Performance- und energieeffiziente Compilierung für digitale SIMD-Signalprozessoren mittels genetischer Algorithmen

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik

von
Markus Lorenz

Dortmund
2003

Tag der mündlichen Prüfung:
Dekan/Dekanin:
Gutachter:

Vorwort

Diese Arbeit ist während meiner Zeit als wissenschaftlicher Mitarbeiter am Lehrstuhl Informatik XII der Universität Dortmund unter der Betreuung von Prof. Dr. Peter Marwedel entstanden. Ich möchte mich hiermit bei allen Personen bedanken, die direkt oder indirekt[1] zu der Entstehung und Vollendung dieser Arbeit beigetragen haben.

Ich danke Herrn Prof. Dr. Peter Marwedel für die Möglichkeit zur Umsetzung dieser Arbeit und für seine Ratschläge und Unterstützung, die er mir während der Entwicklung der Arbeit gegeben hat. Weiterhin möchte ich Herrn Prof. Dr. Wolfgang Banzhaf für die Bereitschaft zur Erstellung des Zweitgutachtens danken.

Ganz besonderer Dank gilt Steven Bashford, Heiko Falk, Birger Landwehr, Stefan Steinke und Lars Wehmeyer für deren vielfältige und großartige Unterstützung zur Erstellung dieser Arbeit. Neben den Kollegen am Lehrstuhl haben auch die Arbeiten der von mir betreuten Diplomanden David Kottmann, Martin Horst und Markus Fiesel einen großen Anteil an dieser Arbeit. Ebenso möchte ich Thorsten Dräger vom Lehrstuhl Mobile Nachrichtensysteme der TU Dresden für die hervorragende Zusammenarbeit danken.

Vor allem möchte ich meiner Frau dafür danken, dass sie dieses Projekt unterstützt und mitermöglicht hat, obwohl die für die Forschung investierte Zeit häufig über die geregelte Arbeitszeit weit hinausging.

[1] Die Arbeit ist von der Deutschen Forschungsgemeinschaft (DFG) gefördert worden.

Für meine Frau Sabine und für meine Kinder Maike und Robin.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Einführung . . . 3

1.1.1 Phasen des Compilierungsprozesses . . . 3

1.1.2 Digitale Signalverarbeitung . . . 5

1.2 Prozessoren der M3-Plattform . . . 7

1.2.1 M3-DSP . . . 8

1.2.2 Energiekostenmodell . . . 9

1.3 Energieoptimierung durch Compiler . . . 13

1.4 Problemanalyse . . . 16

1.5 Zielsetzungen und Überblick . . . 18

2 Compiler-Zwischendarstellungen **21**

2.1 Grundlegende Begriffe . . . 22

2.2 Zwischendarstellungen existierender Compilersysteme . . . 25

2.3 Low-Level Zwischendarstellung (GeLIR) . . . 29

2.3.1 Programmdarstellung . . . 30

2.3.2 Architekturdarstellung . . . 32

2.3.3 Darstellung alternativer Maschinenprogramme . . . 35

2.3.4 Constraintpropagierung . . . 37

2.3.5 Analysen & Optimierungen . . . 39

2.3.6 Graphische Visualisierung . . . 41

2.3.7 XeLIR . . . 42

2.3.8 Simulationsumgebung . . . 43

3 Codegenerierung für digitale Signalprozessoren 47

3.1 Einführung 48

3.1.1 Baumbasierte vs. graphbasierte Codeselektion 48

3.1.2 Bedeutung phasengekoppelter Optimierungsverfahren 49

3.1.3 Bedeutung von Adressgenerierungseinheiten 51

3.1.4 Kombination von Optimierungszielen 52

3.2 Bestehende Verfahren 53

3.2.1 Codegenerierung 53

3.2.2 Adresscode-Generierung 56

3.2.3 Energieoptimierungen 57

3.3 Übersicht 59

3.4 Preprocessing 60

3.5 Genetischer Codegenerator (GCG) 62

3.5.1 Optimierung auf Basis genetischer Algorithmen 63

3.5.2 Mehrzieloptimierung mit genetischen Algorithmen 65

3.5.3 Chromosomale Darstellung 67

3.5.4 Initialisierung 68

3.5.5 Bewertung der Individuen 74

3.5.6 Selektion 75

3.5.7 Crossover 76

3.5.8 Mutation 78

3.6 Adresscode-Generierung 79

3.6.1 Algorithmus zur Adresscode-Generierung 80

3.6.2 Phasenkopplung mit Codegenerierung 84

3.7 Adresscode-Kompaktierung 85

3.8 Bewertung 86

3.8.1 Einstellung der Parameter des genetischen Algorithmus 86

3.8.2 Genetischer Codegenerator 88

3.8.3 Adresscode-Generierung 92

3.8.4 Retargierbarkeit 94

4 SIMD-Optimierungen **97**
4.1 Einführung 98
4.1.1 Allgemeine Problembereiche 99
4.1.2 M3-spezifische Problembereiche 99
4.1.3 Auswirkungen auf den Codegenerator 101
4.2 Bestehende Verfahren 102
4.3 Übersicht 105
4.4 Architekturdarstellung 107
4.5 Programmdarstellung 109
4.6 Vektorisierung von Schleifen 110
4.6.1 Unterstützende Schleifentransformationen 114
4.6.2 Überprüfung auf Vektorisierbarkeit 117
4.6.3 Ausnutzung spezieller Datentransfers 119
4.6.4 Optimierte Anordnung von Arrays 121
4.7 Optimierte Anordnung skalarer Variablen 123
4.7.1 Problemdefinition 125
4.7.2 Lösungsansatz 128
4.7.3 Integration in den Compilierungsprozess 129
4.8 Bewertung 130
4.8.1 Vektorisierung 130
4.8.2 Anordnung skalarer Variablen 135

5 Experimentelle Ergebnisse **139**
5.1 Betrachtete Benchmarks 139
5.2 Bewertung der Compilertechniken 141
5.3 Vergleich mit handgeneriertem Assemblercode 145
5.4 Systemvergleich 146
5.5 HW/SW-Exploration 147

6 Zusammenfassung **151**
6.1 Compiler-Zwischendarstellung (GeLIR) 152
6.2 Zielarchitektur und Energiekostenmodell 153
6.3 Genetischer Codegenerator (GCG) 154
6.4 SIMD-Optimierungen 155
6.5 Konklusion 157

A Referenzcode **159**

A.1 Testroutine complex_multiply 160

A.1.1 Quellprogramm 160

A.1.2 Handgeschriebener Pseudo-Assemblercode 160

A.2 Testroutine complex_update 161

A.2.1 Quellprogramm 161

A.2.2 Handgeschriebener Pseudo-Assemblercode 161

A.3 Testroutine biquad_one_section 162

A.3.1 Quellprogramm 162

A.3.2 Handgeschriebener Pseudo-Assemblercode 162

A.4 Testroutine lattice2 163

A.4.1 Quellprogramm 163

A.4.2 Handgeschriebener Pseudo-Assemblercode 163

A.5 Testroutine dfg1 164

A.5.1 Quellprogramm 164

A.5.2 Handgeschriebener Pseudo-Assemblercode 164

A.6 Testroutine dfg2 165

A.6.1 Quellprogramm 165

A.6.2 Handgeschriebener Pseudo-Assemblercode 165

Literaturverzeichnis **167**

Indexverzeichnis **179**

Kurz-Zusammenfassung

In den letzten Jahren war ein ständig zunehmender Einsatz von *eingebetteten Systemen* in vielen Produkten unseres täglichen Lebens zu verzeichnen. Häufig sind an diese Systeme spezielle Anforderungen bezüglich einer Realzeitfähigkeit, einer geringen Größe und auch zunehmend eines geringen Energiebedarfs gebunden. Um diesen Anforderungen zu genügen und dennoch ein hohes Maß an Flexibilität beim Systementwurf beizubehalten, werden anstelle von anwendungsspezifischer Hardware häufig *digitale Signalprozessoren* (DSPs) zur Datenverarbeitung eingesetzt. Mit diesen wird auch bei Spezifikationsänderungen in späten Entwicklungsphasen i.d.R. keine kosten- und zeitintensive Neuentwicklung der verwendeten Hardware erforderlich. Leider stellt die manuelle Überführung eines Anwendungsprogramms in Assemblercode des Zielprozessors eine äußerst zeitaufwändige und fehlerträchtige Aufgabe dar. Aus diesem Grund werden Compiler benötigt, die in der Lage sind, eine gegebene Anwendung in effizienten Assemblercode zu überführen. Im Vergleich zu *General-Purpose Prozessoren* (GPPs) weisen DSPs jedoch spezielle Architekturmerkmale auf, die von herkömmlichen Compilertechniken nur unzureichend oder gar nicht ausgenutzt werden.

Das Ziel dieser Arbeit besteht in der Entwicklung neuer Compilertechniken für DSPs, um die durch Compiler generierte Codequalität insbesondere hinsichtlich der Ausführungszeit und des Energiebedarfs zu verbessern. Um eine Wiederverwendung der entwickelten Techniken in anderen Compilern zu ermöglichen, setzen diese auf der ebenfalls in dieser Arbeit beschriebenen neuen Zwischendarstellung GeLIR (*Generic Low-Level Intermediate Representation*) auf.

Als Schwerpunkt dieser Arbeit wird ein Codegenerator vorgestellt, der in der Lage ist, eine *graphbasierte* Codeselektion durchzuführen und zusätzlich die Phasen der Codeselektion, Instruktionsanordnung (einschließlich Kompaktierung) und Registerallokation im Sinne einer *Phasenkopplung* simultan löst. Da dies die Lösung eines NP-harten Optimierungsproblems darstellt, ist dem Codegenerator ein Optimierungsverfahren auf Basis eines genetischen Algorithmus zugrunde gelegt. Zusätzlich werden bei der Durchführung der Teilaufgaben Codeselektion, Instruktionsauswahl und Registerallokation bereits Wechselwirkungen mit der nachfolgend durchgeführten Adresscode-Generierung berücksichtigt. Aufgrund der flexiblen Spezifikationsmöglichkeit von Kostenfunktionen in genetischen Opti-

mierungsverfahren ist der Codegenerator unter Verwendung eines Energiekostenmodells in der Lage, eine energieeffiziente Auswahl und Anordnung von Instruktionen durchzuführen.

Als weiterer Schwerpunkt werden Optimierungsverfahren zur effektiven Ausnutzung der parallelen Datenpfade und von SIMD-Speicherzugriffen vorgestellt. Mit der Integration des Energiekostenmodells in den Codegenerator und den Simulator wird dabei mit dieser Arbeit erstmalig das Potential von SIMD-Operationen hinsichtlich der energieeffizienten Ausführung von DSP-Programmen compilerunterstützt untersucht. Durch die beispielhafte Implementierung der Techniken für eine DSP-Architektur und die Retargierung des genetischen Codegenerators auf einen weiteren DSP wird die Anwendbarkeit für reale Prozessoren gezeigt.

Kapitel 1

Einleitung

Durch die zunehmende Miniaturisierung elektronischer Schaltungen in den letzten Jahrzehnten wurden den Anwendern stetig steigende Rechenleistungen bei immer geringer werdender Chipfläche zur Verfügung gestellt. 1965 sagte Gordon Moore in dem später nach ihm benannten *Moore'schen Gesetz* voraus, dass sich in den darauf folgenden zehn Jahren bis 1975 die Anzahl der auf einem integrierten Schaltkreis vorhandenen Transistoren alle 18 Monate verdoppelt [Moo65]. Wie in Abb. 1.1 zu erkennen ist, hat dieses Gesetz bis heute noch annähernd bestand. Intels Paolo Gorgini ist sogar der Meinung, dass dieses Gesetz für weitere 15 Jahre zutrifft. So soll es im Jahre 2014 Chips mit 64 Milliarden Transistoren und 3,6 GHz Taktfrequenz geben [Sti99].

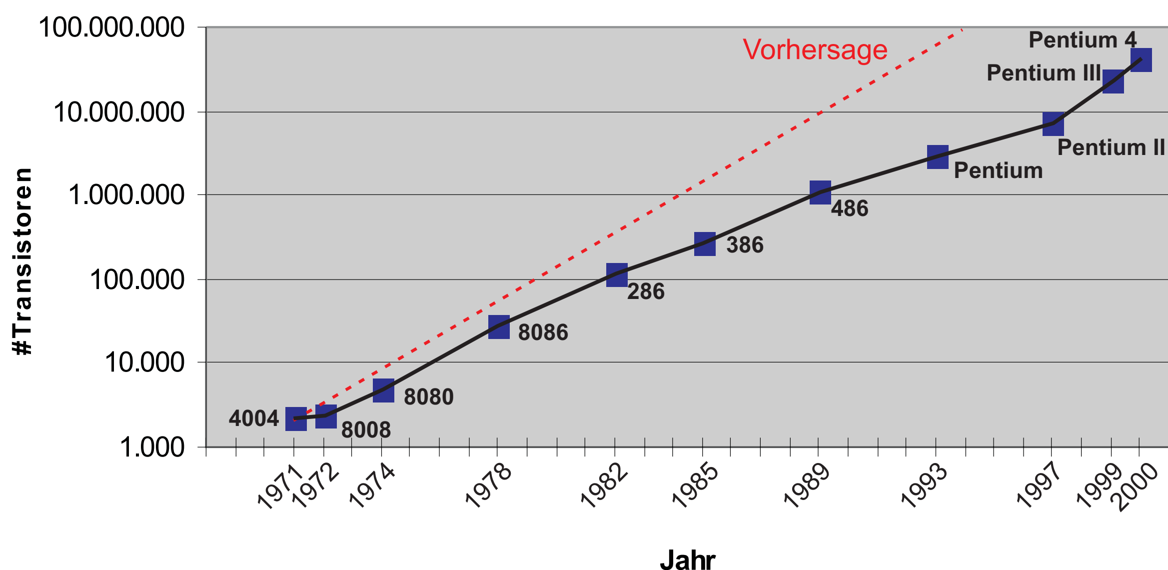


Abb. 1.1: Moore'sches Gesetz am Beispiel der Entwicklung der Intel-Prozessoren (entnommen aus [Int])

Durch die Miniaturisierung besteht die Möglichkeit immer größere, leistungsfähigere Systeme bestehend aus optimierter Hardware und darauf lauffähiger Software auf einem Chip (SoC = *Systems-on-Chip*) zu integrieren. Hierdurch können *eingebettete Systeme* den ständig durch neue Anwendungen gestiegenen Anforderungen bezüglich Ausführungszeit, Chipgröße und in zunehmendem Maße auch geringem Stromverbrauch gerecht werden. Eingebettete Systeme sind in der Regel Bestandteil eines komplexeren Systems und zeichnen sich im Wesentlichen dadurch aus, dass sie physikalische Informationen über Sensoren aufnehmen, verarbeiten und bestimmte Steuerungs- und Regelungsaufgaben durchführen. Typische Einsatzgebiete stellen Anwendungen in Handys (z.B. SMS, WAP), in Automobilen (z.B. Fensterheber, Airbags) und in Flugzeugen (z.B. Kollisionswarngeräte) dar. Insbesondere bei mobilen Geräten, wie Handys, stellt neben einer Echtzeitverarbeitung, einer geringen Größe und einem geringen Gewicht, der Energieverbrauch ein wichtiges Verkaufsargument dar. Während früher diese Aufgaben aufgrund der bestehenden Anforderungen noch von speziell an die Anwendung angepassten ASICs (*Application Specific Integrated Circuits*) durchgeführt werden mussten, wurde in zunehmendem Maße eine Softwarelösung durch den Einsatz von *eingebetteten Prozessoren* möglich. Dies ist äußerst erstrebenswert, da diese programmierbar sind und somit bei Spezifikationsänderungen in späten Designzyklen oder bei Updates/Upgrades, eine kosten- und zeitintensive Neuentwicklung vermeiden (*Time-to-Market*).

Um den genannten Anforderungen gerecht zu werden, werden häufig *digitale Signalprozessoren* (DSPs) eingesetzt, deren Befehlssätze im Vergleich zu *General-Purpose Prozessoren* (GPPs) eine wesentlich effektivere Umsetzung von rechenintensiven Anwendungen erlauben. Leider werden aufgrund der unzureichenden Codequalität der von Compilern generierten Programme, Assemblerprogramme (oder zumindest Teile davon) i.d.R. immer noch per Hand erzeugt. Dies ist jedoch ein sehr zeitaufwändiger Vorgang, der eine hohe Fehleranfälligkeit und eine geringe Portabilität zur Folge hat. Aus diesem Grund besteht ein sehr großer Bedarf an optimierenden Compilern, die an die Architektur angepasst und damit in der Lage sind, die speziellen Architektureigenschaften von DSPs effektiv auszunutzen [MG95].

Nach einer Einführung in die Problematik im nächsten Abschnitt werden kurz die Prozessoren der M3-Plattform, die in dieser Arbeit als Zielarchitektur dienen, vorgestellt. In Abschnitt 1.3 werden dann Möglichkeiten zur Reduzierung des Energieverbrauchs durch Compiler vorgestellt. Nach einer allgemeinen Problemanalyse bestehender DSP-Compiler wird abschließend auf die Zielsetzungen dieser Arbeit eingegangen.

1.1 Einführung

In diesem Abschnitt wird ein Überblick über die zugrunde liegende Problematik der Codegenerierung gegeben, indem zunächst der grundsätzliche Aufbau von Compilern beschrieben wird. Danach werden anhand der Aufgaben der digitalen Signalverarbeitung allgemeine Anforderungen an die Prozessoren, hier im speziellen digitale Signalprozessoren, abgeleitet.

1.1.1 Phasen des Compilierungsprozesses

Die Aufgabe eines Compilers besteht in der Transformation eines gegebenen Quellprogramms in die Sprache des Zielprozessors, unter Wahrung der semantischen Korrektheit und gegebenenfalls Einhaltung weiterer Randbedingungen. Da es sich hierbei um eine sehr komplexe Aufgabe handelt, wird der Compilierungsprozess in mehrere Phasen unterteilt. In Abb. 1.2 sind dies die Teilbereiche *Front-End*, *Middle-End* und *Back-End*, die sich wiederum in mehrere Teilphasen unterteilen.

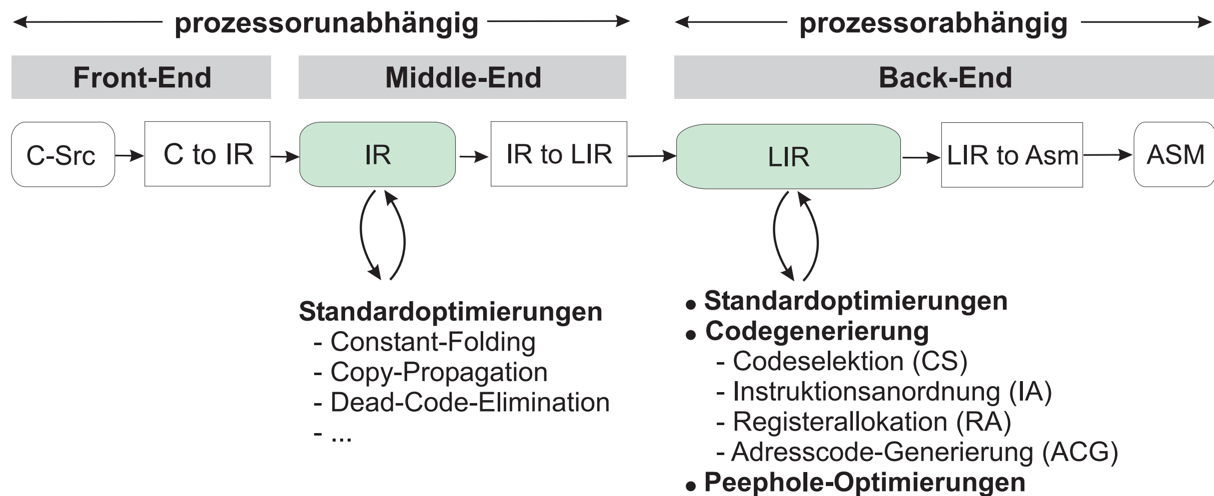


Abb. 1.2: Compilierungsphasen

Das Front-End liest zunächst ein gegebenes Hochsprachenprogramm (hier gegeben in der Programmiersprache C) ein und transformiert dieses nach Durchführung einer lexikalischen, syntaktischen und semantischen Analyse in eine prozessorunabhängige Zwischendarstellung IR (*Intermediate Representation*). Danach können auf dieser Darstellung im Middle-End prozessorunabhängige Standardoptimierungen wie z.B. *Constant-Folding*, *Copy-Propagation* oder *Dead-Code-Elimination* durchgeführt werden, die jeweils eine gegebene IR einlesen und modifiziert zurückschreiben (s. [ASU86, Muc97] für einen Überblick). Diese Optimierungen können für jede betrachtete Zielarchitektur wieder verwendet werden, da bislang keinerlei prozessorspezifische Eigenschaften berücksichtigt worden sind.

Im Back-End gilt es nun, das durch die IR gegebene Programm in ein äquivalentes Programm zu überführen, das auf der zugrunde gelegten Zielarchitektur ausgeführt werden kann. Da dies eine sehr komplexe Aufgabe darstellt, wird dieser Vorgang in eine Reihe von Teilaufgaben unterteilt, die sinnvollerweise wiederum auf einer einheitlichen Zwischendarstellung LIR (*Low-Level* IR) arbeiten sollten. Mit der Durchführung der Codegenerierung werden nun schrittweise architekturspezifische Informationen der LIR hinzugefügt. Dabei werden im Allgemeinen die folgenden Teilaufgaben unterschieden:

- Mit der Durchführung der *Codeselektion* (CS) gilt es, die vorhandenen Operationen der LIR mit elementaren Anweisungen der Zielmaschine (*Maschinenoperationen*) zu überdecken. Beispiele für Maschinenoperationen (MOs) sind Anweisungen zur Durchführung eines Datentransfers oder einer arithmetischen Operation. Wenn eine geringe Ausführungszeit das Optimierungsziel darstellt, kann z.B. eine Minimierung der Anzahl der erforderlichen MOs als Optimierungskriterium dienen. Zur Minimierung des Energieverbrauchs bietet es sich stattdessen an, solche MOs auszuwählen, die den geringsten Energieverbrauch aufweisen. Da in diesem Schritt jedoch keine Entscheidungen hinsichtlich der parallelen Ausführung getroffen werden, kann dies nur einen ungefähren Anhaltspunkt über die letztendlich resultierende Anzahl erforderlicher Prozessorzyklen geben.

- Die Aufgabe der *Instruktionsanordnung* (IA) besteht in der Zuordnung von MOs zu ausführbaren Befehlen der Zielmaschine (*Maschineninstruktionen*) unter Einhaltung der gegebenen Randbedingungen, wie z.B. Datenabhängigkeiten. Im einfachsten Fall enthält dabei jede Maschineninstruktion (MI) genau eine Maschinenoperation. Für Prozessoren mit parallelen Ausführungsmöglichkeiten, wie es bei DSPs üblicherweise gegeben ist, umfasst diese Phase zusätzlich noch die Aufgabe der *Kompaktierung*, bei der die gegebenen MOs zu MIs zusammengefasst werden. Die Ausführungszeit kann reduziert werden, indem die Gesamtzahl der resultierenden MIs minimiert wird. Allerdings ergeben sich auch hier Wechselwirkungen zu den anderen Teilphasen, wie z.B. der Registerallokation.

- Die *Registerallokation* (RA) hat die Aufgabe, alle in einem Programm verwendeten und durch Modifikationen (Transformationen und Optimierungen) hinzugefügte temporäre Variablen (*virtuelle* Register) auf *reale* Register der Zielmaschine abzubilden. Dabei muss entschieden werden, welche Variablen in Registern gehalten werden (*Registervergabe*) und welche Register konkret verwendet werden sollen (*Registerbindung*). Übersteigt die Anzahl der gleichzeitig *lebendigen* Variablen[1] die Anzahl der verwendbaren Register, so müssen Variablen in den Speicher ausgelagert (*gespillt*)

[1] Dies sind Variablen, die zu einem späteren Zeitpunkt der Programmausführung noch verwendet werden, ohne dass sie zuvor mit einem neuen Wert überschrieben werden.

und ein entsprechender *Spillcode* eingefügt werden. Da dies zusätzlicher Taktzyklen und energieintensiver Speicherzugriffe bedarf, besteht das Ziel dieser Phase in der Minimierung des Spillcodes.

- Sind, wie bei DSPs üblich, spezielle *Adressgenerierungseinheiten* (AGUs) vorhanden, führen Codegeneratoren für DSPs zusätzlich noch eine *Adresscode-Generierung* (ACG) durch. Diese hat die Aufgabe, zunächst allen relevanten Variablen konkrete Speicheradressen zuzuweisen und danach alle für die Speicherzugriffe erforderlichen Adressen mit geringst möglichem Overhead zu berechnen.

Da sich die zuvor beschriebenen Teilphasen insbesondere bei Architekturen mit irregulären Befehlssätzen gegenseitig beeinflussen (*Phasenkopplungsproblem*), müssen diese zur Erzielung von optimalen Lösungen simultan gelöst werden. Allerdings stellt bereits die optimale Lösung jeder einzelnen Phase für den allgemeinen Fall die Lösung eines NP-harten Optimierungsproblems dar, so dass aufgrund der erforderlichen Phasenkopplung die Suche nach optimalen oder nahezu optimalen Lösungen erheblich erschwert wird.

Nach der Durchführung von Optimierungen im Back-End können wiederholt Standardoptimierungen, wie z.B. Dead-Code-Elimination ausgeführt werden, diesmal jedoch unter Berücksichtigung architekturspezifischer Merkmale. Abschließend werden i.d.R. auf dem generierten Maschinencode lokale Optimierungen (*Peephole-Optimierungen*) durchgeführt.

1.1.2 Digitale Signalverarbeitung

Aufgrund des zunehmenden Bedarfs an schneller und fehlerfreier Datenverarbeitung werden in immer stärkerem Maße digitale anstelle analoger Signale verarbeitet. Ein wichtiger Grund ist dabei die Möglichkeit der Durchführung von Spezifikationsänderungen in Software. Des Weiteren besteht häufig z.B. auch die Möglichkeit der Rekonstruktion eines durch Rauschen verfälschten Ursprungssignals beim Empfänger und das Anlegen von Kopien ohne Qualitätsverlust.

Der FIR-Filter (FIR = *Finite Impulse Response*) stellt einen häufig umgesetzten Algorithmus in der digitalen Signalverarbeitung dar [EB98]. Das Verhalten eines FIR-Filters N-ter Ordnung, mit dem Eingangswert $x[n-i]$ und dem Filterkoeffizienten $b[i]$ kann durch die Gleichung

$$y[n] = \sum_{i=0}^{N-1} b[i] * x[n-i]$$

beschrieben werden. Der Ausgangswert $y[n]$ zum Zeitpunkt n stellt dann die gewichtete Summe der letzten N Eingangswerte dar. In Abb. 1.3 ist der dazugehörige Signalflussgraph dieses FIR-Filters dargestellt.

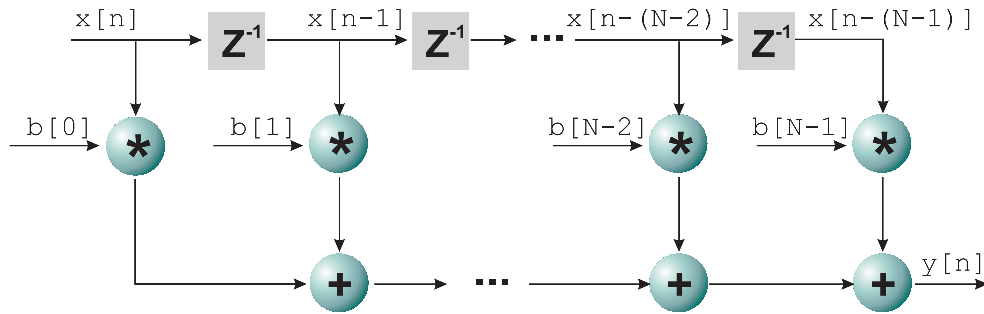


Abb. 1.3: Signalflussgraph eines FIR-Filters N-ter Ordnung

Die mit Z^{-1} benannten Elemente stellen Verzögerungselemente (z.B. Register oder Speicher) dar, die eine Kopie ihres Eingangswertes mit Verzögerung an ihren Ausgang weiterleiten. Im Falle des dargestellten FIR-Filters sind $N-1$ solcher Verzögerungselemente erforderlich, die auch als *Delay-Line* bezeichnet werden. Zu jedem Zeitpunkt enthält die Delay-Line die letzten $N-1$ in das System eingegangenen Signale, die einschließlich des aktuellen Eingabewertes $x[n]$ zur Berechnung des neuen Ausgabesignals $y[n]$ benötigt werden. Für die Berechnung des nachfolgenden Ausgabesignals werden die Werte innerhalb der Delay-Line um jeweils eine Position nach rechts geschoben. Danach kann ein neuer Ausgabewert berechnet werden, indem das aktuelle Eingangssignal und die in der Delay-Line enthaltenen Signale mit den Filter-Koeffizienten $b[0]$ bis $b[N-1]$ multipliziert und aufsummiert werden.

Durch die Wahl der Länge der Delay-Line und der Koeffizienten wird das Verhalten eines solchen FIR-Filters bestimmt. Erfolgt die Umsetzung mittels eines DSPs, können ohne aufwändige Änderungen der Hardware unterschiedliche Filter durch Umprogrammieren des DSPs realisiert werden. Um z.B. die Implementierung des FIR-Filters so effizient wie möglich zu gestalten, stellen DSPs Befehle zur Verfügung, die speziell an derartige Aufgaben angepasst sind [Mar97, EB98]. Dies sind z.B.:

- Bilden der Summe von Multiplikationen mit Hilfe von MAC-Operationen (MAC = *Multiply-Accumulate*).
- Erhöhung der Speicherbandbreite z.B. durch Aufteilung des Speichers in zwei oder mehrere getrennte Datenspeicher, auf die gleichzeitig zugegriffen werden kann. Dies ermöglicht z.B. in der FIR-Routine das parallele Laden des zu verarbeitenden Eingangssignals und eines Filter-Koeffizienten.
- Unterstützung einer (häufig eingeschränkten) parallelen Ausführung von Datenmanipulationen, Datentransfers und/oder Speicherzugriffen. Dazu werden u.a. spezielle Adressgenerierungseinheiten eingesetzt, die aufgrund vorhandener Spezialbefehle eine effektive Adressberechnung parallel zu anderen Funktionseinheiten erlauben.

- Verringerung des Schleifen-Overheads durch die Ausführung einer begrenzten Anzahl von Instruktionen in *Zero-Overhead Hardware-Loops* (ZOL). Diese erlauben nach der Initialisierung eines speziellen Hardwareregisters mit der Anzahl auszuführender Schleifeniterationen die Ausführung der eingebetteten Instruktionen ohne den sonst üblichen Schleifen-Overhead.

1.2 Prozessoren der M3-Plattform

Aufgrund der unterschiedlichen Anwendungsbereiche und den damit verbundenen anwendungsspezifischen Anforderungen, für die DSPs eingesetzt werden sollen, werden von DSP-Herstellern zunehmend Plattformlösungen angestrebt. Dadurch soll die Entwicklung von speziellen ASICs mit den damit verbundenen Nachteilen vermieden werden. So kann mit Hilfe einer DSP-Plattform schnell und kostengünstig ein speziell an die Anwendung angepasster DSP entwickelt werden. Bei steigenden Anforderungen an den DSP kann dadurch eine aufwändige Neuentwicklung oder sogar der Einsatz von ASICs vermieden werden, da die bestehende DSP-Architektur einfach erweiterbar ist [WFL+99]. Beispiele für solche Plattformlösungen stellen der StarCore von Motorola [Sta], der TigerShark von Analog Devices [Tig] und die M3-Plattform von der TU Dresden [FWD+98, WFL+99] dar.

