \gamma(i,t)=1}} \{t\}. \tag{2.11}$$

Ein Befehlsplan heißt *gültig*, wenn alle Befehle, von denen ein Befehl abhängt, vor diesem Befehl ausgeführt werden. Die Menge der Befehle, von denen ein Befehl direkt abhängt, bezeichnen wir mit $D(i)$:

$$D(i) = \{j | j \in I_I, v \in V_I, (j,v) \in D_I, (v,i) \in D_I\} \tag{2.12}$$

Die Menge der Befehle, von denen ein Befehl direkt oder indirekt abhängt, bezeichnen wir mit $D^*(i)$:

$$D^*(i) = D(i) \cup \bigcup_{j \in D(i)} D^*(j) \tag{2.13}$$

Jeder Befehl darf erst dann zur Ausführung kommen, wenn alle seine Operanden berechnet sind:

$$\begin{gathered} \forall i \in I_I, \forall j \in D^*(i), \forall t_i \in \mathbb{N}, \forall t_j \in \mathbb{N} : \\ \gamma(i,t_i) = 1 \wedge \gamma(j,t_j) = 1 \Rightarrow t_j < t_i \end{gathered} \tag{2.14}$$

Die Befehlsplanung darf keine Ressourcenkonflikte verursachen. So kann eine Ausführungseinheit des Zielprozessors in jedem Zeitschritt nicht mehr als einen Befehl bearbeiten. Außerdem darf, wie oben beschrieben, auf ein Register in jedem Zeitschritt nur ein Schreibzugriff stattfinden. Daneben kann es weitere Beschränkungen geben, die in der Beschreibung des Zielprozessors modelliert werden müssen (zum Beispiel für den Zugriff auf externe Busse). Diese sollen aber hier, da sie nichts Grundsätzliches ändern, nicht weiter betrachtet werden.

Es kann noch ein weiteres Problem auftreten, dass allerdings zur Registerzuordnung und zur Befehlsplanung gleichermaßen gehört[5]: Was passiert, wenn zu

[5] Das ist gleichzeitig ein Hinweis darauf, dass diese Prozesse nicht so unabhängig voneinander zu lösen sind, wie es zunächst scheint.

einem Zeitpunkt mehr Werte aktiv sind, als der Zielprozessor passende Register besitzt? Diese Situation lässt sich nicht immer durch einen anderen Befehlsplan lösen. Es führt dann kein Weg daran vorbei, die Aktivitätsintervalle dieser Werte aufzuteilen. Dazu gibt es zwei Möglichkeiten, zwischen denen abgewogen werden muss: Die erste ist, die Werte zeitweise im Speicher abzulegen und bei Bedarf wiederherzustellen. Man spricht in diesem Fall von einem *Registerabwurf* (engl. register spill). Die zweite Möglichkeit ist, die Werte zu einem späteren Zeitpunkt neu zu berechnen.

2.4 Codegenerierung als Optimierungsproblem

Grundsätzlich kann man Codegenerierung als Optimierungsproblem auffassen. Geeignete Zielfunktionen sind zum Beispiel die Zahl der Taktzyklen, die zur Ausführung eines Programms erforderlich sind, oder die Größe des erzeugten Codes.

Die unterschiedlichen Verfahren, die zur Codegenerierung entwickelt wurden, kann man in *klassische* und *nichtklassische* Verfahren einteilen. Die klassischen Verfahren gehen von Prozessormodellen aus, die Befehle im wesentlichen sequentiell abarbeiten und allgemein verwendbare Register besitzen. Diese Verfahren funktionieren bei Standardprozessoren bis heute gut, versagen aber, wie Studien belegen, bei (explizit) parallelen Architekturen mit komplexen Befehlen oder unregelmäßigen Registern. In [35] wird zum Beispiel bei typischen DSP-Aufgaben für einige DSP-Compiler ein Laufzeit-Overhead von bis zu 800% und ein Overhead bei der Codegröße von bis zu 300% gegenüber handoptimiertem Assemblercode festgestellt. Die nichtklassischen Verfahren arbeiten dagegen mit Prozessormodellen, die expliziten Parallelismus auf der Befehlsebene, komplexe Befehle und unregelmäßige Register einschließen. Sie zielen damit auf Architekturen, wie sie zum Beispiel bei DSPs und Multimediaprozessoren eingesetzt werden.

2.4.1 Klassische Verfahren

Zwei Eigenschaften haben alle klassischen Verfahren zur Codegenerierung gemein. Erstens erfolgt die Codegenerierung stets auf der Basis einer Baumdarstellung des zu übersetzenden Programms bzw. Grundblocks. Wenn das Compiler-Front-End eine Graphendarstellung liefert, so wird diese zunächst in Bäume zerlegt, indem gemeinsame Teilausdrücke aufgespalten werden. Diese Teilbäume werden danach getrennt übersetzt. Die zweite Gemeinsamkeit der klassischen Verfahren ist die weitgehend getrennte Optimierung der Teilprobleme Befehlsauswahl, Registerzuordnung und Befehlsplanung.[6]

[6] Mit der zunehmenden Parallelität der Standardprozessoren wird es aber auch hier immer notwendiger, diese Prozesse zumindest Teilweise zu koppeln. So scheint es zum Beispiel bei superskalaren Prozessoren sinnvoll, Registerzuordnung und Scheduling in einen Prozess zu integrieren. In [20] wird ein solches Verfahren vorgestellt.

Eine der wichtigsten Grundlagen für nahezu alle klassischen Codegeneratoren ist die Befehlsauswahl mittels dynamischer Programmierung[7], wie sie Aho in [1] beschreibt. Mit dieser Methode ist es möglich, für eine breite Klasse von Zielarchitekturen optimalen Code für Ausdrucksbäume zu erzeugen. Die Komplexität dieses Algorithmus verhält sich linear zu der Anzahl der Operationen in dem zu übersetzenden Ausdrucksbaum. Es ist möglich, einen großen Teil der dynamischen Entscheidungen für einen gegebenen Befehlssatz vorauszuberechnen und einen Automaten zu konstruieren, der optimalen Code für einen gegebenen Ausdrucksbaum erzeugt. Codegeneratoren, die mit solchen sogenannten BURS[8]-Automaten arbeiten, sind um ein Vielfaches schneller als Codegeneratoren, die auf dynamischen Verfahren basieren. Eine besonders effiziente Methode zur Erzeugung eines solchen Codegenerators wird in [22] beschrieben.

Die Registerzuordnung wird in klassischen Compilern häufig mit einem Verfahren realisiert, das erstmals in [6] beschrieben und seitdem vielfach erweitert wurde. Bei diesem Verfahren wird aus den Aktivitätsintervallen der Variablen ein Konfliktgraph erzeugt. Die Registerzuordnung wird anschließend durch Färbung dieses Graphen bestimmt, indem Werten mit gleicher Färbung das gleiche Register zugeordnet wird. Zur Lösung des Färbungsproblems sind verschiedene Heuristiken bekannt.

Eine explizite Befehlsplanung erfolgt bei den klassischen Verfahren meist nur bei Compilern für superskalare Architekturen und auch dort oft nur als nachgeordneter Schritt. Dabei werden dann einzelne Befehle anhand von einfachen Gruppierungsregeln lokal umgeordnet, um die Auslastung der Einheiten zu verbessern.

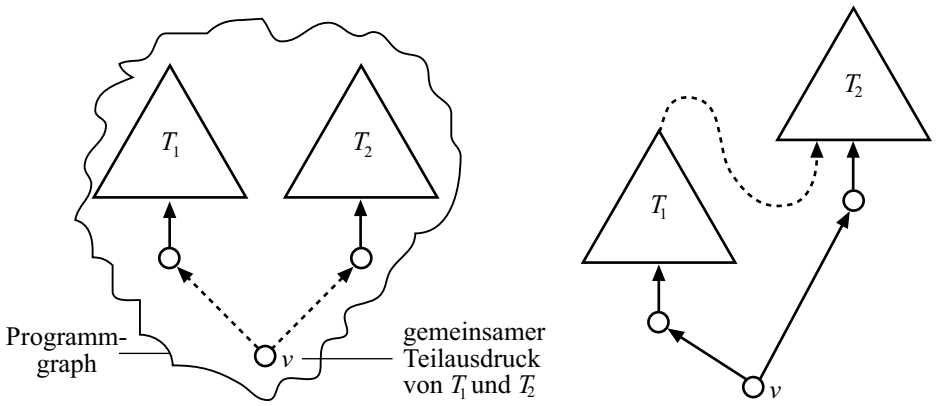
In [3], das eine gute Referenz für den Entwurf und die Implementierung von Compilern darstellt, werden die hier genannten Verfahren zur Codegenerierung (neben vielen weiteren) ausführlicher behandelt.

2.4.2 Nichtklassische Verfahren

Als nichtklassische Verfahren zur Codegenerierung bezeichnen wir Verfahren, die sich in zwei wesentlichen Eigenschaften von den klassischen Verfahren unterscheiden: sie arbeiten direkt auf einer Graphendarstellung des zu übersetzenden Programms und sie optimieren Befehlsauswahl, Registerzuordnung und Befehlsplanung nicht mehr unabhängig voneinander.

Die Verwendung einer Graphendarstellung zur Codegenerierung hat beträchtliche Auswirkungen auf die Komplexität der zu lösenden Optimierungsaufgabe. So wird in [5] bewiesen, dass die Erzeugung von optimalem Code für einen gerichteten azyklischen Graphen (äquivalent zu unserem Programmgraphen) für eine Ein-Register-Maschine $\mathcal{NP}$-vollständig ist. In [2] wird gezeigt, dass dieses Problem selbst auf einer Maschine mit einer unbegrenzten Anzahl von Registern $\mathcal{NP}$-vollständig bleibt.

[7] Eine gute Einführung in die dynamische Programmierung wird zum Beispiel in [25] gegeben.
[8] BURS: Bottom-Up Rewrite System



(a) Zerlegung des Graphen (b) zusätzliche Abhängigkeit

Abbildung 2.9: Zerlegung des Programmgraphen in Bäume

Warum nehmen wir eine solche Erhöhung der Komplexität überhaupt in Kauf? Der Grund dafür ist, dass eine Zerlegung des Programmgraphen in Bäume zusätzliche Datenabhängigkeiten schafft und damit das Parallelisierungs-Potential einschränkt. Abbildung 2.9 stellt diesen Sachverhalt dar. Der Programmgraph besitzt einen gemeinsamen Teilausdruck v. An dieser Stelle erfolgt die Zerlegung des Graphen in die beiden Teilbäume T_1 und T_2. Für diese beiden Teilbäume wird dann unabhängig voneinander lokal optimaler Code generiert. Das hat zur Folge, dass die Befehle von T_1 entweder komplett vor (wie in der Abbildung rechts) oder komplett nach den Befehlen von T_2 ausgeführt werden, obwohl zwischen T_1 und T_2 gar keine Abhängigkeiten bestehen. So lange der Zielprozessor nur wenig Parallelismus zulässt, fällt dieses Problem nicht so stark ins Gewicht, bei hochparallelen Prozessoren führt es aber zu einer inakzeptablen Vernachlässigung des möglichen Parallelismus. Ein Verfahren, das direkt auf dem Programmgraphen arbeitet, ist diesen Parallelitätsbeschränkungen nicht unterworfen.

Die von den nichtklassischen Verfahren anvisierten Prozessorarchitekturen besitzen häufig sehr spezialisierte Einheiten und Register. Die Folge sind starke Abhängigkeiten zwischen Befehlsauswahl, Registerzuordnung und Befehlsplanung. Werden die Teilprobleme unabhängig voneinander gelöst, kann auf diese Abhängigkeiten nicht eingegangen werden, was zu sehr schlechten Lösungen führen kann. Aus diesem Grund wird versucht, die Teilprobleme möglichst stark zu integrieren und gemeinsam zu lösen. Dieses Vorgehen wird als Phasenkopplung bezeichnet.

2.5 Verwandte Arbeiten

Auf dem Gebiet der Compiler für variable Zielarchitekturen (engl. retargetable compiler) gibt es eine Reihe von verschiedenen Ansätzen. Als Beispiel sei hier der Codegenerator *Aviv* [10], [11] aus dem SPAM-Projekt[9] genannt. Dieser verwendet eine besondere Datenstruktur, den Split-Node-DAG, zur Darstellung aller Implementierungsmöglichkeiten eines Programmgraphen. Die Codegenerierungsprobleme werden teilweise gekoppelt und mit Heuristiken gelöst.

Ein anderes Beispiel ist der Compiler *Chess* [21]. Dieser Compiler verwendet eine strukturelle Darstellung des Zielprozessors, den Instruction-Set-Graphen, die mit Befehlsmustern überdeckt wird. Die Befehle werden dann in einem Prozess, der Bundling genannt wird, zu parallelen Befehlsgruppen zusammengefasst. Die Codegenerierungsprozesse werden bei diesem Verfahren nur gering gekoppelt, indem während der Überdeckung Teilbefehlspläne mit einer Heuristik erstellt werden.

Ein ungewöhnliches Beispiel für einen Codegenerator, der auf einer Graphendarstellung des Eingabeprogramms arbeitet, wird in [15] beschrieben. Dieser Codegenerator zielt wie unsere Arbeit auf Prozessoren mit irregulären Datenpfaden und komplexen Befehlen. Das eingesetzte Verfahren basiert auf logischer Programmierung mit Constraints, wozu sich die Autoren des Prolog-Systems ECLiPSe bedienen. In der genannten Publikation wird nur die Befehlsauswahl beschrieben, aber Registerzuordnung und Befehlsplanung wurden inzwischen in den Codegenerator integriert.

Ein interessantes und effizientes Verfahren wird in [31] angegeben. Das Verfahren benutzt sogenannte beschränkte Ausdrucksbäume (Constrained Expression Trees, siehe [9]), für die lokal optimaler Code generiert wird. Ein evolutionärer Algorithmus wird anschließend eingesetzt, um die Befehlsplanung und die Registerzuordnung der Baumtransfervariablen zu optimieren.

Für die integrierte Optimierung von Befehlsauswahl, Registerzuordnung und Befehlsplanung ist [32] ein wichtiges Beispiel. Darin wird das erste Optimierungsmodell beschrieben, dass die Codegenerierungsprozesse vollständig integriert. Dieses Modell wird als ganzzahliges lineares Programm (Integer Linear Program, ILP) formuliert und mit einem Standardverfahren gelöst.[10] Bei diesem Verfahren (und den meisten anderen ILP-basierten Ansätzen) ist garantiert, dass die berechnete Lösung optimal bezüglich der formulierten Zielfunktion und Gültigkeitsbedingungen ist. Der Hauptnachteil der ILP-Verfahren ist jedoch ihr hoher Rechenaufwand, der selbst bei Minimalbeispielen zu sehr langen Laufzeiten führen kann.

[9] SPAM: Synopsys, Princeton, Aachen, MIT

[10] Verfahren zur Lösung von ganzahligen linearen Programmen werden zum Beispiel in [19] behandelt.

3 Grundlagen evolutionärer Algorithmen

Die Basis für unseren Codegenerator bildet ein evolutionärer Algorithmus, der in diesem Kapitel beschrieben wird. Zunächst werden einige wichtige Eigenschaften evolutionärer Algorithmen angegeben und der grundlegende Ablauf der Optimierung beschrieben. Nach der Frage, wie die Bewertung einer Lösung erfolgt, werden die evolutionären Operatoren behandelt, die den Kern eines jeden evolutionären Algorithmus bilden. Abschließend werden noch zwei Modifikationen des Standardalgorithmus vorgestellt. Unser evolutionärer Algorithmus geht in erster Linie auf die genetischen Algorithmen zurück, die meisten der in diesem Kapitel besprochenen Verfahren sind daher [8] entnommen.

3.1 Das Grundprinzip evolutionärer Algorithmen

Evolutionäre Algorithmen können allgemein durch drei Aussagen charakterisiert werden (nach [34]):

1. Es wird eine Menge von Lösungskandidaten betrachtet,
2. diese wird einem Auswahlprozess unterworfen und
3. durch genetische Operatoren wie Rekombination und Mutation verändert.

In Anlehnung an die natürliche Evolution bezeichnet man die Menge der Lösungskandidaten als *Population*, die einzelnen Elemente als *Individuen*. Die Gesamtmenge aller möglichen Individuen bildet den Individuenraum I. Jedes Individuum $i \in I$ enthält die codierten Parameter für eine Lösung des Problems. Diese codierten Parameter bilden das *Genom* $G(i)$ dieses Individuums. Die Anzahl der Gene eines Genoms G bezeichnen wir mit $|G|$. Abbildung 3.1 illustriert diese Begriffe. Evolutionäre Algorithmen arbeiten iterativ, die einzelnen Schritte werden als *Generationen* bezeichnet.

3.1.1 Der Standardalgorithmus

Der grundlegene Ablauf eines evolutionären Algorithmus ist in Abbildung 3.2 dargestellt. Er kann durch folgende Schritte beschrieben werden:

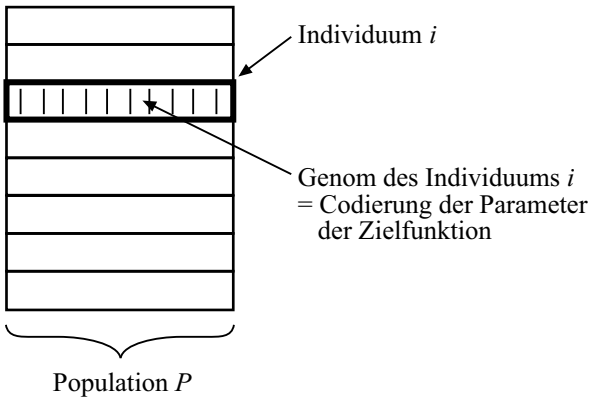


Abbildung 3.1: Population eines evolutionären Algorithmus

- *Initialisiere* die Individuen der Population
- *Bewerte* die Individuen der Population
- Solange die *Abbruchbedingung* noch nicht erfüllt ist:
  - *Selektiere* die Individuen, aus denen die nächste Generation gebildet wird
  - *Rekombiniere* mit einer gewissen Wahrscheinlichkeit p_c jedes der ausgewählten Individuen mit einem anderen
  - *Mutiere* mit einer gewissen Wahrscheinlichkeit p_m jedes der ausgewählten Individuen
  - *Übertrage* die ausgewählten, rekombinierten und mutierten Individuen in die Population für die nächste Generation
  - *Bewerte* die Individuen der neuen Population

Es muss betont werden, dass dieser Algorithmus evolutionäre Algorithmen nicht in ihrer allgemeinsten Form wiedergibt. Die Rekombination kann zum Beispiel auf mehr als zwei Individuen angewendet werden. Außerdem sind viele weitere evolutionäre Operatoren vorgeschlagen worden, auf die hier nicht eingegangen wird. Eine tiefer gehende Diskussion evolutionärer Algorithmen findet sich in den Standardwerken zu diesem Thema ([8], [14]).

3.1.2 Ersetzung der Individuen

Die gewählte Ersetzungsstrategie bestimmt, in welcher Weise die Population der nächsten Generation aus alten und neuen Individuen zusammengesetzt wird. Ein geeignetes Verfahren ist, bis auf die besten n_b Individuen alle Individuen der vorhergehenden Generation zu ersetzen. Die Sicherung der besten Individuen, die oft als Elitismus bezeichnet wird, bewahrt uns vor der Gefahr, die besten

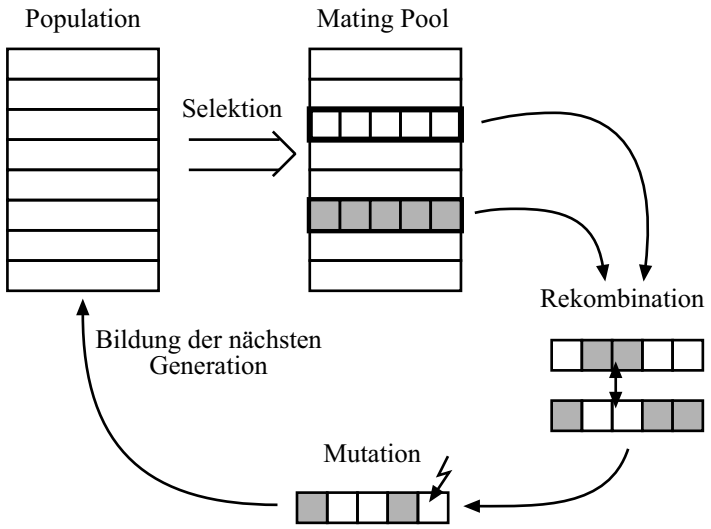


Abbildung 3.2: Evolutionärer Algorithmus

bisher gefunden Lösungen durch Mutation oder Rekombination zu zerstören. Typischerweise wird $n_b = 1$ gewählt.

