Diplomarbeit

Schleifenoptimierungen zur Ausnutzung paralleler Rechenwerke von Prozessoren der M3-DSP Plattform

Martin Horst

Diplomarbeit
am Lehrstuhl 12
des Fachbereichs Informatik
der Universität Dortmund

15. November 2001

Betreuer:

Dipl.-Inform. Markus Lorenz
Prof. Dr. Peter Marwedel

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Zielplattform 2

1.2 Wahl der Hochsprache 3

1.3 Ziel der Diplomarbeit 4

1.4 Übersicht 5

2 Arbeitsumgebung **7**

2.1 Übersicht 7

2.2 Beschreibung der GeLIR 8

2.2.1 Übersicht über die GeLIR Objekt-Struktur 9

2.2.2 Programmdarstellung 10

2.2.3 Architekturdarstellung 12

2.2.4 *GeLIR* Simulator 13

3 Schleifenerkennung **15**

3.1 Grundlagen 15

3.2 Erkennung von Single-Entry Schleifen 17

3.3 Erkennung unendlicher Schleifen 21

3.4 Erkennung von Multi-Entry Schleifen 24

3.5 Erkennung von Schleifenvariablen 27

4 Datenflußanalyse **31**

4.1 Einführung 31

4.2 Grundlagen von Datenflußanalysen 32

4.2.1 Arbeitsweise von Datenflußanalysen 32
4.2.2 Grundlegende Begriffe 33
4.2.3 Darstellung von Abhängigkeiten 33
4.2.4 Forward-/Backward- und Must-/May-Analysen 36
4.2.5 Werte- und speicherbasierte Abhängigkeiten 37
4.2.6 Datenflußverbände 37
4.2.7 Transferfunktionen 38
4.2.8 Distanzvektoren 39
4.3 Überblick über Datenflußanalysemethoden 41
4.3.1 δ-Array Datenflußanalyse 41
4.3.2 Stretched Looped 42
4.3.3 Lazy Verfahren 43
4.3.4 DSA Verfahren 44
4.3.5 Bewertung 44
4.4 δ-Array Datenflußanalyse 45
4.4.1 Schleifenkontrollflußgraphen 45
4.4.2 Datenflußverband 46
4.4.3 Verwendete Operatoren 47
4.4.4 Transferfunktionen 48
4.4.5 Iteratives Lösungsverfahren 51
4.4.6 Parametrisierung 51
4.4.7 Implementierung 53
4.4.8 Beispiel 58

5 Vektorisierung **61**
5.1 Arbeiten zum Thema Vektorisierung 61
5.1.1 Compiling C for Vectorization, Parallelization, and Inline Expansion 61
5.1.2 Exploiting Superword Level Parallelism with Multimedia Instruction Sets 63
5.1.3 A Vectorizing Compiler for Multimedia Extensions 66
5.1.4 Zusammenfassung 67

5.2 Einführung 68
5.2.1 Überblick 68
5.3 Vorausetzungen zur Vektorisierung 69
5.3.1 Schleifengrenzen und Schrittweite der Indexvariable 69
5.3.2 Erkennen von Abhängigkeiten zwischen Arrayzugriffen 70
5.3.3 Gültigkeit von Indextermen 75
5.3.4 Kontrollflußabhängigkeiten 78
5.3.5 Abhängigkeiten durch C/C++ Zeigerarithmetik 80
5.3.6 Abhängigkeiten zu nicht vektorisierbaren Operationen 84
5.3.7 Abhängigkeiten zu anderen Optimierungen 85
5.4 Durchführung der Vektorisierung 87
5.4.1 Schleifenerkennung 87
5.4.2 Ermittlung von Schleifengrenzen 90
5.4.3 Analyse der Speicherabhängigkeiten 91
5.4.4 Analyse der Abhängigkeiten von Operationen 95
5.4.5 Setzen der Vektorbefehle 96

6 Weiterführende Optimierungen 99
6.1 Zero-Overhead Hardware Loops 99
6.2 Loop Unrolling 104
6.3 Optimierte Adreßzuweisungen 112
6.4 Loop Fission 115
6.5 Function Inlining 116
6.6 Unterstützende Maßnahmen zur Vektorisierung 120

7 Untersuchung der empirischen Resultate 127
7.1 Testverfahren 127
7.2 Zero Overhead Hardware Loops 128
7.2.1 Beschreibung der Testroutinen 128
7.2.2 Ergebnisse 131
7.3 Vektorisierung 134

7.3.1 Beschreibung der Testroutinen 134
7.3.2 Ergebnisse 136
7.4 Zusammenfassung 140

8 Zusammenfassung und Ausblick **141**

A Implementierung **143**
A.1 Klassenübersicht 143
A.2 Datei- und Verzeichnisübersicht der Quellcodes 145

B Programmierrichtlinien **149**

Literaturverzeichnis **151**

Abbildungsverzeichnis

1.1 *M3-DSP* Blockschaltbild 2

2.1 Überblick Compilierungsprozeß 8
2.2 GeLIR Objekt-Struktur 10
2.3 GeLIR Objekt Hierarchie 11
2.4 Beschreibung der Zielarchitektur 13
2.5 Übersicht *GeLIR* Simulator 14

3.1 Single-Entry-Multi-Exit Schleife 18
3.2 Unendliche Schleife 23
3.3 Multi-Entry Schleife 24

4.1 Typen von Datenabhängigkeiten bei Array-Referenzen 35
4.2 Schleifenkontrollflußgraph 46

5.1 Beispiel Speicherabhängigkeiten 71
5.2 Beispiel: Datenflußgraph nach Initialisierung 82
5.3 Datenflußgraph nach Datenflußanalyse 83
5.4 Durchführung der Vektorisierung 87
5.5 Beispielschleife 89
5.6 Erstellen eines Schleifenkontrollflußgraphen 94

6.1 Durchführung der ZLoop Optimierung 103

7.1 Ergebnisse Zero Overhead Hardware Loops - Codegröße 131
7.2 Ergebnisse Zero Overhead Hardware Loops - Speicherzugriffe 132

7.3 Ergebnisse Zero Overhead Hardware Loops - Ausführungszeit 133
7.4 Ergebnisse Vektorisierung - Codegröße 137
7.5 Ergebnisse Vektorisierung - Speicherzugriffe 138
7.6 Ergebnisse Vektorisierung - Ausführungszeit 139

Tabellenverzeichnis

4.1 Indexterme zu Beispiel 4.4.4 56
4.2 Transferfunktionen zu Beispiel 4.4.6 59
4.3 Nach Initialisierung 59
4.4 Nach 1. Iteration 59
4.5 Nach 2. Iteration 59

6.1 Segmentierung nicht affiner Arrayzugriffe 108
6.2 Segmentierung affiner Arrayzugriffe 109

A.1 Implementierung: Allgemeine Klassen 143
A.2 Implementierung: δ-Array Analyse Klassen 144
A.3 Implementierung: *GeLIR* verwendente Klassen 147
A.4 Implementierung: *M3-DSP* Optimierungen 148

B.1 Implementierung: Prefixbeschreibung 150

Kapitel 1

Einleitung

Die Welt in der wir leben wird immer mehr durch elektronische Geräte beeinflußt. Nachdem Computer bereits seit längerem im Büro und im Heimbereich etabliert sind, haben sie ihren Siegeszug nun auch im mobilen Bereich angetreten.
Durch die große Verbreitung von Laptops, PDAs und Handies ist zu dem Markt der allgemeinen Prozessoren auch ein großer Markt von Digitalen Signal Prozessoren (DSPs) enstanden. Dadurch hat die Entwicklung von Software für DSPs stark an Bedeutung gewonnen.
Ein Problem bei der Softwareentwicklung für DSP Prozessoren stellt die hohe Komplexität dieser Prozessoren dar. Diese Komplexität ergibt sich aus der Irrigularität der Prozessorarchitektur, dem vorhandensein von Spezialbefehlen und der eingeschränkten Parallelität der Funktionseinheiten. Dies macht eine Softwareentwicklung auf Maschinenebene (wie z.B. bei Assembler) aufwendig und sehr fehleranfällig. Es ist somit wünschenswert, einen Teil dieser Komplexität dem Entwickler zu nehmen. Dazu bietet es sich an, die im allgemeinen durchgeführte Entwicklung von Software auf Maschinenebene durch eine einfachere, auf Hochsprachen (wie C/C++, Pascal oder Java) basierte Entwicklung, zu ersetzen.
Bei der Verwendung von Hochsprachen hängt die Qualität des erzeugten Codes jedoch stark von dem verwendeten Compiler und seinen Optimierungen ab. Moderne Compiler bieten Optimierungen zur Erhöhung der Geschwindigkeit, zur Minimierung der Codegröße und auch neuerdings zur Reduzierung des Stromverbrauches an. Dabei betrachten die meisten Compiler aber nur Optimierungen von skalaren Werten. Optimierungen auf Array- oder Zeiger-Ebene, wie sie für die Optimierung von Programmen für DSPs häufig benötigt werden, bleiben wegen ihrer hohen Komplexität in den meisten Fällen außen vor.
Für die Optimierungen von Programmen für DSPs ist es aber unerläßlich, sich mit diesen Optimierungen auseinanderzusetzen, um eine möglichst hohe Güte des erzeugten Programmcodes zu erreichen. Es soll dem Entwickler somit ein Compiler zur Hand gegeben werden, mit dem er in der Lage ist, Maschinencode zu erzeugen, der gegebene Randbedingungen bzgl. Codegröße, Ausführungsgeschwindigkeit und Engergieverbrauch einhält, ohne dabei maschinenabhängige Programmiersprachen wie Assembler verwenden zu müssen.

1.1 Zielplattform

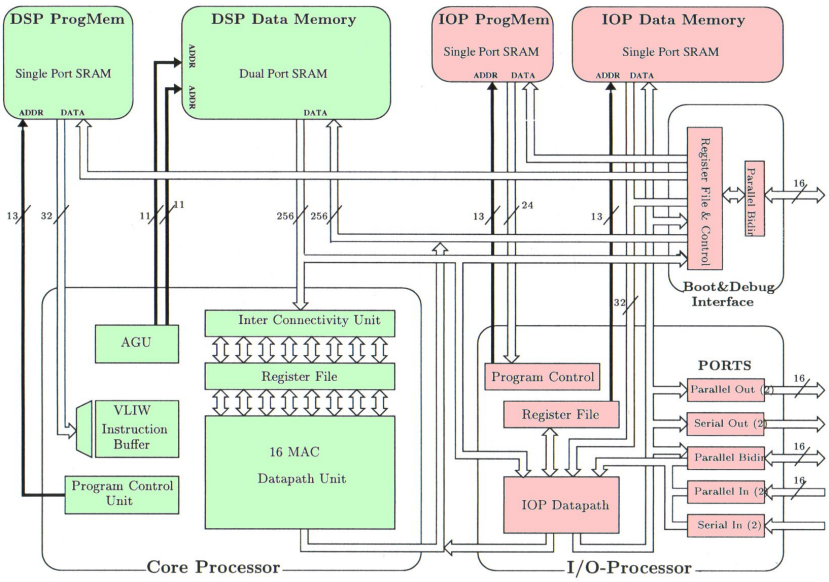


Abbildung 1.1: *M3-DSP* Blockschaltbild

Als Zielplattform für die untersuchten und realisierten Optimierungen wurden Prozessoren der *M3-DSP* Plattform gewählt. Die *M3-DSP* Plattform wurde an der Technischen Universität Dresden ([WF99], [WFL99]) speziell für die Verwendung in unterschiedlichen Einsatzgebieten Entwickelt. Zu diesem Zweck können die Prozessoren der *M3-DSP* Plattform an die gegebenen Anwendungen, für die sie eingesetzt werden sollen, angepaßt werden. Dies beinhaltet z.B. eine wählbare Anzahl von verwendeten Datenpfaden, dem Hinzufügen spezieller Funktionseinheiten oder einer speziellen Anpassung des im Prozessor vorhandenen *Inter-Communication-Network*.
Die *M3-DSP* Plattform folgt dem Prinzip der *feinkörnigen Parallelität*. Damit wird eine parallele Verarbeitung von Daten mittels einer SIMD Architektur, welche auf den vorhandenen parallelen Datenpfaden arbeitet, realisiert. Die Maschineninstruktionen des Prozessors werden in *VLIW* (Very Large Instruction Word) Form verarbeitet. Ein VLIW kann unter anderem Informationen zur Datenanipulation, zum Datentransfer oder von Adreßberechnungen enthalten. Dazu enthält der Prozessor eine Reihe von Funktionseinheiten, wie der *AGU* (Adress Generation Unit), der *DMU* (Data Manipulation Unit) und der *DTU* (Data Transfer Unit).
Zur besseren Ausnutzung des Programmspeichers, wurde eine spezielle Form der VLIWs verwendet, nämlich Tagged VLIW (TVLIW)[1]. Durch die Verwendung von TVLIW werden VLIWs durch einzelne kleinere TVLIW (32-Bit) beschrieben, was zu einer variablen Größe der VLIWs führt. Um weiterhin eine effiziente Ausführung der VLIWs zu erreichen,

[1] Auch bekannt als TISC - Tagged Instruction Set Computer bezeichnet.

wurde ein VLIW Cache integriert, welcher die neuen VLIWs anhand der TVLIWs bildet.

Der *M3-DSP* ist ein Derivat dieser Plattform. Seine Architektur besitzt 16 Datenpfade mit einer Breite von jeweils 16 Bit und der sich daraus ergebenden 256 Bit breiten Speicheranbindung. Er wurde dahingehend entwickelt, um den Anforderungen an einen geringen Stromverbrauch und Echtzeitausführung zu genügen. Eine weitere Eigenschaft ist sein geringe Größe des vorhandenen Programmspeichers, welche bei der Entwicklung von Optimierungen berücksichtigt werden muß.

Der Transfer von Daten aus dem Speicher erfolgt beim *M3-DSP* über das Zwischenegister *M*, wobei jeweils immer eine Gruppe von 32 Byte (16 Werte mit je einer Größe von 16 Bit) geladen werden muß. Nach dem Laden der Werte aus dem Speicher in das Register *M*, können die 16 Werte in die Gruppenregister *A, B, C, D* der Datenpfade transportiert werden. Das im Vorfeld erwähnte Inter-Communication-Network, bietet die Möglichkeit, beim Kopieren der 16 Werte in die Gruppenregister ein Umsortieren dieser Werte vorzunehmen. Dabei hängt die Art der möglichen Transfers von dem verwendeten Verbindungsnetzwerk ab. Zu der bereits erwähnten SIMD Ausführung, kann der *M3-DSP* auch in einem SISD (Single Instruction Single Data) Modus betrieben werden, um skalare Werte zu verarbeiten. Trotz des SISD Modus müssen aber weiterhin immer Speichergruppen in die Gruppenregister des *M3-DSP* geladen werden.

Durch den im SISD Modus entsehenden Overhead durch h"aufige redundante Speicherzugriffe, läßt sich leicht die Notwendig von einer parallelen Datenverarbeitung mittels der SIMD Instruktionen erkennen. Somit besteht ein Bedarf eines Hochsprachencompilers, welcher in der Lage ist, die SIMD Befehle des Prozessors zu unterstützen. Diese Unterstützung der SIMD (Vektor) Befehle, wird im weiteren Verlauf als Vektorisierung bezeichnet.

1.2 Wahl der Hochsprache

Im Bereich der Vektorisierung wurden bereits eine Reihe von Untersuchungen durchgeführt und Optimierungen entwickelt. Ein Großteil dieser Arbeiten basiert auf der Verwendung der Programmiersprache *Fortran*. Dies liegt darin begründet, daß die Programmiersprache Fortran häufig für wissenschaftliche Berechnungen verwendet wurde, welche auf Großrechnern durchgeführt worden sind. Da diese in vielen Fällen auch über Vektoreinheiten verfügten, wurden viele Untersuchungen zur Vektorisierung von Fortran Programmen durchgeführt. Ein weiterer Grund für die häufige Verwendung von Fortran liegt in der geringen Komplexität der Sprache. Durch das nicht Vorhandensein von Zeigerarithmetik und der Einfachheit von Schleifen verringert sich die Komplexität der Analysen, welche zur Durchführung einer Vektorisierung benötigt werden.
Fortran ist aber auf Grund seiner geringen Komplexität nicht für die Hardwarenahe Programmierung von DSPs geeignet, da eine Sprache mit höchstmöglichen Freiheitsgraden benötigt wird. Weiterhin wird eine Sprache benötigt, welche sich leicht auf andere Architekturen portieren läßt. Diese zwei Punkte werden von der Programmiersprache *C/C++* erfüllt. Doch die vorhandenen Freiheitsgrade innerhalb der Programmiersprache C/C++ führen zu einer Reihe von Problemen bei der Vektorisierung. Als Beispiel seien dazu fol-

gende Punkte zu nennen:

- Volatile Variablen können sich außerhalb des eigentlichen Programmes verändern, so daß eine Analyse nicht mehr möglich ist.
- For-Schleifen in C/C++ können weit aus komplizierter werden als in Fortran, da z.B. mehrere Bedindungen gleichzeitig abgefragt werden können. Weiterhin ist es in C/C++ möglich, diese Schleifen innerhalb der Schleife zu verlassen oder sogar von außerhalb in sie hineinzuspringen. All das macht eine Analyse komplizierter.
- Die in C/C++ vorhandene Zeigerarithmetik erschwert die Analyse von Abhängigkeiten zwischen Variablenzugriffen. Dadurch, daß es möglich ist auf Speicherbereiche Zeiger zu legen, können Abhängigkeiten nur schwer erkannt werden. Erschwerend kommt hinzu, daß Zeiger auf unterschiedliche Datentypen mit unterschiedlicher Größe "gecastet" werden können. Weiterhin müssen Funktionen, welche in ihren Schleifen wieder andere Funktionen aufrufen, bei der Vektorisierung außen vor bleiben, da es nicht möglich ist, alle vorhandenen Abhängigkeiten zu erkennen. Dieses Problem kann zum Teil dadurch umgangen werden, in dem diese Funktionen "inline" in die zu analysierende Funktion eingebunden werden.

Aus den oben genannten Gründen bedarf es spezieller Analysetechniken, die in der Lage sind, die genannten Abhängigkeiten zu erkennen. Zu solchen Analysetechniken gehören Analysen wie z.B. die δ-Array Datenflußanalyse zur Erkennung von Speicherabhängigkeiten. Aber auch Analysetechniken zur Erkennung von Schleifengrenzen, Zeigerabhängigkeiten und Abhängigkeiten zu skalaren Variablen werden für die Durchführung einer Vektorisierung benötigt.

1.3 Ziel der Diplomarbeit

Da die Hauptaufgabe von DSPs in der Verarbeitung großer Datenmengen besteht, was hauptsächlich in Schleifen durchgeführt wird, ist bei der Optimierung von Schleifen für DSPs ein großes Optimierungspotential vorhanden. Darum besteht das Ziel der Diplomarbeit in der Untersuchung und Entwicklung von Schleifenoptimierungen für die Prozessoren der *M3-DSP* Plattform.

Die Optimierung von Schleifen ist ein sehr aufwendiges und komplexes Gebiet, so daß von den hier vorgestellten Schleifenoptimierungen zwei Optimierungen für eine Implementierung ausgewählt worden sind. Bei diesen Optimierungen handelt es sich zum einen um die Vektorisierung von Speicherzugriffen und Operationen und zum anderen um die Unterstützung der Zero Overhead Hardware Loops der Prozessoren der *M3-DSP* Plattform. Zusätzlich werden eine Reihe von weiteren Optimierungen vorgestellt, mit denen eine Verbesserung der Speicherzugriffe erzielt werden kann. Dazu gehört die verbesserte Ausnutzung der Gruppenregister mittels Loop Unrolling und die optimierte Adreßgenerierung von Array-Zugriffen in Schleifen.

Zur Durchführung von Schleifenoptimierungen bedarf es einer Reihe von Abhängigkeitsanalysen. Dazu wurden im Vorfeld eine Reihe von erforderlichen Analysen untersucht, wobei die wichtigsten dieser Analysen ebenfalls implementiert worden sind. Eine der wichtigsten Analysen besteht in der Erkennung von Abhängigkeiten bei Speicherzugriffen auf Arrays. Zur Erkennung solcher Abhängigkeiten werden in Kapitel 4 eine Reihe von Analysemethoden vorgestellt, wobei detailliert auf die verwendete und implementierte δ-Datenfluß Analyse eingegangen wird. Zu den weiteren erforderlichen Analysen gehören die Erkennung von Schleifengrenzen, die Erkennung von Abhängigkeiten bzgl. skalaren Variablen und Zeigern und die Überprüfung der Gültigkeit von Indextermen bei Arrayzugriffen.

1.4 Übersicht

In Kapitel 2 wird zu Beginn eine detailierte Beschreibung der Arbeitsumgebung gegeben. Dabei wird die *GeLIR* IR Darstellung beschrieben, auf der die im Rahmen dieser Diplomarbeit entwickelten Optimierungen durchgeführt worden sind. Das Kapitel 3 geht auf die Erkennung von Schleifen ein. Weiterhin wird Verfahren zur Erkennung von Indexvariablen innerhalb von Schleifen vorgestellt. Dieses Kapitel dient als Basis der im weiteren Verlauf dargestellten Analysen und Optimierungen. Kapitel 4 behandelt das Thema Array-Datenflußanalysen, welche für die Vektorisierung von Programmen benötigt werden. Dabei werden eine Reihe von möglichen Analysemethoden vorgestellt. Der Hauptteil des Kapitels beschäftigt sich dann mit der δ-Array Datenflußanalyse, welche im Rahmen dieser Diplomarbeit implementiert worden ist. Dazu wird zuerst auf den theoretischen Hintergrund dieses Verfahrens eingegangen. Im weiteren Verlauf werden dann die für die Optimierungen benötigten Änderungen und Verbesserungen an diesem Verfahren erläutert. Kapitel 5 bildet den Hauptteil dieser Diplomarbeit, nämlich die Vektorisierung von Speicherzugriffen. Zum Beginn dieses Kapitels werden einige Arbeiten vorgestellt, welche das Thema Vektorisierung behandeln. Im Anschluß wird ein Überblick über die durch eine Vektorisierung entstehenden Abhängigkeiten und der somit erforderlichen Analysen gegeben. Der Abschluß dieses Kapitel bildet dann die Bschreibung, wie die Vektorisierung praktisch durchgeführt worden ist. In Kapitel 6 werden eine Reihe weitere Schleifenoptimierungen für die Prozessoren *M3-DSP* Plattform vorgestellt. Dabei wird detailliert auf die im Rahmen dieser Diplomarbeit realisierte Zero Overhead Hardware Loop Optimierung eingegangen. Zum Schluß dieses Kapitels werden eine weitere Reihe von Optimierungen vorgestellt, welche eine fr die Vektorisierung unterstützende Eigenschaft besitzen. Kaptiel 7 stellt die Ergebnisse dar, welche durch die implementierten Optimierungen erreicht worden sind. Das abschließende Kapitel 8 faßt noch einmal kurz die in dieser Diplomarbeit erreichten Ziele zusammen und gibt einen Überblick über weitere Verbesserungen der vorgestellten und realisierten Verfahren.

Kapitel 2

Arbeitsumgebung

Dieses Kapitel beschreibt die Arbeitsumgebung, in der die Optimierungen dieser Diplomarbeit entwickelt worden sind. Hierzu wird zunächst in Kapitel 2.1 der Weg eines C-Programmes vom ersten Übersetzungsschritt über die einzelnen Optimierungen bis zum ausführbaren Assemblerprogramm skizziert und danach in Kapitel 2.2 eine detaillierte Übersicht über die verwendete Zwischendarstellung *GeLIR* (*Low-Level Intermediate Representation*) gegeben.

2.1 Übersicht

Die Abbildung 2.1 gibt einen Überblick über die einzelnen Compilierungs- und Optimierungschritte.

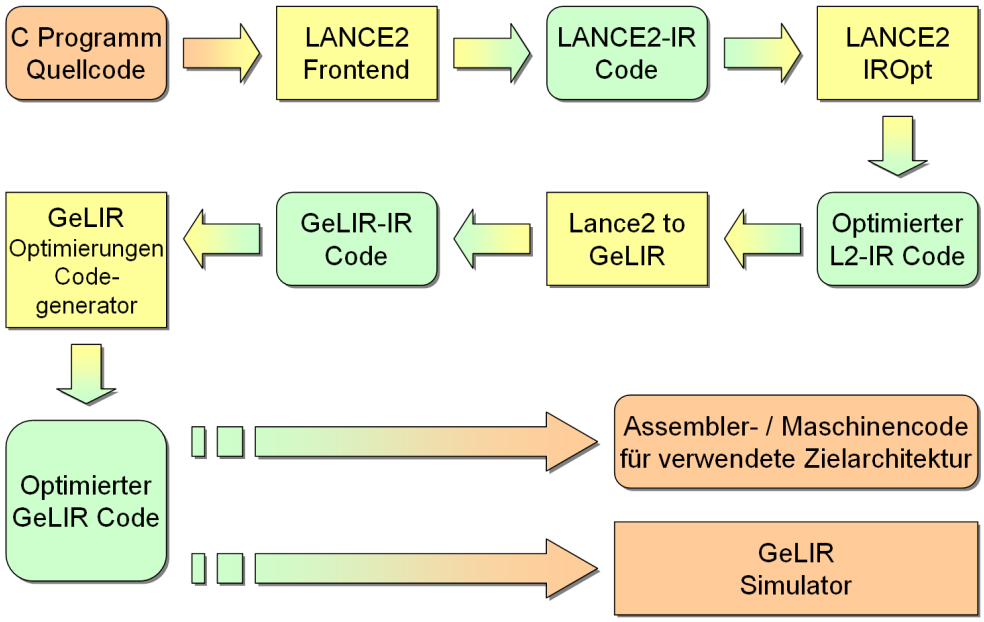


Abbildung 2.1: Überblick Compilierungsprozeß

Im ersten Schritt wird das zu optimierende C Programm mittels des *LANCE2* Frontends [RL01] in eine maschinenunabhängige Zwischendarstellung transformiert und durch das LANCE2 *iropt* Script einer Reihe von maschinenunabhängigen Standardoptimierungen unterzogen.

Danach wird der LANCE2 IR Code in eine initiale *GeLIR* -Darstellung transformiert, bei der zunächst noch keine prozessorspezifischen Merkmale vorhanden sind. Nun ist es möglich architekturspezifische Merkmale abzulegen, die bei den nachfolgenden *GeLIR* -Optimierungen berücksichtigt werden können. Dies umfaßt Standardoptimierungen wie die Dead-Elimination oder die im Rahmen dieser Diplomarbeit durchgeführten Schleifen-optimierungen.

Nach Abschluß der Optimierungen ergibt sich die Möglichkeit mit Hilfe eines Simulators, die Gültigkeit der durchgeführten Optimierungen auf unterschiedlichen Abstraktionsebenen zu validieren. Weiterhin kann der maschinenabhängige *GeLIR* Code als entsprechende Assemblerdarstellung rausgeschrieben werden.

2.2 Beschreibung der GeLIR

Dieser Teil der Diplomarbeit erläutert detailliert die verwendete Zwischendarstellung für Compiler, auf der alle Analysen und Optimierungen durchgeführt worden sind. Die

GeLIR ist eine objektorientierte Klassenbibliothek zur Darstellung von Programmen und der Möglichkeit prozessorspezifische Architekturmerkmale anzugeben. Sie stellt eine Weiterentwicklung der von Steven Bashford [SB00] eingeführten constraintbasierten Zwischenrepräsentation *CoLIR* (constraint based low-level intermediate representation) dar.
Da es heutzutage eine Vielzahl von Prozessoren gibt, welche sich hinsichtlich ihrer Architektur stark unterscheiden, werden viele Optimierungen für unterschiedliche Plattformen neu entwickelt. Daher bietet es sich an, diese Optimierungen auf einer einheitlichen Zwischendarstellung durchzuführen, die als Austauschformat auf den einzelnen Abstraktionsebenen dient. Man benötigt somit eine Schnittstelle, die zum einem einen gewissen Abstraktionsgrad von der zu unterstützenden Hardware bietet, zum anderen aber auch die Möglichkeit bietet, auf gewisse Hardwareeigenschaften einzugehen.
Und genau hier greift die *GeLIR* beide Punkte auf und ermöglicht es dem Entwickler generische Optimierungen zu schreiben, aber auch auf spezifische Hardwareeigenschaften einzugehen.

2.2.1 Übersicht über die GeLIR Objekt-Struktur

Dieser Abschnitt gibt einen Überblick über die *GeLIR* Objekt-Struktur und das Zusammenspiel der einzelnen *GeLIR* Komponenten.

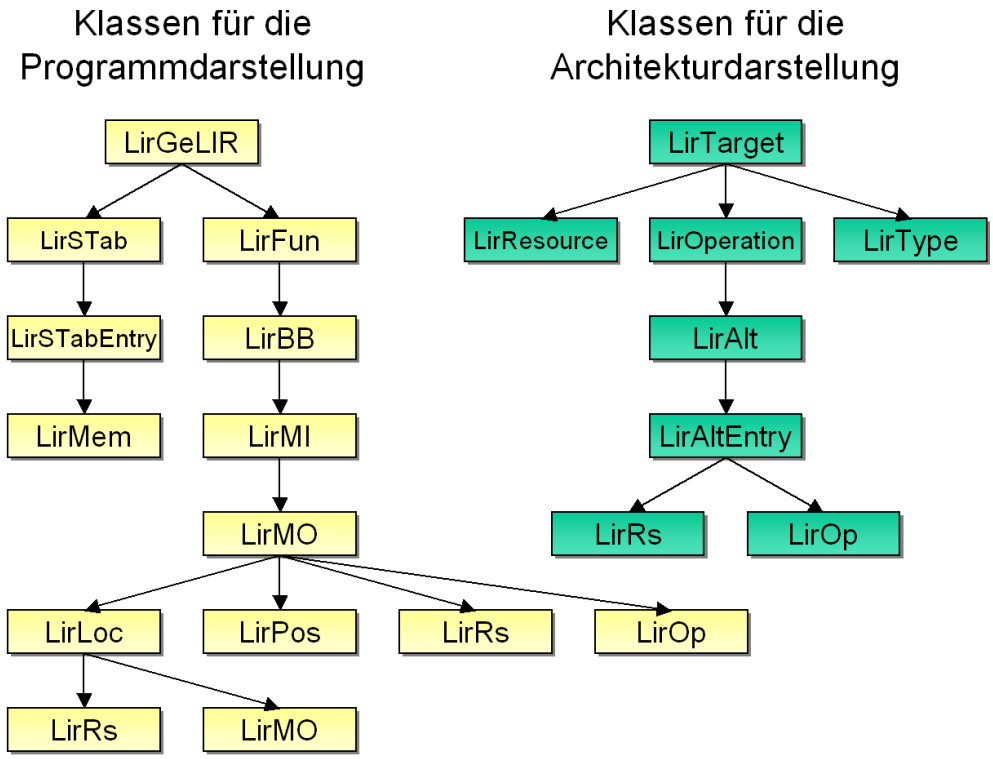


Abbildung 2.2: GeLIR Objekt-Struktur

Die *GeLIR* Objektstruktur unterteilt sich in zwei große Bereiche. Der erste Bereich dient dabei zur Darstellung des eigentlichen Programmes wohingegen der zweite Teil zur Darstellung der architekturspezifischen Merkmale dient.
Somit ist es möglich Prozessorspezifische Befehle, wie Vektoroperationen oder auch Hardwareschleifenbefehle, innerhalb der *GeLIR* darzustellen.

2.2.2 Programmdarstellung

Die *GeLIR* Programmdarstellung ist hierarchisch aufgebaut und entspricht dabei einer Baumdarstellung, wie es aus der Abbildung 2.3 ersichtlich wird.

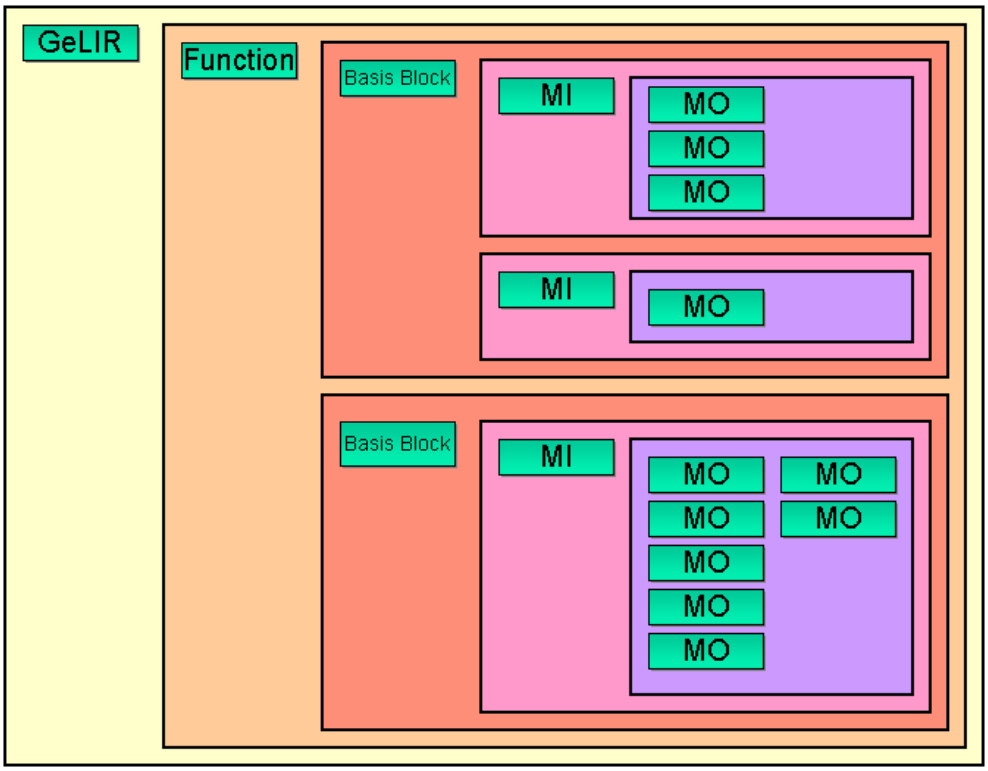


Abbildung 2.3: GeLIR Objekt Hierarchie

Im folgenden werden alle *GeLIR* Objekte zur Programmdarstellung detailliert erklärt:
Das *LirGeLIR* Objekt steht an oberster Stelle der Objekt-Hierarchie. Es beinhaltet alle Informationen über das zu übersetzende Programm im 3-Adreßformat. Alle Funktionen des Programmes werden dabei in einer Liste von *LirFun* Objekten gespeichert. Weiterhin enthält das *LirGeLIR* Objekt zusätzlich zu der Liste aller Funktionen ein *LirSTab* Objekt, welches alle globalen Variablen beinhaltet.
In dem Objekt *LirFun* wird der Programmcode der Funktion in einzelnen Basisblöcken gespeichert, welche von dem Objekttyp *LirBB* sind. Zusätzlich zu dem eigentlichen Programmcode beinhalten die *LirFun* Objekte noch Informationen über die lokalen Variablen der Funktion, welche wiederum in einem *LirSTab* Objekt gespeichert sind. Der Programmfluß zwischen den Basisblöcken wird durch einen Kontrollflußgraphen beschrieben, welcher ebenfalls Teil des *LirFun* Objektes ist.
Die *LirBB* Objekte dienen als Container der eigentlichen Maschinen-Instruktionen (MI). Basisblöcke speichern zu den eigentlichen Maschinenbefehlen noch Informationen über Datenabhängigkeiten, wobei die *Flow*-, *Output*- und *Anti*-Abhängigkeiten berücksichtigt werden. Zusätzlich enthalten Basisblöcke noch Informationen welche Variablen in diesem Basisblock definiert werden *(DefsOut)* und welche Variablen in vorherigen Basisblöcken definiert worden sind und hier verwendet werden *(VarsIn)*. Diese Informationen sind für

eine Reihe von Optimierungen wichtig und ersparen dem Entwickler zum Teil aufwendige Datenflußanalysen bzw. erleichtern die Implementierung bestimmter Datenflußanalysen.
Das *LirMI* Objekt speichert Informationen über Maschineninstruktionen, welche vom Compiler erzeugt worden sind. Eine MI kann wiederum eine oder mehrere Maschinenoperationen (MO) aufnehmen. Dadurch ist es möglich die parallele Ausführung von Instruktionen zu modellieren.
LirMO Objekte beinhalten die elementaren *GeLIR* Maschinenoperationen wie Zuweisungen, arithmetische Operationen, logische Operation oder bedingte und unbedingte Sprungbefehle.

2.2.3 Architekturdarstellung

Wie bereits erwähnt bietet die *GeLIR* die Möglichkeit prozessorspezifische Merkmale darzustellen. Diese Informationen können bei den Optimierungen dazu verwendet werden, die Optimierungen an die Eigenheiten des Prozessors anzupassen. Weiterhin werden diese Informationen im letzten Schritt der Programmübersetzung, nämlich der Codegenerierung verwendet. So werden aus diesen Informationen z.B. die Anzahl der Register des jeweiligen Prozessors ermittelt.

Das *LirTarget* Objekt enthält alle Informationen bzgl. der zu unterstützenden Zielarchitektur. Diese Informationen werden dabei in den Objekten *LirResource*, *LirOperation* und *LirType* gespeichert. Ein Überblick über die Objekthierachie ist in Abbildung 2.4 dargestellt.

- **LirTarget**
  Das *LirTarget* Objekt dient dazu als Container von Objekten, welche die eigentlichen architekurspezifischen Merkmale enthalten. Es entspricht somit dem *LirGeLIR* Objekt, nur daß es Architekturinformationen anstelle von Programminformationen enthält.

- **LirResource**
  Das *LirResource* Objekt beinhaltet Informationen bzgl. Registerfiles, Funktionseinheiten und Instruktionstypen der Zielarchitektur, wobei letztere zur Modellierung paralleler Ausführungsmöglichkeiten von MOs verwendet.

- **LirOperation**
  Das *LirOperation* Objekt verwaltet Informationen über abstrakte und architekturspezifische Operationen. Abstrakte Operationen sind dabei vordefinierte allgemeine Operationen wie Zuweisungen oder arithmetische und logische Operationen, mit denen ein gegebenes Quellprogramm mit 3-Adreßbefehlen dargestellt werden kann.

- **LirType**
  Das *LirType* Objekt dient zur Darstellung von Datentypen in der *GeLIR* Architektur. Mit dem *LirType* können sowohl einfache Datentypen wie *int* oder *char* als auch komplexe zusammengesetzte Typen wie *const char**, Strukturen oder Funktionstypen dargestellt werden.

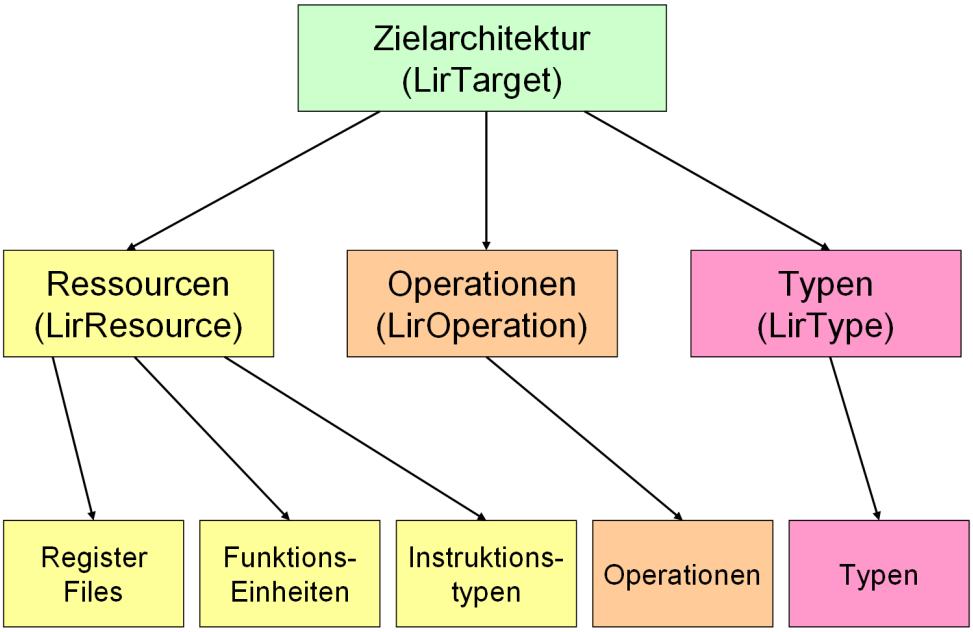


Abbildung 2.4: Beschreibung der Zielarchitektur

2.2.4 *GeLIR* Simulator

Der *GeLIR Simulator* ist ein Tool der *GeLIR* Klassenbibliothek. Er dient dazu, auf der *GeLIR* entwickelte Optimierungen zu validieren. Dem Entwickler von Optimierungen auf Basis der *GeLIR* wird damit ein mächtiges Tool zur Vefügung stellt welches die Entwicklungen von neuen Optimierungen stark beschleunigen kann.

Der Simulator wendet das Verfahren der *"Compiled Simulation"* an. Dazu wird die interne Struktur der *GeLIR* in eine C Quellcode Datei geschrieben. Zuzüglich zu dem eigentlichen Programmcode wird die Datei mit speziellen, vom Simulator erstellen, Programmumgebung versehen, welche in der Lage ist, die eigentliche Programmstruktur zu interpretieren. Diese im Weiteren als *Debug-Umgebung* bezeichnete Programmumgebung unterstützt dabei zwei verschiedene Simulationsmodi:
In dem ersten Modus wird die Simulation nur auf abstrakten Maschinenoperationen durchgeführt. Dies ist für Optimierungen sinnvoll, welche unabhängig von der eigentlichen Zielarchitektur arbeiten. Der zweite Modus dient dagegen der maschinenabhängigen Simulation, in der spezielle Instruktionen, welche nur auf der Hardware der Zielarchitektur ausgeführt werden können, simuliert werden.
Nach der Simulation werden eine Reihe von *Log-Dateien* generiert, welche Informationen wie den Rückgabewert des Programmes oder Werte ausgewählter Variablen beinhalten.