Da der Speicher in den Prozessoren häufig einen Engpass darstellt, wird in vielen Prozessoren der Speicher in mehrere Speicherbänke aufgeteilt. So sind z.B. beim DSP56000 [Mot86], der ADSP-210x-Familie [Dev91] und dem Gepard [GFO97] zwei Speicherbänke vorhanden, auf die gleichzeitig zugegriffen werden kann. Im Unterschied dazu verwendet der Media-Prozessor von MicroUnity einen breiten Standardspeicher, der in Gruppen unterteilt ist [Han96]. Mehrere Daten werden dabei zu einer Gruppe zusammengefasst und über eine gemeinsame Adresse angesprochen. Auf diese Weise geladene Daten können dann nach dem SIMD-Prinzip (SIMD = *Single Instruction Multiple Data*) verarbeitet werden. Allerdings ist bei einer Erweiterung dieser Architektur ein Neuentwurf der Kommunikationseinheit erforderlich.

Die M3-Plattform basiert, wie der Media-Prozessor von MicroUnity, ebenfalls auf dem Gruppenprinzip, ermöglicht jedoch durch die Aufteilung in modulare Einheiten eine einfache Anpassung an applikationsspezifische Spezifikationsänderungen. Eine Anpassung an eine größere Rechenleistung kann z.B. durch eine Erhöhung der Anzahl paralleler Datenpfade erreicht werden, auf denen eine Abarbeitung nach dem SIMD-Prinzip vorgesehen ist. Da i.d.R. nicht zu jedem Zeitpunkt alle Datenpfade benötigt werden, besteht neben der Ausführung von SIMD-Operationen ebenfalls die Möglichkeit der Abarbeitung in einem speziellen *Einstreifen-Modus*. Hierbei erfolgt die Verarbeitung nach dem SISD-Prinzip (SISD = *Single Instruction Single Data*) lediglich auf einem Streifen (*Slice*), indem die

restlichen Datenpfade abgeschaltet werden. Ein Slice besteht dabei aus einem Datenpfad inklusive dazugehöriger Eingangsregister (s. auch Abb. 1.4).

Das Befehlswort ist als VLIW (*Very Long Instruction Word*) organisiert und erlaubt z.B. eine unabhängige Kontrolle zur Datenmanipulation, für Datentransfers, zur Programmsteuerung und Adressgenerierung.

Die Prozessoren dieser skalierbaren Plattform dienen als Zielarchitektur dieser Arbeit, wobei der im folgenden Abschnitt näher beschriebene M3-DSP eine konkrete Instanz darstellt.

1.2.1 M3-DSP

Der M3-DSP stellt eine Instanz mit 16 Slices der skalierbaren M3-Plattform für Anwendungen aus dem Bereich der mobilen Telekommunikation dar (s. Abb. 1.4).

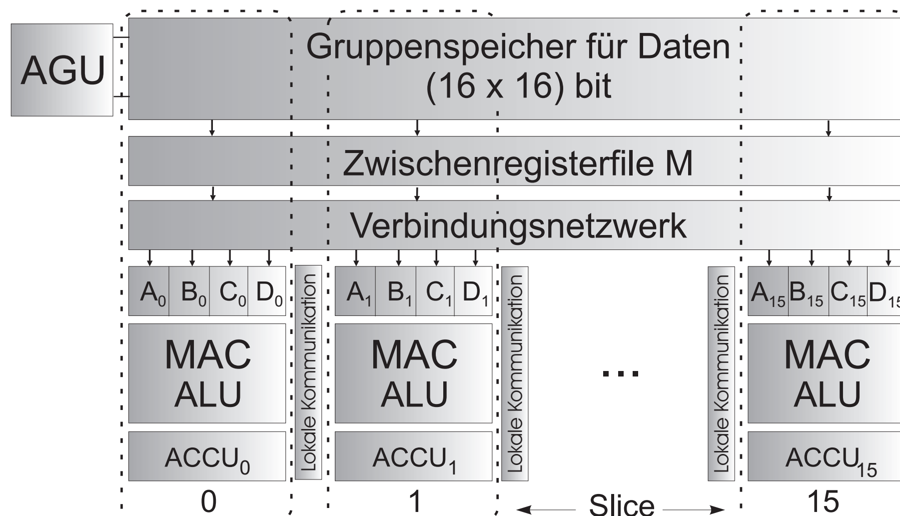


Abb. 1.4: Basisarchitektur des M3-DSPs

Um eine effektive parallele Verwendung aller Datenpfade zu erlauben, ist der auf dem Chip vorhandene Speicher als *Gruppenspeicher* organisiert. Dies bedeutet, dass mit jedem Speicherzugriff auf jeweils eine Gruppe von 16 Datenworten lesend/schreibend zugegriffen wird. Bei einem Ladezugriff wird die adressierte Gruppe in das Zwischenregisterfile `M` geladen, von dem aus die geladenen Daten über ein anwendungsspezifisches Verbindungsnetzwerk in die *Gruppenregisterfiles* der Datenpfade transportiert werden können. Mit „Gruppenregisterfile" wird die Menge der Register aller Slices mit demselben Label (z.B. `A` oder `B` in Abb. 1.4) bezeichnet. Neben dem Transfer einzelner Daten über das Verbindungsnetzwerk sind vor allem auch komplexe SIMD-Datentransfers möglich. So besteht z.B. mit dem *Vektordatentransfer* die Möglichkeit, alle Daten vom Zwischenregisterfile `M` in eines der Gruppenregisterfiles `A`, `B`, `C` oder `D` zu transportieren, wobei jedes Datum

innerhalb desselben Slices verbleibt. Dies bedeutet, dass bei einem Vektordatentransfer vom Zwischenregisterfile `M` in das Gruppenregisterfile `A` die Datentransfers `A[0] = M[0]`, `A[1] = M[1]`, ... , `A[15] = M[15]` innerhalb eines Taktzyklus ausgeführt werden. SIMD-Datentransfers wie *Zurich-Zip* ermöglichen dahingegen einen Transfer über Slice-Grenzen hinaus, indem alle Werte um eine bestimmte Anzahl Slices nach rechts oder links verschoben werden können und unterstützen damit beispielsweise eine effiziente Implementierung des FIR-Filters [LBSL97, DF02]. Um das gesamte Verbindungsnetzwerk klein zu halten, ist keine vollständige Vernetzung umgesetzt, wie dies z.B. mit einem *Kreuzschienenverteiler* (Crossbar-Netz) möglich gewesen wäre. Eine solche Realisierung würde die erforderliche Chipgröße und dadurch auch den Energieverbrauch erheblich erhöhen. Ebenfalls dadurch begründet, ist eine Verwendung der Eingangsregister der Funktionseinheiten in den einzelnen Datenpfaden nur in eingeschränkter Art und Weise möglich. Des Weiteren können zur Verringerung des Schleifen-Overheads bis zu 256 Prozessorinstruktionen in einer Hardwareschleife ausgeführt werden, wobei die Anzahl der Iterationen mit $2^{15} - 1$ Iterationen nach oben begrenzt ist.

Da der M3-DSP alle DSP-typischen Charakteristika aufweist und zusätzlich noch eine SIMD-Ausführung von Operationen unterstützt, stellt dieser Prozessor zur Demonstration der zu entwickelnden Compilertechniken eine geeignete Beispielarchitektur dar.

1.2.2 Energiekostenmodell

Um die Codequalität eines Maschinenprogramms hinsichtlich eines bestimmten Kriteriums beurteilen und optimieren zu können, bedarf es eines geeigneten Kostenmodells, mit dem hinreichend genaue Bewertungen durchgeführt werden können. Zur Bewertung mehrerer unterschiedlicher Codesequenzen während der Compilierung muss das Kostenmodell insbesondere auch eine schnelle Bewertung ermöglichen. Dies ist (in Abwesenheit von hardwaregesteuerten Caches) bezogen auf die Ausführungsgeschwindigkeit und die Codegröße relativ einfach anhand des Instruktionssatzes möglich. So werden in diesen Fällen meist die Anzahl der benötigten Prozessorzyklen oder der erforderliche Programmspeicherplatz aller Instruktionen aufaddiert. Soll jedoch als Optimierungskriterium ein geringer Energieverbrauch dienen, so werden Energieverbrauchswerte einzelner Prozessorinstruktionen oder Teilsequenzen benötigt, die i.d.R. nicht vorhanden bzw. zugänglich sind. Diese Werte müssen dann ermittelt und in einer Form zur Verfügung gestellt werden, die es dem Compiler erlaubt, eine ausreichend genaue Differenzierung von Codesequenzen durchzuführen. Die Ermittlung des Energieverbrauchs einer gegebenen Codesequenz kann grundsätzlich mittels Simulation einer gegebenen Hardwarebeschreibung oder Messung am realen Chip erfolgen, ermöglicht jedoch keine Bewertung von Codesequenzen im Compiler. Aus diesem Grund wird ein geeignetes Kostenmodell benötigt, das einmal aufgestellt, zur Ermittlung des Energiebedarfs beliebiger Instruktionssequenzen im Compiler

dienen kann.

Dazu wurde in [TMW94b, LTMF95] ein Energiekostenmodell auf Instruktionsebene vorgestellt, bei dem der Gesamtenergieverbrauch aus dem Energieverbrauch einer einzelnen Instruktion (*Basis-Energiekosten*) und dem (aufgrund von Zustandsänderungen des Schaltkreises) zusätzlich erforderlichen Energieverbrauch jeweils zweier aufeinander folgender Instruktionen (*Overhead-Energiekosten*) berechnet werden kann. Die relevanten Werte wurden dabei durch Messungen am realen Chip ermittelt. Weitere Energiekostenmodelle auf dieser Basis sind z.B. in [SS99, SBT00, SKWM01] beschrieben.

Mit Hilfe von Strom-Messungen am Siliziumchip des M3-DSPs wurde (in Zusammenarbeit mit der TU Dresden) ebenfalls ein solches Kostenmodell gewonnen [DF02]. Um die Anzahl der durchzuführenden Energiemessungen zu minimieren, wurden in Analogie zu [TMW94b] alle Befehle, die ähnliche Energiekosten verursachen, zu Energiegruppen zusammengefasst und anschließend hinsichtlich ihres Energieverbrauches gleich behandelt. In Tabelle 1.1 sind dies z.B. die Energiegruppen `NOP`, `AGU_1`, `AGU_2`, ... , `LMI_1`. Eine grobe Unterteilung des Befehlssatzes wurde anhand der Funktionseinheiten des VLIW-basierten Instruktionswortes vorgenommen. Dabei handelt es sich bei AGU (*Address Generation Unit*), DMU (*Data Manipulation Unit*), DTU (*Data Transfer Unit*), DAU (*Data Alignment Unit*) und PCU (*Program Control Unit*) um reale Funktionseinheiten des Prozessors, die unabhängig voneinander angesteuert werden können. Die Energiegruppen `LMI_1` (LMI = *Load Move Instruction*) und `NOP` (*No Operation*) stellen Sonderfälle dar.

Bei der Entwicklung des Kostenmodells stellte sich heraus, dass im Falle des M3-DSPs eine Aufteilung der Energiekosten in Basis- und Overheadkosten, wie es in [TMW94b] vorgeschlagen wird, zu großen Abweichungen im Vergleich zu Messungen während der Ausführung auf dem M3-DSP führt. Es hat sich gezeigt, dass eine solche Unterteilung (für den M3-DSP) auch nicht erforderlich ist, weil letztendlich nur der Gesamtenergieverbrauch jeweils zweier aufeinander folgender Maschineninstruktionen benötigt wird. So ergeben sich für den M3-DSP durch eine Zusammenfassung der beiden Einzelkosten zu einem Wert wesentlich genauere Ergebnisse. Des Weiteren sind gegenüber dem in [TMW94b] vorgestellten Modell aufgrund des VLIW-basierten Instruktionswortes Erweiterungen hinsichtlich der Bewertung von parallelen Ausführungsmöglichkeiten erforderlich. Leider gibt es aufgrund der hohen Parallelität trotz der Einteilung in Energiegruppen immer noch eine Vielzahl von Kombinationsmöglichkeiten aufeinander folgender Maschineninstruktionen. Glücklicherweise hat sich herausgestellt, dass der Energieverbrauch komplexerer (paralleler) Befehle mit Hilfe relativ weniger Messungen berechnet werden kann.

Bevor im Folgenden das Energiekostenmodell für den M3-DSP angegeben wird, werden zuvor noch einige zum Verständnis erforderliche Notationen eingeführt:

- Eine Sequenz von Maschineninstruktionen mi_i wird mit *MIs* bezeichnet. Es gilt: $MIs = (mi_1, mi_2, \ldots, mi_i, \ldots, mi_{|MIs|})$, mit $|MIs| \geq 1$

Funktions-einheiten	Energie-gruppen	Maschinenoperationen
NOP	NOP	NOP
AGU	AGU_1	Store
	AGU_2	Load
	AGU_3	Adressregister-Modifikation
DMU	DMU_1	SISD_MAC (Addition), SISD_ADD, SISD_MUL
	DMU_2	SISD_MAC (Subtraktion), SISD_SUB
	DMU_3	SIMD_MAC (Addition), SIMD_ADD, SIMD_MUL
	...	...
DTU	DTU_1	ElementDT, ImmediateDT
	DTU_2	VectorDT
	...	...
DAU	DAU_1	Pack
	DAU_2	ShiftLeft, ShiftRight
PCU	PCU_1	Goto, Push, Pop
LMI	LMI_1	Move, LoadImmediate

Tabelle 1.1: Einteilung des Befehlssatzes in Energiegruppen

- Der Energieverbrauch zweier aufeinander folgend ausgeführter Maschineninstruktionen mi_i und mi_j wird mit E_{mi_i,mi_j} bezeichnet, wobei $i, j \in \{1, \ldots, |MIs|\}$ und $j = i + 1$ gilt.

- Jede Maschineninstruktion mi enthält mindestens eine Maschinenoperation mo. Es gilt:
  $mo_j \in mi$, mit $j \in \{1, \ldots, |mi|\}$

- Die einer Maschinenoperation mo zugeordnete Energiegruppe wird mit EGr_{mo} bezeichnet. Es gilt:
  $EGr_{mo} \in \{NOP, AGU_1, \ldots, LMI_1\}$

- Die Menge aller Energiegruppen der in einer Maschineninstruktion mi enthaltenen Maschinenoperationen wird mit EGr_{mi} bezeichnet. Es gilt:
  $EGr_{mi} = \{EGr_{mo_1}, \ldots, EGr_{mo_{|mi|}}\}$

- Die Menge der im Energiekostenmodell betrachteten Funktionseinheiten wird mit $EnFU$ bezeichnet. Es gilt:
  $EnFU = \{AGU, DMU, DTU, DAU, PCU, LMI\}$.

- Eine Maschinenoperation mo, die auf der Funktionseinheit $fu \in EnFU$ ausgeführt wird, wird mit mo^{fu} bezeichnet.

Der Energieverbrauch E_{MIs} einer Sequenz *MIs* von Maschineninstruktionen kann wie folgt, durch Aufsummieren der Energiewerte zweier aufeinander folgender Maschineninstruktionen ermittelt werden:

$$E_{MIs} = \sum_{i=1}^{|MIs|-1} E_{mi_i, mi_{i+1}}$$

Wenn *EMess* einen gemessenen Wert und *EComp* einen Wert darstellt, der auf der Basis von gemessenen Werten berechnet wird, dann kann der Energieverbrauch von zwei aufeinander folgend ausgeführten Maschineninstruktionen mi_i und mi_j durch folgende Formel bestimmt werden.

$$E_{mi_i, mi_j} = \begin{cases} EMess_{EGr_{mi_i}, EGr_{mi_j}} & \text{, falls Messwert vorhanden} \\ EComp_{mi_i, mi_j} & \text{, sonst} \end{cases}$$

Soll also der Energieverbrauch zweier Maschineninstruktionen mi_1 und mi_2 bestimmt werden, mit $EGr_{mi_1} = \{AGU_1, DAU_1\}$ und $EGr_{mi_2} = \{AGU_2\}$, dann wird zunächst überprüft, ob für diese Kombination von Energiegruppen der Wert $EMess_{\{AGU_1 \| DAU_1\},\{AGU_2\}}$ vorliegt. In diesem Fall würde das dem Eintrag 3,05 in der Zeile `AGU_1` || `DAU_1` und der Spalte `AGU_2` von Tabelle 1.2 entsprechen.

Die Einträge in dieser Tabelle stellen auf die Ausführung eines NOPs genormte Energiewerte für unterschiedliche Kombinationen von Maschineninstruktionen dar. Bei Bedarf besteht jederzeit die Möglichkeit, weitere Messungen durchzuführen und der Energiedatenbasis hinzuzufügen. Ist für eine bestimmte Kombination kein entsprechender Eintrag vorhanden, wird der Energiebedarf zweier aufeinander folgender Maschineninstruktionen mi_i und mi_j, mit $mo_{mi_i} \in mi_i$ und $mo_{mi_j} \in mi_j$, folgendermaßen berechnet:

$$EComp_{mi_i, mi_j} = \sum_{\forall fu \in FU} E_{mo^{fu}_{mi_i}, mo^{fu}_{mi_j}} \quad - \quad (|EnFU| - 1) * EMess_{NOP,NOP}$$

Durch Subtraktion des $(|EnFU| - 1)$-fachen Energieverbrauchs eines NOPs wird berücksichtigt, dass bei allen gemessenen Werten nicht nur der Energieverbrauch einer einzelnen Funktionseinheit gemessen wurde, sondern ebenfalls der Energieverbrauch der restlichen Funktionsheiten, auf denen NOPs ausgeführt werden. Der Energieverbrauch, der auf einer einzelnen Funktionseinheit anfällt, lässt sich mit Hilfe der folgenden Formel berechnen:

$$E_{mo^{fu}_i, mo^{fu}_j} = \begin{cases} EMess_{mo^{fu}_i, mo^{fu}_j} & \text{, falls Messwert vorhanden} \\ EMess_{mo^{fu}_i, NOP} \quad + \quad EMess_{mo^{fu}_j, NOP} & \text{, sonst} \end{cases}$$

Eine Validierung des Energiekostenmodells erfolgte durch einen Vergleich des auf Basis des Energiekostenmodells vorhergesagten Energieverbrauchs, mit dem durch Messung am realen Chip ermittelten. Hierbei ergab sich lediglich eine Abweichung von weniger als 2% im Vergleich zur Ausführung auf dem M3-DSP [DF02].

E-Gruppe	NOP	AGU_1	AGU_2	AGU_3	DMU_1	DMU_2	DMU_3	DTU_1	DTU_2
NOP	1,00	1,57	1,30	1,04	1,31	1,35	5,60	1,22	1,19
AGU_1	1,57	2,14*	1,87*	1,61*	1,88*	1,92*	6,17*	1,79*	1,76*
AGU_2	1,30	1,87*	1,60*	1,29	1,61*	1,65*	5,90*	1,52*	1,49*
AGU_3	1,04	1,61*	1,29	1,07*	1,35*	1,39*	5,63*	1,25*	1,23*
DMU_1	1,31	1,88*	1,61*	1,35*	1,03	5,32	5,34	1,53*	1,50*
DMU_2	1,35	1,92*	1,65*	1,39*	5,32	1,03	6,09	1,57*	1,55*
DMU_3	5,60	6,17*	5,90*	5,63*	5,34	6,09	1,24	5,81*	5,79*
DTU_1	1,22	1,79*	1,52*	1,25*	1,53*	1,57*	5,81*	1,09	1,29
DTU_2	1,19	1,76*	1,49*	1,23*	1,50*	1,55*	5,79*	1,29	1,11
DAU_1	2,40	2,97*	2,70*	2,44*	2,71*	2,76*	7,00*	2,62*	2,59*
DAU_2	2,60	3,17*	2,90*	2,64*	2,91*	2,95*	7,20*	2,82*	2,79*
PCU_1	1,05	1,62*	1,35*	1,08*	1,36*	1,40*	5,65*	1,26*	1,24*
LMI_1	1,06	1,63*	1,36*	1,10*	1,37*	1,42*	5,66*	1,28*	1,26*
AGU_1 ‖ DAU_1	2,86	3,54*	3,05	2,86	3,28*	3,33*	7,57*	3,19*	3,16*
AGU_1 ‖ DAU_2	3,00	3,74*	3,47*	3,21*	3,48*	3,52*	7,77*	3,39*	3,36*
AGU_2 ‖ DTU_1	1,51*	2,09*	1,82*	1,55*	1,83*	1,87*	6,11*	1,39*	1,71*

Tabelle 1.2: Teil der Datenbasis des Energiekostenmodells. Alle mit einem * markierten Werte stellen auf der Basis von real durchgeführten Messungen berechnete Werte dar.

1.3 Energieoptimierung durch Compiler

In diesem Abschnitt wird erläutert, inwiefern ein Energiekostenmodell in einem Compiler zur Energieoptimierung eingesetzt werden kann. Bei genauerer Betrachtung der einzelnen Energiedaten für den M3-DSP wird ersichtlich, dass Load- und Store-Operationen (s. Einträge der Energiegruppen `AGU_1` und `AGU_2` in Tabelle 1.2) im Vergleich zu SISD-Prozessorinstruktionen des Datenpfades (s. z.B. Einträge der Energiegruppen `DTU_1` und `DMU_1`) einen erhöhten Energiebedarf aufweisen. Insbesondere bei Stores ist dies ersichtlich, da diese jeweils in Verbindung mit einer DAU-Operation durchgeführt werden (s. z.B. Eintrag `AGU_1 || DAU_1`). Den höchsten Energiebedarf weisen jedoch die SIMD-Operationen (s. z.B. Energiegruppe `DMU_3`) auf, deren Energieverbrauch um den Faktor vier bis fünf höher ist, als der einer SISD-Operation. Da allerdings im Optimalfall 16 „sinnvolle“ Datenberechnungen parallel durchgeführt werden können, kann damit der Energie-Overhead gegenüber einer SISD-Ausführung wieder mehr als ausgeglichen werden. Des Weiteren ist zu erwarten, dass zusätzlich wesentlich weniger Datentransferbefehle erforderlich sind.

Neben der Entwicklung von Optimierungen zur Verringerung der Ausführungszeit, lohnt sich zwecks Reduzierung des Energieverbrauchs damit auch die Entwicklung von Techniken, die

- zu einer Reduzierung von Speicherzugriffen führen und
- die vorhandenen Datenpfade sinnvoll ausnutzen.

Wie wir anhand von Beispielen im Folgenden sehen werden, kann der Energieverbrauch zusätzlich durch eine geschickte Auswahl und Anordnung von Maschinenoperationen zu Maschineninstruktionen reduziert werden.

In Abb. 1.5 sind zunächst zwei unterschiedliche Schedules a) und b) mit jeweils vier Maschineninstruktionen, repräsentiert durch ihre entsprechende Energiegruppe (z.B. `AGU_2`, `DMU_1`) angegeben.

Schedule a)

Energieverbrauch normiert auf NOPs

```
// MI 1
AGU_2
            } 1,61
// MI 2
DMU_1
            } 1,53
// MI 3
DTU_1
            } 1,53
// MI 4
DMU_1
-----------------
Summe         4,67
              ====
```

Schedule b)

Energieverbrauch normiert auf NOPs

```
// MI 1
AGU_2
            } 1,52
// MI 2
DTU_1
            } 1,53
// MI 3
DMU_1
            } 1,03
// MI 4
DMU_1
-----------------
Summe         4,08
              ====
```

Abb. 1.5: Beispiel zur Energiereduzierung durch Instruktionsanordnung

Als einziger Unterschied zwischen diesen beiden Schedules ist zu erkennen, dass die Maschineninstruktionen MI 2 und MI 3 vertauscht sind. Dadurch begründet ergeben sich für die aufeinander folgenden Befehle unterschiedliche (auf die Ausführung eines NOPs genormte) Energiekosten in Höhe von 4,67 NOPs für Schedule a) und von 4,08 NOPs für Schedule b), was einer Reduzierung von 12,6 % entspricht.

Allerdings besteht nicht nur die Möglichkeit einer Energiereduzierung durch eine Neuanordnung der Instruktionen. Wie in Abb. 1.6 anhand zweier unterschiedlicher Maschinenprogramme für den Ausdruck

$$c = (2 * a) + b + (2 * a);$$

verdeutlicht, kann ebenfalls durch eine geschickte Instruktionsauswahl der Energieverbrauch bei gleicher Ausführungszeit erheblich reduziert werden[2].

[2] In diesem Beispiel wird davon ausgegangen, dass die Variablen a, b und c globale Variablen darstellen und von daher aus dem Speicher geladen und wieder zurückgeschrieben werden müssen.

Maschinenprogramm a)

Energieverbrauch normiert auf NOPs

```
// MI 1: LMI_1
PP = 0;
                                        } 1,58
// MI 2: AGU_2 || DTU_1
M = Mem[PP & 0] ||
C[0] = 2;
                                        } 1,98
// MI 3: AGU_2 || DTU_3
M = Mem[PP & 1] ||
B[0] = M[0];
                                        } 1,77
// MI 4: DMU_1 || DTU_3
Accu[0] = B[0]*C[0] ||
B[0] = M[0];
                                        } 1,41
// MI 5: DTU_1 || DMU_1
A[0] = Accu[0] ||
Accu[0] = B[0]+Accu[0];
                                        } 1,24
// MI 6: DMU_1
Accu[0] = A[0]+Accu[0];
                                        } 3,28
// MI 7: AGU_1 || DAU_1
Mem[PP&2] = Accu;
```

Gesamtenergieverbrauch normiert auf NOPs: 11,26

Maschinenprogramm b)

Energieverbrauch normiert auf NOPs

```
// MI 1: LMI_1
PP = 0;
                                        } 1,58
// MI 2: AGU_2 || DTU_1
M = MEM[PP&0] ||
A[0] = 2;
                                        } 1,69
// MI 3: AGU_2 || DTU_1
M = MEM[PP&1] ||
C[0] = M[0];
                                        } 1,39
// MI 4: DTU_1
B[0] = M[0];
                                        } 1,53
// MI 5: DMU_1
Accu[0] = B[0]+A[0]*C[0];
                                        } 1,03
// MI 6: DMU_1
Accu[0] = Accu[0]+A[0]*C[0];
                                        } 3,28
// MI 7: AGU_1 || DAU_1
MEM[PP&2] = Accu;
```

Gesamtenergieverbrauch normiert auf NOPs: 10,50

Abb. 1.6: Vergleich des Energieverbrauchs unterschiedlicher Maschinenprogramme

Für jede Maschineninstruktion sind jeweils die Energiegruppen der verwendeten Maschinenoperationen mit angegeben. Z.B. wird in Maschinenprogramm a) in MI 2 eine Maschinenoperation aus Energiegruppe `AGU_2` parallel zu einer aus `DTU_1` ausgeführt. Es ist zu erkennen, dass beide Maschinenprogramme mit sieben Maschineninstruktionen diesen Ausdruck umsetzen, wobei jedoch der Energieverbrauch von Maschinenprogramm b) um 6,7 % geringer ist als der von Maschinenprogramm a). Dies lässt sich in diesem Beispiel anhand der Tatsache begründen, dass in Maschinenprogramm b) die Umsetzung mit Hilfe von zwei MAC-Operationen erfolgt (MI 5 und MI 6), während bei Maschinenprogramm a) stattdessen eine Multiplikation (MI 4) und zwei Additionen (MI 5 und MI 6) Verwendung finden. Dadurch ist auch ein zusätzlicher Datentransfer in MI 5 erforderlich. Des Weiteren wird in MI 3 von Maschinenprogramm b) ein weniger energieintensiver Datentransferbefehl aus der Energiegruppe `DTU_1` statt aus der Energiegruppe `DTU_3` verwendet.

1.4 Problemanalyse

Aufgrund mangelnder Techniken zur Handhabung irregulärer Prozessorarchitekturen weisen DSP-Compiler oft eine unzureichende Codequalität in Hinblick auf Realzeitfähigkeit, Codegröße und damit auch Energieverbrauch auf [ZVSM94]. Die verfügbaren Compiler führen i.d.R. eine baumbasierte Codeselektion durch, bei der ein gegebener Datenflussgraph (DFG) an gemeinsamen Teilausdrücken (CSEs = *Common Subexpressions*) in Bäume zerlegt und für jeden der resultierenden Bäume eine separate Codeselektion durchgeführt wird [ASU86, WM95]. Die Durchführung einer baumbasierten Codeselektion für einen Baum mit n Knoten ist zwar sehr laufzeiteffizient in $O(n)$ möglich, weist allerdings insbesondere für irreguläre Architekturen einige Nachteile auf [Bas95]:

- Während in GPPs mit großen Registerfiles CSEs normalerweise in Registern gehalten werden, ist dies bei irregulären Architekturen mit Spezialregistern i.d.R. nicht der Fall. Stattdessen legen herkömmliche Compiler CSEs im Speicher ab und laden diese bei jeder Verwendung neu [AML96, LDKT95], was zu potentiell vermeidbaren Speicherzugriffen und Instruktionen führt.

- Die Phase der Codeselektion wird nur lokal für Bäume durchgeführt. Dadurch können potentielle Überdeckungsmöglichkeiten von Knoten unterschiedlicher Bäume (mit Maschinenoperationen) nicht berücksichtigt werden. Ein solches Verfahren wäre also nicht in der Lage, das energieeffizientere Maschinenprogramm b) in Abb. 1.6 zu generieren. Stattdessen müssten die in MI 5 und MI 6 vorhandenen MAC-Operationen in Einzeloperationen aufgeteilt werden, weil die Multiplikation und die beiden Additionen jeweils unterschiedlichen Bäumen anghören würden.

- Die bei irregulären Architekturen so wichtige Phasenkopplung wird nur eingeschränkt durchgeführt. Mit diesen Verfahren kann die Phase der Codeselektion für Bäume zwar optimal durchgeführt werden, allerdings bezieht sich der Begriff der Optimalität lediglich auf sequentiellen Code. Da im resultierenden Code noch kein (durch Registerallokation erforderlicher) Spillcode enthalten ist, müssen die Phasen der Registerallokation und Codekompaktierung zusätzlich in nachgeschalteten Phasen durchgeführt werden.

- Energiekostenmodelle lassen sich zwar in die einzelnen Codegenerierungs-Phasen integrieren, können allerdings nur sehr ungenaue Abschätzungen über den letztendlichen Energieverbrauch liefern. So können sich energiebewusste Entscheidungen in frühen Phasen später durchaus als ungünstig erweisen, da z.B. das nachträgliche Einfügen von Spillcode die vorhandenen Codesequenzen (und damit auch die Switchingaktivitäten aufeinander folgender Instruktionen) stark beeinflusst.

Um den bei DSPs häufig gegebenen Echtzeitanforderungen gerecht werden zu können, müssen Compiler in der Lage sein, die speziellen Architekturmerkmale zu berücksichtigen. Im einzelnen betrifft dies eine effektive Ausnutzung ...

... von komplexen Operationen wie der MAC-Instruktion, mit der eine Multiplikation gefolgt von einer Addition in einem Taktzyklus ausgeführt werden kann.

... der parallelen Ausführungsmöglichkeiten auf Instruktionsebene (ILP = *Instruction Level Parallelism*). Im Falle der Prozessoren der M3-Plattform umfasst dies zusätzlich die Ausnutzung von SIMD-Operationen in Verbindung mit einer effektiven Nutzung der komplexen Datentransfer-Modi des Verbindungsnetzwerkes.

... gegebener Speicherressourcen, wie z.B. von Speicherbänken oder des On-Chip-Gruppenspeichers der M3-Prozessoren.

... von speziellen Adressgenerierungsbefehlen zur effektiven Ausnutzung der AGU, um den vorhandenen Adressierungs-Overhead so gering wie möglich zu halten.

... vorhandener Spezialbefehle zur Reduzierung des Schleifen-Overheads.