3.1.3 Abbruchkriterien

Es muss noch beantwortet werden, wann der evolutionäre Algorithmus eigentlich zu beenden ist. Eine Möglichkeit ist, die maximale Anzahl zu berechnender Generationen oder die maximale Berechnungszeit vorzugeben. Je nach Problemstellung kann dabei allerdings der Fall eintreten, dass bis zum Abbruchzeitpunkt noch keine gültige Lösung gefunden wurde. Ist eine Mindestbewertung bekannt, die eine akzeptable Lösung besitzen muss, kann diese ebenfalls als Abbruchkriterium dienen. Der Algorithmus wird dann beendet, wenn die beste bis dahin gefundene Lösung mindestens die vorgegebene Qualität besitzt. Erfolgt über eine bestimmte Zahl von Generationen keine weitere Verbesserung der besten gefundenen Lösung (Stagnation), sollte der Algorithmus ebenfalls abgebrochen werden.

Ein evolutionärer Algorithmus kann im allgemeinen Fall keine Informationen über den Abstand der gefundenen Lösungen zum Optimum liefern. Insbesondere kann er nicht entscheiden, ob ein globales Optimum der Zielfunktion gefunden wurde.

3.2 Die Bewertung einer Lösung

Die Frage, wie eine Lösung des Problems zu bewerten ist, die von einem Individuum der Population eines evolutionären Algorithmus repräsentiert wird, hängt

natürlich stark von dem Problem ab, das gelöst werden soll. Es ist wichtig, zwischen der Zielfunktion des Optimierungsproblems und der Bewertungsfunktion des evolutionären Algorithmus zu unterscheiden[1]:

Die *Zielfunktion* $o : S \to \mathbb{R}$ ordnet jeder Lösung s aus der Menge der möglichen Lösungen S einen reellen Wert $o(s)$ zu, der die Qualität dieser Lösung bezüglich des zu optimierenden Problems angibt. An die Zielfunktion werden keine besonderen Anforderungen (wie Stetigkeit, Differenzierbarkeit etc.) gestellt und über ihre Eigenschaften (Ableitungen etc.) werden keine Informationen benötigt. Die einzige Forderung ist, dass der Zielfunktionswert effizient berechenbar sein muss.

Die *Bewertungsfunktion* $f : I \to \mathbb{R}$ ordnet jedem Individuum i aus der Menge der Individuen I einen reellen Wert $f(i)$ zu, der angibt, wie gut dieses Individuum zur Bildung von Nachkommen für die nächste Generation geeignet ist. Die Bewertungsfunktion sollte mit der Zielfunktion in dem Sinne konsistent sein, dass $o(s_1) > o(s_2) \Rightarrow f(i_1) > f(i_2)$ gilt, wenn s_1 und s_2 die von den Individuen i_1 und i_2 repräsentierten Lösungen sind und beide gültige Lösungen darstellen.

Die Gültigkeit einer Lösung spielt bei der Bewertung ebenfalls eine Rolle. Es kann nicht immer verhindert werden, dass im Verlauf der Optimierung mit evolutionären Algorithmen Lösungen entstehen, die bezüglich des formulierten Problems ungültig sind. Es gibt grundsätzlich zwei Möglichkeiten, mit diesen Lösungen umzugehen. Zunächst kann man versuchen, die betroffenen Individuen zu reparieren. Die Umwandlung einer ungültigen in eine gültige Lösung ist allerdings nicht immer möglich und oft mit sehr großem Aufwand verbunden. Wenn die Individuen nicht repariert werden können, bleibt nur die Möglichkeit, ungültige Lösungen schlechter zu bewerten als gültige. Dazu wird in die Zielfunktion ein Bestrafungsterm miteinbezogen, der davon abhängt, welche der Bedingungen an eine gültige Lösung verletzt wurden. Trotzdem kann damit nicht gesichert werden, dass keine ungültigen Lösungen auftreten. Es kann sogar der Fall eintreten, dass überhaupt keine gültige Lösung gefunden wird, und ein Programm, dass einen solchen evolutionären Algorithmus verwendet, muss darauf reagieren können.

3.3 Die evolutionären Operatoren

Dieser Abschnitt beschreibt die evolutionären Operatoren Selektion, Rekombination und Mutation, die in nahezu jedem evolutionären Algorithmus eine Rolle spielen.

3.3.1 Selektion

Der Selektionsoperator wählt aus der Population diejenigen Individuen aus, die zur Bildung der nächsten Generation von Individuen verwendet werden. Es wer-

[1] In vielen Publikationen wird für die Bewertungsfunktion die Bezeichnung Fitnessfunktion verwendet und in einigen wird die Zielfunktion als Bewertungsfunktion bezeichnet.

den genau so viele Individuen ausgewählt, wie beim Übergang zur nächsten Generation zu ersetzen sind. Dabei können einzelne Individuen mehrfach, andere überhaupt nicht ausgewählt werden. Die ausgewählten Individuen werden in eine Zwischenpopulation, den sogenannten *Mating Pool* kopiert.

Natürlich ist es das Ziel, dass sich die Lösungen, die von den Individuen einer Population repräsentiert werden, über die Generationen hinweg verbessern. Deshalb sollten Individuen mit einer besseren Bewertung eine höhere Wahrscheinlichkeit besitzen ausgewählt zu werden, als Individuen mit einer schlechteren Bewertung. Andererseits darf der Selektionsdruck aber auch nicht so groß sein, dass niemals schlechtere Individuen ausgewählt werden. Sonst bestünde die Gefahr, dass die Lösungen frühzeitig gegen ein lokales Optimum der Zielfunktion konvergieren.[2]

Roulette-Wheel-Selektion

Bei der Roulette-Wheel-Selektion erfolgt die Auswahl der Individuen direkt anhand ihrer relativen Bewertung. Abbildung 3.3 veranschaulicht das Verfahren an einem Beispiel. $f(i)$ ist die Bewertung des Individuums i, $f_r(i, P)$ mit

$$f_r(i, P) = \frac{f(i)}{\sum_{j \in P} f(j)} \tag{3.1}$$

seine relative Bewertung gegenüber der Gesamtpopulation P. Man kann sich das Verfahren nun so vorstellen, dass jedem Individuum ein Kreisausschnitt auf einer Drehscheibe zugeordnet wird. Der Anteil der Fläche des zugeordneten Ausschnitts an der Gesamtfläche der Drehscheibe ist dabei gleich der relativen Bewertung des Individuums. Bei jeder Selektionsoperation wird nun (gedanklich) diese Drehscheibe in Rotation versetzt. Schließlich wird das Individuum ausgewählt, welches zu dem Kreisausschnitt gehört, über dem sich der Punkt S befindet, wenn die Drehscheibe anhält. Die Auswahlwahrscheinlichkeit für ein Individuum i ist also $f_r(i, P)$.

Die Idee hinter diesem Verfahren ist folgende Überlegung: Wenn ein Individuum i_1 „doppelt so gut“ ist wie ein Individuum i_2, dann soll i_1 gegenüber i_2 die doppelte Wahrscheinlichkeit haben, ausgewählt zu werden. i_1 wird dann im Durchschnitt doppelt so viele Nachkommen für die nächste Generation bereit stellen wie i_2.

Tournament-Selektion

Die Tournament-Selektion arbeitet ähnlich wie die Roulette-Wheel-Selektion, erzeugt aber einen höheren Selektionsdruck. Für jede Tournament-Selektion

[2] Die evolutionären Strategien, wie sie Rechenberg in [23] beschreibt, wählen prinzipiell die besten Individuen aus. Sie neigen aus diesem Grund dazu, in lokalen Optima hängenzubleiben. Für multimodale Zielfunktionen sind sie deshalb weniger geeignet. In [12] wird das Verhalten von genetischen Algorithmen und Evolutionsstrategien für eine Reihe von Testfunktionen detailliert untersucht.

Individuum	$f(i)$	$f_r(i, P)$
i_1	8	0.5
i_2	4	0.25
i_3	2	0.125
i_4	2	0.125

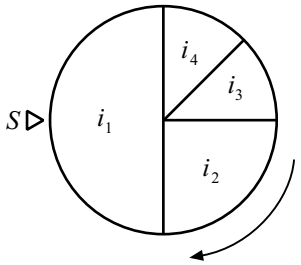
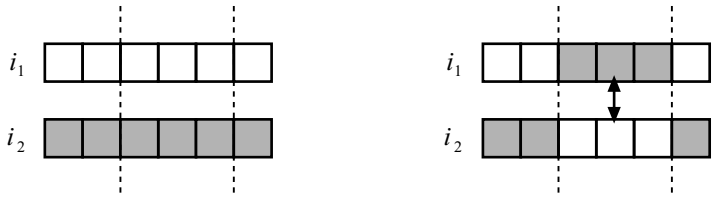


Abbildung 3.3: Roulette-Wheel-Selektion



(a) vor der Rekombination (b) nach der Rekombination

Abbildung 3.4: Rekombination

werden zwei Individuen mittels Roulette-Wheel-Selektion vorselektiert. Das Individuum, das von den beiden die bessere Bewertung besitzt, wird schließlich ausgewählt.

Random-Tournament-Selektion

Wie bei der Tournament-Selektion wird hier das bessere von zwei Individuen gewählt. Allerdings werden die beiden Kandidaten bei der Random-Tournament-Selektion zufällig[3] bestimmt. Dadurch wird ein geringerer Selektionsdruck erzeugt, als bei den beiden anderen vorgestellten Verfahren. Die Random-Tournament-Selektion eignet sich deshalb für Probleme, bei denen die Gefahr besteht, dass die Lösungen zu schnell gegen lokale Optima konvergieren.

3.3.2 Rekombination

Der Rekombinationsoperator tauscht Teile des Genoms zweier Individuen. Das Ziel dieses Operators ist es, gute Teillösungen der beteiligten Individuen zu kombinieren und dadurch neue, bessere Gebiete des Lösungsraumes zu erschließen. Abbildung 3.4 veranschaulicht das Prinzip der Rekombination.

In jeder Generation des evolutionären Algorithmus wird der Rekombinationsoperator mit der Wahrscheinlichkeit p_c auf jedes Individuum i der Zwischen-

[3] Wenn nichts anderes angegeben ist, meinen wir mit zufällig *gleichverteilt* zufällig.

(a) vor der Mutation (b) nach der Mutation

Abbildung 3.5: Mutation

population angewendet. Der Rekombinationspartner j wird zufällig aus allen Individuen der Zwischenpopulation ausgewählt. Es kann sich dabei auch um das selbe Individuum handeln, wobei die Rekombination in diesem Fall keinen Effekt hat. Die zwischen den Individuen i und j zu tauschenden Gene werden zufällig bestimmt. Auch die Anzahl der zu tauschenden Gene wird zufällig bestimmt, und zwar so, dass mit höchster Wahrscheinlichkeit die Hälfte der Gene getauscht wird.

Es liegt auf der Hand, dass die Implementierung eines Rekombinationsoperators stark problemabhängig ist. Wir werden daher erst in Kapitel 4 im Detail auf die von uns verwendeten Rekombinationsoperatoren zu sprechen kommen.

3.3.3 Mutation

Der Mutationsoperator ändert zufällig Teile des Genoms eines Individuums, wie in Abbildung 3.5 gezeigt. Dadurch werden neue Lösungsvarianten in die Population eingebracht. Ein geeigneter Mutationsoperator stellt eine stochastische Absicherung gegen die frühzeitige Konvergenz gegen lokale Optima dar.

In jeder Generation des evolutionären Algorithmus wird der Mutationsoperator mit der Wahrscheinlichkeit p_m auf jedes Individuum i der Zwischenpopulation angewendet. Die zu mutierenden Gene werden zufällig bestimmt. Auch die Anzahl zu mutierender Gene wird zufällig bestimmt, und zwar so, dass kleine Änderungen gegenüber größeren bevorzugt werden.

Da auch die Implementierung eines Mutationsoperators stark vom zu lösenden Problem abhängt, werden wir die von uns verwendeten Mutationsoperatoren ebenfalls erst in Kapitel 4 besprechen.

3.4 Modifikationen des Standardalgorithmus

Die grundlegenden evolutionären Operatoren wurden in den vorhergehenden Abschnitten beschrieben. Ein evolutionärer Algorithmus, der diese Operatoren einsetzt, arbeitet bereits erstaunlich robust. Trotzdem kann es je nach Problemstellung sinnvoll sein, das Verhalten eines evolutionären Algorithmus anzupassen. Dieser Abschnitt beschreibt die von uns verwendeten Modifikationen, viele

weitere werden in [8] angegeben.

3.4.1 Skalierung der Bewertungsfunktion

Die Individuen aus denen die jeweils nächste Generation gebildet wird werden, wie oben beschrieben, nach ihrer relativen Bewertung ausgewählt. Die Verteilung der Bewertungen innerhalb einer Population ändert sich jedoch über die Generationen. Zu Beginn des evolutionären Verfahrens besteht die Population aus mehr oder weniger zufällig initialisierten Individuen und die Bewertungen der Individuen weisen eine große Varianz innerhalb der Population auf. Dies führt dazu, dass Individuen mit einer schlechteren Bewertung sehr schnell aussortiert werden und keine Chance erhalten, Nachkommen zu erzeugen. Die Individuen der späteren Generationen stammen dadurch von sehr wenigen Individuen der Ausgangsgeneration ab. Ein evolutionärer Algorithmus funktioniert aber nur dann gut, wenn die Individuen einer Population bezogen auf ihr Genom möglichst verschieden sind. Es ist deshalb günstig, am Anfang die Unterschiede in den Bewertungen der Individuen zu verringern.

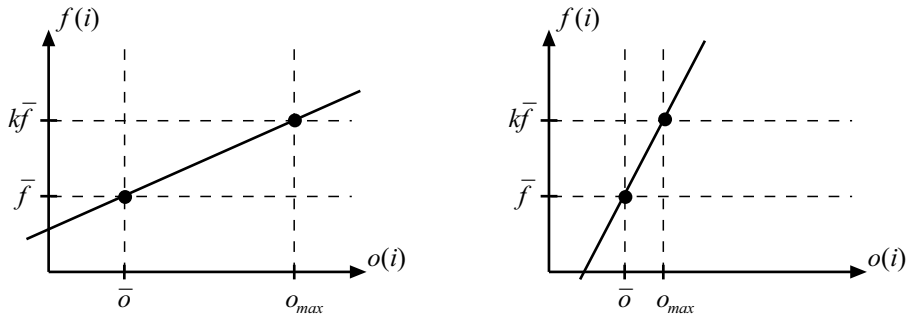
Mit zunehmender Entwicklungsdauer nähern sich die Individuen einer Population in ihren Bewertungen immer mehr an, während die Entwicklungsgeschwindigkeit sinkt. Kleine Verbesserungen der Individuen führen daher zu nur geringfügig besseren relativen Bewertungen. Das heißt aber, dass die Auswahlwahrscheinlichkeit verbesserter Individuen auch nur geringfügig höher ist, als die der anderen Individuen. Es ist deshalb sinvoll, in dieser Situation die Unterschiede in den Bewertungen der Individuen zu verstärken.

Lineare Skalierung

Die lineare Skalierung, die wir bei unserem evolutionären Algorithmus einsetzen, führt genau zu der gewünschten situationsabhängigen Verstärkung oder Abschwächung von Bewertungsunterschieden innerhalb einer Population. Das wird erreicht, indem die Bewertung linear so skaliert wird, dass die mittlere Bewertung innerhalb der Population gleich dem mittleren Zielfunktionswert ist, und die beste Bewertung genau k-mal so gut ist, wie die mittlere Bewertung. k ist die Skalierungskonstante dieses Verfahrens, ein Wert um 2.0 liefert erfahrungsgemäß eine gute Skalierung.

Abbildung 3.6 stellt das Vorgehen für den Fall dar, dass die Zielfunktion maximiert werden soll. $o(i)$ ist der Zielfunktionswert des Individuums i, o_{max} der Zielfunktionswert des besten Individuums (wenn die Zielfunktion minimiert werden soll entsprechend o_{min}) und $\bar{o}$ der mittlere Zielfunktionswert der Population. $f(i)$ ist die Bewertung des Individuums i und $\bar{f}$ die mittlere Bewertung der Population.

Unter der oben getroffenen Annahme $\bar{f} = \bar{o}$ berechnet sich dann $f(i)$ aus der



(a) Abschwächung von Bewertungsunterschieden

(b) Verstärkung von Bewertungsunterschieden

Abbildung 3.6: Lineare Skalierung

Geradengleichung zu

$$f(i) = \frac{(k-1)\bar{o}}{o_{max} - \bar{o}}\Big(o(i) - \bar{o}\Big) + \bar{o} \tag{3.2}$$

wenn die Zielfunktion maximiert werden soll, und zu

$$f(i) = \frac{\left(\frac{1}{k} - 1\right)\bar{o}}{o_{min} - \bar{o}}\Big(o(i) - \bar{o}\Big) + \bar{o} \tag{3.3}$$

wenn die Zielfunktion minimiert werden soll.

3.4.2 Alterung der Individuen

Oft führen Mutation und Rekombination nicht sofort zu einer Verbesserung, häufig sogar zunächst zu einer Verschlechterung eines Individuums. Trotzdem kann es sinnvoll sein, diese Individuen nicht sofort wieder auszusortieren, sondern ihnen einige Generationen Zeit zu geben, sich zu entwickeln. Zu diesem Zweck kann das Alter eines Individuums in die Bewertung einbezogen werden, so dass junge Individuen gegenüber älteren einen Bonus erhalten. Das könnte zum Beispiel bedeuten, dass eine Bestrafung aufgrund verletzter Gültigkeitsbedingungen für junge Individuen reduziert wird.[4] Als Alter eines Individuums wird die Anzahl der Generationen angesehen, die dieses Individuum ohne Veränderung seines Genoms durch Mutation oder Rekombination überdauert hat.

[4] Es ist durchaus angebracht, diese Art der Behandlung jüngerer Individuen mit der Natur zu vergleichen, wo Jungtiere auch nicht sofort dem vollen Konkurrenzkampf ausgesetzt werden.

4 Die Implementierung des Codegenerators

In diesem Kapitel wird die Implementierung unseres Codegenerators beschrieben. Dabei werden die Konzepte angewendet, die in den Kapiteln 2 und 3 vorgestellt wurden. Zunächst wird der Aufbau und die Arbeitsweise des Codegenerators dargestellt. Danach werden die beiden verwendeten evolutionären Optimierungsprozesse zur Befehlsauswahl und zur Registerzuordnung und Befehlsplanung im Detail beschrieben. Dabei geht es neben der Darstellung der Individuen, ihrer Bewertung und ihrer Initialisierung insbesondere um die eingesetzten evolutionären Operatoren.

4.1 Aufbau und Arbeitsweise

Wir haben bereits in Kapitel 2 erläutert, warum es sinnvoll ist, Befehlsauswahl, Registerzuordnung und Befehlsplanung möglichst gemeinsam zu lösen. In unserem Codegenerator sind diese Optimierungsschritte noch nicht vollständig integriert. Stattdessen verwenden wir zwei miteinander gekoppelte Optimierungsprozesse. Der äußere Prozess optimiert die Befehlsauswahl. Eine Lösung der Befehlsauswahl in Form eines Befehlsgraphen wird dann als Eingabe an den inneren Prozess übergeben, der Registerzuordnung und Befehlsplanung für diesen Befehlsgraphen optimiert. Die dabei erreichte Qualität wird gleichzeitig als Qualität des Befehlsgraphen an die Befehlsauswahl zurückgeliefert, die auf dieser Basis neue Lösungen generiert. Das Ergebnis des inneren Optimierungsprozesses wirkt so auf den äußeren Prozess zurück. Die Trennung in diese beiden Optimierungsprozesse ist nicht willkürlich gewählt. Sie ist vielmehr damit zu begründen, dass die Befehlsauswahl die Eingabedaten für die Registerzuordnung und Befehlsplanung modifiziert, was die vollständige Integration dieser Prozesse erschwert.