Zur Validierung wird das zu optimierende Quellprogramm mit dem *GNU C Compilers* übersetzt und das Ergenis anhand von generierten Log-Files mit dem der *GeLIR* - Simulation verglichen.

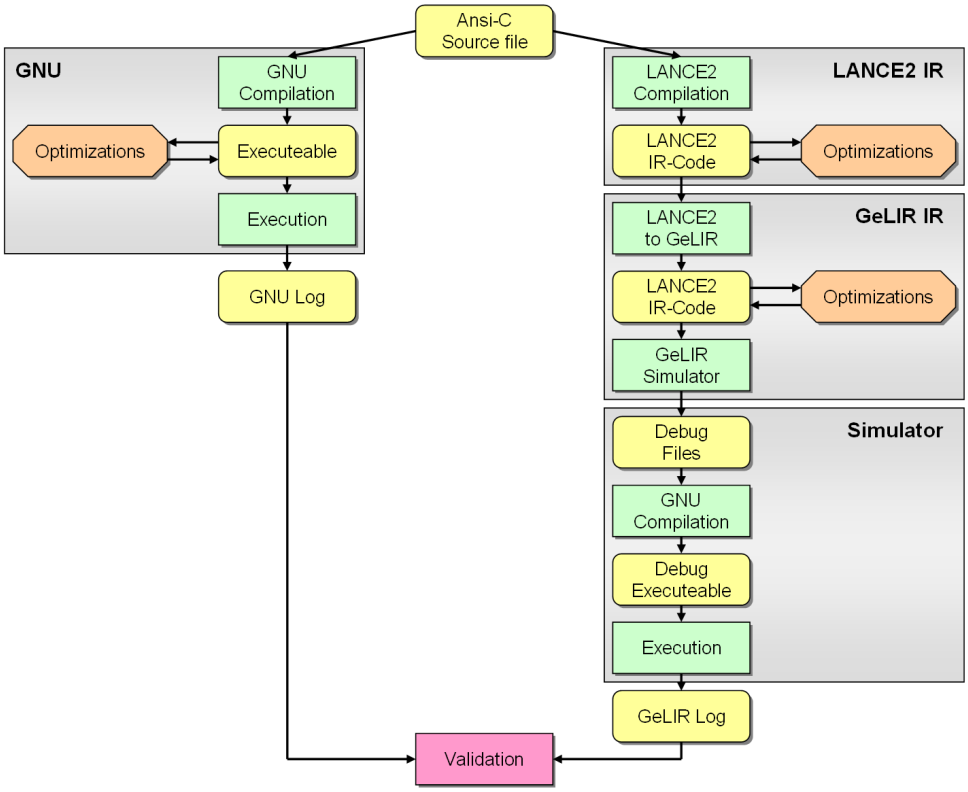


Abbildung 2.5: Übersicht *GeLIR* Simulator

Kapitel 3

Schleifenerkennung

Schleifenerkennung ist ein wichtiger Teil bei der Entwicklung eines Compilers, da ein Großteil der Zeit von Programmen in Schleifen verbracht wird. Somit ist es unerläßlich, Optimierungen auf Schleifen durchzuführen um effizienten Machinencode zu erhalten.
Um Programmcode in Schleifen optimieren zu können, bedarf es in vielen Fällen komplexer Analysen, da Abhängigkeiten zwischen Variablen über die Iterationsgrenzen einer Schleife hinweg vorhanden sind. Des weiteren werden in Schleifen häufig Arrayzugriffe verwendet, deren Abhängigkeitsanalyse ein noch komplexeres Problem darstellen.
Da Schleifenanalysen i.allg. sehr aufwendig sind, arbeiten viele Schleifenanalysen häufig nur auf *strukturierten* Schleifen, wie auch die in Kaptel 4 vorgestellte und implementierte δ-Array Datenflußanalyse. Mit strukturierte Schleifen werden Schleifen bezeichnet, welche über genau einen Basisblock betreten und über genau einen Basisblock verlassen werden. Solche Schleifen besitzen somit keine Sprünge in oder aus den Schleifenkörper heraus.

3.1 Grundlagen

Schleifenanalysen arbeiten, wie viele Optimierungen auch, auf Kontrollflußgraphen. Dabei stehen die Knoten innerhalb der Kontrollflußgraphen für die einzelnen Instruktionen des Programmes. Durch die gerichteten Kanten des Graphen werden die verschiedenen möglichen Kontrollflußwege (welche durch die Verwendung von bedingten Spüngen entstehen) modelliert.

Definition 3.1.1 *Ein* Kontrollflußgraph *(CFG) ist ein Graph $G = (N, E, s, e)$, mit einer Knotenmenge N, der Kantenmenge $E \subseteq N \times N$ (dargestellt als $n_i \rightarrow n_k$), einem Startknoten s und einem Endknoten s mit s, e $\in$ N. Wenn n ein direkter Kontrollflußvorgänger von m ist, so gilt (n, m) $\in$ E, n ist Vorgänger von m, m ist Nachfolger von n. Der Startknoten s hat keinen Vorgänger, der Endknoten e keinen Nachfolger.*

Definition 3.1.2 *Ein* Pfad π *in einem $CFG(N, E, s, e)$ besteht aus einer Folge von*

Kanten. π *beginnt an einem Knoten* n_1 *und endet an einem Knoten* n_k : $\pi = (n_1, n_2), \ldots, (n_{k-1}, n_k)$ *mit* $(n_i, n_{i+1}) \in E, i < k$. *Ein Pfad beginnend bei einem Knoten* n_i *und endend bei einem Knoten* n_k *wird durch* $n_i \rightsquigarrow n_k$ *dargestellt.*

Um die Komplexität von Kontrollflußgraphen gering zu halten, bietet es sich an, Instruktionen, welche immer sequentiell abgearbeitet werden, in einem Block zusammenzufassen. Diese Blöcke von Instruktionen werden als *Basisblöcke* bezeichnet.

Definition 3.1.3 *Ein* Basisblock *ist eine Folge von Knoten* $(n_1, \ldots, n_k)$ *eines CFG* $FG = (N, E, s, e)$, *so daß* n_i *der einzige Vorgänger von* n_{i+1} *und* n_{i+1} *der einzige Nachfolger von* n_i *ist.*

Die Erkennung von Schleifen und der darin enthaltenen Basisblöcke arbeitet in den meisten Fällen nach dem folgenden Prinzip. Zuerst wird mittels eines Graphendurchlaufes nach der Existenz von Schleifen gesucht. Um festzustellen, ob ein CFG eine Schleife enthält, genügt es zu überprüfen, ob bei einem Graphendurchlauf bereits erreichte Knoten erneut erreicht werden. Die Kante zwischen dem aktuellen und dem im Vorfeld erreichten Knoten wird als *Rückwärtskante* bezeichnet.

Definition 3.1.4 *Sei CFG* $FG = (N, E, s, e)$ *ein Kontrollflußgraph mit einer Knotenmenge N, der Kantenmenge* $E \subseteq N \times N$, *dem Startknoten s, dem Endknoten e und zwei beliebigen Knoten a, b mit s, e, a,* $b \in N$. *Wenn für alle Wege* $w_i \in FG$ *über a und b gilt:* $w_i = s \rightsquigarrow a \rightsquigarrow b \rightsquigarrow e$. *Dann bezeichnet* $b \hookrightarrow a$ *die* Rückwärtskante *der Schleife* $a \rightsquigarrow b$.

Definition 3.1.5 *Eine* Schleife *innerhalb eines CFG* $FG = (N, E, s, e)$ *besteht aus zwei ausgezeichneten Knoten e und a, wobei e den Eingangsknoten der Schleife und a den Ausgangsknoten der Schleife bezeichnet. Die Knoten a und e bilden eine Schleife innerhalb des CFG wenn gilt:* $e \rightsquigarrow a \wedge a \hookrightarrow e$.

Wie in der Einführung dieses Kapitels bereits erwähnt worden ist, genügt es nicht nur die Existenz einer Schleife und der darin enthaltenen Basisblöcke zu ermitteln. Für eine Vielzahl von Analysenmethoden und Optimierungen ist es wichtig, die Struktur der Schleife zu erkennen. So ist es in vielen Programmiersprachen (wie C/C++) möglich, von außen in einen Schleifenkörper zu springen oder den Schleifenkörper vorzeitig zu verlassen. Solche, als unstrukturierte bezeichneten Schleifen, können somit über mehrere Basisblöcke erreicht oder verlassen werden. Damit besitzen Sie nicht die für viele Analysen und Optimierungen benötigte *Single-Entry/Single-Exit* Eigenschaft.

Definition 3.1.6 *Seien die Knoten a und e die Eingangs- und Ausgangsknoten der Schleife. Eine Schleife erfüllt die* Single-Entry/Single-Exit *Eigenschaft genau dann, wenn für alle Knoten* $n_i \in \{n_0 = a, n_1, \ldots, n_{k-1}, n_k = e\}$ *der Schleife gilt:* $a \rightsquigarrow n_i \rightsquigarrow e$.

Definition 3.1.7 *Sei CFG* $FG = (N, E, s, e)$ *ein Kontrollflußgraph und* s *der Startknoten des Kontrollflußgraphen. Ein Schleife erfüllt die* Single-Entry Eigenschaft *genau dann, wenn für alle Knoten* $n_i \in \{n_0 = a, n_1, \ldots, n_k\}$ *der Schleife gilt:* $s \rightsquigarrow a \rightsquigarrow n_i$.

Definition 3.1.8 *Sei CFG* $FG = (N, E, s, e)$ *ein Kontrollflußgraph und* e *der Endknoten des Kontrollflußgraphen. Ein Schleife erfüllt die* Single-Exit Eigenschaft *genau dann, wenn für alle Knoten* $n_i \in \{n_0, \ldots, n_k = b\}$ *der Schleife gilt:* $n_i \rightsquigarrow b \rightsquigarrow e$.

Einen weiteren Spezialfall von Schleifen stellen unendliche Schleifen dar. Damit werden Schleifen bezeichnet, welche über keinen Basisblock innerhalb der Schleife verlassen werden. Da auch solche Arten von Schleifen Probleme bei Analysen und Optimierungen hervorrufen können, ist es im Vorfeld notwendig, diese Form von Schleifen zu erkennen.

Definition 3.1.9 *Sei CFG* $FG = (N, E, s, e)$ *ein Kontrollflußgraph und* e *der Endknoten des Kontrollflußgraphen. Eine Schleife ist* unendlich*, wenn für alle Knoten* $n_i \in \{n_0, \ldots, n_k\}$ *der Schleife gilt:* $n_i \not\rightsquigarrow e$.

In dem Kapitel 3.2 wird nun ein Algorithmus von Robert Morgan [RM97] vorgestellt, mit dem Schleifen mit der *Single-Entry-Single/Exit* und der *Single-Entry/Multi-Exit* Eigenschaft erkannt werdeb können.
In den den Kapiteln 3.3 und 3.4 werden dann erweiterte Versionen des Algorithmus vorgestellt, die sowohl in der Lage sind unendliche Schleifen als auch Schleifen mit der *Multi-Entry* Eigenschaft zu erkennen.

3.2 Erkennung von Single-Entry Schleifen

Die Schleifenerkennung basiert auf dem Prinzip, daß ein Basisblock, welcher der Eingang einer Schleife ist, eine Rückwärtskante besitzt die auf ihn zeigt. Um dies festzustellen, wird der gesamte Kontrollflußgraph mittels einer Tiefensuche durchlaufen. Jeder bereits besuchte Basisblock, welcher wieder erreicht wird, ist ein Eingangsblock einer Schleife.

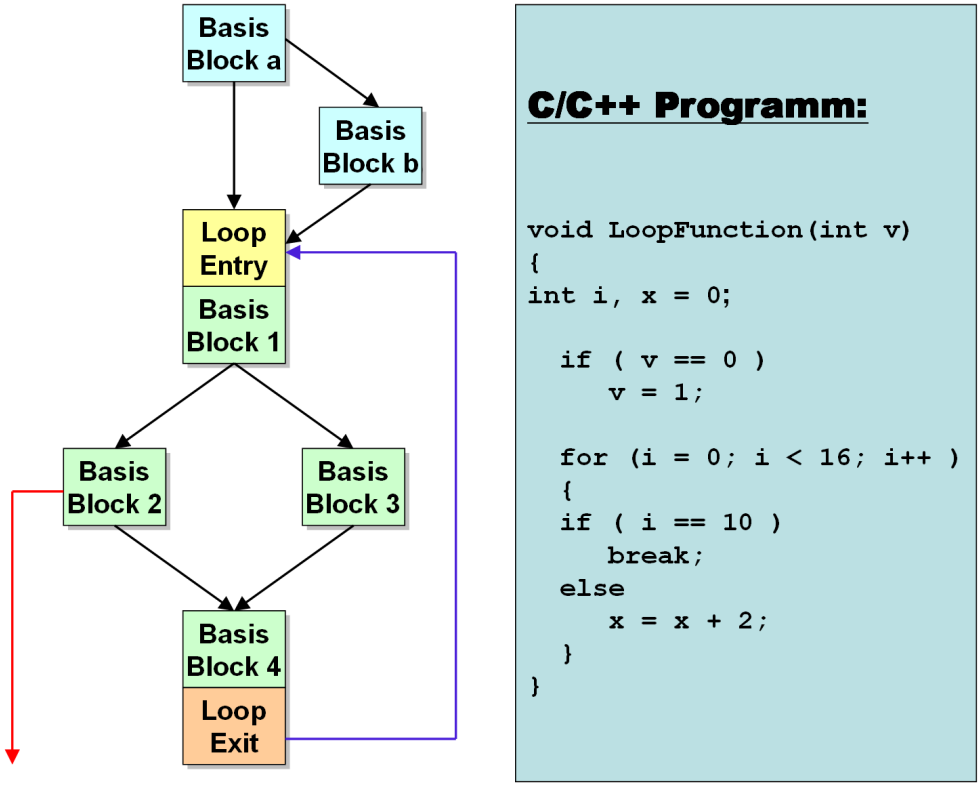


Abbildung 3.1: Single-Entry-Multi-Exit Schleife

Betrachtet man die Abbildung 3.1, so würde ein Tiefendurchlauf durch den Graphen z.B. die Basisblöcke in der Reihenfolge: [a, b, 1, 2, 4, 1, 3] erreichen. Somit wird der Basisblock 1 als Eingang der Schleife und der Basisblock 4 als Ausgang der Schleife erkannt. Es ist sinnvoll die gefundenen Rückwärtskanten zu speichern, da diese für das Auffinden aller Basisblöcke innerhalb einer Schleife benötigt werden.

Definition 3.2.1 *Ein Basisblock ist ein Eingang (Loop-Entry) einer Schleife, wenn er eine Rückwärtskante besitzt, die auf ihn zeigt.*
Sei CFG $FG = (N, E, s, e)$ *ein Kontrollflußgraph, a Basisblock und* $\{n_0 = a, n_1, \ldots, n_k\}$ *die Menge aller Basisblöcke innerhalb einer Schleife. Dann ist a der Eingang (*Loop-Entry*) der Schleife, wenn für wenigstens ein* $n_i \in \{n_0 = a, n_1, \ldots, n_k\}$ *gilt:* $n_i \hookrightarrow a$.

Definition 3.2.2 *Ein Basisblock ist ein Ausgang (Loop-Exit) einer Schleife, wenn er eine Rückwärtskante besitzt, die von ihm ausgeht. Basisblöcke, welche den Ausgang einer Schleife bilden, werden auch als Exit-Knoten bezeichnet. Sei CFG* $FG = (N, E, s, e)$ *ein*

Kontrollflußgraph, a der Eingangsbasisblock der Schleife und $\{n_0 = a, n_1, \ldots, n_k\}$ *die Menge aller Basisblöcke innerhalb der Schleife. Dann bezeichnet ein Basisblock* $n_i \in \{n_0 = a, n_1, \ldots, n_k\}$ *den Ausgang (*Loop-Exit*) einer Schleife, wenn folgendes gilt:* $n_i \hookrightarrow a$

Nachdem im ersten Schritt alle Basisblöcke, welche Eingänge von Schleifen sind, ermittelt wurden, geht es nun darum alle sich in der Schleife befindlichen Basisblöcke zu erkennen. Die Idee hinter dem Algorithmus *Find-Loop* von [RM97] basiert darauf, den Kontrollflußgraphen rückwärts zu durchlaufen und zwar solange, bis der Schleifeneingangs-Basisblock wieder erreicht wird. Der Vorteil gegenüber einer einfachen Tiefen-/Breitensuche besteht darin, daß auch Schleifen korrekt verarbeitet werden können, welche Sprünge aus der Schleife heraus besitzen, da eine normale Tiefen-/Breitensuche über Sprünge aus der Schleife heraus, die Schleife verlassen und diese wiederum durch eine äußere Schleife betreten könnte. Eine korrekte Erkennung aller sich in der Schleife befindlichen Basisblöcke würde in diesem Fall dann nicht erfolgen.

Der Find-Loop Algorithmus (Algorithmus 3.2.1) basiert auf einem Work-List Algorithmus, indem er zwei Listen verwaltet. Die *Queue* Liste beinhaltet alle Basisblöcke, welche innerhalb der Schleife liegen, deren Vorgänger aber noch nicht verarbeitet worden sind. In der *Loop* Liste werden alle in der Schleife befindlichen Basisblöcke gesammelt.

Bei der Initialisierung werden nur alle Vorgängerbasisblöcke der im Vorfeld bestimmten Schleifeneingangsbasisblöcke betrachtet, welche mit einer Rückwärtskante auf den Eingangsbasisblock der Schleife zeigen. Dadurch werden nur Wege innerhalb von Schleifen zurückverfolgt, da der Durchlauf über den Graphen immer bei dem Ausgang der Schleife beginnt.
Im Hauptteil des Algorithmus wird jeweils ein Basisblock aus der Work-List (*Queue*) entfernt und alle Vorgängerbasisblöcke betrachtet. Jeder Vorgängerbasisblock, der nicht der Schleifeneingangsbasisblock ist und noch nicht in der *Loop* Liste vorhanden ist, wird in beide Listen eingefügt. Der Algorithmus läuft so lange, bis die Work-List (*Queue*) leer ist. Die Implementierung entspricht somit einer Breitensuche, wobei die Work-List nur mit Ausgangsbasisblöcken von Schleifen initialisiert wird.

Algorithmus 3.2.1 *Find Loop: Beispielimplementierung des Find Loop Algorithmus zum Auffinden von Schleifen mit der Single-Entry/Single-Exit und Single-Entry/Multi-Exit Eigenschaft.*

```
procedure FindLoop(B: basisblock; BackEdge: Set)
  Loop  = EMPTY;
  Queue = EMPTY;

  foreach P element Pred(B) do
     if (P, B) element BackEdge then
        if (P not element Loop) And (P <> B) then
           add P to Queue
           add P to Loop
        endif
     endif
  endfor

  while Queue not is empty do
     take X from Queue
     delete X from Queue

     foreach P element Pred(X) do
        if (P <> B) and (P not element Loop) then
           add P to Queue
           add P to Loop
        endif
     endfor
  endwhile

  add B to Loop;
endprocedure FindLoop
```

An dieser Stelle sei nun ein Ablauf des *Find Loop* Algorithmus gegeben, welcher sich auf die Beispielschleife in der Abbildung 3.1 bezieht.

Schritt 1 - Tiefensuche
Im ersten Schritt wird ein Tiefendurchlauf durch den Graphen durchgeführt, welcher die Basisblöcke in der folgenden Reihenfolge findet: $\{a, b, 1, 2, 4, 1, 3\}$
Da Basisblock 4 wieder auf den bereits erreichten Basisblock 1 zeigt, ist Basisblock 1 als Eingang der Schleife und Basisblock 4 als Ausgang der Schleife indentifiziert worden. Die Kante zwischen den Basisblöcken 4 und 1 ist ein Rückwärtskante und wird somit zum Abschluß in den Menge *BackEdge* $\{(4, 1)\}$ eingefügt.

Schritt 2 - Initialisierung

Im zweiten Schritt wird nun der *Find Loop* Algorithmus mit dem Basisblock 1 aufgerufen. Dabei wird untersucht, welche Kanten von den Vorgängerbasisblöcken von Basisblock 1 in der Menge *BackEdge* liegen. Da weder (a, 1) noch (b, 1) in der Menge *BackEdge* vorhanden ist, wird nur der Basisblock 4 in die *Queue* und *Loop* Mengen eingefügt. Wir erhalten als Ergebnis nun $Loop = \{4\}$ und $Queue = \{4\}$.

Schritt 3 - Ablauf
Der Graph wird nun ausgehend vom *Exit*-Knoten der Schleife rückwärts durchlaufen. Dabei ergibt sich die folgende Abarbeitungsreihenfolge der Knoten im Graphen: Basisblock 4 wird aus der *Queue* entfernt und dessen Vorgänger betrachtet. Die erreichbaren Basisblöcke von Basisblock 4 sind die beiden Basisblöcke 2 und 3. Da diese nicht in der *Loop* Liste vorhanden sind, werden sie in die *Loop* und *Queue* Liste eingefügt. Man erhält somit $Queue = \{2, 3\}$ und $Loop = \{2, 3, 4\}$.
Im nächsten Schritt wird der Basisblock 2 aus der *Queue* Liste entfernt. Da dieser auf den Basisblock 1 zeigt, welcher der Eingang der Schleife ist, wird dieser nicht weiter betrachtet. Das gleiche passiert mit Basisblock 3, womit sich die Mengen mit folgenden Werten $Queue = \emptyset$ und $Loop = \{2, 3, 4\}$ ergeben. Da die *Queue* Liste leer ist, terminiert der Algorithmus.
Zum Abschluß wird der Basisblock 1 in die *Loop* Liste eingefügt und man erhält als Ergebnis $Loop = \{1, 2, 3, 4\}$ und hat somit alle Basisblöcke innerhalb der Schleife gefunden.

3.3 Erkennung unendlicher Schleifen

Unendliche Schleifen sind in Programmen eher selten anzutreffen, was ihr Vorhanden sein aber nicht gänzlich ausschließt. Ein Einsatzgebiet ist z.B. das Behandeln von Interrupts, indem die Schleife nur auf das Eintreffen dieser wartet, während der eigentliche Programm/-ablauf normal abläuft. Eine weitere Ursache für das Vorhandensein von unendlichen Schleifen liegt in Programmierfehlern.

Beispiel 3.3.1 *Unendliche Schleife*

```
int CountLines(const char* pcszFileName)
{
FILE* pFile;
int nLines = 0;
char szLine[256];

  pFile = fopen(pcszFileName, "rt");
  if ( pFile == NULL )
     return 0;

  /* Unendliche Schleife ohne Abbruchbedingung. */
  while ( 1 )
    {
    fgets(szLine, 256, pFile);
    nLines++;
    }

  fclose(pFile);
  return nLines;
}
```

In diesem Beispiel hat der Entwickler vergessen zu überprüfen, ob das Dateiende erreicht worden ist. Die Schleife läuft somit bis in alle Unendlichkeit weiter und das Programm terminiert nicht. Auch für solche Fälle ist es sinnvoll unendliche Schleifen zu erkennen, um den Entwickler auf Fehler dieser Art aufmerksam machen zu können.

Ein weiterer Grund unendliche Schleifen zu erkennen und auch zu entfernen besteht darin, daß eine Reihe von global optimierenden Algorithmen fehlerhafte Ergebnisse liefern, wenn sie auf unendliche Schleifen treffen. Vieler dieser Algorithmen versuchen die Anzahl der Berechnungen vom Beginn bis zum Ende einer Schleife zu minimieren. Wird das Ende der Schleife nicht gefunden, so kann dies zu einem unerwarteten Verhalten des Algorithmus führen.

Die Erkennung von unendlichen Schleifen kann mittels eines Tiefensuche-Algorithmus durchgeführt werden, welcher beginnend mit dem letzten Basisblock einer Funktion rückwärts über den Kontrollflußgraphen läuft. Da eine unendliche Schleife keinen Exit-Knoten besitzt, existiert auch keine Kante zwischen einem Knoten einer Schleife und den darunter liegenden Basisblöcken. Dies hat zur Folge, daß die Basisblöcke innerhalb der Schleife nicht erreicht werden. Alle Knoten die durch den Tiefensuche-Algorithmus nicht erreicht worden sind gehören einer unendliche Schleife an.

Das Entfernen einer unendlichen Schleife wird dann durch das Einfügen einer Kante zwischen dem Exit-Knoten des CFGs und einem beliebigen Knoten, welcher bei der Tiefensuche besucht worden ist, erreicht. Ist ein solcher Knoten im CFG nicht vorhanden, so muß zusätzlich ein solcher Knoten eingefügt werden. Diese einfache Vorgehensweise ist semantisch korrekt, da in dem eigentlichen Programm die neu hinzugefügte Kante nie verwendet wird.
Nach dem Einfügen der Kante wird der Algorithmus erneut auf den Kontrollflußgraphen angewendet und zwar solange, bis keine unendliche Schleife mehr gefunden wird.

Beispiel 3.3.2 *Erkennen einer unendliche Schleife*

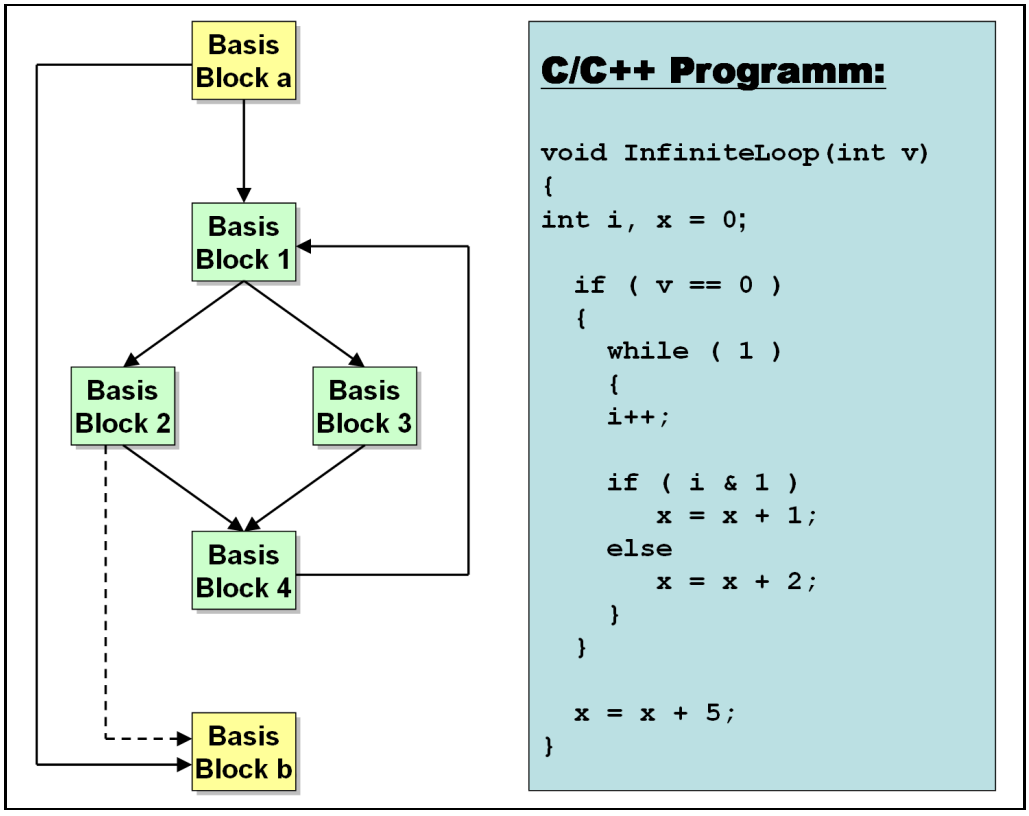


Abbildung 3.2: Unendliche Schleife

Der Tiefendurchlauf würde angewendet auf den Kontrollflußgraphen in Abbildung 3.2 die beiden Basisblöcke a und b finden. Somit wurde erkannt, daß Basisblöcke 1, 2, 3 und 4 zu einer unndlichen Schleife gehören. Um diese zu entfernen wird nun eine Kante zwischen einem beliebigen inneren Knoten des CFGs und dem Ausgangsknoten des CFGs gelegt. In dem angegebenen Beispiel 3.2 entspricht dies der Kante zwischen den Basisblöcken 2

und b.
Der zweite Tiefendurchlauf läuft nun auch über die neu hinzugefügte Kante und erreicht somit die Basisblöcke 1, 2, 3 und 4. Da alle Basisblöcke innerhalb der Funktion erreicht sind, existiert keine unendliche Schleife mehr und der Algorithmus terminiert.

3.4 Erkennung von Multi-Entry Schleifen

Multi-Entry Schleifen sind in sauber strukturierten und programmierten Programmen eigentlich nicht anzutreffen. Trotzdem ist es notwendig Multi-Entry Schleifen zu erkennen, da sie in Sprachen wie C/C++ vorkommen können.
Multi-Entry Schleifen stellen ein großes Problem dar, da viele Optimierungstechniken nur mit Single-Entry Schleifen arbeiten können. Es ist somit notwendig Multi-Entry Schleifen nicht nur zu erkennen, sondern diese ebenfalls in Single-Entry Schleifen umzuwandeln.

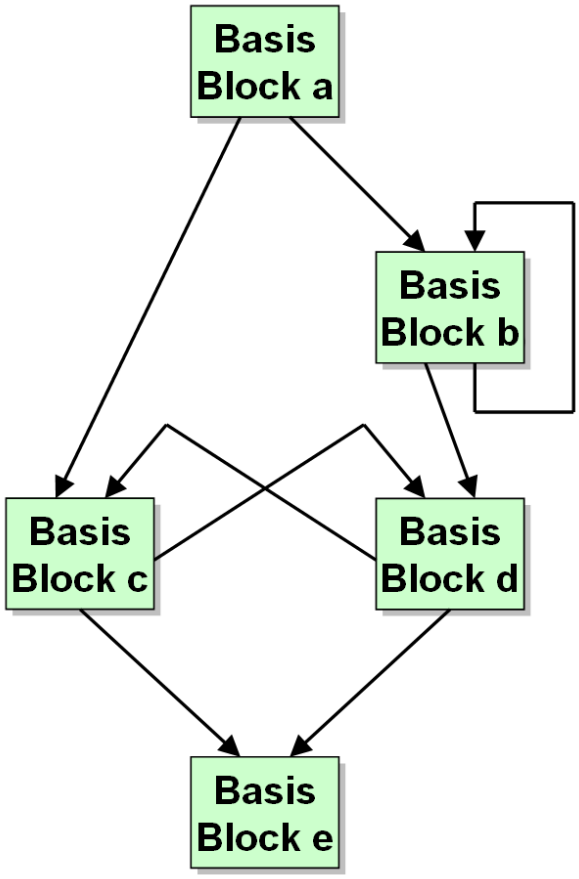


Abbildung 3.3: Multi-Entry Schleife

Das Erkennen von Multi-Entry Schleifen stellt den in Kapitel 3.2 beschriebenen Algorithmus vor ein Problem. Im ersten Schritt des Algorithmus wird eine Tiefensuche auf dem Kontrollflußgraphen durchgeführt, um den Exit-Knoten der Schleife zu ermitteln. Betrachtet man den Kontrollflußgraphen in Abbildung 3.3, so könnte ein Tiefendurchlauf durch den Graphen die Basisblöcke in der Reihenfolge $\{a, c, d, e, b\}$ erreichen. Dadurch, daß d eine Rückwärtskante auf c besitzt, wäre der Basisblock c als Schleifeneingang und der Basisblock d als Schleifenausgang erkannt worden. Ein anschließender, von Basisblock d startender, rückwärtslaufender Tiefendurchlauf durch den Graphen würde dann jedoch die Basisblöcke $\{d, c, a, b\}$ erreichen und als Basisblöcke innerhalb der Schleife identifizieren.

Um dieses Problem zu vermeiden muß der Algorithmus dahingegen modifiziert werden, daß er an geeigneter Stelle mit dem Durchlaufen des Graphen abbricht.
Die Idee dahinter besteht nun darin, den Durchlauf des Graphen an dem Knoten abzubrechen, welcher am nächsten an der Schleife liegt.

Definition 3.4.1 *Sei CFG $FG = (N, E, s, e)$ ein Kontrollflußgraph mit einer Knotenmenge N, der Kantenmenge $E \subseteq N \times N$ und zwei beliebigen Knoten a, b mit a, $b \in N$. Der Knoten a dominiert den Knoten b, wenn alle Wege zum Knoten b über den Knoten a verlaufen. Knoten a wird als* Dominatorknoten *bezeichnet.*

In der Abbildung 3.3 wäre der Dominatorknoten der Knoten b, da er die untere Schleife $\{c, d\}$ und sich selbst dominiert.

Der Algorithmus zur Erkennnung von Multi-Entry Schleifen basiert auf dem Algorithmus aus Kapitel 3.2 zur Erkennung von Single-Entry Schleifen. Dazu wird für jeden betrachteten Basisblock Z eine Menge von Basisblöcken verwaltet, welche mit *Generators(Z)* bezeichnet wird. Die Basisblöcke innerhalb der Menge *Generators(Z)* sind Basisblöcke, welche den Kontrollfluß zum eigentlichen Schleifenkörper führen. Anstelle der Ermittlung der Basisblöcke innerhalb der Schleife, werden die Basisblöcke identifiziert, welche zu der eigentlichen Schleife führen. Ist die Menge eines Basisblocks Z nicht leer (es gilt: $Generator(Z) \neq \emptyset$), so ist der Basisblock Z ein Dominatorknoten einer Multi-Entry Schleife und die Existenz einer Multi-Entry Schleife wurde erkannt. In diesem Fall bricht das Verfahren ab.

Algorithmus 3.4.1 *Find Loop: Beispielimplementierung des modifizierten Find Loop Algorithmus zum Auffinden von Multi-Entry/Single-Exit und Multi-Entry/Multi-Exit Schleifen.*

```
function FindLoop(B: basisblock; BackEdge: Set) as boolean;
  Z = B;
  Loop = Empty;
  Queue = Empty;

  for each P element Pred(B) do
      if (P, B) element BackEdge then
         if (P element Loop) And (P <> B) then
            Z = CommonDominator(Z, P);
            Add P to Queue;
            Add P to Loop;
         endif
      endif
  endfor

  if (Z <> B) then
     Generators(Z) = Generators(Z) Or Loop;
     return false; // Multi-Entry loop found
  endif

  while Queue not is empty do
     take X from Queue
     delete X from Queue

     foreach P element Pred(X) do
        if (P <> B) and (P not element Loop) then
           add P to Queue
           add P to Loop
        endif
     endfor
  endwhile

  add B to Loop;
  return true; // Single-Entry loop found
endprocedure FindLoop
```

Der Algorithmus 3.4.1 liefert false zurück, wenn eine Multi-Entry Schleife gefunden worden ist. In diesem Fall ist ein weiteres Vorgehen nötig, um die Basisblöcke innerhalb der Multi-Entry Schleife zu ermitteln. In dem Fall, daß eine Single-Entry Schleifen gefunden worden ist, arbeitet dieser Algorithmus wie der Algorithmus 3.2.1 und liefert somit alle Basisblöcke innerhalb dieser Schleife.

Zur Erkennung von Basisblöcken innerhalb von Multi-Entry Schleifen wird auf den Algorithmus von Robert Morgan [RM97] verwiesen, da eine Beschreibung dieses komplexen Verfahrens über den Rahmen der Diplomarbeit hinausgehen würde.

3.5 Erkennung von Schleifenvariablen

Das Erkennen von Induktionsvariablen stellt für die Vektorisierung einen entscheidenden Punkt dar. Vor allem das Erkennen von Induktionsvariablen in Schleifen und deren Grenzen sorgt dafür, daß Abhängigkeitsanalysen (siehe Kapitel 4 dieser Diplomarbeit) genauer durchgeführt werden können.

Definition 3.5.1 *Induktionsvariablen: Induktionsvariablen sind Variablen, deren Wert sich mittels einer arithmetischen Operation fortlaufend innerhalb eines Programmteiles verändert.*

Das hier vorgestellte Verfahren stammt von Muchnick [SM97] und wird in seinen Grundzügen erklärt, da eine detaillierte Beschreibung des Verfahrens über den Rahmen der Diplomarbeit hinausgehen würde.

Betrachtet sei nun das Beispiel 3.5.1. In diesem Beispiel sind die beiden Induktionsvariablen *nBasic*, *nDepend* Teil einer Schleife in der sie fortlaufend verändert werden (zur einfacheren Beschreibung lassen wir die Induktionsvariable i der Schleife außen vor).

Beispiel 3.5.1 *Induktionsvariablen*

```
int i, nBasic = 0, nDepend = 0;

  for ( i = 0; i < 16; i++ )
  {
    nBasic++;
    nDepend = nDepend + nBasic * 2;
  }
```

Wie sich leicht erkennen läßt, wird die Variable *nBasic* durch einen konstanten Wert verändert, wohingegen die Variable *nDepend* in Abhängigkeit der Variable *nBasic* modifiziert wird. Da das Erkennen von Induktionsvariablen, deren Wert durch eine Konstante geändert wird, einfacher ist als bei Induktionsvariablen mit komplexen arithmetischen Operationen, bietet sich eine Klassifizierung dieser beiden Variablentypen an.

Definition 3.5.2 *Einfache Induktionsvariablen: Einfache Induktionsvariablen sind Variablen, welche nur durch einen konstanten Wert fortlaufend verändert werden.*

Definition 3.5.3 *Abhängige Induktionsvariablen: Abhängige Induktionsvariablen sind Variablen, welche durch komplexe arithmetische Ausdrücke mit weiteren Variablen, verändert werden.*

In dem angegebenen Beispiel 3.5.1 wird somit die Variable *nBasic* als Basic, und die Variable *nDepend* als abhängige Induktionsvariable klassifiziert.

Zu Beginn des Verfahrens werden alle Variablen innerhalb der Schleife als Induktionsvariablen betrachtet. Dabei wird jeder Induktionsvariable i die lineare Gleichung $i = b * biv + c$ zugeordnet, wobei b und c Konstanten und biv eine einfache Induktionsvariable bezeichnet. Zu Beginn der Analyse sind diese Werte *nil* (leer).
Weiterhin werden Induktionsvariablen anhand ihrer *einfachen Induktionsvariablen* (biv) in Klassen eingeteilt. Dabei werden die Induktionsvariablen der selben Klasse zugeordnet, wenn sie von der gleichen einfachen Induktionsvariable abhängig sind.

Zur Identifikation werden sequentiell die einzelnen Anweisungen innerhalb der Schleife inspiziert. Zu Beginn werden nur die einfachen Induktionsvariable betrachtet, welche die Form $i = i + d$ besitzen, wobei i für die Induktionsvariable und d für eine Konstante steht. Für eine solche einfache Induktionsvariable ergibt sich die lineare Gleichung $i = 1 \times i + 0$. Wird ein solche einfache Induktionsvariable mehrfach innerhalb der Schleife verändert, so wird für jede Änderung eine Kopie der Variable angelegt.

```
  i   = 1

Loop:
  // Gebrauch von i
  i = i + 2
  // Gebrauch von i
  i = i + 4
  // Gebrauch von i
  goto Loop
```

a)

```
  i1  = -3
  i   = 1

Loop:
  // Gebrauch von i
  i1  = i1 + 6
  // Gebrauch von i1
  i   = i + 6;
  // Gebrauch von i
  goto Loop
```

b)

In diesem Beispiel wird i innerhalb der Schleife a) zweimal verändert. Durch das Aufteilen der Variable i in die Variablen i und $i1$ wird erreicht, daß sich jede Variable innerhalb der Schleife b) nur einmal ändert.

Im zweiten Teil der Analyse werden nun die abhängigen Induktionsvariablen ermittelt. Abhängige Induktionvariablen bekommen den Wert einer einfachen oder wiederum einer

abhängigen Induktionsvariablen zugewiesen. Dabei hat eine abhängige Induktionsvariable die Form $j = e \times i$. i bezeichnet dabei eine Induktionsvariable, e kann leer, eine Konstante oder ebenfalls eine Induktionsvariable sein. Für den Fall, daß i eine einfache Induktionsvariable darstellt, ergibt sich die lineare Gleichung für j als $j = e \times (1 \times i + 0) = e \times i$. In dem anderen Fall hat i als abhängige Variable die Form $i = b_1 \times i_1 + c_1$. Somit ergibt sich die lineare Gleichung für j als: $j = (e \times b_1) \times i_1 + e \times c_1$.
Durch fortlaufendes Betrachten der Variablenzugriffe innerhalb der Schleife, werden alle möglichen Induktionsvariablen erkannt und mit ihrem entsprechenden Indexterm versehen.

Beispiel

Betrachtet sei nun das folgende Programmbeispiel im 3-Adreßformat:

```
    t1  = 202
    i   = 1;

  Loop:
    t2  = i > 100
    if t2 goto Exit

    t1  = t1 - 2
    t3  = &A
    t4  = 4 * i
    t5  = t3 + t4
    *t5 = t1

    i   = i + 1
    goto Loop

  Exit:
```

Zuerst wird die einfache Induktionsvariable $t1$ gefunden. Die einzige Zuweisung von $t1$ innerhalb der Schleife ist $t1 = t1 - 2$. Somit ist $t1$ nur von sich selbst abhängig, womit sich die lineare Gleichung von $t1$ zu $t1 = 1 \times t1 + 0$ ergibt. Die Variable i ist die zweite und letzte einfache Induktionsvariable mit der linearen Gleichung $i = 1 \times i + 0$.
Die Variable $t3$ erhält eine Konstante und wird in dieser Schleife nicht verändert. Somit ist ihr Wert von keiner anderen Variablen abhängig. $t4$ hingegen ist von der einfachen Induktionsvariable i abhängig. Seine lineare Gleichung ergibt sich somit zu $t4 = 4 \times (1 \times i + 0) = 4 * i$. Zum Schluß wird die Variable $t5$ ausgewertet. Diese hängt dabei von den beiden Variablen t3 und t4 ab. Somit ergibt sich die linearen Gleichung für $t5$ als $t5 = \&A + i * 4$.