Um die Korrektheit des generierten Codes zu gewährleisten, müssen bei der Compilierung eine Reihe von Randbedingungen in der Verwendung von Registern, Datentransfers und der Parallelisierung berücksichtigt werden. Im Falle der Compilierung für Prozessoren der M3-Plattform umfasst dies jedoch eine Reihe weiterer Randbedingungen, die vor allem durch den Gruppenspeicher und die Sonderfunktionalität des Datenpfades 0 (Einstreifen-Modus) verursacht werden:

- Handhabung irregulärer Datentransfer-Modi.
  Um z.B. auch bei einer Abarbeitung im SISD-Modus eine effektive Bereitstellung von Daten zu gewährleisten, sind speziell abgestimmte Datentransferbefehle zu und von Registern des Datenpfades 0 vorhanden.

- Beibehaltung der Datenkonsistenz des Gruppenspeichers.
  Im Gegensatz zu üblicherweise verwendeten Techniken muss ein Codegenerator für die M3-Prozessoren in der Lage sein, anstelle *einzelner* Daten, *Gruppen* von Daten zu laden, zu speichern und zu spillen.

- Einhaltung weiterer Randbedingungen,
  die sich aufgrund von verwendeten Datentransferbefehlen für die Adresscode-Generierung ergeben.

Da architekturspezifische Optimierungen häufig fest in den entsprechenden Compiler integriert sind, können diese Optimierungen nicht in anderen Compilern genutzt werden.

Dies führt dazu, dass bestimmte Optimierungen für andere Architekturen immer wieder neu implementiert werden. Aus diesem Grund ist ein allgemeines Austauschformat (in Form einer LIR) zwischen den einzelnen Optimierungsphasen äußerst wünschenswert, bei dem prozessorspezifische Architekturmerkmale abgelegt und von den zu entwickelnden Optimierungen abgefragt werden können. Neben der Wiederverwendbarkeit einzelner Optimierungen oder bestimmter Teile, wird durch ein solches allgemeines Austauschformat eine modulare und generische Entwicklung von performance- und energieeffizienten Optimierungstechniken möglich.

1.5 Zielsetzungen und Überblick

Der Schwerpunkt dieser Arbeit besteht in der Entwicklung von neuen Compilertechniken für DSPs, mit dem Ziel, den vorhandenen Overhead herkömmlicher Compilertechniken vor allem in Bezug auf die Ausführungszeit und den Energieverbrauch zu reduzieren. Dabei gilt es alle entwickelten Techniken in einem einheitlichen Back-End (Codegenerator) zu integrieren. Als Zielarchitekturen dienen die parallelen Prozessoren der M3-Plattform, von denen im speziellen der M3-DSP betrachtet wird.

Um eine modulare und generische Implementierung der entwickelten Compilertechniken zu ermöglichen und des Weiteren die Erweiterbarkeit um weitere Optimierungen zu gewährleisten, setzen alle in dieser Arbeit beschriebenen Techniken auf der in Kapitel 2 eingeführten Zwischendarstellung GeLIR (*Generic Low-Level Intermediate Representation*) auf. Durch die Verwendung von GeLIR als einheitlichem Austauschformat können alle implementierten Techniken auf einfache Art und Weise auch für andere Zielarchitekturen adaptiert werden.

Als Schwerpunkt dieser Arbeit wird in Kapitel 3 ein Codegenerator vorgestellt, der in der Lage ist, eine *graphbasierte* Codeselektion durchzuführen und zusätzlich die Phasen der Codeselektion, Instruktionsanordnung (einschließlich Kompaktierung) und Registerallokation im Sinne einer *Phasenkopplung* simultan löst. Da dies die Lösung eines NP-harten Optimierungsproblems darstellt, ist dem Codegenerator ein Optimierungsverfahren auf Basis eines genetischen Algorithmus zugrunde gelegt. Zusätzlich werden bei der Durchführung der Teilaufgaben Codeselektion, Instruktionsauswahl und Registerallokation bereits Wechselwirkungen mit der nachfolgend durchgeführten Adresscode-Generierung berücksichtigt. Als weitere wichtige Eigenschaft des genetischen Optimierungsverfahrens wird eine einfache Berücksichtigung unterschiedlicher Kostenfunktionen besprochen, die u.a. eine Optimierung hinsichtlich der Ausführungszeit, des Energieverbrauchs und deren Kombination ermöglicht.

Als weiterer Schwerpunkt dieser Arbeit werden in Kapitel 4 Verfahren vorgestellt, die eine Ausnutzung von SIMD-Befehlen für die Prozessoren der M3-Plattform ermöglichen, bzw.

zu einer effektiveren Ausnutzung führen. Dies betrifft insbesondere die Vektorisierung von Schleifen. Es wird sich zeigen, dass häufig erst durch eine optimierte Anordnung der Arrays im On-Chip-Gruppenspeicher und durch eine Anwendung von Schleifentransformationen eine Schleife vektorisiert werden kann. Zur Reduzierung des vorhandenen Overheads bei der Handhabung des Gruppenspeichers im SISD-Modus wird in diesem Kapitel des Weiteren ein Verfahren zur Anordnung von skalaren Variablen zu Gruppen vorgestellt, mit dem vor allem die Anzahl der mittels SIMD-Anweisungen durchzuführenden Speicherzugriffe gegenüber einer einfachen Anordnung drastisch verringert werden kann. Als Kern dieses Verfahrens wird ein genetisches Partitionierungsverfahren verwendet, das auch allgemein zur Lösung von Partitionierungsproblemen verwendet werden kann.

Bevor in Kapitel 6 eine Zusammenfassung dieser Arbeit gegeben wird, werden für die vorgestellten Techniken in Kapitel 5 experimentelle Ergebnisse für eine Reihe von realen DSP-Routinen und einer MP3-Anwendung präsentiert. Dies schließt ebenfalls eine am Beispiel des M3-DSPs durchgeführte HW/SW-Exploration ein, bei der die Auswirkungen von Hardware-Änderungen auf die resultierende Codequalität untersucht werden.

Kapitel 2

Compiler-Zwischendarstellungen

Bei der Entwicklung von Compilern spielen Zwischendarstellungen eine zentrale Rolle, da diese allgemein als Austauschformat für die Compilertechniken dienen. So erlauben Zwischendarstellungen aufgrund genormter Schnittstellen u.a. die Wiederverwendung bereits implementierter Techniken in unterschiedlichen Compilern und die simultane Entwicklung von Compilertechniken, wodurch in erheblichem Maße Kosten und Zeit eingespart werden können. Um die Entwicklung neuer Compilertechniken zu vereinfachen, werden im Allgemeinen Informationen zur Verfügung gestellt, die Aussagen über die Anwendbarkeit bzw. semantische Korrektheit bestimmter Modifikationen (Transformationen und Optimierungen) erlauben.

Insbesondere wenn sehr umfangreiche und komplexe Modifikationen vorgenommen werden, sollte eine Validierung so einfach wie möglich umsetzbar sein. Dazu ist eine Simulation und graphische Visualisierung beliebiger Zwischenresultate sehr wünschenswert. Mit Hilfe einer Simulation wird nicht nur die Möglichkeit zu einer automatisierten Validierung geschaffen, sondern auch ermöglicht, durchgeführte Optimierungen auf ihre Effektivität hin zu überprüfen. Dabei sollten zur Bewertung zumindest Daten über die Anzahl der ausgeführten Zyklen bereitgestellt werden. Da in dieser Arbeit ein Schwerpunkt auf der Entwicklung von energieeffizienten Optimierungen liegt, sind zusätzlich Angaben über den Energieverbrauch erforderlich, die allerdings von herkömmlichen Simulatoren nicht zur Verfügung gestellt werden.

Im Vergleich zu einer rein maschinenunabhängigen (*High-Level*) Darstellung stellt die Umsetzung dieser Anforderungen auf der maschinenabhängigen (*Low-Level*) Ebene ein wesentlich größeres Problem dar. So müssen zunächst die erforderlichen architekturspezifischen Merkmale für eine möglichst breite Klasse von Architekturen geeignet zur Verfügung gestellt werden.

Im nachfolgenden Abschnitt werden zunächst einige zum Verständnis des Aufbaus von Compiler-Zwischendarstellungen erforderliche Begriffe und Grundlagen dargelegt. Aufgrund der engen Verwandtschaft von der in dieser Arbeit verwendeten und weiterent-

wickelten Zwischendarstellung GeLIR (*Generic Low-Level IR*) und der von Bashford entwickelten CoLIR (*Constraint based Low-Level IR*), wurden einige der nachfolgend eingeführten Begriffe und Definitionen aus [Bas01] übernommen.

2.1 Grundlegende Begriffe

Die Programmdarstellung in Compiler-Zwischendarstellungen orientiert sich häufig an der allgemeinen Struktur von Programmen imperativer Programmiersprachen, bei der ein Programm aus einer Menge von Funktionen besteht. Diese ergeben sich direkt aus den im Quellprogramm verwendeten Funktionen. Eine Darstellung der dort enthaltenen Kontrollstrukturen (wie z.B. Verzweigungen), erfolgt häufig mittels Kontrollflussgraphen.

Definition 2.1 (Kontrollflussgraph) *Ein Kontrollflussgraph (CFG) ist ein gerichteter Graph* $G = (V, E)$*, dessen Knoten* $v \in V$ *entsprechend des potentiell möglichen Kontrollflusses über Kanten* $e_{i,j} = (v_i, v_j) \in E \subseteq V \times V$ *miteinander verbunden werden. Die Knoten selbst enthalten sequentiell auszuführende Anweisungen und können aufgrund von Verzweigungen des Kontrollflusses mehrere Nachfolger haben.*

Die Knoten eines Kontrollflussgraphen können z.B. Basisblöcke (s. Definition weiter unten) sein, deren enthaltene Anweisungen auf unterschiedliche Arten dargestellt werden können. Dies kann wiederum mittels eines Kontrollflussgraphen geschehen, bei dem entsprechend der Ausführungsreihenfolge zweier Anweisungen Kanten eingefügt werden.

Definition 2.2 (Basisblock) *Ein Basisblock (BB) stellt eine maximale Sequenz von Anweisungen dar, bei der sich der Kontrollfluss nur nach der letzten Anweisung der Sequenz aufteilen und nur bei der ersten Anweisung der Sequenz wieder zusammenfließen kann.*

In Abb. 2.1 wird dies anhand der Aufteilung einer Funktion entsprechend ihres Kontrollflusses in Basisblöcke verdeutlicht. Wie zu erkennen ist, erfolgt eine Unterteilung der Funktion `main` in vier über Kontrollflusskanten verbundene Basisblöcke BB 1 bis BB 4. Während aufgrund der Kontrollflussverzweigung nach der Beendigung von BB 1 keine allgemeine Aussage darüber gemacht werden kann, ob die in BB 2 oder BB 3 enthaltenen Anweisungen ausgeführt werden, steht die Ausführungsreihenfolge der Anweisungen (AMOs = abstrakte Maschinenoperationen) innerhalb der einzelnen Basisblöcke fest.

Definition 2.3 (Abstrakte Maschinenoperation) *Eine abstrakte Maschinenoperation (AMO) stellt eine maschinenunabhängige, elementare Anweisung der Zwischendarstellung dar.*

```
int x, y;

int main()
{
 int ret;

 if(x<y)
  {
   ret = 2 * x + 2;
  }
 else
  {
   ret = 2;
  }

 return ret;
}
```

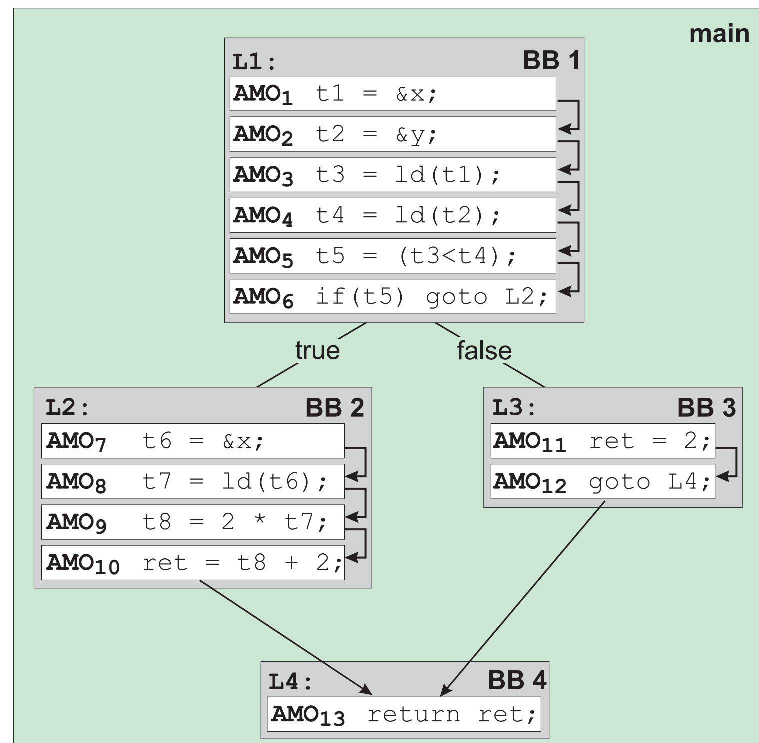


Abb. 2.1: Quellprogramm und dazugehöriger Kontrollflussgraph mit Drei-Adressbefehlen

Die Darstellung des Zwischenprogramms in Compilern erfolgt, wie auch in diesem Beispiel, häufig als Folge von *Drei-Adressbefehlen*.

Definition 2.4 (Drei-Adressbefehl) *Ein Drei-Adressbefehl hat die Form* `x = y op z`*, wobei* `x`*,* `y` *und* `z` *Namen, Konstanten oder vom Compiler generierte temporäre Werte und* `op` *ein binärer arithmetischer oder logischer Operator ist. Die Anzahl und Semantik der Operatoren ist unabhängig vom zugrunde gelegten Quellprogramm fest vorgegeben. In diesem Befehl wird* `x` *definiert und* `y` *und* `z` *werden verwendet.*

Mit Hilfe von Sonderfällen, wie `x = op y`, bei der `op` einen unären Operator darstellt oder `x = y`, bei der eine Kopie des Wertes von `y` erzeugt wird, erfolgt die Darstellung des gesamten Quellprogramms. In dieser Darstellung sind jedoch noch keinerlei architekturspezifischen Merkmale berücksichtigt. Mit der Durchführung der Codegenerierung gilt es daher, eine semantisch äquivalente Darstellung des durch AMOs repräsentierten Programms mit Maschinenbefehlen zu erzeugen. Dazu müssen den Optimierungen u.a. alle vorhandenen Ressourcen des Prozessors bekannt gemacht werden:

Definition 2.5 (Sequentielle Ressourcen) *Sequentielle Ressourcen eines Prozessors sind lesbare bzw. schreibbare Ressourcen wie Registerbänke, Speicherbänke oder Ein- und Ausgabeports, deren Inhalte mehr als einen Instruktionszyklus erhalten bleiben.*

Definition 2.6 (Flüchtige Ressourcen) *Flüchtige Ressourcen sind lesbare bzw. schreibbare Ressourcen, deren Inhalt nur innerhalb eines Instruktionszyklus gültig ist. Dies sind z.B. Signalleitungen zwischen Funktionseinheiten oder Registern zum Zwischenspeichern eines Resultats einer Funktionseinheit, das noch im gleichen Instruktionszyklus von einer anderen Funktionseinheit gelesen wird.*

Grundsätzlich können also sequentielle Ressourcen im Gegensatz zu flüchtigen Ressourcen zum Speichern von Zwischenergebnissen verwendet werden. In Abhängigkeit davon, ob sequentielle Ressourcen oder flüchtige Ressourcen verwendet werden, ergeben sich die folgenden Überdeckungsmöglichkeiten von AMOs mit unterschiedlichen Maschinenoperationen:

Definition 2.7 (Maschinenoperation) *Eine Maschinenoperation (MO) ist eine elementare Operation auf einem Prozessor, wobei die Operanden aus sequentiellen Ressourcen gelesen werden und das Resultat in eine sequentielle Ressource geschrieben wird. Eine Maschinenoperation ist an weitere Ressourcen gebunden, wie z.B. an Funktionseinheiten, auf denen die Operation ausgeführt wird.*

Definition 2.8 (Partielle Maschinenoperation) *Eine partielle Maschinenoperation benutzt (lesend oder schreibend) mindestens eine flüchtige Ressource.*

Definition 2.9 (Komplexe Maschinenoperation) *Eine komplexe Maschinenoperation setzt sich aus mindestens zwei partiellen Maschinenoperationen zusammen und ermöglicht somit die Ausführung komplexer Ausdrücke, wie z.B. die MAC-Operation.*

Definition 2.10 (Faktorisierte Maschinenoperation) *Eine faktorisierte Maschinenoperation (FMO) ist eine Repräsentation alternativer Maschinenoperationen zu einem gegebenen Operator. Sie umfasst die Repräsentation alternativer Mengen von Ressourcen, die einer elementaren Operation auf einem Prozessor zur Verfügung stehen. Dies können alternative Register-Ressourcen für Resultate und Operanden sein sowie alternative Funktionseinheiten, auf denen die Operation ausgeführt werden kann.*

Für den Fall, dass eine Unterscheidung von AMOs und den unterschiedlichen Arten von MOs im Kontext nicht von Bedeutung ist, wird im Folgenden der Begriff MO verwendet.

Im Gegensatz zu einer architekturunabhängigen Programmdarstellung muss bei der Berücksichtigung architekturspezifischer Merkmale insbesondere auch die Möglichkeit zur Darstellung paralleler Ausführungsmöglichkeiten gegeben sein. Aus diesem Grund werden ein oder mehrere MOs (oder auf einer etwas abstrakteren Ebene auch AMOs) zu einer Maschineninstruktion zusammengefasst:

Definition 2.11 (Maschineninstruktion) *Eine Maschineninstruktion (MI) repräsentiert eine Menge von parallel auszuführenden Maschinenoperationen auf einem Prozessor.*

Definition 2.12 (Maschineninstruktionstyp) *Ein Maschineninstruktionstyp (oder auch einfach nur Instruktionstyp) gibt eine maximale Menge von parallel ausführbaren Maschinenoperationen an, so dass keine Maschinenoperation mehr zusätzlich parallel ausgeführt werden kann.*

In einer MI können also je nach Instruktionstyp mehrere (beliebig komplexe) MOs zu einem Maschinenbefehl zusammengefasst werden.

Zur Erleichterung von semantisch korrekten Modifikationen werden neben einer Darstellung des Quellprogramms als Drei-Adresscode auch graphbasierte Darstellungen verwendet, die im Gegensatz zu einer rein flussorientierten Darstellung Auskunft über vorhandene Datenabhängigkeiten zwischen AMOs geben. Es werden die folgenden Arten von Datenabhängigkeiten unterschieden:

Definition 2.13 (Datenflussabhängigkeit) *Wenn AMO_i vor AMO_j ausgeführt wird und AMO_i eine Variable definiert, die AMO_j verwendet, dann liegt eine Datenflussabhängigkeit zwischen diesen beiden AMOs vor.*

Definition 2.14 (Ausgabeabhängigkeit) *Wenn AMO_i vor AMO_j ausgeführt wird und beide AMOs dieselbe Variable definieren, dann liegt eine Ausgabeabhängigkeit zwischen diesen beiden AMOs vor.*

Definition 2.15 (Antiabhängigkeit) *Wenn AMO_i vor AMO_j ausgeführt wird und AMO_i eine Variable als Argument verwendet, die AMO_j definiert, dann liegt eine Antiabhängigkeit zwischen diesen beiden AMOs vor.*

Zur Verdeutlichung sind in Abb. 2.2 Beispiele zu den oben aufgeführten Arten von Datenabhängigkeiten angegeben. Für detailliertere Informationen zu diesem Thema möchten wir an dieser Stelle auf [ASU86, Muc97] verweisen.

Bevor in Abschnitt 2.3 näher auf das GeLIR-System eingegangen wird, werden im folgenden Abschnitt zunächst einige existierende Compiler-Zwischendarstellungen vorgestellt.

2.2 Zwischendarstellungen existierender Compilersysteme

Die Wahl der Zwischendarstellung und der damit verbundenen Tools zur Durchführung von Analysen und Optimierungen stellt einen wichtigen Aspekt bei der Entwicklung eines

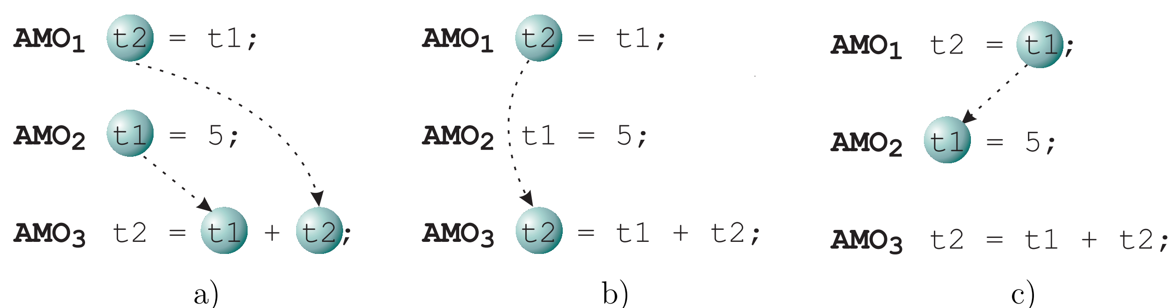


Abb. 2.2: Beispiele für unterschiedliche Arten der Datenabhängigkeit: a) Datenflussabhängigkeit b) Ausgabeabhängigkeit c) Antiabhängigkeit

Compilers dar. So sind neben den bereits zu Beginn dieses Kapitels genannten Anforderungen weitere Aspekte zu berücksichtigen. Z.B. sollte das verwendete Compilersystem für Forschungszwecke frei verfügbar sein, über eine LIR verfügen, die die Modellierung von irregulären Prozessorarchitekturen erlaubt und zusätzlich eine einfache Anpassung bereits existierender und die Einbindung neuer phasengekoppelter Optimierungstechniken erlaubt.

Aufgrund der Vielzahl existierender Compilersysteme und derer zugrunde gelegten Zwischendarstellungen kann hier nur eine kleine Auswahl der in der Forschung verwendeten Tools vorgestellt werden. Als Quellen wurden neben den jeweils angegebenen Referenzen, zusätzlich [LM01] und [Bas01] verwendet.

- *CoSy* [CoS] stellt ein kommerzielles Produkt dar, das auch zu Forschungszwecken verwendet werden darf. Es umfasst Front-Ends für die Sprachen C/C++, Java, DSP-C, Fortran 95 und HPF, eine Reihe von Standardoptimierungen und eine Anbindung an einen Back-End-Generator. Weitere Unterstützung im Back-End ist u.a. in Form eines Schedulers und eines Registerallokators vorhanden. CoSy ermöglicht zwar die schnelle Entwicklung neuer Compiler, basiert aber auf Standardtechniken für GPPs, mit nur geringer Unterstützung für irreguläre Architekturen und fehlender Phasenkopplung.

- *SUIF* [SUI] wurde an der Stanford Universität zu Forschungszwecken entwickelt und besteht hauptsächlich aus einem Front-End mit einigen Standardoptimierungen. Es werden zwei maschinenunabhängige Zwischendarstellungen (High- und Low-Level SUIF) angeboten, wobei letztere aus der High-Level-Darstellung gewonnen wird, indem vorhandene Hochsprachenkonstrukte, wie z.B. Schleifen oder Arrayzugriffe, durch assemblernahe Konstrukte ersetzt werden. Mit der Bezeichnung „Low-Level“ ist also keine maschinenabhängige Darstellung verbunden, wie der Name vermuten lassen könnte. Aus beiden Darstellungen besteht die Möglichkeit, wiederum Code der Ursprungssprache zu erzeugen. Dies ermöglicht unter Verwendung

eines herkömmlichen Standardcompilers eine einfache Validierung von Optimierungen und Transformationen. Die in [Fal02, FM03] beschriebenen Optimierungen machen z.B. Gebrauch von dieser Möglichkeit und können auf diese Weise für eine Reihe unterschiedlicher Prozessoren in Form eines Vorverarbeitungsschrittes dienen. SUIF bietet keine Möglichkeit der Spezifikation von architekturspezifischen Merkmalen und somit keine besonderen Modellierungsmöglichkeiten für irreguläre Architekturen, so dass sich eine Verwendung von SUIF „nur" auf das Front-End und Middle-End beschränken würde.

- *Zephyr* [JP01] wurde an der Universität von Virginia in Kooperation mit der Princeton Universität entwickelt und stellt ein retargierbares Back-End dar, das als Front- und Middle-End SUIF benutzt. Die SUIF-IR wird mittels eines Code-Expanders in eine Low-Level Zwischendarstellung RTL (Register Transfer Lists) überführt. Diese Darstellung entspricht zunächst unoptimiertem Assemblercode, der mittels des Tools VPO (Very Portable Optimizer) schrittweise verbessert wird. VPO führt dazu eine Instruktionsauswahl und maschinenunabhängige Standardoptimierungen, gefolgt von einer Registerallokationsphase in einer Schleife durch. Auch wenn in [JP01] mit Hilfe von Zephyr ein Compiler für einen DSP vorgestellt wurde, ist Zephyr vornehmlich für den Einsatz von GPPs geeignet, da die Optimierungsphasen unabhängig voneinander durchgeführt werden.

- *LANCE* [Leu00a] ist ein an der Universität Dortmund entwickeltes Compilersystem, das neben einem ANSI-C Front-End und einer Reihe von maschinenunabhängigen Standardoptimierungen auch eine Anbindung an den Codegenerator-Generator Olive [Tji93] zur Verfügung stellt. Die Zwischendarstellung ist als Drei-Adressformat realisiert. Analog zu SUIF besteht auch hier die Möglichkeit, die interne Darstellung mit Hilfe eines herkömmlichen Compilers zu validieren. Die Entwicklung neuer Back-Ends wird neben der Anbindung an Olive durch die Adaption vorhandener Techniken z.B. zur Registerallokation, Codekompaktierung (für VLIW-Architekturen) und optimierten Adresszuweisung unterstützt. Allerdings sind auch hier keine Konzepte zur Unterstützung stark irregulärer Architekturen und zur Phasenkopplung vorhanden.

- *Trimaran* [Tri] ist mit dem Hauptziel der Durchführung von maschinenabhängigen Optimierungen zur Ausnutzung von Parallelität auf Instruktionsebene entwickelt worden. Das Trimaran-System besteht aus einem Front-End (IMPACT), einem Back-End (ELCOR) und einer Zwischendarstellung, auf der Analysen, Optimierungen und Transformationen ausgeführt werden. Die Maschinenbeschreibungen werden mit Hilfe von MDES (Machine Description) vorgenommen. Zusätzlich existiert ein ASCII-basiertes Zwischenformat (Rebel), das es ermöglicht, beliebige

Zwischenzustände zu speichern und wieder einzulesen. Ein zyklengenauer Simulator der HPL-PD-Architektur ist mittels einer Hardwarebeschreibung konfigurierbar und liefert u.a. Informationen über die Ausführungszeit, Verzweigungshäufigkeiten und Ressource-Verwendungen. Aufgrund der besonderen Ausrichtung auf VLIW-Architekturen ist eine Anpassung der vorhandenen Tools an stark irreguläre Architekturen sehr schwierig.

- *SPAM* [SPA] wurde zur Entwicklung von retargierbaren Compilern für eingebettete Prozessoren, insbesondere DSPs, entwickelt. Als Front- und Middle-End wird SUIF verwendet. Dem ist ein Back-End (TWIF) nachgeschaltet, das insbesondere eine Bibliothek von maschinenunabhängigen Optimierungen enthält, die durch Angabe prozessorspezifischer Parameter auf neue Zielarchitekturen angepasst werden können. Neben einigen Standardoptimierungen stehen Techniken zur Durchführung einer baumbasierten Codeselektion für irreguläre Architekturen, zur Ausnutzung der Spezialbefehle von Adressgenerierungseinheiten, zur Ausnutzung mehrerer Speicherbänke und zur Durchführung einer Codekompaktierung zur Verfügung.

- *PROPAN* [Käs00] stellt ein System dar, das die Generierung von machinenabhängigen Postpass-Optimierungen, insbesondere für irreguläre Architekturen, erlaubt. Die Beschreibung der Zielarchitektur wird dabei mittels TDL (Target Description Language) vorgenommen. Des Weiteren ist an PROPAN ein phasengekoppelter Optimierer angebunden, der eine globale Instruktionsanordnung und Registerallokation (ohne Berücksichtigung von Spillcode) auf Basis der ganzzahlig linearen Programmierung durchführt [Käs01].

- *COCOON* [Bas01] stellt ein Codegenerierungs-System mit dem Ziel der Entwicklung phasengekoppelter maschinenabhängiger Optimierungen für irreguläre Architekturen dar. Alle implementierten Techniken sind auf der Basis der Constraint-Logikprogrammierung (CLP) entwickelt worden und arbeiten auf der generischen Zwischendarstellung CoLIR. Diese erlaubt die Darstellung von abstrakten Maschinenoperationen der IR und von alternativen Maschinenprogrammen. Die durch Architektureigenschaften vorgegebenen Einschränkungen, wie eingeschränkte Parallelität und irreguläre Datentransferwege, werden durch die Formulierung von Constraints spezifiziert.

Die einzigen der hier dargestellten Compilersysteme, deren Zwischendarstellungen eine Unterstützung von irregulären Architekturen aufweisen, sind SPAM, PROPAN und COCOON. Insbesondere die von COCOON zugrunde gelegte Zwischendarstellung CoLIR bietet durch die Möglichkeit der Darstellung alternativer Maschinenprogramme eine sehr gute Unterstützung zur Entwicklung von phasengekoppelten Optimierungstechniken. Die

im folgenden Abschnitt beschriebene Zwischendarstellung GeLIR stellt eine Weiterentwicklung von CoLIR dar.

2.3 Low-Level Zwischendarstellung (GeLIR)

GeLIR [GeL] stellt eine in C++ programmierte Zwischendarstellung von Compilern dar, mit der neben einer maschinenunabhängigen Darstellung des Quellprogramms auch die Möglichkeit der Darstellung von alternativen Maschinenprogrammen besteht (s. auch Abb. 2.3). Prozessorspezifische Merkmale können losgelöst von der Programmdarstellung in generischer Form abgelegt werden, wodurch die Entwicklung von Optimierungen für eine breite Klasse von Prozessoren ermöglicht wird. Da mit den GeLIR-Datenstrukturen auch die Darstellung von Maschinenprogrammen möglich ist, kann als Ausgangspunkt sowohl ein Hochsprachenprogramm als auch Assemblercode dienen, wobei bislang jedoch nur der erste Weg mittels Schnittstellen unterstützt wird.