Beide Optimierungsprozesse, Befehlsauswahl und Registerzuordnung/Befehlsplanung, basieren auf evolutionären Algorithmen, wie sie in Kapitel 3 beschrieben wurden. Abbildung 4.1 stellt den Aufbau unseres Codegenerators und die Kopplung der Optimierungsprozesse dar. Der Programmgraph und die Maschinenbeschreibung sind die Eingabedaten für die Befehlsauswahl. Diese arbeitet als evolutionärer Algorithmus mit einer Population von Individuen, wobei jedes Individuum eine eigene Befehlsauswahl und damit einen Befehlsgraphen

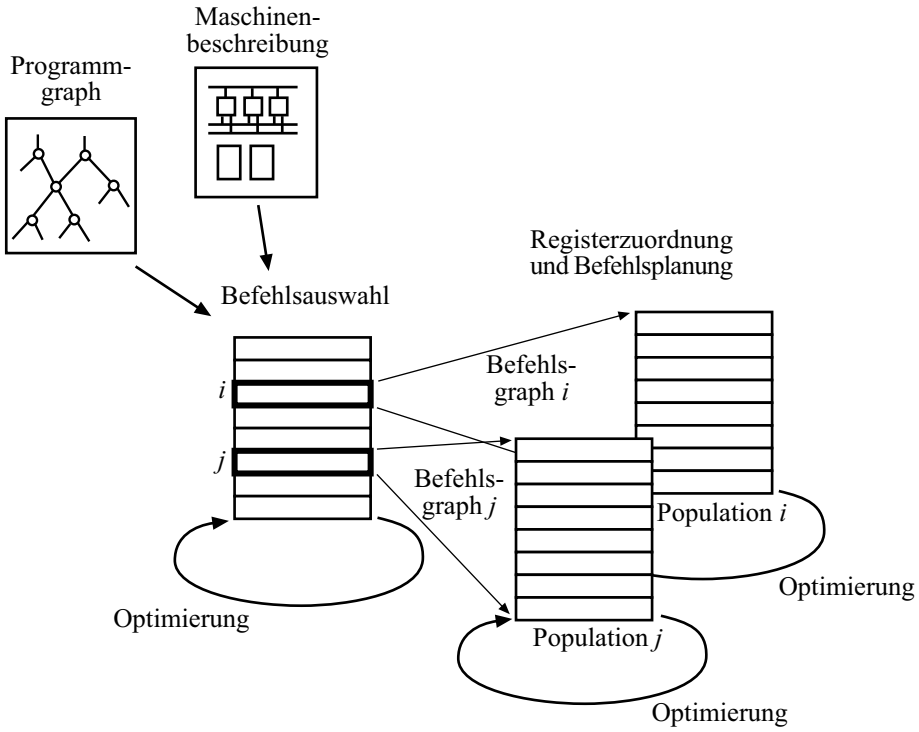


Abbildung 4.1: Aufbau des Codegenerators

repräsentiert. Um die Individuen der Befehlsauswahlpopulation zu bewerten, wird für jedes Individuum ein eigener Optimierungsprozess gestartet, der die Registerzuordnung und die Befehlsplanung für den von diesem Individuum repräsentierten Befehlsgraphen optimiert. Diese Optimierung erfolgt ebenfalls mit einem evolutionären Algorithmus, so dass für jedes Befehlsauswahlindividuum eine eigene Population für die Registerzuordnung und Befehlsplanung erstellt wird. In der Abbildung ist dieses Verfahren für die Individuen i und j dargestellt.

Unser Optimierungsziel ist, die Anzahl der Schritte zu minimieren, die zur Ausführung des Eingabeprogramms erforderlich sind. Das heißt: für einen gegebenen Programmgraphen $G_P = (O_P \cup V_P, D_P)$ und eine gegebene Maschinenbeschreibung $M = (n_e, I_M, M_M, R_M)$,

$$\text{minimiere } l(\gamma) \tag{4.1}$$

so, dass die in Kapitel 2 beschriebenen Gültigkeitsbedingungen für die dabei berechnete Befehlsauswahl σ, die Registerzuordnung ρ und den Befehlsplan γ erfüllt sind.

Die folgenden Abschnitte beschreiben die Implementierung des Codegenerators im Detail. Zur Erläuterung der Implementierung werden wir ein durchgängiges Beispiel benutzen. Es handelt sich dabei um einen Teil der Schleife zur Berechnung eines Skalarprodukts.[1] Abbildung 4.2 zeigt noch einmal den Programmgraphen dieses Beispiels.

4.2 Befehlsauswahl

Wir beginnen mit der Beschreibung der Implementierung des äußeren Optimierungsprozesses, der Befehlsauswahl. Dieser Prozess verwendet einen evolutionären Algorithmus mit Roulette-Wheel-Selektion und linearer Skalierung. Die Eingabedaten für die Befehlsauswahl sind die Maschinenbeschreibung des Zielprozessors und der Programmgraph des Grundblocks, für den Code generiert werden soll.

4.2.1 Darstellung der Individuen

Als Ergebnis der Befehlsauswahl ist ein Befehlsgraph zu erstellen, für den dann Registerzuordnung und Befehlsplanung optimiert werden können. Es vereinfacht allerdings die Implementierung der evolutionären Operatoren, wenn die Darstellung einer Lösung der Befehlsauswahl nicht direkt durch einen Befehlsgraphen erfolgt. Stattdessen verwenden wir eine Tabelle, in der für jede Operation des Programmgraphen der Befehl steht, durch den diese Operation ausgeführt wird. Zusammen mit den Befehlsmustern der Maschinenbeschreibung ergibt das eine

[1] Die Inkrementierung einer Zählvariablen und das Prüfen der Schleifenbedingung wurden weggelassen.

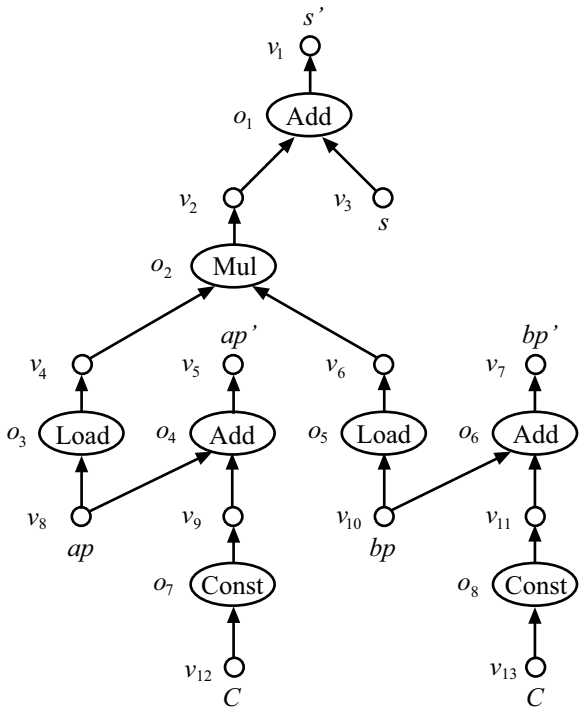


Abbildung 4.2: Programmgraph der Skalarprodukt-Berechnung

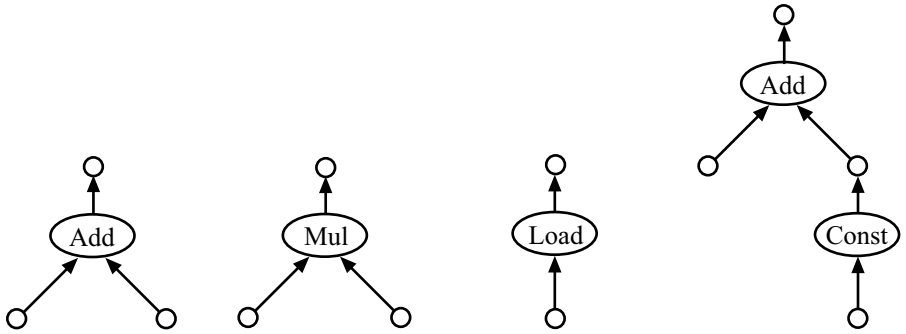
Operation	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8
zugeordneter Befehl	i_1	i_2	i_3	i_4	i_5	i_6	i_4	i_6

Tabelle 4.1: Zuordnung von Befehlen zu Operationen

eindeutige Darstellung der Befehlsauswahl. Tabelle 4.1 zeigt die Darstellung einer Befehlsauswahl. Dabei wurden die Befehlsmuster verwendet, die in Abbildung 4.3 dargestellt sind. i_1 ist ein Add-Befehl, i_2 ein Mul-Befehl, i_3 und i_5 sind Load-Befehle und i_4 und i_6 sind Inc-Befehle.

4.2.2 Bewertung der Individuen

Der Zielfunktionswert für ein Individuum i der Befehlsauswahl ist gleich dem Zielfunktionswert der Registerzuordnung und Befehlsplanung für den Befehlsgraphen, der von i repräsentiert wird. Sowohl bei der Initialisierung der Individuen als auch bei den evolutionären Operatoren wird sichergestellt, dass nur gültige Individuen erzeugt werden. In der Zielfunktion der Befehlsauswahl ist deshalb kein Term zur Bestrafung verletzter Bedingungen erforderlich.



(a) Add (b) Mul (c) Load (d) Inc

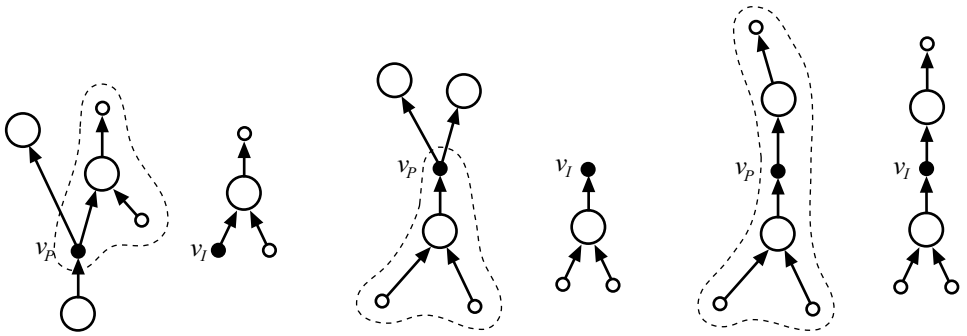
Abbildung 4.3: Befehlsmuster

4.2.3 Initialisierung der Population

Um die Individuen der Population initialisieren zu können, muss zunächst für jede Operation des Programmgraphen bestimmt werden, durch welche Befehle diese Operation überdeckt werden kann. Die Bestimmung der möglichen Überdeckungen erfolgt durch ein einfaches rekursives Verfahren. Für jeden berechneten Wert v des Programmgraphen (das sind die Werte, die keine Eingabewerte sind) und jeden Befehl i wird dabei versucht, den Teilbaum mit der Wurzel v durch das Befehlsmuster von i (beginnend mit dem Ausgabewert von i) zu überdecken. Ist eine solche Überdeckung erfolgreich, wird der Befehl i in die Menge der Befehle aufgenommen, mit denen die Operation überdeckt werden kann, die den Wert v berechnet. Ob ein Wert oder eine Operation des Programmgraphen von einem Wert oder einer Operation eines Befehlsmusters überdeckt werden kann, wird folgendermaßen bestimmt:

- Ein Wert v_P des Programmgraphen kann durch einen Wert v_I eines Befehlsmusters überdeckt werden, wenn v_P und v_I verträglich sind und gilt:

  1. v_I ist ein Eingabewert des Befehls *oder*
  2. v_P und v_I sind keine Eingabewerte *und* v_I ist der Ausgabewert des Befehls *und* die Operation, die v_P berechnet, kann durch die Operation, die v_I berechnet, überdeckt werden, *oder*
  3. v_P und v_I sind keine Eingabewerte *und* v_P ist kein Ausgabewert des Programms *und* die Operation, die v_P berechnet, kann durch die Operation, die v_I berechnet, überdeckt werden. Die Bedingung dass v_P kein Ausgabewert des Programms sein darf verhindert, dass ein Ausgabewert derart überdeckt wird, dass er sich innerhalb eines Befehls befindet und sein Ergebnis nicht nach außen gelangen kann.



(a) Fall 1 (b) Fall 2 (c) Fall 3

Abbildung 4.4: Überdeckung eines Wertes

Die drei Fälle werden in Abbildung 4.4 noch einmal dargestellt. Links befindet sich jeweils ein Ausschnitt des Programmgraphen, rechts das Befehlsmuster. Der vom Befehlsmuster überdeckte Teil des Programmgraphen sowie die Werte v_P und v_I sind gekennzeichnet.

- Eine Operation o_P des Programmgraphen kann durch eine Operation o_I eines Befehlsmusters überdeckt werden, wenn o_P und o_I verträglich sind und alle Operandenwerte von o_P durch die entsprechenden Operandenwerte von o_I überdeckt werden können.

Nachdem für jede Operation die Menge der Befehle bestimmt ist, durch welche sie überdeckt werden kann, werden den Operationen des Befehlsgraphen zufällig Befehle aus dieser Menge zugeordnet. Dabei wird sichergestellt, dass allen Operationen, die von einem ausgewählten Befehl überdeckt werden, dieser Befehl (und kein anderer) zugeordnet wird.

4.2.4 Evolutionäre Operatoren

Bei der Befehlsauswahl werden ein Mutationsoperator und ein Rekombinationsoperator eingesetzt. Die Zahl der zu mutierenden Gene wird so bestimmt, dass mit höchster Wahrscheinlichkeit ein Gen mutiert wird und größere Mutationen bis hin zum gesamten Genom zwar weniger wahrscheinlich, aber trotzdem möglich sind. Mindestens ein Gen wird auf jeden Fall mutiert. Die Zahl der bei der Rekombination zwischen den Individuen zu tauschenden Gene wird so bestimmt, dass mit höchster Wahrscheinlichkeit die Hälfte aller Gene getauscht werden.

Die Gültigkeit einer Befehlsauswahl wird von beiden Operatoren erhalten. Sowohl bei der Mutation als auch bei der Rekombination wird zudem darauf geachtet, dass der Teilgraph des Programmgraphen, dessen Befehlszuordnungen durch

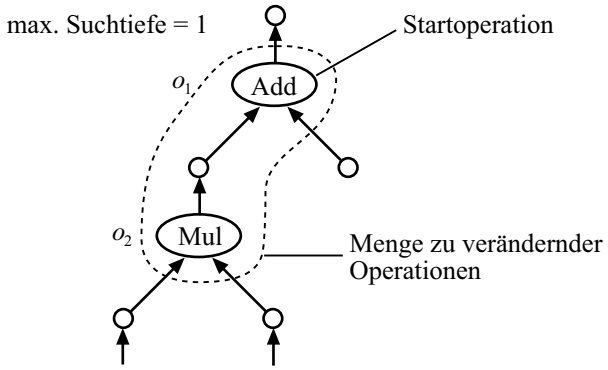


Abbildung 4.5: Bestimmung der Menge zu verändernder Operationen

die Operatoren verändert werden, zusammenhängend ist. Das wird erreicht, indem zunächst eine Startoperation bestimmt wird, an der die Veränderung der Befehlsauswahl beginnt. Aus der Zahl der zu verändernden Gene wird dann eine maximale Suchtiefe im Programmgraphen abgeleitet. Abbildung 4.5 skizziert das Verfahren. Als Startoperation wurde o_1 gewählt, die maximale Suchtiefe ist 1. Die einzige Operation, die mit dieser Suchtiefe erreicht werden kann, ist o_2, also bildet $\{o_1, o_2\}$ die Menge der zu verändernden Operationen.

Mutation der Befehlsauswahl

Bei der Mutation der Befehlsauswahl werden einer Teilmenge der Operationen des Programmgraphen neue Befehle zugeordnet. Da der Teilgraph, der durch diese Teilmenge definiert wird, zusammenhängend ist, kann die Mutation auf die gleiche Weise erfolgen wie die Initialisierung, wobei nur dieser Teilgraph betrachtet wird. Ein Problem tritt allerdings dann auf, wenn die zur Mutation ausgewählten Befehle (das sind die Befehle, die den für die Mutation ausgewählten Operationen zugeordnet sind) zusätzlich Operationen überdecken, die nicht zur Mutation ausgewählt sind. In diesem Fall muss die Menge ausgewählter Operationen um diese Operationen vergrößert werden. Abbildung 4.6 zeigt das Vorgehen. Der abgebildete Teilgraph des Programmgraphen wurde komplett durch einen MAC-Befehl (siehe Abschnitt 2.3.1) überdeckt. Zur Mutation ausgewählt ist aber nur die Additionsoperation o_1. Damit die Gültigkeit der Befehlsauswahl durch die Mutation nicht beeinflusst wird, muss daher zur ausgewählten Menge die Operation o_2 hinzugenommen werden.

Rekombination der Befehlsauswahl

Bei der Rekombination der Befehlsauswahl werden die Befehlszuordnungen für die Operationen eines Teilgraphen des Programmgraphen vom Rekombinationspartner übernommen. Dabei tritt ein ähnliches Problem auf, wie bei der Mutation. Auch hier dürfen keine Befehle durch die Rekombination geteilt wer-

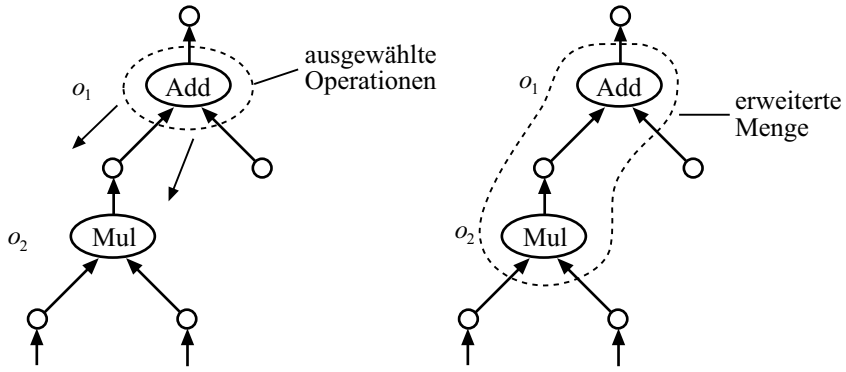


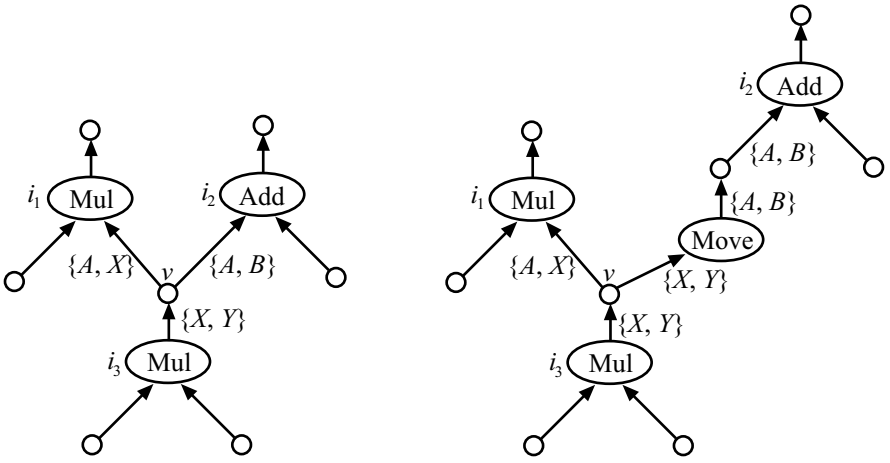
Abbildung 4.6: Erweiterung der Menge zu verändernder Operationen

den. Die Startoperation wird deshalb so gewählt, dass sie in beiden beteiligten Individuen die Wurzeloperation eines Befehls bildet. Zusätzlich wird, genau wie bei der Mutation, die Menge der zur Rekombination ausgewählten Operationen so erweitert, dass alle Operationen, die von den ausgewählten Befehlen überdeckt werden, in dieser Menge enthalten sind. Danach können die zugeordneten Befehle einfach getauscht werden.