Kapitel 4

Datenflußanalyse

4.1 Einführung

Bei der Durchführung von Optimierungen kommt es zu einer starken Veränderung des ursprünglichen Programmes. Dabei sind Veränderungen nicht beliebig durchführbar, da dies zu fehlerhaften Ergebnissen führen kann. Viele Optimierungen basieren darauf, daß Werte in einer anderen Reihenfolge verändert oder Berechnungen sogar gänzlich entfernt werden. Dabei muß beachtet werden, ob durch die Veränderungen der Instruktionen Werte von Variablen erhalten bleiben, welche im fortlaufenden Programmfluß verwendet werden. Es ist somit notwendig, die Abhängigkeiten von Variablen zu berücksichtigen, um semantisch korrekte Optimierungen durchführen zu können. Zur Ermittlung der Abhängigkeiten von Variablen werden Datenflußanalysen eingesetzt.
Die im Rahmen dieser Diplomarbeit vorgestellte und realisierte Vektorisierung benötigt ebenfalls die Verwendung einer Datenflußanalyse. Da Operationen, welche durch die Optimierung vektorisiert werden sollen, sich hier nur auf den Zugriff von Arrays beschränken, wird ein Spezialfall der Datenflußanalysen benötigt. Dieses Kapitel der Diplomarbeit stellt eine Reihe von Datenflußanalysemethoden vor, welche speziell für die Analyse von Arrayzugriffen entwickelt worden sind.

Eine Array-Datenflußanalyse kommt dabei bei einer Vektorisierung in mehreren Fällen zum Einsatz. Zum einen wird sie dazu benötigt, abhängige Speicherzugriffe auf Arrayelemente zu erkennen, bei denen es nach einer Vektorisierung zu fehlerhaften Ergebnissen kommen würde. Zum anderen kann sie für eine Reihe weiterer, eine Vektorisierung unterstützender, Optimierungen eingesetzt werden.

Im nachfolgenden werden zuerst Grundlagen und Begriffe von Datenflußanalysen definiert und erklärt, die für das weitere Verständnis der vorgestellten Analysen benötigt werden. Danach werden dann eine Reihe von Array-Datenflußanalysen vorgestellt und ihre Vor- und Nachteile beschrieben. Auf dieser Basis wird dann eine Bewertung vorgenommen. Zum Abschluß wird die im Rahmen dieser Diplomarbeit verwendete δ-Array Datenflußanalyse detailliert vorgestellt. Dabei wird zuerst der theoretische Hintergrund dieser Analyse beschrieben und im nachfolgenden auf die Implementierung und die durchgeführten Mo-

difikationen eingegangen.

4.2 Grundlagen von Datenflußanalysen

Dieser Abschnitt gibt einen Überblick über grundlegenden Begriffe von Datenflußanalysen. Dabei wird die Funktionsweise von Datenflußanalysen beschrieben und aufgezeigt, wie ermittelte Abhängigkeiten dargestellt werden können.

4.2.1 Arbeitsweise von Datenflußanalysen

Datenflußanalysen werden im allgemeinen auf Kontrollflußgraphen oder einer speziellen Form dieser Graphen durchgeführt. Um eine einfachere Ermittlung der Abhängigkeiten von Instruktionen innerhalb der Knoten des Kontrollflußgraphen zu erreichen, werden bei der Durchführung der Analyse jeweils nur die Vorgänger- bzw. Nachfolgerknoten jedes einzelnen Knotens betrachtet. Durch mehrmaliges Durchführen dieser Vorgehensweise werden die Abhängigkeiten durch den Graphen propagiert. Sind alle Abhängigkeiten zwischen den Knoten des Graphen bestimmt, tritt keine Veränderung der Ergebnisse an den Knoten auf. In diesem Fall hat sich die Lösung der Abhängigkeitsanalyse "stabilisiert", womit die gesuchte Lösung bestimmt worden ist.

Im Gegensatz zu einer realen Ausführung eines Programmes, kann bei einer Datenflußanalyse nicht bestimmt werden, über welche Kontrollflußwege das Programm ausgeführt wird. Es müssen deshalb zur Bestimmung der Abhängigkeiten immer alle möglichen Kontrollflußwege innerhalb des CFGs betrachtet werden. Man spricht deshalb von einer "statischen" Analyse, da die Dynamik, die Programme bei einer realen Ausführung entwickeln, nicht nachgebildet werden kann.

Beispiel 4.2.1 *Statische Analyse*

```
for ( i = 0; i < 16; i++ )
{
if ( i >= x )
   y = 10;

z = y + 20;
}
```

In dem Beispiel 4.2.1 hängt die Ausführung der Anweisung $y = 10$ von dem Wert der Variable x ab. Je nachdem welchen Wert x hat, kann die Anweisung $y = 10$ nicht, manchmal oder immer ausgeführt werden. Da der Wert von x nicht bestimmt werden kann, muß bei einer statischen Analyse immer von beiden möglichen Kontrollflußwegen ausgegangen werden. Es läßt sich somit leicht die eingeschränkte Aussagekraft von statischen Analysen erkennen.

4.2.2 Grundlegende Begriffe

Das Prinzip von Datenflußanalysen besteht darin, zu untersuchen wie Variablen durch Instruktionen verwendet oder verändert werden. Dazu ist es sinnvoll, die Art und Weise von Variablenzugriffen zu klassifizieren.

Gebrauch Mit *Gebrauch* wird der lesenden Zugriff einer Variable bereichnet.

Definition *Definition* bezeichnet einen schreibenden Zugriff auf eine Variable.

Referenz Allgemein wird der Zugriff auf eine Variable als *Referenz* bezeichnet. Es spielt dabei keine Rolle auf dieser Zugriff lesend oder schreibend erfolgt.

Nach der Unterteilung der Variablenzugriffe in lesende und schreibende Zugriffe, werden Zugriffe auf Variablen weiterhin danach unterteilt, ob nach einem Zugriff auf eine Variable eine bestimmte Eigenschaft dieser Variable gilt oder nicht mehr gilt.
Dabei werden zwei Arten von Referenzen unterschieden:

Erzeugend Man spricht von einer erzeugenden Referenz (*generate*), wenn nach dem Zugriff auf eine Variable eine bestimmte Eigenschaft der Variable gilt.

Vernichtend Man spricht von einer vernichtenden Referenz (*kill*), wenn nach dem Zugriff auf eine Variable eine bestimmte Eigenschaft der Variable nicht mehr gilt.

Beispiel 4.2.2 *Variablenreferenzen*

```
a = 1; // Erzeugende Referenz
b = a; // Erzeugende Referenz
a = 2; // Erzeugende und vernichtende Referenz
```

In dem Beispiel 4.2.2 wird der Variablen a der Wert 1 zugewiesen. Dabei handelt es sich um eine erzeugende Referenz, da die Variable zum ersten Mal initialisiert wird. In dem Ausdruck $b = a$ wird die Variable a gelesen und ihr Wert in die Variable b übertragen. Auf a erfolgt somit ein lesender Zugriff und b wird definiert. Es handelt sich dabei ebenso um eine erzeugenden Referenz. Zum Schluß wird der Wert von a auf 2 geändert. Da a durch diesen Zugriff zum einen seinen ursprünglichen Wert verliert, handelt es sich dabei um eine vernichtende Referenz. Zum anderen wird der Variable a der Wert 2 zugewiesen, womit sich durch die erneute Definition auch eine erzeugende Referenz ergibt.

4.2.3 Darstellung von Abhängigkeiten

Zur Darstellung von Abhängigkeiten bietet sich die Verwendung von Abhängigkeitsrelationen an. Dabei entstehen Abhängigkeiten zwischen zwei Instruktionen durch ihre

Ausführungsreihenfolge, welche durch den Kontrollfluß des Programmes bestimmt sind. Weiterhin treten Abhängigkeiten zwischen zwei Instruktionen bzgl. der gemeinsam genutzten Daten auf.

Definition 4.2.1 *Zwei Instruktionen S_1 und S_2 stehen genau dann zueinander in der Relation $S_1 \triangleleft S_2$, wenn die Instruktion S_1 der Instruktion S_2 in der Ausführungsreihenfolge vorangeht. D.h., es existiert ein Pfad im CFG mit $S_1 \rightsquigarrow S_2$.*

Kontrollflußabhängigkeiten entstehen durch die Verwendung von Kontrollstrukturen wie *If-Then-Else* innerhalb des Programmcodes. Dabei besitzen zwei Instruktionen eine Kontrollflußabhängigkeit, wenn die Ausführung der einen Instruktion von der Ausführung der anderen Instruktion abhängt.

Beispiel 4.2.3 *Kontrollflußabhängigkeiten*

```
for ( i = 0; i < 16; i++ )
{
  if ( i > 10 )
     x = x + 1;
}
```

In dem Beispiel 4.2.3 hängt dabei die Ausführung der Instruktion $x = x + 1$ von dem Ergebnis der Instruktion $i > 10$ ab. Damit ist $x = x+1$ von der Instruktion $i > 10$ abhängig und es besteht eine Kontrollflußabhängigkeiten zwischen diesen beiden Instruktionen.

Bei einer *Datenabhängigkeit* wird die Ausführungsreihenfolge durch den Datenfluß der Variablen eingeschränkt. Zwei Instruktionen sind somit voneinander abhängig, wenn Ergebnisse der ersten Instruktion in der zweiten Instruktion verwendet werden.

Beispiel 4.2.4 *Datenabhängigkeiten*

```
x = x + 1;
y = x + 2;
```

In dem Beispiel 4.2.4 besteht eine Datenabhängigkeit zwischen den Instruktionen $x = x+1$ und $y = x + 2$, da der Wert der Variable x in der Instruktion $y = x + 2$ verwendet wird.

Da die allgemeine Aussage über Datenabhängigkeiten sehr ungenau ist, sollten zu diesem Zweck die einzelnen Fälle möglicher Datenabhängigkeiten betrachtet werden. Diese Fälle ergeben sich durch die Klassifizierung, welche Arten von Zugriffen (Gebrauch/Definition) auf den abhängigen Variablen der betrachteten Instruktionen stattfinden. Anhand der Klassifizierung ergeben sich die vier verschiedenen Fälle *Flow*, *Anti*, *Output* und *Input* von Datenabhängigkeiten.

Definition 4.2.2 *Wenn $S_1 \triangleleft S_2$ gilt und S_1 eine Variable definiert, die in S_2 gebraucht und zwischen S_1 und S_2 nicht redifiniert wird, dann existiert zwischen S_1 und S_2 eine* Datenabhängigkeit (Flow Dependence).

Definition 4.2.3 *Wenn $S_1 \triangleleft S_2$ gilt und S_1 eine Variable gebraucht, die S_2 definiert, ohne daß eine Redifinition der betreffenden Variable zwischen S_1 und S_2 stattfindet, dann besteht zwischen S_1 und S_2 eine* Antiabhängigkeit (Anti Dependence).

Definition 4.2.4 *Wenn $S_1 \triangleleft S_2$ gilt und S_1 und S_2 die gleiche Variable definieren und zwischenzeitlich keine weitere Definition dieser Variablen erfolgt, so besteht eine* Ausgabeabhängigkeit (Output Dependence) *zwischen S_1 und S_2.*

Definition 4.2.5 *Wenn $S_1 \triangleleft S_2$ gilt und S_1 und S_2 die gleiche Variable lesen und zwischenzeitlich keine erneute Definition dieser Variable erfolgt, so besteht eine* Eingabeabhängigkeit (Input Dependence) *zwischen S_1 und S_2.*

Zur Verdeutlichung der einzelnen Fälle kann die Abbildung 4.1 betrachtet werden.

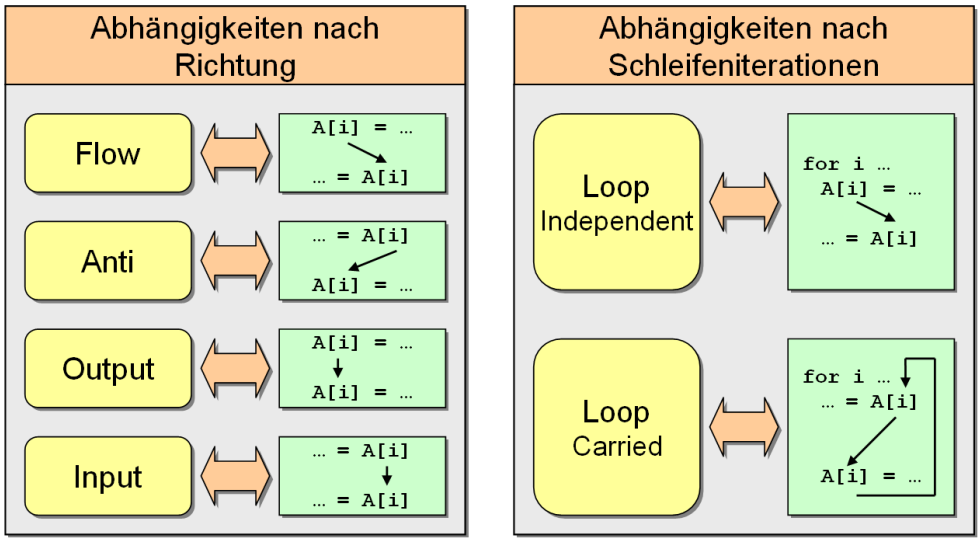


Abbildung 4.1: Typen von Datenabhängigkeiten bei Array-Referenzen

Für die Analyse von Abhängigkeiten auf Array-Referenzen muß weiterhin noch betrachtet werden, ob diese Abhängigkeiten innerhalb des Schleifenkörpers bestehen, oder ob

Abhängigkeiten auch über die Schleifengrenzen hinweg vorhanden sind.
Dabei werden Abhängigkeiten innerhalb des Schleifenkörpers als *schleifenunabhängig (loop independent)* und Abhängigkeiten über die Schleifengrenzen hinweg als *schleifenabhängig (loop carried)* bezeichnet.

4.2.4 Forward-/Backward- und Must-/May-Analysen

Je nach verwendeter Optimierung muß entschiedenen werden, in welche Richtung der Datenfluß propagiert wird und ob gewisse Eigenschaften erfüllt sein können oder müssen.

Forward-/Backward-Analysen geben an, ob der Datenfluß vorwärts oder rückwärts durch den Graphen propagiert wird. Welche Form der Propagierung gewählt wird, hängt von der zu unterstützenden Optimierung ab. So wird z.B. für die Optimierung *RLE (Redundant Load Elimination)* eine Forward-Analyse benötigt, wohingegen für die Optimierung *RSE (Redudant Store Elimination)* eine Backward-Analyse zum Einsatz kommt.

Must-/May-Analysen unterscheiden sich dahingehend, ob die Gültigkeit von Datenflußeigenschaften über alle möglichen Kontrollflußwege hinweg erfüllt sein müssen *(Must)* oder erfüllt sein können *(May)*. Bei einer Programmausführung, welche über verschiendene Wege des Kontrollflußgraphen verlaufen kann, werden einige Instruktionen immer und andere Instruktionen potentiell ausgeführt. Dazu sei das folgende Beispiel 4.2.5 betrachtet.

Beispiel 4.2.5 *Must-/May-Analysef*

```
for ( i = 0; i < 16; i++ )
{
x = 5;

  if ( i > 10 )
     y = 10;

z = 15;
}
```

In diesem Beispiel werden die Variablen x und z über alle möglichen Kontrollflußwege definiert. Da sie im Sinne der Analyse immer gültig sind, werden sie durch die Must-Analyse als solche erkannt. Die Variable z wird nur über einen Weg des Kontrollflußgraphen definiert. Da nicht mit Bestimmtheit entschieden werden kann, ob sie gültig ist oder nicht, wird sie von der Must-Analyse als ungültig eingestuft. Im Gegensatz dazu wird sie bei einer May-Analyse als gültig erkannt, da May-Analysen die potentielle Gültigkeit von Variablen ermitteln.

4.2.5 Werte- und speicherbasierte Abhängigkeiten

Die Bestimmung von Abhängigkeiten zwischen Arrayzugriffen ist durch die statische Analyse sehr eingeschränkt und in vielen Fällen nicht entscheidbar. Dies liegt darin begründet, daß der reale Programmfluß, wie er durch das Ausführen eines Programmes besteht, nicht nachvollzogen werden kann. Um trotzdem eine Lösung von Datenabhängigkeiten zu erhalten, bieten sich die zwei folgenden Vorgehensweisen an:

1. Einschränkung der Analyse auf einfache Fälle.
2. Die Bestimmung einer Approximation anstatt einer exakten Lösung.

Bei einer Approximation der Lösung eines Datenflußproblems entstehen Ergebnisse, welche zu der normalen Lösung (abhängig, nicht abhängig) auch die Unentscheidbarkeit von Abhängigkeiten enthalten können. Da bei diesen Ergebnissen nicht eindeutig entschieden werden kann, ob eine Abhängigkeit vorliegt oder nicht, sind sie nur für eine eingeschränkte Gruppe von Optimierungen verwendbar. Die Vektorisierung von Speicherzugriffen benötigt wie viele andere Optimierungen auch, die Bestimmung einer eindeutigen Lösung. Um dies zu erreichen, muß dazu eine Einschränkung der Problemklasse bzgl. der Analyse vorgenommen werden.

Eine Möglichkeit zur Reduzierung der Problemklasse besteht darin, anstatt *wertebasierter* Abhängigkeiten *speicherbasierte* Abhängigkeiten zu ermitteln. Dabei wird nicht der Datenfluß zwischen zwei abhängigen Instruktionen betrachtet sondern es wird ermittelt, ob zwei unterschiedliche Instruktionen den gleichen Speicherbereich referenzieren.

4.2.6 Datenflußverbände

Zur Durchführung einer statischen Analyse bedarf es weiterhin einer formalen Darstellung der Ausprägung des zu untersuchenden Programmes. Dabei werden mit der "Ausprägung eines Programmes" im folgenden die Werte von Variablen bezeichnet, die durch die *Datenflußverbände* modelliert werden.

Definition 4.2.6 *Ein Verband L besteht aus einer Menge von Elementen, der Trägermenge, und zwei Operatoren $\sqcap$ (meet) und $\sqcup$ (join) mit folgenden Eigenschaften:*

1. *Abgeschlossenheit*
   $\forall x, y \in L : \exists z, w \in L : x \sqcap y = z \wedge x \sqcup y = w$
2. *Kommutativität*
   $\forall x, y \in L : x \sqcap y = y \sqcap x \wedge x \sqcup y = y \sqcup x$
3. *Assoziativität*
   $\forall x, y, z \in L : (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z) \wedge (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$

4. *Supremum*
   $\exists \bot \in L : \forall x \in L : x \sqcap \bot = \bot$
   Das Element $\bot$ *wird als* BOTTOM *bezeichnet.*

5. *Infimum*
   $\exists \top \in L : \forall x \in L : x \sqcup \top = \top$
   Das Element $\top$ *wird als* TOP *bezeichnet.*

Definition 4.2.7 *Ein Verband L wird als* DISTRIBUTIVER VERBAND *bezeichnet, falls folgendes gilt:* $\forall x, y, z \in L : (x \sqcap y) \sqcup z = (x \sqcup z) \sqcap (y \sqcup z)$

Definition 4.2.8 *Die auf einem Verband L indizierte Partielle Ordnung* $(L, \sqsubseteq)$ *ist durch* $\forall x, y \in L : x \sqsubseteq y \Longleftrightarrow x \sqcap y = x$ *definiert. Ebenso ist diese Definition auch mit dem* $\sqcup$ *Operator möglich. Die Operatoren* $\sqsubset$, $\sqsupset$ *und* $\sqsupseteq$ *sind analog definiert.*

Definition 4.2.9 *Die Höhe eines Verbandes L ist definiert als:*
$height(L) = max\{n | \exists x_1, x_2, \ldots x_n : \bot \sqsubset x_1 \sqsubset x_2 \sqsubset \ldots \sqsubset x_n \sqsubset \top\}$

4.2.7 Transferfunktionen

Durch Datenflußverbände werden Eigenschaften von Datenabhängigkeiten modelliert. Da sich diese Eigenschaften beim Ausführen von Programminstruktionen verändern, wird ein Formalismus benötigt, welcher in der Lage ist, diese Veränderung auf Basis der Datenflußverbände durchzuführen.
Zu diesem Zweck werden *Transferfunktionen* verwendet, welche die Veränderung durch Instruktionen modellieren. Transferfunktionen definieren somit, wie die Ausprägung eines Programmes durch die entsprechenden Instruktionen verändert wird.

Definition 4.2.10 *Transferfunktionen sind Funktionen* $f : L \rightarrow L$, *die der Kantenmenge oder der Kantenmenge des Kontrollflußgraphen* $CFG = (N, E, s, e)$ *zugeordnet werden. Somit gilt:* $tf : M \rightarrow (L \rightarrow L)$, *mit* $M = \{N, E\}$.

Die konkrete Definition von Transferfunktionen hängt dabei von den folgenden drei Eigenschaften ab:

- Die Art der zu beschreibenen Instruktionen.
- Von den beteiligten Operatoren.

- Der Art der zu untersuchenden Datenflußeigenschaft.

Werden bei der Analyse nur Instruktionen selber betrachtet, so ist es ausreichend festzustellen, welche Operanden verwendet werden. Die Art und Weise, wie Operanden verändert werden, spielt in diesem Fall keine Rolle. Somit ist es z.B. nicht von Bedeutung, ob Werte von Operanden durch arithmetische Ausdrücke oder durch konstante Werte verändern.

Beispiel 4.2.6 *Operandenzugriffe*

```
a = 1;
b = 2;
c = a + b;
```

In dem Beispiel 4.2.6 werden die Variablen a, b, c verändert. Die Variable c erhält dabei das Ergebnis des Ausdrucks $a + b$. Werden zur Beschreibung der Transferfunktionen nur Instruktionen betrachtet, so spielt das Ergebnis der arithmetischen Operation $a + b$ keine Rolle. Vielmehr ist es von Bedeutung, welche Veränderung des Programmstatus der Zugriff auf eine Variable hat. Damit ergibt sich eine weitaus einfachere Klasse von Transferfunktionen, da nur Speicherzugriffe selber und nicht die Art der Speicherzugriffe modelliert werden muß.

4.2.8 Distanzvektoren

Zur Darstellung von Array-Datenabhängigkeiten sind eine Reihe von Darstellungsformen entwickelt worden. Wichtige Darstellungsformen in diesem Bereich stellen dabei die *Distanzvektoren*, *Richtungsvektoren* und *Datenflußabhängigkeitspaare* dar.

Distanzvektoren geben die Iterationsdistanzen innerhalb einer Schleife an. Dabei bezeichnen Iterationsdistanzen die Differenzen einer Induktionsvariable zwischen Referenzen verschiedener Variablen. Richtungsvektoren stellen eine Vereinfachung der Distanzvektoren dar, da sie nur das Vorzeichen der Distanz der Datenabhängigkeit wiederspiegeln. Die Art des Vorzeichens gibt dabei an, ob die Induktionsvariablen zwischen der Definition und dem Gebrauch von Variablen vergrößert oder verkleinert werden. Die exakteste Darstellung von Array-Datenabhängigkeiten wird mit den Datenflußabhängigkeitspaaren erreicht. Dabei fassen sie für eine Schleife die Induktionsvariablen einer Definition und eines abhängigen Gebrauchs zusammen.

Das aus [BF99] und [MAL92] stammende Beispiel soll nun die einzelnen Darstellungsfor-

men verdeutlichen.

Beispiel 4.2.7 *Datenabhängigkeiten innerhalb von Schleifen*

```
for ( i = 0; i <= L; i++ )
{
  for ( j = 0; j <= M; j++ )
  {
    for ( k = 0; k <= N; k++ )
    {
      A[j] = A[j] + ...
    }
  }
}
```

Datenfluß-Richtungsvektoren

$$\begin{bmatrix} i \\ j \\ k \end{bmatrix} : \left\{ \begin{bmatrix} 0 \\ 0 \\ + \end{bmatrix}, \begin{bmatrix} + \\ 0 \\ - \end{bmatrix} \right\}$$

Datenfluß-Distanzvektoren

$$\begin{bmatrix} i \\ j \\ k \end{bmatrix} : \left\{ \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ -N \end{bmatrix} \right\}$$

Datenflußabhängigkeitspaare

$$\left\{ \left(\begin{bmatrix} i \\ j \\ k-1 \end{bmatrix}, \begin{bmatrix} i \\ j \\ k \end{bmatrix} \right) \middle| \begin{array}{l} 0 \leq i \leq L \\ 0 \leq j \leq M \\ 0 \leq k \leq N \end{array} \right\} \cup \left\{ \left(\begin{bmatrix} i-1 \\ j \\ N \end{bmatrix}, \begin{bmatrix} i \\ j \\ 0 \end{bmatrix} \right) \middle| \begin{array}{l} 1 \leq i \leq L \\ 0 \leq j \leq M \end{array} \right\}$$

In dem Beispiel 4.2.7 enthält der innere Schleifenkörper jeweils eine lesende und eine schreibende Array-Referenz. Beim Übergang der inneren Schleife in eine neue Iteration wird der in der vorherigen Iteration geschriebene Wert von $A[j]$ gelesen. Somit ist die Datenflußabhängigkeit bzgl. k von der niedrigeren zur höheren Iteration gerichtet und hat die Distanz 1. Dies wird durch den Richtungsvektor als $(0, 0, +)$ und durch den Distanzvektorvektor (0, 0, 1) ausgedrückt. Die zweite Datenabhängigkeit ergibt sich durch das Schleifenende von k. Wird das Schleifenende von k erreicht, so springt der Wert von N auf Null zurück. In dem Fall, daß auch j das Schleifenende erreicht, wird der Zähler von i um eins erhöht. Da zuvor von j geschriebene Elemente wieder überschrieben werden, besteht die zweite Datenabhängigkeit zwischen den Arrayzugriffen. Durch die Erhöhung von i um Eins, ist die Abhängigkeit zu i vorwärtsgerichtet. Da k von N auf Null zurückspringt entsteht eine rückwärtsgerichtete Abhängigkeit. Somit ergibt sich für den Richtungsvektor

das Tupel $(+, 0, -)$ und für den Distanzvektor das Tupel $(1, 0, -N)$. Die exakte Darstellung gibt dabei in allgemeiner Form die Beziehung zwischen den Induktionsvariablen bei der Definition und beim Gebrauch an.

4.3 Überblick über Datenflußanalysemethoden

Dieses Kapitel widmet sich dem Überblick über eine Reihe von Datenflußanalysen für Arrayreferenzen. Es wird dabei kurz auf die Funktionsweise der verschiedenen Verfahren eingegangen und ihre Vor- und Nachteile erläutert.

4.3.1 δ-Array Datenflußanalyse

Die von Duesterwald, Gupta und Soffa [DGS93] vorgestellte δ-Array Datenflußanalyse dient der Analyse von Abhängigkeiten zwischen den Zugriffen von Array-Elementen. Sie unterstützt eine Vielzahl von Optimierungen wie Load-/Store-Optimierungen und Loop-Unrolling.
Die δ-Array Datenflußanalyse setzt für die Analyse gewisse Bedingungen voraus. Dazu gehört, daß zu analysierende Schleifen die Single-Entry/Single-Exit Eigenschaft (siehe Kapitel 3) erfüllen müssen. Weiterhin müssen Indexfunktionen in affiner Form (siehe Kapitel 4.4) vorliegen.

Trotz ihrer Einschränkung auf affine Indexterme und strukturierte Schleifen besitzt die δ-Array Datenflußanalyse eine Reihe von Vorteilen. Ein großer Vorteil besteht in der Parametrisierbarkeit des Verfahrens. So ist die δ-Array Datenflußanalyse durch geringfügige Änderungen in der Lage, *Must*- und *May*-Analysen sowie *Forward*- und *Backward*-Analysen durchzuführen. Weiterhin ist sie nicht nur auf eindimensionale Arrays beschränkt, sondern läßt sich ebenfalls auf multidimensionale Arrays verallgemeinern.

Der Kern der Analyse besteht in der Berechnung von *Iterationsdistanzen* innerhalb der Knoten des *Schleifenkontrollflußgraphen*. Dabei bezeichnen die Iterationsdistanzen die Anzahl von Iterationen, in denen eine Lösung an einem Knoten des Schleifenkontrollflußgraphen seine Gültigkeit behält. Der Schleifenkontrollflußgraph ist ein spezieller Graph, welcher nur für die Analyse benötigte Instruktionen enthält. Weitere Informationen über *Schleifenkontrollflußgraphen* können dem Kapitel 4.4.1 dieser Diplomarbeit entnommen werden.

Zur Bestimmung des Datenflusses geht die δ-Array Datenflußanalyse in ähnlicher Weise vor, wie iterative Datenflußverfahren auf skalaren Variablen. Dazu wird jeder Instruktion innerhalb des Schleifenkontrollflußgraphen eine *Transferfunktion* zugeordnet, die das Verhalten der Instruktion statisch modelliert. Nach der Bestimmung der *Transferfunktion* wird ein iteratives Lösungsverfahren eingesetzt, welches die Lösung des durch die Transferfunktionen entstehenden Datenflußgleichungssystems ermittelt.

Vorteile des Verfahrens
Die δ-Array Datenflußanalyse unterstützt eine Vielzahl von Optimierungen. Dies ist hauptsächlich in der Parametrisierbarkeit des Verfahrens begründet. Dadurch ist es möglich, durch geringe Änderungen unterschiedliche Arten von Analysen durchzuführen. Durch seine geringe Komplexität ist es nicht nur relativ einfach zu implementieren sondern auch an bestimmte Gegebenheiten leicht anzupassen. Weiterhin besitzt die δ-Array Datenflußanalyse eine gute Laufzeit und einen geringen Ressourcenverbrauch.

Nachteile des Verfahrens
Ein Nachteil des Verfahrens besteht in der ausschließlichen Verwendung von affinen Indextermen (wir werden in Kapitel 4.4 sehen, daß sich dieses durch leichte Änderung des Verfahrens vermeiden läßt). Weiterhin müssen zu analysierende Schleifen die Single-Entry/Single-Exit Eigenschaft erfüllen. Erschwerend kommt hinzu, daß die Möglichkeit einer Schleifenanalyse stark von der Komplexität der enthaltenen Indexterme abhängt. Ab einem gewissen Grad ist eine Analyse nicht mehr möglich.

4.3.2 Stretched Looped

Das *Stretched Looped* Verfahren (Bodik, Gupa, [BG96]) besitzt im Gegensatz zur δ-Array Datenflußanalyse eine bessere Approximationsgüte. Dies wird dadurch erreicht, indem Instruktionen nicht nur auf Iterationsebene, sondern auch auf Instruktionsebene analysiert werden.

Neben der Voraussetzung von affinen Indextermen darf pro Instruktion innerhalb der zu analysierenden Schleife nur ein Arrayzugriff erfolgen. Diese weitere Einschränkung läßt sich aber mittels Einfügen von temporären Variablen umgehen.
Die Berechnung des Datenflusses geschieht letztendlich mittels des δ-Verfahrens.

Das Prinzip des Verfahrens besteht darin, alle Array-Referenzen die das gleiche Array-Element betreffen, gemeinsam zu betrachten (Instruktionsebene). Diese gemeinsam betrachteten Array-Referenzen werden in *Kongruenzklassen* zusammengefaßt. Durch diese Zusammenfassung der Array-Referenzen ist es möglich, die Abhängigkeiten auf der Ebene der einzelnen Knoten (Instruktionen) zu betrachten. Alle Referenzen die außerhalb dieser Klasse liegen können für die Analyse des einzelnen Knotens ignoriert werden, da sie für die Bestimmung der Abhängigkeiten keine Rolle spielen. Dadurch ist es möglich, mehrere aufeinanderfolgende Iterationen, die die gesamte Lebensdauer eines Array-Elementes umfassen, gleichzeitig zu analysieren.

Ebenso wie in der δ-Array Datenflußanalyse ist es möglich, das Verfahren zu parametrisieren. Damit ergibt sich ebenfalls die Möglichkeit, durch geringe Änderungen des Verfahrens Vorwärts-/Rückwärts und Must-/May-Probleme zu analysieren.

Vorteile des Verfahrens
Das *Stretched Looped* Verfahren bietet gegenüber dem δ-Array Verahren eine höhere Präzision. Dadurch, daß die δ-Array Datenflußanalyse für die Analyse mitverwendet wird, kann eine Implementierung des Verfahrens, sofern eine Implementierung der δ-Array Datenflußanalyse bereits vorhanden ist, schneller erfolgen.

Nachteile des Verfahrens
Ebenso wie die δ-Array Datenflußanalyse benötigt das *Stretched Looped* Verfahren affine Indexterme für die Analyse. Schleifen mit nicht affinen Indextermen können somit nicht analysisert werden.
Der Aufwand der Implementierung liegt dabei beträchtlich über dem des δ-Array Verfahrens. Es muß somit abgeschätzt werden, ob ein solcher Aufwand die zu erwartenden Ergebnisse rechtfertigt.
Ein weiteres Problem stellt die Funktion *it* dar, welche eine Segmentierung der verwendeten G (generate) und K (kill) Mengen durchführt. Die Implementierung dieser Funktion ist nicht fest vorgegeben. Somit hängt die Güte des Analyseverfahrens stark von der Implementierung dieser Funktion ab.

4.3.3 Lazy Verfahren

Das *Lazy* Verfahren (Maslov [M93]) ist in der Lage für affine Indexterme die Datenflußabhängigkeiten genau zu berechnen und für nicht affine Indexterme eine Approximation zu liefern. Damit entfällt im Gegensatz zu dem δ-Array Verfahren und dem Stretched-Loop Verfahren die Begrenzung auf affine Indexterme. Ebenso wie das Stretched-Loop Verfahren arbeitet das Lazy Verfahren auf Instruktionsebene, um eine höhere Güte der Analyse zu erhalten.

Die Analyse arbeitet dabei nach folgendem Prinzip. Ausgehend von lesenden Instruktionen wird der naheliegende Instruktionsraum nach schreibenden Instruktionen hin untersucht. Diese Untersuchung wird dann im weiteren Verlauf der Analyse auf größere Distanzen ausgedehnt. Damit wird zum einem eine höhere Präzision erreicht und zum anderen die Effizienz des Verfahrens gesteigert.

Für die Analyse von affinen Indexterme existieren keine Einschränkungen bzgl. der Kenntnis von Schleifengrenzen. Weiterhin müssen Schleifen nicht in der Single-Entry/Single-Exit Form vorliegen. Das Approximationsverfahren, welches auf nicht affinen Indextermen arbeitet, setzt dagegen strukturierte Schleifen und eine genaue Kenntnis über den Iterationsbereich der Schleife voraus. Eine Analyse von *While bzw. Do-While* Schleifen oder Schleifen mit *Goto* und *Break* Anweisungen ist somit nicht möglich.

Vorteile des Verfahrens
Der große Vorteil des *Lazy* Verfahrens besteht in seiner genauen Analyse von affinen Indextermen und der Möglichkeit auch nicht affine Indexterme verarbeiten zu können.

Nachteile des Verfahrens
Als Nachteil sei vor allem die hohe Komplexität des Verfahrens zu nennen. Dies spiegelt sich nicht nur in einer sehr aufwendigen Implementierung wieder, sondern resultiert unter Umständen in einer sehr langen Laufzeit der Analyse. Weiterhin ist das Verfahren nicht parametrisierbar, so daß nur Optimierungen unterstützt werden, die genau diese Datenflußanalysemethode benötigen. Auch ist das Verfahren nicht geeignet, *Anti-*, *Input-* oder *Output-* Abhängigkeiten zu bestimmen, was zu einer weiteren Einschränkung bzgl. der

unterstützten Optimierungen führt.

4.3.4 DSA Verfahren

Das *DSA* Verfahren (Dynamic Single Assignment) von Rau [R91] gestattet ebenfalls die Analyse von affinen und nicht affinen Indextermen. Weiterhin ist das Verfahren parametrisierbar und kann auf beliebigen Kontrollflußgraphen arbeiten, womit eine Beschränkung auf Single-Entry/Single-Exit entfällt.

Das Verfahren unterscheidet sich in der Hinsicht von den vorgestellten Verfahren, daß zur Analyse eine spezielle angepaßte IR verwendet wird. Diese IR ist daraufhin ausgelegt, eine genaue Darstellung von Abhängigkeiten zwischen Instruktionen zu ermöglichen.
Bei der Anwendung des Verfahrens sind allerdings zwei Voraussetzungen zu beachten. Die verwendeten Indexterme sollten in einer affinen Form vorliegen, da nicht affine Ausdrücke zu einer Verschlechterung der Ergebnisse führen oder eine Analyse gegebenenfalls unmöglich machen. Weiterhin müssen alle Indexterme von der gleichen Indexvariable abhängig sein, welche sich nur um einen konstanten Wert ändern darf.

Vorteile des Verfahrens
Ein großer Vorteil des *DSA* Verfahrens ist seine Parametrisierbarkeit und Möglichkeit sowohl affine als auch nicht affine Ausdrücke in Indextermen zu analysieren. Weiterhin ist das *DSA* Verfahren in der Lage sowohl wertebasierte als auch speicherbasierte Abhängigkeitsanalysen durchzuführen.

Nachteile des Verfahrens
Nachteilig für das *DSA* Verfahren ist seine geringe Verbreitung und die Verwendung einer eigenen IR Darstellung. Dadurch existieren kaum Implementierungen von Standardoptimierungen, die auf dieser IR arbeiten. Schwerwiegender ist jedoch die Tatsache, daß das Verfahren unter Umständen nicht terminiert. Ein weiterer Nachteil liegt in der schlechten Analyse von nicht affinen Indextermen.
Als Ursache für die geringe Verbreitung dieses Verfahrens kann zu den bereits erwähnten Punkten die schlechte Dokumentation genannt werden. Ob in Zukunft weitere detaillierte Arbeiten zu dieser Analyse veröffentlicht werden, ist zu diesem Zeitpunkt mehr als fraglich.

4.3.5 Bewertung

Eine eindeutige Bevorzugung eines der Verfahren ist nicht möglich, da alle Verfahren Vor- und Nachteile haben.
Im Rahmen dieser Diplomarbeit fiel die Wahl auf die δ-Array Datenflußanalyse. Einer der Gründe dafür ist die geringe Komplexität des Verfahrens, so daß eine Implementierung im Zeitrahmen einer Diplomarbeit möglich war. Ein weiterer Grund für die Wahl der δ-Array Datenflußanalyse bestand darin, daß dieses recht einfache Verfahren leicht anpaßbar und somit für eine Analyse von Speicherdistanzen zur Vektorisierung geeignet ist.

4.4 δ-Array Datenflußanalyse

Der folgende Abschnitt dieser Diplomarbeit schildert nun im Detail die Funktionsweise der δ-Array Datenflußanalyse. Dabei wird im ersten Teil der theoretische Hintergrund des Verfahrens und seine Funktionsweise erläutert. Der zweite Teil gibt einen Überblick über die Veränderungen und Erweiterungen, welche bei der Implementierung des Verfahrens durchgeführt worden sind.

4.4.1 Schleifenkontrollflußgraphen

Die δ-Array Datenflußanalyse benötigt zur Analyse einen *Schleifenkontrollflußgraphen* (LCFG), der gegenüber normalen Kontrollflußgraphen folgende Erweiterungen und Eigenschaften besitzt:

Definition 4.4.1 *Ein* SCHLEIFENKONTROLLFLUSSGRAPH *(Loop Control Flow Graph, LCFG) FG=(N, E) mit der Knotenmenge* N *und Kantenmenge* E *repräsentiert den Kontrollfluß innerhalb eines Schleifenkörpers. Die Knoten N bezeichnen dabei Instruktionen, wobei jeder Knoten N nur jeweils eine Instruktion enthalten darf. Die Kantenmenge E stellt die möglichen Kontrollflußübergänge im Graphen dar. Im Gegensatz zu normalen Kontrollflußgraphen besitzen Schleifenkontrollflußgraphen einen ausgezeichneten Knoten* EXIT*, welcher den expliziten Übergang zur nächsten Iteration modelliert. Dabei führen alle Kanten, die den Schleifenkörper verlassen, zum* EXIT *Knoten. Alle ausgehenden Kanten des* EXIT *Knotens führen zum Beginn der Schleife.*

Beispiel 4.4.1 *Schleifenkontrollflußgraph (LCFG)*

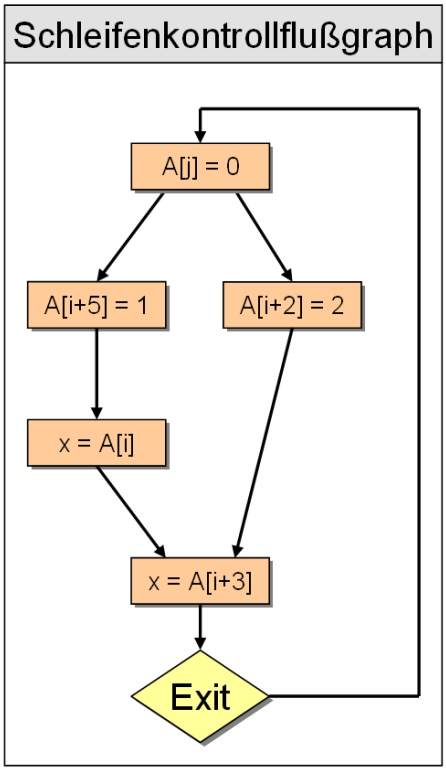


Abbildung 4.2: Schleifenkontrollflußgraph

Zur Durchführung einer δ-Array Datenflußanalyse muß somit der vorhandene CFG in einen für die Analyse verwendbaren LCFG transformiert werden. Eine Vorgehensweise zur Erstellung eines Schleifenkontrollflußgraphen aus einem normalen Kontrollflußgraphen wird im späteren Verlauf dieser Diplomarbeit in Kapitel 5.4 ausführlich beschrieben.