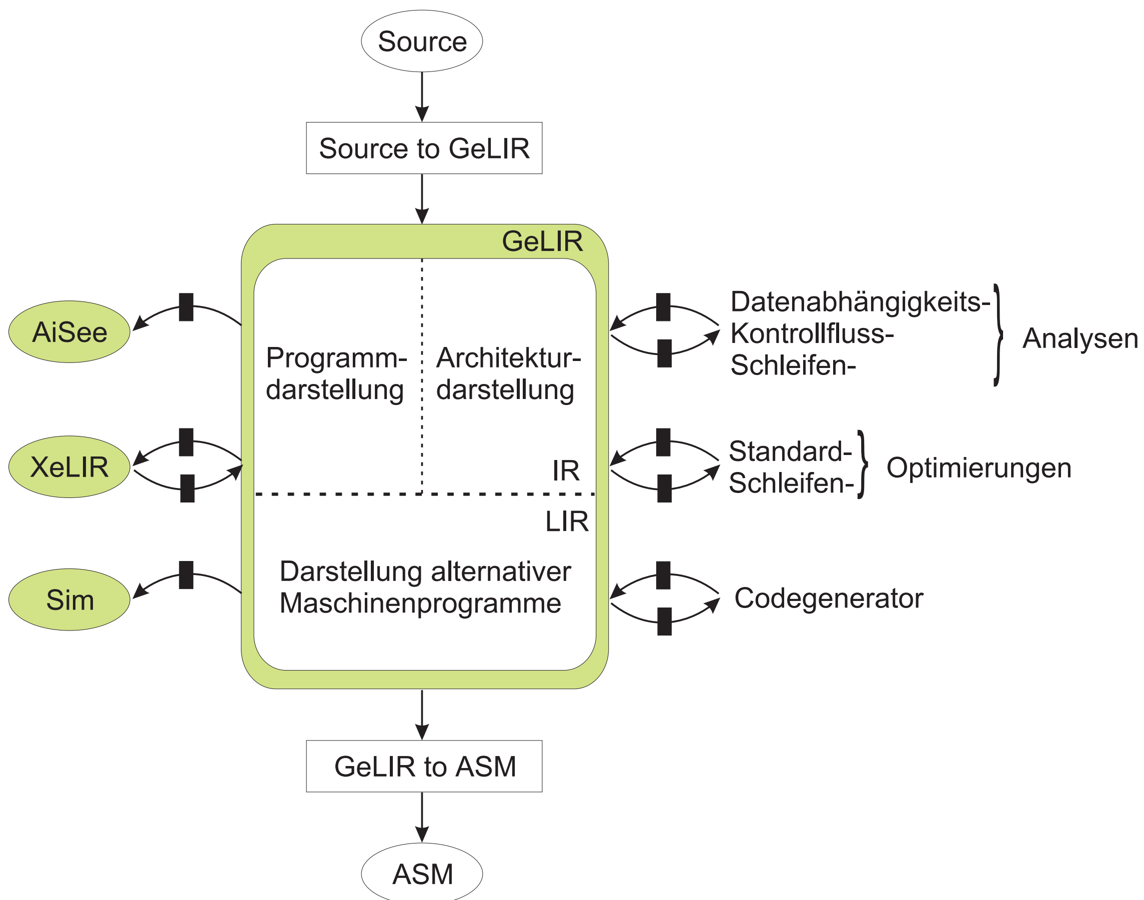


Abb. 2.3: Übersicht der GeLIR-Entwicklungsumgebung

Irreguläre Architekturen zeichnen sich dadurch aus, dass die Ausführung von Befehlen mit der Einhaltung einer Vielzahl von Randbedingungen (Constraints) verbunden ist. Dies

kann z.B. Restriktionen bezüglich der Verwendung bestimmter Ressource-Kombinationen oder der parallelen Ausführungsmöglichkeiten betreffen. Mittels eines in GeLIR integrierten Constraintpropagierungs-Algorithmus (s. auch Abschnitt 2.3.4) wird deswegen ein Mechanismus zur Verfügung gestellt, mit dem die Einhaltung einiger dieser Constraints sichergestellt werden kann.

Neben diversen Analysen und Optimierungen, die die Entwicklung neuer Compiler unterstützen, sind des Weiteren Schnittstellen (durch schwarze Kästen in Abb. 2.3 angedeutet) zum graphischen Visualisierungsprogramm aiSee [aiS], zum XML-basierten Textformat XeLIR [Fie01] und zu einer Simulations- und Debuggingumgebung vorhanden. Der im Rahmen dieser Arbeit entwickelte Codegenerator besitzt ebenfalls Schnittstellen zur GeLIR-Entwicklungsumgebung und kann somit auf deren volle Funktionalität zurückgreifen.

In den nachfolgenden Abschnitten wird zunächst näher auf den eigentlichen GeLIR-Kern mit der Programm- und Architekturdarstellung, der Darstellung alternativer Maschinenprogramme und dem Mechanismus zur Constraintpropagierung eingegangen. Dem schließt sich eine kurze Beschreibung der vorhandenen Analysen, Optimierungen und Tools an. Eine Beschreibung des Codegenerators erfolgt in Kapitel 3.

2.3.1 Programmdarstellung

Zur Programmdarstellung werden in GeLIR eine Reihe von C++-Klassen verwendet, die sich an der allgemeinen Struktur von Programmen imperativer Programmiersprachen orientieren. In Abb. 2.4 betrifft dies zunächst die schattiert dargestellten Klassen *LirGeLIR*, *LirFun*, *LirBB*, *LirMI* und *LirMO*. Die Pfeile geben hier an, dass eine bestimmte Klasse (z.B. LirGeLIR) ein oder mehrere Objekte einer anderen Klasse (z.B. LirFun) benutzt.

In den einzelnen Objekten dieser Klassen werden eine Reihe von Informationen verwaltet, auf die im Folgenden näher eingegangen wird:

- *LirGeLIR:* Ein Objekt dieser Klasse enthält neben den in einem Programm vorkommenden Funktionen eine globale Symboltabelle (*LirSTab*), deren enthaltene Symboltabelleneinträge (*LirSTabEntry*) Informationen (wie z.B. Typ) über die verwendeten Programmvariablen enthalten. Falls vorhanden, werden weitere Informationen über Positionen im Speicher oder über Initialisierungswerte globaler oder statischer Variablen in eigenständigen Objekten der Klassen *LirMem* bzw. *LirInitValue* hinterlegt.

- *LirFun:* Während in der Symboltabelle eines LirGeLIR-Objektes nur global verwendete Variablen verwaltet werden, werden in der Klasse LirFun alle lokal in dieser Funktion verwendeten Variablen verwaltet. Zusätzlich werden hier u.a. Informationen über Aufruf- und Rückgabeparameter dieser Funktion hinterlegt. Des Weiteren

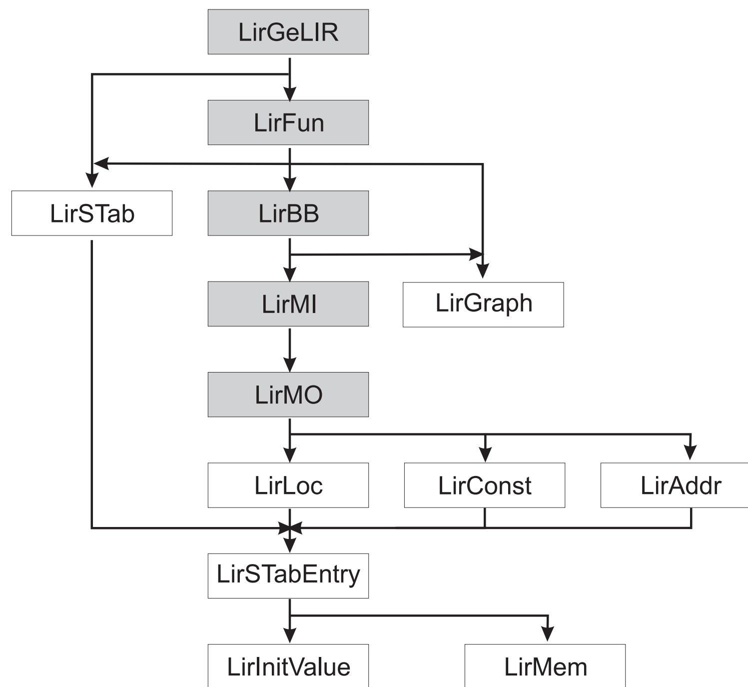


Abb. 2.4: Klassenübersicht zur Programmdarstellung

werden graphbasierte Zwischendarstellungen (*LirGraph*) für Kontrollfluss und globale Datenflussabhängigkeiten verwaltet.

- *LirBB:* Mit einem Objekt dieser Klasse wird ein Basisblock einer bestimmten Funktion repräsentiert. Neben der Verwaltung der MIs sind in diesem Objekt graphbasierte Zwischendarstellungen zur Darstellung von Datenabhängigkeiten (Datenfluss-, Ausgabe- und Antiabhängigkeiten) vorhanden. Zusätzlich werden Listen von Variablen (V_{in} und V_{out}) verwaltet, mit deren Hilfe Aussagen über den (globalen) Datenfluss zwischen den Basisblöcken gemacht werden können. So enthält die V_{in}-Liste Variablen, die bis zu ihrer ersten Verwendung in diesem Basisblock nicht definiert werden und daher entweder in einem anderen Basisblock zuvor definiert werden, oder ohne vorherige Initialisierung verwendet werden. Die V_{out}-Liste enthält Variablen, deren Werte am Ende des Basisblocks noch Gültigkeit haben, also nicht neudefiniert worden sind.

- *LirMI:* Zur expliziten Darstellung von Parallelität kann jede MI mehrere MOs aufnehmen. Allerdings dürfen zur Wahrung der semantischen Korrektheit von Programmen – mit Ausnahme von partiellen MOs – zwischen diesen keine Datenabhängigkeiten vorhanden sein. Die Einhaltung dieser und weiterer Randbedingungen (z.B. Ressource-Constraints) zur Parallelisierung von Maschinenoperationen ist Aufgabe des Codegenerators.

- *LirMO:* Die Darstellung einer AMO erfolgt in Drei-Adresscode, so dass durch jedes LirMO-Objekt genau eine elementare Anweisung des Quellprogramms umgesetzt wird. Dies können z.B. Anweisungen wie Addition, Shiftleft, Load, Move oder Copy sein. Mit einer Copy-Anweisung kann ausgedrückt werden, dass keine, eine oder mehrere Move-Anweisungen ausgeführt werden müssen. Häufig ist zur Durchführung der Codegenerierung eine weitere Klassifizierung dieser Objekte erforderlich. So können z.B. Adressberechnungen speziell markiert werden, um anzudeuten, dass diese Anweisung auf einer speziellen Funktionseinheit (z.B. AGU) ausgeführt werden soll. Die zuvor erwähnten Variablen (V_{in} und V_{out}) werden ebenfalls mittels eines LirMO-Objektes dargestellt. Zur Vermeidung unnötiger Sonderfälle stellen Pointer-Ausdrücke (Load und Stores), Konstanten und Adressen eigene AMOs dar. Ausdrücke der Form `x = 5 + y;` werden demnach durch zwei AMOs `t1 = 5;` und `x = t1 + y;` ausgedrückt. Informationen über vorhandene Definitionen und Argumente werden in *LirLoc*-Objekten hinterlegt, die u.a. einen Symboltabelleneintrag enthalten. In den Argumentlokationen können zur Darstellung komplexer MOs Verweise auf partielle MOs abgelegt werden.

2.3.2 Architekturdarstellung

Um eine Wiederverwendung von Optimierungen im Back-End für unterschiedliche Zielarchitekturen zu unterstützen, stellt GeLIR Datenstrukturen zur Verfügung, in denen architekturspezifische Merkmale abgelegt und abgerufen werden können. Alle spezifizierten Eigenschaften der zugrunde gelegten Zielarchitektur werden als separate Objekte (*LirResource*, *LirOperation* und *LirType*) in einem zentralen *LirTarget*-Objekt gespeichert (s. Abb. 2.5):

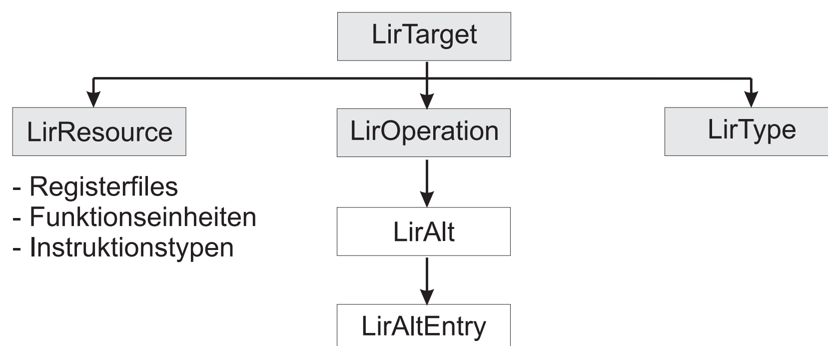


Abb. 2.5: Klassenübersicht zur Zielarchitekturdarstellung

LirType: Typ-Spezifikationen können in GeLIR beliebig komplex vorgenommen werden. So besteht die Möglichkeit, komplexe Datentypen, wie z.B. `int*` oder Funktionstypen wie `int = (int` $\times$ `int` $\times$ `int)` durch mehrere einfache Typen zusammenzusetzen. Da ins-

besondere bei DSP-Befehlssätzen häufig eine bitgenaue Typangabe erforderlich ist, kann neben der Größe auch die Bezugsgröße *Bit* oder *Byte* spezifiziert werden. Aufgrund der gewöhnlicherweise vorhandenen Abweichungen von *abstrakten* Typen des Quellprogramms und *realen* Typen der Zielmaschine, können die von der Zielmaschine unterstützten Typen explizit als solche markiert werden.

LirResource: Die Ressourcen einer Zielarchitektur werden in GeLIR in die drei folgenden Bereiche unterteilt:

- *Registerfile*: Mit der Spezifizierung eines Registerfiles können insbesondere Angaben über die Größe (Anzahl enthaltener Registerelemente) sowie den Datentyp (z.B. int oder float) gemacht werden, der von dieser Ressource aufgenommen werden kann. Um die Implementierung generischer Optimierungen zu ermöglichen, werden alle Register-Ressourcen klassifiziert (z.B. Hauptspeicher, (Adress-)Register oder flüchtige Ressource). Je nach Bedarf können Registerfiles weiter in Registerelemente aufgesplittet werden, die wiederum als eigenständige Ressourcen beschrieben werden können. Dies ist insbesondere erforderlich, wenn bestimmte Elemente eines Registerfiles gegenüber anderen desselben Registerfiles unterschiedliche Verwendungsmöglichkeiten besitzen. Zum Beispiel ist es beim M3-DSP erforderlich, dass die im SISD-Modus zu verarbeitenden Daten in Registern des Datenpfades 0 vorliegen.

- *Funktionseinheiten*: Vorhandene Funktionseinheiten der Zielarchitektur werden ebenfalls in separaten Objekten gekapselt. Die Angabe von Attributen erscheint hier bislang nur für die Anzahl der zur Verfügung stehenden Instanzen einer bestimmten Funktionseinheit sinnvoll. Bei Bedarf können analog zu der Spezifizierung von Registerfiles weitere Angaben gemacht werden.

- *Instruktionstypen*: Instruktionstypen dienen der Modellierung von parallelen Ausführungsmöglichkeiten, indem zwei MOs nur dann derselben MI zugeordnet werden dürfen, wenn diese denselben Instruktionstypen besitzen.

LirOperation: Analog zur Spezifizierung von Typen wird zwischen den vordefinierten abstrakten GeLIR-Operationen (AMOs) zur maschinenunabhängigen Darstellung von Programmen und den auf der Zielarchitektur vorhandenen Operationen (MOs) unterschieden. Zusätzlich besteht mit Hilfe eines *LirAlt*-Objektes die Möglichkeit der Spezifizierung von faktorisierten MOs (FMOs), durch die Angabe alternativer Mengen von Ressourcen. Dies stellt ein sehr wichtiges Modellierungsmittel dar. So können hierdurch zum einen alle für eine bestimmte Lokation (Definition oder Argument) verwendbaren Register-Ressourcen abgefragt werden, zum anderen kann aber auch ermittelt werden, welche weiteren Ressourcen miteinander auf welche Art und Weise kombiniert werden dürfen. Da bei irregulären

Architekturen nicht immer alle vorhandenen Ressourcen beliebig miteinander kombiniert werden dürfen, können alternative Ausführungsmöglichkeiten auf mehrere *LirAltEntry*-Objekte aufgeteilt werden. Eine beliebige Kombination aller in einem solchen Objekt angegebenen Ressourcen ist dann möglich. Um abweichende Ausführungszeiten einer Maschinenoperation auf unterschiedlichen Funktionseinheiten berücksichtigen zu können, ist für konkrete *LirAltEntry*-Objekte u.a. die Angabe der erforderlichen Ausführungszeit und Latenzzeit möglich.

Die Modellierung von Operationen der Zielmaschine mit Hilfe von FMOs wird in Abb. 2.6 und 2.7 beispielhaft anhand der Multiplikation und der Addition des M3-DSPs veranschaulicht:

	LirAltEntry 1
Op ={MUL} FU ={DMU} IT ={1} Def ={ACCU,'*'} Arg1={A,B,'CNST1','CNST2'} Arg2={A,C,D,ACCU}	**Attribute:** Exec-Time = 1 Latency = 1 Swapped-Args = false

	LirAltEntry 2
Op ={MUL} FU ={DMU} IT ={1} Def ={ACCU,'*'} Arg1={A,C,D,ACCU} Arg2={A,B,'CNST1','CNST2'}	**Attribute:** Exec-Time = 1 Latency = 1 Swapped-Args = true

Abb. 2.6: M3-DSP: Multiplikation

	LirAltEntry 1
Op ={ADD} FU ={DMU} IT ={1} Def ={ACCU} Arg1={A,B,ACCU,'CNST0'} Arg2={A,C,D,ACCU,'*'}	**Attribute:** Exec-Time = 1 Latency = 1 Swapped-Args = false

	LirAltEntry 2
Op ={ADD} FU ={DMU} IT ={1} Def ={ACCU} Arg1={A,C,D,ACCU,'*'} Arg2={A,B,ACCU,'CNST0'}	**Attribute:** Exec-Time = 1 Latency = 1 Swapped-Args = true

Abb. 2.7: M3-DSP: Addition

Da es sich bei beiden Operationen um kommutative Operationen handelt, sind jeweils zwei *LirAltEntry*-Objekte vorhanden, die Mengen von Ressourcen enthalten, bei denen jeweils die Registerverwendungen der Argumente, gegeben in den Mengen (`Arg1` und `Arg2`), gespiegelt wurden. Da bei der Ausgabe von Assemblercode die Reihenfolge der ausgegebenen Ressourcen eine wichtige Rolle spielt, wird dies zusätzlich durch ein entsprechendes Attribut (`Swapped-Args`) vermerkt. Das Ergebnis der Operation kann nur einem der in der Menge `Def` enthaltenen Ressourcen zugewiesen werden. Gültige Ressource-Kombinationen für (`Def` $\times$ `Arg1` $\times$ `Arg2`) von LirAltEntry 1 in Abb. 2.6 sind z.B. (`ACCU, A, C`) und (`'*', A, C`) aber nicht (`'*', 'CNST2', B`). Zur Modellierung von komplexen Operationen wie der MAC-Operation werden die bereits erwähnten flüchtigen Ressourcen verwendet (hier: `'*'`). Entsprechend der semantischen Bedeutung der MAC-Operation kann diese Ressource als Definition einer Multiplikation und als Argument einer Addition verwendet werden. In Abb. 2.8 ist dies anhand der Überdeckung einer Multiplikation und einer Addition mit Ressourcen verdeutlicht[1].

Des Weiteren darf eine in der Menge `Op` enthaltene Operation nur auf den in `FU` gegebenen Funktionseinheiten ausgeführt werden. Eine parallele Ausführung einer dieser Operatio-

[1] Die Abbildung wurde mit Hilfe des Visualisierungsprogramms aiSee [aiS] generiert (s. auch Abschnitt 2.3.6).

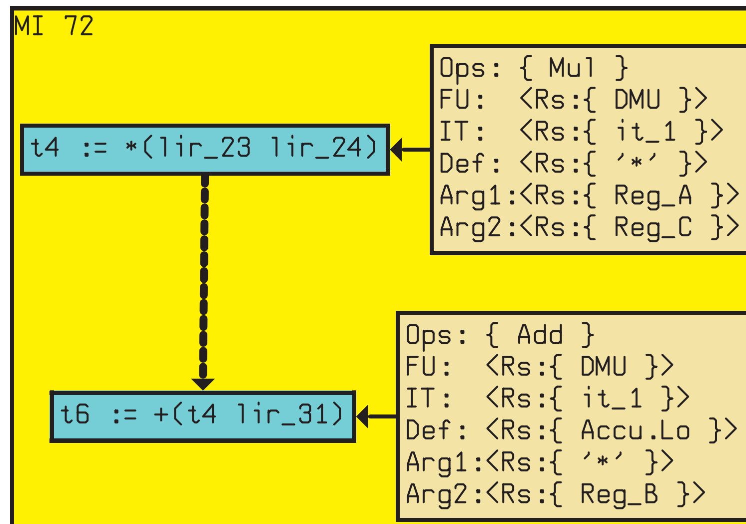


Abb. 2.8: Beispiel zur Modellierung einer MAC-Operation

nen darf nur mit Operationen erfolgen, die denselben in IT gegebenen Instruktionstypen aufweisen.

Um eine Verbindung zwischen AMOs und MO herzustellen, werden bei den AMOs Ressource-Alternativen von MOs eingetragen, die eine semantisch äquivalente Umsetzung erlauben. So könnten z.B. bei einer AMO, die eine Multiplikation realisiert, die Ressource-Alternativen der MO aus Abb. 2.6 eingetragen werden. Wären weitere Umsetzungsmöglichkeiten der Multiplikation möglich, z.B. durch eine MO, die auf einer anderen Funktionseinheit ausgeführt wird, dann könnten diese ebenfalls eingetragen werden. Anhand dieser vorgenommenen Spezifikationen können dann z.B. im Rahmen der Constraintpropagierung (s. Abschnitt 2.3.4) Einschränkungen von Ressource-Alternativen durchgeführt werden, die ungültige Auswahlkombinationen von Ressourcen vermeiden.

2.3.3 Darstellung alternativer Maschinenprogramme

Die Darstellung alternativer Maschinenprogramme stellt eine wichtige Eigenschaft von GeLIR dar und vereinfacht insbesondere die Entwicklung von phasengekoppelten Optimierungen. Eine prinzipielle Vorgehensweise könnte darin bestehen, zu Beginn der Codegenerierung für eine gegebene GeLIR-Programmdarstellung alle möglichen Maschinenprogramme darzustellen. Die Aufgabe des Codegenerators würde dann in der Auswahl des Programms liegen, das eine vorgegebene Kostenfunktion optimiert. Dabei liegt ein konkretes (gültiges) Maschinenprogramm vor, wenn alle Ressource-Mengen bis auf ein Element eingeschränkt wurden, eine Anordnung der MOs zu MIs vorgenommen wurde und zusätzlich alle Randbedingungen bezüglich Ressourcen und Datenabhängigkeiten eingehalten werden. Die Auswahl eines Maschinenprogramms kann je nach verwendeter

Codegenerierungs-Technik in einem Schritt oder auch in mehreren Schritten erfolgen, indem jeweils nur Einschränkungen bestimmter Ressource-Mengen vorgenommen werden. In Abb. 2.9 ist für ein gegebenes GeLIR-Codefragment mit Drei-Adressbefehlen eine äquivalente Darstellung in Form eines Datenflussgraphen abgebildet. Mit Hilfe der an die jeweiligen Graphknoten gebundenen Ressource-Alternativen besteht nun die Möglichkeit, unterschiedliche Maschinenprogramme zu erzeugen. In diesem Beispiel wird davon ausgegangen, dass zum aktuellen Zeitpunkt bereits teilweise Einschränkungen der Ressource-Alternativen vorgenommen worden sind.

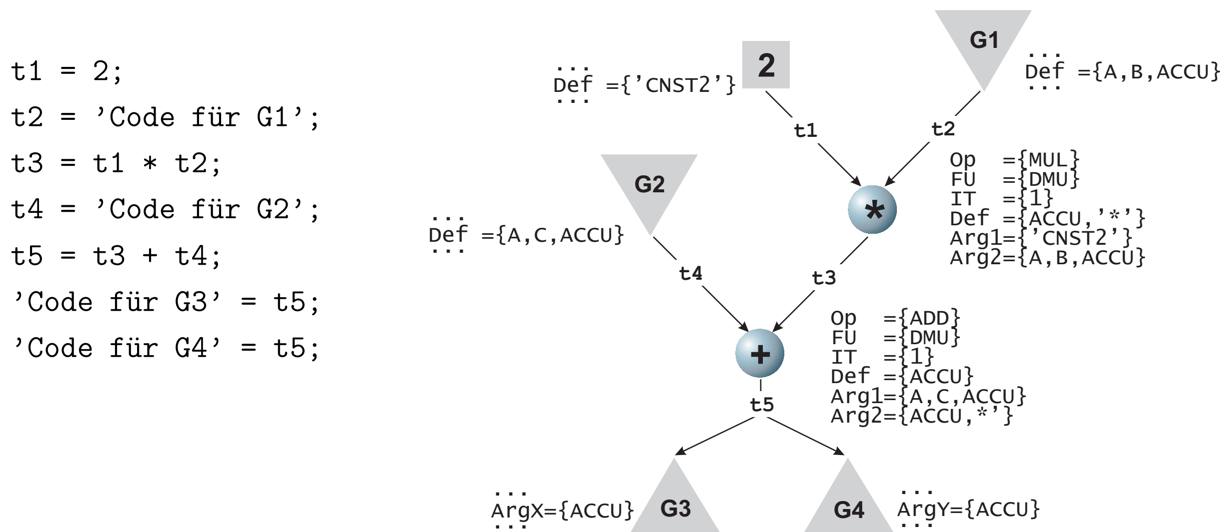


Abb. 2.9: Alternative Maschinenprogramme für ein gegebenes GeLIR-Codefragment

Ohne die ursprüngliche Ausführungsreihenfolge der Anweisungen zu ändern, könnten z.B. die in Abb. 2.10 dargestellten Ressource-Zuweisungen vorgenommen werden. Bei allen drei Programmen wurde die Konstante 2 der flüchtigen Ressource `'CNST2'` zugewiesen, da diese als Einzige ausgewählt werden konnte. Da es sich hierbei um eine flüchtige Ressource handelt, erfolgt also keine Zwischenspeicherung des Wertes in einem Register und geht so auf direktem Weg als Operand in die Multiplikation ein. Analog dazu wird in Programm 3 das Ergebnis der Multiplikation in die flüchtige Ressource `'*'` geschrieben und in der nachfolgenden Addition verwendet, was der Ausführung einer MAC-Operation entspricht.

Bei genauerer Betrachtung der Programme fällt jedoch auf, dass die getroffenen Ressource-Zuweisungen zu fehlerhaftem Code führen würden:

- In Programm 1 wird mit der vierten Anweisung `ACCU = 'Code für G2'` die zuvor beschriebene Register-Ressource `ACCU` überschrieben, obwohl diese noch benötigt wird.
- In Programm 2 definiert die zweite Anweisung die Register-Ressource B, obwohl die davon datenabhängige dritte Anweisung das entsprechende Argument in der

```
'CNST2' = 2;
A = 'Code für G1';
ACCU = 'CNST2 * A;
ACCU = 'Code für G2';
ACCU = ACCU + ACCU;
'Code für G3' = ACCU;
'Code für G4' = ACCU;
```

Programm 1

```
'CNST2' = 2;
B = 'Code für G1';
ACCU = 'CNST2' * A;
C = 'Code für G2';
ACCU = ACCU + C;
'Code für G3' = ACCU;
'Code für G4' = ACCU;
```

Programm 2

```
'CNST2' = 2;
B = 'Code für G1';
'*' = 'CNST2' * B;
C = 'Code für G2';
ACCU = '*' + C;
'Code für G3' = ACCU;
'Code für G4' = ACCU;
```

Programm 3

Abb. 2.10: Beispielprogramme

Register-Ressoure A erwartet. Um diesen Fehler zu vermeiden, muss garantiert werden, dass zwischen der Definition einer MO und dem entsprechenden Argument einer davon datenabhängigen MO, gültige Datentransferwege existieren (→ *Kantenkonsistenz*).

- Im Gegensatz zu den beiden vorherigen Programmen führen die in Programm 3 vorgenommenen Zuweisungen auf den ersten Blick zu keinen Konflikten. Allerdings wird bei näherer Betrachtung der dritten Anweisung `'*' = 'CNST2' * B` schnell klar, dass die verwendeten Ressource-Alternativen in dieser Weise nicht miteinander kombiniert werden dürfen, da diese Kombination in keiner der in Abb. 2.6 angegebenen *LirAltEntry*-Objekte der Spezifikation der M3-Multiplikation vorkommt (→ *Knotenkonsistenz*).

Allgemein kann gesagt werden, dass die Berücksichtigung und Einhaltung solcher Ressource-Konflikte zu den Aufgaben des Codegenerators gehört. Um jedoch die Implementierung neuer Codegenerierungs-Techniken zu vereinfachen, sind auf den GeLIR-Datenstrukturen Algorithmen implementiert, die bei sachgemäßer Anwendung Ressource-Konflikte, wie sie in den Programmen 2 und 3 vorkommen, vermeiden. Da diese Algorithmen eine wichtige Rolle bei der Durchführung der Codegenerierung spielen, wird im nachfolgenden Abschnitt kurz darauf eingegangen und deren prinzipielle Arbeitsweise erläutert.

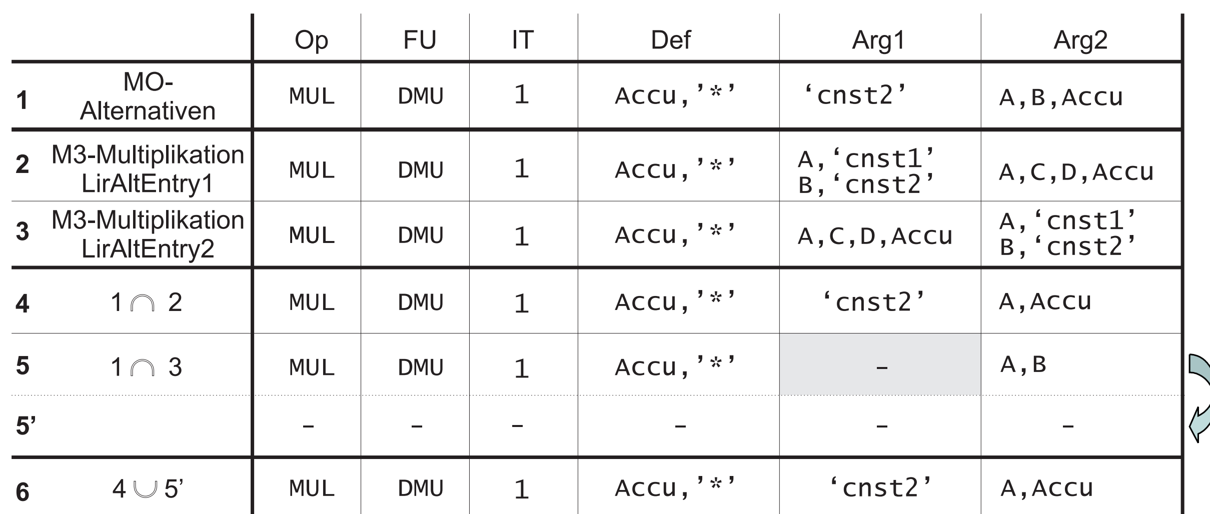
2.3.4 Constraintpropagierung

Wie im vorherigen Abschnitt erläutert wurde, müssen bei der Durchführung der Codegenerierung eine Reihe von Randbedingungen eingehalten werden. Während bei Architekturen mit homogenen Befehlssätzen die zuvor beschriebenen Probleme der Kanten- und Knotenkonsistenz eine eher untergeordnete Rolle spielen, werden zur Codegenerierung für DSPs Mechanismen benötigt, mit deren Hilfe Probleme, wie sie bei den Programmen 2

und 3 in Abb. 2.10 vorgekommen sind, vermieden werden können. Während die Kantenkonsistenz das Vorhandensein von Datentransferwegen zwischen zwei datenflussabhängigen MOs zusichern soll, soll durch die Knotenkonsistenz für einen speziellen Knoten (MO) gewährleistet werden, dass nur gültige Ressource-Kombinationen verwendet werden. Für das Programm 3 in Abb. 2.9 wäre es also sinnvoll gewesen, den Datenflussgraphen vor Auswahl der Ressourcen auf Knoten- und Kantenkonstistenz hin zu überprüfen.

Knotenkonsistenz

Zur Wahrung der Knotenkonsistenz muss ein Abgleich der aktuell zur Auswahl stehenden Ressource-Alternativen einer AMO mit den spezifizierten Ressource-Kombinationen vorgenommen werden. Durch die Anwendung der Mengenoperationen *Durchschnitt* und *Vereinigung* kann die Knotenkonsistenz effizient sichergestellt werden. In Abb. 2.11 ist dies am Beispiel des Multiplikations-Knotens aus Abb. 2.9 veranschaulicht.