4.2.5 Erstellung des Befehlsgraphen

Aus der Befehlsauswahl, die durch die Zuordnung von Befehlen zu Operationen definiert wird, muss schließlich ein Befehlsgraph erstellt werden, der dann an die Registerzuordnung und Befehlsplanung übergeben wird. Der Befehlsgraph ist durch die Befehlszuordnung und die Befehlsmuster der Maschinenbeschreibung eindeutig bestimmt. Der Aufbau des Befehlsgraphen erfolgt rekursiv. Mit dem Aufbau des Befehlsgraphen wird bei den Ausgabewerten des Programmgraphen begonnen. Für jeden Wert wird der Befehl zum Befehlsgraphen hinzugefügt, welcher der Operation zugeordnet ist, die diesen Wert berechnet. Für jeden Befehl werden die Operandenwerte dieses Befehls zum Befehlsgraphen hinzugefügt. Die folgenden Schritte beschreiben den Algorithmus, der den Befehlsgraphen für einen Wert v erstellt.

- Wenn der Wert v noch nicht bearbeitet wurde:
  - Füge v zum Befehlsgraphen hinzu und kennzeichne v als bearbeitet.
  - Wenn v kein Eingabewert ist:
    - Sei o die Operation, die den Wert v berechnet. Bestimme den Befehl i, der o zugeordnet ist und füge i und eine Kante von i nach v zum Befehlsgraphen hinzu.
    - Für jeden Operandenwert v_o der Operation o:
      - Erstelle den Befehlsgraphen für den Wert v_o (Rekursion).



(a) vor dem Einfügen (b) nach dem Einfügen

Abbildung 4.7: Einfügen von Kopierbefehlen

– Füge eine Kante von v_o nach i zum Befehlsgraphen hinzu.

Dieser Algorithmus wird so lange für jeden noch nicht bearbeiteten Wert des Programmgraphen ausgeführt, bis alle Werte bearbeitet wurden.

Einfügen von Kopierbefehlen

Es gibt noch ein Problem, das schon in Abschnitt 2.3.2 angesprochen wurde. Es muss beim Aufbau des Befehlsgraphen verhindert werden, dass später bei der Registerzuordnung einzelnen Registerwerten kein Register zugeordnet werden kann.

Betrachten wir einen Registerwert v. Sei i_P der Befehl, der diesen Wert berechnet und seien $i_{C1} \dots i_{Cn}$ die Befehle, die v als Operanden verwenden. $R^{(P)}(v)$ ist die Menge der Register, die dem Wert v aus Sicht des Befehls i_P zugeordnet werden können. Analog sind $R^{(C1)}(v) \dots R^{(Cn)}(v)$ die Mengen der Register, die dem Wert v aus Sicht der Befehle $i_{C1} \dots i_{Cn}$ zugeordnet werden können. Dann wird zwischen dem Wert v und jedem Befehl i_{Ck} für den $R^{(P)}(v) \cap R^{(Ck)}(v) = \emptyset$ ist, ein geeigneter Kopierbefehl eingefügt. Diese Methode fügt nicht immer die minimal benötigte Anzahl von Kopierbefehlen in den Befehlsplan ein.[2] Trotzdem wurde sie aufgrund der einfachen Implementierung gewählt. Abbildung 4.7 zeigt das Einfügen von Kopierbefehlen in den Befehlsplan.

[2]Das ist im Grunde ein eigenes Optimierungsproblem.

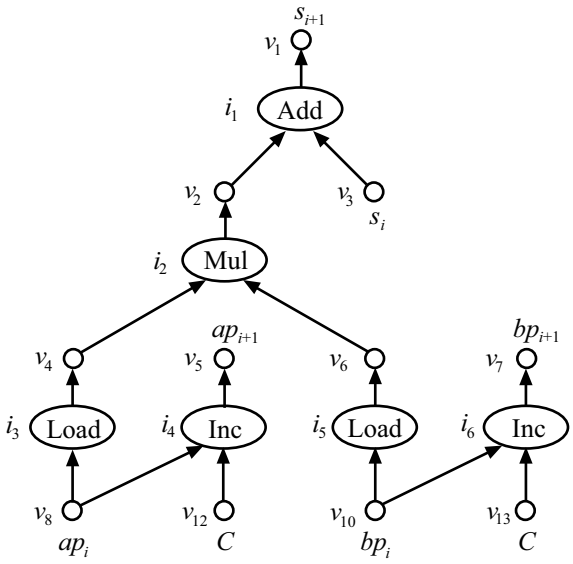


Abbildung 4.8: Befehlsgraph zur Skalarprodukt-Berechnung

4.3 Registerzuordnung und Befehlsplanung

Wir kommen jetzt zur Beschreibung der Implementierung des inneren Optimierungsprozesses, der Registerzuordnung und Befehlsplanung. Dieser Prozess verwendet einen evolutionären Algorithmus mit Tournament-Selektion, linearer Skalierung der Bewertungsfunktion und Alterung der Individuen. Als Eingabe erhält dieser Prozess die Maschinenbeschreibung des Zielprozessors und das Ergebnis der Befehlsauswahl in Form eines Befehlsgraphen. Abbildung 4.8 zeigt einen Befehlsgraphen für die Berechnung eines Skalarprodukts, der uns für die weiteren Abschnitte als Beispiel dienen soll. Er wurde aus dem Programmgraphen auf Seite 34 durch Überdeckung mit den Befehlsmustern auf Seite 35 erzeugt. Zu beachten ist, dass die Werte v_9 und v_{11} im Befehlsgraphen nicht enthalten sind, da sie sich innerhalb der Inc-Befehle i_4 bzw. i_6 befinden.

4.3.1 Darstellung der Individuen

Die Individuen der Registerzuordnung und Befehlsplanung bestehen aus zwei Teilen. Der erste Teil codiert den Befehlsplan. Aus Gründen der Effizienz wird der Befehlsplan eines Individuums in zwei Formen dargestellt, die zueinander konsistent gehalten werden. Neben einer direkten Darstellung des Befehlsplans (Abbildung 4.9), in der für jeden Zeitschritt und jede Ausführungseinheit des Prozessors der auszuführende Befehl eingetragen wird, enthält ein Individuum noch eine Tabelle, die für jeden Befehl den zugeordneten Zeitschritt angibt (Tabelle 4.2). Der zweite Teil eines Individuums codiert die Registerzuordnung. Dieser Teil besteht aus einer Tabelle, die für jeden Registerwert das zugeordne-

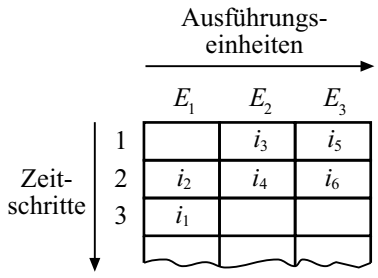


Abbildung 4.9: Befehlsplan

Befehl	i_1	i_2	i_3	i_4	i_5	i_6
Zeitschritt	3	2	1	2	1	1

Tabelle 4.2: Zuordnung von Zeitschritten zu Befehlen

te Register enthält (Tabelle 4.3).

4.3.2 Bewertung der Individuen

Das Ziel der Registerzuordnung und Befehlsplanung ist die Minimierung der Länge des Befehlsplans für eine gegebene Befehlsauswahl. Der Zielfunktionswert eines Individuums i hängt jedoch neben der Befehlsplan-Länge noch davon ab, welche der in 2.3.2 und 2.3.3 genannten Gültigkeitsbedingungen die von i repräsentierte Lösung verletzt.

Registerbeschränkungen

Das Codegenerierungsverfahren stellt sicher, dass jedem Registerwert genau ein Register zugeordnet wird und dass die zugeordneten Register mit den Befehlen verträglich sind, die diesen Wert berechnen oder als Operand verwenden. In die Bewertung muss deshalb nur noch die Beschränkung einbezogen werden, dass ein Register zu einem Zeitpunkt maximal einen Wert aufnehmen kann.

Die Anzahl der Werte, die einem Register r zu einem Zeitpunkt t zugeordnet sind, ist durch

$$n_v(r,t) = \sum_{v \in A(t)} \rho(v,r) \tag{4.2}$$

Registerwert	v_1	v_2	v_3	v_4	v_5
zugeordnetes Register	GR_3	GR_1	GR_3	GR_1	AR_1
Registerwert	v_6	v_7	v_8	v_{10}	
zugeordnetes Register	GR_2	AR_2	AR_1	AR_2	

Tabelle 4.3: Zuordnung von Registern zu Registerwerten

gegeben. Da ein Register zu einem Zeitpunkt maximal einen Wert aufnehmen kann, ist die Zahl der verletzten Registerbeschränkungen $c_R(r,t)$ für das Register r zum Zeitpunkt t

$$c_R(r,t) = \max\{n_v(r,t) - 1, 0\}. \tag{4.3}$$

Die Zahl der insgesamt verletzten Registerbeschränkungen c_R ist somit

$$c_R = \sum_{r \in R_M} \sum_{t \in \mathbb{N}} c_R(r,t). \tag{4.4}$$

Befehlsplan-Beschränkungen

Die Initialisierung der Individuen und die evolutionären Operatoren stellen sicher, dass jedem Befehl ein Zeitschritt zugewiesen wird und dass keine Ausführungseinheit in einem Zeitschritt mehrfach belegt ist. Nicht sichergestellt ist jedoch, dass jeder Befehl nach allen Befehlen ausgeführt wird, von denen dieser Befehl abhängt. Diese Bedingung muss also in die Bewertung eines Individuums miteinfließen. Die Beschränkung $c_S(i,j)$ des Befehls i bezüglich des Befehls j ist

$$c_S(i,j) = \begin{cases} 1 & \text{wenn } j \in D^*(i), \gamma(i,t_i) = 1, \gamma(j,t_j) = 1 \text{ und } t_i \leq t_j, \\ 0 & \text{sonst.} \end{cases} \tag{4.5}$$

Die Zahl der insgesamt verletzten Befehlsplanbeschränkungen c_S ist dann

$$c_S = \sum_{i \in I_I} \sum_{j \in D^*(i)} c_S(i,j). \tag{4.6}$$

Berechnung der Zielfunktion

Die verletzten Register- und Befehlsplanbeschränkungen gehen als Bestrafungsterm quadratisch in die Zielfunktion ein. Die Bestrafung wird allerdings für neu entstandene Individuen um die Hälfte reduziert (siehe Alterung der Individuen, Abschnitt 3.4.2). Wenn $a(i)$ das Alter des Individuums i angibt, das heißt die Anzahl der Generationen, seit denen das Genom von i nicht durch Mutation oder Rekombination verändert wurde, ist die Bestrafung infolge verletzter Gültigkeitsbeschränkungen c_{RS} gegeben durch

$$c_{RS} = \frac{2(c_R + c_S)}{\max\{1, 2 - a(i)\}}. \tag{4.7}$$

Die Zielfunktion berechnet sich dann zu

$$o(i) = l(\gamma)(1 + c_{RS}^2). \tag{4.8}$$

4.3.3 Initialisierung der Population

Die Initialisierung der Registerzuordnung verläuft sehr einfach. Für jeden Registerwert des Befehlsgraphen wird zufällig ein Register aus der Menge der diesem Wert zuweisbaren Register gewählt. Enthält der Befehlsplan Registervorgaben oder zyklische Abhängigkeiten, so werden diese bei der Initialisierung der Registerzuordnung beachtet. Für die Initialisierung des Befehlsplans wurden zwei Varianten implementiert. Die erste initialisiert den Befehlsplan ebenfalls zufällig. Bei der zweiten Variante wird der Befehlsplan zunächst topologisch sortiert. Den Befehlen werden dann in der Reihenfolge ihrer Sortierung Zeitschritte zugewiesen. Dadurch entsteht ein Ablaufplan, der keine Befehlsabhängigkeiten verletzt. Allerdings sind die Individuen der Ausgangspopulation gegenüber einer zufälligen Initialisierung weniger vielfältig. Beide Initialisierungsvarianten stellen sicher, dass jeder Befehl für einen Zeitschritt geplant wird und dass keiner Ausführungseinheit in einem Zeitschritt mehr als ein Befehl zugewiesen wird.

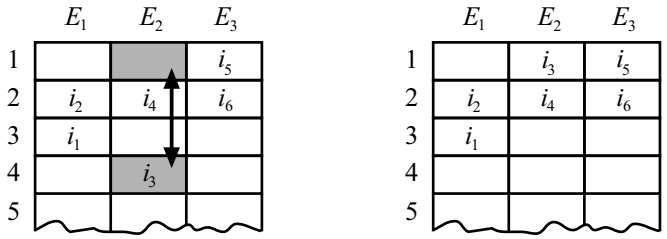
4.3.4 Evolutionäre Operatoren

Bei der Registerzuordnung und Befehlsplanung werden vier verschiedene Mutationsoperatoren und einer von drei Rekombinationsoperatoren gleichzeitig eingesetzt. Die verschiedenen Mutationsoperatoren besitzen jeweils eine eigene Mutationswahrscheinlichkeit. Die Zahl der zu mutierenden Gene wird wie bei der Befehlsauswahl so bestimmt, dass mindestens und mit höchster Wahrscheinlichkeit ein Gen mutiert wird. Auch hier sind größere Mutationen bis zum gesamten Genom möglich, aber weniger wahrscheinlich. Bei der Rekombination wird die Zahl der Gene, die zwischen den Individuen getauscht werden, wieder so bestimmt, dass mit höchster Wahrscheinlichkeit die Hälfte aller Gene getauscht werden.

In Abschnitt 4.3.2 wurde beschrieben, wie die von einer Lösung verletzten Gültigkeitsbedingungen in die Bewertung dieser Lösung (und des betreffenden Individuums) einfließen. Die verwendeten evolutionären Operatoren sind so konstruiert, dass sie folgende Gültigkeitsbedingungen *nicht* verletzen, wenn sie nicht bereits vor der Anwendung des Operators verletzt waren:

- Jedem Befehl ist stets ein Zeitschritt zugeordnet. Keiner Ausführungseinheit ist zu einem Zeitpunkt mehr als ein Befehl zugeordnet.
- Jedem Registerwert ist stets ein Register zugeordnet, dass diesem Wert zuweisbar ist. Die Menge der zuweisbaren Register enthält für alle Registerwerte mindestens ein Register.
- Registervorgaben und zyklische Abhängigkeiten werden beachtet.

Alle anderen Gültigkeitsbedingungen werden von den evolutionären Operatoren nicht beachtet. Die Anwendung eines Operators kann also durchaus Bedingungen verletzten, die vor seiner Anwendung erfüllt waren.



(a) vor der Mutation (b) nach der Mutation

Abbildung 4.10: Mutation des Befehlsplans

Mutation der Registerzuordnung

Bei der Mutation der Registerzuordnung wird zufällig der zu mutierende Wert aus der Menge der Registerwerte bestimmt. Diesem Wert wird dann zufällig ein neues Register zugeordnet. Das neue Register wird dabei aus der Menge der diesem Wert zuweisbaren Register gewählt.

Mutation des Befehlsplans

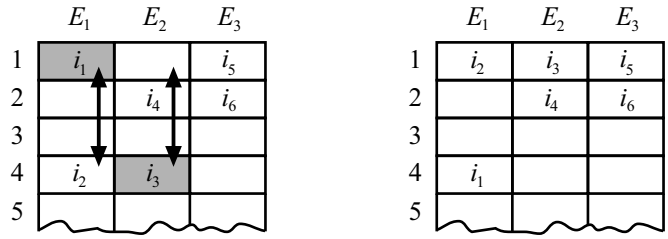
Bei der Mutation des Befehlsplans wird zufällig ein Befehl des Befehlsplans gewählt und der zugeordnete Ausführungsschritt bestimmt. Da ein Befehl fest an eine Ausführungseinheit gebunden ist, darf der ausgewählte Befehl nur innerhalb dieser Einheit versetzt werden. Dazu wird zufällig ein Zeitschritt bestimmt und der Eintrag des Befehlsplans an dieser Stelle (der leer sein kann) mit dem ausgewählten Befehl vertauscht. Bei der Mutation des Befehlsplans wird keine Rücksicht auf die Abhängigkeiten zwischen den Befehlen genommen.

Abbildung 4.10 zeigt das Verfahren an einem Beispiel. Der ausgewählte Befehl ist i_3 auf der Ausführungseinheit E_2. i_3 ist für den Zeitschritt 4 geplant und wird mit dem Eintrag im Zeitschritt 1 getauscht.

Gerichteter Befehlstausch

Eine völlig zufällige Mutation des Befehlsplans reicht möglicherweise nicht immer aus, um verletzte Gültigkeitsbedingungen zu beseitigen. Aus diesem Grund verwenden wir mit dem gerichteten Befehlstausch einen Mutationsoperator, der Informationen über die Befehlsabhängigkeiten zur Steuerung verwendet.

Beim gerichteten Befehlstausch werden zufällig zwei Befehle i und j bestimmt. Diese können der gleichen Ausführungseinheit oder verschiedenen Einheiten zugeordnet sein. Der Befehlstausch wird nur ausgeführt, wenn einer der Befehle direkt oder indirekt von dem anderen abhängt, wenn also $i \in D^*(j)$ oder $j \in D^*(i)$ gilt, und die Abhängigkeit durch die aktuelle Befehlsplanung verletzt wird. Liegen die Befehle auf der gleichen Ausführungseinheit, so werden



(a) vor dem Befehlstausch (b) nach dem Befehlstausch

Abbildung 4.11: Gerichteter Befehlstausch

sie im Befehlsplan einfach vertauscht. Sind i und j verschiedenen Einheiten zugeordnet, werden die beiden Einträge im Befehlsplan bestimmt, die sich an den gleichen Zeitschritten wie i und j befinden, aber auf der jeweils anderen Ausführungseinheit liegen. Die Befehle i und j werden dann mit diesen Einträgen auf ihrer Einheit vertauscht. Durch den Tausch der Befehle werden i und j in die richtige Reihenfolge zueinander gebracht. Allerdings können durch den Tausch Abhängigkeiten zwischen anderen Befehlen verletzt werden.

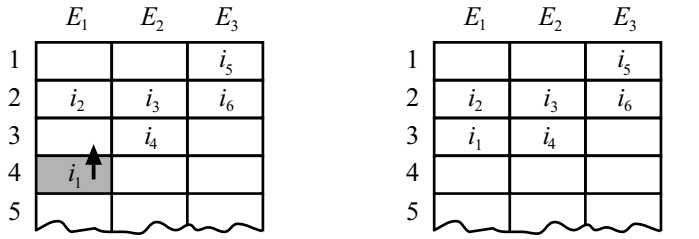
Abbildung 4.11 zeigt, wie der Befehlstausch abläuft. Die ausgewählten Befehle sind i_1 und i_3, die für die Zeitschritte 1 bzw. 4 geplant sind. Da i_1 und i_2 verschiedenen Einheiten zugeordnet sind, werden sie mit den Einträgen der Zeitschritte 4 bzw. 1 des Befehlsplans getauscht, die jeweils auf ihrer Einheit liegen.

Kompaktierung des Befehlsplans

Das Ziel des Optimierungsverfahrens ist es, die zur Ausführung des Befehlsgraphen benötigten Zeitschritte, die wir als Länge des Befehlsplans definiert haben, zu minimieren. Es scheint deshalb sinnvoll, einen Mutationsoperator einzusetzen, der einen Befehlsplan in dieser Richtung beeinflusst.

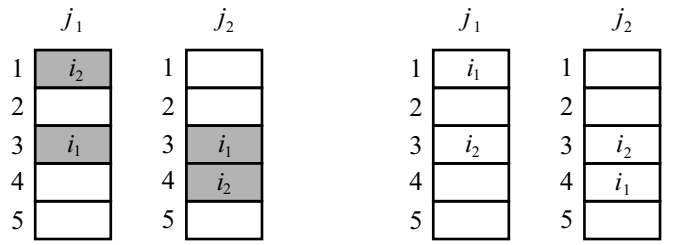
Bei der Kompaktierung des Befehlsplans wird zufällig ein Befehl i ausgewählt. Befindet sich im Befehlsplan auf der Ausführungseinheit, die diesem Befehl zugeordnet ist, im Zeitschritt direkt vor dem, für den i geplant ist, kein Eintrag, so wird der Befehl i um einen Schritt früher geplant. Ist der Eintrag vor dem Befehl i nicht leer, wird nichts verändert.

Abbildung 4.12 zeigt, wie die Kompaktierung arbeitet. Der ausgewählte Befehl ist i_1, der für den Zeitschritt 4 geplant ist. Da sich im Befehlsplan auf der Einheit E_1 im Zeitschritt 3 kein Eintrag befindet, wird der Befehl i_1 in den Zeitschritt 3 verschoben. Die Länge des Befehlsplans wird dadurch um 1 verkürzt.