4.4.2 Datenflußverband

Das Ergebnis einer δ-Array Datenflußanalyse gibt die maximale Anzahl von Iterationen an, in der ein Wert eines Arrayelementes seine Gültigkeit behält. Zur Darstellung dieser als *Iterationsdistanzen* bezeichneten Werte, wird bei der δ-Array Datenflußanalyse ein linearer, streng monoton steigender Verband L mit $L = \{\bot, 0, 1, 2, \ldots, \top = UB - 1\}$ verwendet. Der Wert UB gibt dabei die maximaler Anzahl von Iterationen einer Schleife an. Mit dem *Infimum* Wert $\bot$ wird angegeben, daß ein Wert seine Gültigkeit sofort verliert.

Supremum $\top$ gibt dagegen an, daß ein Wert seine Gültigkeit nie verliert.

Beispiel 4.4.2 *Iterationsdistanzen*

```
void IterationDistance()
{
int A[16];
int i;

  for ( i = 0; i < 10; i++ )
  {
    A[i] = 1;
    A[i + 5] = 2;
  }
}
```

In dem Beispiel 4.4.2 schreiben die beiden Anweisungen $A[i] = 1$ und $A[i + 5] = 2$ in den gleichen Speicherbereich eines Arrays. Die Iterationsdistanz zwischen $A[i] = 1$ und $A[i + 5] = 2$ beträgt 5, da nach fünf Iterationen der geschriebene Wert der Anweisung $A[i + 5] = 2$ durch den Wert der Anweisung $A[i] = 1$ überschrieben und somit vernichtet wird. Im Gegensatz dazu beträgt der umgekehrte Abstand $\top$, da die Anweisung $A[i+5] = 2$ niemals einen Wert der Anweisung $A[i] = 1$ überschreibt.

4.4.3 Verwendete Operatoren

Zur Durchführung der δ-Array Datenflußanalyse werden die drei Operatoren Mininimum ($\wedge$), Maximum ($\vee$) und demr spezielle Inkrement-Operator (++) verwendet.

Min Funktion

$$\wedge(x,y) = \begin{cases} \bot & , falls\ y\ =\ \bot\ oder\ y\ =\ \bot \\ x & , falls\ y\ =\ \top \\ y & , falls\ x\ =\ \top \\ min(x,y) & , sonst \end{cases}$$

Max Funktion

$$\vee(x,y) = \begin{cases} \top & , falls\ y\ =\ \top\ oder\ y\ =\ \top \\ x & , falls\ y\ =\ \bot \\ y & , falls\ x\ =\ \bot \\ max(x,y) & , sonst \end{cases}$$

Die *Min* und *Max* Funktionen modellieren dabei das Aufeinander treffen verschiedener Kontrollflußpfade. Sie realisieren damit die entsprechenden Operatoren $\wedge$ und $\vee$ des Verbandes L.

Inkrement-Operator

$$x++ = \begin{cases} \top & , falls\ x = \top \\ \bot & , falls\ x = \bot \\ x+1 & , sonst \end{cases}$$

Der Inkrement-Operator stellt eine Erweiterung der Operatoren des Verbandes L dar. Mit diesem Operator wird der Übergang von einer Iteration der Schleife zur nächsten vollzogen.

4.4.4 Transferfunktionen

Die im folgenden verwendeten *Transferfunktionen* modellieren das Verhalten von Instruktionen, wie sie für die Durchführung der δ-Array Datenflußanalyse benötigt werden. Dazu müssen die Instruktionen dahingehend unterschieden werden, ob sie die Gültigkeit einer Datenflußeigenschaft erzeugen oder vernichten. Dementsprechend wird eine Klassifizierung der Transferfunktionen durchgeführt. Diese Klassifizierung richtet sich danach, ob eine Transferfunktion eine erzeugende Eigenschaft modelliert, ob sie eine Eigenschaft erhält oder ob sie zur Modellierung des Iterationsübergangs verwendet wird. Damit ergeben sich die drei Arten der in der δ-Array Datenflußanalyse verwendeten Transferfunktionen:

- **Erzeugungsfunktion**
  Erzeugungsfunktionen modellieren den Fall einer erzeugenden Referenz in einem Knoten des Schleifenkontrollflußgraphen. Somit behält eine Eigenschaft ihre anfängliche Gültigkeit mit der Iterationsdistanz Null.
- **Erhaltungsfunktion**
  Erhaltungsfunktionen dienen der Anpassung der Iterationsdistanzen innerhalb des Kontrollflusses. Sie können die Iterationsdistanzen unverändert lassen, diese verringern oder sogar vollständig zurücknehmen.
- **Exit-Funktion**
  Die Exit-Funktion erhöht die Iterationsdistanzen eintreffender gültiger Eigenschaften. Mit dieser Funktion werden die Übergänge in eine nächste Iteration der Schleife modelliert.

Im folgenen wird nun der Aufbau der Transferfunktionen im Detail beschrieben.

Erzeugungsfunktion

Für den Fall einer erzeugenden Referenz ist die folgende Transferfunktion f_n^d folgendermaßen definiert:

$$f_n^d(x) = max\{x, 0\}$$

Dabei bezeichnet n einen Knoten innerhalb des LCFG, welcher die Referenz eines anderen Knotens d beeinflußt. Durch $max\{x, 0\}$ wird sichergestellt, daß der Datenfluß, welcher den Knoten d passiert, mindestens den Wert 0 erhält.

Erhaltungsfunktion

Durchläuft der Kontrollfluß einen Knoten im Kontrollflußgraphen, so ist die maximale neue Iterationsdistanz zu bestimmen. Dies hängt damit zusammen, daß Instanzen innerhalb des aktuell betrachteten Knotens, vorherige Definitionen vernichten können. Die für die Bestimmung der neuen Iterationsdistanzen verwendete *Erhaltungsfunktion* ist folgendermaßen definiert:

$$f_n^d(x) = min\{x, p_n^d\}$$

Zur Bestimmung der Konstante p_n^d sind einige Fallunterscheidungen zu treffen. Das d in der Konstante p_n^d bezeichnet die Definition $d = X[idx_1(i)]$, die den Knoten n passiert. Mit X wird ein Array bezeichnet, welches durch d referenziert wird. Die Indexfunktion $idx_1(i)$ stellt eine affine von der Induktionsvariablen i abhängige Funktion dar.
Es bleibt nun zu untersuchen, wieviele der vorherigen Instanzen erhalten bleiben. Somit ergeben sich folgende Fallunterscheidungen zu Bestimmung der Konstante p_n^d:

1. Zwischen dem aktuellen Knoten n und dem Knoten der Definition d besteht keine Abhängigkeit. Dieses trifft zu, wenn einer der folgenden beiden Punkte erfüllt ist.

   (a) Der Knoten n enthält keine Definition des Arrays X. Es wird entweder nur ein lesender Zugriff auf das Array X durchgeführt, oder es erfolgt überhaupt kein Zugriff.

   (b) Der Knoten n enthält eine Definition eines Arrays Y, wobei gilt: $Y \neq X$.

   Der Wert der Konstante p_n^d ist in diesem Fall $p_n^d = \top$.

2. In dem anderen Fall enthält der Knoten n eine Definition $d' = X[idx_2(i)]$, welche eine vorherige Instanz einer Definition $d = X[idx_1(i)]$ vernichtet. Somit muß die maximale Anzahl der Iterationen bestimmt werden, bis die Definition d' die vorherige Definition d vernichtet.

   Vom Knoten n aus betrachtet, haben die Instanzen von $d = X[idx_1(i)]$ die Form $d = X[idx_1(i-\delta)]$. Dabei gibt δ die Anzahl der Iterationen an, welche die Definition $d = X[idx_1(i)]$ von der Definition $d' = X[idx_2(i)]$ trennt. Weiterhin muß betrachtet werden, ob das Erreichen des Knoten n von d aus über die Iterationsgrenze der Schleife hinweg geschieht oder nicht. Zu diesem Zweck wird zusätzlich die Konstante pr verwendet, welche folgendermaßen definiert ist:

   $$pr(d,n) = \begin{cases} 0 & , falls\ d\ in\ in\ einem\ Vorgänger\ von\ n\ enthalten\ ist \\ 1 & , sonst \end{cases}$$

   Die Konstante pr dient dazu, die Anzahl der Iterationsdistanzen über die Schleifengrenze hinweg zu korrigieren.

   Mit diesen Voraussetzungen kann die Konstante p_n^d formal beschrieben werden:

$$p_n^d = max\{\delta | \forall i \in [1, \ldots, UB], \forall \delta, pr(d,n) \leq \delta < UB : idx_2(i) \neq idx_1(i-\delta)\}$$

Die Formel drückt aus, daß das maximale δ gesucht wird, für das die Definitionen d und d' nicht miteinander im Konflikt stehen. Um die Iterationsdistanzen praktisch bestimmen zu können, bedarf es einer Umformulierung der Formel, da die Iterationsdistanz aus $idx_1(i) \neq idx_2(i)$ nicht direkt hervorgeht.
Zu diesem Zweck werden die durch $idx_1(i)$ und $idx_2(i)$ dargestellten affinen Ausdrücke in ihre Bestandteile zerlegt. Man erhält somit die Ausdrücke:

$$idx_1(i) = a_1 \times i + b_1$$

und

$$idx_2(i) = a_2 \times i + b_2$$

Zur Berechnung der Distanzen, werden die beiden Gleichungen nach Null umgeformt. Man erhält damit:

$$idx_1(i - k(i)) - idx_2(i) = 0 \iff$$
$$a_1 \times (i - k(i)) + b_1 - a_2 \times i + b_2 = 0$$

Dabei gibt $k(i)$ die Iterationsdistanz für ein i an, bei der zwei Referenzen das gleiche Arrayelement adressieren. Um den Wert von $k(i)$ zu erhalten, wird die Formel nach $k(i)$ umgeformt:

$$k(i) = \frac{a_1 - a_2}{a_1} \times \frac{b_1 - b_2}{a_1}$$

Eingesetzt in die Formel zur Berechnung von p_n^d ergibt sich nun:

$$p_n^d = max\{\delta | \forall i \in [1, \ldots, UB], \forall \delta, pr(d,n) \leq \delta < UB : \delta \neq k(i)$$

Exit-Funktion
Beim Durchlaufen des *Exit-Knotens* geht der Kontrollfluß der Schleife in eine nächste Iteration über. Dabei wird der Wert der Schleifenindexvariable um eins erhöht. Um zu erreichen, daß ungültige Lösungen stets erhalten bleiben und das über alle Iterationen gültige Lösungen nicht weiter wachsen, wird der in Kapitel 4.4.3 beschriebene *Inkrement-Operator* verwendet. Die *Exit-Funktion* ist somit folgendermaßen definiert:

$$f_{exit}^d(x) = x++$$

4.4.5 Iteratives Lösungsverfahren

Nachdem für jeden Knoten der Schleife die entsprechende Transferfunktion bestimmt worden ist, muß im letzten Schritt der Analyse der Datenfluß ermittelt werden. Zu diesem Zweck wird ein *Datenfluß-Gleichungssystem* aufgestellt, das den Datenfluß innerhalb der Schleife modeliert.
Dazu werden für alle Knoten n der Schleife die zwei Vektoren $IN[n] = \{x_1, \ldots, x_m\}$ und $OUT[n] = \{y_1, \ldots, y_m\}$ gebildet. Der Wert von m bezeichnet die Anzahl der Knoten innerhalb des Schleifenkontrollflußgraphen. Die einzelnen Werte x_i und y_i geben die maximale Iterationsdistanz zwischen den Knoten n und i jeweils beim Erreichen und beim Verlassen des Knoten n an.
Somit bezeichnen $IN[n,d] = x_d$ bzw. $OUT[n,d] = y_d$ die maximalen Iterationsdistanzen x_d und y_d zwischen den beiden Knoten n und d jeweils beim Erreichen und Verlassen des Knotens n. Damit kann das Datenfluß-Gleichungssystem wie folgt initialisiert werden:

$$IN[n,d]^0 = \begin{cases} \perp & , falls\ n = Schleifeneintritt \\ \bigwedge\limits_{m \in pred(n)} OUT[m,d]^0 & , sonst \end{cases}$$

$$OUT[n,d]^0 = \begin{cases} \top & , falls\ d \in G[n] \\ IN[n,d]^0 & , sonst \end{cases}$$

Zu Beginn wird die Iterationsdistanz der OUT für erzeugende Knoten überschätzt und die Werte der Iterationsdistanz des Schleifeneingangs unterschätzt. In allen anderen Fällen werden die Iterationsdistanzen mittels der Minimum Funktion ($\wedge$) durch den Kontrollflußgraphen propagiert.

Nach der Initialisierung muß das Datenfluß-Gleichungssystem gelöst werden. Zu diesem Zweck werden die folgenden Operationen iterativ angewendet und zwar solange, bis sich das Gleichungssystem stabilisiert, das heißt, daß nach der i-ten Iteration das Ergebnis mit der $i-1$-ten Iteration übereinstimmt.
Die Operationen zur Lösung des Datenfluß-Gleichungssystem sind wie folgt definiert:

$$IN[n,d]^{i-1} = \bigwedge_{m \in pred(n)} OUT[m,d]^i$$

$$OUT[n,d]^i = f_n^d(IN[n,d]^i)$$

Nach der Stabilisierung des Datenflußgleichungssystems sind alle Iterationsdistanzen zwischen allen Knoten innerhalb des LCFGs bestimmt.

4.4.6 Parametrisierung

Einer der Vorteile der δ-Array Datenflußanalyse ist die Möglichkeit zur Parametrisierung. Mit Parametrisierung ist gemeint, daß das Verfahren relativ einfach an unterschiedliche

Datenflußprobleme angepaßt werden kann. Somit sind neben der standardmäßig vorgesehenen *Must-Forward*-Analyse auch alle anderen möglichen Kombinationen aus *Must/May*- und *Forward/Backward*-Analysen möglich.

Vorwärts- und Rückwärtsprobleme

Die bisher vorgestellte Vorgehensweise stellt die Lösung einer Forward-Analyse dar. Da es aber eine Reihe von Problemen gibt, welche eine Rückwärtsbetrachtung des Datenflusses benötigen, ist es sinnvoll auch Rückwärtsanalysen zu betrachten.

Einer der Vorteile der δ-Array Datenflußanalyse liegt darin, daß eine Rückwärtsanalyse ohne große Änderungen möglich ist. Zu diesem Zweck wird der LCFG, auf dem die Datenflußanalyse arbeitet, invertiert. Bei der Analyse fließt der Datenfluß somit in die entgegengesetzte Richtung. Dies beinhaltet, daß Informationen von Nachfolgerknoten zu Vorgängerknoten und von späteren Iterationen zu früheren Iterationen transportiert werden, wodurch sich sich die positiven und negativen Iterationsdistanzen umkehren. Da dies aber nur die *Erhaltungsfunktion* betrifft, ist somit nur ein Anpassen der Funktion $k(i)$ nötig. Die Funktion $k(i)$ ergibt sich somit zu:

$$k(i) = \frac{a_2 - a_1}{a_1} \times \frac{b_2 - b_1}{a_1}$$

Must- und May-Probleme

Das bisher vorgestellte Verfahren bezog sich nur auf die Analyse von *Must*-Problemen. Um die δ-Array Datenflußanalyse auch für *May*-Probleme einsetzen zu können, bedarf es einiger Änderungen bzgl. der partiellen Ordnung des Verbandes L, dessen Operatoren, der *Erhaltungsfunktion* und der Initialisierung des *Datenfluß-Gleichungssystems.*

Um eine May-Analyse durchführen zu können, muß die partielle Ordnung des Verbandes L umgekehrt werden. Damit ergibt sich die Definition des Verbandes wie folgt:

$$\sqsubseteq: \forall x \in [\bot = UB - 1, \ldots, 0, \top] : x_i < x_{i+1}$$

Somit bezeichnet nun $\top$ die Gültigkeit über keine Distanz und $\bot$ die Gültigkeit über alle Distanzen. Dementsprechend müssen die Operatoren des Verbandes ($\wedge$ und $\vee$) angepaßt werden. Dazu reicht es, die jeweiligen Operatoren zu vertauschen.
Damit gilt: $\wedge = max$ und $\vee = min$.

Im Gegensatz zu einem *Must*-Problem verliert bei einem *May*-Problem eine Definition erst seine Gültigkeit, wenn dies eindeutig nachgewiesen worden ist. Um dies zu erreichen

muß die *Erhaltungsfunktion* ($f_n^d(x) = max\{x, p_n^d\}$) entsprechend angepaßt werden. Dies geschieht durch eine geänderte Bestimmung der Konstanten p_n^d. Im Gegensatz zu einer *Must*-Analyse, wo der Wert der Konstanten p_n^d unterschätzt wird, muß dieser Wert bei einer *May*-Analyse überschätzt werden. Durch die Überschätzung der Lösung wird ausgedrückt, daß Instanzen von d solange erhalten bleiben, bis diese mit Sicherheit von Elementen in $K[n]$ vernichtet werden. Dieser Fall kann nur auftreten, wenn zwischen zwei Definitionen eine konstante Iterationsdistanz liegt. Also gilt:

$$d = X[idx(i)] \text{ mit } d \in K$$

und

$$d' = X[idx(i+c)]$$

Mit der Konstante c wird somit ausgedrückt, daß in $c-1$ Iterationen der Schleife der Wert der Definition d' erhalten bleibt. Nach c Iterationen wird der Wert der Definition d' von der Definition d eindeutig vernichtet.
Damit ergibt sich die folgende Berechnung für die Konstante p_n^d:

$$p_n^d = \begin{cases} \top & \forall i \in [0, \ldots, UB-1] : k(i) = pr(d,n) \\ c-1 & \forall i \in [0, \ldots, UB-1] : k(i) = c,\ c > pr(d,n) \\ \bot & ,\ sonst \end{cases}$$

Durch die Veränderung der Ordnung des Verbandes L, muß weiterhin die Initialisierung des Datenfluß-Gleichungssystem verändert werden. In diesem Fall ist es nur Notwendig alle Werte der IN und OUT Vektoren mit dem Wert $\bot$ zu belegen. Das entspricht im Falle einer May-Analyse einer initialen Überschätzung der Lösung des Datenfluß-Gleichungssystems.

4.4.7 Implementierung

Die im Rahmen dieser Diplomarbeit implementierte δ-Array Datenflußanalyse wurde in einigen Punkten modfiziert und verbessert. Diese Veränderungen waren nötig, um ein besseres Ergebnis bzgl. der Vektorisierung zu erhalten. Dieses Kapitel beschreibt nun, welche Veränderungen an dem Verfahren durchgeführt worden sind.

Berechnung der Iterationsdistanz p_n^d

Die Berechnung der Iterationsdistanz p_n^d wurde so verbessert, daß sich folgende zwei Fälle ebenfalls bestimmten lassen:

- Bestimmung der Iterationsdistanzen von affinen Ausdrücken ohne Kenntnis der Iterationsanzahl der Schleife.
- Bestimmung der Iterationsdistanz von nicht affinen Ausdrücken.

Der erste Punkt spiegelt den Fall wieder, daß für die beiden affinen Ausdrücke $a_1 \times i + b_1$

und $a_2 \times i + b_2$ gilt: $a_1 = a_2$. In diesem Fall ergibt sich die Iterationsdistanz zu $|b_1 - b_2|$, welche ohne Kenntnis bzgl. der Iterationsanzahl der Schleife bestimmt werden kann.

Im zweiten Fall kann eine einfache Bestimmtung der Iterationsdistanzen nicht mehr erfolgen. Zu diesem Zweck wird der komplette Lösungsraum der Iterationsdistanzen berechnet. Da dies nur für eine kleine Iterationsanzahl möglich ist, muß dazu die Anzahl der Iterationen bekannt sein.

Optimierte Bestimmung der Indexterme
Zur bestimmung der Indexterme von Arrayzugriffen wurde eine Verfahren gewählt, welches eine optimale Bestimmung der jeweiligen Indexterme ermöglicht. Dazu wird im Vorfeld eine Variablenanalyse durchgeführt um zu bestimmen, welche Variablen sich innerhalb der Programm-Darstellung verändern und welche Variablen ihren Wert konstant beibehalten. Anhand dieser Informationen ist es möglich, die Indexterme innerhalb der Programm-Darstellung so zu bestimmen, daß eine möglichst hohe Anzahl von konstanten Werten verwendet werden kann.

Zur besseren Verdeutlichung sein nun das Beispiel 4.4.3 betrachtet.

Beispiel 4.4.3 *Indextermbestimmung*

```
int main()
{
int i, j;
int A[20];

  for ( i = 0; i < 16; i++ )
  {
    A[i] = 0;

    if ( i & 1 )
    {
      j = 0;
    }
    else
    {
      j = 4;
    }

    A[i + j] = 5;
  }

  return 0;
}
```

In der ursprünglichen Version der δ-Array Datenflußanalyse wären die Indexterme für $A[i] = 0$ und $A[i+j] = 5$ als $idx_1 = \&A + i * 4$ und $idx_2 = \&A + i * 4 + j * 4$ bestimmt worden. Durch die Einschränkung der normalen δ-Array Datenflußanalyse auf affine Ausdrücke, wäre eine Analyse dieser Schleife nicht möglich gewesen.

In der verbesserten Version der δ-Array Datenflußanalyse wäre im Rahmen der Variablenanalyse die Variable j als konstant eingestuft worden. Durch rekursives Ersetzen der Variable durch ihre Zuweisungen ergeben sich für den Arrayzugriff $A[i+j] = 5$ die Indexterme $idx_{2,1} = \&A + i*4 + 0*4 = \&A + i*4$ und $idx_{2,2} = \&A + i*4 + 4*4 = \&A + i*4 + 16$. Nach der Zusammenfassung der Konstanten ergeben sich wieder zwei affiner Ausdrücke, welche eine Analyse der Schleife ermöglichen. Da durch dieses Vorgehen mehrere Indexterme pro Knoten innerhalb des Schleifenkontrollflußgraphen entstehen können, wurde die δ-Array Datenflußanalyse auch im Hinblick auf Multi-Indexterme angepaßt.

Multi-Indexterme

Die verbesserte Version der δ-Array Datenflußanalyse ist im Gegensatz zu der von Duesterwald, Gupta und Soffa [DGS93] publizierten Version in der Lage, auch mit mehreren

Indextermen pro Knoten innerhalb des Schleifenkontrollflußgraphen zu arbeiten. Diese Erweiterung wurde nötig im Hinblick auf die bereits vorgestellte verbesserte Bestimmung von Indextermen.

Zur besseren Verdeutlichung sei nun das folgende Beispiel 4.4.4 betrachtet:

Beispiel 4.4.4 *Programm mit Multi-Indextermen*

```
int main()
{
int i, j;
int A[20];

  for ( i = 0; i < 16; i++ )
  {
    A[i] = 0;

    if ( i & 1 )
    {
      j = i + 1;
    }
    else
    {
      j = i + 2;
    }

    A[j] = 5;
  }

  return 0;
}
```

Mit einer Größe des Datentypen *int* von 32-Bit ergeben sich folgende Indexterme zur Adressierung des Speichers des Array A:

C Code	**Indexterme**
A[i] = 0	$\&A + i \times 4$
A[j] = 5	$\&A + (i+1) \times 4$ $\&A + (i+2) \times 4$

Tabelle 4.1: Indexterme zu Beispiel 4.4.4

Man erkennt, daß für den Arrayzugriff $A[j]$ zwei verschiedene Indexterme existieren. Aus

diesem Grund ist die implementierte Version der δ-Array Datenflußanalyse in der Lage, solche mehrfach auftretenden Indexterme zu verarbeiten.

Abhängigkeitsanalyse bzgl. Speicherdistanzen
Für die Durchführung einer Vektorisierung bedarf es im Gegensatz zu anderen Optimierungen nicht der Bestimmung von Iterationsdistanzen. Vielmehr werden die Distanzen von Speicherzugriffen auf Arrays benötigt, was einer entsprechenden Anpassung des Verfahrens bedurfte. Das Verfahren ist somit in der Lage nicht nur Iterationsdistanz zu bestimmen, sondern auch die für Vektorisierung benötigten Distanzen von Speicherzugriffen. Dies ist im Hinblick zur Analyse von C/C++ Programmen nötig, da die C/C++ Programmiersprache durch ihre Zeigerarithmetik und den damit verbundenen Aliaseffekten weitere Abhängigkeiten erzeugt, welche es zu ermitteln gilt.

Dazu sei das Beispiel 4.4.5 betrachtet:

Beispiel 4.4.5 *Aliaseffekt durch die Verwendung von Zeigervariablen*

```
int main()
{
int i;
int A[20];
char* pA;

  pA = (char*) &A;

  for ( i = 0; i < 16; i++ )
  {
    A[i]  = 0;
    pA[i] = 1;
  }

  return 0;
}
```

Durch die verwendete Zeigervariable pA schreiben zwei verschiedene Variablen in den gleichen Speicherbereich des Arrazs A. Erschwerend kommt hinzu, daß die Variable pA einen anderen Typ als die Variable A besitzt, dessen Größe sich vom Typ der Variablen A unterscheidet. Da sich *A[i]* und *pA[i]* auf unterschiedliche Speicherbereiche beziehen, ist eine einfache Bestimmung der Iterationsdistanzen nicht mehr ausreichend. Weitere Informationen zu dieser Problematik können dem Kapitel 5 dieser Diplomarbeit entnommen werden.

4.4.8 Beispiel

Den Abschluß dieses Kapitels bildet ein Beispiel für die Durchführung einer δ-Array Datenflußanalyse.

Beispiel 4.4.6 *Programm für δ-Array Datenflußanalyse*

```
int main()
{
int i;
int A[16];

  for ( i = 0; i < 16; i++ )
  {
    A[i + 0] = 0;

    if ( i & 1 )
    {
      A[i + 1] = 1;
    }
    else
    {
      A[i + 2] = 2;
    }

    A[i + 5] = 5;
  }

  return 0;
}
```

Das verwendete Beispiel 4.4.6 enthält die vier schreibenden Speicherzugriffe $A[i+0] = 0$, $A[i+1] = 1$, $A[i+2] = 2$ und $A[i+5] = 5$, welche in dasselbe Array schreiben. Man erkennt, daß der Speicherzugriff $A[i+0] = 0$ von den anderen Speicherzugriffen nie verändert wird. Somit sind alle Werte der *Erhaltunsfunktionen* mit $\top$ belegt, da die Gültigkeit nie verloren geht. Der Wert des Speicherzugriff $A[i+1] = 1$ wird ebenfalls von den Speicherzugriffen $A[i+2] = 2$ und $A[i+5] = 5$ nie verändert, womit sich die Werte der entsprechenden Erhaltungsfunktionen ebenfalls zu $\top$ ergeben. Dahingegen wird dieser Wert von dem Speicherzugriff $A[i+0] = 0$ nach einer Iteration überschrieben. Somit ergibt sich für die Erhaltungsfunktion der Wert eins. Das Berechnen der weiteren Iterationsdistanzen der Erhaltungsfunktionen geschieht analog.

	A[i + 0] = 0	**A[i + 2] = 2**	**A[i + 5] = 5**	**A[i + 1] = 1**
A[i + 0] = 0	max(x, 0)	min(x, 2)	min(x, 5)	min(x, 1)
A[i + 2] = 2	min(x, ⊤)	max(x, 0)	min(x, 3)	min(x, ⊤)
A[i + 5] = 5	min(x, ⊤)	min(x, ⊤)	max(x, 0)	min(x, ⊤)
A[i + 1] = 1	min(x, ⊤)	min(x, 1)	min(x, 4)	max(x, 0)
f^d_{exit}	x++	x++	x++	x++

Tabelle 4.2: Transferfunktionen zu Beispiel 4.4.6

Nach dem Aufstellen aller Transferfunktionen (Siehe Tabelle 4.2) wird nun die Datenflußanalyse durchgeführt. Man erkennt die effiziente Laufzeit des Verfahrens, da sich die Lösung nach bereits zwei Iterationen stabilisiert hat (Tabelle 4.5).

	A[i + 0] = 0	**A[i + 2] = 2**	**A[i + 5] = 5**	**A[i + 1] = 1**
A[i + 0] = 0	⊤	⊤	⊤	⊤
A[i + 2] = 2	⊤	⊤	⊤	⊤
A[i + 5] = 5	⊤	⊤	⊤	⊤
A[i + 1] = 1	⊤	⊤	⊤	⊤
f^d_{exit}	⊥	⊥	⊥	⊥

Tabelle 4.3: Nach Initialisierung

	A[i + 0] = 0	**A[i + 2] = 2**	**A[i + 5] = 5**	**A[i + 1] = 1**
A[i + 0] = 0	⊤	2	5	4
A[i + 2] = 2	⊤	⊤	3	⊤
A[i + 5] = 5	⊤	⊤	⊤	⊤
A[i + 1] = 1	⊤	1	4	⊤
f^d_{exit}	⊤	⊤	⊤	⊤

Tabelle 4.4: Nach 1. Iteration

	A[i + 0] = 0	**A[i + 2] = 2**	**A[i + 5] = 5**	**A[i + 1] = 1**
A[i + 0] = 0	⊤	2	5	4
A[i + 2] = 2	⊤	⊤	3	⊤
A[i + 5] = 5	⊤	⊤	⊤	⊤
A[i + 1] = 1	⊤	1	4	⊤
f^d_{exit}	⊤	⊤	⊤	⊤

Tabelle 4.5: Nach 2. Iteration

Kapitel 5

Vektorisierung

5.1 Arbeiten zum Thema Vektorisierung

In dem Bereich Vektorisierung sind etliche Untersuchungen und Implementierungen vorgenommen worden. Dabei unterscheiden sich die Vorgehensweisen wie die Vektorisierung durchgeführt wird erheblich voneinander. Die nun in den folgenden Abschnitten vorgestellten Verfahren wurden dementsprechend ausgewählt, um einen guten Überblick über die Möglichkeiten zur Vektorisierung zu geben.

5.1.1 Compiling C for Vectorization, Parallelization, and Inline Expansion

In dem Artikel von Allen und Johnson [AJ88] wird zur Vektorisierung auf eine eigene High-Level IR Form zurückgegriffen. Der Grund dafür ist, daß durch die Darstellung eines Programmes in (Low-Level) IRs wichtige Informationen verloren gehen können. So werden Schleifen werden beispielsweise direkt in *If-Goto* Anweisungen transformiert. Dies kann dazu führen, daß die Grenzen der Schleife nicht mehr ermittelt werden können. Ein weiteres Problem stellen Optimierungen auf der (Low-Level) IR Ebene dar. Viele Optimierungen führen eine starke Veränderung des Originalcodes durch und erzeugen zusätzlich redundanten Code, welcher standardmäßig direkt durch geeignete weitere Optimierungen (wie *Dead-Code Elimination*) entfernt wird. Durch diese Vorgehensweise ist es nicht ausgeschlossen, daß wichtige Informationen zur Vektorisierung verloren gehen können.

Der Artikel in [AJ88] gibt einen Überblick über Techniken zur Transformation von C Programmen in ein für die Vektorisierung verbesserte Form. Zu diesem Zweck werden die zu vektorisierenden C Programme mittels der drei folgenden Verfahren transformiert.

Function Inlining
Wie bereits in Kapitel 6.5 dieser Diplomarbeit erläutert wurde, können Abhängigkeitsanalysen über Funktionsebenen hinweg nicht analysiert werden. Zu diesem Zweck werden kleinere Funktionen in die zu analysierende Funktion eingesetzt.

For-Loop Transformation
Die wichtigsten Schleifen bei der Vektorisierung stellen *For*-Schleifen dar. Das Problem von C *For*-Schleifen besteht darin, daß sie für serielle Architekturen entwickelt worden sind. C *For*-Schleifen können komplexe Abbruchbedingungen enthalten oder ihre Schleifenkörper mittels *Break* und *Goto* Anweisungen verlassen. Da für die Vektorisierung aber i.allg. eine genau Kenntnis über die Schleifengrenzen erforderlich ist, werden diese Schleifen in die Form von *Fortran DO-ENDO* Schleifen transformiert. *Fortran DO-ENDO* Schleifen sind immer strukturiert und können somit besser für ein Vektorisierung verwendet werden.

Induction Variable Substitution
Viele Schleifen in C können Seiteneffekte beinhalten, die es zu analysieren gilt. Viele der Analysemethoden arbeiten auf affinen Ausdrücken oder benötigen eine eindeutige Schleifenindexvariable. Zu diesem Zweck werden Schleifen ohne eine Schleifenindexvariable in eine Schleife mit einer Schleifenindexvariable transformiert.

Dazu sei das folgende Beispiel betrachtet:

```
while(n)
{
  *a++ = *b++;
  n--;
}
```

Übersetzt ergibt sich folgender Programmcode:

```
while(n)
{
  temp1 = &a;
  a = temp1 + 4;
  temp2 = &b;
  b = temp2 + 4;
  *temp1 = *temp2;
  temp3 = n;
  n = temp3 - 1;
}
```

Man erkennt, daß es sich dabei um einen ganz normalen Kopierbefehl handelt. Um nun eine Datenflußanalyse durchführen zu können, wird die Indexvariable i in die Terme eingesetzt. Damit ergibt sich der folgende Ausdruck:

```
for ( i = 0; i < n; i++ )
{
  *(a + i * 4) = *(b + i * 4);
}
```

Dieser Ausdruck kann nun leicht durch eine vorhandene Datenflußanalyse analysiert

werden.

Wie bereits Anfangs erwähnt, handelt es sich hierbei nicht um ein vollständiges eigenes Verfahren. Vielmehr wird versucht, mittels einer geschickt gewählten High-Level IR und verschiedenen Transformationen, C Programmcode in eine möglichst optimale Form für eine Vektorisierung zu bringen. Die Vektorisierung erfolgt dann letztendlich durch bekannte Verfahren, welche für die Verwendung in *Fortran* Compilern, entwickelt worden sind.

Vorteile des Verfahrens

- Durch die Transformation von Schleifen- und Variablen-Zugriffen ist es möglich, bereits vorhandene Vektorisierungsalgorithmen, welche für die Sprache *Fortran* entwickelt worden sind, zu verwenden.
- Die verwendete High-Level IR ermöglicht eine genauere Analyse von Abhängigkeiten, was zu besseren Resultaten bei der Vektorisierung führen kann.

Nachteile des Verfahrens

- Durch die Verwendung einer eigenen High-Level IR sind möglicherweise bereits vorhandene LIR Optimierungen nicht mehr anwendbar.
- Die Transformation der High-Level IR in eine Low-Level IR, zur Verwendung vorhandener Optimierungen, kann zu fehlerhaftem Programmcode führen, wenn die Low-Level Optimierungen nicht an die Vektorbefehle angepaßt worden sind.

5.1.2 Exploiting Superword Level Parallelism with Multimedia Instruction Sets

Das von Samuel Larsen und Saman Amarasinghe [LA00] vorgestellte Verfahren basiert auf der Idee, Vektorisierung nur auf Basisblock-Ebene durchzuführen. Zu diesem Zweck wird in dem Verfahren im ersten Schritt die betrachtete Schleife n-mal abgerollt, wobei n z.B. für die Anzahl der Datenpfade des Prozessors stehen könnte. Dadurch wird ein Großteil der Abhängigkeitsanalysen auf eine in den meisten Fällen bereits bestehende Optimierung verlagert. Im zweiten Schritt werden *isomorphe Anweisungen* innerhalb eines Basisblocks identifiziert. Dabei bezeichnen isomorphe Anweisungen, welche die gleichen Operationen in der gleichen Reihenfolge beinhalten.

Beispiel 5.1.1 *Isomorphe Anweisungen*

```
A[I + 0] = B[I + 0] + 10;
A[I + 1] = B[I + 1] + 10;
```

Die beiden Anweisungen in Beispiel 5.1.1 sind in dem oben genannten Sinne isomorphe Anweisungen, da sie die gleichen Operationen durchführen, nur mit unterschiedlichen Indizes.

In dem nächsten Schritt, welcher als *Packing* bezeichnet wird, werden isomorphe Anweisungen zu einer Gruppe zusammengefaßt, wenn dies möglich ist. Im Abschluß wird nun untersucht, ob die neu entstandene Gruppe vektorisierbar ist.

Das folgende Beispiel 5.1.2 verdeutlicht das Verfahren. Dabei wird von einem Prozessor mit 4 Datenpfaden ausgegangen.

Beispiel 5.1.2 *Vektorisierung*

(a) Originiale Schleife

```
for ( i = 0; i < 16; i++ )
{
  localdiff = ref[i] - curr[i];
  diff     += abs(localdiff);
}
```

(b) Abgerollte Schleife

```
for ( i = 0; i < 16; i += 4 )
{
  localdiff = ref[i + 0] - curr[i + 0];
  diff     += abs(localdiff);

  localdiff = ref[i + 1] - curr[i + 1];
  diff     += abs(localdiff);

  localdiff = ref[i + 2] - curr[i + 2];
  diff     += abs(localdiff);

  localdiff = ref[i + 3] - curr[i + 3];
  diff     += abs(localdiff);
}
```

(c) Schleife nach Packaging

```
for ( i = 0; i < 16; i += 4 )
{
  localdiff0 = ref[i + 0] - curr[i + 0];
  localdiff1 = ref[i + 1] - curr[i + 1];
  localdiff2 = ref[i + 2] - curr[i + 2];
```

```
    localdiff3 = ref[i + 3] - curr[i + 3];

    diff     += abs(localdiff0);
    diff     += abs(localdiff1);
    diff     += abs(localdiff2);
    diff     += abs(localdiff3);
  }
```

(d) Schleife nach Vektorisierung

```
  for ( i = 0; i < 16; i += 4 )
  {
    localdiff = ref[i] simd_minus curr[i];

    diff     += abs(localdiff[0]);
    diff     += abs(localdiff[1]);
    diff     += abs(localdiff[2]);
    diff     += abs(localdiff[3]);
  }
```

Vorteile des Verfahrens

- Das Verfahren ist relativ einfach zu implementieren, da ein Hauptteil der Analysen auf bereits vorhandene Optimierungen verlagert werden kann. Auch die weiteren Analysen sind relativ einfach zu implementieren, da sie nur auf Basisblock-Ebene durchgeführt werden. Ein Betrachten des Daten- und Kontrollflusses der entsprechenden Funktion ist somit nicht nötig.
- Schleifen können auch zum Teil vektorisiert werden, wenn nicht vektorisierbar Instruktionen in der Schleife vorhanden sind.

Nachteile des Verfahrens

- Das Verfahren setzt eine direkte Kopplung zwischen dem Compiler und der Vektorisierungseinheit voraus. Abgerollte Schleifen, welche nicht vektorisiert werden können, müssen vom Compiler erneut übersetzt werden, da die sonst entstehende Codegröße nicht mehr im Verhältnis zu dem zu erzielenden Geschwindigkeitsgewinn steht.
- Durch das Loop-Unrolling entstehen eine Vielzahl von Anweisungen, welche nicht vektorisiert werden können. Man betrachte dazu den Programmcode in Beispiel 5.1.2(d), wo die Anweisung *diff += abs(localdiff[0])* viermal erzeugt wurde. Diese Anweisungen sorgen für eine starke Erhöhung der Codegröße.
- Das Verfahren arbeitet nur auf Schleifen, welche keine Kontrollflußverzweigungen besitzen. Dadurch wird gewährleistet, daß sich der Programmcode innerhalb der Schleife in einem Basisblock befindet. Dies stellt allerdings keine starke Einschränkungen

bzgl. einer Vektorisierung dar, da eine Analyse der Datenabhängigkeiten über verschiedene Kontrollflußwege hinweg, in den meisten Fällen nicht mehr möglich ist (siehe Kapitel 5.3.4 dieser Diplomarbeit).

Schlußfolgerung

Das Verfahren ist sehr gut geeignet, um eine Vektorisierung durchzuführen. Einer der Vorteile besteht darin, daß auch Teile von Schleifen vektorisiert werden können, selbst wenn Abhängigkeiten zu skalaren Operationen bestehen. Als Nachteil bleibt zu nennen, daß es im Extremfall zu einer starken Vergrößerung des Codes kommen kann. Vor allem bei Prozessoren, welche nur über einen kleinen Programmspeicher verfügen, kann dies zu Problemen führen. In diesem Fall ist es sinnvoll, die Schleife mit Optimierungen wie *Loop-Fission* in mehere Teilschleifen zu zerlegen, um einen kompakteren Maschinencode zu erhalten.

5.1.3 A Vectorizing Compiler for Multimedia Extensions

In der Arbeit von N. Sreraman, R. Govindarajan [SG00] wird ein Vektor Compiler vorgestellt, welcher auf dem SUIF Compiler der Universität Stanford basiert. Das Ziel dieser Arbeit besteht in der Ausnutzung von Multimedia Befehlen in modernen Prozessoren. Dabei kam als Referenzprozessor ein Intel x86 Prozessor zum Einsatz, wobei für die Vektorisierung der erweiterte *MMX* (Multi Media eXtension) Befehlssatz des Prozessors verwendet worden ist.

Wie die im Rahmen dieser Diplomarbeit durchgeführte Vektorisierung wird auch hier auf einer Low-Level IR Ebene gearbeitet. Dabei wurden eine Reihe von Optimierungen zur Schleifentransformation eingesetzt, die eine hohe Ausbeute an vektorisierbaren Schleifen erzeugen. Zu diesen Optimierungen gehören unter anderem die Optimierungen *Strip Mining*, *Scalar Expansion* und *Loop Distribution* (Loop Fission), welche in Kapitel 6.6 mit weiteren unterstützenden Optimierungen vorgestellt werden. Der Vektorisierungsprozeß läuft dabei folgendermaßen ab.

Im ersten Schritt werden die Abhängigkeiten von Arrayzugriffen analysiert. Die dabei verwendeten Algorithmen zur Erkennung von Kontrollfluß- und Datenflußabhängigkeiten arbeiten auf gewöhnlichen Kontrollfluß- und Datenflußgraphen. Dabei wurden bekannte Verfahren eingesetzt, wie sie in [?], [W96], [ZC91] und in Kapitel 4 diese Diplomarbeit beschrieben sind. Sind keine Abhängigkeiten zwischen Arrayzugriffen erkannt worden, werden die oben genannten Optimierungen durchgeführt, um die Schleife in einer für die Vektorisierung verbesserte Form zu bringen. Im zweiten Schritt wird die zu vektorisierende Schleife mittels Loop Distribution in Einzelschleifen zerteilt, so daß die Vektorisierung verhindernde Instruktionen in anderen Teilschleifen zum liegen kommen. Im letzten Schritt wird durch die Schleifentransformationen *Scalar Expansion* und *Strip Mining* die Schleife in eine entgültig vektorisierbare Form gebracht.