		Op	FU	IT	Def	Arg1	Arg2
1	MO-Alternativen	MUL	DMU	1	Accu,'*'	'cnst2'	A,B,Accu
2	M3-Multiplikation LirAltEntry1	MUL	DMU	1	Accu,'*'	A,'cnst1' B,'cnst2'	A,C,D,Accu
3	M3-Multiplikation LirAltEntry2	MUL	DMU	1	Accu,'*'	A,C,D,Accu	A,'cnst1' B,'cnst2'
4	1 ∩ 2	MUL	DMU	1	Accu,'*'	'cnst2'	A,Accu
5	1 ∩ 3	MUL	DMU	1	Accu,'*'	-	A,B
5'		-	-	-	-	-	-
6	4 ∪ 5'	MUL	DMU	1	Accu,'*'	'cnst2'	A,Accu

Abb. 2.11: Beispiel der Vorgehensweise zur Wahrung der Knotenkonsistenz

In der ersten Zeile sind die aktuell auswählbaren Ressourcen für den Multiplikations-Knoten angegeben, während in den Zeilen 2 und 3 die laut Spezifikation zulässigen Ressource-Kombinationen für die Multiplikations-MO (s. auch Abb. 2.6) aufgelistet sind. Letztere dienen im Prinzip als Template für die in Zeile 1 gegebenen Ressource-Alternativen. Die Zeilen 4 und 5 enthalten die Ergebnisse der angegebenen Durchschnittsbildung der Ressourcen und stellen jeweils gültige Ressource-Kombinationen dar. Als Besonderheit ist in Zeile 5 jedoch zu erkennen, dass eine der gegebenen Ressource-Mengen (`Arg1`) leer ist. Dies bedeutet, dass bezüglich der mit LirAltEntry 2 spezifizierten Ressource-Kombinationen keine gültige Übereinstimmung existiert, so dass alle anderen Ressourcen dieser Zeile ebenfalls als leer angenommen werden müssen (s. Zeile 5'). Die letztendlich auswählbaren MO-Alternativen werden dann durch Bildung der Vereinigung der Zeilen 4 und 5' gebildet.

Da keine Abhängigkeiten zu anderen Graphknoten bestehen, reicht es zur Sicherstellung der Knotenkonsistenz aus, jeden Knoten des Graphen einmal zu betrachten. Dies sieht bei der Wahrung der Kantenkonsistenz anders aus:

Kantenkonsistenz
Wenn n_i und n_j zwei Knoten des Datenflussgraphen sind, zwischen denen eine Datenflussabhängigkeit besteht, dann muss zugesichert werden, dass n_i nur solche Ressourcen definieren kann, die bei n_j an der entsprechenden Argumentposition noch zur Auswahl stehen. Umgekehrt darf n_j das zu verarbeitende Datum nur in solchen Ressourcen erwarten, die n_i auch definieren kann.

Durch einfache Schnittbildung der beiden relevanten Ressource-Mengen kann die Einhaltung dieser Bedingung erreicht werden und führt somit zu gültigen Datentransferwegen zwischen je zwei datenflussabhängigen Graphknoten.

Zur Wahrung der Knoten- und Kantenkonsistenz in einem Graphen sind diese Vorgehensweisen in einem Algorithmus (*Constraintpropagierungs-Algorithmus*) zusammengefasst und werden auf den GeLIR-Datenstrukturen zur Verfügung gestellt. Dabei werden Einschränkungen von Alternativen eines Graphknotens solange über die Datenflussabhängigkeitskanten an andere Knoten propagiert, bis keine Veränderungen mehr stattfinden. Die Laufzeit dieses Algorithmus ist im Wesentlichen von der Summe einzuschränkender Ressourcen *res* der vorhandenen Knoten sowie den gegebenen Datenflussabhängigkeitskanten E abhängig und kann durch $O(res * |E|)$ abgeschätzt werden.

Die Anwendung des Constraintpropagierungs-Algorithmus auf das in Abb. 2.9 gegebene Beispiel führt zu den in Abb. 2.12 verdeutlichten Ressource-Einschränkungen. Zunächst stellt der Algorithmus bei Überprüfung der Knotenkonsistenz des Multiplikationsknotens fest, dass die Auswahl der Ressource `B` zu keiner gültigen Ressource-Kombination führt und löscht diese daraufhin aus der Menge `Arg2`. Eine nachfolgend durchgeführte Überprüfung der Kantenkonsistenz zum Vorgängerknoten „G1“ bewirkt dann, dass die Ressource `B` ebenfalls aus der Menge `Def` des Knotens G1 gelöscht wird und somit nicht mehr ausgewählt werden kann.

2.3.5 Analysen & Optimierungen

Zur Durchführung von Optimierungen werden i.d.R. Analysen benötigt, die Informationen darüber bereitstellen, ob bestimmte Teilschritte die Semantik des Programms verändern oder nicht. Soll z.B. die Ausführungsreihenfolge zweier Anweisungen vertauscht werden, muss sichergestellt sein, dass zwischen diesen Anweisungen keine Datenabhängigkeiten existieren. In den GeLIR-Datenstrukturen werden aus diesem Grund für jeden Basisblock graphbasierte Darstellungen für Datenfluss-, Ausgabe- und Antiabhängigkeiten zur

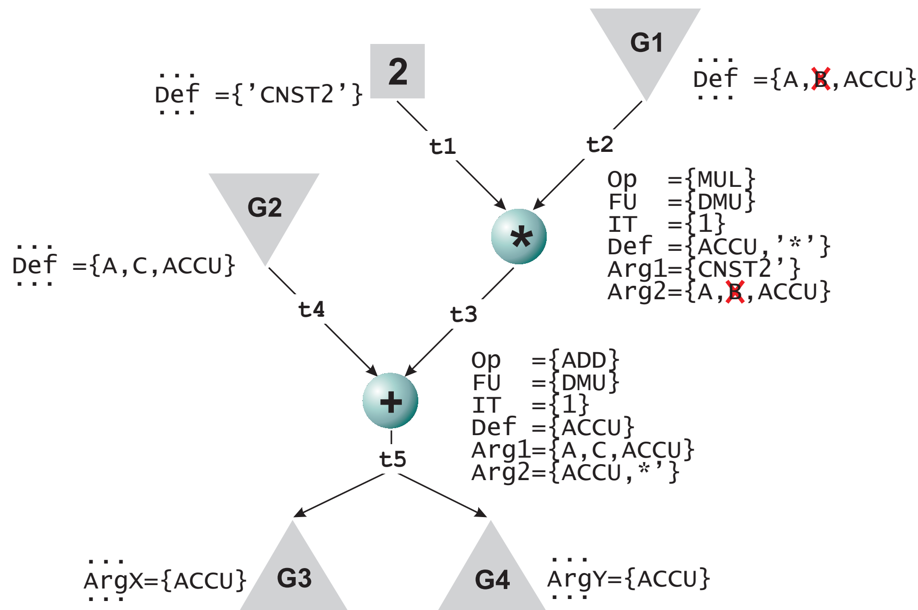


Abb. 2.12: Auswirkungen der Constraintpropagierung auf das Beispiel von Abb. 2.9

Verfügung gestellt. Zur Ermittlung des globalen Datenflusses zwischen den Basisblöcken einer Funktion werden zusätzlich in jeder Funktion ein globaler Datenflussgraph und ein Kontrollflussgraph verwaltet. Neben diesen üblicherweise von IRs zur Verfügung gestellten „einfachen“ Datenabhängigkeitsanalysen für skalare Variablen, ist auf den GeLIR-Datenstrukturen eine δ-Array-Datenflussanalyse zur Analyse von Abhängigkeiten zwischen Arrayzugriffen und eine Schleifenanalyse zur Ermittlung der in einer Schleife ausgeführten Basisblöcke vorhanden. Für nähere Informationen bezüglich der Umsetzung der δ-Array-Datenflussanalyse und der Schleifenanalyse soll an dieser Stelle auf die Diplomarbeit von Horst [Hor01b] verwiesen werden.

Im Rahmen der Diplomarbeit von Hornbach [Hor01a] wurden einige maschinenunabhängige Standardoptimierungen entwickelt, die mit Hilfe einer entsprechenden Parametrisierung ebenfalls maschinenspezifisch anwendbar sind. In diesem konkreten Fall wurden die RISC-Architekturen (am Beispiel des ARM7TDMI) als Zielplattform betrachtet. Im Rahmen dieser Arbeit wurden die Optimierungen *Constant-Folding*, *Constant-Propagation*, *Copy-Propagation*, *Dead-Code-Elimination*, *Redundant-Load-Elimination* und *Redundant-Store-Elimination* umgesetzt. Neben den zuvor erwähnten Standardoptimierungen sind zusätzlich generische Schleifenoptimierungen zur Ausnutzung von Zero-Overhead Hardware-Loops (ZOLs) und SIMD-Operationen (Vektorisierung) vorhanden [LWDL02], wobei auf letztere noch in Kapitel 4 näher eingegangen wird.

2.3.6 Graphische Visualisierung

Eine graphische Ausgabe beliebiger Zwischenzustände, z.B. vor und nach Durchführung einer neu entwickelten Optimierung, bietet eine gute Möglichkeit, diese Optimierung auf ihr erwartetes Verhalten hin zu überprüfen. Aus diesem Grund ist eine Schnittstelle zum Visualisierungsprogramm *aiSee* [aiS] vorhanden. Zur Veranschaulichung ist in Abb. 2.13 ein Kontrolldatenflussgraph (linker Teil) und ein reiner Kontrollflussgraph (rechter Teil) für das in Abb. 2.1 gegebene Quellprogramm abgebildet.

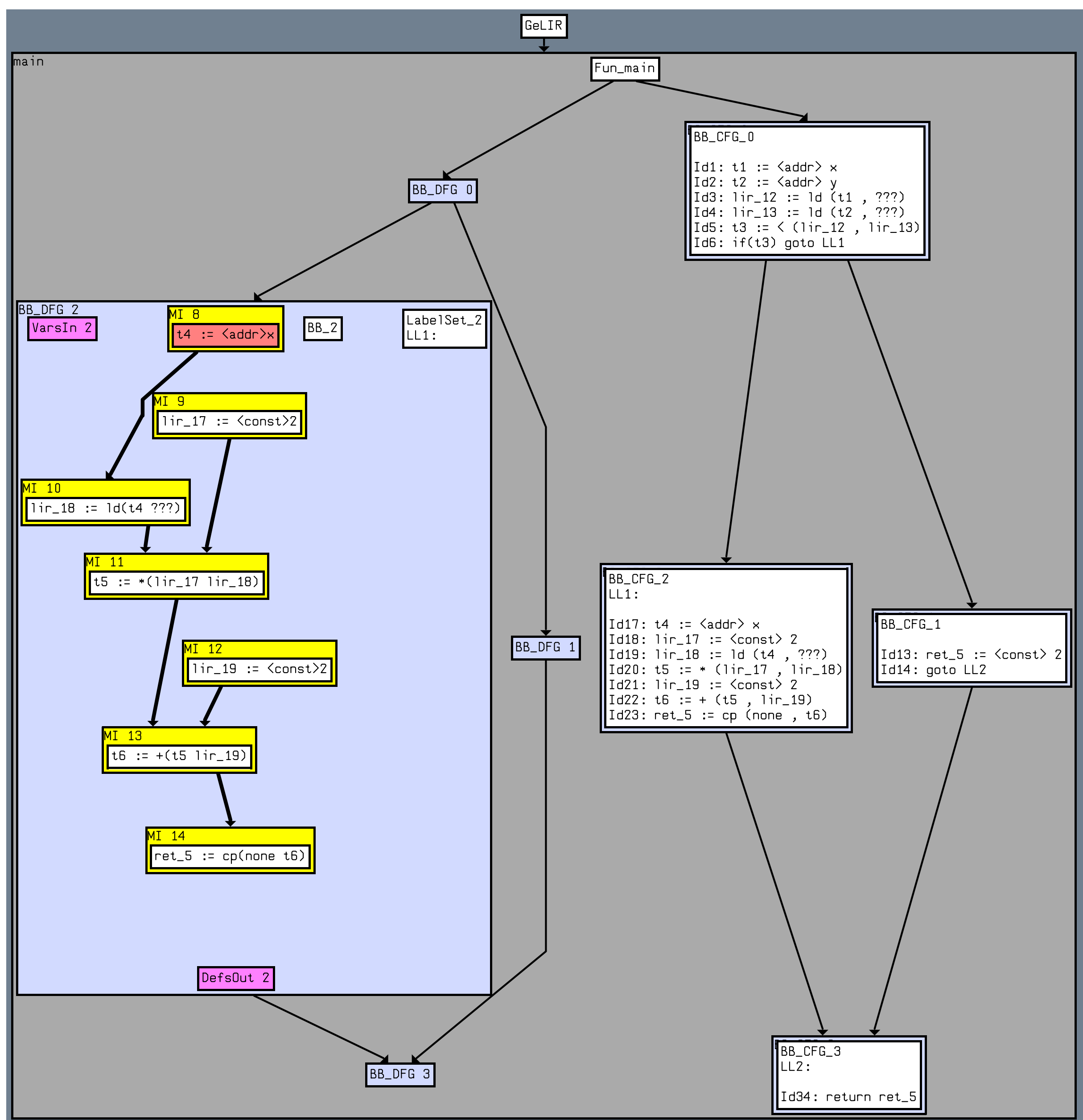


Abb. 2.13: Kontrolldatenflussgraph (linker Teil) und Kontrollflussgraph (rechter Teil) für das Programm aus Abb. 2.1

Nicht benötigte Darstellungen können bei Bedarf ein- bzw. ausgeblendet werden. In der Darstellung des Datenflussgraphen von Basisblock 2 ist zu erkennen, dass jede MI genau eine MO enthält. MOs, die zur Adressberechnung auf einer Adressgenerierungseinheit ausgeführt werden sollen und entsprechend klassifiziert sind, werden zur leichteren Identifikation farbig hinterlegt (s. MO in MI 8 in Abb. 2.13). Weitere Informationen, z.B. über Ausgabe- und Antiabhängigkeiten, Typen und Speicherpositionen von verwendeten Variablen und alternativen Ressourcen können ebenfalls selektiv angezeigt werden. Abb. 2.14 verdeutlicht z.B. die Darstellung alternativer Maschinenprogramme durch FMOs.

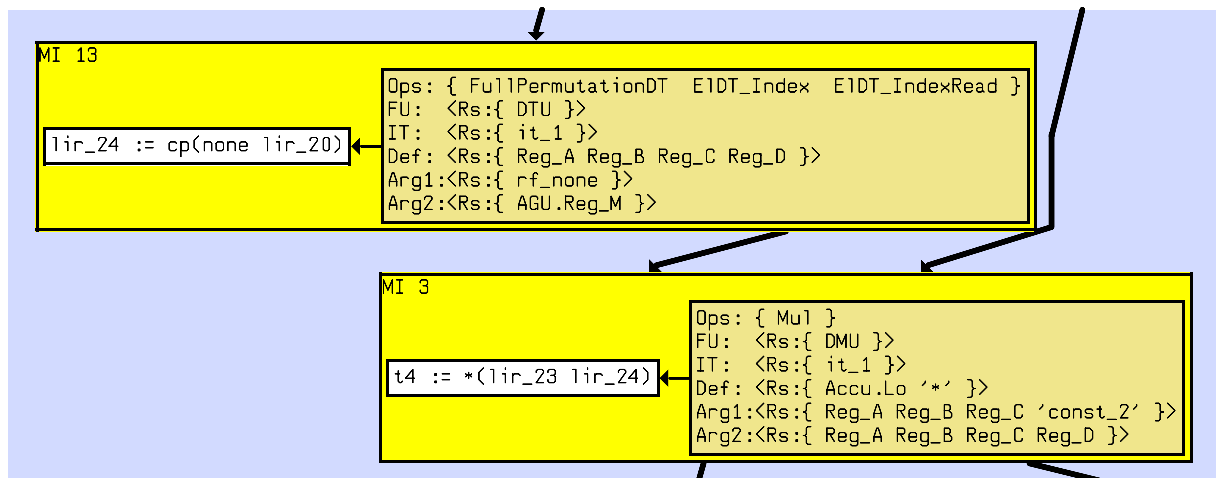


Abb. 2.14: Darstellung alternativer Maschinenprogramme

In Abb. 2.15 wird am Beispiel von Basisblock 2 die Zuordnung mehrerer MOs zu einer MI (hier: MI 104) zum Ausdruck von Parallelität veranschaulicht. Innerhalb einer solchen MI können die eingebetteten MOs wiederum aus beliebig vielen partiellen MOs zusammengesetzt sein.

2.3.7 XeLIR

Zum Speichern und Reproduzieren von Zwischenresultaten kann mit XeLIR eine XML-basierte GeLIR-Darstellung verwendet werden, auf der auch die Durchführung von Handoptimierungen und Architekturspezifikationen möglich ist. Durch die Verwendung von XML als zugrunde gelegtem Textformat besteht aufgrund des standardisierten XML-Sprachstandards die Möglichkeit, auf eine Reihe von Tools, insbesondere Parser und Editoren, zurückzugreifen. Diese erleichtern u.a. die Realisierung von Handoptimierungen und die Implementierung von Anwendungen auf dem generierten Textformat XeLIR, wie das Schreiben von Assemblercode in eine Datei sowie Peephole-Optimierungen. Ebenso besteht dadurch eine (eingeschränkte) Möglichkeit der Überprüfung von XML-Darstellungen

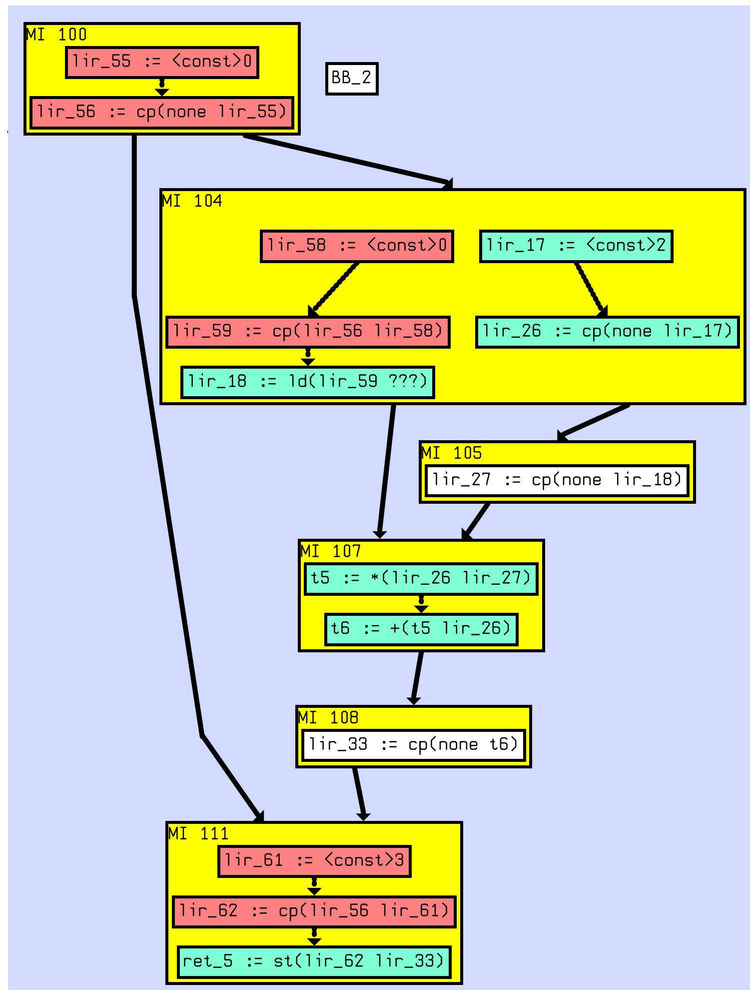


Abb. 2.15: Datenflussgraph mit parallel ausführbaren und komplexen MOs

auf Korrektheit. Nähere Einzelheiten dazu können der Diplomarbeit von Fiesel entnommen werden [Fie01].

2.3.8 Simulationsumgebung

Zur Validierung von implementierten Compilertechniken besteht die Möglichkeit der Simulation einer gegebenen GeLIR-Darstellung auf unterschiedlichen Abstraktionsebenen. Im Grundsatz wird hier zwischen einer maschinenunabhängigen Simulation und einer maschinenabhängigen Simulation unterschieden: Auf der abstrakten Ebene erfolgt eine Simulation der GeLIR-Darstellung, ohne die Berücksichtigung eventuell vorhandener Ressourcen-Bindungen. Die Symboltabelleneinträge der AMOs werden in dieser Darstellung als *virtuelle* Register und die abstrakten vordefinierten GeLIR-Operationen wie z.B. `ADD`, `MOVE` oder `SHL` als Operatoren verwendet. Zur Überprüfung der Korrektheit des Programms können nach Durchführung der Simulation die Inhalte bestimmter Variablen

überprüft werden. Mit der Durchführung einer hardwarenahen Simulation werden die an eine AMO gebundenen Ressourcen berücksichtigt. Dies betrifft insbesondere verwendete reale Register-Ressourcen, die im Vergleich zur abstrakten Simulation nun als Platzhalter für die zu verarbeitenden Werte dienen. Um benutzerdefinierte Operationen der Zielmaschine simulieren zu können, besteht die Möglichkeit deren Semantik bei der Spezifikation in Form von Berechnungsvorschriften anzugeben.

Die Durchführung der Simulation erfolgt nach dem Prinzip der *kompilierten* Simulation [HKN+01], bei dem die gegebene Programm- und Architekturdarstellung als C/C++-Code in Dateien geschrieben wird. Die Programmdarstellung besteht, wie in Abb. 2.16 verdeutlicht, im Wesentlichen aus einer Reihe von Zuweisungen und Funktionsaufrufen (z.B. `ImmedDT_Index_126`), deren konkrete semantische Bedeutung in einer separaten Datei festgelegt sind und deren Bedeutung bei der Spezifikation der Zielarchitektur angegeben werden muss. Diese Dateien werden mit weiteren Hilfsdateien unter Verwendung eines herkömmlichen C/C++-Compilers (z.B. GNU) kompiliert, dessen Ergebnis ein ausführbares Programm darstellt. Mit der Ausführung dieses Programms werden eine Reihe von Informationen generiert, die Auskunft über das Verhalten des gegebenen GeLIR-Programms bei Ausführung auf der spezifizierten Zielarchitektur geben. Im Einzelnen werden Informationen bezüglich der Anzahl ausgeführter Instruktionszyklen, die Anzahl von Speicherzugriffen und in unserem Fall auch den Energieverbrauch bereitgestellt.

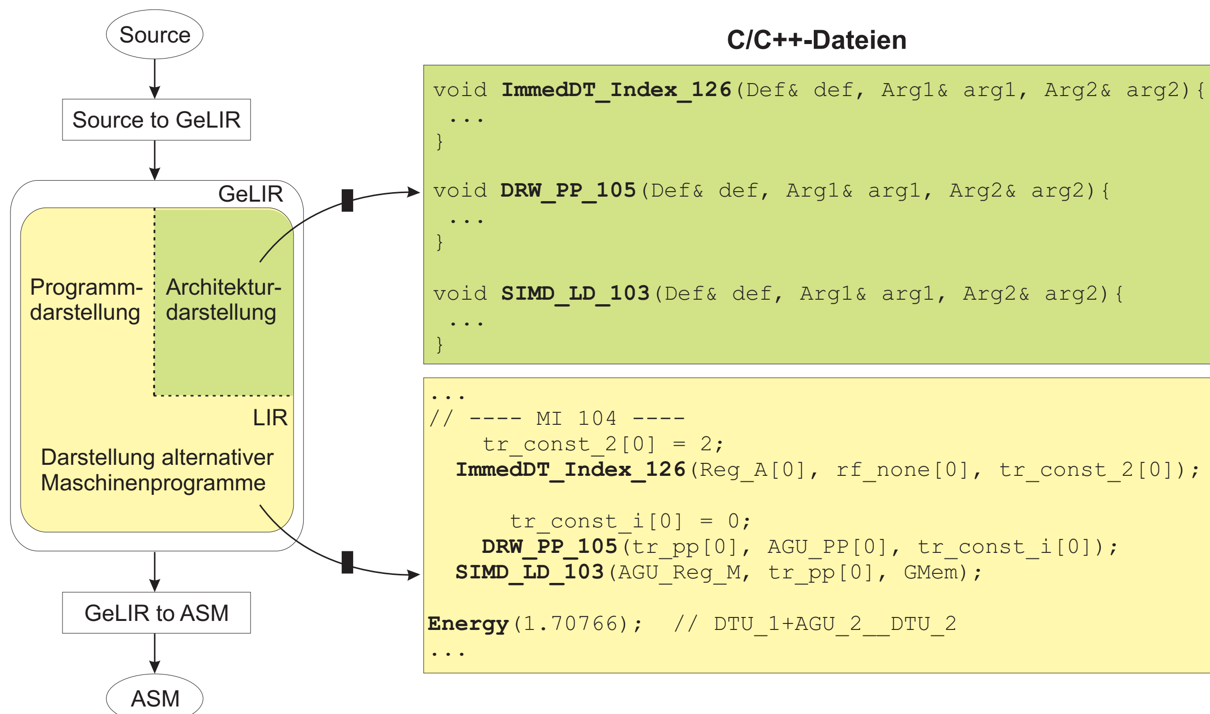


Abb. 2.16: Prinzip der kompilierten Simulation

Eine Validierung erfolgt, indem das ursprüngliche Quellprogramm ebenfalls direkt mit

dem Standardcompiler übersetzt und das Ergebnis der Ausführung (Werte bestimmter Variablen) der beiden Programme verglichen wird (s. Abb. 2.17).

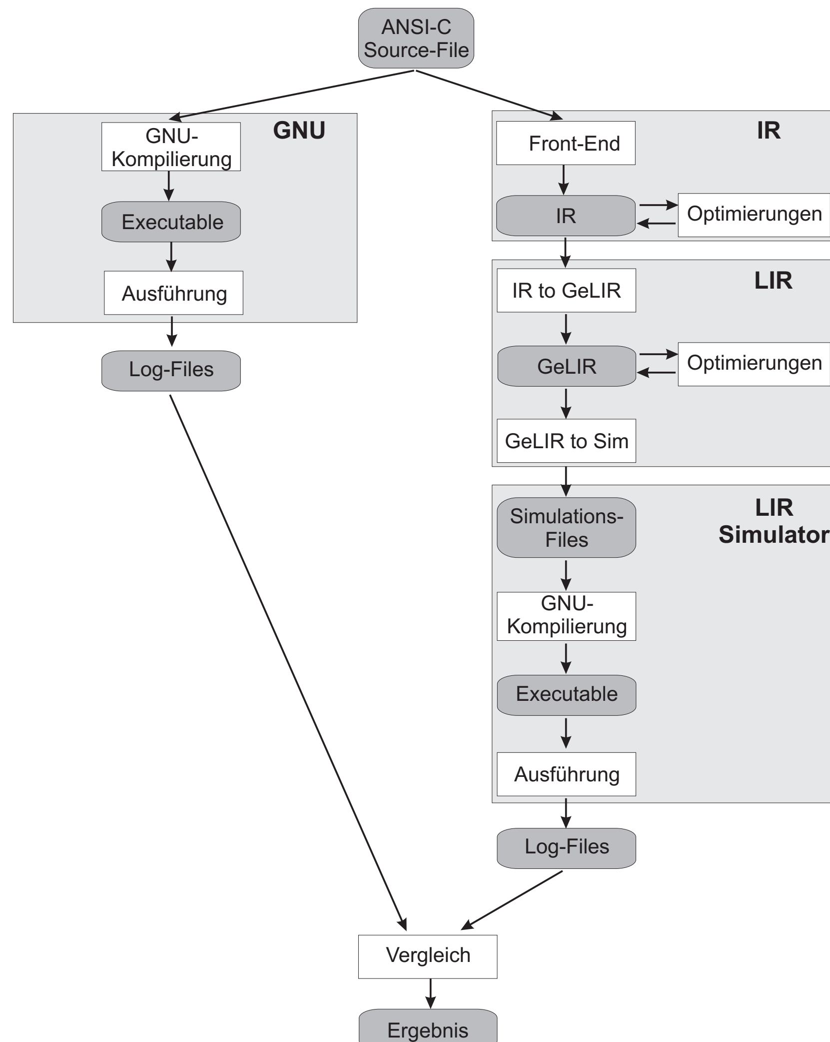


Abb. 2.17: Debug-Umgebung in GeLIR

Die Verwendung des GeLIR-Simulators weist im Vergleich zu speziellen Simulatoren konkreter Zielarchitekturen einige wesentliche Vorteile auf: So besteht insbesondere die Möglichkeit der Simulation und Validierung unmittelbar nach Abschluss einer Optimierung, ohne dass der gesamte nachfolgende Compilierungsprozess fortgesetzt werden muss. Des Weiteren wird durch die einfache Art und Weise, mit der Architekturänderungen spezifiziert werden können, die Durchführung einer HW/SW-Exploration erheblich vereinfacht. Ein weiterer Nachteil in der Verwendung herkömmlicher Simulatoren ist auch darin zu sehen, dass i.d.R. keine Energiekostenmodelle eingebunden sind, so dass keine Bewertung von Programmen hinsichtlich des Energieverbrauchs möglich ist. Eine nachträgliche Einbindung ist in der Regel aufgrund fehlender Quellprogramme ebenfalls nicht möglich.

Kapitel 3

Codegenerierung für digitale Signalprozessoren

Nach der Durchführung von maschinenunabhängigen Standardoptimierungen im Middle-End ist es die Aufgabe der Codegenerierung, das gegebene maschinenunabhängige Programm in ein ausführbares Programm der Zielmaschine zu überführen. Zur Erzielung von effizientem Assemblercode gilt es dabei, die architekturspezifischen Merkmale der Zielarchitektur möglichst effektiv auszunutzen. Bei der Entwicklung von Codegeneratoren für General-Purpose Prozessoren und digitale Signalprozessoren kommt es zu unterschiedlichen Gewichtungen der Zielsetzungen. Während für beide Arten von Zielarchitekturen die semantische Korrektheit des generierten Codes obligatorisch ist, wird beim Einsatz von Compilern für GPPs größerer Wert auf eine hohe Übersetzungsgeschwindigkeit gelegt, die i.d.R. auch erfüllt werden. Aufgrund der speziellen Einsatzgebiete von DSPs weisen deren Architekturen im Vergleich zu denen von GPPs besondere Merkmale auf, mit deren Hilfe schnellere und energieeffizientere Assemblerprogramme möglich sind. Leider führt dies auch zu Befehlssätzen, die entweder nicht oder nur sehr unzureichend von Codegeneratoren für GPPs gehandhabt werden können, so dass ein großer Bedarf an Codegeneratoren besteht, die speziell auf die besonderen Architektureigenschaften von DSPs abgestimmt sind. Da DSP-Programme i.d.R. einen geringeren Umfang haben und nicht so oft neu übersetzt werden müssen, wird bei DSP-Codegeneratoren üblicherweise eine längere Compilierungsdauer zugunsten effizienteren Codes akzeptiert [Leu99].

Nach einer Einführung im nächsten Abschnitt folgt eine Übersicht der bestehenden Arbeiten in diesem Bereich. Beginnend mit einer Übersicht des Codegenerators in Abschnitt 3.3 wird dann auf die Umsetzung des entwickelten Codegenerators eingegangen. Abschließend folgt eine Bewertung der entwickelten Techniken anhand einiger Testroutinen.

3.1 Einführung

In Ergänzung zu den in Abschnitt 1.4 bereits aufgeführten Nachteilen herkömmlicher Codegenerierungs-Verfahren wird im nachfolgenden Abschnitt genauer auf die Vorteile einer graphbasierten gegenüber einer baumbasierten Codeselektion (CS) eingegangen und die Bedeutung einer Phasenkopplung der Teilaufgaben Codeselektion, Instruktionsanordnung (IA) und Registerallokation (RA) aufgezeigt. Dem schließt sich eine Beschreibung des Einsatzes von Adressgenerierungseinheiten zur effektiven Umsetzung von Adressberechnungen an. Abschließend wird die Relevanz der Kombination unterschiedlicher Optimierungsziele erläutert.