(a) vor der Kompaktierung (b) nach der Kompaktierung

Abbildung 4.12: Kompaktierung des Befehlsplans



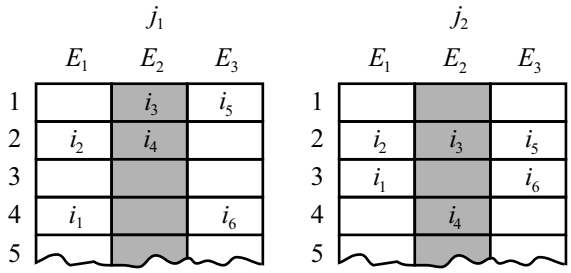
(a) vor der Rekombination (b) nach der Rekombination

Abbildung 4.13: Rekombination der Befehlsreihenfolge

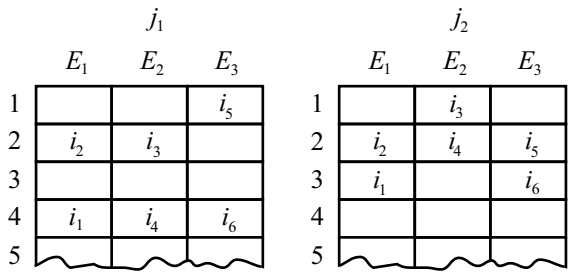
Rekombination der Befehlsreihenfolge

Dieses Rekombinationsverfahren tauscht die Reihenfolge einer ausgewählten Menge von Befehlen zwischen den beteiligten Individuen aus. Der Austausch wird für jede Ausführungseinheit einzeln durchgeführt. Zunächst werden die Zeitschritte der zu rekombinierenden Befehlsmenge festgehalten. Danach werden in jedem der beteiligten Individuen die ausgewählten Befehle innerhalb der festgehaltenen Zeitschritte in die Reihenfolge gebracht, die sie im jeweils anderen Individuum aufweisen. Es wird bei diesem Verfahren nur die relative Ordnung der ausgewählten Befehle getauscht, die Menge der zugeordneten Zeitschritte wird nicht verändert.

Abbildung 4.13 zeigt die Rekombination der Befehlsreihenfolge der Individuen j_1 und j_2 (wobei nur eine Ausführungseinheit der Befehlspläne dargestellt ist). Die ausgewählten Befehle, deren Reihenfolge getauscht wird, sind i_1 und i_2. Wären weitere Befehle auf der dargestellten Ausführungseinheit vorhanden, würden sie von der Rekombination nicht beeinflusst.



(a) vor der Rekombination



(b) nach der Rekombination

Abbildung 4.14: Rekombination der Ausführungseinheiten

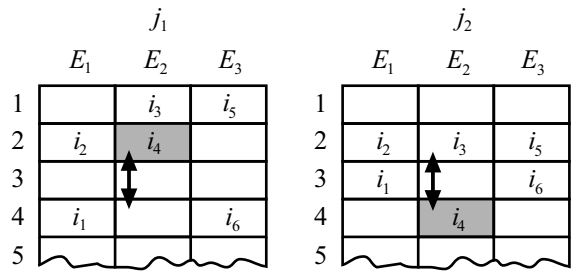
Rekombination der Ausführungseinheiten

Bei der Rekombination der Ausführungseinheiten werden die Zeitschritte aller Befehle, die einer Ausführungseinheit zugeordnet sind, zwischen den Individuen getauscht. Die zu tauschenden Einheiten werden zufällig bestimmt.

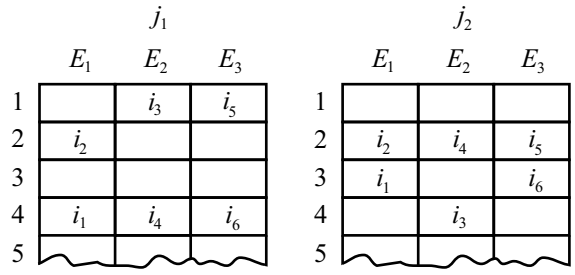
Abbildung 4.14 stellt dieses Rekombinationsverfahren dar. Zwischen den Individuen j_1 und j_2 werden die Zeitschritte aller Befehle, die der Einheit E_2 zugeordnet sind, getauscht.

Rekombination der Befehlspositionen

Bei der Rekombination der Befehlspositionen werden die Zeitschritte einer ausgewählten Menge von Befehlen zwischen den Individuen getauscht. Der Tausch wird für jeden ausgewählten Befehl einzeln vorgenommen. Dazu werden die Zeitschritte bestimmt, für die der zu tauschende Befehl in den beiden Individuen geplant ist. Der Befehl wird dann mit dem Eintrag des Befehlsplans getauscht, der sich an dem Zeitschritt befindet, für den der ausgewählte Befehl in dem jeweils anderen Individuum geplant ist. Durch dieses Verfahren können neben den ausgewählten Befehlen also weitere Befehle versetzt werden. Zusätzlich zu



(a) vor der Rekombination



(b) nach der Rekombination

Abbildung 4.15: Rekombination der Befehlspositionen

den Positionen der ausgewählten Befehle werden zwischen den Individuen noch die Register getauscht, die den Ergebniswerten dieser Befehle zugeordnet sind.

Abbildung 4.15 zeigt dieses Verfahren. Der ausgewählte Befehl ist i_4, der in den Individuen für die Zeitschritte 2 bzw. 4 geplant ist. Durch den Positionstausch wird der Befehl i_3, der im Individuum j_2 vor der Rekombination für den Zeitschritt 2 geplant ist, nach 4 versetzt.

4.4 Beschränkungen des Codegenerators

Der Codegenerator ist so, wie er hier beschrieben wurde, noch einigen Beschränkungen unterworfen. Dieser Abschnitt beschreibt diese Beschränkungen und gibt Hinweise, wie sie aufgehoben werden könnten.

- Die Situation, dass prinzipiell zu wenig Register vorhanden sind, um alle aktiven Werte aufzunehmen, kann zur Zeit von diesem Codegenerator nicht aufgelöst werden. Das Problem wurde bereits in Abschnitt 2.3.3 beschrieben. Der Codegenerator müsste in so einem Fall Registerabwürfe oder Neuberechnungen in den Befehlsgraphen einfügen und so die Akti-

vitätsintervalle einiger Werte trennen. Eine Möglichkeit dies zu erreichen wäre, den Mutationsoperator der Befehlsauswahl zufällig Register-Speicher-Register-Transfers in den Befehlsgraphen einfügen zu lassen. Dann ergäbe sich allerdings das Problem, das die verschiedenen Individuen der Befehlsauswahl verschiedene Programmgraphen implementieren würden. Eine Lösung dieses Problems kann wohl erst dadurch erreicht werden, dass man die Transferoperationen getrennt von den „normalen“ Operationen betrachtet.

- Es findet keine Optimierung über der Ebene von Grundblöcken statt. In verschiedenen Publikationen wurde jedoch gezeigt, dass die Optimierung über Grundblockgrenzen hinaus Vorteile bringen kann, zum Beispiel bei der Optimierung verschachelter Schleifen [7] und der Optimierung des Kontrollflusses. Bei DSPs, die parallel auf mehrere Speicherblöcke zugreifen können, kommt außerdem der Optimierung der Speicherzuweisung an Variablen und der Auswahl geeigneter Addressierungsarten eine besondere Bedeutung zu [4], [27], [30].

- Das Prozessormodell, das der Codegenerator verwendet, ist noch nicht ausdrucksstark genug, um aktuelle Prozessoren detailliert zu beschreiben. So gibt es keine Multizyklen-Befehle und Pipeline-Effekte werden ignoriert. Diese beiden Einschränkungen ließen sich dadurch beseitigen, dass für die Registerzuordnung und Befehlsplanung zusätzliche Gültigkeitsbedingungen formuliert werden. Außerdem wäre eine Anpassung der evolutionären Operatoren erforderlich. Schwerer wiegt, dass es zur Zeit keine Möglichkeit gibt, komplexere Beschränkungen der Zielarchitektur zu formulieren. Das betrifft zum Beispiel die parallele Ausführbarkeit von Befehlen. Weil Adressierungsarten nicht explizit betrachtet werden, sondern sich aus den Befehlsmustern ergeben (die Bäume sein müssen), können komplexere Adressierungsarten nicht spezifiziert werden. Dazu gehören zum Beispiel die bei DSPs häufig vorhandenen Adressierungsarten Ring-Adressierung und Bit-Inverse Adressierung.

5 Experimentelle Untersuchungen

Um zu untersuchen, in wieweit der entwickelte Codegenerator die in Kapitel 1 gesteckten Ziele erreicht, ist eine Reihe von Experimenten erforderlich. In diesem Kapitel werden zunächst die für die Experimente verwendeten Prozessorarchitekturen und Beispielprogramme beschrieben. Danach werden einige der durchgeführten Experimente erläutert und die Ergebnisse der Untersuchungen dargestellt. Am Ende werden die ermittelten Ergebnisse ausgewertet und daraus entsprechende Schlussfolgerungen gezogen.

5.1 Prozessorarchitekturen

Unser Codegenerator soll sich automatisch an die Eigenschaften einer gegebenen Zielarchitektur anpassen. Es ist deshalb sinnvoll, ihn mit Prozessormodellen zu untersuchen, die verschiedene Kategorien von Prozessoren wiederspiegeln. Es wäre allerdings im Rahmen dieser Arbeit zu aufwendig, reale Prozessoren vollständig nachzubilden. Außerdem ist unser Prozessormodell nicht stark genug, um alle Merkmale moderner Prozessoren zu beschreiben. Wir werden deshalb drei Prozessoren beispielhaft modellieren, die sich an realen Vorbildern orientieren, ohne jedoch alle Details nachzubilden. Insbesondere werden wir nur die Eigenschaften spezifizieren, die wir für unsere Untersuchungen benötigen.

Prozessoren besitzen verschiedene Möglichkeiten, die Argumente eines Befehls zu adressieren. Tabelle 5.1 zeigt die von uns verwendeten Adressierungsarten.

Adressierung	Symbolik	Bedeutung
Register	Rx	Der Inhalt des Registers Rx bildet das Argument des Befehls.
Direkt	x	Der Wert x ist das Argument des Befehls.
Indirekt	*Rx	Der Wert an der Adresse, die im Register Rx enthalten ist, ist das Argument des Befehls.
Indirekt mit Postinkrement	*Rx++	Zusätzlich zur normalen indirekten Adressierung wird der Inhalt des Registers Rx nach der Adressierung auf die Folgeadresse des adressierten Wertes gesetzt.

Tabelle 5.1: Adressierungsarten

5.1.1 P1: C3x-ähnliche Architektur

Das erste Prozessormodell repräsentiert digitale Signalprozessoren mit zwei parallelen Recheneinheiten, gemischtem Register/Speicher-Zugriff (dass heisst, Rechenbefehle können Registeroperanden und Speicheroperanden verwenden) und allgemein verwendbaren Registern. Das Modell orientiert sich an der C3x-Serie von Texas Instruments. Abbildung 5.1 zeigt die Prozessorarchitektur. Der Prozessor besitzt einen Multiplizierer und eine arithmetisch-logische Einheit (ALU), die parallel arbeiten können, sowie zwei Adresseinheiten (ARAU0 und ARAU1). Für Operanden und Ergebnisse von Multiplizierer und ALU stehen acht allgemeine Register GR0–GR7 zur Verfügung. Die Addresseinheiten verwenden acht Adressregister AR0–AR7 zur Adressierung von Speicheroperanden. Der Prozessor kann eine Multiplikation und eine Addition oder zwei Datentransfers oder eine Multiplikation bzw. Addition und einen Datentransfer parallel ausführen. Zusätzlich können die Adresseinheiten per Postinkrement-Adressierung zwei Adresswerte aktualisieren. Abschnitt A.1 enthält eine Übersicht der Befehle, die für diesen Prozessor modelliert wurden.

5.1.2 P2: DSP56000-ähnliche Architektur

Das zweite Prozessormodell ist ein Beispiel für einen digitalen Signalprozessor mit Load/Store-Architektur und beschränkt verwendbaren Registern. Dieses Modell orientiert sich an der DSP56000-Serie von Motorola. Abbildung 5.2 zeigt die Prozessorarchitektur. Im Gegensatz zu P1 besitzt dieser Prozessor nur einen Ausführungspfad für allgemeine Berechnungen. Dieser kann dafür einen MAC-Befehl (Multiply-Accumulate) in einem Schritt ausführen. Als Eingaberegister stehen X0–X3 und Y0–Y3 zur Verfügung, als Akkumulatorregister A0–A3. Die Quelloperanden aller Rechenoperationen müssen Eingaberegister oder Akkumulatoren sein, der Zieloperand muss ein Akkumulator sein. ALU-Befehle können bei diesem Prozessor nur Registeroperanden adressieren, für das Laden und Speichern von Registern sind die beiden Adresseinheiten (X-AGU und Y-AGU) zuständig. Derartige Architekturen werden als Load/Store-Architekturen bezeichnet. Die beiden Adresseinheiten greifen auf getrennte Speicherbänke zu. Als Adressregister dienen der ersten Adresseinheit die Register R0–R3, der zweiten Adresseinheit die Register R4–R7. Jede der beiden Adresseinheiten dieses Prozessors kann parallel zur ALU einen Registerwert in ein anderes Register kopieren, einen Wert aus dem Speicher laden oder in den Speicher schreiben. Man bezeichnet diese Architekturen deshalb als Dual-Load-Execute-Architekturen. Parallel zum Lade- bzw. Speichervorgang können die entsprechenden Adresswerte mittels Postinkrement-Adressierung aktualisiert werden. Abschnitt A.2 enthält eine Übersicht der Befehle, die für diesen Prozessor modelliert wurden.

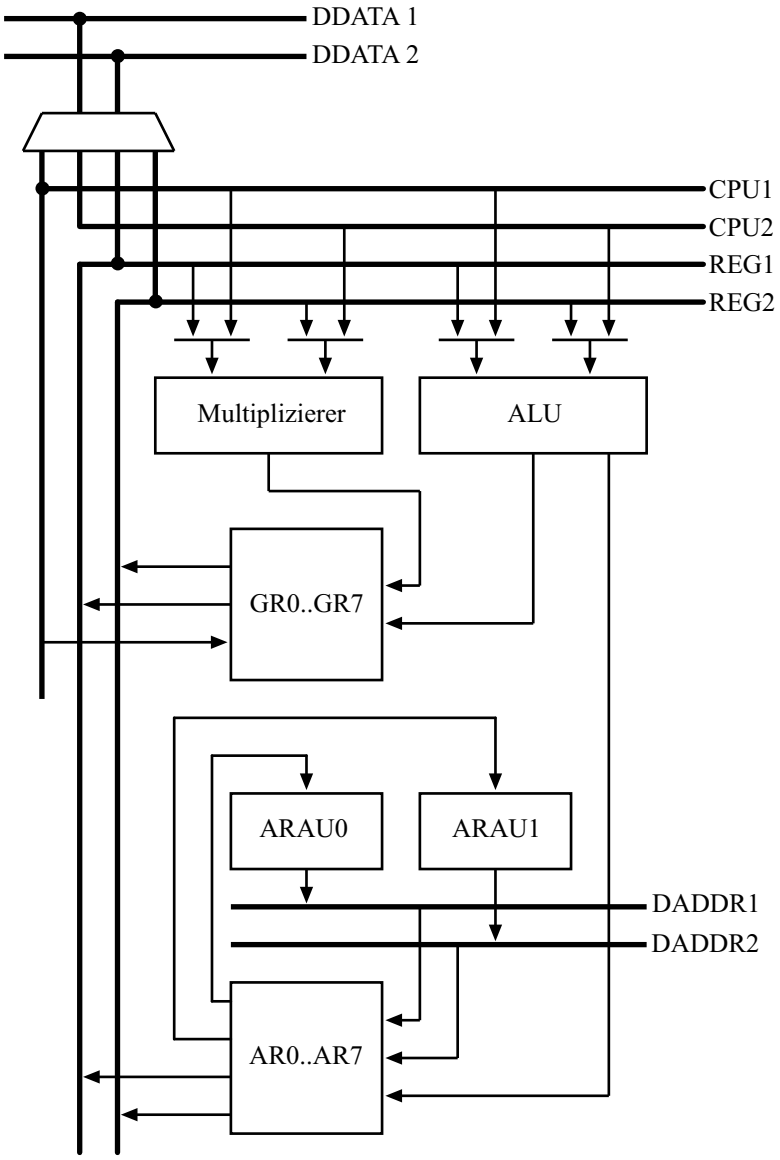


Abbildung 5.1: P1: eine C3x-ähnliche Architektur

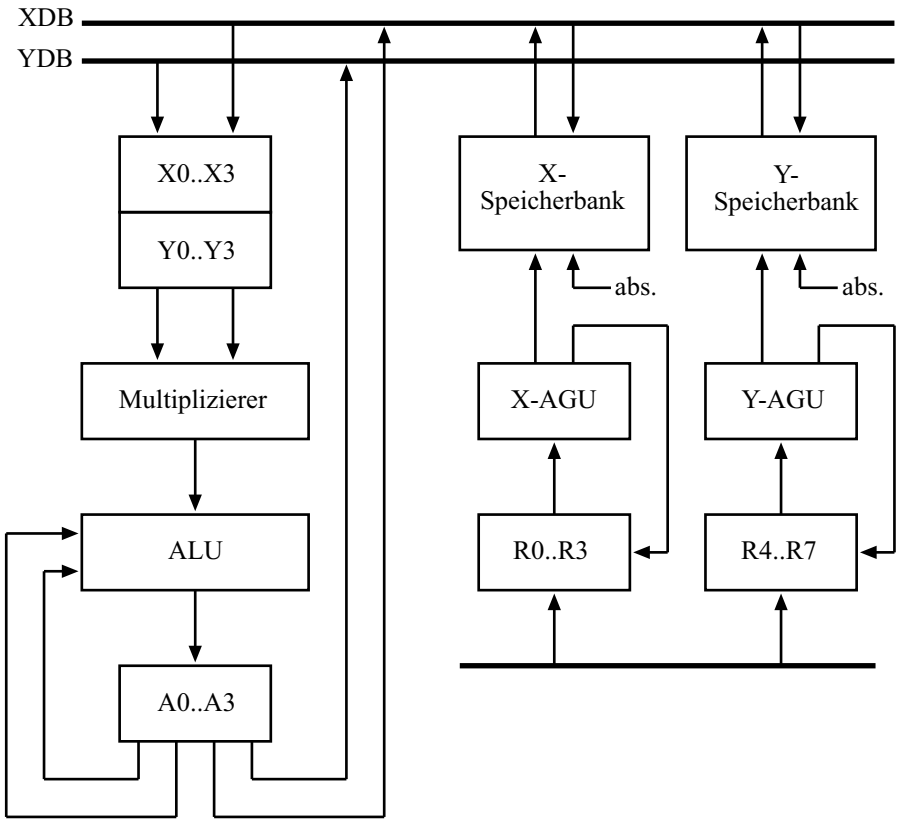


Abbildung 5.2: P2: eine DSP56000-ähnliche Architektur

5.1.3 P3: VLIW-Architektur

Für unsere dritte Prozessorarchitektur orientieren wir uns an modernen VLIW-Prozessoren. Prozessoren nach dem VLIW-Modell besitzen mehrere orthogonale parallele Einheiten. Ein Befehlswort eines solchen Prozessors besteht aus jeweils einem Befehl für jede Einheit. Die Architektur dieses Prozessors besteht aus zwei Einheiten, die jeweils (bis auf die Register) unserem ersten Prozessormodell P1 entsprechen. Dieser Prozessor kann also theoretisch in jedem Schritt vier Befehle parallel ausführen, die jeweils Rechenbefehle oder Datentransfers sein können. Den beiden Einheiten stehen gemeinsam 16 Adressregister AR0–AR15 und 16 allgemeine Register GR0–GR15 zur Verfügung. Wir werden diese Prozessorarchitektur benutzen, um das Parallelisierungspotential unserer Programme und derartiger Prozessoren zu untersuchen. Die Befehle dieses Prozessors entsprechen denen von P1, mit der Änderung, dass statt AR0–AR7 bzw. GR0–GR7 bei diesem Prozessor AR0–AR15 bzw. GR0–GR15 verwendet werden können.