Vorteile des Verfahrens

- Ein großer Vorteil des Verfahrens liegt darin, daß es viele Untersuchungen und Implementierungen von Analysen, Schleifentransformationen und Optimierungen gibt.
- Ein weiterer Vorteil liegt in der Möglichkeit (im Gegensatz zu dem hier vorgestellten Verfahren in Kapitel 5.1.2), auch Schleifen mit Kontrollflußverzweigungen vektorisieren zu können. Dadurch ergibt sich eine weitaus größere Reihe von vektorisierbaren Schleifen, was sich auch im Ergebnis dieser hier vorgestellten Implementierung wiederspiegelt.

Nachteile des Verfahrens

- Der Nachteil des Verfahrens liegt in seiner Komplexität, da die zum Einsatz kommenden Analysen und Transformationen in ihrer Implementierung sehr aufwendig sind. Kann bei der Vektorisierung nicht auf vorhandene Implementierungen zurückgegriffen werden, so bedarf es eines großen Arbeitsaufwands, diese zu implementieren.

Schlußfolgerung

Das hier vorgestellte Verfahren (wie es auch dem dieser Diplomarbeit entspricht), bietet einen sehr hohen Vektorisierungsgrad von Schleifen, was sich in den Ergebnissen der durchgeführten Untersuchungen wiederspiegelt. Durch die Vektorisierung wurde eine Geschwindigkeitszuwachs von einem Faktor 2 bis zu 6.5 erreicht, wobei die Güte des generierten Assemblercodes in manchen Fällen 85% der Güte von handoptimierten Assemblercode entsprach. In Anbetracht, daß die MMX SIMD Instruktionen 4 32-Bit Werte parallel verarbeiten können, kann von einer weiteren Steigerung der Ausführungsgeschwindigkeit bei Prozessoren mit einer höheren Anzahl von Datenpfaden ausgegangen werden. Vor allem bei Prozessoren, welche stark auf die parallele Verarbeitung von Daten ausgelegt sind (wie Prozessoren der *M3-DSP* Plattform), kann mit einer weiteren Steigerung der Ausführungsgeschwindigkeit gerechnet werden.

5.1.4 Zusammenfassung

Die hier vorgestellten Verfahren zeigen nur einen Teil der Möglichkeiten zur Durchführung von Vektorisierungen. Wie sich gezeigt hat, kann eine Vektorisierung auf einer hohen Ebene des Compilierungsprozesses, ebenso wie auch auf LIR Ebene durchgeführt werden. Dabei hängt die Wahl, auf welcher Ebene die Vektorisierung durchgeführt wird, von dem verwendeten Compiler und den vorhandenen Tools ab. Auch die Frage, welche Zwischenrepresentation des Quellcodes verwendet werden sollte, kann nicht eindeutig beantwortet werden, da jede Zwischenrepresentation ihre Vor- und Nachteile hat. So wurde z.B. in [BIK99] eine gänzlich neue Zwischensprache entworfen, die ein C Quellprogramm in eine C ähnliche Sprache übersetzt, welche mit einer Reihe von Informationen bzgl. der möglichen parallelen Ausführung von Instruktionen angereichert wird. Dadurch wurde es möglich, eine Reihe der benötigten Analysen von den jeweiligen Optimierungen zu nehmen. Dies erleichtert die Implementierung neuer Optimierungen, hat aber zur Folge, daß viele bereits

vorhandene Optimierungen neu geschrieben werden müssen. Es kann somit nicht eindeutig entschieden werden, wo und auf welcher Basis Vektorisierungen durchgeführt werden sollten.
Die im Rahmen dieser Diplomarbeit realisierte Vektorisierung wurde auf der Basis der *GeLIR* entwickelt. Durch die in den *GeLIR* Strukturen vorhandenen Informationen bzgl. der verwendeten Zielarchitektur, konnte eine generische Lösung von einer Reihe von Prozessoren der *M3-DSP* Plattform realisiert werden.

Dieses Kapitel behandelt den Schwerpunkt der Diplomarbeit, nämlich die Vektorisierung für Prozessoren der *M3-DSP* Plattform.

5.2 Einführung

Die Vektorisierung zählt zu den wichtigsten Optimierungen für die *M3-DSP* Plattform, wobei als Grund dafür die zwei folgenden entscheidenen Punkte zu nennen sind:

- Die Prozessoren der *M3-DSP* Plattform sind durch ihre Vektoreinheit hochgradig auf paralle Verarbeitung von Daten ausgelegt. Es ist somit zur Erhöhung der Ausführungsgeschwindigkeit von Programmen unerläßlich, diese zur parallelen Verarbeitung von Daten vorhandenen Einheiten der Prozessoren gezielt auszunutzen.
- Ein Speicherzugriff auf einzelne Variablen zur Bearbeitung von skalaren Werten ist nicht möglich. Vielmehr wird pro Speicherzugriff immer eine Gruppe von Werten geladen, auch wenn nur ein Wert benötigt wird. Obwohl bereits Techniken zur Verringerung der Speicherzugriffe untersucht worden sind ([LKB01], [K00]), hat es sich gezeigt, daß trotz dieser Optimierungen weiterhin Speicherzugriffe mit einem gewissen Overhead verbunden sind. Um eine optimale Ausnutzung der Gruppenregister zu erhalten, ist die Vektorisierung von Speicherzugriffen unumgänglich.

Somit wurde das Ziel dieser Diplomarbeit dahingehend gesetzt, eine generische Vektorisierung für die Prozessoren der *M3-DSP* Plattform zu implementieren.

5.2.1 Überblick

Der Schwerpunkt der untersuchten und implementierten Optimierungen liegt in der Vektorisierung von Speicherzugriffen und arithmetischen Operationen, womit sich dieses Kapitel der Diplomarbeit beschäftigt.

Der erste Teil dieses Kapitels geht auf die Problematiken ein, welche bei der Vektorisierung von Speicherzugriffen entstehen. Dabei werden Analysen vorgestellt, welche zur Durchführung einer stets gültigen Vektorisierung notwendig sind. Der zweite Teil behandelt dann die pratische Durchführung der Vektorisierung. Es wird dabei Schritt für Schritt auf die Implementierten Analysen und die Durchführung der Optimierung eingegangen. In dem nachfolgenden Kapitel 6 werden dann Optimierungen vorgestellt, welche zum einem

spezielle Eigenschaften der *M3-DSP* Prozessoren ausnutzen und zum anderen Optimierungen, welche einen positiven Einfluß auf die Vektorisierbarkeit von Schleifen haben.

5.3 Vorausetzungen zur Vektorisierung

Um eine Vektorisierung auf LIR Ebene durchführen zu können, bedarf es einer Reihe von Voraussetzungen welche erfüllt sein müssen. Dieser Abschnitt geht nun im Detail auf diese Voraussetzungen ein. Dabei werden entstehende Probleme diskutiert und aufgeizeigt, wie solche Probleme vermieden bzw. behoben werden können.

5.3.1 Schleifengrenzen und Schrittweite der Indexvariable

Das Ermitteln von Schleifengrenzen und Schrittweiten der in Schleifen vorhandenen Variablen ist von essentieller Bedeutung für die Vektorisierung. Ohne genaue Kenntnis über die Grenzen einer Schleife und der Schrittweite der beteiligten Indexvariablen, ist eine stets gültige Vektorisierung nicht durchführbar

Betrachtet sei dazu das folgende Beispiel 5.3.1, welches eine nicht vektorisierbare Schleife enthält. Der angenommene Prozessor, für den die Schleife vektorisiert werden soll, soll dabei 16 Datenpfade (*Slices*) enthalten, wobei jeder Slice auf einem 16-Bit Integer Wert arbeitet.

Beispiel 5.3.1 *Nicht vektorisierbare Schleife*

```
void NonVectorizableLoopFunction()
{
int i;
int A[24];

  for ( i = 2; i < 14; i+= 2)
  {
    A[i] = 0;
  }
}
```

Die Vektorisierung der Schleife ist aus folgenden Gründen nicht oder nur bedingt möglich:

- Die Größe des Arrays läßt keine Vektorisierung zu, da sie kleiner ist als die Größe der Vektoreinheit des Prozessor. Ein Vektorzugriff würde somit über die Grenzen des Arrays erfolgen, was zu einem fehlerhaften Programmverhalten führt. Zur Lösung dieses Problems bietet sich eine Vergrößerung der Array Grenzen auf eine durch

die Größe der Vektoreinheit teilbare Länge an. Weiterhin besteht die Möglichkeit, die Schleife in zwei Teilschleifen mittels *Strip Mining* (6.6) so zu zerlegen, daß eine Vektorisierung auf einer der Teilschleifen durchgeführt werden kann.

- Die Schrittweite des Indexwertes ist 2, so daß jeweils nur jedes zweite Element des Arrays verändert wird. Eine Vektorisierung ist somit nicht ohne weiteres möglich, da durch den Vektorzugriff alle Elemente des Arrays verändert werden. Manche Prozessoren (wie die Prozessoren der *M3-DSP* Plattform) erlauben das Maskieren von einzelnen Datenpfaden. Dadurch wird es möglich auch Schleifen mit einem Schrittwert von mehr als zu vektorisieren. Ist ein Maskieren der Datenpfade nicht möglich, so besteht die Möglichkeit, durch umkopieren der Elemente in ein temporäres Array, eine Vektorisierung zu ermöglichen. Dabei muß der entstehende Overhead mit dem zu erwartenden Gewinn an Performance abgeschätzt werden.

- Der Startwert der Induktionsvariable ist ungleich Null. Manche Vektorprozessoren, wie die Prozessoren der *M3-DSP* Familie, sind nicht in der Lage alle möglichen Speicherbereiche eines Arrays direkt zu lesen. Im Beispiel 5.3.1 würde der Prozessor das Array vom Beginn an (A[0]) in das Speicherregister laden und dort auch wieder speichern. Ein Verändern aller Werte würde somit zu einem falsch berechneteten Ergebnis führen, da die Werte A[0] und A[1] ebenfalls verändert werden würden. Auch hier besteht die Möglichkeit der Verwendung von *Condition Code Registern* oder dem umkopieren der Elemente in ein tempräres Array, was aber wiederum zu einem weiteren Overhead an Instruktionen führt.

- Die Lauflänge der Schleife (beträgt in diesem Fall 6) läßt ebenfalls eine Vektorisierung nicht zu, da dort ebenfalls mehr Werte gelesen bzw. verändert werden, als wie im Programmcode angegeben. Dieses Problem läßt sich ebenso mittels *Strip Mining* oder durch die Verwendung von *Condition Code Registern* beseitigen

Die Erkennung von Schleifengrenzen und der Schrittweite der Schleifenvariablen kann auf unterschiedliche Art und Weise erfolgen. Im besten Fall werden diese Informationen von dem Frontend des Compilers zur Verfügung gestellt, so daß diese auf der LIR Ebene ausgelesen werden können. Ist dies nicht der Fall, so bleibt nur die Möglichkeit diese Informationen aus dem LIR Programmcode selbständig zu ermitteln. Es bietet sich dazu an, beginnend von dem bedingten Sprungbefehl der Schleife aus den Kontrollfluß rückwärts zu verfolgen und mittels der Abhängigkeiten bzgl. der beteiligten Variablen die entsprechenden Informationen zu ermitteln. Die detailierte Vorgehensweise hängt dabei stark von der Darstellung der Schleifen auf der LIR Ebene ab. Da diese Darstellung zum Teil stark zwischen den verwendeten LIR Darstellungen variiert, verweisen wir den Leser auf das Kapitel 5.4 dieser Diplomarbeit, in dem die Erkennungen von Schleifen anhand der *GeLIR* durchgeführt wird.

5.3.2 Erkennen von Abhängigkeiten zwischen Arrayzugriffen

Um eine stets gültige Berechnung nach einer Vektorisierung zu erhalten, muß im Vorfeld überprüft werden, ob schreibende Speicherzugriffe durch Vektorbefehle Speicherbereiche

überschreiben, welche zu dem Zeitpunkt der Iteration noch nicht hätten verändert werden dürfen.

Das Beispiel in Abbildung 5.1 vedeutlicht die Problematik. In diesem Beispiel werden in der zweiten Schleife der Funktion Ergebnisse der $i-1ten$ Berechnung in der $iten$ Berechnung verwendet. Somit existiert eine Abhängigkeit zwischen den Arrayzugriffen mit der Distanz 1.

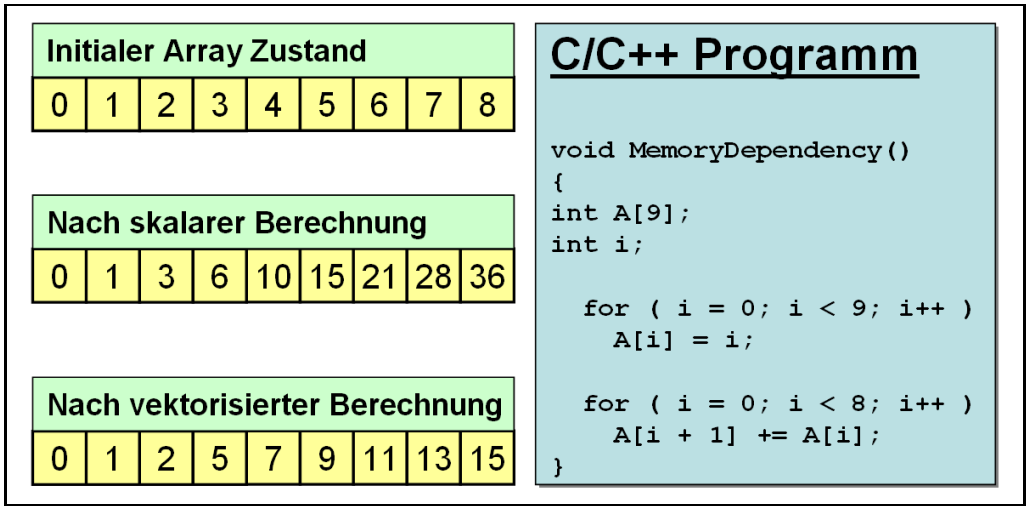


Abbildung 5.1: Beispiel Speicherabhängigkeiten

Für die Erkennung solcher Abhängigkeiten reicht eine einfache Distanzberechnung zwischen den einzelnen Arrayzugriffen nicht mehr aus. Der Grund dafür liegt darin, daß Abhängigkeiten zwischen verschiedenen Arrayzugriffen über Schleifengrenzen hinweg getragen werden können. Es ist somit der Einsatz einer Array-Datenflußanalyse notwendig. Zur Durchführung dieser Analysen kann eine der in Kapitel 4 vorgestellten Analysemethoden verwendet werden.

Bedingt durch die in C/C++ vorhandene Zeigerarithmetik ist eine einfache Abhängigkeitsanalyse über Indizes nicht mehr möglich. Vielmehr muß eine Modifikation der Datenflußanalysen erfolgen, um eine Abhängigkeitsanalyse zu ermöglichen, welche die Adressen

der Speicherzugriffe betrachtet.

Beispiel 5.3.2 *Speicherbasierte Abhängigkeit*

```
void MemoryDependencyFunction()
{
int  A[16];
char *pA;

  pA  = (char*) &A;
  pA += 3;

  *pA = 0;
}
```

Das Beispiel 5.3.2 stellt einen dieser Fälle dar. In diesem Fall greifen die beiden Variablen A und pA auf den gleichen Speicherbereich zu, obwohl es sich um zwei unterschiedliche Variablen unterschiedlichen Typs handelt.
Zu diesem Zweck muß die jeweilige Analysemethode durch die folgenden Punkte modifiziert werden:

- Die Abfrage, daß Speicherzugriffe von zwei unterschiedlichen Variablen immer unabhängig sind, muß an die in C/C++ vorhandene Zeigerarithmetik angepaßt werden. Bevor eine Analyse durchgeführt werden kann, müssen im Vorfeld die Adressen der Zeigervariablen bestimmt werden. Anhand dieser Adressen kann dann eine Zeigervariable der entsprechenden Array Variable zugeordnet werden.

- Die Basisadressen der Arrays müssen in den Indextermen, durch welche die Adreßberechnung des Speicherzugriffs durchgeführt wird, mit eingebunden werden. Sind diese Adressen zum Zeitpunkt der Analyse nicht bekannt, so muß ein künstliches Speicherlayout geschaffen werden. Dabei kommt es nur auf die Adresse und die Größe der einzelnen Variablen an. Die Reihenfolge der Variablen würde nur in dem Fall eine Rolle spielen, indem über den Bereich einer Variable hinaus der Speicher referenziert werden würde. Da dies eine Verletzung der Programmierrichtlinien und somit einen Fehler des Programmes darstellt, muß dies nicht weiter betrachtet werden.

- Die Größe der Elemente innerhalb eines Arrays muß in die Indexterme mit aufgenommen werden.

Das Ergebnis der Analyse liefert dann nicht mehr die Anzahl der Iterationen, in der ein Wert seine Gültigkeit behält. Stattdessen erhält man die maximale Distanz in Bytes zwischen zwei abhängigen Speicherzugriffen.

Nach der Berechnung der Distanzen zwischen Arrayzugriffen muß nun die Frage gestellt, wann eine Schleife vektorisiert werden kann und wann nicht. Ob eine Schleife vektorisiert werden kann hängt dabei von der Distanz zwischen zwei Arrayzugriffen zum einem und von der Laufrichtung der Schleife zum anderen ab.
Zur Fallunterscheidung sei nun G die Größe der Gruppenregister des Prozessors und D die Distanz zwischen zwei Speicherzugriffen in Byte.
Weiterhin werden die folgenden Konstanten verwendet:

&A1, &A2 Basisadressen der jeweiligen Arrays.

s1, s2 Bezeichnen die konstante Größe der Arrayelemente.

d1, d2 Bezeichnen einen beliebigen konstanten Wert.

I Mit I wird der Iterationswert der Schleife bezeichnet.

Fall 1: $G \geq D$
In diesem Fall ist eine Vektorisierung ohne weiteres möglich, da es zu keinem überlappenden Speicherzugriff bzgl. der Speicherzugriffe der Gruppenregister kommt.

Fall 2: $G < D$
In dem Fall, daß der Abstand zwischen zwei Speicherzugriffen kleiner ist als die Größe des Vektorregister, sind drei Unterscheidungen zu treffen. Dazu sei I der konstante Wert, welcher die Schleifeninduktionsvariable i mittels $i = i + I$. verändert. Weiterhin seien *pA1* und *pA2* die Adressen zweier Speicherzugriffe, welche mittels der Terme $pA1 = \&A1 + i * s1 + d1$ und $pA2 = \&A2 + i * s2 + d2$ berechnet werden. Den Variablen und Konstanten innerhalb der Terme kommt dabei folgende Bedeutung zu:

Die Speicherzugriffe erfolgen in der Reihenfolge **pA1 *pA2*, d.h. das zuerst mittels des Zeigers **pA1* und danach mittels des Zeigers **pA2* auf den Speicher zugegriffen wird. Dabei stellt **pA1* einen lesenden oder schreibenden und **pA2* einen schreibenden Speicherzugriff dar.
In den folgenden drei Fällen ist eine Vektorisierung weiterhin möglich:

Fall 2.1: $pA1 = pA2$
In dem Fall das die Distanz zwischen zwei Arrayzugriffen Null ist kommt es zu keinem überlappenden Speicherzugriff. Eine Vektorisierung ist somit ohne Betrachtung der Laufrichtung der Schleife möglich.

Beispiel 5.3.3 *Schleife mit der Eigentschaft* $pA1 = pA2$

```
void LoopFunction()
{
int A[16];
int i;

  for ( i = 0; i < 16; i++ )
      A[i] = A[i] + 1;
}
```

Fall 2.2: $pA1 < pA2 \wedge I < 0$
Die Elemente werden hinter den gelesenen Elementen gschrieben. Da die Schleife rückwärts läuft werden keine geschriebenden Elemente wieder referenziert. Somit besteht keine Abhängigkeit zwischen den Speicherzugriffen.

Beispiel 5.3.4 *Schleife mit der Eigentschaft* $pA1 < pA2 \wedge I < 0$

```
void LoopFunction()
{
int A[16];
int i;

  for ( i = 14; i >= 0; i-- )
      A[i + 1] = A[i] + 1;
}
```

Fall 2.3: $pA1 > pA2 \wedge I > 0$
Dies entspricht dem Fall 2.2, nur daß die Schleife in die entgegengesetzte Richtung läuft.

Beispiel 5.3.5 *Schleife mit der Eigentschaft* $pA1 > pA2 \wedge I > 0$

```
void LoopFunction()
{
int A[16];
int i;

  for ( i = 0; i < 15; i++ )
      A[i] = A[i + 1] + 1;
}
```

5.3.3 Gültigkeit von Indextermen

Nach der Analyse der Speicherabhängigkeiten ist es notwending die daraus entstandenen Indexterme genauer zu betrachten. Es muß dabei untersucht werden, ob die Berechnung der Speicheradressen eine Vektorisierung zulassen oder nicht. Dabei hängt die Gültigkeit der Indexterme von dem verwendeten Codegenerator und von den Eigenschaften des unterstützten Prozessors ab. Weiterhin kommt als Hauptkriterium die Linearität der Speicherzugriffe hinzu, welche sich durch die verwendeten Indexterme und Variablen ergeben, da bei nichtlinearen Speicherzugriffen eine Vektorisierung nicht möglich.

Beispiel 5.3.6 *Gültige und ungültige Indexterme*

```
for ( i = 0; i < 16; i++ )
{
  x = A[i];
  B[i] = x;
}
```

Beispiel a)

```
for ( i = 0; i < 16; i++ )
{
  x = A[i + 3];
  B[i + 0] = x;
}
```

Beispiel b)

```
for ( i = 0; i < 16; i++ )
{
  x = A[i * 5];
  B[i] = x;
}
```

Beispiel c)

```
for ( i = 0; i < 16; i++ )
{
  x = A[-1 * i + 15];
  B[i] = x;
}
```

```
        Beispiel d)


for ( i = 0; i < 16; i++ )
{
  x = A[(i * i) & 15];
  B[i] = x;
}

        Beispiel e)
```

Zur besseren Verdeutlichung der Problematik werden die einzelnen Fälle anhand der Beispiele 5.3.6 a) bis e) erläutert. Dabei bezeichnet A ein Array vom Typ *int*, wobei der Datentyp *int* eine Größe von 16-Bit (2 Byte) besitzt.

Fall a)
Im Fall a) ergibt sich der Indexterm zu $\&A + i * 2$. Da die For-Schleife von 0 bis 15 läuft, wird vorwärts linear über das Array gelesen. Dies stellt einen Spezialfall dar, indem keine speziellen Behandlungen durch den Codegenerator erforderlich sind.

Fall b)
Der Fall b) stellt ein Problem für Prozessoren dar, welche nicht direkt auf jede beliebige Speicheradresse zugreifen können (wie es bei den Prozessoren der *M3-DSP* Plattform der Fall ist). Um solche Fälle handhaben zu können, muß der Codegenerator die nicht zu verwendenden Daten maskieren. Dies sorgt vor allem bei der Verwendung der Werte in anderen Operationen für Probleme. In dem Fall b) könnte der *M3-DSP* Prozessor nur 13 anstatt der üblichen 16 Werte innerhalb der Gruppenregister verwenden. Beim Schreiben in das Array B würden somit 3 Elemente zuwenig kopiert. Es bietet sich in diesem Fall an, jeweils zwei SIMD Lese- und Schreiboperationen durchzuführen. In dem Fall, daß weitere Abhängigkeiten zu anderen Operationen vorhanden sind, kann durch die steigende Komplexität eine Vektorisierung verhindert werden.

Fall c)
Auch wie Fall b) stellt dieser Fall den Codegenerator vor Probleme. Durch den verwendeten Faktor 5 wird nur jedes 5. Element des Arrays gelesen. Dabei entspricht die entstehenden Problematik für den Codegenerator dem Fall b). Ein weiteres Problem entsteht, wenn der verwendete Faktor des Indexterms nicht ein Vielfaches der Datenpfade innerhalb des Prozessors entspricht. In diesem Fall kommt es zu einer asymetrischen Maskierung der nicht verwendeten Elemente innerhalb der Gruppenregister. Bei dem Arrayzugriff $A[i*5]$ würden nach einer Vektorisierung in der ersten Iteration der Schleife die Elemente $\{0, 5, 10, 15\}$ verwendet wohingegen in der zweiten Iteration die Elemente $\{4, 9, 14\}$ verwendet werden würden. Eine Anpassung des Codegenerators an diesen Fall ist kaum noch möglich. In diesem Fall kann durch ein umkopieren der Elemente in ein temporäres Array, eine Vektorisierung ermöglicht werden. Dabei muß abgeschätzt werden, ob der entstehende Overhead durch den Gewinn der Vektorisierung ausgeglichen werden kann.

Fall d)
Da die Elemente des Arrays A Rückwärts gelesen werden, müssen diese im Falle einer Vektorisierung in der umgekehrten Reihenfolge in den Gruppenregistern zum liegen kommen, wenn diese eine weitere Verarbeitung erforderlich macht. Wird eine Invertierung der Reihenfolge der Elemente innerhalb der Gruppenregister nicht unterstützt, so bietet es sich an, entsprechenden Programmcode dafür einzufügen. Da das Kopieren der Elemente aus einem Array in ein temporäres Array Zeit und Speicher kostet, sollte im Vorfeld untersucht werden, wie häufig auf diese Array zugegriffen wird, ohne das eine Änderung stattfindet. Erst ab zwei oder mehr Zugriffen würde eine solche Veränderungen des Programmcodes einen Vorteil bringen. Findet überhaupt keine Änderung auf diesem Array statt, so kann das Umkopieren aus den Schleifenkörper gezogen werden, was zu einer weiteren Verbesserung der Optimierung führt.

Fall e)
Der Fall e) stellt einen nicht vektorisierbaren Arrayzugriff dar. Durch den Ausdruck $(i * i)\&15)$ kommt es zu einer fast zufälligen nichtlinearen Berechnung der Speicherzugriffe. Wie in den anderen Fallen kann dieses Problem durch das Kopieren der einzelnen Elemente innerhalb der Schleife in ein temporäres Array gelöst werden.

Damit im Falle der Arrayzugriffe überhaupt ein linearer Speicherzugriff gegeben ist, müssen die Indexterme die folgendende Form besitzen: $idx = a \times i + b \pm f$. Dabei bezeichnen a und b jeweils zwei Konstanten und i eine Variable, welche sich innerhalb der Schleife ändert. Die Funktion f stellt dabei einen beliebigen Term dar, welcher wiederum Konstanten und Variablen enthalten kann. Damit die Affinität des Indexterms idx gewahrt bleibt, dürfen sich die Variablen innerhalb der Funktion f in der zu analysierenden Schleife nicht verändern. Dadurch verhält sich die Funktion f konstant und verhindert somit nicht eine Vektorisierung, wenn die Funktion f durch die arithmetischen Operationen $+$ und $-$ mit dem Rest des Indexterms verbunden sind.

Beispiel 5.3.7 *Konstante funktion f*

```
for ( i = 0; i < 16; i++ )
{
  for ( j = 0; j < 16; j++ )
  {
    A[j + i * 4 ] = 0;
  }
}
```

In dem Beispiel 5.3.7 ergibt sich der entsprechende Indexterm zu $idx = \&A + j * 2 + i * 8$ und die Funktion f zu $f = i * 8$. Da i innerhalb der Schleife von j nicht verändert wird, ist der Wert von f innerhalb der Schleife j konstant und kann ignoriert werden. Damit ergibt sich der zu analysierende Index zu $idx = \&a + i * 2$, wodurch ein linearer Zugriff auf das Array A gegeben ist.

Nach der Überprüfung der Indexterme müssen diese von der Vektorisierungseinheit daraufhin überprüft werden, wieviele Elemente pro Vektorzugriff maximal verarbeitet werden können, wenn die unterstütze Prozessorarchitektur ein Maskieren von nicht verwendeten Werten unterstütz. Diese Informationen sind notwendig, um die Schrittweite der Schleife anpassen zu können, wobei die niedrigste Zahl den Faktor zur Anpassung der Schleifenschrittweite darstellt.

Beispiel 5.3.8 *Bestimmung der maximalen parallelen Vektorzugriffe*

```
for ( i = 0; i < 16; i++ )
{
  x = A[i * 2];
  B[i] = x;
}
```

In dem Beispiel 5.3.8 wird aus dem Array A jedes zweite Element gelesen. Somit können bei 16 Datenpfaden maximal 8 Elemente pro Iteration parallel verarbeitet werden. Da der Zugriff auf das Array B 16 Elemente pro Iteration zuläßt, beträgt der maximale Faktor zur Anpassung der Schleifenschrittweite 8. Damit ergibt sich das Beispielprogramm 5.3.8 zu folgendem vektorisierten Programm:

Beispiel 5.3.9 *Vektorisierte Schleife*

```
for ( i = 0; i < 16; i += 8 )
{
  x = SimdLoad(&A + i * 2, 2);
  SimdStore(&B + i, 1, x);
}
```

5.3.4 Kontrollflußabhängigkeiten

Kontrollflußabhängigkeiten entstehen, wenn innerhalb von Verzweigungen in der Schleife Speicherzugriffe auf die zu analysierenden Array-Variablen stattfinden. Weiterhin muß überprüft werden, ob durch den verzweigten Kontrollfluß unterschiedliche Indexterme zur

Adreßberechnung von Arrayzugriffen entstehen.

Beispiel 5.3.10 *Kontrollflußabhängigkeiten*

```
void Function()
{
int i;
int* A[16];

  for ( i = 0; i < 16; i ++ )
  {
    if ( i & 1 )
      A[i*2] = 10;
    else
      A[i*4] = 10;
  }
}
```

Fall a)

```
void Function()
{
int i, j;
int* A[16];

  for ( i = 0; i < 16; i ++ )
  {
    if ( i & 1 )
      j = i * 2;
    else
      j = i * 4;
  }

  A[j] = 10;
}
```

Fall b)

Wie in dem Beispiel 5.3.10 gut zu erkennen ist, wird je nach Kontrollfluß eine andere Adresse des Arrays A verändert. Da durch eine statische Analyse nicht entschieden werden kann, wie der Kontrollfluß verläuft, ist eine Analyse nicht mehr möglich wenn die Adreßberechnungen innerhalb der Verzweigungen unterschiedlich sind. Aus diesem Grund muß die verwendete Datenflußanalyse an diese Fälle angepaßt werden.

Fall a)
Für den Fall a) genügt es festzustellen, daß auf unterschiedlichen Kontrollflußwegen unterschiedliche Arrayzugriffe stattfinden. Dies kann z.B. nach der Bestimmung der Indexterme geschehen, indem diese miteinander verglichen werden. In dem Fall von unterschiedlichen Indextermen kann die Analyse direkt abgebrochen werden, da eine Vektorisierung der Speicherzugriffe nicht möglich ist.

Fall b)
Im Fall b) bedarf es einer erweiterten Form der Indextermbestimmung. Dazu muß der Kontrollfluß über alle möglichen Wege zurückverfolgt werden. Dabei werden alle Variablen innerhalb der Indexterme rekursiv durch ihre Zuweisungen ersetzt. Entstehen in diesem Schritt mehrere unterschiedliche Indexterme pro Speicherzugriff, so muß auch an dieser Stelle das Verfahren abgebrochen werden, da eine eindeutige Bestimmung der Speicherzugriffe ebenfalls nicht mehr möglich ist.

Um trotzdem eine Vektorisierung durchführen zu können, kann mittels der Optimierung *Loop Unswitching* (Kapitel 6.6) versucht werden, die Kontrollflußverzweigung aus der

Schleife zu entfernen.

5.3.5 Abhängigkeiten durch C/C++ Zeigerarithmetik

Die in C/C++ vorhandene Zeigerarithmetik stellt für viele Optimierungen ein Problem dar. Durch Zeiger können Speicherbereich beschrieben werden, welche zu einer anderen Variablen gehören. Die daraus entstehenden "Alias" Effekte können bei Nichtbeachtung zu fehlerhaften Optimierungen führen.
Dazu sei das folgende Beispiel betrachtet, indem eine *Dead-Code* Elimination durchgeführt wird.

Beispiel 5.3.11 *Fehlerhafte Dead-Code Elimination*

```
Vor Dead-Code Elimination
-------------------------

void DeadCodeFunction()
{
int i;
int* A[10];

  A[2] = &i;
  i = 5;

  return *A[2];
}
```

```
Nach Dead-Code Elimination
--------------------------

void DeadCodeFunction()
{
int i;
int* A[10];

  A[2] = &i;


  return *A[2];
}
```

Eine einfache Datenflußbasierte Dead-Code Elimination würde in diesem Fall eine fehlerhafte Optimierung liefern, da die Variable i als "Dead" erkannt worden wäre. Auch bei anderem Optimierungen wie der Vektorisierung können solche Abhängigkeiten durch Zeiger zu einer fehlerhaften Analyse und somit zu einer ungültigen Optimierung führen. Es muß somit im Vorfeld der Optimierung überprüft werden, ob Zeiger innerhalb einer Funktion verwendet werden. Ist dies der Fall, so muß bestimmt werden auf welche Variablen diese Zeiger verweisen. Ist dies nicht eindeutig möglich, so muß ein Teil oder ggf. die gesamte Optimierung unterbunden werden.

Für die Vektorisierung ergeben sich dabei folgende Einschränkungen bzgl. der Verwendung von Zeigern, wenn ein schreibender bzw. lesender Zugriff über diese Zeiger erfolgt. Ist einer der genannten Fälle erfüllt, so kann eine Analyse der Speicherabhängigkeiten und somit eine Vektorisierung nicht durchgeführt werden.

- Zeiger, deren Werte nicht bekannt sind verhindern die Analyse von Speicherabhängigkeiten, da nicht bestimmt werden kann, welche Speicherbereiche gelesen

bzw. geschrieben werden. Dies ist z.B. der Fall, wenn Zeiger als Parameter einer Funktion übergeben werden.

- Zeiger, die sich im Laufe der Funktion verändern können nur analysiert werden, wenn die Veränderung durch einen konstanten Wert geschieht. Bei einer Änderung durch eine Variable deren Wert nicht bekannt ist oder deren Änderung nicht nachvollzogen werden kann, ist eine Analyse der Speicherabhängigkeiten ebenfalls nicht mehr möglich.
- Zeiger, die als Werte innerhalb von Arrays oder anderen Listenstrukturen gespeichert oder an Funktionen übergeben werden. Um eine eindeutige Analyse zu ermöglichen, müssen die Wege der Werte von Zeigern statisch eindeutig nachvollzogen werden können. Dies entspricht einer Wertebasierten Analyse von Variablen, welche aus Komplexitätsgründen, nur auf skalaren Variablen durchgeführt werden kann.

Die Art und Weise wie die Wertebasierten Abhängigkeiten überprüft werden sollen, hängt dabei von den zu erreichenden Resultaten der Optimierungen und von den Optimierungen selber ab. Für die Dead-Code Elimination genügt es i. allg. zu überprüfen, ob ein Zeiger auf eine lokale Variable verweist. In diesem Fall würde die Variable als nicht "Dead" eingestuft werden. In dem Beispiel 5.3.11 würde eine solche Überprüfung dazu führen, daß der Ausdruck $i = 5$ nicht entfernt wird.
Bei der Vektorisierung bedarf es allerdings besserer Verfahren, da auch häufig Zeiger zur Adressierung von Arrays verwendet werden. Da eine wertebasierte Analyse von Arrayelementen oder anderen komplexen Strukturen nicht mehr durchzuführen ist, muß die Komplexität der Analyse dadurch eingeschränkt werden, indem nur skalare Variablen bei der Analyse der Zeigerwerte betrachtet werden.

Der folgende Abschnitt stellt nun ein Verfahren vor, mit dem zum einem die wertebasierten Abhängigkeiten von Zeigervariablen ermittelt werden. Zum anderen werden gleichzeitig die durch die Abhängigkeiten entstehenden Terme aufgelöst.

Im ersten Schritt werden alle Ausdrücke innerhalb der Funktion ermittelt, welche Werte einer skalaren Variable zuweisen. Es ist dabei notwendig sich nicht nur auf Ausdrücke mit Zeigervariablen zu beschränken, da es in C/C++ möglich ist, Werte von Zeigervariablen normalen skalaren Variablen (wie int) zuzuordnen. Jeder dieser Ausdrücke wird dann als Knoten in einen Datenflußgraphen eingetragen, wobei jeder Knoten aus drei Elementen besteht, welche im folgenden sind:

- Der Name der Variable, der ein Wert zugewiesen wird.
- Eine Menge von Termen, welche durch das Einsetzen der verwendeten Variablen entstehen
- Eine Menge von Variablen, von der diese Variablen abhängig sind.

Bei der Initialisierung wird der Term, welcher sich durch die Instruktion ergibt, in die Menge der Terme eingefügt. Alle in diesem Term verwendeten Variablen, werden in die Menge der abhängigen Variablen eingetragen. Zur besseren Übersicht sei dazu die Abbildung 5.2 zu betrachten.

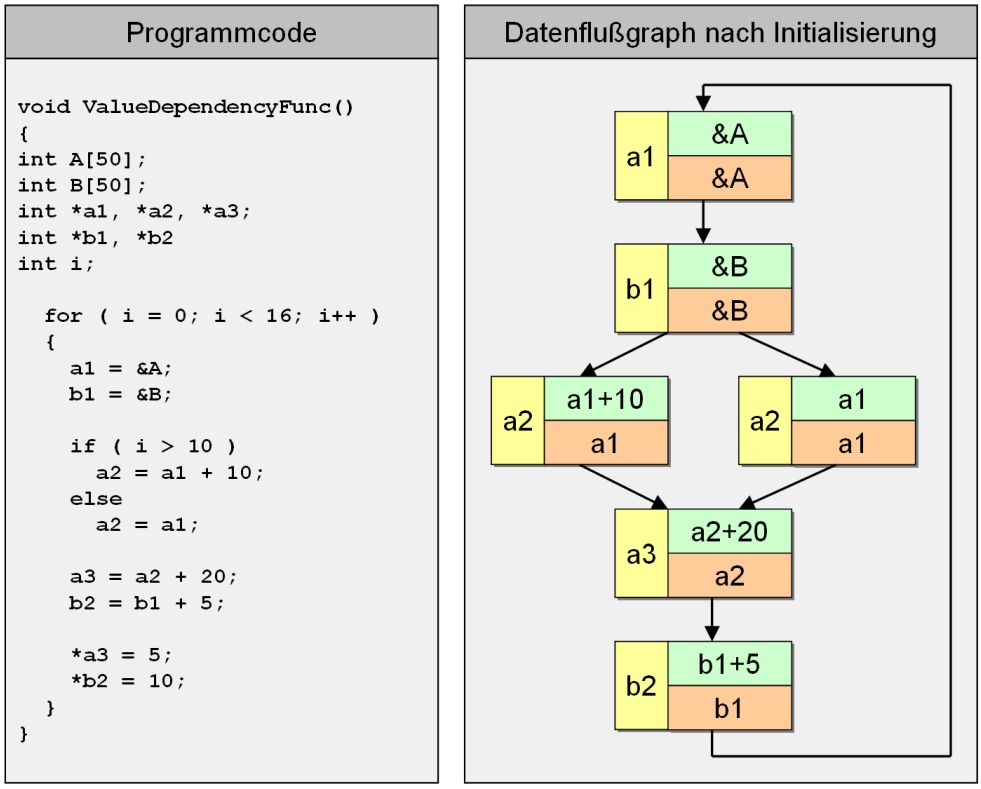


Abbildung 5.2: Beispiel: Datenflußgraph nach Initialisierung

Nach der Initialisierung wird die Datenflußanalyse durchgeführt. Dazu werden die Werte der Knoten vorwärts durch den Graphen propagiert. Bei einer Implementierung bietet es sich an, nicht die Knoten selber sondern eindeutige Indizes oder Zeiger auf die Knotenelemente zu verwenden.
Erreicht ein Knotenelement einen anderen Knoten, so wird überprüft ob dessen Variable, innerhalb der vorhandenen Ausdrücke vorkommt. Ist dies der Fall, so wird die Variable innerhalb des Terms durch den Term des eintreffenden Knotens ersetzt. Tritt der Fall ein, daß mehrere Knoten mit der gleichen Variablen einen Knoten erreichen, so werden entsprechend viele Kopien der Terme angelegt, in denen dann die entsprechende Variable ersetzt werden. Weiterhin wird die Menge aller abhängigen Variablen um die abhängigen Variablen des eintreffenden Knotens ergänzt. Die Abbildung 5.3 verdeutlicht das Verfahren.

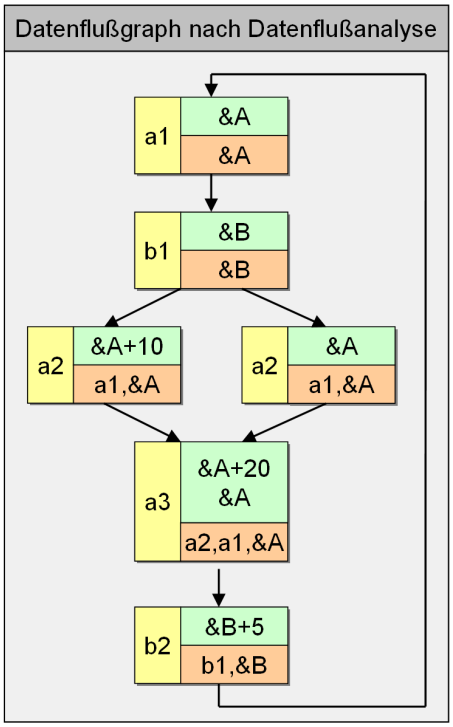


Abbildung 5.3: Datenflußgraph nach Datenflußanalyse

Nach einer gewissen Anzahl von Iterationen stabilisiert sich der Graph. Die Abhängigkeitsmengen enthalten nun zu jeder Variable alle Variablen, die von dieser Variable abhängig sind.