3.1.1 Baumbasierte vs. graphbasierte Codeselektion

In der Regel führen Codegeneratoren eine baumbasierte Codeselektion durch, deren Funktionalität z.B. in Codegenerator-Generatoren wie BEG [ESL89], Twig [AGT89], iburg [FHP92] und Olive [Tji93] zur Verfügung gestellt wird. Eine Beschreibung der Architektur wird dabei in Form einer Baumgrammatik vorgenommen und ermöglicht damit insbesondere für GPPs ein hohes Maß an Retargierbarkeit. Die zugrunde gelegten Codeselektions-Verfahren (auch *Tree-Pattern-Matcher* genannt) führen in linearer Zeit (in Abhängigkeit zur Anzahl der Baumknoten) mit Hilfe der dynamischen Programmierung für jeden Ausdrucksbaum des Quellprogramms eine Überdeckung mit Prozessorinstruktionen durch [ASU86, WM95]. Da diese Verfahren nur auf Bäume anwendbar sind, muss zunächst eine Aufteilung der gegebenen Datenflussgraphen in Bäume vorgenommen werden. Für jeden dieser Bäume kann dann eine optimale Überdeckung mit Prozessorinstruktionen bestimmt werden, wobei das Ergebnis jedoch nur sequentielle Anweisungen enthält. Wie in Abb. 3.1 verdeutlicht, erfolgt die Aufteilung eines Datenflussgraphen in Bäume häufig anhand der CSEs (hier: Ergebnis der Multiplikation).

Anhand dieses einfachen Beispiels wird bereits das große Optimierungspotential in der Verwendung von graphbasierten gegenüber baumbasierten Codeselektions-Verfahren deutlich: So sind, unter Vernachlässigung eventuell zusätzlich erforderlicher Datentransfer- und Adressbefehle, bei der Verwendung einer baumbasierten Codeselektion zur Überdeckung der beiden generierten Bäume insgesamt neun MOs (s. schattierte Ellipsen) erforderlich. Von diesen stellen sechs MOs Speicherzugriffe (LD- und ST-Knoten) dar. Mit der Durchführung einer graphbasierten Codeselektion kann die Anzahl erforderlicher MOs auf fünf und die Anzahl der Speicherzugriffe auf drei beträchtlich reduziert werden. Dies lässt erwarten, dass mit der Durchführung einer graphbasierten im Vergleich zu einer baumbasierten Codeselektion neben einer Reduzierung der Ausführungszeit auch der Energieverbrauch drastisch reduziert werden kann. In [ASU77] wurde allerdings gezeigt, dass die Erzeugung von optimalem Code selbst für einen virtuellen Prozessor mit unend-

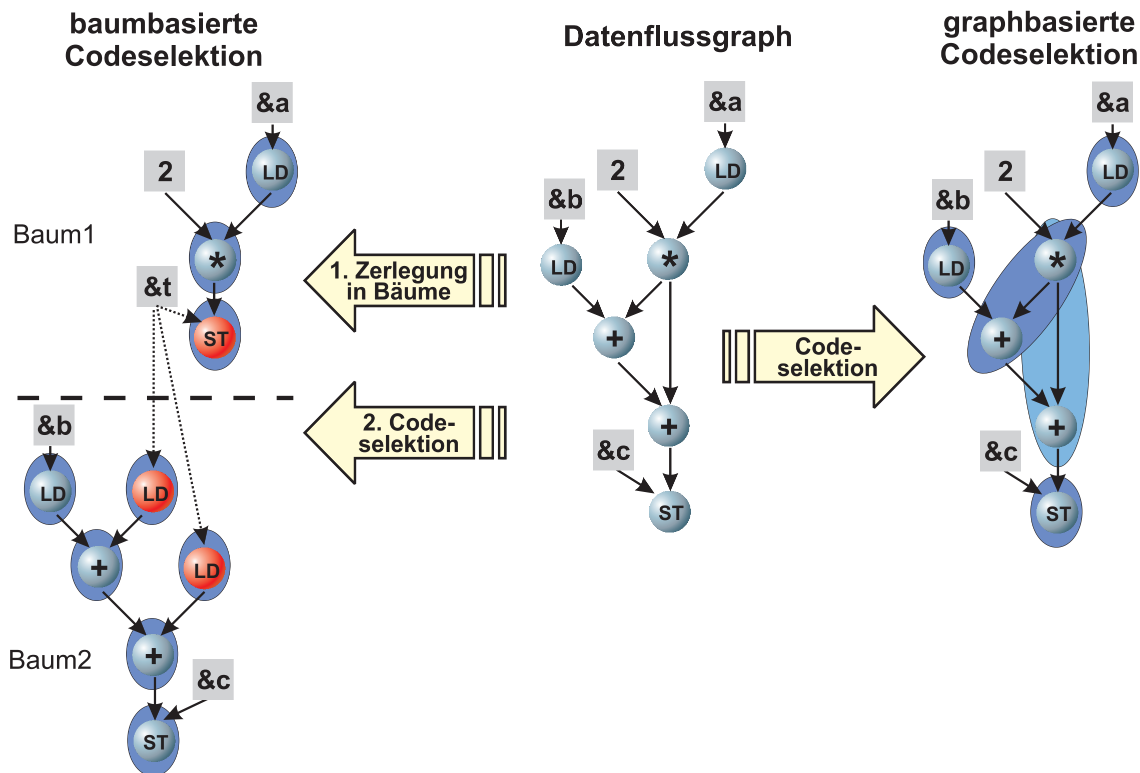


Abb. 3.1: Baumbasierte vs. graphbasierte Codeselektion

lich vielen Registern die Lösung eines NP-harten Problems bedeutet. Aus diesem Grund sind Optimierungsverfahren wünschenswert, die in polynomieller Laufzeit eine optimale Lösung möglichst gut annähern.

3.1.2 Bedeutung phasengekoppelter Optimierungsverfahren

Eine simultane Betrachtung der Teilphasen CS, IA und RA im Sinne einer Phasenkopplung ist insbesondere für irreguläre Prozessoren von besonderer Bedeutung. So weist der in dieser Arbeit betrachtete M3-DSP z.B. dedizierte Verbindungsstrukturen und heterogene Registerfiles auf. Diese Eigenschaften führen auf der Hardwareseite zwar zu einer Reduzierung der Chipfläche und der Leistungsaufnahme einzelner Prozessorinstruktionen, erschweren allerdings eine effiziente Codeerzeugung. Gründe hierfür sind u.a. darin zu sehen, dass selbst bei einer Verwendung von optimalen Verfahren für die Teilprobleme kein optimaler Code garantiert werden kann, wenn diese unabhängig voneinander ausgeführt werden. Verschärft wird dieser Umstand sogar noch dadurch, dass bereits die Ermittlung einer optimalen Lösung der Teilprobleme i.d.R. die Lösung eines NP-harten Optimierungsproblems darstellt. Die vor allem bei irregulären Prozessoren auftretenden Wechselwirkungen zwischen den Teilphasen der Codegenerierung werden im Folgenden näher beleuchtet, indem für jeweils zwei aufeinander folgende Teilphasen die jeweiligen

Wechselwirkungen aufgezeigt werden.

- Codeselektion und Instruktionsanordnung

  CS → IA
  Da häufig nur bestimmte Maschinenoperationen parallel zueinander ausgeführt werden können, besteht mit der Durchführung der Codeselektion die Gefahr, eine Auswahl von MOs vorzunehmen, die zwar zu einer minimalen Anzahl von Maschinenoperationen führt, sich aber schlecht parallelisieren lässt.

  IA → CS
  Werden die Operationen (AMOs) vor der Durchführung der Codeselektion zu MIs zusammengefasst, können sich u.U. Kombinationen von MOs ergeben, die nicht parallelisiert werden dürfen. Eine Auflösung dieser Konflikte kann in so einem Fall nur durch Einfügen weiterer MIs und die getrennte Ausführung der MOs aufgelöst werden.

- Instruktionsanordnung und Registerallokation

  IA → RA
  Eine Anordnung von MOs zu MIs mit minimaler Anzahl von MIs kann durch den in der Registerallokation nachträglich einzufügenden Spillcode wieder zunichte gemacht werden.

  RA → IA
  Werden die Variablen vor der Instruktionsanordnung an bestimmte Register gebunden, können sich aufgrund der entstandenen Ressource-Abhängigkeiten erhebliche Einschränkungen bei der Anordnung der MOs zu MIs ergeben. Des Weiteren ist es im Allgemeinen vor der Durchführung der Instruktionsanordnung sehr schwierig zu entscheiden, ob und welche Variablen in den Speicher gespillt werden müssen.

- Registerallokation und Codeselektion

  RA → CS
  Eine Bindung von Variablen an Register kann zu einer eingeschränkten Auswahl von MOs für den zu überdeckenden Graphknoten führen und sogar die Überdeckung mit einer gültigen Operations-Alternative unmöglich machen. Des Weiteren wird hierdurch die Bildung von komplexen MOs ausgeschlossen.

  CS → RA
  Mit der Auswahl der MOs wird die Verwendung von bestimmten Registern impliziert, so dass sich wiederum starke Auswirkungen auf den zu generierenden Spillcode ergeben.

In Ergänzung zu den bereits aufgeführten Wechselwirkungen treten bei DSPs im Allgemeinen Wechselwirkungen zur Adresscode-Generierung auf. Diese ergeben sich hauptsächlich dadurch, dass es nach der Durchführung von CS, IA und RA mehrere Instruktionssequenzen mit einer minimalen Anzahl von MIs geben kann, die jeweils unterschiedliche Speicherzugriffs-Sequenzen enthalten. Da diese Speicherzugriffs-Sequenzen in Verbindung mit einem zu bestimmenden Speicherlayout wiederum einen großen Einfluss auf die Verwendung vorhandener Adressierungsbefehle aufweisen, ist die letztendlich resultierende Codequalität auch vom Zusammenspiel dieser beiden Phasen abhängig.

Bei genauerer Betrachtung der Auswirkungen des Gruppenspeichers der M3-Prozessoren wird klar, dass zusätzlich noch ein *Meta-Phasenkopplungsproblem* existiert. Aufgrund der Tatsache, dass mit jedem Speicherzugriff eine Gruppe von Daten betroffen ist, besteht ein enger Zusammenhang zwischen der vorhandenen Anordnung von Variablen zu Gruppen und der Anzahl auszuführender Speicherzugriffe. Wie in Abschnitt 4.7 noch näher erläutert wird, kann die Anzahl der erforderlichen Speicherzugriffe dadurch verringert werden, indem Variablen, auf die häufig zeitnah zugegriffen wird, derselben Gruppe zugewiesen werden.

3.1.3 Bedeutung von Adressgenerierungseinheiten

In DSP-Anwendungen erfolgt die Datenhaltung häufig in Arrays, deren Elemente mit nahezu beliebig komplexen Zugriffsfunktionen adressiert werden können. Dies lässt bereits vermuten, dass eine entsprechende Hardwareunterstützung zur Berechnung von Adressen die Programmausführungszeit erheblich reduzieren kann. Aus diesem Grund enthalten DSPs üblicherweise separate Adressgenerierungseinheiten (AGUs), mit denen Speicherzugriffe und Adressberechnungen parallel zu Operationen des Datenpfades durchgeführt werden können. Im Allgemeinen ist eine gewisse Anzahl von Adresspointer-Registern AR vorhanden, mit deren Hilfe eine Adressierung des Speichers erfolgt. Damit die Adresse des nächsten auszuführenden Speicherzugriffs nicht in einem separaten Taktzyklus berechnet werden muss, kann der im entsprechenden Adresspointer-Register enthaltene Wert nach dem Speicherzugriff um einen bestimmten Offset $\mathit{off} \in \mathbb{Z}$ modifiziert werden. Je nach Quelle des verwendeten Offset wird zwischen den beiden folgenden Adressierungen unterschieden:

- *Auto-Inkrement*
  Der Offset stellt eine Konstante dar[1], mit der jedoch nur relativ kleine Speicher-Differenzen überbrückt werden können. Beim M3-DSP muss dieser Offset z.B. aus dem Bereich $[-128, \ldots, 127]$ sein, wodurch bei einer gegebenen Gruppengröße von

[1] Im Falle einer negativen Konstante wird in diesem Zusammenhang auch von *Auto-Dekrement*-Befehlen gesprochen.

16 Daten insgesamt Adressen von Elementen aus maximal 16 Gruppen (8 vorherige, 7 nachfolgende und der aktuellen) berechnet werden können.

- *Auto-Modify*
  Ist die Neuberechnung einer Adresse mittels einer Konstanten aufgrund eines zu großen erforderlichen Offsets nicht möglich, kann stattdessen der benötigte Offset in ein Modify-Register *MR* geladen werden, aus dem dieser dann ausgelesen wird. Dies hat den Vorteil, dass ausreichend große Adressbereiche übersprungen werden können, erfordert bei Bedarf allerdings auch zusätzliche Zyklen zum Laden des Offsets in das entsprechende Modify-Register.

Eine weitere Adressierungsart beim M3-DSP besteht in der Adressierung des Speichers relativ zu einer Seitenadresse (*Page-Pointer-Adressierung*). Hierbei wird ausgehend von der im Page-Pointer-Register *PP* vorhandenen Seitenadresse mit Hilfe eines Offsets $off \in [0, \ldots, 1023]$ innerhalb dieser Seite adressiert, ohne die im *PP*-Register enthaltene Adresse zu modifizieren.

Da eine Ausnutzung der speziellen AGU-Anweisungen nur bei Kenntnis des Speicherlayouts und einer gegebenen Speicherzugriffs-Sequenz möglich ist, wird die Adresscode-Generierung nach der Durchführung der Teilaufgaben CS, IA und RA in einem separaten Schritt durchgeführt. Dabei gilt es im Allgemeinen die Anzahl zusätzlich erforderlicher MIs so gering wie möglich zu halten. Steht zur Adressierung des Speichers nur ein Adressregister *AR* zur Verfügung, wird in diesem Zusammenhang auch vom SOA-Problem (SOA = *Simple-Offset-Assignment*) gesprochen. Ist mehr als ein Adressregister vorhanden, handelt es sich um das GOA-Problem (GOA = *General-Offset-Assignment*).

3.1.4 Kombination von Optimierungszielen

Als Schwerpunkt dieser Arbeit wird als Optimierungsziel neben einer üblicherweise betrachteten Reduzierung der Ausführungszeit auch eine Reduzierung des Energieverbrauchs von DSP-Programmen angestrebt. Da die schnellsten Programme nicht immer gleichzeitig auch die energieeffizientesten Programme darstellen müssen, sind Optimierungsverfahren erforderlich, die in der Lage sind mehrere Zielsetzungen zu berücksichtigen. In Gegenwart von Realzeitanforderungen sind z.B. Verfahren wünschenswert, die unter den Programmen mit der geringsten Ausführungszeit das energieeffizienteste bestimmen. Spielen Realzeitanforderungen eine weniger große Rolle, dann könnte auch das Programm gesucht sein, das von den Programmen mit dem geringsten Energieverbrauch, die schnellste Ausführung garantiert. Ebenso ist es vorstellbar, dass beliebige Zwischenstufen gesucht werden. Zur Realisierung eines Codegenerators, der eine derartige Kombination von Zielsetzungen zulässt, ist eine große Flexibilität des zugrunde gelegten Optimierungsverfah-

rens essentiell, um bei veränderten Zielsetzungen eine Neuimplementierung von Techniken zu vermeiden.

3.2 Bestehende Verfahren

In diesem Abschnitt wird ein Überblick über bestehende Arbeiten im Bereich der Codegenerierung gegeben. Dazu wird im nachfolgenden Abschnitt zunächst auf Arbeiten eingegangen, die sich mit den Teilaufgaben der Codeselektion, Instruktionsanordnung und Registerallokation befassen. In den beiden darauf folgenden Abschnitten folgt dann eine kurze Darstellung von Verfahren zur Adresscode-Generierung und von Techniken, die speziell das Ziel einer Energiereduzierung aufweisen.

3.2.1 Codegenerierung

Eine Vielzahl der in Compilersystemen integrierten Codegeneratoren verwendet Codeselektions-Verfahren auf Basis von Tree-Pattern-Matchern. Da der Einsatz dieser Verfahren für Prozessoren mit irregulären Befehlssätzen eine Reihe von Nachteilen nachsichzieht, wurden Erweiterungen dieser Verfahren entwickelt. So werden z.B. in [AM95, Ert99] Techniken vorgestellt, die auch für Graphen eine optimale Codeselektion in linearer Berechnungszeit erlauben. Während das in [Ert99] beschriebene Verfahren lediglich auf GPPs anwendbar ist, kann in [AM95] zumindest eine stark eingeschränkte Klasse von Zielarchitekturen mit heterogenen Registersätzen gehandhabt werden. Basierend auf dem in [AM95] beschriebenen optimalen Codeselektions-Verfahren für Graphen wird in [AML96] eine Heuristik vorgestellt, die eine Unterteilung des Graphen in Bäume an CSEs vornimmt, deren Wert aufgrund von architekturspezifischen Eigenschaften ohnehin in den Speicher geschrieben werden muss.

Da Tree-Pattern-Matcher für Zielarchitekturen mit heterogenen Registersätzen CSEs gewöhnlich im Speicher ablegen und von dort bei jeder Verwendung laden müssen, wurde in [Leu00c] eine Erweiterung dieser Verfahren um einen auf *Simulated-Annealing* basierenden Algorithmus vorgeschlagen, mit dem es möglich ist, zumindest einige der CSEs in Registern zu halten. Eine Überdeckung von abstrakten Maschinenoperationen unterschiedlicher Bäume mit Maschinenoperationen ist allerdings nach wie vor nicht möglich. Ebenso wird lediglich nur eine eingeschränkte Kopplung der Phasen CS, IA und RA vorgenommen.

Ein großes Problem bei der Durchführung der Codegenerierung besteht in der Realisierung einer für optimale Ergebnisse erforderlichen simultanen Betrachtung der Teilaufgaben CS, IA und RA. In [KL98] wurde ein Codegenerierungs-Verfahren auf Basis der *ganzzahlig linearen Programmierung* (GLP) vorgestellt, mit dem die Phasen der Instruk-

tionsanordnung und der Registerallokation (ohne die Berücksichtigung von Spillcode) simultan gelöst werden. Dabei wird das zugrunde gelegte Optimierungsproblem als eine Menge von (Un-)Gleichungen aufgefasst, mit deren Hilfe die vorhandenen Randbedingungen spezifiziert werden. Dieses Gleichungssystem wird dann unter Berücksichtigung einer ebenfalls spezifizierten Kostenfunktion mit Hilfe eines herkömmlichen Lösungsverfahrens (*GLP-Solver*) gelöst. Der Vorteil dieses Verfahrens ist, dass auf elegante Weise eine simultane Betrachtung der Codegenerierungs-Phasen erzielt werden kann. Weitere Codegenerierungs-Verfahren auf Basis der ganzzahlig linearen Optimierung werden von Wilson [WGHB94] und Gebotys [Geb97] vorgestellt. Allerdings stellt die Lösung der Gleichungssysteme wiederum die Lösung eines NP-harten Optimierungsproblems dar. Um die Laufzeiten des GLP-Solvers in vertretbaren Grenzen zu halten, können entweder nur kleine Programmfragmente übersetzt werden, oder es müssen bereits bei der Problembeschreibung Einschränkungen bezüglich der Optimalität der Lösungen in Kauf genommen werden.

Der von Bashford [BL99, Bas01] vorgestellte Codegenerator verwendet das Prinzip der *Constraint-Programmierung*. Hier werden Randbedingungen z.B. hinsichtlich der Verwendung von Ressourcen oder der Ausführungsreihenfolge von Maschinenoperationen durch die Formulierung von Constraints sichergestellt. Nach der Darstellung der zu übersetzenden Anwendung als eine Menge alternativer Maschinenprogramme wird zunächst eine graphbasierte Codeselektion und eine Instruktionsanordnung durchgeführt, die jeweils für sich genommen optimal gelöst werden. Eine Phasenkopplung wird dadurch erzielt, indem in jeder Phase nur die erforderlichen Ressourcen gebunden werden und dadurch die Ausgabe wiederum eine Menge von alternativen Maschinenprogrammen darstellt. Jedes der erhaltenen Maschinenprogramme stellt dann bezüglich des vorherigen Schrittes *eine* optimale Lösung dar und räumt damit den nachfolgenden Phasen weitreichende Flexibilität ein. Da die Laufzeiten zur Bestimmung der optimalen Lösung für größere Graphen wiederum sehr hoch sein können, wird auch eine Heuristik vorgestellt, mit der eine Aufteilung des Graphen in kleinere Probleme vorgenommen wird, ohne allzu große Einbußen hinsichtlich der Codequalität hinnehmen zu müssen.

Römer beschreibt in [RF98a, RF98b] ein Phasenkopplungsverfahren für irreguläre Architekturen, das ebenfalls in der Lage ist, eine Codeselektion auf Graphen durchzuführen. Dabei wird das Codegenerierungs-Problem mit Hilfe eines Zustandsdiagramms (*Trellis*) dargestellt. Das Problem liegt dabei in der Suche eines optimalen Pfades ausgehend von einem Startzustand zu einem Endzustand. Der ermittelte Pfad stellt letztendlich die bezüglich einer gegebenen Kostenmetrik (hier: die Anzahl erforderlicher Maschinenbefehle) optimale Sequenz von Maschinenbefehlen dar. Die Suche nach dem besten Pfad kann dabei mit dem aus der Nachrichtentechnik bekannten Virterbi-Algorithmus durchgeführt werden, dessen Arbeitweise auf dem Prinzip der dynamischen Programmierung beruht. Dazu wird in jedem Zustand eine Menge von Pfaden verwaltet, die ausgehend

vom Anfangszustand diesen Zustand mit minimalen Kosten erreichen können. Alle Pfade, die diesen Zustand mit höheren Kosten erreichen, brauchen nachfolgend nicht weiter betrachtet werden. Allerdings besteht das Problem dieses Verfahrens darin, dass die Anzahl der Zustände mit der Größe des Graphen exponentiell ansteigt. Aus dem Grund wird mit dem *M-Algorithmus* [LA86] die Anzahl der gleichzeitig zu verfolgenden Pfade auf eine feste Anzahl M begrenzt. Wird z.B. M auf eins gesetzt, entspricht dies dem bekannten List-Scheduling-Algorithmus, der in diesem Ansatz in Verbindung mit Heuristiken zur Auswahl von Befehlen verwendet wird. Je höher der Wert für M gewählt wird, desto bessere Ergebnisse (bei steigenden Laufzeiten) sind zu erwarten. Leider sind keine Ergebnisse dieses Verfahrens veröffentlicht, so dass keine Rückschlüsse auf die Laufzeit und die erzielte Codequalität gezogen werden können.

In [LDKT95] wird ein Verfahren beschrieben, das eine optimale graphbasierte Instruktionsauswahl für Architekturen mit irregulärem Datenpfad durch Formulierung als *Binate-Covering*-Problem realisiert, ohne jedoch Auswirkungen auf die nachfolgenden Phasen der Instruktionsanordnung und der Registerallokation mit einzubeziehen. Da die Berechnung einer exakten Lösung nur für kleine Probleme praktikabel ist, werden zur Verringerung der Problemkomplexität zusätzlich Heuristiken vorgeschlagen. Konkrete Ergebnisse werden leider nicht vorgestellt.

Mutation Scheduling [NN94] stellt ein Verfahren dar, mit dem die Phasen der Codeselektion und der Registerallokation in der Instruktionsanordnungs-Phase integriert sind. Dabei werden z.B. bei der Codeselektion mit jedem im Programm definierten Wert Mengen gleichwertiger Ausdrücke assoziiert, von denen in einem Programm immer genau einer verwendet wird. Muss ein Ausdruck durch einen anderen ersetzt werden, erfolgt eine heuristische Auswahl des neuen Ausdrucks. In diesem Zusammenhang wird auch von *Mutation* gesprochen. Ergebnisse werden für drei VLIW-Architekturen präsentiert, von denen lediglich eine Architektur eine geringfügige Irregularität aufweist.

In [HD98] wird mit *AVIV* ein phasengekoppeltes Codegenerierungs-Verfahren für VLIW-Prozessoren vorgeschlagen. Dazu wird zunächst eine Konvertierung der Anwendung in einen *Split-Node*-Graphen vorgenommen, der alle möglichen Wege der Implementierung der Anwendung auf diesem Prozessor beinhaltet. Allerdings kann sich hierbei die Anzahl der Graphknoten bereits für kleine Datenflussgraphen sehr stark erhöhen (bis zum Faktor 6,6 in [HD98, Han99]). Danach werden mittels eines heuristischen Branch-and-Bound-Verfahrens die Phasen CS und IA in einem Schritt durchgeführt, wobei durch das Einfügen von Spillcode bereits die Auswirkungen der nachträglich durchgeführten Registerallokation mitberücksichtigt werden. Es kann allerdings aufgrund der pessimistischen Abschätzung vorkommen, dass mehr Spills eingefügt werden, als eigentlich erforderlich sind. Ergebnisse werden für fünf einfache Routinen, bestehend aus einem Basisblock für unterschiedliche VLIW-Architekturen, vorgestellt. Die betrachteten Architekturen stellen dabei einfache virtuelle VLIW-Architekturen dar, die sich in der Anzahl der parallelen

Funktionseinheiten (zwei bis vier), den auf diesen ausführbaren Operationen und den verfügbaren Registern (zwei oder vier) unterscheiden. Ein Vergleich der Codequalität des Compilers mit handgeneriertem Code ergab für diese Routinen jeweils nur geringe Abweichungen.

Genetische Algorithmen (GAs) stellen Optimierungsverfahren dar, die in der Lage sind, selbst in großen Suchräumen häufig optimale oder nahezu optimale Lösungen zu finden. Aus diesem Grund wurden in der Vergangenheit Verfahren zur Lösung des Scheduling-Problems auf Basis von genetischen Algorithmen entwickelt. Der von Beaty [Bea91] vorgestellte genetische Algorithmus verwendet dabei ein List-Scheduling-Verfahren zur Instruktionsanordnung für den RISC-Prozessor RS/6000 von IBM [Gro90]. Ergebnisse für fünf Benchmarks zeigen, dass bessere Ergebnisse erzielt werden können als bei Verwendung eines reinen List-Scheduling-Verfahrens.

Ein weiteres reines Scheduling-Verfahren auf Basis eines genetischen Algorithmus wird von Zeitlhofer in [ZW99] vorgestellt, mit dem auch die Zuweisung von Operationen für Architekturen mit eingeschränkter Parallelität möglich ist. Experimentelle Ergebnisse werden allerdings leider nur für eine Lattice-Routine bei Betrachtung einer virtuellen Architektur mit zwei orthogonal zueinander verwendbaren Funktionseinheiten vorgestellt, so dass eine weitergehende Bewertung der Qualität dieses Verfahren nicht möglich ist.

Ein Scheduling-Verfahren auf einer höheren Ebene auf Basis von genetischen Algorithmen wird von Fröhlich [Frö01] vorgeschlagen. Dabei wird mit dem Ziel einer optimierten Anordnung von Teilblöcken eines Programms ein GA in Kombination mit anderen Optimierungsverfahren ausgeführt. Die Codegenerierung der jeweiligen Teilblöcke wird dabei mit Hilfe eines baumbasierten Codeselektions-Verfahren und einem List-Scheduling-Verfahren zur Registerallokation und Kompaktierung durchgeführt. Da in den betrachteten Teilblöcken keine Auslagerung von Variablen in den Speicher möglich ist, wird davon ausgegangen, dass alle Variablen in Registern gehalten werden können.

3.2.2 Adresscode-Generierung

Das Problem der Adresscode-Generierung besteht darin, die in einem Programm verwendeten Variablen derart im Speicher anzuordnen, dass möglichst effektiv Gebrauch von speziellen AGU-Befehlen, wie z.B. Auto-Inkrement und Auto-Modify gemacht werden kann. Zur Lösung des SOA/GOA-Problems wurden darum in den letzten Jahren eine Reihe von Verfahren vorgestellt (s. auch [Leu00a] für einen Überblick), von denen nachfolgend einige kurz beschrieben werden.

In [Bar92] wurde von Bartley der erste Algorithmus zur Lösung des SOA-Problems vorgestellt, indem das SOA-Problem als Graphproblem dargestellt wird. Die Knoten des Graphen stellen dabei die Variablen $V = \{v_1, \ldots, v_n\}$ dar, deren Zugriffsreihenfolge während

der Programmausführung in Form einer Variablenzugriffs-Sequenz $S = (s_1, \ldots, s_m)$, mit $s_i \in V$ gegeben ist. Zwischen je zwei Knoten werden gewichtete Kanten eingefügt, deren Gewicht sich aus der Anzahl der aufeinander folgenden Zugriffe der Variablen ergibt. Dies ist für zwei Variablen genau dann gegeben, wenn diese an aufeinander folgenden Positionen in der Variablenzugriffs-Sequenz stehen. Das Ziel der Optimierung besteht nun darin, für die gegebenen Variablen eine Zuordnung zu Adressen zu bestimmen, so dass möglichst häufig Auto-Inkrement-Befehle ausgenutzt werden können. Die Suche nach der optimalen Lösung für das Adresszuweisungs-Problem besteht letztlich in der Suche eines maximal gewichteten Hamilton-Pfades im Graphen. Alle nicht ausgenutzten Kanten des Graphen stellen dabei nicht ausgenutzte Möglichkeiten der Anwendung von Auto-Inkrement-Befehlen dar, die es dementsprechend zu minimieren gilt. Die Verwendung der von Bartley vorgestellten Heuristik führt für diese Problemklasse bereits zu guten Ergebnissen.

Liao zeigte in [LDK+95], dass das SOA-Problem bereits unter Ausnutzung *eines* Adressregisters und einem Offset von eins ein NP-hartes Optimierungsproblem darstellt. Neben einem modifizierten Kruskal-Algorithmus zur Konstruktion von Hamilton-Pfaden stellte er in [LDK+95] auch ein Verfahren zur Lösung des GOA-Problems vor, mit dem k Adressregister gehandhabt werden können. Da eine Partitionierung der Variablen zu Adressregistern vorgenommen wurde, konnte als Kern des Lösungsverfahrens wiederum ein Verfahren zur Lösung des SOA-Problems verwendet werden.

Leupers und Marwedel [LM96] modifizierten das von Liao vorgestellte Verfahren durch die Verwendung einer besseren Heuristik zur Variablen-Partitionierung und durch den Einsatz einer *Tie-Break*-Heuristik. Erweiterungen dieser Verfahren zur Handhabung größerer Auto-Inkrement-Bereiche und den Einsatz von Modify-Registern erfolgten dann z.B. in [WG97, LD98]. Eine Erweiterung über Basisblock-Grenzen hinaus wurde z.B. von Leupers [Leu98] und Araujo [AOC02] vorgeschlagen.

Neben der Suche nach einem geeigneten Speicherlayout und einem optimierten Einsatz der AGU-Ressourcen für eine gegebene Speicherzugriffs-Sequenz müssen bei der Adresscode-Generierung für den M3-DSP weitere Punkte beachtet werden. So wird bei keinem der Verfahren die Problematik der Adresszuweisung für einen Gruppenspeicher, wie er bei den M3-Prozessoren vorliegt, betrachtet.

3.2.3 Energieoptimierungen

In den letzten Jahren stellte, neben den klassischen Optimierungszielen wie Ausführungszeit und Codegröße, in zunehmendem Maße eine Verringerung des Energieverbrauchs einen wichtigen Punkt bei der Entwicklung von eingebetteten Systemen dar. Dabei wurden zur Reduzierung des Energieverbrauchs bislang die meisten Bemühungen im Bereich des Hardwareentwurfs vorgenommmen. Allerdings besteht aufgrund des zunehmenden

Einsatzes von Prozessoren ein stetig steigender Bedarf an einer Reduzierung des Energieverbrauchs auf der Softwareebene. Da der Schwerpunkt dieser Arbeit auf der Entwicklung von Compilertechniken liegt, werden nachfolgend einige compilergesteuerte Verfahren zur Reduzierung des Energieverbrauchs von ausgeführten Programmen vorgestellt. Ein Überblick über Optimierungen auf Hardware- bzw. Softwareebene befindet sich z.B. in [TMW94a, RP96, MPS98, MB02].