5.2 Beispielprogramme

Dieser Abschnitt beschreibt die Programme, mit denen wir den Codegenerator untersuchen. Wir haben für diesen Zweck die Grundblöcke einiger typischer DSP-Aufgaben spezifiziert. Da wir zur Zeit keinen Compiler besitzen, der die Programmgraphen der Grundblöcke generiert, ist die Erstellung solcher Beispiele sehr aufwendig. Aus diesem Grund mussten wir uns auf einige überschaubare Beispiele konzentrieren. Sie bieten trotzdem genügend Spielraum, um die Eigenschaften des Codegenerators zu untersuchen. Die Beispiele sind allesamt die Rümpfe von Schleifen, wobei wir die Behandlung von Zählvariablen und Schleifenbedingungen weggelassen haben. Um die Komplexität der Beispiele zu erhöhen, können die Schleifen beliebig oft aufgerollt werden. Dabei werden mehrere Iterationen der nicht aufgerollten Schleife in einer Iteration der aufgerollten Schleife ausgeführt und die Anzahl der Iterationen um den entsprechenden Faktor verringert. Aus 20 Iterationen (20×1) werden bei vierfachem Aufrollen also 5 Iterationen der vierfachen Länge (5×4).

5.2.1 Berechnung eines Skalarprodukts

Eine der grundlegendsten Aufgaben für die digitale Signalverarbeitung und andere Anwendungen ist die Berechnung eines Skalarprodukts. Sie soll uns deshalb als erstes Beispiel dienen. Der Programmgraph der Skalarprodukt-Berechnung wurde bereits mehrfach gezeigt (zum Beispiel auf Seite 34). Die Werte ap und bp zeigen auf die Elemente der zu multiplizierenden Vektoren a und b, s enthält das Teilskalarprodukt. Die Werte s und s', ap und ap' sowie bp und bp' besitzen zyklische Abhängigkeiten, da sie jeweils die gleichen Werte in aufeinanderfolgenden Iterationen darstellen. Sie sollten deshalb jeweils im gleichen Register

gespeichert werden. Wenn die Schleife k-mal aufgerollt wird, besitzt der Programmgraph dieses Beispiels $10k + 3$ Werte und $8k$ Operationen.

5.2.2 Berechnung zweier Skalarprodukte

In vielen Anwendungen müssen mehrere Skalarprodukte berechnet werden, zum Beispiel bei der Multiplikation von Matrizen. Die einzelnen Berechnungen sind voneinander unabhängig und sollten ideal parallelisierbar sein. Um zu untersuchen, wie stark unser besonders paralleler Prozessor P3 davon profitieren kann, haben wir dieses Beispiel hinzugefügt, das zwei unabhängige Skalarprodukte berechnet. Die Werte *ap*, *bp*, *cp* und *dp* zeigen bei diesem Beispiel auf die Elemente der zu multiplizierenden Vektoren a, b, c und d, s und t enthalten die Teil-Skalarprodukte. Der Programmgraph dieses Beispiels besitzt, wenn die Schleife k-mal aufgerollt wird, $20k + 6$ Werte und $16k$ Operationen.

5.2.3 Gewichtete Vektorsumme

Bei diesem Beispiel wird die gewichtete Summe c zweier Vektoren a und b als $c = k_0a + k_1b$ gebildet. Abbildung 5.3 zeigt den Programmgraphen für die Berechnung eines Elements von c. Die Gewichte k_0 und k_1 werden in Registern übergeben. Zyklische Abhängigkeiten bestehen zwischen a_{i+1} und a_i und zwischen b_{i+1} und b_i. Außerdem dürfen die Register, in denen k_0 und k_1 gespeichert werden, nicht beschrieben werden, um die Gewichte nicht zu zerstören. Der Programmgraph der gewichteten Vektorsumme besitzt, wenn die Schleife k-mal aufgerollt wird, insgesamt $15k + 5$ Werte und $12k$ Operationen.

5.2.4 IIR-Filter

Ein typisches Einsatzgebiet für digitale Signalprozessoren ist die Anwendung von Filtern auf digitale Signalströme. Dieses Beispiel implementiert einen IIR-Filter[1], der in Abbildung 5.4 dargestellt ist. Die Abbildung verwendet eine bei Filtern übliche Darstellung: die Dreiecke sind Verstärker (Multiplizierer) mit den Koeffizienten $c_0 \dots c_2$, die Kästchen sind Verzögerungselemente. x ist das Eingabesignal und y das transformierte Signal. In der Abbildung nicht dargestellt sind die Operationen, die zum Laden bzw. Speichern der Werte und zum Aktualisieren der Adressen von x und y erforderlich sind. Die Koeffizienten des Filters werden dem Grundblock als Adresswerte übergeben, x und y als Zeiger auf die entsprechenden Datenpuffer. Zyklische Abhängigkeiten bestehen bei den Zeigern auf x bzw. y zwischen aufeinanderfolgenden Iterationen, zwischen y_i und y_{i-1}, sowie zwischen x_i und x_{i-1}. Wird die Schleife k-mal aufgerollt, besitzt der Programmgraph des IIR-Filters $13k + 10$ Werte und $11k + 3$ Operationen.

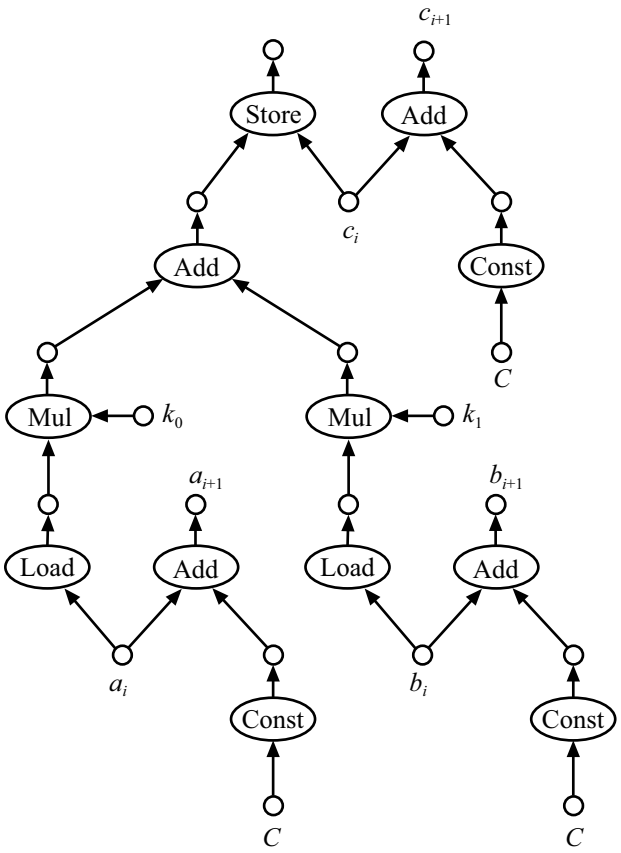


Abbildung 5.3: Gewichtete Vektorsumme

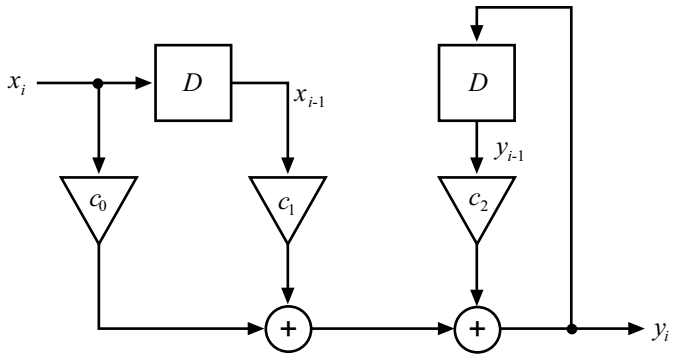


Abbildung 5.4: IIR Filter

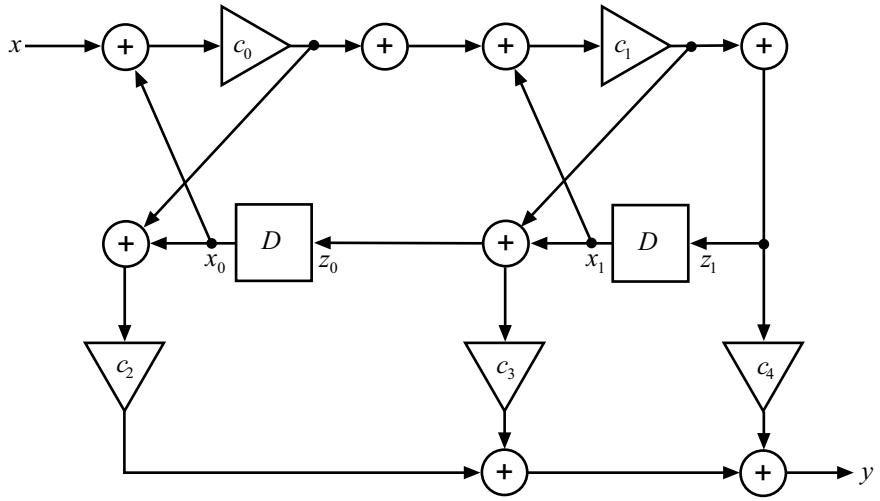


Abbildung 5.5: Lattice Filter 2. Ordnung

5.2.5 Lattice Filter

Als zweites Beispiel für einen Filter verwenden wir einen Lattice Filter 2. Ordnung, der in Abbildung 5.5 dargestellt wird. Die Speicherung der Werte erfolgt auf die gleiche Weise wie beim IIR-Filter. Zyklische Abhängigkeiten bestehen hier bei den Zeigern auf x bzw. y zwischen aufeinanderfolgenden Iterationen, zwischen x_0 und z_0 sowie zwischen x_1 und z_1. Wenn die Schleife k-mal aufgerollt wird, besitzt der Programmgraph dieses Beispiels $21k + 14$ Werte und $19k + 5$ Operationen.

5.3 Ergebnisse der Experimente

In diesem Abschnitt werden einige unserer Experimente und ihre Ergebnisse beschrieben. Die bei den Experimenten verwendeten Parametersätze sind im Anhang B aufgeführt. Die Optimierungszeiten der Beispiele wurden auf einem Intel Mobile Celeron 366MHz mit 160MB RAM bestimmt.

5.3.1 Bestimmung geeigneter Parameter

Die beiden evolutionären Algorithmen des Codegenerators besitzen eine Reihe von Parametern, die das Verfahren entscheidend beeinflussen. Im einzelnen sind dies die Populationsgröße, die Skalierungskonstante, die maximale Anzahl zu berechnender Generationen und die Anzahl beobachteter Generationen. Bei der Registerzuordnung und Befehlsplanung kommen der verwendete Rekombinationsoperator, die Rekombinationswahrscheinlichkeit, die Wahrscheinlichkeit des

[1] IIR: Infinite Impulse Response

$\|P\|$	f_{min}	f_{max}	f_{mean}	t_{opt}
5	14	910	208.2	3 min
10	12	90	31	6 min
20	12	14	13.2	13 min
50	11	14	13	35 min

Tabelle 5.2: Zusammenhang zwischen Populationsgröße und Codequalität

gerichteten Befehlstausches, die Wahrscheinlichkeit der Befehlsplankompaktierung, die Wahrscheinlichkeit der Registermutation, die Wahrscheinlichkeit der Befehlsplanmutation und die zufällige oder nichtzufällige Initialisierung der Individuen hinzu, bei der Befehlsauswahl die Rekombinationswahrscheinlichkeit und die Mutationswahrscheinlichkeit.

Die maximale Anzahl zu berechnender Generationen und die Anzahl beobachteter Generationen bilden gemeinsam die Abbruchbedingungen für den Algorithmus. Registerzuordnung und Befehlsauswahl werden abgebrochen, wenn die maximale Generationenzahl erreicht ist oder wenn sich der Zielfunktionswert des besten Individuums über die beobachteten Generationen nicht mehr verändert hat. Die Befehlsauswahl wird abgebrochen, wenn die maximale Generationenzahl erreicht ist oder wenn sich der Zielfunktionswert des besten Individuums über die beobachteten Generationen nicht mehr verändert hat und das beste Individuum eine gültige Lösung repräsentiert.

Aufgrund der zahlreichen Parameter wurden geeignete Werte für die Mutations- und Rekombinationswahrscheinlichkeiten aus der Literatur entnommen oder in frühen Experimenten bestimmt. Bei den Rekombinationsoperatoren für die Registerzuordnung und Befehlsplanung haben wir die weitaus besten Ergebnisse mit der Rekombination der Befehlspositionen (Abschnitt 4.3.4) erzielt, so dass wir diesen Operator bei allen Beispielen einsetzen.

5.3.2 Zusammenhang zwischen Rechenzeit und Codequalität

Die Abbruchbedingungen und die Größen der Populationen bestimmen die Rechenzeit, die für den Optimierungsvorgang zur Verfügung steht, und damit die erreichbare Codequalität. Um die Abhängigkeit der Codequalität von der Rechenzeit zu untersuchen, wurde bei ansonsten festen Parametern des Parametersatzes 1 die Populationsgröße der Registerzuordnung und Befehlsplanung variiert. Tabelle 5.2 zeigt die Ergebnisse der Codegenerierung für den Lattice Filter mit P1 als Zielarchitektur. $|P|$ ist die Populationsgröße der Registerzuordnung/Befehlsplanung, f_{min} der Zielfunktionswert des besten Individuums, f_{max} der Zielfunktionswert des schlechtesten Individuums und f_{mean} der mittlere Zielfunktionswert der Population. t_{opt} ist die insgesamt benötigte Rechenzeit.

Zu beobachten ist, dass sich mit zunehmender Populationsgröße die Codequalität des besten Individuums verbessert. Stärker als die Qualität des besten Individuums verbessert sich jedoch die mittlere Codequalität der Population

$\|O_P\|$	8	16	24	32	40	48	56
P1	38 s	4 min	9 min	19 min	27 min	36 min	1:01 h
P2	1 min	4 min	17 min	30 min	48 min	1 h	1:04 h
P3	47 s	3 min	15 min	29 min	34 min	42 min	47 min

Tabelle 5.3: Zusammenhang zwischen Problemgröße und Rechenzeit

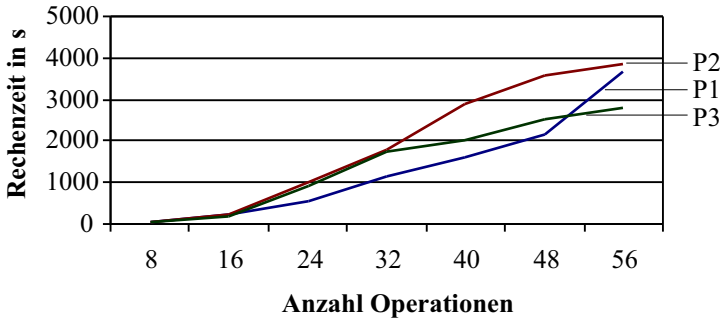


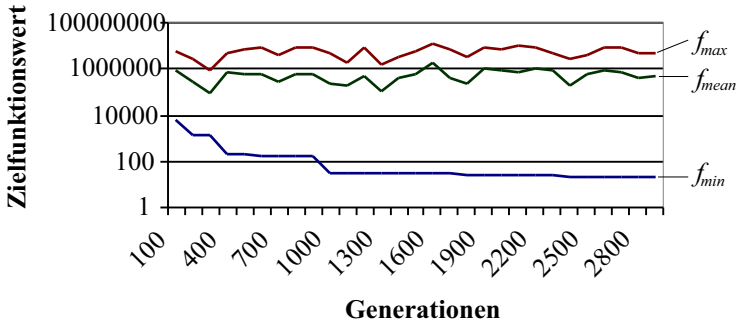
Abbildung 5.6: Zusammenhang zwischen Problemgröße und Rechenzeit

mit der Populationsgröße. Allerdings scheint es keinen Sinn zu haben, die Populationsgröße beliebig weiter zu erhöhen, da der Gewinn an Codequalität mit zunehmender investierter Rechenzeit abnimmt. Da die geeigneten Parameter der Abbruchbedingungen offenbar von der Zielarchitektur und dem zu übersetzenden Programm abhängig sind, ist wohl eine situationsabhängige Anpassung dieser Parameter erforderlich, die automatisch oder manuell erfolgen könnte.

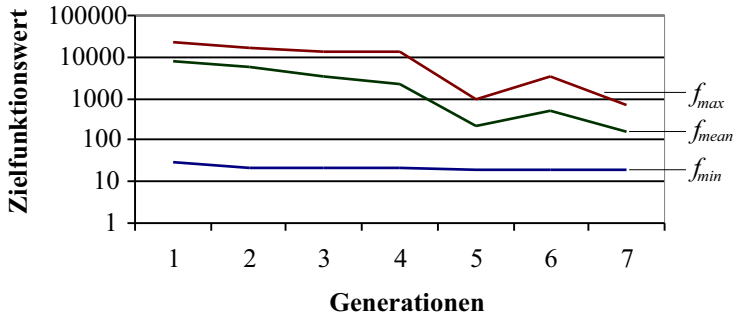
5.3.3 Zusammenhang zwischen Problemgröße und Rechenzeit

Es ist bei jedem Verfahren wichtig, den Zusammenhang zwischen der Problemgröße und der benötigten Rechenzeit zu kennen. Leider lässt sich in unserem Fall die Problemgröße nicht eindeutig bestimmen. Größe und Struktur des Programmgraphen gehen ebenso in die Problemgröße ein, wie der Aufbau der Zielarchitektur. Außerdem erschwert die stochastische Natur unseres Optimierungsverfahrens eine genaue Analyse. Um trotzdem eine Aussage über diesen Zusammenhang treffen zu können, haben wir die Komplexität eines Beispiel schrittweise durch Aufrollen der Schleife erhöht und bei festen Parametern des Parametersatzes 2 die benötigte Rechenzeit festgestellt. Tabelle 5.3 zeigt die Ergebnisse für das Skalarprodukt-Beispiel, wobei vereinfacht die Anzahl der Operationen $|O_P|$ des Programmgraphen als Komplexität des Beispiels betrachtet wird. Abbildung 5.6 veranschaulicht die Entwicklung der Optimierungszeiten für die drei Beispielprozessoren.

Die Optimierungszeiten für die verschiedenen Prozessoren entwickeln sich unterschiedlich. Der Einfluss der Zielarchitektur auf die Problemgröße ist daraus



(a) Registerzuordnung und Befehlsplanung



(b) Befehlsauswahl

Abbildung 5.7: Konvergenz des Optimierungsprozesses

deutlich erkennbar. Für P2 wird bei allen Beispielen die meiste Optimierungszeit benötigt. Das ist auf die vielen Beschränkungen dieses Prozessors zurückzuführen, die die Komplexität des Optimierungsprozesses erhöhen. Bis auf zwei Beispiele dauert die Optimierung für P3 länger als für P1. Der größere Parallelismus, den P3 gegenüber P1 bietet, erhöht hier die Zahl der Lösungsmöglichkeiten.

Anhand der gesammelten Daten kann man vermuten, dass sich die Rechenzeit des Verfahrens innerhalb eines bestimmten Bereiches in etwa linear mit der Gesamtkomplexität des Problems entwickelt. Jedoch kommt es immer wieder zu größeren stochastischen Ausreißern, die eine genauere Vorhersage der Rechenzeit erschweren.

5.3.4 Konvergenz des Optimierungsprozesses

Um Aussagen darüber treffen zu können, ob evolutionäre Algorithmen ein geeignetes Optimierungsverfahren für die Codegenerierungsprobleme darstellen, ha-

Prozessor	f_{min}	f_{max}	f_{mean}	t_{opt}
P1	12	85	21.5	35 min
P2	13	15	14.2	36 min
P3	11	19	13.1	44 min

Tabelle 5.4: Lattice Filter, $|O_P| = 24$, $|V_P| = 35$

ben wir die Folgen der besten, schlechtesten und mittleren Funktionswerte über die Generationen beobachtet. Dazu wurde eine vierfach aufgerollte Schleife zur Berechnung einer gewichteten Vektorsumme für den Prozessor P1 mit Parametersatz 3 optimiert. Abbildung 5.7 zeigt das Ergebnis, wobei für die Registerzuordnung und Befehlsplanung nur einer der vielen Optimierungsläufe dargestellt wird. Die Grafiken zeigen von oben nach unten den Verlauf der schlechtesten, mittleren und besten Funktionswerte (wegen der großen Differenzen der Funktionswerte sind diese logarithmisch skaliert).