Nachdem die Abhängigkeiten zwischen den Variablen ermittelt worden sind, muß nun festgestellt werden, ob alle Adressen der Zeigervariablen eindeutig bestimmt werden können. Dazu müssen alle ermittelten Variablen die folgenden Bedingungen erfüllen.

- Keine der Variablen darf einem Arrayelement zugewiesen werden, da in diesem Fall eine wertebasierte Abhängigkeitsanalyse nicht mehr möglich ist. Das gleiche gilt für die Übergabe als Referenzen oder als Zeiger an Funktionen oder beim Einfügen in komplexe Datenstrukturen wie Listen oder Bäume. Jeder Fall, in dem eine eindeutige Zielbestimmung des Wertes nicht mehr erfolgen kann, muß abgefangen werden.
- Alle Werte der Zeigervariablen müssen konstant sein oder dürfen sich nur durch die ausgezeichnete Indexvariable der Schleife ändern. Wird eine Veränderung eines Zeigers durch einen konstanten Wert durchgeführt, so muß der ermittelte Term durch einen neuen, von der Indexvariable der Schleife abhängigen Term, ersetzt werden.

- Alle Werte der Zeiger müssen bekannt sein. Werte von Zeigervariable die z.B. als Parameter übergeben werden, können nicht bestimmt werden.

Sind alle diese Punkte erfüllt, so können die Werte der Zeigervariablen anhand der ermittelten Terme berechnet werden. Die konstanten Adressen können dann in der verwendeten Analyse zur Erkennung von Speicherabhängigkeiten verwendet werden. Dazu bietet es sich an, die Adressen der Zeiger den entsprechenden Variablen zuzuordnen, auf die sie zeigen. Dadurch wird eine aussagekräftigere Analyse der Speicherabhängigkeiten erreicht, da keine Abhängigkeiten mehr zwischen unterschiedlichen Variablen vorkommen können.

5.3.6 Abhängigkeiten zu nicht vektorisierbaren Operationen

Zuzüglich zu den Werte- und Speicherabhängigkeiten können auch noch Abhängigkeiten zu anderen nicht vektorisierbaren Operationen vorhanden sein. Dies resultiert in der Tatsache, daß nach der Vektorisierung pro Iteration n anstatt einem Element innerhalb des Arrays verarbeitet werden. Dabei bezeichnet n die Anzahl der Arrayelemente, welche die Datenpfade des entsprechenden Prozessors gleichzeitig verarbeiten können.

Beispiel 5.3.12 *Nicht vektorisierbaren Operationen*

```
Normale Schleife
----------------

void LoopFunction()
{
int A[8];
int i, x = 0;

  for ( i = 0; i < 8; i++ )
  {
    A[i] = 1;
    x = x + A[i];
  }
}

        a)
```

```
Vektorisierte Schleife
----------------------

void LoopFunction()
{
int A[8];
int i, x = 0;

  for ( i = 0; < 8; i += 4 )
  {
    VectorMove A[i], 1
    x = x + A[i];
  }
}

                b)
```

In dem Beispiel 5.3.12 verarbeitet die Vektoreinheit des angenommenen Prozessors vier Integer Werte gleichzeitig. Daraus resultiert, daß die Schleife b) zwei anstatt der acht Iterationen der Schleife a) durchführt. Da der Wert der skalaren Variable x von der Anzahl der Iterationen abhängt, kann eine einfache Vektorisierung, durch setzen der entsprechenden Vektorbefehle bei den Arrayzugriffen, nicht durchgeführt werden. In Fällen von nicht vektorisierbaren Operationen muß somit eine Anpassung des Schleifenkörpers erfolgen. Dabei bieten sich die folgenden zwei Verfahren an:

- **Loop Unrolling**
  Der Schleifenkörper wird n-mal abgerollt, wobei n den Faktor der eingesparten Iterationen wiedergibt.

- **Loop Fission**
  Die Schleife wird in Teilschleifen zerlegt, wobei jede Schleife entweder nur skalare Operationen oder nur Vektor-Operationen enthält.

Beispiel 5.3.13 *Eliminierung von nicht vektorisierbaren Operationen*

```
Loop Unrolling
--------------

void LoopFunction()
{
int A[8];
int i, x = 0;

  for ( i = 0; < 8; i += 4 )
  {
    VectorMove A[i], 1

    x = x + A[i + 0];
    x = x + A[i + 1];
    x = x + A[i + 2];
    x = x + A[i + 3];
  }
}
```

```
Loop Fission
------------

void LoopFunction()
{
int A[8];
int i, x = 0;

  for ( i = 0; < 8; i += 4 )
  {
    VectorMove A[i], 1
  }

  for ( i = 0; < 8; i++ )
  {
    x = x + A[i];
  }
}
```

Bei kleinen Schleifenkörpern und Vektoreinheiten mit geringer Größe bietet sich das *Loop Unrolling* Verfahren an, da dort der Overhead durch weitere Schleifen und deren Initialisierung eingespart wird. In den anderen Fällen, vor allem wenn der Speicherplatzverbrauch des Programmes eine tragende Rolle spielt, sollte dem *Loop Fission* Verfahren der Vorrang gegeben werden.

5.3.7 Abhängigkeiten zu anderen Optimierungen

Die Möglichkeit der Vektorisierung einer Funktion hängt von der Güte der durchgeführten Analysen ab. Die für diesen Zweck zum Einsatz kommenden Analysenmethoden sind durch ihre statische Analyse dabei stark von dem verwendeten Programmcode abhängig. Somit sind Optimierungen, welche vor der eigentlichen Vektorisierung durchgeführt werden, mit bedacht zu wählen.
So wie es eine Reihe von Optimierungen gibt, welche eine Vektorisierung unterstützenden

können (*Loop-Unrolling*, *Loop-Fission* oder *Loop-Strength-Reduction*), so können Optimierungen eine Vektorisierung verschlechtern oder sogar gänzlich verhindern. Als Beispiel sei dazu das *Register-Pipelining* (Rüdiger Schwarz [RS00]) zu nennen. Register Pipelining ist eine erweiterte Form der *Redundant Load Elimination* (Lars Hornbach [LH01]), welche den Code so optimiert, daß wiederverwendete Werte von Arrayzugriffen auch über Schleifengrenzen hinweg in Registern zwischengespeichert werden können. Durch diese Optimierung wird eine Schleife mit einer Reihe von weiteren skalaren Operationen versehen, die eine Vektorisierung verhindern können und somit durch andere Optimierungen (z.B. Loop-Fission) wieder entfernt werden müssen.

Beispiel 5.3.14 *Register Pipelining*

```
Ursprüngliche Funktion
----------------------

void LoopFunction()
{
int A[16];
int i, x;

  A[0] = 0;

  for ( i = 0; < 16; i++ )
  {
    x = A[i];
    .
    .
    .
    A[i + 1] = i;
  }
}
```

```
Nach Register Pipelining
------------------------

void LoopFunction()
{
int A[16];
int i, t1;

  t1 = 0;

  for ( i = 0; < 16; i++ )
  {
    x = t1;
    .
    .
    .
    A[i + 1] = i;
    t1 = i;
  }
}
```

Wie im Beispiel 5.3.14 zu erkennen ist, werden Arrayzugriffe innerhalb der Schleife entfernt, welche für eine Vektorisierung hätten verwendet werden können. Somit wirkt sich ein vorgezogenes Register-Pipelining negativ auf eine mögliche Vektorisierung aus. Im Gegensatz dazu kann ein nach der Vektorisierung durchgeführtes Register-Pipelining zur Verbesserung des Programmcodes beitragen. Dazu sollte der verwendete Prozessor über eine große Anzahl von Gruppenregistern verfügen.

Welche Optimierungen eine Vektorisierung unterstützen oder behindern können hängt wie Eingangs erwähnt von den Optimierungen aber auch von der Reihenfolge und der Kombination dieser Optimierungen ab. Um eine möglichst hohe Anzahl von vektorisierbaren

Schleifen zu erhalten, sollten zu diesem Thema weitere Untersuchungen durchgeführt werden.

5.4 Durchführung der Vektorisierung

Dieser Abschnitt der Diplomarbeit beschreibt nun Detail die Durchführung der Vektorisierung auf der *GeLIR* Ebene. Die Durchführung der Vektorisierung unterteilt sich dabei in folgenden 5 Hauptpunkte, wie sie in Abbildung 5.4 dargestellt sind.

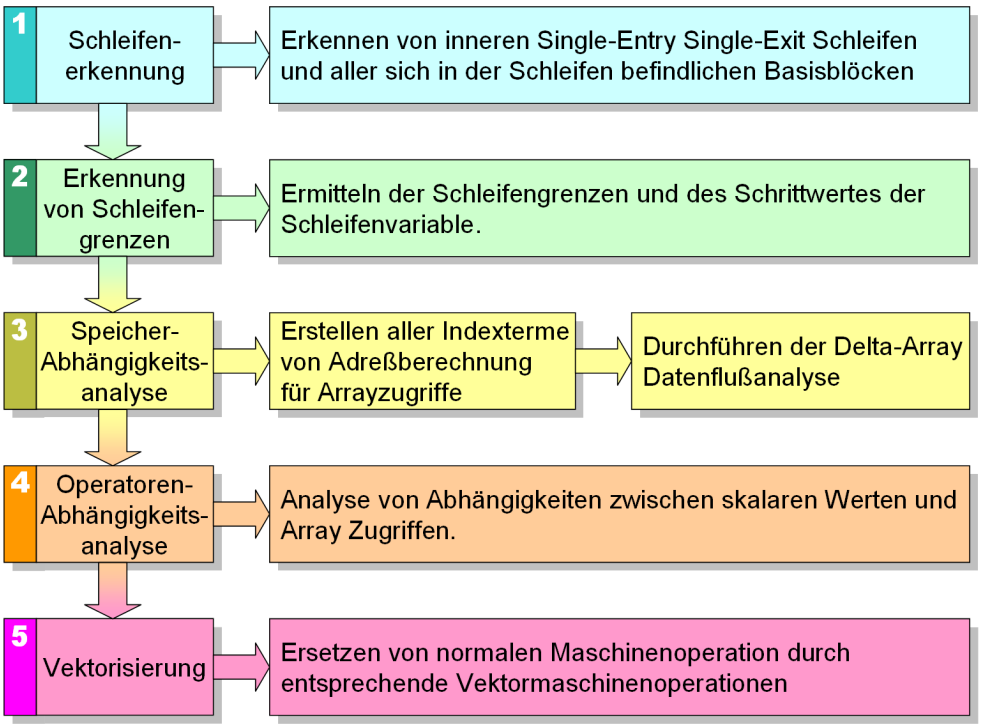


Abbildung 5.4: Durchführung der Vektorisierung

5.4.1 Schleifenerkennung

Im ersten Schritt der Vektorisierung wird eine Funktion auf ihre Schleifen hin analysiert. Die Erkennung der Schleifen erfolgt dabei in drei Schritten. Zuerst werden mittels des in Kapitel 3 vorgestellten Verfahrens zur Erkennung von *Single-Entry Schleifen* alle Schleifen und ihre zugehörigen Basisblöcke identifiziert und in Listen abgespeichert. Im darauffolgenden Schritt wird die Schleife auf ihre *Single-Exit* Eigenschaft hin überprüft, da nur Schleifen mit der *Single-Entry-Single-Exit* Eigenschaft durch das *δ-Array-Datenflußanalyse* Verfahren analysiert werden können. Die Idee zur Erkennung der *Single-Exit* Eigenschaft be-

steht darin, daß nur Sprünge innerhalb des Schleifenkörpers vorhanden sein dürfen, welche als Ziel wiederum Basisbblöcke des Schleifenkörpers haben. Eine Ausnahme bilden dabei Sprünge aus dem Exit Knoten der Schleife heraus, wie sie durch komplexe Abbruchkriterien einer Schleife entstehen. Da in diesen Fällen eine Bestimmung der Schleifengrenzen nicht mehr möglich ist, braucht dieser Fall bei einer Vektorisierung nicht weiter Betrachtet werden. Da die Menge aller Basisblöcke innerhalb einer Schleife bereits vorhanden ist, ist dies durch ein einfaches Prüfen auf das Vorhandensein der Sprungziel-Basisblöcke innerhalb der Menge der Schleifenbasisblöcke möglich. Im letzten Schritt wird die Schleife daraufhin analysiert, ob es sich um eine innere Schleife handelt, da die implementierte Version der *δ-Array-Datenflußanalyse* die Standardimplementierung des Verfahrens darstellt, welches nur auf inneren Schleifen arbeitet. Um dies festzustellen genügt es, die Anzahl der in den Basisblockmengen vorhandenen Rückwärtskanten zu ermitteln. Ist die Anzahl der Rückwärtskanten eins, so handelt es sich bei der vorhandenen Schleife um eine innere Schleife.

Beispiel 5.4.1 *Erkennung von inneren Single-Entry-Single-Exit Schleifen*

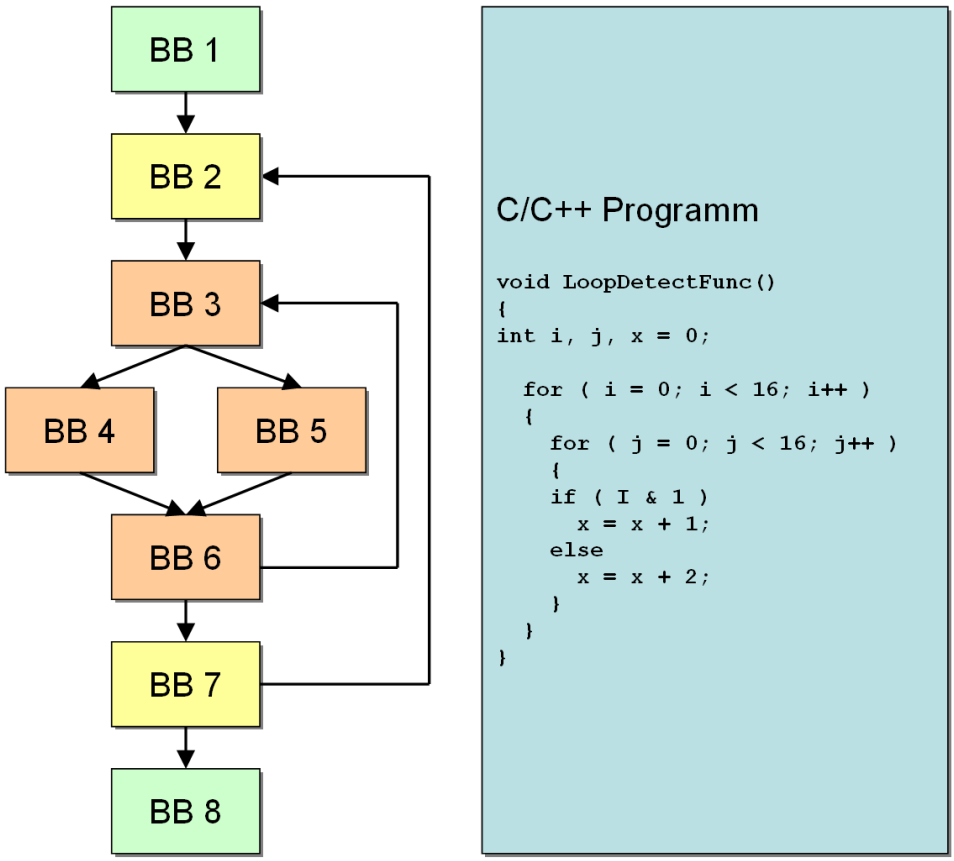


Abbildung 5.5: Beispielschleife

In der Abbildung 5.5 werden im ersten Schritt die einzelnen Schleifen und ihre dazugehörigen Basisblöcke identifiziert. Man erhält somit die beiden Basisblockmengen $Loop1 = \{2, 3, 4, 6, 7, 5\}$ und $Loop2 = \{3, 4, 6, 5\}$.
Im zweiten Schritt werden die Schleifen auf ihre *Single-Exit* Eigenschaft hin überprüft. Dazu wird für jeden Basisblock innerhalb der Mengen überprüft, ob durch vorhandene Sprünge die Schleife verlassen wird. Da dies im oben angegebenen Beispiel für die Menge $Loop1$ noch für die Menge $Loop2$ gilt, erfüllen beide Schleifen die Single-Exit Eigenschaft.
Im letzten Schritt wird die Anzahl der Rückwärtskanten aller Basisblöcke innerhalb der Mengen ermittelt. In der Menge $Loop1$ besitzen die Basisblöcke 6 und 7 jeweils eine Rückwärtskante. Somit ist die Anzahl der Rückwärtskanten zwei und es handelt sich nicht um eine innere Schleife. In der Menge $Loop2$ hingegen besitzt nur der Basisblock 6 eine Rückwärtskante, womit eine innere Schleife ermittelt worden ist.

5.4.2 Ermittlung von Schleifengrenzen

Zur Erkennung der Schleifengrenzen wird ein Verfahren angewendet, das auf dem bereits vorgestellten Verfahren zur Erkennung von Schleifenvariablen aus dem Kapitel 3.5 dieser Diplomarbeit basiert. Die Verfahrensweise wird nun anhand des Beispiels 5.4.2 erklärt.

Beispiel 5.4.2

```
C Programm
----------

void LoopFunction()
{
int i;
int A[16];

  for ( i = 0; i < 16; i++ )
    {
      A[i] = 0;
    }
}
```

```
LIR Darstellung
---------------

void LoopFunction()
{
  i = 0;

loop1:
  t1  = &A;
  t2  = i * 2;
  t3  = t1 + t2;
 *t3 = 0;

  i   = i + 1;
  t4  = i < 16;
  if (t4) goto loop1;
}
```

Folgende Schritte werden zur Erkennung der Schleifengrenze und der Schrittweite der Indexvariable durchgeführt:

1. Lokalisieren des Sprungbefehls.
   Im ersten Schritt wird der Exit-Knoten der Schleife auf Sprungbefehle hin durchsucht. Da nur innere Schleifen mit der *Single-Entry-Single-Exit* Eigenschaft betrachten werden, können Rücksprünge nur in diesem Basisblock der Schleife auftreten. Werden in diesem Basisblock mehrere Sprungbefehle zum Anfang der Schleife gefunden, so handelt es sich dabei um eine Schleife mit mehreren Abbruchkriterien. Die genaue Bestimmung von Schleifengrenzen ist in solchen Schleifen nicht mehr möglich, so daß das Verfahren an dieser Stelle abbricht.

2. Lokalisieren der Indexvariable und der oberen Schleifengrenze.
   In dem Beispiel 5.4.2 wurde der Sprungbefehle *"if (t4) goto loop1;"* gefunden. Der Datenfluß wird nun zum Anfang der Schleife zurückverfolgt bis die Definition der Variable *"t4"* gefunden worden ist. In Schleifen mit einer festen oberen Grenzen, muß einer der Parameter des Vergleichbefehls eine Konstante sein. Der andere Parameter beinhaltet die Indexvariable. Man erhält somit den Befehl *"t4 = i ¡ 16;"*

die Indexvariable *"i"* und die obere Schleifengrenze von 16. Dieser Wert muß später bedingt durch den < Operator mittels der Schrittweite nach unten korregiert werden.

3. Ermittlung der Schrittweite der Indexvariablen.
   Um die Schrittweite einer Indexvariablen genau festellen zu können, müssen die folgenden Bedingungen erfüllt sein:

   (a) Der Wert, der die Indexvariable verändert, muß konstant sein.

   (b) Die Indexvariable darf nur einmal in der Schleife verändert werden.

   Dies ist durch ein einfaches durchsuchen der Basisblöcke innerhalb der Schleife durchführbar, indem alle schreibenden Referenzen der Indexvariable ermittelt werden. Ist einer der angegeben Punkte nicht erfüllt, so bricht das Verfahren in diesem Fall ab. Wir erhalten nun den Befehl *"i = i + 1;"*. Die Schrittweite der Indexvariablen beträgt somit 1.

4. Ermittlung des Startwertes der Indexvariablen.
   Das Ermitteln des Startwertes der Indexvariablen gestaltet sich aufwendiger. Es ist dazu notwendig, vom Schleifeneingang über alle möglichen Kontrollflußwege bis zum Anfang der Funktion zurückzulaufen. Um den Startwert exakt ermitteln zu können, müssen folgende Punkte erfüllt sein:

   (a) Der Startwert der Indexvariable muß konstant sein.

   (b) Die Initialisierung der Indexvariable muß über alle möglichen Kontrollflußwege gleich sein.

   Wir erhalten somit mit der einzigen Definition *"i = 0;"* den Startwert 0.

Nach der Korrektur der oberen Grenze der Schleife wurden folgende Werte ermittelt:

- Startwert: 0
- Zielwert: 15
- Schrittweite: 1
- Iterationen: 16

Es bleibt zum einem zu überprüfen, ob die Lauflänge der Schleife ein Vielfaches der Größe des Gruppenregisters darstellt. Weiterhin müssen wie im Vorfeld beschrieben, die Indexterme dahingehend überprüft werden, on jedes Element des Arrays gelesen wird. Im Hinblick auf den *M3-DSP* Prozessor wäre diese Schleife nun vektorisierbar.

5.4.3 Analyse der Speicherabhängigkeiten

Die Durchführung zur Erkennung von Datenabhängigkeiten ist in folgende zwei Bereiche aufgeteilt

1. Erstellen der Indexterme von Arrayzugriffen.
2. Erstellen des Schleifenkontrollflußgraphen und Durchführen der Analyse.

1 Ermitteln von Indextermen von Arrayzugriffen
Die Berechnung der Indexterme von Arrayzugriffen bassiert ebenfalls wie die Erkennung von Schleifenvariabeln auf dem in Kapitel 3.5 vorgestellten Verfahren. Beginnend von einem Speicherzugriff wird der Kontrollfluß sukzessive zurückverfolgt. Dabei werden Variablen rekursiv durch ihre entsprechenden Terme ersetzt. Um eine stets gültige Berechnung der Indexterme durchführen zu können, muß im Vorfeld eine Analyse über die vorhandenen Variablen innerhalb einer Funktion erfolgen. In dieser Analyse werden Variablen dahingehend klassifiziert, ob sie konstant sind oder sich innerhalb von Schleifen verändern. Diese Vorgehensweise ist nötig, damit sich innerhalb von Schleifen veränderte Variablen nicht ihre konstante Initialisierung ersetzt werden.

Beispiel 5.4.3 *Erkennung von Indextermen von Arrayzugriffen*

```
C/C++ Programm
--------------

void IndexTermFunction()
{
int A[16];
int i;

  for ( i = 0; i < 16; i++ )
      A[i] = 0;
}
```

```
IR-Darstellung
--------------

void IndexTermFunction()
{
int A[16];
int i, t1, *t2, *t3, t4;

  i   = 0;

Loop:
  t1  = i * 4;
  t2  = &A;
  t3  = t1 + t2;
  *t3 = 0;

  i   = i + 1;
  t4  = i < 16;
  if (t4) goto Loop;
  }
```

Angewendet auf das Beispiel 5.4.3 ergibt sich dabei folgende Ausführungsreihenfolge:

1. Der Ausdruck **t3 = 0* wird in der *GeLIR* als Speicherzugriff auf ein Array gesondert gekennzeichnet. Somit stellt eine Erkennung dieser Terme kein Problem dar und die Variable *t3* dient als Ausgangspunkt zur Erstellung des Indexterms. Somit gilt: $IndexTerm = t3$.

2. Beginnend von *t3 = 0* wird nun der Kontrollfluß zurückverfolgt bis eine Definition von *t3* gefunden worden ist. Im Beispiel wird der der Ausdruck *t3 = t1 + t2* lokalisiert und in den Indexterm eingetragen, womit sich dieser zu $IndexTerm = (t1 + t2)$ ergibt. Optional besteht die Möglichkeit zur Generierung der Indexterme den in den Basisblöcken vorhandenen Datenflußgraphen zu verwenden.

3. Nun wird die Variable *t1* betrachtet und nach dem angegebenen Verfahren ausgewertet. Der Indexterm wird zu: $IndexTerm = ((i * 4) + t2)$

4. Im vorletzten Schritt wird *t2* analysiert. Da es sich bei um eine Adreßzuweisung des Arrays handelt, wird die entsprechende Adresse des Arrays eingetragen. In dem Fall das die Adresse des Arrays 1000 beträgt ergibt sich somit der Indexterm: $IndexTerm = ((i * 4) + 1000)$

5. Die einzige Variable innerhalb des Indexterms ist nun die Variable *i*. Da diese im Vorfeld durch eine Analyse als Schleifenvariable erkannt worden ist, bricht das Verfahren an dieser Stelle ab.

Der Indexterm ist somit berechnet und kann für die δ-Array Datenflußanalyse verwendet werden.

2 Erstellen des Schleifenkontrollflußgraphen und Durchführen der Analyse
Zum Durchführen der δ-Array Datenflußanalyse muß der zu analysierende Teilkontrollflußgraph der Schleife in einen wie in Kapitel 4.4.1 beschriebenen Schleifenkontrollflußgraphen (LCFG) umgewandelt werden.
Die Idee hinter dem als *Split & Merge* bezeichneten Verfahren basiert darin, zuerst für jede Operation innerhalb des Graphen einen eigenen Knoten anzulegen (*Split* Schritt). Danach werden alle nicht zur Analyse gehörenden Knoten des Graphen sukzessive entfernt, wobei alle Vorgängerknoten mit den Nachfolgerknoten verbunden werden. Um im Abschluß einen gültigen Schleifenkontrollflußgraphen zu erhalten, wird vor dem *Split & Merge* Schritt der spezielle *Exit* Knoten des Schleifenkontrollflußgraphen eingefügt.

Beispiel 5.4.4 *Erstellen eines Schleifenkontrollflußgraphen*

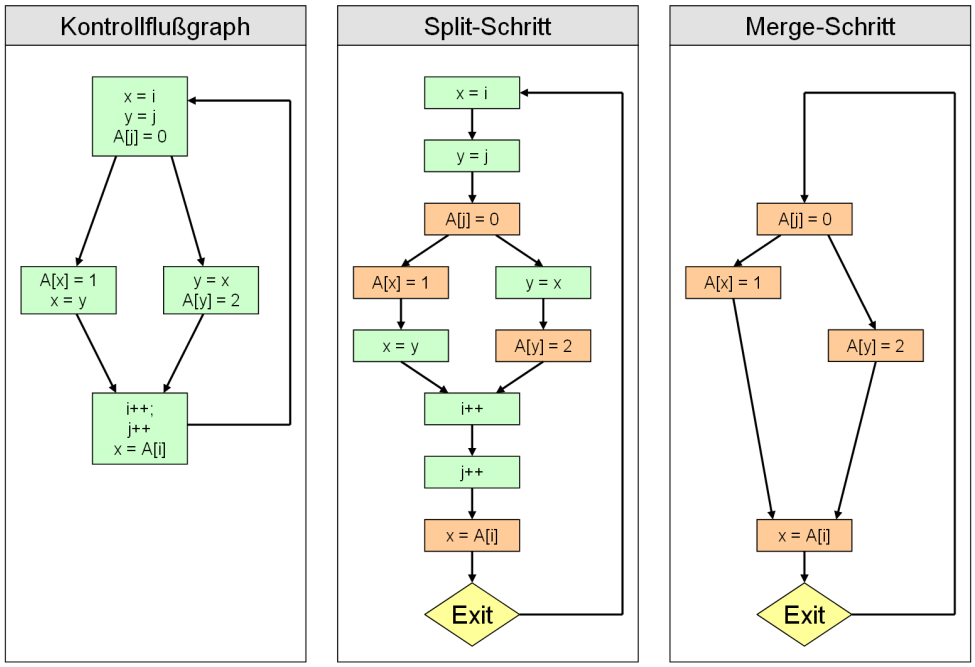


Abbildung 5.6: Erstellen eines Schleifenkontrollflußgraphen

Die Abbildung 5.6 zeigt die Vorgehensweise im Detail. Zuerst wurden die vier Basisblöcke zerteilt, so daß jede Operation einen eigenen Knoten innerhalb des Schleifenkontrollflußgraphen erhält. Danach wurden alle nicht für die δ-Array Datenflußanalyse relevanten Operationen entfernt. In dem angegebenen Beispiel wurde zuerst die Operation $x = i$ entfernt und die Kante des *Exit* Knotens wurde mit dem Knoten der Operation $y = j$ verbunden. Das Verfahren wird solange durchgeführt, bis keine irrelevanten Knoten mehr vorhanden sind. Das Ergebnis ist dann der für die δ-Array Datenflußanalyse benötigte Schleifenkontrollflußgraph.

Nach dem Erzeugen des Schleifenkontrollflußgraphen werden die im ersten Schritt ermittelten Indexterme den im Schleifenkontrollflußgraph befindlichen Knoten zugeordnet. Die Zuordnung der Indexterme zu den entsprechenden Knoten im Graphen geschieht durch eine eindeutige ID, welche sich aus den Zeigern der entsprechenden LirMO Objekte ergibt.
Die Erstellung der Ergebnis-Matrix und das Durchführen der Datenflußanalyse wird von dem implementierten δ-Array Datenflußanalyse Framework automatisch durchgeführt. Als Ergebnis wird eine Matrix zurückgeliefert, welche die maximalen Distanzen bzgl. Speicherzugriffen zwischen allen Arrayzugriffen enthält.

Im Abschluß werden nun die Distanzen der Speicherzugriffe auf ihre Gültigkeit nach den in Kapitel 5.3.2 vorgestellten Bedingungen hin überprüft. Sind alle diese Bedingungen erfüllt, wird mit der Vektorisierung der Schleife fortgefahren.

5.4.4 Analyse der Abhängigkeiten von Operationen

Die im Rahmen dieser Diplomarbeit implementierte Analyse umfaßt aus Zeitgründen nur eine einfache Analyse zur Bestimmung der Abhängigkeiten von Operationen. Zu diesem Zweck werden alle in Vektorbefehlen zusätzlich verwendeten Variablen ausgelesen und alle diese Variablen verwendenden Operationen ermittelt.

Beispiel 5.4.5 *Erkennung von Abhängigkeiten zwischen skalaren und Variablen in Vektorbefehlen*

```
void VariableDependencyFunction()
{
int A[16];
int i, x, y, z;

  for ( i = 0; i < 16; i++ )
  {
    x = A[i];
    y = x + 10;
    z = y + 20;
    A[i] = z;
  }
}
```

In dem Beispiel 5.4.5 stellen die Arrayzugriffe $x = A[i]$ und $A[i] = z$ zwei Vektorbefehle dar. Somit sind die beiden Variablen x und z als von den Vektorbefehlen abhängig. Im weiteren Verlauf der Analyse werden nun die beiden Operationen $y = x + 10$ und $z = y + 20$ gefunden, so daß auch die Variable z von den Vektorbefehlen abhängig ist.

Da eine Veränderung des Schleifenkörpers nicht stattfindet, kann eine Schleife nur vektorisiert werden, wenn für die Verwendung der abhängigen Variablen folgende Bedingungen erfüllt sind:

- Die Veränderung der Variable erfolgt durch einen Vektorbefehl.
- Die Veränderung der Variable erfolgt durch einen konstanten Wert.
- Die Veränderung der Variable erfolgt durch andere Variablen, welche alle ebenfalls von Vektorbefehlen abhängig sein müssen.
- Jede Variable muß in jeder Iteration

- Alle Variablen innerhalb der Schleife müssen von Vektorbefehlen abhängig sein.

Zur Verbesserung der Ergebnisse bietet es sich an, die vorgestellten Verfahren aus Kapitel 5.3.6 zu verwenden, um eine möglichst hohe Anzahl von verktorisierbaren Schleifen zu erhalten.

5.4.5 Setzen der Vektorbefehle

Das Setzen der Vektorbefehle erolgt mittels der in der *GeLIR* verwendeten Überdeckungen von Maschinenoperationen. Jeder abstrakten MO in der *GeLIR* werden ein oder mehrere mögliche verwendbare Maschinenoperationen zugeordnet. So kann z.B. die abstrakte *GeLIR* Operation *LIR_ADD* für die möglichen Maschinenoperationen $\{Add, SimdAdd\}$ stehen.
Dazu wird im ersten Schritt der Codegenerator aufgerufen, welcher für alle vorhandenen Operationen (*LirMO* Objekte) eine Menge von Maschinenoperationen generiert. Die Vektorisierungseinheit durchsucht daraufhin diese Menge nach den entsprechenden Maschinenoperationen, welche für die Vektorisierung verwendet werden müssen. Sind die Maschinenoperationen in der Menge vorhanden, so wird die Auswahl dadurch eingeschränkt, daß alle anderen Maschinenoperationen aus der Menge entfernt werden.

Beispiel 5.4.6 *Überdeckung von Maschinenoperationen*

```
Funktion in              Maschinenoperationen        Maschinenoperationen
LIR Darstellung          vor Vektorisierung          nach Vektorisierung
---------------          --------------------        --------------------

void Function()
{
int A[16];
int i, x, t1;

  i = 0;                 {SimdStore}                 {SimdStore}

  Loop:

    x = A[i];            {SimdLoad}                  {SimdLoad}
    x = x + 10;          {Add, SimdAdd}              {SimdAdd}
    A[i] = x;            {SimdStore}                 {SimdStore}

    i = i - 1;           {Sub, SimdSub}              {Sub, SimdSub}
    t1 = i < 16;         {Compare <}                 {Compare <}

  if (t1) goto Loop;     {JNE}                       {JNE}
}
```

In dem Beispiel 5.4.6 wurden die beiden Arrayzugriffe $x = A[i]$ und $A[i] = x$ und die davon abhängige Operation $x = x + 10$ vektorisiert. Dementsprechend wurden die Mengen der Maschinenoperationen dahingehend verändert, daß nur noch Vektorbefehle von dem Codegenerator verwendet werden können. Es fand dabei eine Veränderung der folgenden Menge statt:

$$\mathbf{x = x + 10} \qquad \{Add, SimdAdd\} \qquad \rightarrow \{SimdAdd\}$$

Nach der Einschränkung der Mengen überprüft der Codegenerator, ob eine Abbildung auf die entsprechenden Maschinenoperationen möglich ist. Ist dies der Fall, so wird der entsprechende vom *M3-DSP* ausführbare Code erzeugt. Die Vektorisierung ist damit abgeschlossen.

Kapitel 6

Weiterführende Optimierungen

In diesem Kapitel werden nun eine Reihe von Optimierungen und Transformationen für Schleifen vorgestellt. Dabei beziehen sich eine Reihe dieser Optimierungen speziell auf die Prozessoren der *M3-DSP* Plattform. Weiterhin werden Optimierungen und Transformationen beschrieben, welche Schleifen in eine für die Vektorisierung verbesserte Form bringen oder sogar eine Vektorisierung erst ermöglichen.

6.1 Zero-Overhead Hardware Loops

Der *M3-DSP* ist ein hochgradig auf Vektorarithmetik hin ausgelegter Prozessor. Speicherzugriffe können nur durch SIMD Befehle erfolgen, so daß selbst bei skalaren Operationen jeweils ein Speicherblock in der Größe der Datenpfade geladen werden muß. Der Overhead, welcher durch die vielen redundanten Speicherzugriffe ensteht, sorgt für eine starke Verringerung der Ausführungsgeschwindkeit von Programmen. Somit ist es sinnvoll, nach geeigneten Maßnahmen zu suchen, diesen Overhead zu reduzieren. Eine Maßnahme besteht darin, für eine Schleife und deren Indexvariable die vom Prozessor zur Verfügung gestellten Hardwareschleifenbefehle zu verwenden.
Die vom *M3-DSP* zur Verfügung gestellte Hardwareschleife ist in ihrem Bereich und ihrer Schrittweite stark eingeschränkt. Der maximale Bereich der Schleife liegt dabei bei 32768 und die Schrittweite beträgt immer -1. Weiterhin können nur maximal 256 Maschinenoperationen in einer Hardwareschleife verwendet werden.

Betrachtet sein nun das folgenden Beispielprogramm, welches eine Schleife in LIR Form

darstellt.

Beispiel 6.1.1 *Schleife in LIR Form*

```
 1 :   void LoopFunction()
 2 :   {
 3 :   int A[16];
 4 :   int i, *t1, *t2, *t3, t4;

 5 :     i = 0;

 6 :   LoopLabel:
 7 :     t1   = &A;
 8 :     t2   = i * 4;
 9 :     t3   = t1 + t2;
10 :     *t3  = 0;

11 :     i    = i + 1;
12 :     t4   = i < 16;
13 :     if (t4) goto LoopLabel
14 :   }
```

Dabei sind die folgenden Maschinenoperationen von essentieller Bedeutung für die Durchführung der Optimierung:

Loop Init MO Diese Maschinenoperation setzt den Startwert der in der Schleife verwendeteten Index Variable *(Zeile 5)*.

Loop Iterator MO Die Loop Iterator Maschinenoperation ist für die Veränderung der Indexvariable zuständig *(Zeile 11)*.

Loop Conditional MO Maschinenoperation, welche den Wert der Indexvariable daraufhin überprüft, ob diese sich im Bereich der Schleife befindet oder nicht. *(Zeile 12)*

Loop Jump MO Bei dieser MO handelt es sich um einen bedingten Sprung, welcher je nach dem Ergebnis aus der Schleifenbedingung MO die Schleife erneut aufruft oder die Schleife verläßt. *(Zeile 13)*

Mittels des in Kapitel 3.5 vorgestellten Verfahren zur Schleifenerkennung werden zuerst die Grenzen und die Schrittweite der Schleife ermittelt. Aus diesen Werten wird dann die Anzahl der Iterationen der Schleife berechnet.

Zur Durchführung der Optimierung werden im folgenden die entsprechenden Hardwareschleifen-Befehle in den Programmcode eingefügt. Dabei ist das Verfahren zur

Schleifenerkennung dahingehen angepaßt worden, daß neben der Schleifengrenze und der Schrittweite der Indexvariable weiterhin alle für die Schleife relevanten Maschinenoperationen ermittelt werden.

Zur Darstellung der Hardwareschleifen-Befehle ist die *GeLIR* Darstellung durch folgende zwei Maschinenoperationen erweitert worden:

ZLoop Dieser Befehle setzt den Wert des internen Schleifenzähler des *M3-DSP* Prozessors. Weiterhin wird ein spezielles Sprunglabel eingefügt, welches von der *ZGoto*-MO verwendet wird.

ZGoto Der *ZGoto* stellt den vom *M3-DSP* zur Verfügung gestellten Hardwareschleifenbefehl dar. Beim Erreichen des Befehls wird automatisch der Schleifenzähler um eins erniedrigt und ein Vergleich auf Null durchgeführt. In dem Fall, daß der Schleifenzähler ungleich Null ist, wird der Sprung durchgeführt. Ansonsten wird die Schleife verlassen.

Die endgültige Optimierung erfolgt letztendlich in drei Schritten:

1. Die Lauflänge der Schleife wird anhand der Schleifengrenzen und dem Wert der Schrittweite der Indexvariable ermittelt.
2. Der *Zloop* Befehl wird am Anfang des Schleifeneingangs-Basisblock eingefügt.
3. Der bedingte Sprungbefehle innerhalb der Schleife wird entfernt und durch den *ZGoto* Befehl ersetzt.
4. Als letzter Schritt sollte eine *Dead-Code Elimination* durchgeführt werden, um durch die Optimierung entstandenen überflüssigen Maschinenoperationen zu entfernen. Dies betrifft vor allem den Schleifen-Kontrollcode, wenn die Indexvariable nicht innerhalb der Schleife verwendet wird.

Das Beispiel 6.1.2 vedeutlicht nun die Vorgehensweise. Dazu wurde der Programmcode aus

Beispiel 6.1.1 verwendet und durch die entsprechenden Hardwareschleifenbefehle ergänzt.

Beispiel 6.1.2 *Schleife nach ZLoop Optimierung*

```
Nach Z-Loop Optimierung                Nach Dead-Code Elimination
-----------------------                --------------------------

void LoopFunction()                    void LoopFunction()
{                                      {
int A[16];                             int A[16];
int i, *t1;                            int *t1;

i = 0;
t1   = &A;                             t1   = &A;

ZLoopLabel:                            ZLoopLabel:
  ZLoopInit(16, ZLoopLabel);             ZLoopInit(16, ZLoopLabel);

  t1   = t1 + 4;                         t1   = t1 + 4;
  *t1  = 0;                              *t1  = 0;

  i    = i + 1;

  ZGoto ZLoopLabel;                    ZGoto ZLoopLabel;
}                                      }
```

Durchführung der Optimierung für Zero-Overhead Hardware Loops

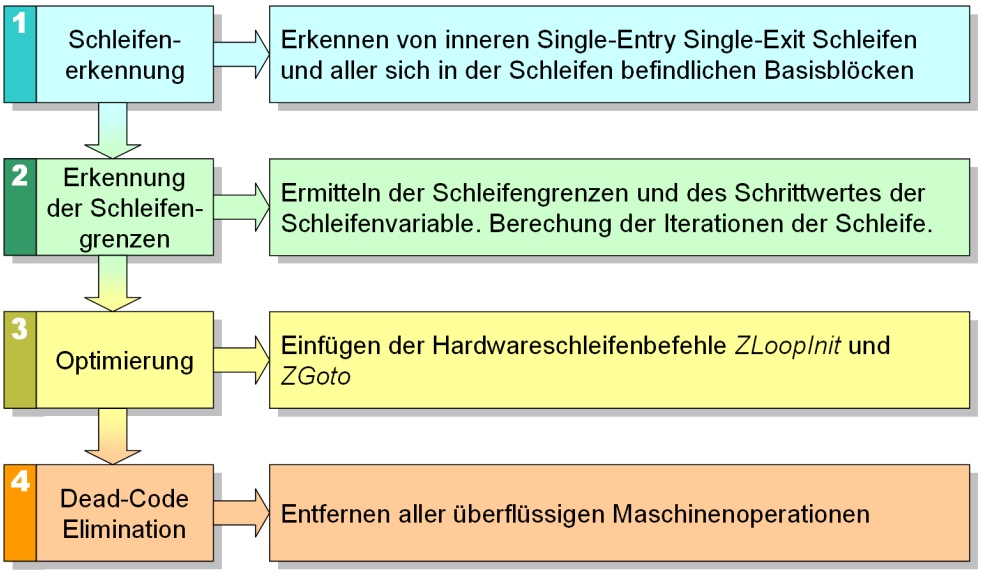


Abbildung 6.1: Durchführung der ZLoop Optimierung

Die in den ersten beiden Schritten zu ermittelnden Informationen werden durch die in der Vektorisierung bereits verwendeten Klasse ermittelt. Die Optimierung beschränkt sich somit nur auf das Einfügen der entsprechenden Hardwareschleifen-Maschinenoperation und der Durchführung der *Dead-Code Elimination*.