Eine Möglichkeit zur Reduzierung des Energieverbrauchs besteht durch eine Neuanordnung der Instruktionen zur Reduzierung der Switchingaktivität aufeinander folgender Befehle (*Cold-Scheduling*). In [STD94] wird z.B. von einer Reduzierung der Switchingaktivität im Bereich von 20% bis 30% für einen RISC-Prozessor berichtet, bei einer geringfügigen Verringerung der Ausführungszeit von 2% bis 4%. Leider werden keine Angaben über die letztendlichen Auswirkungen auf den Energieverbrauch gemacht.

In [LLHT00] wird ein Scheduling-Verfahren für VLIW-Architekturen vorgeschlagen, mit dem Ziel der Minimierung der Switchingaktivitäten des Instruktions-Busses. Als Kostenmaß wird die *Hamming-Distanz* verwendet, die die Anzahl unterschiedlicher Bits zwischen zwei binären Strings (oder Instruktionen) angibt. Dazu werden zunächst mittels List-Scheduling die vorhandenen Operationen zu VLIW-Instruktionen (MIs) angeordnet, mit dem Ziel der Reduzierung der Ausführungszeit. Danach werden zwei Scheduling-Schritte jeweils mit dem Ziel der Reduzierung der Hamming-Distanz durchgeführt. Im *horizontalen* Scheduling wird dabei zunächst eine Neuanordnung der vorhandenen MIs vorgenommen. Die resultierenden MIs dienen dann als Eingabe für das *vertikale* Scheduling, bei dem die in den MIs enthaltenen MOs anderen MIs zugeordnet werden dürfen, ohne jedoch die Anzahl der MIs zu erhöhen. Ergebnisse werden für eine Reihe von Benchmarks für zwei virtuelle VLIW-Architektur vorgestellt. Ein Vergleich der optimierten Benchmarks mit dem Ergebnis des List-Schedulers, das als Ausgangsbasis der Optimierungen diente, weisen eine Reduzierung der Switchingaktivität von durchschnittlich 13% bzw. 20% auf. Leider wird in dieser Arbeit ebenfalls lediglich die Switchingaktivität eines Busses als Bewertungsmaßstab genommen, so dass wiederum keine Aussagen über reale Energieeinsparungen möglich sind.

Neben einer Neuanordnung von Instruktionen zur Reduzierung des Energieverbrauchs besteht z.B. die Möglichkeit bei der Durchführung der Codeselektion unter Befehlen mit gleicher Funktionalität, energieeffizientere Befehle auszuwählen. In [TMW94a] wird vorgeschlagen, dazu eine modifizierte Kostenfunktion eines Tree-Pattern-Matchern zu verwenden, indem statt der üblicherweise verwendeten Kosten hinsichtlich der Ausführungszeit die Energiekosten verwendet werden.

Des Weiteren bietet sich eine Vermeidung von energieintensiven Speicherzugriffen an, indem Variablen möglichst in Registern gehalten werden. Dazu können für Architekturen mit homogenen Registerfiles z.B. globale Registerallokations-Techniken wie das *Gra-*

phfärben von Chaitin [CAC+81] eingesetzt werden, mit denen der Spillcode und damit die Anzahl der Speicherzugriffe minimiert wird. Leider sind diese Verfahren bei irregulären Prozessoren, wie dem M3-DSP nicht ohne weiteres anwendbar. In jedem Fall sollten hier graphbasierte Codeselektions-Verfahren verwendet werden, von denen einige in Abschnitt 3.2.1 beschrieben wurden.

3.3 Übersicht

Bevor im Folgenden auf Details der Realisierung des neu entwickelten Codegenerierungs-Verfahrens eingegangen wird, erfolgt in diesem Abschnitt zunächst eine Darstellung des groben Ablaufs. Im einzelnen ergeben sich die in Abb. 3.2 angegebenen Teilschritte:

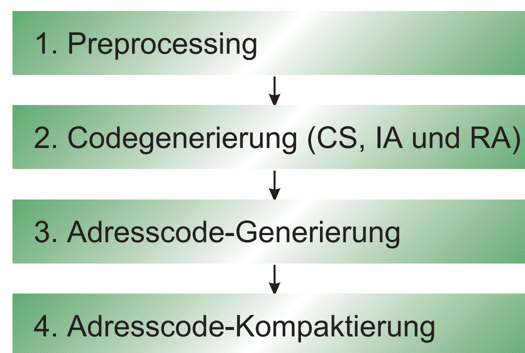


Abb. 3.2: Teilschritte im Back-End

1. Preprocessing (→ Abschnitt 3.4)
   In diesem Schritt werden einige Voraussetzungen zur Durchführung der nachfolgenden Schritte geschaffen. Dies betrifft hauptsächlich die Generierung von alternativen Maschinenprogrammen für die zugrunde gelegte Architektur.

2. Codegenerierung (→ Abschnitt 3.5)
   Die Aufgabe der Codegenerierung besteht in der Bindung konkreter Ressourcen an Graphknoten durch Einschränkung aller Ressource-Mengen auf genau ein Element und in der Zuweisung eines Ausführungszeitpunktes, so dass eine gegebene Zielfunktion optimiert wird. Wie bereits erwähnt stellt dies die Lösung eines NP-harten Optimierungsproblems dar, so dass effiziente Lösungsverfahren erforderlich sind, die das Optimum möglichst gut annähern. Da hierzu die Überwindung von lokalen Optima erforderlich ist, ist dem entwickelten Codegenerator ein Optimierungsverfahren auf Basis eines genetischen Algorithmus zugrunde gelegt (s. [LDL+01, LWDL02]). Das Ergebnis dieses Optimierungsschrittes stellt einen mit Ressourcen überdeckten Datenflussgraphen dar, bei dem bereits Spillcode eingefügt ist.

3. Adresscode-Generierung (→ Abschnitt 3.6)
   Das Ziel dieses Schrittes besteht zunächst in der Bestimmung eines geeigneten Speicherlayouts (*vertikale Adresszuweisung*) durch eine Anordnung der bereits festgelegten Gruppen im Speicher. Danach werden alle zur Adressierung des Speichers erforderlichen Anweisungen bestimmt und ohne Berücksichtigung von parallelen Ausführungsmöglichkeiten in den bereits vorhandenen GeLIR-Code eingefügt, so dass mit Abschluss dieses Schrittes bereits gültiger Assemblercode vorliegt.

4. Adresscode-Kompaktierung (→ Abschnitt 3.7)
   Die Aufgabe dieses Schrittes ist es, die im vorherigen Schritt eingefügten MOs zur Adressierung des Speichers den endgültigen Maschineninstruktionen zuzuweisen, so dass die gegebene Kostenfunktion optimiert wird. Dazu wird wiederum der bereits zur Durchführung der Teilaufgaben CS, IA und RA verwendete genetische Algorithmus eingesetzt.

3.4 Preprocessing

Vor der Durchführung der Teilaufgaben CS, IA und RA besteht Bedarf an einigen Vorverarbeitungsschritten (s. Abb. 3.3). Im einzelnen sind dies:

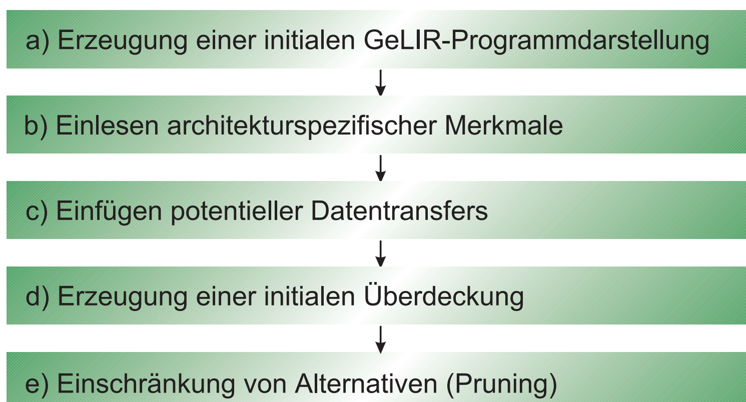


Abb. 3.3: Teilschritte in der Preprocessing-Phase des Back-Ends

a) Erzeugung einer initialen GeLIR-Programmdarstellung.
   Zunächst wird das Quellprogramm mit Hilfe des LANCE-Front-Ends in die LANCE-Zwischendarstellung überführt. Nach der Durchführung einiger der dort vorhandenen maschinenunabhängigen Standardoptimierungen wird die LANCE-IR anschließend in eine initiale GeLIR-Darstellung transformiert. Da alle nachfolgend

durchgeführten Optimierungen und Transformationen unabhängig von den LANCE-Datenstrukturen arbeiten, sind grundsätzlich Konvertierungen von beliebigen anderen Zwischendarstellungen (wie z.B. SUIF) in die GeLIR-Datenstrukturen möglich.

b) Einlesen architekturspezifischer Merkmale.
In diesem Schritt werden die spezifizierten Architekturmerkmale in die internen GeLIR-Datenstrukturen übernommen. Da der Codegenerator neben Performance- ebenfalls Energieoptimierungen durchführen soll, wird auch die Datenbasis des Energiekostenmodells abgelegt. Mit Hilfe des in Abschnitt 1.2.2 beschriebenen Energiekostenmodells kann dann im Codegenerator und im Simulator der Energieverbrauch von GeLIR-Programmen bestimmt werden.

c) Einfügen potentieller Datentransfers.
Zur Modellierung des möglichen Datenflusses zwischen jeweils zwei datenflussabhängigen Graphknoten werden zusätzliche Graphknoten (Copy-MOs) eingefügt. Zu diesem Zeitpunkt werden auch erforderliche Datentransfers zwischen Datenpfad- und AGU-Registern eingefügt, um diese bereits vor der Durchführung der Adresscode-Generierung berücksichtigen zu können. Der generierte Adresscode kann dann nachträglich eingefügt werden, ohne gegen bestehende Registerzuordnungen zu verstoßen und macht somit keine erneute Registerallokations-Phase erforderlich.

d) Erzeugung einer initialen Überdeckung.
Für jeden vorhandenen Graphknoten wird eine Überdeckung mit Ressourcen der Zielmaschine erzeugt. Dies kann im einfachsten Fall unabhängig von der mit diesem Knoten assoziierten AMO und anderen Knoten des Graphen geschehen. An dieser Stelle besteht bereits die Möglichkeit den Suchraum sinnvoll einzuschränken.

e) Einschränkung von Alternativen (Pruning).
In diesem Schritt können basierend auf der zuvor generierten initialen Überdeckung der Graphknoten weitere Einschränkungen des Suchraumes vorgenommen werden, ohne die optimale Lösung auszuschließen. Eine Einschränkung der Ressource-Alternativen wird dabei durch Überprüfung der Knoten- und Kantenkonsistenz mittels der in Abschnitt 2.3.4 vorgestellten Constraintpropagierung vorgenommen. Dies führt dazu, dass zu einem bestimmten Zeitpunkt sichergestellt ist, dass für jeden Graphknoten nur Ressourcen ausgewählt werden können, für die es eine gültige Ressource-Kombination gibt. Des Weiteren wird die Existenz von mindestens einem gültigen Datentransferpfad zwischen je zwei datenabhängigen Knoten gewährleistet.

3.5 Genetischer Codegenerator (GCG)

Im Zuge dieser Phase ist es die Aufgabe des Codegenerators, durch eine Einschränkung der Ressource-Alternativen und die Festlegung einer Ausführungsreihenfolge eine gegebene Kostenfunktion zu optimieren. Es ergibt sich für die Codegenerierung der in Algorithmus 3.1 skizzierte Ablauf.

Algorithmus 3.1 (Codegenerierung)

```
(1) HorizontalAddressAssignment(gelir);
    FOR EACH Funktion fun OF gelir DO
(2)   HorizontalAddressAssignment(fun);
      FOR EACH Basisblock bb OF fun DO
(3)     GCGPreProcessing(bb);
(4)     GCGRun(bb);
      END;
    END;
```

(1 u. 2) `HorizontalAddressAssignment(`*gelir*`)`
`HorizontalAddressAssignemnt(`*fun*`)`

Wie bereits in Abschnitt 3.1.2 auf Seite 51 erwähnt, ergibt sich für den M3-DSP aufgrund des Gruppenspeichers ein Meta-Phasenkopplungsproblem, da die Umsetzung der Phasen CS, IA und RA stark von der Zuordnung (oder Partitionierung) der Daten zu Gruppen des Gruppenspeichers abhängig ist. Aus diesem Grund wird in diesen Schritten zunächst für alle Daten eine Anordnung zu Gruppen des M3-Gruppenspeichers vorgenommen (*horizontale Adresszuweisung*). Entsprechende Techniken zur Ermittlung einer guten Zuordnung sind für das weitere Verständnis dieses Kapitels nicht erforderlich und werden in Verbindung mit der Ausnutzung von SIMD-Operationen in Kapitel 4 beschrieben.

In Schritt 1 werden zunächst alle skalaren Variablen und alle komplexen Datentypen wie z.B. Arrays mit globalem oder statischem Gültigkeitsbereich Gruppen zugewiesen. Danach erfolgt in Schritt 2 eine Zuweisung aller lokal definierten Arrays und Pointer-Variablen. Dies betrifft keine Daten, die aufgrund mangelnder Anzahl freier Register in den Speicher ausgelagert (*gespillt*) werden müssen, da diese erst nach Durchführung der Registerallokation bekannt sind und somit erst im Verlaufe der Codegenerierung Gruppen zugewiesen werden.

(3) `GCGPreProcessing(`*bb*`)`

Vor Starten des genetischen Codegenerators GCG werden hier einige Vorverarbeitungsschritte durchgeführt. Dies betrifft z.B. das Setzen diverser Optimierungsparameter zur Steuerung des genetischen Algorithmus (z.B. Populationsgröße

und Mutationswahrscheinlichkeit) und die Erzeugung von Ausführungsreihenfolge-Beschränkungen zwischen bestimmten Graphknoten, wie Daten-, Output- und Anti-Abhängigkeiten. Des Weiteren besteht durch die Spezifikation von Sequentialisierungskanten die Möglichkeit für zwei Graphknoten, zwischen denen keine Datenabhängigkeit existiert, eine bestimmte Ausführungsreihenfolge zu erzwingen.

(4) `GCGRun(bb)`
In diesem Schritt wird die eigentliche Codegenerierung durchgeführt. Dabei gilt es eine möglichst gute Überdeckung von Graphknoten des Basisblocks *bb* mit alternativen Ressourcen zu ermitteln und alle MOs einer MI zuzuordnen. Anschließend werden die GeLIR-Datenstrukturen mit dem für *bb* besten ermittelten Ergebnis aktualisiert. Dies geschieht durch die Einschränkung der vorhandenen Ressource-Alternativen auf genau ein Element in jeder Menge und eine Neuanordnung der MOs zu MIs. Des Weiteren wird zur Wahrung der Datenkonsistenz eine Aktualisierung der GeLIR-Speicherinformationen von den in den Speicher gespillten Variablen vorgenommen, deren Werte in anderen Basisblöcken wiederverwendet werden.

In den nachfolgenden Abschnitten wird zunächst kurz auf die allgemeine Arbeitsweise von genetischen Algorithmen und anschließend auf die Umsetzung des entwickelten genetischen Codegenerators eingegangen.

3.5.1 Optimierung auf Basis genetischer Algorithmen

Zur Lösung von komplexen Optimierungsproblemen haben sich in der Vergangenheit vielfach genetische Algorithmen (GAs) bewährt. Diese nehmen sich die Natur als Vorbild und lösen Optimierungsprobleme durch Nachahmung des biologischen Evolutionsprozesses [Hol92, Bä96]. Dazu besteht in einem GA eine Population aus mehreren Individuen, die jedes für sich genommen i.d.R. jeweils eine potentielle Lösung des Optimierungsproblems darstellen. Die Repräsentation eines Individuums erfolgt mittels eines Chromosoms, das in einzelne Gene unterteilt ist, die wiederum die Variablen des Optimierungsproblems darstellen. Das Ziel des GAs besteht nun in der Suche nach einer optimalen Belegung der Gene mit Werten (*Allele*), so dass eine gegebene Kostenfunktion optimiert wird. Die Suche wird dabei durch die Anwendung genetischer Operatoren wie *Selektion*, *Mutation* und *Crossover* auf die Individuen der Population gesteuert.

Eine sehr wichtige Eigenschaft von GAs ist, dass bevorzugt Genmaterial von gut (an die Kostenfunktion) angepassten Individuen in nachfolgende Generationen übernommen wird, wobei auch ungünstige Entscheidungen, die in einer früheren Optimierungsphase (Generation) gemacht wurden, revidiert werden können. Aus diesem Grund sind GAs besonders zur Lösung von komplexen Phasenkopplungsproblemen, wie es auch in unserem Fall vorliegt, geeignet.

In Abb. 3.4 ist der Ablauf eines genetischen Algorithmus am Beispiel des entwickelten genetischen Codegenerators skizziert.

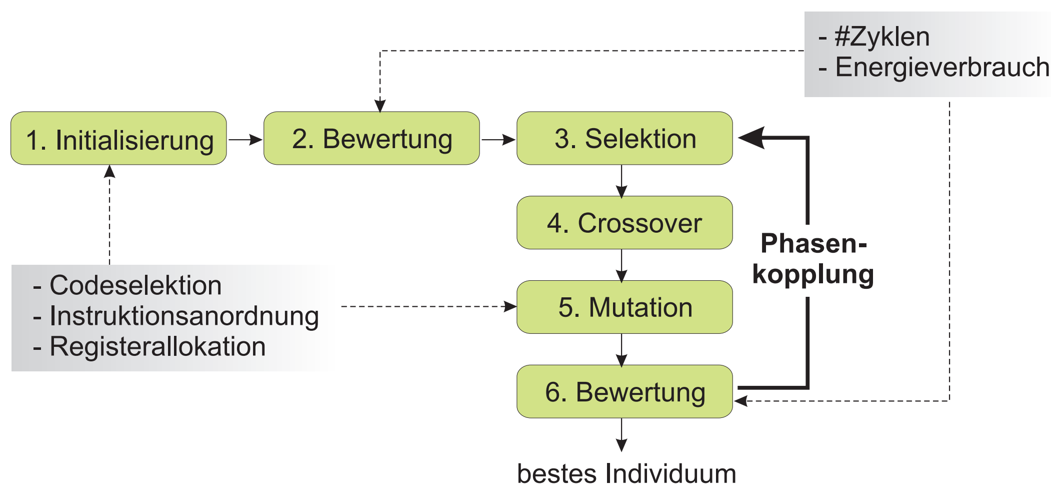


Abb. 3.4: Ablauf des genetischen Algorithmus zur Codegenerierung (GCG)

In der Initialisierungsphase (s. Schritt 1) werden alle Individuen der Anfangspopulation initialisiert, indem die Teilaufgaben der Codegenerierung (CS, IA und RA) durchgeführt werden. Hierzu wird ein probabilistisches *List-Scheduling*-Verfahren verwendet. Eine anschließende Bewertung (s. Schritt 2) der initialisierten Individuen kann anhand unterschiedlicher Kriterien vorgenommen werden. In unserem Fall betrifft dies zum einen die Anzahl der erforderlichen Zyklen und zum anderen den Energieverbrauch der gegebenen Instruktionssequenz. Anhand dieser Bewertung werden dann in der Selektionsphase (s. Schritt 3) die Individuen ausgewählt, die ihre Gene in die nächste Generation vererben dürfen. Im nachfolgenden Schritt werden diese Individuen mittels Crossover zu neuen Individuen rekombiniert und danach einer Mutation unterzogen. Da in unserem Fall durch die Anwendung des Crossover-Operators ungültige Individuen entstehen können, wird in der Mutationsphase eine Korrektur durchgeführt, bei der in Analogie zu der Vorgehensweise im Initialisierungsschritt wiederum die Teilaufgaben CS, IA und RA durchgeführt werden. Solange die Abbruchbedingung (z.B. eine max. Anzahl zu simulierender Generationen) nicht erfüllt ist, dient die anschließende Bewertung in Schritt 6 wiederum als Grundlage für die Selektion in Schritt 3. Im anderen Fall terminiert der Algorithmus und der entsprechende GeLIR-Basisblock wird mit der besten gefundenen Lösung modifiziert.

Im nachfolgenden Abschnitt wird zunächst auf die Möglichkeit der Durchführung einer Mehrzieloptimierung bei Verwendung von genetischen Algorithmen eingegangen. Danach erfolgt eine Beschreibung der chromosomalen Darstellung des Optimierungsproblems, der Initialisierungs- und Bewertungsphase sowie der genetischen Operatoren des Crossovers und der Mutation.

3.5.2 Mehrzieloptimierung mit genetischen Algorithmen

Wie bereits zuvor dargelegt, besteht ein großer Bedarf an Programmen, die einerseits schnell und andererseits energieeffizient sind. Da nicht immer das schnellste Programm auch das energieeffizienteste sein muss, bietet sich mit der Verwendung von genetischen Algorithmen ein Lösungsmechanismus zur Handhabung dieser Problematik an. Unter der Annahme, dass m Zielsetzungen und n Entscheidungsvariablen vorhanden sind, gilt es also o.B.d.A. das Minimierungsproblem $F(\vec{x}) = (F_1(\vec{x}), \ldots, F_m(\vec{x}))$, mit $\vec{x} = (x_1, \ldots, x_n) \in X$ zu lösen (s. auch [ZDT99]). Dabei wird gesagt, dass eine Lösung $a \in X$ eine andere Lösung $b \in X$ *dominiert*, wenn a bezüglich aller gegebenen Zielkriterien mindestens so gut wie b abschneidet und in mindestens einem besser.

Definition 3.1 (Dominanz) *Wenn $a, b \in X$ o.B.d.A. Lösungen eines Minimierungsproblems $F(\vec{x})$ mit m Zielkriterien darstellen, dann wird gesagt, dass die Lösung a die Lösung b dominiert (geschrieben als: $a \prec b$), wenn gilt:*

$$\begin{aligned} &\forall i \in \{1, \ldots, m\} : F_i(a) \leq F_i(b) \qquad \wedge \\ &\exists j \in \{1, \ldots, m\} : F_j(a) < F_j(b) \end{aligned}$$

Analog dazu kann der Begriff der *Nicht-Dominanz* definiert werden:

Definition 3.2 (Nicht-Dominanz) *Wenn $a \in X$ eine Lösung eines Optimierungsproblems darstellt, dann wird gesagt, dass die Lösung a bezüglich einer Menge $X' \subseteq X$ nicht dominiert wird, wenn es kein Element in dieser Menge gibt, das a dominiert. D.h. es gilt:*

$$\nexists a' \in X' : a' \prec a$$

Darauf basierend kann der Begriff der *Pareto-Optimalität* definiert werden, der bei der Mehrzieloptimierung mit genetischen Algorithmen eine wichtige Rolle spielt:

Definition 3.3 (Pareto-Optimalität) *Wenn $a \in X$ eine Lösung eines Optimierungsproblems darstellt, dann wird die Lösung a als pareto-optimal bezeichnet, wenn es in X keine Lösung gibt, die a dominiert.*

Das Ziel der Optimierung stellt also eine Menge von pareto-optimalen Lösungen dar, unter denen es letztlich gilt eine auszuwählen. Besonders wünschenswert sind hier natürlich Lösungen, die hinsichtlich möglichst vieler Zielkriterien das beste Ergebnis erzielen. Dabei gilt es jedoch wiederum unterschiedliche Gewichtungen der Zielsetzungen zu berücksichtigen.

In der Literatur (s. z.B. [Nis97]) werden u.a. die folgenden Möglichkeiten zur Realisierung einer Mehrzieloptimierung aufgeführt:

- Aggregation
  Bei diesem Ansatz gehen die einzelnen Zielkriterien mit unterschiedlichen Gewichtungen in den Gesamt-Zielfunktionswert ein. Wenn ω_i das Gewicht des i-ten Zielkriteriums darstellt, dann ergibt sich bei m Zielkriterien für eine Lösung $\vec{x} \in X$ als Gesamt-Zielfunktionswert:

  $$F(\vec{x}) = \omega_1 * F_1(\vec{x}) + \ldots + \omega_m * F_m(\vec{x})$$

  Insbesondere bei vielen Zielkriterien stellt die Wahl der einzelnen Gewichtungen ein Problem dar. Das Ergebnis eines Optimierungslaufes besteht letztendlich in einer Lösung, die den gegebenen Gesamt-Zielfunktionswert annähert. Eine Auswahl zwischen mehreren pareto-optimalen Lösungen nach Beendigung der Optimierung ist also nicht möglich.

- Wechselnde Zielsetzungen
  Bei diesem Ansatz werden die Individuen bezüglich aller gegebenen m Zielkriterien bewertet. Die Selektion wird dann in m Teilschritten durchgeführt, wobei in jedem dieser Teilschritte ein entsprechender Anteil der Individuen bezüglich eines der Ziele selektiert wird. Individuen, die hinsichtlich mehrerer Zielkriterien gute Lösungen darstellen, dürfen dadurch ihre Gene bevorzugt in die nächste Generation vererben, was tendentiell zu guten Lösungen hinsichtlich mehrerer Zielkriterien führt.

- Pareto-basierte Ansätze
  Ein Ansatz bei dem die Dominanz (bzw. Nicht-Dominanz) von Lösungen gegenüber anderen eine wichtige Rolle spielt, wird z.B. in [Gol89] beschrieben. Dazu werden alle Individuen, die nicht dominiert werden, dem Rang eins zugewiesen. Von den übrig gebliebenen werden dann diejenigen dem nachfolgenden Rang zugewiesen, die von keinem der restlichen Individuen dominiert werden. Dies wird fortgesetzt, bis jedes Individuum einem Rang zugewiesen wurde. Im nachfolgenden Selektionsschritt werden dann Individuen mit einem hohen Rang bevorzugt ausgewählt.

In unserem Fall ist ein Speichern mehrerer pareto-optimaler Lösungen nicht ohne weiteres möglich, da unterschiedliche Lösungen für einen Basisblock i.d.R. auch zu unterschiedlichen Speicherlayouts führen, die an nachfolgende Optimierungsschritte weitergereicht werden müssen. Aus dem Grund bietet sich hier zur Mehrzieloptimierung eine Aggregation von Energie- und Ausführungszeit an. Dies soll dennoch die Kombination mit anderen Ansätzen nicht ausschließen. Ein Vergleich unterschiedlicher Vorgehensweisen zur Durchführung einer Mehrzieloptimierung wird z.B. in [ZDT99] vorgestellt.

3.5.3 Chromosomale Darstellung

Die chromosomale Darstellung einer Lösung des Optimierungsproblems ist essentiell bei der Umsetzung von genetischen Algorithmen. Da die Codegenerierung für einzelne Basisblöcke einer Funktion durchgeführt wird, muss mit einem solchen Chromosom die Darstellung von beliebigen Instruktionssequenzen eines Basisblocks, gegeben durch einen Datenflussgraphen, möglich sein. Um DFGs in Assemblercode abbilden zu können, kodiert in unserem Ansatz ein Individuum die Maschinencodesequenz eines BBs. Dazu werden wie in Abb. 3.5 verdeutlicht, die in der graphbasierten Zwischendarstellung des Quellprogramms vorhandenen Graphknoten als Gene dargestellt, so dass also jedes Individuum dieselbe Anzahl von Genen (hier 23) aufweist. Die schwarz hinterlegten Nummern an den Graphknoten geben die Position des dazugehörigen Gens auf dem Chromosom an.

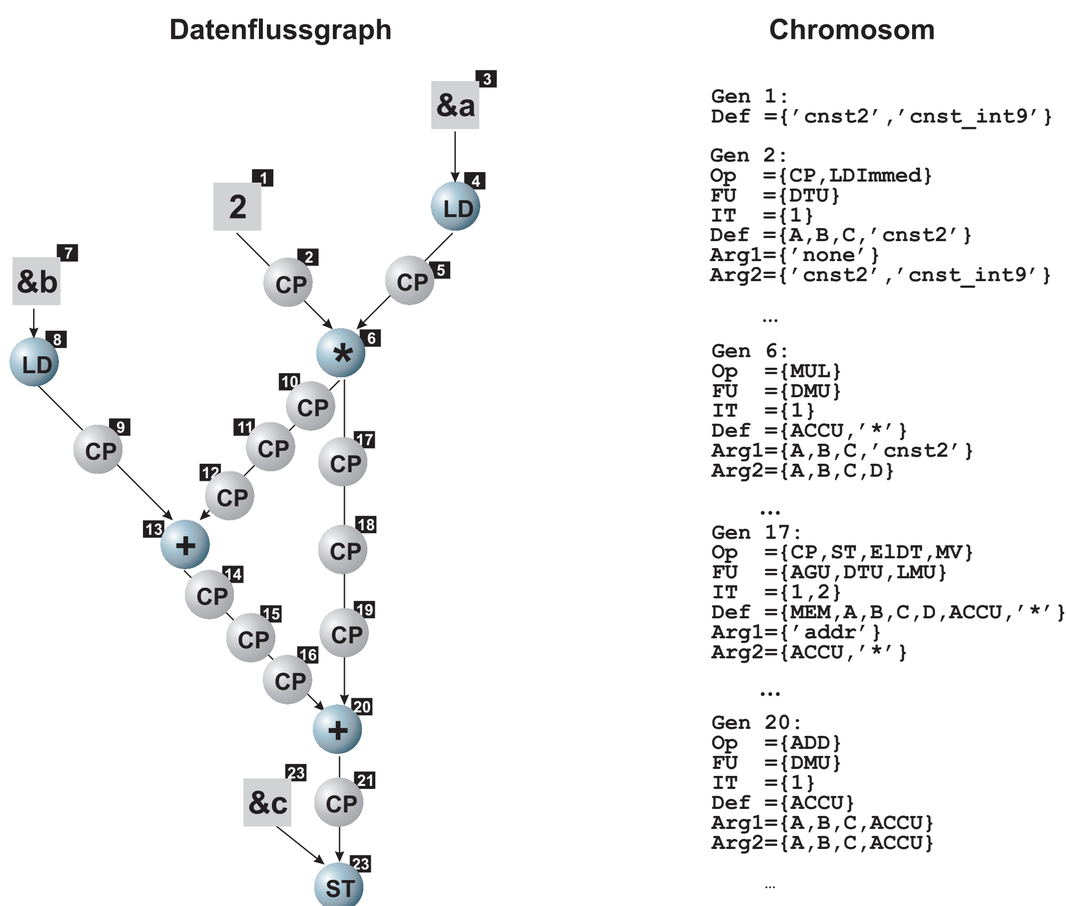


Abb. 3.5: Chromosomale Darstellung

Die Knoten des Graphen entsprechen demnach den auszuführenden Operationen, wobei die potentiell möglichen Datentransfers durch *CP*-AMOs (CP = Copy) repräsentiert werden. Die zur Auswahl stehenden Ausprägungen (z.B. alternative Ressourcen, relativer Ausführungszeitpunkt) eines Gens werden mit dem jeweiligen Gen gespeichert. Durch die gegebene Darstellung der Kombinationsmöglichkeiten von Ressourcen ist keine spezielle

Handhabung zur Ausnutzung von kommutativen Operationen erforderlich, da bereits bei der Spezifikation der Operationen entsprechend gespiegelte Ressource-Kombinationen als separate Alternativen eingefügt wurden. Hieraus ergeben sich bezüglich der Durchführung der Codegenerierung trotz der Existenz von Spezialregistern keine Sonderfälle.