Es ist zu beobachten, dass sich bei beiden Optimierungsverfahren der beste Zielfunktionswert stetig verringert (eine Folge davon, dass wir das beste Individuum jeder Generation sichern). Die Verbesserung pro Generation wird mit zunehmender Generationenzahl geringer. Bei der Befehlsauswahl verbessern sich tendenziell auch die mittleren und schlechtesten Funktionswerte, während bei der Registerzuordnung und Befehlsplanung keine Verbesserung dieser Werte über die Generationen zu beobachten ist. Dieses nicht unbedingt gewünschte Verhalten kann durch die relativ hohen Mutationswahrscheinlichkeiten erklärt werden, mit denen wir arbeiten. Diese sind allerdings erforderlich, damit ständig genügend neues Genmaterial in der Population erzeugt wird, um den Suchraum ausreichend abzudecken.

5.3.5 Bewertung der Codequalität

Eines der wichtigsten Kriterien für die Bewertung eines Codegenerators ist die erreichbare Codequalität. Da wir bei unserem Codegenerator von deutlich längeren Optimierungszeiten gegenüber klassischen Verfahren ausgehen, stellen wir auch entsprechende Ansprüche an den erzeugten Code. Besonders wichtig ist, dass der Parallelismus des Zielprozessors gut ausgenutzt wird und dass nach Möglichkeit spezielle Einheiten oder komplexe Befehle verwendet werden.

Um einen Überblick über die Optimierungsqualität und die Unterschiede zwischen den Zielarchitekturen zu erhalten, haben wir Beispiele unterschiedlicher Komplexität für alle drei Prozessoren mit Parametersatz 3 optimiert. Die Ergebnisse sind in den Tabellen 5.4–5.7 zusammengefasst (die Zahl der Operationen $|O_P|$ und die Zahl der Werte $|V_P|$ der Beispiele sind jeweils angegeben).

Prozessor	f_{min}	f_{max}	f_{mean}	t_{opt}
P1	11	70	18.5	49 min
P2	9	60	16.2	52 min
P3	9	925	115.1	47 min

Tabelle 5.5: IIR-Filter, 2-fach aufgerollt, $|O_P| = 25$, $|V_P| = 36$

Prozessor	f_{min}	f_{max}	f_{mean}	t_{opt}
P1	11	75	18.7	1:19 h
P2	11	845	95.7	1:05 h
P3	6	3480	952.2	38 min

Tabelle 5.6: Doppeltes Skalarprodukt, 2-fach aufgerollt, $|O_P| = 32$, $|V_P| = 46$

Vergleich 1: Doppeltes Skalarprodukt mit P1 und P3

Wir werden an einem Beispiel versuchen, die Ausnutzung des Parallelismus des Zielprozessors zu beurteilen. Als Beispiel verwenden wir die Berechnung zweier Skalarprodukte, deren Schleifen jeweils 2-fach aufgerollt wurden. Zum Vergleich dient uns der erzeugte Code für die Prozessoren P1 und P2, die sich bis auf die Zahl der Register und den verfügbaren Parallelismus gleichen. Der Code der für die beiden Prozessoren erzeugt wird, ist in den Abbildungen 5.8 und 5.9 dargestellt, die Tabellen 5.8 und 5.9 zeigen die zugehörigen Registerzuordnungen.

Die Ausführungspläne sind folgendermaßen zu lesen: die erste Spalte enthält den Zeitschritt, das Symbol „|“ steht für die parallele Ausführung von Befehlen. Zu beachten ist, dass der Postinkrement-Adressierungsmodus in unserem Prozessormodell nicht direkt dargestellt werden kann, da die Befehlsmuster Bäume sein müssen und eine explizite Betrachtung von Adressierungsarten zur Zeit nicht erfolgt. Die Adresseinheiten werden deshalb als eigenständige Einheiten modelliert, die parallel zu den anderen Einheiten Befehle ausführen können. Aus Platzgründen sind die parallel ausgeführten Befehle für den Prozessor P3 in Abbildung 5.9 jeweils auf zwei Zeilen aufgeteilt.

Die Parallelisierbarkeit dieses Beispielprogramms wird vom Codegenerator gut ausgenutzt. Die Ausführung benötigt auf dem P3 nur 6 Zeitschritte, gegenüber 11 auf dem P1. Interessant ist, dass die Optimierung für den Prozessor P3 mit 38 min nur etwa die Hälfte der Zeit in Anspruch nimmt, die eine Optimierung für den Prozessor P1 mit 1:19 h benötigt. Ein derartiges Verhältnis der

Prozessor	f_{min}	f_{max}	f_{mean}	t_{opt}
P1	20	697	166.3	2:27 h
P2	16	22795	2883.8	3:09 h
P3	12	6090	1098	2:50 h

Tabelle 5.7: Gewichtete Vektorsumme, 4-fach aufgerollt, $|O_P| = 48$, $|V_P| = 65$

```
 0                      | MPYF *AR6, *AR2, GR6 | AR2++ | AR6++
 1   ADDF GR6, GR7, GR6 | LDF *AR0, GR7        | AR0++ |
 2   LDI AR2, AR1       | MPYF *AR6, *AR2, GR3 |       |
 3   LDI AR6, AR2       | MPYF *AR3, GR7, GR2  | AR1++ | AR3++
 4   ADDF GR3, GR6, GR7 | LDI AR3, AR7         |       | AR2++
 5   ADDF GR2, GR1, GR4 | LDF *AR3, GR0        |       | AR7++
 6                      | LDI AR2, AR6         |       |
 7   LDI AR0, AR1       | LDI AR1, AR2         |       |
 8                      | LDF *AR0, GR5        |       | AR1++
 9   LDI AR1, AR0       | MPYF GR0, GR5, GR3   |       |
10   ADDF GR3, GR4, GR1 | LDI AR7, AR3         |       |
```

Abbildung 5.8: Code für doppeltes Skalarprodukt, 2-fach aufgerollt, P1

```
 0                        |                        |        | AR2++
   | LDF *AR12, GR3        | MPYF *AR2, *AR7, GR2   | AR7++  |
 1   LDI AR7, AR3          | MPYF GR3, *AR5, GR4    |        | AR12++
   | ADDF GR2, GR8, GR0    | MPYF *AR2, *AR7, GR5   |        | AR5++
 2                        | LDI AR2, AR14          | AR3++  |
   | ADDF GR5, GR0, GR8    | LDF *AR12, GR11        |        |
 3   LDI AR3, AR7          |                        | AR14++ |
   | ADDF GR4, GR14, GR10  | LDI AR12, AR11         |        |
 4                        | LDI AR14, AR2          | AR5++  | AR11++
   |                      | MPYF GR11, *AR5, GR11  |        |
 5   ADDF GR11, GR10, GR14 |                        |        |
   | LDI AR11, AR12        |                        |        |
```

Abbildung 5.9: Code für doppeltes Skalarprodukt, 2-fach aufgerollt, P3

Wert	ap_0	ap_1	ap_2	bp_0	bp_1	bp_2
Register	AR6	AR6	AR6	AR2	AR2	AR2
Wert	cp_0	cp_1	cp_2	dp_0	dp_1	dp_2
Register	AR3	AR3	AR3	AR0	AR0	AR0
Wert	s_0	s_1	s_2	t_0	t_1	t_2
Register	GR7	GR6	GR7	GR1	GR4	GR1

Tabelle 5.8: Registerzuordnung für Abbildung 5.8

Wert	ap_0	ap_1	ap_2	bp_0	bp_1	bp_2
Register	AR12	AR12	AR12	AR5	AR5	AR5
Wert	cp_0	cp_1	cp_2	dp_0	dp_1	dp_2
Register	AR2	AR2	AR2	AR7	AR7	AR7
Wert	s_0	s_1	s_2	t_0	t_1	t_2
Register	GR14	GR10	GR14	GR8	GR0	GR8

Tabelle 5.9: Registerzuordnung für Abbildung 5.9

```
 0   LDF c1, GR0         | LDF c2, GR7         |        |
 1   LDF *AR2, GR3       | MPYF GR0, GR1, GR2  |        | AR2++
 2   LDF c0, GR5         | MPYF GR0, GR3, GR0  |        |
 3   LDI AR2, AR7        | MPYF GR5, GR3, GR1  |        |
 4   ADDF GR2, GR1, GR6  | MPYF GR7, GR4, GR1  |        | AR7++
 5   ADDF GR6, GR1, GR4  |                     |        |
 6   LDF *AR2, GR1       |                     |        |
 7   STF GR4, *AR3       | MPYF GR5, GR1, GR5  |        | AR3++
 8   ADDF GR0, GR5, GR3  | MPYF GR7, GR4, GR5  |        |
 9   ADDF GR3, GR5, GR4  |                     |        |
10   LDI AR7, AR2        | STF GR4, *AR3       |        | AR3++
```

Abbildung 5.10: Code für IIR-Filter, 2-fach aufgerollt, P1

```
 0                       | LOADF c0, X1   | LOADF *R6, A3  |       | R6++
 1   MPYF X1, A3, A2     | LOADF c2, X2   | LOADF c1, Y2   |       |
 2   MPYF Y2, A3, A3     | MOVE R6, R3    |                |       |
 3   MACF Y2, Y1, A2, A1 | LOADF *R3, Y1  |                |       |
 4   MACF X2, X0, A1, A2 |                |                |       |
 5   MACF X1, Y1, A3, A3 |                | STOREF A2, *R7 | R7++ |
 6   MACF X2, A2, A3, X0 | MOVE R7, R2    |                |       |
 7                       |                | STOREF X0, *R7 | R2++ |
 8                       | MOVE R2, R7    |                | R6++ |
```

Abbildung 5.11: Code für IIR-Filter, 2-fach aufgerollt, P2

Optimierungszeiten für P1 und P3 wurde nur bei diesem Beispiel beobachtet.

Vergleich 2: IIR-Filter mit P1 und P2

An einem zweiten Beispiel soll die Ausnutzung von speziellen Einheiten bzw. komplexen Befehlen durch den Codegenerator nachvollzogen werden. Wir wählen dazu einen 2-fach aufgerollten IIR-Filter als Programmbeispiel und P1 und P2 als Zielarchitekturen. Der Code der für dieses Beispiel erzeugt wird, ist in den Abbildungen 5.10 und 5.11 dargestellt, die Tabellen 5.10 und 5.11 enthalten die zugehörigen Registerzuordnungen. (xp und yp sind die Zeiger auf die Datenpuffer für die x- und y-Werte)

Der IIR-Filter benötigt auf dem P2 trotz der starken Einschränkungen dieses

Wert	xp_0	xp_1	xp_2	yp_0	yp_1	yp_2
Register	AR2	AR2	AR2	AR3	AR3	AR3
Wert	x_{-1}	x_0	x_1	y_{-1}	y_0	y_1
Register	GR1	GR3	GR1	GR4	GR4	GR4

Tabelle 5.10: Registerzuordnung für Abbildung 5.10

Wert	xp_0	xp_1	xp_2	yp_0	yp_1	yp_2
Register	R6	R6	R6	R7	R7	R7
Wert	x_{-1}	x_0	x_1	y_{-1}	y_0	y_1
Register	Y1	A3	Y1	X0	A2	X0

Tabelle 5.11: Registerzuordnung für Abbildung 5.11

Prozessors nur 9 Zeitschritte, gegenüber 11 Zeitschritten auf dem P1. Ein Grund dafür ist, dass der P2 über einen MAC-Befehl verfügt, der eine Multiplikation und eine Addition in einem Zeitschritt ausführt, und dabei nur eine Einheit blockiert. Auf dem P1 sind dafür zwei Einheiten erforderlich, die dann für Lade- und Speicheroperationen nicht verfügbar sind. Der Codegenerator nutzt automatisch die Möglichkeiten des gegebenen Programms, den MAC-Befehl (als Beispiel für einen komplexen Befehl) einzusetzen.

An dem Ablaufplan in Abbildung 5.11 lässt sich auch eine Schwachstelle unseres Codegenerators zeigen. Den Zeigern auf die y-Werte ist das Adressregister R7 zugeordnet. Im Zeitschritt 6 wird der Wert von R7 nach R2 kopiert, dort im Schritt 7 um eine Adresse weitergesetzt, und in Schritt 8 wieder nach R7 zurückkopiert. Dieses Verhalten hat seinen Ursprung darin, dass die Befehlsauswahl nicht mit den anderen Prozessen integriert ist, sondern in einem separaten Prozess erfolgt. Als Adressinkrementbefehl wurde der Befehl ausgewählt, der R2 um eine Adresse weiter setzt. Registerzuordnung und Befehlsplanung können darauf nur noch durch das Einfügen von Kopierbefehlen zwischen R7 und R2 reagieren, was zu diesem seltsamen Code führt. Durch das Berechnen weiterer Generationen der Befehlsauswahl (und damit mehr Rechenzeit) hätte der Code noch verbessert werden können.

5.4 Auswertung und Schlussfolgerungen

Es ist sicher ein vielversprechender Ansatz, evolutionäre Algorithmen für die Codegenerierung einzusetzen. Gegenüber anderen Optimierungsverfahren besitzen evolutionäre Algorithmen den Vorteil, dass keine Heuristiken gefunden werden müssen, die sich später möglicherweise als ungeeignet herausstellen (wie bei problemspezifischen Optimierungsverfahren). Es müssen auch keine komplexen Modelle des Problems entwickelt werden, bei denen bereits die Art wie die Lösungsbedingungen formuliert sind die Leistungsfähigkeit des Codegenerators beeinflusst (wie bei der ganzzahligen linearen Programmierung). Demgegenüber steht allerdings die Schwierigkeit, geeignete Codierungen und evolutionäre Operatoren zu finden.

Zur Qualität des erzeugten Codes lässt sich sagen, dass die spezifischen Eigenschaften der Zielprozessoren hinsichtlich der Befehlsparallelität und der komplexen Befehle ausgenutzt werden. Wir haben in dieser Hinsicht also unser Ziel erreicht. Zu beobachten ist jedoch, dass die Codequalität stark von der verfügba-

ren Rechenzeit abhängt, die dem Codegenerator durch die Populationsgrößen und die Abbruchbedingungen vorgegeben wird. Wird zu wenig Rechenzeit eingesetzt, enthalten die Lösungen Stellen, an denen auf den ersten Blick weitere Optimierungsmöglichkeiten sichtbar sind. Bei heuristischen Verfahren treten solche Stellen nicht auf, weshalb es sinnvoll sein könnte, die Individuen der evolutionären Algorithmen mit Lösungen zu initialisieren, die bereits durch ein klassisches Verfahren voroptimiert wurden.

Die beobachteten Laufzeiten des Codegenerators liegen deutlich über denen klassischer Compiler. Trotzdem können Compilierzeiten dieser Größenordnung in bestimmten Fällen tolerierbar sein. Wenn man bedenkt, dass besonders im Bereich eingebetteter Systeme, etwa zur digitalen Signalverarbeitung, einzelne Algorithmen nur einmal compiliert werden, und dann teilweise über mehrere Jahre in den Systemen, in denen sie eingebaut sind, unverändert ablaufen, spielt die Compilierzeit nicht mehr so eine große Rolle. Insbesondere dann nicht, wenn durch die starke Optimierung des Codes kostengünstigere Hardware zur Implementierung der Anwendung verwendet werden kann.

Daraus ergeben sich die folgenden möglichen Anwendungsgebiete für unseren Codegenerator:

- Optimierung kritischer Programmstellen: Sicher ist dieser Codegenerator aufgrund der benötigten Rechenzeit weniger geeignet, komplette Programme zu übersetzen. Da die meisten Programme nur einen geringen Anteil an zeitkritischem Code besitzen, ist das auch gar nicht erforderlich. Vielmehr kann ein solcher Codegenerator dazu dienen, die zeitkritischen inneren Schleifen zu optimieren, während der Rest des Programms von einem klassischen Compiler übersetzt wird. Besonders bei Systemen die unter Echtzeitbedingungen betrieben werden, und die auf sehr beschränkter Hardware laufen sollen, kann sich dieser Aufwand auszahlen.

- Untersuchung und Auswahl von Prozessorarchitekturen: In der Systementwicklung hat sich in den letzten Jahren ein Paradigma durchgesetzt, dass sich durch die Stichworte „Spezifizieren, explorieren und verfeinern“ [28] beschreiben lässt. In der Spezifikationsphase wird in einem sehr frühen Stadium des Entwurfsprozesses eine ausführbare Spezifikation des Gesamtsystems erstellt. Die Explorationsphase dient dazu, zwischen verschiedenen Realisierungsalternativen dieser Spezifikation abzuwägen. In der anschließenden Verfeinerungsphase wird der Detailgrad der Systembeschreibung erhöht. Exploration und Verfeinerung werden so lange fortgesetzt, bis eine vollständige strukturelle Beschreibung des Systems vorliegt. Ein Codegenerator, wie er in dieser Arbeit entwickelt wurde, kann in der Explorationsphase dazu eingesetzt werden, die Eignung verschiedener Prozessoren für eine gegebene Anwendung zu untersuchen. Da sich der Codegenerator automatisch an die speziellen Eigenschaften eines Prozessors anpasst, kann auf dieser Basis die Auswahl von Prozessoren für die Implementierung eines Systems erfolgen.

Ein Vergleich mit verwandten Verfahren ist immer schwierig, da in den meisten Publikationen Details fehlen, um die Experimente vollständig nachvollziehen zu können. Das gilt insbesondere für Aussagen über die Codequalität. Der Laufzeitbedarf unseres Codegenerators liegt in der Größenordnung vergleichbarer nichtklassischer Verfahren, die ebenfalls auf Graphen arbeiten und die Codegenerierungsprobleme gemeinsam optimieren. Mit Verfahren, die Heuristiken anwenden oder Bedingungen an die Kostenfunktion des Problems stellen, kann unser Codegenerator bezüglich der Laufzeit jedoch nicht mithalten.

Mit unseren Untersuchungen konnten wir die zwei hauptsächlichen Problemstellen unseres Codegenerierungsverfahrens identifizieren:

1. Es treten bei den evolutionären Algorithmen zu viele ungültige Lösungen auf. Die gewählten Codierungen und die konstruierten evolutionären Operatoren können nicht alle Gültigkeitsbedingungen der Optimierungsprobleme erhalten, weshalb eine Bestrafung verletzter Bedingungen in der Zielfunktion erfolgen muss. Dadurch verbringt unser Codegenerator bei komplexeren Beispielen einen großen Teil seiner Rechenzeit damit, verletzte Bedingungen aufzulösen, anstatt gültige Lösungen zu verbessern.

2. Die Optimierungsverfahren sind nicht vollständig integriert. Durch die Abtrennung der Befehlsauswahl von der Registerzuordnung und Befehlsplanung wird zu viel Rechenzeit damit verbraucht, Registerzuordnungen und Befehlspläne für schlechte Befehlsauswahlen zu optimieren, die im nachhinein ohnehin verworfen werden.

Von einer Verbesserung dieser beiden Punkte kann eine drastische Verbesserung der Codequalität bzw. der benötigten Rechenzeit erwartet werden.

6 Zusammenfassung

In diesem Kapitel werden die durch die Diplomarbeit gewonnenen Erkenntnisse zusammengefasst. Zum einen wird der Beitrag herausgestellt, den diese Arbeit zur Entwicklung der Codegenerierung liefert. Zum anderen werden einige Ansätze für weitere Untersuchungen gegeben, mit denen diese Arbeit fortgesetzt werden könnte.

6.1 Beitrag der Diplomarbeit

Diese Diplomarbeit liefert im wesentlichen zwei Beiträge auf dem Gebiet der Codegenerierung mit nichtklassischen Verfahren:

1. Es wurde ein Verfahren zur Codegenerierung für variable Zielarchitekturen entwickelt, dass automatisch optimierten Code für eine gegebene Zielarchitektur erzeugen kann. Das Verfahren eliminiert zwei wesentliche Einschränkungen klassischer Codegeneratoren: es arbeitet vollständig auf einer Graphendarstellung des Eingabeprogramms und es optimiert Befehlsauswahl, Registerzuordnung und Befehlsplanung als integrierte bzw. gekoppelte Prozesse. Diese Eigenschaften werden allgemein als entscheidend für die Erzeugung von hochoptimiertem Code bezeichnet. Das entwickelte Verfahren passt sich von selbst an die gegebene Zielarchitektur an und nutzt den vorhandenen Parallelismus und komplexe Befehle des Prozessors. Das Codegenerierungsverfahren wurde implementiert und seine Leistungsfähigkeit an Beispielen gezeigt.