Durch die offene Struktur der *GeLIR* und der damit verbundenen Möglichkeit, die Menge der Maschinenoperationen durch prozessorspezifische Maschinenoperationen zu erweitern, stellt sich das Einfügen der Hardwareschleifen-Maschinenoperation als denkbar einfach dar. Anhand der in der Schleifenerkennung lokalisierten Schleifen-Maschinenoperation wird die Position der neuen Maschinenoperation ermittelt. Da das Einfügen der neuen Hardwareschleifen-Maschinenoperation mittels der in der *GeLIR* implementierten Funktionen erfolgt, ist keine Anpassung der *GeLIR* Strukturen an die neuen Maschinenoperation nötig.

Zur Durchführung der *Dead-Code Elimination* wurde der von *Lars Hornbach* [LH01] im Rahmen der Diplomarbeit *Generische Low-Level Optimierungen für RISC-Architekturen* implementierte und auf den *GeLIR* Strukturen arbeitende Algorithmus verwendet.

Zur Codegenerierung wird im Abschluß der am Lehrstuhl für den *M3-DSP* entwickelte Codegenerator verwendet, welcher die neuen Hardwareschleifen-Maschinenoperation in für den *M3-DSP* ausführbaren Code übersetzt.

6.2 Loop Unrolling

Die Optimierung *Loop Unrolling* spielt für den *M3-DSP* eine entscheidende Rolle. Wie bereits vorgestellt ist *Loop Unrolling* essentiell wichtig für die Vektorisierung. Doch auch zur Optimierung von skalaren Speicherzugriffen und der damit verbundenen Reduzierung der Anzahl von Speichertranfers kann *Loop Unrolling* eingesetzt werden, wenn eine Vektorisierung der Schleife nicht möglich ist.

Das Lesen des Speichers erfolgt bei den *M3-DSP* Prozessoren immer über das *RegM* Register, dessen Größe durch die Anzahl der Datenpfade des Prozessors bestimmt ist. Beim *M3-DSP* besitzt dieses Registers eine Größe von 32 Byte, welches in 16 Datenpfade mit jeweils 16-Bit pro Datenpfad aufgeteilt ist. In Schleifen kann dieses zu einem enormen Overhead von Speicherzugriffen führen.
Es ist zu beachten, daß das folgende Beispielprogramm 6.2.1 nur zur Verdeutlichung der Problematik dient und keinen ausführbaren Maschinencode für die *M3-DSP* Prozessoren darstellt.

Beispiel 6.2.1 *Schleife in Pseudo Code*

```
void LoopFunction()
{
int A[64];
int i, *t1, t2, t3, addr, index;

LoopLabel:
  t1    = &A;
  t2    = i >> 4;
  addr  = t1 + t2;
  index = i & 15;
  Load RegM, addr

  t3    = RegM[index];
  t4    = t3 + 1;
  RegA[index] = t4;

  Store RegA, addr

  i     = i + 1;
  t4    = i < 64;
  if (t4) goto LoopLabel
}
```

Pro Iteration wird in der Schleife ein 32 Byte großer Block aus dem Speicher in das *RegM* Regsister kopiert und nach der Durchführung der Addition wieder in den Speicher

geschrieben. Somit werden pro Iteration 64-Byte über den Datenbus transportiert. Bei den im Beispiel angegebenen 64 Iterationen ergibt sich der Speichertransfer zu 4096 Byte.

Um diesen Overhead an Speicherzugriffen zu verringern, bietet es sich an, die Schleife um einen Faktor abzurollen, welcher ein Vielfaches der Anzahl der Datenpfade darstellt. Dadurch wird die Verwendung von speziellen Maschinencode zur Behandlung von assimetrischen Speicherzugriffen über die Größe der Datenpfade hinaus verhindert. Weiterhin sollte die Größe der Schleife eine entscheidene Rolle spielen, damit eine zu starke Vergröße-

rung des Schleifenkörpers vermieden wird.

Beispiel 6.2.2 *4-Fach abgerollte Schleife in Pseudo Code*

```
void LoopFunction()
{
int A[64];
int i, *t1, t2, t3, addr, index;

LoopLabel:
  t1    = &A;
  t2    = i >> 4;
  addr  = t1 + t2;
  index = i & 15;
  Load RegM, addr

  t3    = RegM[index + 0];
  t4    = t3 + 1;
  RegM[index + 0] = t4;

  t3    = RegM[index + 1];
  t4    = t3 + 1;
  RegM[index + 1] = t4;

  t3    = RegM[index + 2];
  t4    = t3 + 1;
  RegM[index + 2] = t4;

  t3    = RegM[index + 3];
  t4    = t3 + 1;
  RegM[index + 3] = t4;

  Store RegM, addr

  i     = i + 4;
  t4    = i < 64;
  if (t4) goto LoopLabel
}
```

Das Beispielprogramm 6.2.2 vedeutlicht nun die vorgehensweise bei der Optimierung. Die Schleife wurde in diesem Beispiel 4-mal abgerollt, so daß sich die Anzahl der Speicherzugriffe auf ein Viertel reduzieren ließen.
Zur einfacheren Übersicht wurde hier das *RegM* Register des *M3-DSP* Prozessors zur verdeutlichung verwendet. Der Leser sollte dabei aber im Auge behalten, daß das *RegM* Register in Wirklichkeit nur beim Transport von Daten aus dem Speicher verwendet wird.

Zur Durchführung der Optimierung bedarf eines erweiterten Verfahrens des *Loop Unrolling*. Im Gegensatz zu den im allgemeinen implementierten Verfahren, welche mit einem festen Wert des Abrollfaktors arbeiten, muß dieses Verfahren die Größe des Programmcodes innerhalb der Schleife, den Schrittwert der Schleife und die Art der Speicherzugriffe berücksichtigen.
Die Größe des Programmcodes ist deshalb von elementarer Bedeutung, da die Größe des Programmspeichers der *M3-DSP* Plattform Prozessoren stark begrenzt ist. Somit kann ein zu häufiges Abrollen von Schleifen dazu führen, daß der entstehende Programmcode nicht mehr in den Arbeitsspeicher des Prozessors paßt.
Dem Schrittweit der Schleife und der Art der Speicherzugriffe kommt dabei eine andere Bedeutung zu. Die Zwischenegister des *M3-DSP* Prozessors werden in dem Fall der Optimierung als eine Art Cache verwendet, um die Anzahl der Speicherzugriffe zu minimieren. Somit ist es notwendig die Speicherzugriffe innerhalb des Arrays daraufhin zu analysieren, inwieweit sich diese lokal verhalten. Damit ist gemeint, ob Speicherzugriffe häufig auf Speicherbereiche zugreifen, welche durch das verwendete Register abgedeckt werden.

Beispiel 6.2.3 *Nicht sinnvoll abrollbare Schleife*

```
void LoopFunction()
{
int A[256];
int i;

  for ( i = 0; i < 256; i += 16 )
  {
    RegM[i + 0] = 0;
  }
}
```

Das im Beispiel 6.2.3 dargestellte Programm bewegt sich mit einer Schrittweite von 32-Byte über das Array[1]. In diesem Fall wäre ein Abrollen der Schleife nicht sinnvoll, da pro Iteration nur einmal auf einen durch das *RegM* Register zwischengespeicherten Speicherbereich zugegriffen wird.

Zur Ermittelung der Anzahl von Speicherzugriffen innerhalb eines durch das *RegM* zwischengespeicherten Speichersegement, müssen die für die Arrayzugriffe verwendeten Indexterme bestimmt werden. Damit eine Gleichverteilung der Zugriffe auf die einzelnen

[1]Der Datentyp *int* stellt einen 16-Bit großen Datentyp dar

Speichersegmente stattfindet, müssen diese Terme in affiner Form vorliegen.

Beispiel 6.2.4 *Nicht affine Arrayzugriffe*

```
void LoopFunction()
{
char A[256];
int i;

  for ( i = 0; i < 16; i++ )
  {
    A[i * i] = 0;
  }
}
```

In der Annahme, daß der Datentyp *char* einen 8-Bit Typen darstellt, erhält man den Indexterm $addr = i^2$. Mit einer Segmentgröße von 32 Byte ergibt sich die folgende Tabelle von Segmentzugriffen:

i	Addresse	Segment
0	0	0
1	1	0
2	4	0
3	9	0
4	16	0
5	25	0
6	36	1
7	49	1
8	64	2
9	81	2
10	100	3
11	121	3
12	144	4
13	169	5
14	196	6
15	225	7

Tabelle 6.1: Segmentierung nicht affiner Arrayzugriffe

Anhand der Tabelle läßt sich leicht erkennen, daß die Zugriffe auf die einzelnen Segmente unregelmäßig erfolgen. Für das Segment 0 würde der Abrollfaktor 6 betragen, wohingegen der Abrollfaktor für das Segment 7 eins betragen würde. Ein Abrollen der Schleife zum zwischenspeichern von Daten ist somit nicht sinnvoll.

Beispiel 6.2.5 *Affine Arrayzugriffe*

```
void LoopFunction()
{
long A[256];
int i;

  for ( i = 0; i < 16; i++ )
  {
    A[i] = 0;
  }
}
```

Das Beispiel 6.2.5 stellt nun einen positiven Fall dar. Der Index des Arrayzugriffs ist mit $addr = i * 4$ (long stellt einen 32-Bit Datentyp dar) ein affiner Ausdruck. Damit egibt sich die folgende Tabelle für die Segmentzugriffe:

i	Addresse	Segment
0	0	0
1	4	0
2	8	0
3	12	0
4	16	0
5	20	0
6	24	0
7	28	0
8	32	1
9	36	1
10	40	1
11	44	1
12	48	1
13	52	1
14	56	1
15	60	1

Tabelle 6.2: Segmentierung affiner Arrayzugriffe

Man erkennt, daß in jedem Segment 8 Speicherzugriffe stattfinden. Durch die Gleichverteilung dieser Speicherzugriffe ist ein abrollen der Schleife mit dem Faktor 8 somit sinnvoll.

Damit ergibt sich der folgende, in Pseudocode dargestellte, Programmcode:

Beispiel 6.2.6 *Arrayzugriffe in abgerollter Schleife*

```
void LoopFunction()
{
long A[256];
long *addr;
int i;

  for ( i = 0; i < 16; i +=8 )
  {
    RegM[i * 4 + 0] = 0;
    RegM[i * 4 + 1] = 0;
    RegM[i * 4 + 2] = 0;
    RegM[i * 4 + 3] = 0;
    RegM[i * 4 + 4] = 0;
    RegM[i * 4 + 5] = 0;
    RegM[i * 4 + 6] = 0;
    RegM[i * 4 + 7] = 0;

    addr = &A + i * 4;
    Store RegM, addr;
  }
}
```

Die Anzahl der Speicherzugriffe wurde in dem Beispiel 6.2.6 um den Faktor acht reduziert. Es ist somit sehr gut das Potential dieser Optimierung zu erkennen, falls die Vektorisierung einer Schleife nicht möglich ist.

Zur Ermittlung der Indexterme können die bereits im Vorfeld verwendeten Verfahren (siehe Kapitel 3.5 und 5.4.3 dieser Diplomarbeit) verwendet werden. Auch das Ermitteln von Schleifengrenzen und die Schrittweite der Schleifenvariablen wurde bereits ausführlich erklärt (Kapitel 5.3.1).

Nach der Ermittlung der Indexterme erfolgt nun die Überprüfung auf ihre Affinität. Dies zu Überprüfen gestaltet sich relativ einfach und kann auf vielfältige Art und Weise geschehen. Eine Möglichkeit besteht darin, den Term in seine Komponenten bestehend aus Variable, Konstanten und Operatoren zu zerlegen. Diese Komponenten können dann nach einer

bestimmten Priorität sortiert werden, wie z.B. $Variable > Konstante > * > +$.

Beispiel 6.2.7 *Pseudoalgorithmus für Affinitätstest*

```
function IsTermAffine(term: String): boolean;
begin
var list: List;

  list = SplitTerm(term);

  if ( list.GetCount() <> 5 )
     then return false;

  SortList(list);

  return GetEntryType(list[0]) == TYPE_VARIABLE and
         GetEntryType(list[1]) == TYPE_CONSTANT and
         GetEntryType(list[2]) == TYPE_CONSTANT and
         GetEntryType(list[3]) == TYPE_OPERATOR_MUL and
         GetEntryType(list[4]) == TYPE_OPERATOR_PLUS;
end
```

Als Beispiel sei der affine Term $2*i+4$ betrachtet. Der vorgestellte Pseudoalgorithmus 6.2.7 würde diesen Term in die Komponenten i, 2, 4, $*$, $+$ zerlegen und in dieser Reihenfolge in der Liste *list* abspeichern. Eine Überprüfung der einzelnen Komponenten nach dem oben angegebenen Schema würde somit diesen Term als affinen Term erkennen.

Nach der Bestimmung der Indexterme und dem Test auf ihre Affinität wird für jeden einzelnen Term die maximale Zahl bestimmt, um welche die Schleife abgerollt werden kann. Die kleinste sich ergebende Zahl bestimmt dann den Faktor, um den die Schleife

abgerollt wird.

Beispiel 6.2.8 *Maximaler Abrollaktor*

```
void LoopFunction()
{
int A[16];
int B[32];
int i;

  for ( i = 0; i < 16; i++ )
  {
    A[i] = 1;
    B[i * 2] = 2;
  }
}
```

Bei einer Größe von 16-Bit für den Datentyp *int* und bei einer Größe von 32-Byte pro Register, könnte die Schleife für den Arrayzugriff $A[i] = 1$; um den Faktor 16 und für den Arrayzugriff $B[i * 2] = 2$; um den Faktor 8 abgerollt werden. Somit ergibt sich für das Beispiel 6.2.8 der maximale Abrollfaktor von 8.

Abschließend kann gesagt werden, daß der Einsatz dieser vorgestellten Methode nur bei kleinen Schleifen zum Einsatz kommen sollte. Große Schleifen mit einem großem Abrollfaktor erzeugen sehr viel Code, was wiederum zu Problemen mit dem Speicherplatz führen kann. Weiterhin ist die Anzahl der Zwischenregister der *M3-DSP* Prozessoren stark beschränkt. Bei Schleifen mit vielen Arrayzugriffen auf unterschiedlichen Speicherbereichen kann das dazu führen, daß der Codegenerator Spillcode für die Register erzeugen muß. Es könnte unter Umständen zu einer Verschlechterung der Laufzeit und einer Erhöhung der Speicherzugriffe führen. Große Schleifen sollten somit vorher, wenn dies möglich ist, mittels der *Loop Fission* Optimierung (Kapitel 6.4 zerteilt werden, um eine effizienteres Ausnutzung der vorhandenen Register zu erhalten. Weitere Techniken zur besseren Ausnutzung der parallelen Datenpfade bei skalaren Variablen können dem Artikel [LKB01] und der Diplomarbeit von David Kottmann [K00] entnommen werden.

6.3 Optimierte Adreßzuweisungen

Eine weitere Verringerung der Speicherzugriffe und eine somit einhergehende Steigerung der Ausführungsgeschwindigkeit kann dadurch erreicht werden, indem die vom *M3-DSP* Prozessor zur Verfügung gestellten Adreß- $(P_0, \ldots P_3)$ und Modify-Register $(M_0, \ldots M_3)$ verwendet werden. Die Adreßregister P_x werden zur Adressierung des zu ladenen Speicherbereichs verwendet. Mittels der Modify-Register M_x können die Adressen

der Zeigerregister erhöht bzw. erniedrigt werden. Bei einem linearen Speicherzugriff ist somit möglich, alle Operationen zur Adressberechnung mittels dieser Register durchzuführen.

Der Aufwand dieser Optimierung ist relativ gering, da auf bereits vorhandene Werkzeuge zurückgegriffen werden kann, welche im folgenden sind:

- Werkzeug zur Erkennung von Schleifen.
- Werkzeug zur Ermittlung der Indexterme von Arrayzugriffen.
- Werkzeug für Affinitätstest der Indexterme.

Die Optimierung kann dann mittels der folgenden vier Schritte durchgeführt werden:

1. **Schleifenerkennung**
   Zuerst werden die Grenzen der Schleife und die Schrittweite der Schleifenindexvariable ermittelt. Der Schleifenindexwert wird für die Initialisierung des Modify-Register M_x verwendet. Der Startwert der Schleife wird dagegen für den Startwert des Adreßregister P_x benötigt.

2. **Ermittlung der Indexterme**
   Im zweiten Schritt werden alle Indexterme für Arrayzugriffe ermittelt. Diese Indexterme müssen affine Ausdrücke darstellen, damit ein linearer Arrayzugriff gewährleistet ist. Ist ein Ausdruck affin, so wird dieser mit dem Startwert der Schleife berechnet, um den Initialisierungswert des *RegPx* zu erhalten.
   Für das Beispiel 6.3.1 ergibt sich folgende Berechnung (unter der Annahme das die Adresse des Arrays A bei 100 liegt):
   $RegP1 = \&A + 4 * StartwertSchleife = 100 + 4 * 0 = 100.$
   Der Wert des Registers M_x ergibt sich aus der Schrittweite der Schleife und der Größe der Arrayelemente. Man erhält damit folgende Berechnung:
   $M_x = sizeof(int) * SchrittWeite = 4 * 1 = 4.$

3. **Einfügen der Befehle für Hardwarezeiger- und Modifikationsregister**
   Die Operationen zur Initialisierung der Register werden direkt an die Initialisierung der Schleifenindexvariable gebunden. Der Speicherzugriffsbefehl wird entfernt und durch den entsprechenden Hardwarebefehl ersetzt. Der Befehl zur Veränderung des P_x Registers wird zum Schluß unterhalb des Befehls zur Veränderung der Schleifenindexvariable eingefügt.

4. **Dead-Code Elimination**
   Nach dem Einfügen der neuen Operationen ist ein *Dead-Code Elimination* unbedingt durchzuführen, daß ein Großteil der vorhandenen Operationen (z.B. für die Adreßberechnung) nicht mehr benötigt werden.

Beispiel 6.3.1 *Schleife nach dem Einfügen der Hardwareregisterunterstützung*

```
Nach Z-Loop Optimierung
und Dead-Code Elimintaion
-------------------------

void LoopFunction()
{
int A[16];
int i, *t1, *t2, *t3;

i = 0;
ZLoopInit 16;

ZLoopLabel:
  ZLoopInit(16, ZLoopLabel);
  t1   = &A;
  t2   = i * 4;
  t3   = t1 + t2;
  *t3  = 0;

  i    = i + 1;


  ZGoto LoopLabel;
}
```

```
Nach Verwendung der
Hardwareregister
-------------------

void LoopFunction()
{
int A[16];
int i, *t1, *t2, *t3;
Register P1, M1;

i = 0;

P1 = &A;
M1 = 4;

ZLoopLabel:
  ZLoopInit(16, ZLoopLabel);
  t1   = &A;
  t2   = i * 4;
  t3   = t1 + t2;
  Store (P1, M1), 0;

  i = i + 1;

  ZGoto LoopLabel;
}
```

Anhand des Beispiels 6.3.1 läßt sich sehr gut die Vorgehensweise bei der Optimierung erkennen. Der *Store (P1, M1), 0* Befehl des Prozessor schreibt den Wert Null in den Speicher und erhöht automatisch den Wert des Adreß-Register mittels des Wertes des

Modify-Register *M1*.

Beispiel 6.3.2 *Schleife nach Dead-Code Elimination*

```
Nach Dead-Code Elimination
--------------------------

void LoopFunction()
{
int A[16];
Register P1, M1;

ZLoopLabel:
  ZLoopInit(16, ZLoopLabel);
  P1 = &A;
  M1 = 4;

  LoopLabel:
    Store (P1, M1), 0;

  ZGoto LoopLabel;
}
```

Im Beispiel 6.3.2 wurde nun die *Dead-Code Elimination* durchgeführt. Dadurch konnten alle skalaren Variablen entfernt werden.

Es ist leicht ersichtlich, daß durch die Verwendung der Adreß- und Modify-Register des *M3-DSP* sehr gute Resultate bzgl. der Erhöhung der Ausführungsgeschwindigkeit von Programmen erzielen lassen.

6.4 Loop Fission

Die *Loop Fission* Optimierung [BGS94] (auch als *Loop Distribution* bezeichnet) ist für sich gesehen keine eigentliche Optimierung eine Transformation, da sie alleine betrachtet keine Verbesserung der Codegüte erreicht. Vielmehr kann sie zur Unterstützung weiterer Optimierungen wie der *Vektorisierung* (Kapitel 5) oder beim modifizierten *Loop Unrolling* Verfahren (Kapitel 6.2) eingesetzt werden.

Der Einsatz der *Loop Fission* Optimierung bietet sich in den Fällen an, wo Programmteile innerhalb einer Schleife eine Optimierung verhindern. Zu diesem Zweck werden diese

Programmteile aus der Schleife entfernt und vor oder nach der Schleife eingebunden.

Beispiel 6.4.1 *Loop Fission*

```
Nicht vektorisierbare      Nach Loop Fission          Nach
Schleife                                              Vektorisierung
---------------------      -----------------          --------------

void LoopFunction()        void LoopFunction()        void LoopFunction()
{                          {                          {
int A[16];                 int A[16];                 int A[16];
int i, x = 0;              int i, x = 0;              int i, x = 0;

  for (i=0; i<16; i++)       for (i=0; i<16; i++)       for (i=0; i<16; i+=4)
  {                          {                          {
    A[i] = 1;                  A[i] = 1;                  SimdStore &A[i], 1
    x = x + i;               }                          }
  }
}                            for (i=0; i<16; i++)       for (i=0;
i<16; i++)
                             {                          {
                               x = x + i;                 x = x + i;
                             }                          }
                           }                          }
```

In dem Beispiel 6.4.1 ist eine Vektorisierung nicht möglich, da die skalare Variable x von der Anzahl der Iterationen der Schleife abhängt. Mittels der *Loop Fission* Optimierung wurde nun der Programmteil, welcher für die Veränderung der Variable x zuständig ist, aus der Schleife entfernt und in einen eigenen Schleifenkörper eingebettet. Durch diesen Schritt existiert keine Abhängigkeit mehr zwischen dem Programm innerhalb des Schleifenkörpers und der Anzahl der Iterationen der Schleife. Ein Vektorisierung der Schleife ist somit möglich.

6.5 Function Inlining

Die Optimierung *Function Inlining* wird in den meisten Fällen dazu verwendet, den Overhead, welcher beim Aufruf von Funktionen entsteht, zu reduzieren. Dazu wird der Programmcode einer Funktion direkt in den Programmcode der aufrufenden Funktion einge-

setzt.

Beispiel 6.5.1 *Function Inlining*

```
Vor Function Inlining
---------------------

int Square(int nValue)
{
  return nValue * nValue;
}

void PrintSquare()
{
int i, nResult;

  for ( i = 0; I < 16; I++ )
  {
    nResult = Square(i);
    cout << nResult << endl;
  }
}
```

```
Nach Function Inlining
----------------------

void PrintSquare()
{
int i, nResult;

  for ( i = 0; I < 16; I++ )
  {
    nResult = i * i;
    cout << nResult << endl;
  }
}
```

In dem Beispiel 6.5.1 wurde der Aufruf der Funktion *Square* entfernt und durch den entsprechenden Programmcode dieser Funktion ersetzt. Der durch den Aufruf von *Square* entstehende Overhead wurde somit entfernt, was zu einer Erhöhung der Ausführungsgeschwindigkeit des Programmcodes führt. Als Nachteil ist die damit verbundene Erhöhung des Programmcodes zu beachten, was im schlechtesten Fall zu einer Erhöhung der Cachemiss-Rate führt.

Doch nicht nur zur Erhöhung der Ausführungsgeschwindigkeit ist die *Function Inlining* Optimierung von Bedeutung. Sie kann ebenfalls zur Unterstützung der Vektorisierung eingesetzt werden.
Eines der Hauptprobleme bei der Vektorisierung von *C/C++* Programmen besteht in der von *C/C++* unterstützten Zeigerarithmetik. Durch die Verwendung von Zeigern kommt es zu einem Alias Effekt bei Variablen. D.h., daß mehrere Variablen ein und dieselbe Speicheradresse bezeichnen, was eine Abhängigkeitsanalyse zwischen den Variablen erschwert. Werden Zeiger von Variablen als Parameter an Funktionen übergeben, so ist eine Abhängigkeitsanalyse nicht mehr möglich.

Dazu sei das folgende Beispiel betrachtet: 6.5.2:

Beispiel 6.5.2 *Funktionsaufruf mit Zeigerübergabe*

```
void SetPointerValue(byte* pbyData, int nPos, byte byValue)
{
  pbyData[nPos] = byValue;
}

void PointerCallFunction()
{
byte byData[16];
byte* pbyData;
int i;

  pbyData = &byData;

  for ( i = 0; I < 16; I++ )
  {
    SetPointerValue(pbyData, i, 1);
  }
}
```

In dem Beispiel 6.5.2 wird die Zeigervariable *pbyData* auf die Adresse der Variable *byData* gesetzt. Somit referenzieren die Variablen *byData* und *pbyData* den gleichen Speicherbereich. Durch die Übergabe des Zeigers an die Funktion *SetPointerValue* und der damit verbundenen Veränderung des Speicherbereiches, werden Abhängigkeiten bzgl. der beiden Variablen über Funktionsgrenzen hinweg getragen. Eine Abhängigkeitsanalyse ist in solchen Fällen nicht mehr möglich, da der Funktion *SetPointerValue* beliebige Adressen übergeben werden können.

Um dieses Problem zu umgehen, bietet es sich an, solche Funktionen vor der Analyse, mittels der *Function Inlining* Optimierung in die zu analysierende Funktion einzusetzen.

Beispiel 6.5.3 *Funktionsaufruf mit Zeigerübergabe nach Inlining*

```
void PointerCallFunction()
{
byte byData[16];
byte* pbyData;
int i;

  pbyData = &byData;

  for ( i = 0; I < 16; I++ )
  {
    pbyData[i] = 1;
  }
}
```

Das Beispiel 6.5.3 zeigt nun die Vorgehensweise auf. Der Körper der Funktion *SetPointerValue* wurde in den Körper der Funktion eingesetzt *PointerCallFunction* eingesetzt. Da nun alle Referenzen auf die Variablen *byData* und *pbyData* innerhalb der Funktion stattfinden, ist eine Datenflußanalyse wiederum möglich.

Folgende Gesichtspunkte kommen bei der Frage, wann eine Funktion mittels *Function Inlining* in eine andere Funktion eingesetzt werden sollte:

- **Funktionen ohne Zeigerübergabe**
  Funktionen, die als Parameter nur skalare Variablen enthalten, können bei der Datenflußanalyse ignoriert werden, da keine Abhängigkeiten über die Funktionsgrenzen hinweg transportiert werden. Ein *Function Inlining* ist somit nicht notwendig.

- **Funktionen mit weiteren Funktionsaufrufen**
  Das Inlining von Funktionen, welche weitere Funktionen aufrufen, sollte vermieden werden, da durch das rekursive Inlining von weiteren Funktionen die Codegröße sehr stark ansteigen kann. Weiterhin ist der durch starkes Inlining entstehende Code häufig nicht mehr vektorisierbar, da eine Reihe neuer Abhängigkeiten bzgl. Speicherzugriffen entstehen können.

- **Funktionen mit Schleifen**
  Funktionen mit internen Schleifen sollten ebenfalls nicht verwendet werden, da durch das Hinzufügen von weitere Schleifen innerhalb der zu analysierenden Schleife, eine Erkennung der Abhängigkeiten von Speicherzugriffen nicht mehr oder nur noch bedingt möglich ist.

Abschließend kann gesagt werden, das dieses Verfahren sich nur für einfache kleine Funktionen eignet. Da eine saubere Programmierung aber auch das Erstellen solcher kleiner Funktionen beinhaltet, ist diese Optimierung für die Vektorisierung als sinnvoll zu erachten.

6.6 Unterstützende Maßnahmen zur Vektorisierung

Die Möglichkeit der Durchführung einer Vektorisierung hängt wie im Vorfeld bereits beschrieben, von der Struktur der zu analysierenden Schleife ab. Um eine möglichst hohe Anzahl von vektorisierbaren Schleifen zu erhalten, sollten zuvor Schleifentransformationen durchgeführt werden, um dieses Ziel zu erreichen. Methoden für Schleifentransformationen wurden bereits ausführlich theoretisch untersucht. Da ein Vorstellen dieser Untersuchungen den Rahmen der Diplomarbeit übersteigen würde, wird der Leser auf die entsprechende Literatur von [B93], [DRV00], [WL90] und [X00] verwiesen.

Anstelle eines theoretischen Hintergrunds werden nun einige Schleifentransformationen vorgestellt ([BGS94], [L01] und [PW01]), welche Schleifen in eine für die Vektorisierung verbesserte Form bringen können. Zum besseren Verständnis der Optimierungen und Transformationen wird jeweils ein praktisches Beispiel zu den Beschreibungen gegeben.

Loop-Invariant Code Motion
Bei dieser Optimierungen werden Berechnungen, welche sich innerhalb der Schleife nicht verändern, vor die Schleife gezogen und durch eine Konstante ersetzt. Dadurch ergibt sich für die Vektorisierung, neben der so zusätzlich gewonnenen Erhöhung der Ausführungsge-

schwindigkeit, die Möglichkeit, dieser Schleife zu vektorisieren.

Beispiel 6.6.1 *Loop-Invariant Code Motion*

```
for ( i = 0; i < 16; i++ )
{
  A[i] = sqrt(x);
}

  a) Originale Schleife


c = sqrt(x)

for ( i = 0; i < 16; i++ )
{
  A[i] = c;
}

  b) Nach Code Motion
```

Loop Unswitching
Loop Unswitching entfernt Bedingte Sprünge innerhalb der Schleife und erzeugt für jede Verzweiung jeweils eine neue Schleife. Diese Transformation kann eine Schleife in eine vektorisierbare Form bringen, da Verzweigungen innerhalb einer Schleife häufig eine Vek-

torisierung verhindern (siehe Kapitel 5.3.4).

Beispiel 6.6.2 *Loop Unswitching*

```
for ( i = 0; i < 16; i++ )
{
  if ( x < 8 )
     A[i] = 8;
  else
     A[i] = 16;
}
```

a) Originale Schleife

```
if ( x < 8 )
{
  for ( i = 0; i < 16; i++ )
    A[i] = 8;
}
else
{
  for ( i = 0; i < 16; i++ )
    A[i] = 16;
}
```

b) Nach Loop Unswitching

Loop Interchange
Bei Zugriffen auf mehrdimensionalen Arrays kommt es häufig vor, das Elemente anstatt zeilenweise spaltenweise gelesen werden. Da dies eine Vektorisierung verhindert, bietet es sich an, solche Zugriffe in verschachelten Schleifen zu entfernen. Hängen diese Zugriffe von den Schleifenvariablen ab, so sollten wenn möglich, die beiden Schleifen zu vertauscht

werden. Dies ist die Vorgehensweise der *Loop Interchange* Optimierung.

Beispiel 6.6.3 *Loop Interchange*

```
for ( i = 0; i < 16; i++ )
  for ( j = 0; j < 16; j++ )
    B[i] += A[i][j]

  a) Originale Schleife


for ( j = 0; j < 16; j++ )
  for ( i = 0; i < 16; i++ )
    B[i] += A[i][j]

  b) Nach Loop Interchange
```

Strip Mining
Eine weitere unterstützende Transformation für die Vektorisierung ist das *Strip Mining*. Beim Strip Mining wird eine Schleife in zwei Teilschleifen zerteilt, so daß die Grenzen der ersten Schleife ein Vielfaches der Datenpfade des Prozessors darstellen. Dadurch wird vermieden, daß Vektoroperationen über die Grenzen von Arrays hinaus auf den Speicher zugreifen.

Beispiel 6.6.4 *Strip Mining*

```
for ( i = 0; i < loopbound; i++ )
    B[i] = A[i];

  a) Originale Schleife


vectorloopbound = (loopbound / 16) * 16;
for ( i = 0; i < vectorloopbound; i++ )
    B[i] = A[i];

scalarloopbound = loopbound - vectorloopbound;
for ( i = 0; i < scalarloopbound; i++ )
    B[i] = A[i];

  b) Nach Strip Mining
```

Loop Coalescing
Loop Coalescing transformiert ineinander verschachtelte Schleifen zu einer einzigen Schlei-

fe. Dadurch können die komplexen Abhängigkeiten, welche durch die Verwendung mehrerer Indexvariablen entstehen, reduziert werden.

Beispiel 6.6.5 *Loop Coalescing*

```
for ( i = 0; i < 16; i++ )
  for ( j = 0; j < 16; j++ )
    B[i] += A[i][j]

  a) Originale Schleife


for ( k = 0; k < 256; k++ )
{
  i = k % 16;
  j = k / 16;
  B[i] += A[i][j]
}

  b) Nach Strip Mining
```

Loop Peeling
Die *Loop Peeling* Transformation entfernt einige Iterationen aus einer Schleife und führt diese vor oder nach der Schleife aus. Diese Optimierung ist vor allem für Vektorprozessoren interessant, welche nicht in der Lage sind alle Speicherzellen direkt auszulesen. Beginnt ein Speicherzugriff innerhalb einer Schleife an einer Adresse, deren Werte nicht an den Anfang der Register geladen werden kann, so müssen die zuviel gelesenen Werte maskiert werden. Dieser durch das Maskieren entstehende Overhead kann durch Loop Peeling vermieden

werden.

Beispiel 6.6.6 *Loop Peeling*

```
for ( i = 14; i < 32; i++ )
  A[i] = 0;

  a) Originale Schleife


A[14] = 0;
A[15] = 0;

for ( i = 16; i < 32; i++ )
  A[i] = 0;

  b) Nach Loop Peeling
```

Reduction Recognition
Reduction Recognition ist im Hinblick auf skalare Operationen interessant, welche einer Vektorisierung im Wege stehen. Die Idee hinter der Transformation besteht darin, zuerst Zwischenergebnisse von arithmetischen Operationen auf skalaren Werten durch die Vektorbefehle des Prozessors berechnen zu lassen. Zum Schluß wird dann anhand der Zwischenergebnisse, mittels der entsprechenden arithmetischen Operationen, das entgültige Ergebnis der skalaren Operation berechnet.

Beispiel 6.6.7 *Reduction Recognition*

```
for ( i = 0; i < 256; i++ )
  x = x + A[i];

  a) Originale Schleife


for ( i = 0; i < 256; i += 16 )
  SimdAdd(R, A[i]);

for ( i = 0; i < 16; i++ )
  x = x + R[i];

  b) Nach Reduction Recognition
```

Scalar Expansion
In vielen Fällen scheitert eine Vektorisierung an der Verwendung von skalaren Variablen

innerhalb von Schleifen. Um eine Vektorisierung solcher Schleifen zu ermöglichen, kann die Transformation *Scalar Expansion* verwendet werden. Dabei werden skalare Variablen durch temporäre Arrays ersetzt, so daß eine Vektorisierung der Schleife möglich wird.

Beispiel 6.6.8 *Scalar Expansion*

```
for ( i = 0; i < 16; i++ )
{
  t = A[i];
  B[i] = t + t % 16;
}

  a) Originale Schleife


for ( i = 0; i < 16; i++ )
{
  T[i] = A[i];
  B[i] = T[i] + T[i] % 16;
}

  b) Nach Scalar Expansion
```

Die hier vorgestellten Optimierungen und Transformationen bilden nur einen kleinen Teil der vorhandenen Möglichkeiten, Schleifen in eine für Vektorisierung verbesserte Form zu bringen. Aber auch allgemeine Optimierungen, wie *Redundant Load/Store Elimination*, *Dead-Code Elimination*, *Constant Folding* und *Copy Propagation* können einen positiven Einfluß auf die Vektorisierbarkeit von Schleifen haben, da sie die Komplexität der Schleife reduzieren können.

Kapitel 7

Untersuchung der empirischen Resultate

In den vorherigen Kapiteln wurden Schleifenoptimierungen für die Prozessoren der *M3-DSP* Plattform vorgestellt, von denen die Optimierungen *Vektorisierung* und *Zero Overhead Hardware Loops* genauer untersucht und implementiert worden sind. Dabei decken die beiden Optimierungen jeweils den Fall einer Optimierung zur Ausnutzung spezieller Hardwarebefehle der *M3-DSP* Prozessoren und einer Optimierung zur Ausnuztung der parallelen Datenpfade der Prozessoren ab.
In diesem Kapitel werden nun Ergebnisse für diese Optimierungen vorgestellt.

7.1 Testverfahren

Zur Durchführung der Untersuchungen wurde der in Kapitel 2.2.4 beschriebene *GeLIR* Simulator verwendet. Dabei wurden die verwendeten Benchmark Programme jeweils mit und ohne Einsatz der implementierten Optimierungen mit Hilfe des am Lehrstuhl 12 entwickelten genetischen Codegenerators (GCG) [LLM01] übersetzt und ausgeführt und bzgl. der folgenden Kriterien beurteilt:

- Codegröße des übersetzten Programmes in Maschineninstruktionen.
- Anzahl der benötigten Speicherzugriffe.
- Anzahl der benötigten Taktzyklen.

Um die Korrektheit der Optimierungen zu überprüfen, wurden jeweils die durch die Programmausführung entstehenden Ergebnisse mit den Ergebnissen des GCC Compilers verglichen.

7.2 Zero Overhead Hardware Loops

In diesem Abschnitt werden experimentelle Ergebnisse unter Verwendung der Zero Overhead Hardware Loops erreicht worden sind. Dazu wurden neun Programme ausgewählt und jeweils mit und ohne Optimierung ausgeführt. Drei dieser Programme (zloop_1.c bis zloop_3.c) wurden selbst entwickelt, um das vorhandene Optimierungspotential dieser Optimierung aufzuzeigen. Die restlichen sechs untersuchten Programme stammen aus der *DSP Stone Benchmark Suite* [DSP], um die Optimierung auch anhand real existierender Programme zu untersuchen. Da nach einer Hardwareschleifen-Optimierung redundante Instruktionen entstehen, wurde jeweils im Anschluß eine Dead Code Elimination durchgeführt.

7.2.1 Beschreibung der Testroutinen

zloop_1.c

```
int ret = 10;

int main(int argc, char** argv)
{
int i;

  for (i = 0; i < 16; i++)
  {
    ret += 2;
  }

return ret;
}
```

Das *zloop_1.c* Programme stellt eine sehr einfache Schleife dar, bei der die Schleifenindexvariable i nicht im Schleifenkörper verwendet wird und somit als reiner Schleifenzähler fungiert. In diesem Fall könnte mittels der Dead-Code Elimination der gesamte Schleifenkontrollcode entfernt werden.

zloop_2.c

```
int ret = -310;

  int A[64] = { 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
                1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
                1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
                1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8 };

  int B[64] = { 8, 7, 6, 5, 4, 3, 2, 1, 8, 7, 6, 5, 4, 3, 2, 1,
                8, 7, 6, 5, 4, 3, 2, 1, 8, 7, 6, 5, 4, 3, 2, 1,
                8, 7, 6, 5, 4, 3, 2, 1, 8, 7, 6, 5, 4, 3, 2, 1,
                8, 7, 6, 5, 4, 3, 2, 1, 8, 7, 6, 5, 4, 3, 2, 1 };

  int main(int argc, char** argv)
  {
  int i;
  int x = 0;

    for (i = 0; i < 16; i++)
    {
      ret += A[x] * B[x];
      x += 4;
    }

  return ret;
  }
```

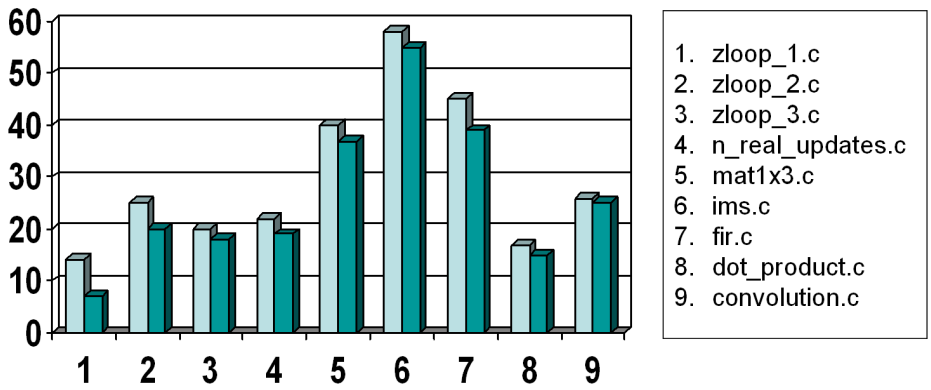
In dem Beispielprogramm *zloop_2.c* ist ebenfalls keine der Instruktionen innerhalb der Schleife von der Schleifenvariablen i abhängig. Allerdings wird hier zur Adressierung des Arrays eine Hilfsvariable x verwendet, deren Wert sich in jeder Iteration um einen konstanten Wert erhöht, jedoch unabhängig von i ist. Auch hier wird nach der Optimierung der gesamte Schleifenkontrollcode redundant.

zloop_3.c

```
int ret = -918;

int A[64] = { 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
              1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
              1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
              1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8 };

int B[64] = { 8, 7, 6, 5, 4, 3, 2, 1, 8, 7, 6, 5, 4, 3, 2, 1,
              8, 7, 6, 5, 4, 3, 2, 1, 8, 7, 6, 5, 4, 3, 2, 1,
              8, 7, 6, 5, 4, 3, 2, 1, 8, 7, 6, 5, 4, 3, 2, 1,
              8, 7, 6, 5, 4, 3, 2, 1, 8, 7, 6, 5, 4, 3, 2, 1 };

int main(int argc, char** argv)
{
int i;

  for (i = 0; i < 64; i++)
  {
    ret += A[i] * B[i];
  }

return ret;
}
```

Das Beispielprogramm *zloop_3.c* stellt einen Fall dar, indem die Schleifenvariable i auch im Schleifenkörper verwendet wird.