2. Das entwickelte Verfahren basiert vollständig auf evolutionären Algorithmen. Es konnte damit gezeigt werden, dass evolutionäre Algorithmen prinzipiell geeignet sind, die bei der Codegenerierung auftretenden schwierigen Optimierungsprobleme zu lösen.

6.2 Vorschläge für weitere Untersuchungen

Das entwickelte Codegenerierungsverfahren sollte eine gute Basis für weitere Untersuchungen auf dem Gebiet der Compilierung mit nichtklassischen Verfahren bieten. Durch folgende Erweiterungen könnte kurzfristig eine Verbesserung des Codegenerators ereicht werden:

- Parallelisierung des Codegenerators: Da der zur Optimierung verwendete evolutionäre Algorithmus mit einer Menge von möglichen Lösungen arbeitet die voneinander unabhängig sind, ist der Codegenerator fast ideal parallelisierbar. Es sollte eine überschaubare Aufgabe sein, den Codegenerator sowohl für den Einsatz auf Parallelrechnern als auch für den Einsatz auf den immer stärker eingesetzten Workstation-Clustern zu modifizieren. Für die Parallelisierung evolutionärer Algorithmen wurde bereits eine Reihe von Strategien beschrieben. Ein Überblick zu diesem Thema wird in Kapitel 6 von [24] gegeben.

- Begrenzung des Suchraumes: Bei dem in dieser Arbeit vorgestellten Verfahren findet keine Begrenzung des Suchraumes statt. Es wurde jedoch mehrfach beschrieben, dass mit einer geeigneten Begrenzung des Suchraumes — ohne dabei optimale Lösungen zu verlieren — eine beträchtliche Verbesserung der Laufzeit erreicht werden kann. Da wir mit einem stochastischen Verfahren arbeiten, bedeutet eine Verbesserung der Laufzeit gleichzeitig eine Verbesserung der Qualität, die in einer vorgegebenen Zeit erreicht werden kann. Insbesondere der Prozess der Registerzuordnung und Befehlsplanung würde davon stark profitieren. In [18] und [17] wird die Einschränkung des Suchraumes von ILP-basierten Befehlsplanern beschrieben. Ein einfaches Verfahren, das auch in unseren Codegenerator mit geringem Aufwand integriert werden könnte, ist die Begrenzung der möglichen Planungsschritte für die einzelnen Befehle durch die Berechnung des frühestmöglichen und des spätestmöglichen Zeitschrittes, für die jeder Befehl geplant werden kann. Das Zeitintervall, in dem ein Befehl geplant werden kann, wird dabei durch die Befehle eingeschränkt, von denen dieser Befehl abhängig ist bzw. die von diesem Befehl abhängen. In [33] wird dieses Verfahren bei einem auf evolutionären Algorithmen basierenden Befehlsplaner angewendet. Dadurch wird eine Verkleinerung des Suchraumes um viele Größenordnungen erreicht.

Soll die Codegenerierung mit evolutionären Algorithmen längerfristig untersucht werden, könnten die folgenden Vorschläge sinnvolle Ansatzpunkte sein:

- Neue Codierung und Operatoren: Unser Codegenerator verwendet bei größeren Beispielen sehr viel Rechenzeit dafür, verletzte Gültigkeitsbedingungen aufzulösen, was leider nicht immer gelingt. Der erste Ansatz sollte deshalb sein, nach neuen Codierungen und evolutionären Operatoren zu suchen, die per Definition möglichst viele der Gültigkeitsbedingungen erhalten. Es ist nicht klar, ob es möglich ist, in jedem Fall *alle* Bedingungen zu erhalten, aber es kann sicher eine deutliche Verbesserung erreicht werden.

- Vollständige Integration der Codegenerierungsprobleme: Die Trennung von Befehlsauswahl und Registerzuordnung und Befehlsplanung in zwei gekop-

pelte Prozesse behindert offensichtlich die Erzeugung von Code guter Qualität in annehmbareren Laufzeiten. Dieses Problem kann nur durch eine vollständige Integration der Codegenerierungsprobleme gelöst werden. Die Hauptschwierigkeit dabei ist, dass die Befehlsauswahl die Eingabedaten für die anderen Prozesse (nämlich den Befehlsgraphen) verändert. Möglicherweise ist es sinnvoll, auf das Konzept des Befehlsgraphen zu verzichten oder ihn durch eine andere Datenstruktur zu ersetzen. Als Ausgangspunkt für ein vollständig integriertes Modell könnte das ILP-Modell von Wilson [32] dienen.

- Optimierung über Grundblockgrenzen: Für eine vernünftige Optimierung eines Programms (d.h. eine Optimierung, die mit der durch einen erfahrenen Entwickler vergleichbar ist) muss die Einschränkung auf einzelne Grundblöcke aufgegeben werden, denn für viele Optimierungen (zum Beispiel Softwarepipelining, Retiming von Schleifen) ist der Blick über die Grundblöcke hinaus notwendig. Das nächste Ziel sollte also sein, komplette Funktionen inklusive des Kontrollflusses oder sogar ganze Module zu optimieren.

- Integration von globalen Strategien: Die Optimierung größerer Strukturen wie Funktionen und Module könnte hinsichtlich Laufzeit und Ergebnisqualität verbessert werden, indem noch eine Ebene darüber gelegt wird, die über die Anwendung globale Strategien entscheidet. Solche Strategien sind zum Beispiel das Aufrollen von Schleifen, die vor-Ort-Erweiterung von Funktionen (inlining), die Verwendung von Softwarepipelining, Retiming von Schleifen, Umordnung von Datenstrukturen etc. Die Beurteilung verschiedener Strategien könnte durch einen evolutionären Algorithmus erfolgen. In [34] wird zum Beispiel ein evolutionärer Algorithmus eingesetzt, um bei der Softwaresynthese zwischen Speicherbedarf und Laufzeit des zu übersetzenden Programms abzuwägen.

A Befehlssätze der Beispielprozessoren

A.1 P1

Syntax:	ADDF *src*1, *src*2, *dst*
Operation:	$dst \leftarrow src1 + src2$
Operanden:	*src*1: GR0–GR7 oder *AR0–*AR7 *src*2: GR0–GR7 oder *AR0–*AR7 *dst*: GR0–GR7
Beschreibung:	Die Summe von *src*1 und *src*2 wird in das Register *dst* geladen. Alle Werte sind Fließkomma-Werte.

Syntax:	MPYF *src*1, *src*2, *dst*
Operation:	$dst \leftarrow src1 * src2$
Operanden:	*src*1: GR0–GR7 oder *AR0–*AR7 *src*2: GR0–GR7 oder *AR0–*AR7 *dst*: GR0–GR7
Beschreibung:	Das Produkt von *src*1 und *src*2 wird in das Register *dst* geladen. Alle Werte sind Fließkomma-Werte.

Syntax:	LDF *src*, *dst*
Operation:	$dst \leftarrow src$
Operanden:	*src*: GR0–GR7 oder *AR0–*AR7 oder *AR0++–*AR7++ oder eine Adresse x *dst*: GR0–GR7
Beschreibung:	Der Operand *src* wird in das Register *dst* geladen. *src* und *dst* sind Fließkomma-Werte.

Syntax:	STF *src*, *dst*
Operation:	$dst \leftarrow src$
Operanden:	*src*: GR0–GR7 *dst*: *AR0–*AR7 oder *AR0++–*AR7++ oder eine Adresse x
Beschreibung:	Der Inhalt des Registers *src* wird an die Adresse *dst* geschrieben. *src* und *dst* sind Fließkomma-Werte.

Syntax:	LDI *src*, *dst*
Operation:	$dst \leftarrow src$
Operanden:	*src*: AR0–AR7
	dst: AR0–AR7
Beschreibung:	Der Operand *src* wird in das Register *dst* geladen. *src* und *dst* sind Integer-Werte bzw. Adressen.

Syntax:	*src*++
Operation:	$src \leftarrow src + k$
Operanden:	*src*: AR0–AR7
Beschreibung:	Das Adressregister *src* wird um eine Adresse weiter gesetzt.

A.2 P2

Syntax:	ADDF *src*1, *src*2, *dst*
Operation:	$dst \leftarrow src1 + src2$
Operanden:	*src*1: A0–A3 oder X0–X3 oder Y0–Y3
	*src*2: A0–A3 oder X0–X3 oder Y0–Y3
	dst: A0–A3
Beschreibung:	Die Summe von *src*1 und *src*2 wird in das Register *dst* geladen. Alle Werte sind Fließkomma-Werte.

Syntax:	MPYF *src*1, *src*2, *dst*
Operation:	$dst \leftarrow src1 * src2$
Operanden:	*src*1: A0–A3 oder X0–X3 oder Y0–Y3
	*src*2: A0–A3 oder X0–X3 oder Y0–Y3
	dst: A0–A3
Beschreibung:	Das Produkt von *src*1 und *src*2 wird in das Register *dst* geladen. Alle Werte sind Fließkomma-Werte.

Syntax:	MACF *src*1, *src*2, *src*3, *dst*
Operation:	$dst \leftarrow src1 * src2 + src3$
Operanden:	*src*1: A0–A3 oder X0–X3 oder Y0–Y3
	*src*2: A0–A3 oder X0–X3 oder Y0–Y3
	*src*3: A0–A3
	dst: A0–A3
Beschreibung:	Das Produkt von *src*1 und *src*2 wird zu *src*3 addiert und in das Register *dst* geladen. Alle Werte sind Fließkomma-Werte.

Syntax:	LOADF *src*, *dst*
Operation:	*dst* ← *src*
Operanden:	*src*: *R0–*R3 oder *R0++–*R3++ oder eine Adresse *x* *dst*: X0–X3 oder A0–A3
Beschreibung:	Der Operand *src* wird in das Register *dst* geladen. *src* und *dst* sind Fließkomma-Werte.

Syntax:	LOADF *src*, *dst*
Operation:	*dst* ← *src*
Operanden:	*src*: *R4–*R7 oder *R4++–*R7++ oder eine Adresse *x* *dst*: Y0–Y3 oder A0–A3
Beschreibung:	Der Operand *src* wird in das Register *dst* geladen. *src* und *dst* sind Fließkomma-Werte.

Syntax:	STOREF *src*, *dst*
Operation:	*dst* ← *src*
Operanden:	*src*: X0–X3 oder A0–A3 *dst*: *R0–*R3 oder *R0++–*R3++ oder eine Adresse *x*
Beschreibung:	Der Inhalt des Registers *src* wird an die Adresse *dst* geschrieben. *src* und *dst* sind Fließkomma-Werte.

Syntax:	STOREF *src*, *dst*
Operation:	*dst* ← *src*
Operanden:	*src*: Y0–Y3 oder A0–A3 *dst*: *R4–*R7 oder *R4++–*R7++ oder eine Adresse *x*
Beschreibung:	Der Inhalt des Registers *src* wird an die Adresse *dst* geschrieben. *src* und *dst* sind Fließkomma-Werte.

Syntax:	MOVEF *src*, *dst*
Operation:	*dst* ← *src*
Operanden:	*src*: X0–X3 oder Y0–Y3 oder A0–A3 *dst*: X0–X3 oder Y0–Y3 oder A0–A3
Beschreibung:	Der Operand *src* wird in das Register *dst* geladen. *src* und *dst* sind Fließkomma-Werte.

Syntax:	MOVE *src*, *dst*
Operation:	*dst* ← *src*
Operanden:	*src*: R0–R7 *dst*: R0–R7
Beschreibung:	Der Operand *src* wird in das Register *dst* geladen. *src* und *dst* sind Integer-Werte bzw. Adressen.

Syntax:	$src{+}{+}$
Operation:	$src \leftarrow src + k$
Operanden:	src: R0–R7
Beschreibung:	Das Adressregister src wird um eine Adresse weiter gesetzt.

B Parametersätze der Testprogramme

B.1 Parametersatz 1

Befehlsauswahl

Populationsgröße:	5
Skalierungskonstante:	2.0
max. Generationen:	10
beobachtete Generationen:	2
Rekombinationswahrscheinlichkeit:	0.6
Mutationswahrscheinlichkeit:	0.1

Registerzuordnung und Befehlsplanung

Populationsgröße:	10
Skalierungskonstante:	2.0
max. Generationen:	5000
beobachtete Generationen:	500
zufällige Initialisierung:	nein
Rekombinationswahrscheinlichkeit:	0.6
Wahrscheinlichkeit des gerichteten Befehlstausches:	0.4
Wahrscheinlichkeit der Befehlsplankompaktierung:	0.2
Wahrscheinlichkeit der Registermutation:	0.2
Wahrscheinlichkeit Befehlsplanmutation:	0.2

B.2 Parametersatz 2

Befehlsauswahl

Populationsgröße:	10
Skalierungskonstante:	2.0
max. Generationen:	10
beobachtete Generationen:	3
Rekombinationswahrscheinlichkeit:	0.5
Mutationswahrscheinlichkeit:	0.1

Registerzuordnung und Befehlsplanung

Populationsgröße:	10
Skalierungskonstante:	2.0
max. Generationen:	5000
beobachtete Generationen:	500
zufällige Initialisierung:	nein
Rekombinationswahrscheinlichkeit:	0.5
Wahrscheinlichkeit des gerichteten Befehlstausches:	0.5
Wahrscheinlichkeit der Befehlsplankompaktierung:	0.2
Wahrscheinlichkeit der Registermutation:	0.2
Wahrscheinlichkeit Befehlsplanmutation:	0.2

B.3 Parametersatz 3

Befehlsauswahl

Populationsgröße:	10
Skalierungskonstante:	2.0
max. Generationen:	10
beobachtete Generationen:	3
Rekombinationswahrscheinlichkeit:	0.5
Mutationswahrscheinlichkeit:	0.1

Registerzuordnung und Befehlsplanung

Populationsgröße:	20
Skalierungskonstante:	2.0
max. Generationen:	5000
beobachtete Generationen:	500
zufällige Initialisierung:	nein
Rekombinationswahrscheinlichkeit:	0.5
Wahrscheinlichkeit des gerichteten Befehlstausches:	0.5
Wahrscheinlichkeit der Befehlsplankompaktierung:	0.2
Wahrscheinlichkeit der Registermutation:	0.2
Wahrscheinlichkeit Befehlsplanmutation:	0.2

Literaturverzeichnis

[1] Alfred V. Aho: *Code Generation Using Tree Matching and Dynamic Programming.* ACM Transactions on Programming Languages and Systems, 11[4]:491–516, 1989.

[2] Alfred V. Aho, S. C. Johnson und Jeffrey D. Ullman: *Code Generation for Expressions with Common Subexpressions.* Journal of the ACM, 24[1]:146–160, 1977.

[3] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman: *Principles of Compiler Design.* Addison-Wesley, 1986.

[4] Guido Costa Souza de Araújo: *Code Generation Algorithms for Digital Signal Processors.* Dissertation, Dept. of Electrical Engineering, Princeton University, 1997.

[5] J. Bruno und Ravi Sethi: *Code Generation for a One-register Machine.* Journal of the ACM, 23[3]:502–510, 1976.

[6] G. J. Chaitin: *Register Allocation and Spilling via Graph Coloring.* In: *Proceedings of the ACM SIGPLAN'82 Conference on Programming Language Design and Implementation*, Seiten 98–105, 1982.

[7] Wei-Kai Cheng und Youn-Long Lin: *Code Generation of Nested Loops for DSP Processors with Heterogeneous Registers and Structural Pipelining.* ACM Transactions on Design Automation of Electronic Systems, 4[3], 1999.

[8] David E. Goldberg: *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[9] Marting Gotschlich und Bernhard Wess: *Automatic Generation of Constrained Expression Trees for Global Optimized DSP Assembly Code.* In: *Proceedings of the International Conference on Signal Processing Applications and Technology*, Seiten 732–736, 1996.

[10] Silvana Hanono und Srinivas Devadas: *Instruction Selection, Resource Allocation, and Scheduling in the Aviv Retargetable Code Generator.* In: *Proceedings of the 35th Design Automation Conference*, Seiten 510–515, 1998.

[11] Silvana Zimi Hanono: *Aviv: A Retargetable Code Generator for Embedded Processors*. Dissertation, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1999.

[12] F. Hoffmeister und T. Bäck: *Genetic Algorithms and Evolution Strategies: Similarities and Differences*. Technischer Bericht Sys-1/92, Universität Dortmund, 1992.

[13] John H. Holland: *Adaption in Natural and Artificial Systems*. MIT Press, 1992.

[14] John R. Koza: *Genetic Programming*. MIT Press, 1992.

[15] Rainer Leupers und Steven Bashford: *Graph Based Code Selection Techniques for Embedded Processors*. ACM Transactions on Design Automation of Electronic Systems, 5[4]:794–814, 2000.

[16] Zbigniew Michalewicz: *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 1998.

[17] M. Narasimhan und J. Ramanujam: *Improving the Computational Performance of ILP-based Problems*. In: *Proceedings of the ACM/IEEE International Conference on Computer Aided Design*, Seiten 593–596, 1998.

[18] Mukund Narasimhan: *Exact Scheduling Techniques for High Level Synthesis*. Dissertation, Dept. of Computer and Electrical Engineering, Louisiana State University, 1995.

[19] G. L. Nemhauser und L. A. Wolsey: *Integer and Combinatorial Optimization*. John Wiley and Sons, 1988.

[20] Shlomit S. Pinter: *Register Allocation with Instruction Scheduling: a New Approach*. Technischer Bericht, Dept. of Electrical Engineering, Technion – Israel Institute of Technology, 1993.

[21] Johan Van Praet, Dirk Lanneer, W. Geurts und Gert Goossens: *Processor Modeling and Code Selection for Retargetable Compilation*. ACM Transactions on Design Automation of Electronic Systems, 6[3], 2001.

[22] Todd A. Proebsting: *BURS Automata Generation*. ACM Transactions on Programming Languages and Systems, 17[3]:461–486, 1995.

[23] Ingo Rechenberg: *Evolutionsstrategie*. Friedrich Frommann Verlag, Stuttgart, 1972.

[24] Eberhard Schöneburg, Frank Heinzmann und Sven Feddersen: *Genetische Algorithmen und Evolutionsstrategien*. Addison-Wesley, 1994.

[25] Robert Sedgewick: *Algorithms in C++*. Addison-Wesley, 1992.

[26] Ron Shamir: *Advanced Topics in Graph Algorithms.* Technischer Bericht, Tel-Aviv University, 1994.

[27] Ashok Sudarsanam: *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors.* Dissertation, Dept. of Electrical Engineering, Princeton University, 1998.

[28] Jürgen Teich: *Digitale Hardware/Software-Systeme: Synthese und Optimierung.* Springer-Verlag, 1997.

[29] R. Weinberg: *Computer Simulation of a Living Cell.* Dissertation, University of Michigan, 1970.

[30] Bernhard Wess: *Minimization of Data Address Computation Overhead in DSP Programs.* Design Automation for Embedded Systems, 4:167–185, 1999.

[31] Bernhard Wess: *Simulated Evolutionary Optimization of DSP Programs.* In: *Proceedings of the International Conference on Signal Processing Applications and Technology*, 1999.

[32] Tom Wilson, Gary Grewal, Ben Halley und Dilip Banerji: *An Integrated Approach to Retargetable Code Generation.* In: *Proceedings of the 7th International Symposium on High-Level Synthesis*, Seiten 70–75, 1994.

[33] Thomas Zeitlhofer und Bernhard Wess: *Operation Scheduling for Parallel Functional Units Using Genetic Algorithms.* In: *Proceedings of the IEEE International Conf. on Acoustics, Speech, and Signal Processing*, 1999.

[34] Eckart Zitzler: *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications.* Dissertation, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich, 1999.

[35] V. Zivojnovic, J. M. Velarde, S. Christian und H. Meyr: *DSPstone: A DSP-oriented Benchmarking Methodology.* In: *Proceedings of the International Conference on Signal Processing Applications and Technology*, Seiten 715–720, 1994.