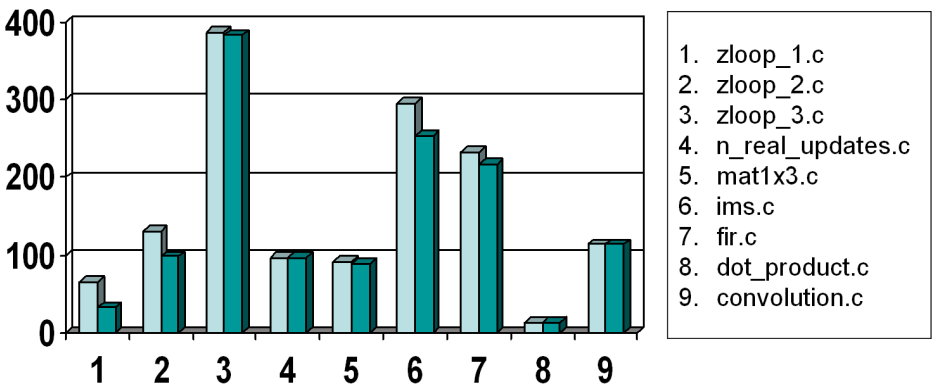
7.2.2 Ergebnisse



Programm	1	2	3	4	5	6	7	8	9
Ohne Optimierung	14	25	20	22	40	58	45	17	26
Mit Optimierung	7	20	18	19	37	55	39	15	25
Verbesserung in %	50%	20%	10%	14%	7.5%	5%	14%	12%	4%

Abbildung 7.1: Ergebnisse Zero Overhead Hardware Loops - Codegröße

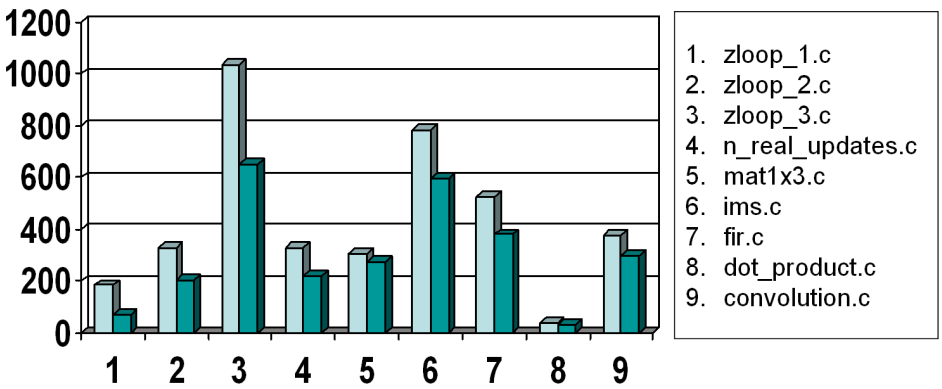
Wie in Abbildung 7.1 zu sehen ist, wurde die Codegröße in den Testroutinen *zloop_1.c* und *zloop_2.c* reduziert, da in diesen beiden Programmen nach der Optimierung der komplette Kontrollcode der Schleife entfernt werden konnte. In Programmen, in denen die Schleifenindexvariable noch im Schleifenkörper verwendet wird, gibt es keine oder kaum Verbesserungen bzgl. der Codegröße. In diesen Fällen besteht die einzige Veränderung gegenüber der unoptimierten Variante darin, daß das Testen der Abbruchbedingung nun durch einen Hardwarebefehl durchgeführt wird.
Im Durchschnitt wurde die Codegröße bei den *zloop*-Programmen um 26% und bei denen der DSP Stone Benchmark Suite um 9.4% reduziert. Der Durchschnitt aller Programme ergab 15.1% Verbesserung, was im Hinblick darauf, daß es sich bei dieser Optimierung um eine Optimierung zur Steigerung der Ausführungsgeschwindigkeit handelt, als gut zu betrachten ist.



Programm	1	2	3	4	5	6	7	8	9
Ohne Optimierung	66	131	386	97	91	296	232	13	114
Mit Optimierung	33	98	386	97	90	254	217	13	114
Verbesserung in %	50%	26%	0%	0%	0%	15%	16%	0%	0%

Abbildung 7.2: Ergebnisse Zero Overhead Hardware Loops - Speicherzugriffe

In Abbildung 7.2 zeigt sich bzgl. der Anzahl von Speicherzugriffen ein ähnliches Bild wie bei den Ergebnissen der Codegröße. In dem Fall, daß der Kontrollcode der optimierten *For-Schleife* redundant und somit entfernt wird, ergibt sich eine Reduzierung der Anzahl von Speicherzugriffen. Das dies nicht nur für entsprechend ausgewählte Testroutinen gilt, sondern auch für allgemeine Programme, erkennt man an den Ergebnissen der Benchmark-programme *ims.c* und *fir.c*. Die durchschnittliche Reduzierung der Speicherzugriffe lag bei den *zloop*-Programmen bei 25.3% und bei den DSP Stone Benchmark-Programmen bei 6.1%. Im gesamten Durchschnitt ergab sich eine Verbesserung der Speicherzugriffe von 11.8%.



Programm	1	2	3	4	5	6	7	8	9
Ohne Optimierung	185	331	1034	328	305	787	527	38	377
Mit Optimierung	73	206	654	220	272	593	381	30	301
Verbesserung in %	61%	38%	37%	33%	11%	25%	28%	22%	20%

Abbildung 7.3: Ergebnisse Zero Overhead Hardware Loops - Ausführungszeit

Anhand der gewonnenen Werte läßt sich in Abbildung 7.3 leicht erkennen, daß die Verwendung der Zero Overhead Hardware Loops entscheidend zur Steigerung der Ausführungsgeschwindigkeit beitragen kann. Auch hier werden die besten Ergebnisse in den Schleifen erzielt, in denen die Schleifenindexvariable nicht im Schleifenkörper verwendet wird. Aber auch in Programmen, in denen der Wert der Schleifenvariable verwendet wird, konnte eine gute Verbesserung der Ausführungsgeschwindigkeit erzielt werden. Beachtenswert sind dabei die sehr guten Ergebnisse der Programme der DSP Stone Benchmark Suite, in denen eine Leistungsverbesserung von bis zu 33% erreicht worden ist. Im Durchschnitt ergab sich eine Steigerung der Ausführungsgeschwindigkeit der *zloop*-Programme von 45% und von den Programmen der DSP Stone Benchmark Suite von 26%. Die durchschnittliche Verbesserung aller Programme von 30% zeigt sehr gut das Potential dieser Optimierung auf.

7.3 Vektorisierung

Im folgenden werden die Ergebnisse der Optimierung Vektorisierung vorgestellt. Bei den verwendeten Programmen *vector_1.c* bis *vector_3.c* handelt es sich wiederum um reine Testroutinen, welche das Optimierungspotential einer Vektorisierung gut verdeutlichen. Da aus den in Kapitel 5 genannten Gründen die Durchführung der Vektorisierung sehr starken Restriktionen unterliegt, konnte nur das Programm *n_real_updates.c* aus der DSP Stone Benchmark Suite vektorisiert werden.

7.3.1 Beschreibung der Testroutinen

vector_1.c

```
int ret = -246;
int A[64];
int B[64] = { 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
              1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
              1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
              1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8 };


int main(int argc, char** argv)
{
  int i, j;

  for (i = 0; i < 64; i++)
    {
      A[i] = B[i];
    }

  return ret;
}
```

Bei dem Programm *vector_1.c* handelt es sich um eine einfache Kopierroutine, wie sie in Programmen häufig vorkommt. Es ist somit gut für eine Vektorisierung geeignet.

vector_2.c

```
int ret = -918;
int A[64] = { 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
              1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
              1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
              1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8 };

int B[64] = { 8, 7, 6, 5, 4, 3, 2, 1, 8, 7, 6, 5, 4, 3, 2, 1,
              8, 7, 6, 5, 4, 3, 2, 1, 8, 7, 6, 5, 4, 3, 2, 1,
              8, 7, 6, 5, 4, 3, 2, 1, 8, 7, 6, 5, 4, 3, 2, 1,
              8, 7, 6, 5, 4, 3, 2, 1, 8, 7, 6, 5, 4, 3, 2, 1 };

int C[64];


int main(int argc, char** argv)
{
  int i, j;

  for (i = 0; i < 64; i++)
    {
      C[i] = A[i] * B[i];
    }

  return ret;
}
```

Das Programm *vector_2.c* stellt einen sehr einfachen Fall von arithmetischen Operationen auf Arrays dar. Damit kann sehr gut die effiziente Ausführung von arithmetischen Operationen mittels SIMD Instruktionen verdeutlicht werden.

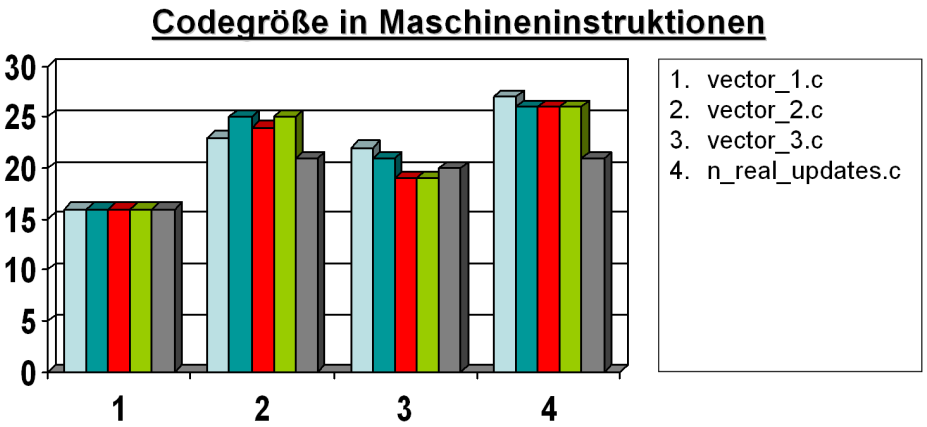
vector_3.c

```
int ret = -374;
int A[64] = { 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
              1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
              1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
              1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8 };

int B[64];


int main(int argc, char** argv)
{
  int i, j, x;

  for (i = 0; i < 64; i++)
    {
      x = A[i];
      x = x + 2;
      B[i] = x;
    }

  return ret;
}
```

Das Programm *vector_3.c* stellt einen Spezialfall von vektorisierbaren skalaren Werten dar. Hierbei wird ausgenutzt, daß Konstanten mit einem "Broadcast"-Befehl in einem Taktzyklus in alle Register eines Gruppenregisters transportiert werden können.

7.3.2 Ergebnisse

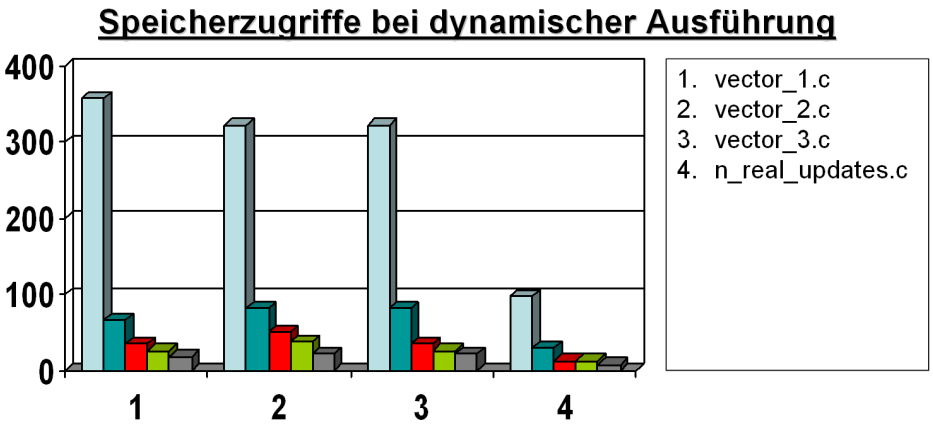
Alle nachfolgenden Ergebnisse werden für Prozessoren der *M3-DSP* Plattform mit jeweils 4, 8, 12 und 16 Datenpfaden (Slices) vorgestellt und mit den Ergebnissen ohne Optimierung verglichen.



Programm	1	2	3	4
Ohne Vektorisierung	16	23	22	27
Vektorisierung mit 4 Slices	16	25	21	26
Vektorisierung mit 8 Slices	16	24	19	26
Vektorisierung mit 12 Slices	16	25	19	26
Vektorisierung mit 16 Slices	16	21	20	21
Verbesserung in % bei 4 Slices	0%	-8%	5%	4%
Verbesserung in % bei 8 Slices	0%	-4%	14%	4%
Verbesserung in % bei 12 Slices	0%	-8%	14%	4%
Verbesserung in % bei 16 Slices	0%	9%	10%	13%

Abbildung 7.4: Ergebnisse Vektorisierung - Codegröße

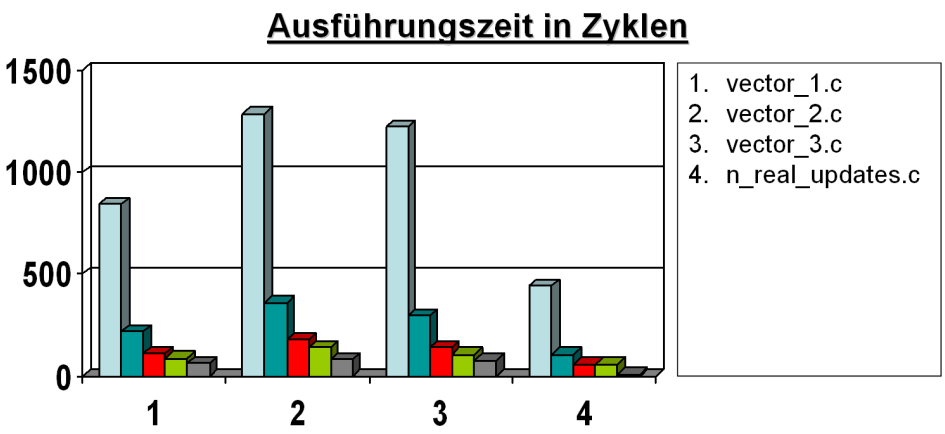
Wie in Abbildung 7.4 zu erkennen ist, konnte die Codegröße durch die Vektorisierung nicht nennenswert verringert werden. In manchen Fällen kam es dabei zu einer relativ geringen Verschlechterung der Codegröße, wobei die Ursache dafür zum einen im genetischen Algorithmus zur Codegenerierung und zum anderen in der veränderten Adreßcodegenerierung [LKB01] zu finden ist.



Programm	1	2	3	4
Ohne Vektorisierung	358	322	322	97
Vektorisierung mit 4 Slices	66	82	82	29
Vektorisierung mit 8 Slices	34	50	34	13
Vektorisierung mit 12 Slices	26	38	26	13
Vektorisierung mit 16 Slices	18	22	22	7
Verbesserung in % bei 4 Slices	72%	75%	75%	70%
Verbesserung in % bei 8 Slices	91%	84%	89%	87%
Verbesserung in % bei 12 Slices	93%	88%	92%	87%
Verbesserung in % bei 16 Slices	95%	93%	93%	93%

Abbildung 7.5: Ergebnisse Vektorisierung - Speicherzugriffe

Wie sich leicht anhand von Abbildung 7.5 erkennen läßt, ergibt sich nach der Vektorisierung eine drastische Reduzierung der Speicherzugriffe. Dabei wurde im besten Fall eine Reduzierung der Speicherzugriffe um bis zu 95% erreicht. Dies resultiert in der Tatsache, daß zum Verarbeiten der Werte innerhalb der Arrays, nicht für jeden einzelnen Wert ein Speicherzugriff erfolgen muß. Es ist somit zu erwarten, daß durch die starke Reduzierung der Speicherzugriffe, auch eine starke Reduzierung des Stromverbrauchs erreicht wird.



Programm	1	2	3	4
Ohne Vektorisierung	841	1289	1225	440
Vektorisierung mit 4 Slices	217	361	297	104
Vektorisierung mit 8 Slices	113	177	137	56
Vektorisierung mit 12 Slices	87	141	105	56
Vektorisierung mit 16 Slices	61	81	77	27
Verbesserung in % bei 4 Slices	74%	72%	76%	76%
Verbesserung in % bei 8 Slices	87%	86%	89%	87%
Verbesserung in % bei 12 Slices	90%	89%	91%	87%
Verbesserung in % bei 16 Slices	93%	94%	93%	94%

Abbildung 7.6: Ergebnisse Vektorisierung - Ausführungszeit

In Abbildung 7.6 sind die Resultate bzgl. der Ausführungszeit dargestellt. Hier ergibt sich das gleiche Bild wie bei der Anzahl der Speicherzugriffe. Durch die Vektorisierung wurde eine Geschwindigkeitssteigerung von bis zu 94% erreicht. Es ist somit leicht ersichtlich, welche Tragweite die Vektorisierung zur Steigerung der Ausführungsgeschwindigkeit bei den *M3-DSP* Prozessoren hat.

7.4 Zusammenfassung

Wie man an den gewonnenen Ergebnissen sehr gut erkennen kann, stellt die *Zero Overhead Hardware Loop* Optimierung aus Kapitel 6.1 eine sehr gute Optimierung für die Prozessoren der *M3-DSP* Plattform dar. Bei allen Benchmark Programmen kam es zu einer teils starken Verbesserung der Ausführungsgeschwindigkeit. Weiterhin konnte bei fast allen der betrachteten Programme die Anzahl der Speicherzugriffe und die Codegröße reduziert werden. Dabei ist es weiterhin nennenswert, daß es in keinem Fall zu einer Verschlechterung der Ergebnisse kam. Wie bereits festgestellt wurde, kommt es in den Fällen zu sehr guten Ergebnissen, in denen der Schleifen-Kontrollcode redundant wird und somit durch eine Dead-Code Elimination entfernt werden kann. Es kann festgestellt werden, daß diese Optimierung in erster Linie zur Erhöhung der Ausführungsgeschwindigkeit dient. Bedingt durch die Art der Optimierung ist eine Verbesserung bzgl. der Codegröße und der Anzahl der Speicherzugriffe nur in geringem Umfang möglich.

Wie zu erwarten wurden die besten Ergebnisse bzgl. der Reduzierung von Speicherzugriffen und der Erhöhung der Ausführungsgeschwindigkeit durch die Vektorisierung erreicht. Dabei konnte ein linearer Anstieg der Verbesserungen bzgl. der Anzahl der verwendeten Datenpfade (Slices) festgestellt werden.
Bei den verwendeten Testroutinen handelt es sich um speziell ausgesuchte Programme, welche zum jetzigen Stand der Optimierungen vektorisiert werden können. Um eine möglichst große Anzahl von Schleifen vektorisieren zu können, bedarf es einer Vielzahl von zusätzlichen Optimierungen, welche eine Schleife in eine für die Vektorisierung geeignete Form bringen. Da aus Zeitgründen diese Optimierungen im Rahmen einer Diplomarbeit nicht implementiert werden können, konnte somit nur ein kleiner Kreis von Programmen vektorisiert werden. Es wäre in diesem Fall wünschenswert, wenn im weiteren Verlauf der Arbeiten am Lehrstuhl 12 auch die in Kapitel 6.6 vorgestellten Optimierungen realisiert werden würden.

Anhand der erzielten Ergebnisse kann klar festgestellt werden, daß die untersuchten und implementierten Optimierungen einen großen Gewinn für die Prozessoren der *M3-DSP* Plattform darstellen. Die Reduzierung der Speicherzugriffe um bis zu 95% und die Erhöhung der Ausführungsgeschwindigkeit um bis zu 94% zeigen das Potential dieser Optimierungen auf. Weiterhin besteht die Möglichkeit die beiden Verfahren zu kombinieren, was zu einer weiteren Verbesserung der Ergbnisse führt.

Kapitel 8

Zusammenfassung und Ausblick

Optimierende Compiler sind bereits lange ein Thema bei der Softwareentwicklung. Große Softwareprojekte sind heutzutage nur mittels Hochsprachen zu realisieren, welche von der eigentlichen Maschinenebene abstrahieren. Aber auch bei Software für kleine portable Rechner ist ein Bedarf an optimierenden Compilern vorhanden, da in Hochsprachen entwickelte Programme eine einfachere Wartung zulassen und leichter auf andere Systeme zu portieren sind.

Die in diesem Rahmen der Diplomarbeit vorgestellten und implementierten Optimierungen wurden zwar speziell für die Verwendung auf Prozessoren der *M3-DSP* Plattform erstellt, können jedoch auch für andere Prozessoren mit entsprechenden Befehlssätzen verwendet werden.
Vektorisierung ist ein schwieriges Gebiet bei der Compilerentwicklung und bedarf einer Vielzahl von Analysen und Sicherheitsabfragen, um stets ein korrektes Ergebnis zu erzeugen. Diese Analysen, welche nur statisch ausgeführt werden können, sind in ihrer Aussagekraft stark eingeschränkt. Dies hat zur Folge, daß Schleifen mit komplexen Array-Indexfunktionen, nicht mehr analysiert werden können. Weiterhin kommt hinzu, daß die C/C++ Hochsprache, bedingt durch ihre Zeigerarithmetik, solche Analysen erschwert oder sogar unmöglich macht. Somit ist eine Vektorisierung von Schleifen nur für die Schleifen möglich, welche in ihrer Komplexität eingeschränkt sind. Stellt man sich nun die Frage, ob ein Vektorisierung von C/C++ Programmen Sinn macht, so kann man diese Frage eindeutig mit Ja beantworten. Durch ihre große Verbreitung und einfache Portierung ist die C/C++ Programmiersprache die am häufigsten verwendete Sprache für die plattformübergreifende Entwicklungen von schnellen und ressourcensparenden Programmen. Das eine Vektorisierung auch in dieser komplexen Sprache möglich ist, zeigen die in Kapitel 7 erhaltenen Ergbenisse auf. Weiterhin können die Ergebnisse durch die Kombination der implementierten Verfahren verbessert werden.

Die implementierte δ-Array Datenflußanalyse wurde in Hinblick auf Geschwindigkeit, Erweiterbarkeit und Präzision entwickelt. Durch ihren modularen und objektorientierten Aufbau ist sie einfach zu verwenden und an die entsprechenden Optimierungen anzupassen. Ferner wurde Wert darauf gelegt, sie nicht nur für die verwendete *GeLIR* IR Darstellung zugänglich zu machen. Durch ihr, mittels *Template* Klassen realisiertes Framework

Konzept, ist sie auch für andere IR Darstellungen einsetzbar.
Sie wurde bis zum jetzigen Zeitpunkt für die Optimierungen *Vektorisierung* und *Redundant Load/Store Elimination* (Lars Hornbach [LH01]) eingesetzt und hat sich in diesen Fällen bewährt. Auch bei den weiteren, für die Optimierungen benötigten, Analysen, wurde Wert auf eine modulare Entwicklung gelegt. Dadurch können diese Analysen ebenfalls leicht in anderen, auf der *GeLIR* arbeitenden Optimierungen, verwendet werden.

Nachdem durch diese Diplomarbeit der Grundstein für die Vektorisierung von C/C++ Programmen gelegt worden ist, bedarf es weiterer Untersuchungen zur Verbesserung des vorhandenen Verfahrens. Dies betrifft vor allem eine Verbesserung und Erweiterung der vorhandenen Abhängigkeitsanalyse auf verschachtelte Schleifen. Weiterhin besteht ein großer Bedarf an unterstützenden Optimierungen, welche die Komplexität von Indextermen in Schleifen reduzieren und Abhängigkeiten zu skalaren Variablen entfernen können.

Anhang A

Implementierung

A.1 Klassenübersicht

Klassenname	Funktion
CEquation<TYPE>	Template Klasse zur Berechnung von Termen, welche als String übergeben werden. Variablen werden dabei mit ihren gesetzten Werten berechnet. Diese Klasse dient als Grundlage der Klasse CDeltaEquation, in der die Speicherabstände zwischen jeweils zwei Array-Zugriffen berechnet werden.
CListFind<TYPE>	Einfache Template Klasse, welche das Suchen in der STL list Klasse ermöglicht.
CMatrix<TYPE>	Diese Template Klasse dient der Darstellung von Matrizen mit variabler Größe. Sie ist Grundlage der CDeltaMatrix Klasse.
CPointerList<TYPE>	Diese Hilfsklasse verwaltet Listen von dynamisch allokierten Speicherbereichen.
CStack<TYPE>	Diese Template Klasse implementiert einen einfachen Stack. Sie wird von CEquation Klasse zur Berechnung der Terme verwendet.
CStringStream	Diese Klasse bildet Stringstreams nach, welche in der STL Implementierung des GNU C Compilers fehlen. Sie wird in einigen Template Klassen dazu verwendet, um Strings in Zahlen oder umgekehrt zu konvertieren.

Tabelle A.1: Implementierung: Allgemeine Klassen

Klassenname	**Funktion**
CDeltaValue<TYPE>	Diese Template Klasse stellt die in der δ-Array Analyse verwendeten Werte von Verbänden dar. Zuzüglich zu den normalen Werten, kann diese Klasse auch die Wert *Top* und *Bottom* annehmen. Dazu sind alle arithmetischen und logischen Operatoren überladen worden.
CDeltaMatrix<TV, TI>	Diese Klasse stellt die δ-Array Matrix dar, welche das Ergebnis nach der Analyse enthält. Sie ist von der CMatrix Klasse abgeleitet, stellt aber zusätzliche Funktionen zur Verfügung, wie das auslesen der Matrix Elemente über gesetzte Indizes.
CDeltaGraph<TI>	Diese Klasse implementiert den Schleifen Kontrollflußgraphen LCFG. Der Graph beinhaltet in jedem Knoten einen Term, welcher die Adreßberechnung eines Array Zugriffs darstellt. Zu den normalen Graphen Funktionen wie das Einfügen und Löschen von Knoten, unterstützt die Klasse spezielle für δ-Array Datenflußanalyse wichtige Funktionen, wie verschiedene Graphendurchläufe für eine optimale Laufzeit der Datenflußanalyse oder das Invertieren des Graphen für Backward-Analysen.
CDeltaEquation<TYPE>	Diese Klasse berechnet die Speicherabstände zwischen jeweils zwei Array-Zugriffen. Dazu wird die CEquation Klasse verwendet.
CDeltaAnalysis<TV, TI>	Diese Klasse bildet den Hauptteil der δ-Array Datenflußanalyse. Zuerst werden die Abstände der Array-Speicherzugriffe mittels der CDeltaEquation bestimmt. Diese Ergebnis werden dann in die CDeltaMatrix geschrieben. Danach wird Datenflußanalyse mittels des Schleifen-Kontrollflußgraphen (CDeltaGraph) und anhand der Werte in der Matrix (CDeltaMatrix) durchgeführt. Diese Klasse unterstützt Must-/May- sowie Forward-/Backward-Analysen.

Tabelle A.2: Implementierung: δ-Array Analyse Klassen

A.2 Datei- und Verzeichnisübersicht der Quellcodes

```
+---UTIL
|   +---SRC
|   |       ml_util.cpp
|   |       stringstream.cpp
|   |
|   +---INCL
|   |       equation.h
|   |       listfind.h
|   |       matrix.h
|   |       ml_util.h
|   |       ptrlist.h
|   |       stack.h
|   |       stringstream.h
|   |
|   \---DELTA_ANALYSIS
|           deltaanalysis.h
|           deltaequation.h
|           deltagraph.h
|           deltamatrix.h
|           deltavalue.h
|
+---LOOP_OPT
|   |   Makefile
|   |
|   +---SRC
|   |       m3dspdeltaanalysis.cpp
|   |       m3dspvectorization.cpp
|   |       m3dsphardwareloops.cpp
|   |
|   \---INCL
|           m3dspdeltaanalysis.h
|           m3dspvectorization.h
|           m3dsphardwareloops.h
|
```

```
\---GELIR_UTIL
    +---SRC
    |       gelircreateindex.cpp
    |       gelirinfo.cpp
    |       gelirmemorylayout.cpp
    |       termlist.cpp
    |       funinfo.cpp
    |       gelirvariableanalysis.cpp
    |       gelirloopanalysis.cpp
    |
    \---INCL
            gelircreateindex.h
            gelirinfo.h
            gelirmemorylayout.h
            termlist.h
            funinfo.h
            gelirvariableanalysis.h
            gelirloopanalysis.h
```

Klassenname	**Funktion**
CFunInfo	Dies ist eine Hilfeklasse für die *GeLIR* . Sie beinhaltet verschiedene Funktionen wie Graphendurchläufe oder das Auffinden von Schleifenkörpern. Sie wird in vielen Klassen zur Gewinnung von Informationen bzgl. des Kontrollflußgraphens einer Funktion verwendet.
CGeLIRInfo	Dies ist eine weitere Hilfeklasse, welche einige häufig verwendete Funktionen zur Ermittlung von *GeLIR* Informationen zur Verfügung stellt. Weiterhin beinhaltet Sie Funktionen für Debug Ausgaben von *GeLIR* Strukturen.
CGeLIRLoopAnalysis	Diese Klasse ermittelt anhand einer *GeLIR* Funktion die Grenzen, Schrittweite und entsprechenden Instruktionen einer Schleife. Sie wird von der CM3DSPHardwareLoops Klasse verwendet, um einen entsprechenden Schleifenkörper zu finden und gegebenenfalls durch Operationen zu ersetzen, welche die *M3-DSP* Hardwareschleifen Befehle ausnutzen.
CGeLIRMemoryLayout	Diese Klasse Teilt jeder Variable eine eigene Speicheradresse zu. Dabei könenn Arrays auf einen beliebigen Wert "alignt" werden. Sie wird von von den δ-Array Datenflußanalyse Klassen verwendet, um die Speicherabstände von Array-Zugriffen bestimmen zu können.
CGeLIRVariableAnalysys	Diese Klasse führt eine Analyse bzgl. aller Variablen in einer Funktion durch. Dabei werden Variablen in Klassen eingeteilt. Diese Klassen unterscheinden sich darin, wie oft eine Variable gelesen bzw. verändert wird und ob eine Variable in einem Basis Block einer Schleife referenziert wird. Sie wird von der CGeLIR-CreateIndex Klasse verwendet um eine genauere Auflösung der Terme zu erhalten.
CGeLIRCreateIndex	Diese Klasse wird ebenfalls von der δ-Array Datenflußanalyse verwendet. Sie berechnet zu jedem Array Zugriff auf *GeLIR* Ebene die entsprechenden Terme. Dabei wird der Kontrollfluß über alle möglichen Kontrollflußwege verfolgt. Sie verwendet die CGeLIR-VariableAnalysys Klasse, um Variablen welchen einen Konstanten Wert innerhalb es bestimmten Bereiches besitzen, gegebenenfalls durch den Konstanten Wert zu ersetzen.
CTermList	CTermList stellt eine Container Klasse dar, welche für die rekursive Auflösung von Indextermen innerhalb der Klasse CGeLIR-VariableAnalysys verwendet wird.

Tabelle A.3: Implementierung: *GeLIR* verwendente Klassen

Klassenname	**Funktion**
CM3DSPDeltaAnalysis	Diese Klasse implementiert die δ-Array Datenflußanalyse für den *M3-DSP* auf der *GeLIR* Datenstruktur. Sie ist von der CDeltaAnalyse Klasse abgeleitet. In der abstrakten Funktion CreateControlFlowGraph() (welche automatisch vom CDeltaAnalysis Framework aufgerufen wird) wird der für die Analyse benötigte Schleifen-Kontrollflußgraph aufgebaut. Das Ergebnis, welches diese Klasse liefert, wird in der Klasse CM3DDSPVectorization verwendet, um zu entscheiden ob die Vektorisierung durchgeführt werden kann oder nicht.
CM3DSPHardwareLoops	Diese Klasse ersetzt normale Schleifen auf der *GeLIR* ebene, durch spezielle Befehle, welche die *M3-DSP* Hardwareschleifen ausnutzen. Dazu wird die CGeLIRLoopAnalysis Klasse verwendet, um die nötigen Informationen bzgl. einer Schleife zu ermitteln.
CM3DDSPVectorization	Diese Klasse implementiert die Vektorisierung für den *M3-DSP* . Im ersten Schritt wird dazu eine Delta-Array Datenflußanalyse durchgeführt, um festzustellen ob eine Vektorisierung möglich ist. In weiteren Schritten werden Abhängigkeiten bzgl. verwendeter Variablen und Befehle überprüft. Näheres dazu kann den Kapiteln 4 und 5 entnommen werden.

Tabelle A.4: Implementierung: *M3-DSP* Optimierungen

Anhang B

Programmierrichtlinien

Bei der Entwicklung der Optimierungen wurde zur Bezeichnung von Variablen, Typen und Klassen die *Ungarische Notation* verwendet.

Bezeichnung von Typen und Klassen

- **Klassen**
  Klassen werden immer mit einem großen "C" eingeleitet, wobei der zweite Buchstabe im Namen der Klasse ebenfalls groß geschrieben wird.
  Beispiel: *class CMyClass*

- **Enumerations**
  Aufzählungen beginnen mit einem großen "E", gefolgt von einem weiteren großen Buchstaben.
  Beispiel: *enum EMyItems* { ... }

- **Typbezeichnungen**
  Alle Typen werden mit einem großen "P" eingeleitet, gefolgt von einem weiteren großen Buchstaben.
  Beispiel: *typedef int TMyProgramID*

- **Pointertypbezeichnungen**
  Pointer von Typbezeichnungen werden mit einem großen "P" eingeleitet und ebenfalls mit einem weiteren großen Buchstaben fortgeführt.
  Beispiel: *typedef TMyProgramID* PMyProgramID*

Bezeichnung von Variablen
Alle Variablen beginnen mit einem Prefix, welcher den Typ der Variable angibt. Dieser Prefix wird immer klein geschrieben. Darauf folgt der Variablenname, welcher mit einem großen Buchstaben beginnt.

Prefix	**Typbezeichnung**
b	bool
a	char
s	short
us	unsigned short
i, n	int
ui	unsigned int
l	long
ul	unsigned long
f	float
d	double
sz	C String zero terminated
str	String Klasse
it	STL Iterator
map	STL map
set	STL set
list	STL list
lir	*GeLIR* Objekte
p	Pointer
r	Reference
c	Konstante Variable

Tabelle B.1: Implementierung: Prefixbeschreibung

Die Prefixe "p", "r" und "c" dürfen nur in Verbindung mit anderen Prefixen verwendet werden.
Beispiel: *const char* pcszText*

Lokalität von Variablen

- **Globale Variablen**
  Globale Variablen beginnen immer mit dem "g_" Perfix.
  Beispiel: *int g_iGlobalVariable*

- **Variablen in Klassen**
  Variablen in Klassen (Member Variablen) werden mit dem Prefix "m_" eingeleitet.
  Beispiel: *int m_iMemberVariable*

- **Lokale Variablen Funktionen**
  Lokale Variablen in Funktionen erhalten kein Prefix.
  Beispiel: *int iLocalVariable*

Literaturverzeichnis

[AJ88] Randy Allen, Steve Johnson:
Compiling C for Vectorization, Parallelization and Inline Expansion.
Proceedings of the SIGPLAN '88, Conference on Programming Language Design and Implementation, Atlanta, Georgia, June 22-24, 1988.

[B93] Utpal Banerjee
Loop Transformations for Restructuring Compilers
ISBN 0-7923-9318-X, Kluwer Academic Publishers, 1993

[BF99] Björn Franke:
Analysen und Methoden optimierender Compiler zur Steigerung der Effizienz von Speicherzugriffen in eingebetteten Systemen.
Universität Dortmund, Fachbereich Informatik, Lehrstuhl 12, Technische Informatik, August 1999.

[BG96] R. Bodik, R. Gupa
Array Data Flow Analysis for Load-Store Optimizations in Fine-Grain Architectures
International Journal of Parallel Programming, 24(6):481-512, December 1996

[BGS94] D. Bacon, L. Graham, O. Sharp:
Compiler Transformations for High-Performance Computing
Computer Science Division, University of California, Berkeley, California 94720. ACM Computing Surveys, Vol. 26, No. 4:345-420, December 1994.

[BIK99] M. Boeckhold, Ireneusz Karkowski, Henk Corporaal:
Transforming and Parallelizing ANSI C Programs Using Pattern Recognition.
Delft University of Technology, Mekelweg 4, P.O. Box 5031, 2600 GA Delft, The Netherlands, 1999.

[DGS93] E. Duesterwald, R. Gupta, M. Soffa:
A Practical Data Flow Framework for Array Reference and its Use in Optimization.
Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation, 28(6):68-77, 1993.

[DRV00] Alain Darte, Yves Robert, Frederic Vivien
Scheduling and Automatic Parallelization
ISBN 0-8176-4149-1, Birkhäuser, 2000

[DSP] DSPstone
A DSP-Oriented Benchmark Methodology
Aachen, University of Technology, Integrated Systems for Signal Processing

[F00] Prof. Dr. G. Fettweiß
M3-DSP *User's Manual*
Technische Universität Dresden, 13 September 2000.

[FWL] Markus Fiesel, Markus Lorenz, Lars Wehmeyer:
An introduction to the Generic Low Level Intermediate Representation GeLIR
University of Dortmund, Germany, June 11, 2001.
http://ls12-www.cs.uni-dortmund.de/research/gelir/

[K00] David Kottmann
Adreßzuweisungen für den M3-DSP
Diplomarbeit am Lehrstuhl 12 des Fachbereichs Informatik der Universität Dortmund, 2000

[L01] E Christopher Lewis
CIS 570: Modern Programming Language Implementation
Online documentation, Spring 2001
http://www.cis.upenn.edu/ eclewis/cis570/

[LA00] Samuel Larsen, Saman Amarasinghe
Exploiting Superword Level Parallelism with Mulditmedia Instructions Sets
MIT Laboratory for Computer Science, Cambridge, MA 02139
Copyright ACM 2000

[LH01] Lars Hornbach
Generische Low-Level Optimierungen für RISC-Architekturen
Diplomarbeit am Lehrstuhl 12 des Fachbereichs Informatik der Universität Dortmund, 2001

[LKB01] Markus Lorenz, David Kottmann, Steven Bashford, Rainer Leupers, Peter Marvedel
Optimized Address Assignment for DSPs with SIMD Memory Access
Dept. of Computer Science 12, University of Dortmund, Germany. ASP-DAC 2001.

[LLM01] Markus Lorenz, Rainer Leupers, Peter Marwedel, Thorsten Dräger, Gerhard P. Fettweis
Low-Energy DSP Code Generation Using a Genetic Algorithm
ICCD, Texas, Austin 2001

[M93] V. Maslov
Lazy Array Data-Flow Dependency Analysis
Technical Report CS-TR-3110.1, University of Maryland, Collage Park CS, July 1993.

[MAL92] D. E. Mayden, S. P. Amarasinghe, M. S. Lam
Data Dependence and Data-Flow Analysis of Arrays
Conference Record of the 5th Workshop on Languages and Compilers for Parallel Computing, 1992.

[PW01] David Padua, Sumedh Wale
CS320 Introductions to Parallel Programming for Scientists and Engineers
Online documentation, Spring 2000
http://polaris.cs.uiuc.edu/ padua/cs320/

[R91] B. R. Rau
Data Flow and Dependency Analysis for Instruction Level Parallelism
Lecture Notes in Computer Science, Vol. 589, p. 236-250, 1991.

[RL01] Rainer Leupers
LANCE: A C Compiler Platform for Embedded Processors Embedded Systems/Embedded Intelligence
Universität Dortmund, Fachbereich Informatik, Lehrstuhl 12, Technische Informatik, February 2001
http://ls12-www.cs.uni-dortmund.de/lance/

[RM97] Robert Morgan
Building an Optimizing Compiler
August 1997

[RS00] Rüdiger Schwarz:
Reduktion des Energiebedarfs von Programmen für den ARM-Prozessor durch Registerpipelining.
Diplomarbeit am Lehrstuhl 12, Fakultät Informatik, Universität Dortmund, August 2000.

[SB00] Steven Bashford
Constraintbasierte Codegenerierung für eingebettete Prozessoren
Dissertation zur Erlangung des Grades eines Doktors der Naturwissenschaften der Universität Dortmund am Fachbereich Informatik, Dortmund 2000
http://eldorado.uni-dortmund.de:8080/FB4/ls12/forschung/2001/Bashford

[SG00] N. Seraman, R. Govindarajan
A Vectorizing Compiler for Multimedia Extensions
International Journal of Parallel Programming, Special Issue on Instruction and Loop-Level Parallelism, 2000

[SM97] Steven Muchnick
Advanced Compiler Design and Implementation
Morgan Kaufmann Publishers, July 1997

[W96] M. Wolf
High Performance Compilers for Parallel Computing
Addision-Wesley Publishing Company, Reading, MA, 1996

[WL90] M. Wolf, M. Lam
A Loop Transformation Theory and an Algorithm to Maximize Parallelism
Computer Systems Laboratory, Stanford University, CA 94305, 1990

[WF99] M. Weiss, P. Fettweis:
A Scalable DSP Architecture for Mobile Communication Applications.
Proceedings of the Fourth Australasian Computer Architecture Conference, Auckland, New Zealand, January 18-21 1999.

[WFL99] M. Weiss, P. Fettweis, M. Lorenz, R. Leupers, R. Marwedel
Toolumgebung für plattformbasierte DSPs der nächsten Generation.
DSP Deutschland, 1999

[X00] Jingling Xue
Loop Tiling For Parallelism
ISBN 0-7923-7933-0, Kluwer Academic Publishers, 2000

[ZC91] H. Zima, B. Chapman
Supercompilers for Parallel and Vector Computers
Addision-Wesley Publishing Company, Reading, MA, 1991