3.5.4 Initialisierung

Die Initialisierung wird für jedes Individuum separat durchgeführt und hat das Ziel, jedem Gen des Individuums ein (oder in unserem Fall) mehrere eindeutige Merkmale (Allele) zuzuordnen, so dass dieses Individuum eine potentielle Lösung des zugrunde gelegten Optimierungsproblems darstellt. In unserem Fall gilt es also, in dieser Phase für jedes Gen eine Reihe von Ressourcen auszuwählen und zuzuordnen. Dabei ist es wünschenswert, möglichst unterschiedliche Individuen zu erzeugen, um einer vorzeitigen Konvergenz des GAs in einem lokalen Optimum vorzubeugen. Um bereits in der Anfangsphase des Optimierungsprozesses möglichst gut an die Kostenfunktion angepasste Individuen zu erzeugen, bietet sich in dieser Phase auch der Einsatz von Heuristiken an, ohne jedoch die Erzeugung einer möglichst heterogenen Population aus den Augen zu verlieren. In unserem Fall bedeutet dies zum Beispiel, dass im Optimalfall jedes Individuum der Population eine andere Codesequenz darstellt. Offensichtlich ist es dazu erforderlich, die Teilaufgaben CS, IA und RA durchzuführen. Das Grundprinzip dieses Verfahrens ist in Algorithmus 3.2 in Form eines Pseudocode-Algorithmus dargestellt und wird nachfolgend näher beschrieben.

Algorithmus 3.2 (Initialisierung)

```
      WHILE ready_set ≠ empty DO
        // Durchführung der Constraintpropagierung nach den Schritten (2) bis (5)
(1)     gene = InstructionScheduling(individuum);
(2)     op_gene = CodeSelection(gene);
(3)     fu_gene = SelectFunctionalUnit(gene);
(4)     it_gene = SelectInstructionType(gene);
(5)     def_gene = RegisterAllocation(gene);
(6)     cs_gene = Compaction(gene);
(7)     Update(ready_set);
      END;
```

Das Verfahren basiert auf der Durchführung einer Variante des *List-Schedulings* [Bak74], bei der alle zu einem bestimmten Zeitpunkt ausführbaren Operationen in einer Menge (`ready_set`) verwaltet werden. Die Schritte (1) bis (7) werden solange durchlaufen, wie Elemente in dieser Menge vorhanden sind. Nach den Schritten (2) bis (5) wird jeweils eine Constraintpropagierung durchgeführt.

(1) `InstructionScheduling(`*`individuum`*`)`

Während beim traditionellen List-Scheduling, die in der Menge ***`ready_set`*** enthaltenen Elemente bezüglich eines heuristischen Auswahlkriteriums nach Prioritäten geordnet und aufgrund dieser Sortierung ausgewählt werden, erfolgt hier analog zu der in [Bea91] beschriebenen Vorgehensweise auch eine probabilistische Auswahl. Diese wird dabei z.B. anhand des frühest- ($ASAP$ = As-Soon-As-Possible) bzw. spätestmöglichen ($ALAP$ = As-Late-As-Possible) Ausführungszeitpunktes einer Operation oder bezüglich der Differenz von ALAP- und ASAP-Werten (*Mobilität*) vorgenommen.

Alle Gene, die auf der AGU auszuführende Operationen kodieren, werden in der nachfolgenden Adresscode-Generierung behandelt und brauchen in dieser Phase nicht näher betrachtet werden.

Die Komplexität dieses Teilschrittes beträgt $O(|V|)$.

(2-5) `CodeSelection(`*`gene`*`)`
`SelectFunctionalUnit(`*`gene`*`)`
`SelectInstructionType(`*`gene`*`)` und
`RegisterAllocation(`*`gene`*`)`

Für das zuvor ausgesuchte Gen wird nun aus den gegebenen Ressource-Alternativen probabilistisch eine Operation, eine Funktionseinheit, ein Instruktionstyp und eine Definitionsregister-Ressource ausgesucht. Nach jedem Teilschritt wird jeweils mittels Constraintpropagierung die Knoten- und Kantenkonsistenz sichergestellt.

Wenn set_{gene} die Ressource-Menge darstellt, aus der eine Auswahl erfolgen soll, dann kann die Komplexität dieses Teilschrittes mit $O(|set_{gene}|)$ abgeschätzt werden.

(6) `Compaction(`*`gene`*`)`

In diesem Schritt gilt es, das aktuelle Gen einem konkreten Kontrollschritt cs_{gene} zuzuweisen. Dazu wird zunächst einmal die Menge der möglichen Ausführungszeitpunkte bestimmt, aus denen später eine Auswahl erfolgt. Wenn cs_{max} den bislang höchsten vergebenen Kontrollschritt darstellt, dann ergeben sich die folgenden Möglichkeiten:

- Auswahl des nächsten zu vergebenden Kontrollschritts $cs_{max} + 1$.
- Parallele Ausführung mit Operationen, denen bereits ein Kontrollschritt (kleiner gleich cs_{max}) zugewiesen wurde und zu denen keine Ressource-Konflikte bestehen. Auf die Vorgehensweise zur Ermittlung der relevanten Kontrollschritte wird im Anschluss an die Beschreibung dieses Algorithmus eingegangen.
- In Gegenwart von CSEs kann es vorkommen, dass das Ergebnis einer Operation bereits in einer Register-Ressource vorliegt, in die auch die aktuelle Operation schreiben kann. Ist ein solcher Fall gegeben, kann ohne Rücksicht auf weitere

Ressource-Konflikte auf die Ausführung dieser Operation verzichtet werden. Dies würde dann durch das Legen eines *Bypass*[2] und die Zuweisung zum speziellen Kontrollschritt 0 umgesetzt werden. Eine genauere Beschreibung der Vorgehensweise erfolgt weiter unten in diesem Abschnitt auf Seite 71 anhand eines Beispiels.

Da cs_{max} nicht größer als die Anzahl von Graphknoten werden kann, beträgt die Komplexität dieses Teilschrittes $O(|V|)$.

(7) `Update(ready_set)`
Abschließend wird das behandelte Gen aus der Menge ***ready_set*** entfernt und es werden die Gene aufgenommen, die nun nicht mehr gegen Daten-, Anti-, Output-, und Sequentialisierungs-Constraints verstoßen.
Die Komplexität dieses Schrittes beträgt $O(|E|)$.

Die Laufzeit zur Initialisierung eines Genes (entspricht einem Schleifendurchlauf) beträgt $O(|E|)$ und kann dementsprechend für alle Gene mit $O(|V| * |E|)$ abgeschätzt werden.

Parallelisierung unter Berücksichtigung von Ressource-Constraints

Bei der Zuweisung einer Operation zu einem Kontrollschritt, in dem bereits andere Operationen ausgeführt werden, ist die Einhaltung einer Reihe von Ressource-Constraints erforderlich. Mit Hilfe des Algorithmus 3.3 kann die Menge `cs_set` von Kontrollschritten ermittelt werden, zu denen eine parallele Ausführung möglich ist. Die dort verwendeten Variablen *def*, *arg*, *fu* und *it* stellen Ressource-Mengen für Zielregister, Argumentregister, Funktionseinheiten und Instruktionstypen dar. Mit *gene*-indizierte Variablen repräsentieren dabei Ressource-Mengen vom aktuell zuzuordnenden Gen und die mit *cs*-indizierten Variablen jeweils die Vereinigung aller bisher im Kontrollschritt *cs* verwendeten Ressourcen.

Beginnend beim bislang höchsten zugewiesenen Kontrollschritt cs_{max} wird sukzessive bis zum Kontrollschritt 1 bzw. Erreichen einer Abbruchbedingung getestet, ob eine parallele Ausführung zum aktuellen Kontrollschritt möglich ist. Dazu wird mit dem ersten Teil von Bedingung `(1)` zugesichert, dass bereits zuvor definierte Register-Ressourcen nicht vorzeitig neudefiniert werden und mit dem zweiten Teil, dass in den vom zuzuordnenden Gen verwendeten Argument-Registern die benötigten Werte bereits vorliegen (→ Datenflussabhängigkeit). Wird gegen diese Bedingung verstoßen, ist eine Zuweisung an kleinere Kontrollschritte nicht mehr möglich und der Algorithmus terminiert. Mit der Bedingung

[2] In [Bas01] wird in diesem Zusammenhang von einer „Überlagerung von Datentransferpfaden“ gesprochen, die allerdings nur unter der Voraussetzung möglich ist, wenn zwei FMOs „absolut identische Datentransfer-Operationen“ repräsentieren.

Algorithmus 3.3 (Kontrollschritt-Ermittlung)

```
    FROM cs = cs_max DOWNTO 1 DO
(1)     IF def_gene ∩ def_cs ≠ empty AND arg_gene ∩ def_cs ≠ empty DO
          STOP;
        END;

(2)     IF it_gene == it_cs AND fu_gene ≠ fu_cs DO
          cs_set.insert(cs);
        END;

(3)     IF def_gene ∩ arg_cs ≠ empty DO
          STOP;
        END;
    END;
```

(2) wird sichergestellt, dass dieselben Instruktionstypen verwendet werden und dass eine Funktionseinheit innerhalb desselben Kontrollschritts nicht mehrfach verwendet wird. Wenn diese Bedingung erfüllt ist, kann die durch das Gen *gene* repräsentierte Operation im aktuellen Kontrollschritt *cs* ausgeführt werden und wird somit in die Menge `cs_set` eingefügt. Der Algorithmus terminiert ebenfalls, wenn die Operation des Gens Register-Ressourcen definiert, die im aktuellen Kontrollschritt *cs* von einer anderen Operation verwendet werden (s. Bedingung (3)).

Vermeidung der Ausführung von Operationen mittels Bypass

Wie bereits in Abschnitt 3.1.1 festgestellt, weist die Durchführung einer graphbasierten im Vergleich zu einer baumbasierten Codeselektion ein erhebliches Optimierungspotential auf. So besteht neben einer effektiveren Umsetzung der Codeselektion ebenfalls die Möglichkeit, einmal berechnete Werte von CSEs bis zu ihren Verwendungen in Registern zu halten. Mit dem auf Seite 67 in Abb. 3.5 dargestellten Datenflussgraphen, wie er nach Einfügen der potentiell möglichen Datentransfers vorkommen kann, taucht nun allerdings das folgende Problem auf: Das Ergebnis der CSE (s. Ergebnis der Multiplikation) wird in zwei nachfolgenden Operationen (s. Additionen) benötigt. Aus diesem Grund ist ein Transport dieses Wertes auf jeweils separaten Verbindungswegen zu den jeweiligen Verwendungen vorgesehen. Dass dies jedoch nicht die Möglichkeit zur Nutzung identischer Datentransferwege abdeckt, wird in Abb. 3.6 a) bis d) anhand des relevanten Ausschnitts aus dem DFG aus Abb. 3.5 verdeutlicht.

Bereits im Verlauf der Codegenerierung abgearbeitete Gene (oder Graphknoten) sind schattiert dargestellt und bereits mit den zugewiesenen MOs und der Zielregister-Ressource markiert. So wurden in Teil a) der Abbildung bereits die Gene 6, 10, 11, 17 und

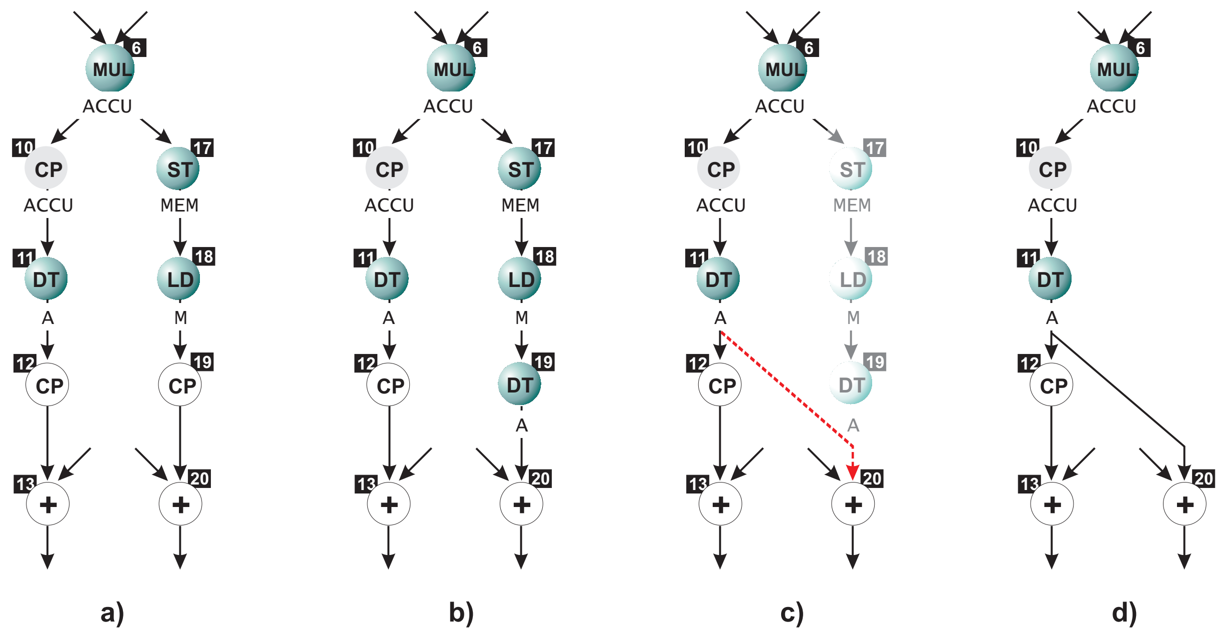


Abb. 3.6: Beispiel der Umsetzung eines Bypass

18 abgearbeitet. Gen 10 führt dabei eine virtuelle Kopierfunktion aus und Gen 11 einen Datentransfer vom `ACCU` in die Ressource `A`, während im rechten Teil die CSE zunächst in den Speicher geschrieben (Gen 17) und danach wieder aus dem Speicher geladen wird (Gen 18). Teil b) der Abbildung zeigt nun, dass dem Gen 19 die MO `DT` und das Zielregister `A` zugewiesen wurde. Da der zu schreibende Wert allerdings bereits im Register `A` vorliegt (s. Gen 11), kann auf die Ausführung von Gen 19 verzichtet und ein Bypass von Gen 11 nach Gen 20 gelegt werden (s. Teil c)). Dies führt dazu, dass die zuvor ausgeführten Speicherzugriffe in Gen 17 und 18 nicht mehr erforderlich sind. Diese stellen somit Dead-Code dar und können eliminiert werden (Teil d)). Auf diese Weise können nicht nur Datentransfers eliminiert werden, sondern ebenfalls Load-, Store- und arithmetische Operationen. Die Ermittlung, ob bestimmte Gene denselben Wert definieren, kann einmalig vor der Durchführung der Codegenerierung mit Hilfe der Zuweisung von *Value*-Nummern durchgeführt werden. Zwei Gene erhalten dabei dieselbe Value-Nummer, wenn diese denselben Wert definieren. Ein solcher Algorithmus ist auf den GeLIR-Datenstrukturen implementiert und kann in [Muc97] nachgelesen werden.

In Abb. 3.7 ist ein mit Hilfe des in diesem Abschnitt vorgestellten Verfahrens initialisiertes Individuum abgebildet. Es ist zu erkennen, dass bei allen Genen die Mengen alternativer Auswahlmöglichkeiten auf genau ein Element eingeschränkt und ebenso eine Zuordnung der Gene zu Kontrollschritten (also Zuordnung von MOs zu MIs) vorgenommen wurde. Aus Platzgründen wurden keine Gene angegeben, die mit einer virtuellen Copy-Anweisung überdeckt wurden und somit nicht ausgeführt werden.

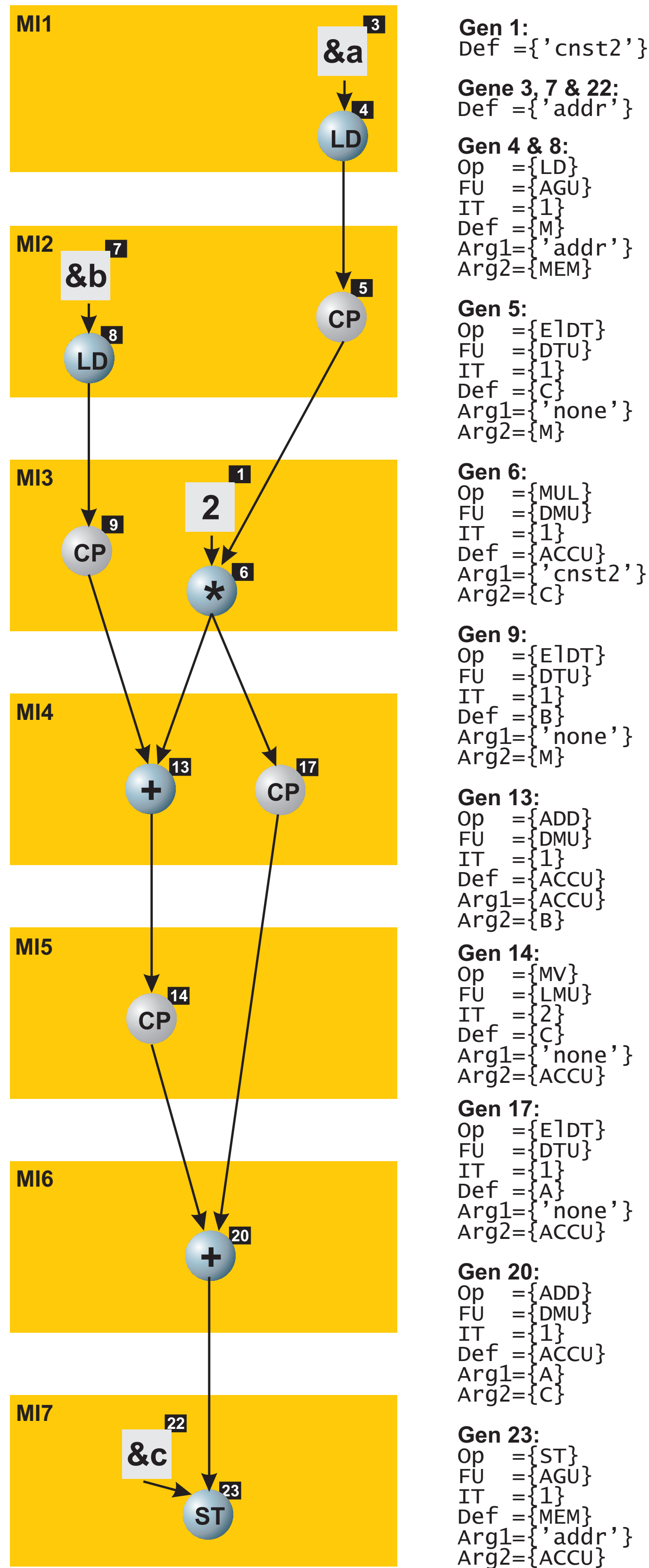


Abb. 3.7: Initialisiertes Individuum

3.5.5 Bewertung der Individuen

Die Bewertung der Individuen einer Population dient zur Differenzierung der unterschiedlichen Lösungen. O.B.d.A. stellen Individuen, die geringe Kosten verursachen, bessere Lösungen dar, als solche mit höheren Kosten und werden im nachfolgenden Selektionsschritt mit einer größeren Wahrscheinlichkeit berücksichtigt. Aufgrund der Komplexität der Problemstellung kann es dazu kommen, dass Individuen gegen bestimmte Randbedingungen (wie z.B. Ressource-Constraints) verstoßen. Da es in diesem Fall erforderlich wäre, Korrekturcode einzufügen, wird ein solches Individuum ind mit einer Strafe $penalty_{ind}$ belegt, die mit einem Gewicht von ω_p mit in die Bewertung eingeht. Die Höhe der Strafe richtet sich dabei nach dem Ausmaß des Fehlers und könnte z.B. die Anzahl von Prozessorinstruktionen umfassen, die zur Korrektur zusätzlich eingefügt werden müssten. Als sinnvoller Wert für ω_p könnte auch ein Wert dienen, der sicherstellt, dass Individuen, die gegen Randbedingungen verstoßen, schlechter bewertet werden als solche, die alle Randbedingungen erfüllen.

Eine Optimierung hinsichtlich unterschiedlicher Optimierungsziele kann aufgrund der flexiblen Gestaltungsmöglichkeit der Bewertungsfunktion in einem GA auf sehr einfache Weise realisiert werden. Da wir in unserem Fall neben einer Reduzierung der Ausführungszeit auch eine Reduzierung des Energieverbrauchs anstreben, werden im Folgenden alternativ verwendbare Kostenfunktionen vorgestellt.

Minimierung der Ausführungszeit

Die Bewertung eines Individuums bezüglich der Ausführungszeit ist in $O(1)$ möglich, da bereits während der Erzeugung des Individuums alle erforderlichen Informationen generiert werden. So reicht neben der Anzahl der Verstöße gegen Ressource-Constraints, die Kenntnis der Anzahl benötigter Kontrollschritte cs_{ind} eines Individuums aus. Die Kosten für ein Individuum ind ergeben sich somit aus einem Straf- und einem Zeitanteil:

$$cost_{ind} = \underbrace{\omega_p * penalty_{ind}}_{Strafanteil} + \underbrace{\omega_{cs} * cs_{ind}}_{Zeitanteil} \tag{3.1}$$

Wird ω_{cs} auf eins und ω_p groß genug gewählt (z.B. max. Anzahl ausführbarer Kontrollschritte), so werden Individuen, die gegen keine Randbedingungen verstoßen, in jedem Fall besser bewertet als die übrigen.

Minimierung des Energieverbrauchs

Die Bestimmung des Energieverbrauchs $energy_{ind}$ einer bestimmten Codesequenz eines Individuums ind kann mittels des in Abschnitt 1.2.2 beschriebenen Energiekostenmodells

in $O(|MIs|)$ sehr effizient durchgeführt werden, da beginnend mit der ersten MI nur Kosten zur unmittelbar nachfolgenden MI bestimmt werden müssen. Als Kostenfunktion kann z.B.

$$cost_{ind} = \underbrace{\omega_p * penalty_{ind}}_{Strafanteil} + \underbrace{\omega_{en} * energy_{ind}}_{Energieanteil} \tag{3.2}$$

verwendet werden.

Kombination von Optimierungszielen

Die alleinige Optimierung des Energieverbrauchs ist normalerweise nicht ausreichend, da i.d.R. weitere Randbedingungen bezüglich der Ausführungszeit vorhanden sind. Aus diesem Grund ist es wichtig, dass eine Codegenerierung unter Berücksichtigung mehrerer, oftmals widersprüchlicher, Optimierungsziele möglich ist. Soll zum Beispiel aufgrund von vorhandenen Realzeitbedingungen eine Optimierung in erster Linie hinsichtlich der Ausführungszeit vorgenommen werden, könnte die folgende Kostenfunktion verwendet werden:

$$cost_{ind} = \underbrace{\omega_p * penalty_{ind}}_{Strafanteil} + \underbrace{\omega_{cs} * cs_{ind}}_{Zeitanteil} + \underbrace{\omega_{en} * energy_{ind}}_{Energieanteil} \tag{3.3}$$

Diese Kostenfunktion enthält in Ergänzung zu der in Gleichung 3.1 angegebenen Kostenfunktion einen Energieanteil. Wenn $energy_{max}$ den maximal möglichen Energieverbrauch darstellt und ω_{en} auf $(energy_{max})^{-1}$ gesetzt wird, werden hierdurch lediglich Kosten von kleiner gleich eins verursacht, so dass „schnellere“ Lösungen in jedem Fall gegenüber den energieärmeren bevorzugt werden. Weitere Optimierungsziele wie die Minimierung der Codegröße können auf analoge Weise berücksichtigt werden.

3.5.6 Selektion

Vor Durchführung der eigentlichen Selektion wird eine Elite-Selektion durchgeführt, indem eine bestimmte Anzahl von Individuen unverändert in die nächste Generation übernommen wird. Im Zuge der Selektion gilt es nun, auf der Basis der zuvor durchgeführten Bewertung, diejenigen Individuen auszuwählen, die ihre Gene in die nächste Generation vererben dürfen. Zur Durchführung der Selektion wird der Auswahlalgorithmus *Stochastic Universal Sampling* (SUS) angewendet (s. z.B. [Nis97] für Details).

3.5.7 Crossover

Der Crossover-Operator hat die Aufgabe, das Genmaterial zweier Individuen (*Eltern*) zu neuen Individuen (*Kinder*) zu rekombinieren, die jedes für sich genommen wiederum eine Lösung des Optimierungsproblems darstellen. Die erzeugten Nachkommen bestehen dabei teilweise aus Genen des einen und teilweise aus Genen des anderen Elter. Das Crossover wird dabei nur mit einer bestimmten Wahrscheinlichkeit durchgeführt. Wünschenswert sind hier Crossover-Varianten, die nur gültige Individuen erzeugen, also korrektheitserhaltend sind. Leider ist dies aufgrund der großen Anzahl einzuhaltender Randbedingungen (wie z.B. Datenabhängigkeiten und Ressourcenbeschränkungen) nicht ohne weiteres umsetzbar. Aus diesem Grund wird in der nachfolgend durchlaufenen Mutationsphase zusätzlich eine Korrektheitsüberprüfung aktueller Genbelegungen durchgeführt, um wiederum gültige Individuen zu erzeugen. Dadurch können alle gängigen Crossover-Verfahren verwendet werden. Häufig verwendete Varianten sind:

- Einpunkt-Crossover
  Bei dieser Crossover-Variante werden die Gene zweier Elter derart miteinander kombiniert, dass alle Gene ausgetauscht werden, die nach einer probabilistisch bestimmten Position auf dem Chromosom liegen. In Abb. 3.8 ist dies anhand von Chromosomen mit neun Genen verdeutlicht, deren Allele hier beispielhaft Ausführungszeitpunkte der damit assoziierten Operation darstellen. Der Pfeil in diesem Beispiel gibt den Crossover-Punkt an und besagt also, dass die Gene vier bis neun ausgetauscht werden sollen.

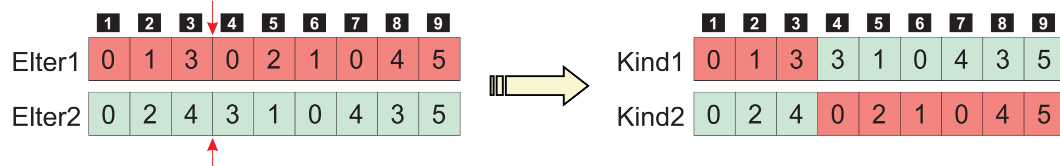


Abb. 3.8: Beispiel zur Umsetzung des Einpunkt-Crossovers

- Zweipunkt-Crossover
  Diese Crossover-Variante entspricht der des Einpunkt-Crossover, allerdings werden hier alle Gene ausgetauscht, die zwischen zwei probabilistisch bestimmten Stellen des Chromosoms vorkommen (s. Abb. 3.9). Eine Erweiterung dieser Crossover-Variante auf eine beliebige Anzahl von Crossover-Punkten ist problemlos möglich und wird üblicherweise als N-Punkt-Crossover bezeichnet.

- Uniform-Crossover
  Beim Uniform-Crossover wird für jedes einzelne Gen des Chromosoms separat mit

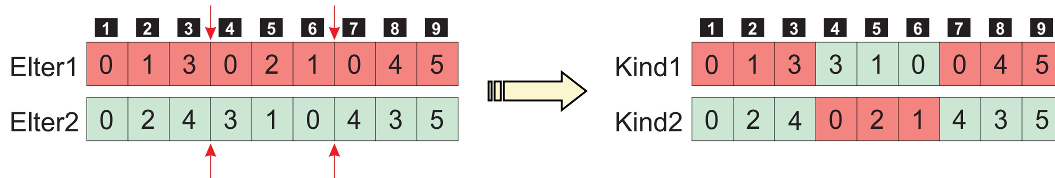


Abb. 3.9: Beispiel zur Umsetzung des Zweipunkt-Crossovers

einer bestimmten Wahrscheinlichkeit entschieden, ob dieses Gen ausgetauscht werden soll (s. auch markierte Gene in Abb. 3.10).

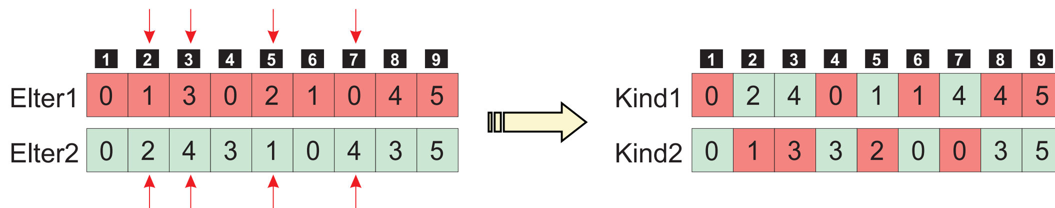


Abb. 3.10: Beispiel zur Umsetzung des Uniform-Crossovers

Alle zuvor vorgestellten Crossover-Varianten haben gemein, dass sie einen Austausch der Gene aufgrund ihrer relativen Anordnung auf dem Chromosom ohne Berücksichtigung der aktuellen Genbelegungen durchführen. Die in [SMM+91] vorgestellten Untersuchungen mit sechs unterschiedlichen Crossover-Varianten für zwei reale Anwendungsprobleme zeigen, dass für jedes der Anwendungsprobleme jeweils eine anderere Crossover-Variante zu bevorzugen ist. Im Vergleich mit Crossover-Varianten, die lediglich einen Austausch der Gene aufgrund ihrer Position auf dem Chromosom vornehmen, erwies sich bei Problemstellungen mit Nachfolge-Relationen wie dem bekannten *Traveling-Salesman-Problem* (TSP) ein, gegenüber dem in [WSS91] vorgestellten, verbessertes *Edge-Recombination*-Crossover als geeigneter.

Eine einfache Crossover-Variante, die einen Austausch von Genen unter Berücksichtigung der aktuellen Ausführungsreihenfolge der Gene vornimmt, wird nachfolgend beschrieben. Diese Crossover-Variante (im Folgenden *CS-Crossover* genannt) basiert auf der Idee, die bis zu einem bestimmten Kontrollschritt cs_{cross} zugewiesenen Gene eines Individuums unverändert zu lassen, so dass bis zu diesem Ausführungszeitpunkt zunächst wiederum gültige Teillösungen entstehen. Es werden also nur die Gene ausgetauscht, die einem späteren Kontrollschritt als cs_{cross} zugewiesen wurden. Der Parameter cs_{cross} wird für jedes durchzuführende Crossover zwischen zwei Elter neu bestimmt. In Abb. 3.11 ist dies für $cs_{cross} = 2$ verdeutlicht.

Modifikationen dieser Crossover-Variante sind z.B. in Anlehnung zum N-Punkt-Crossover möglich, indem durch die Bestimmung mehrerer Kontrollschritte eine Reihe von Teilbereichen ausgetauscht werden.

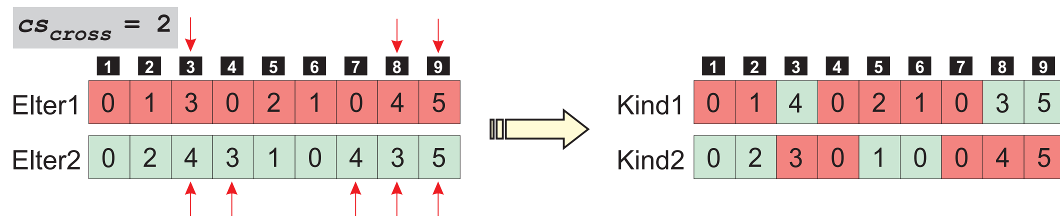


Abb. 3.11: Beispiel zur Umsetzung des CS-Crossovers

Es bleibt festzuhalten, dass durch diese Crossover-Variante nicht die Einhaltung aller Constraints zugesichert werden kann. Es ist jedoch zu erwarten, dass durch die Berücksichtigung der Ausführungszeitpunkte zumindest größere Teilbereiche konsistent gehalten werden können. An dieser Stelle ist mit Sicherheit noch Spielraum zur Entwicklung von speziell an diese Aufgabe angepassten Crossover-Varianten. Beispielsweise könnte eine Anpassung der im Bereich der *genetischen Programmierung* veröffentlichten graphbasierten Crossover-Varianten (s. z.B. [Pol97, KB02]) Potential für Verbesserungen offenbaren. Eine Bewertung der in diesem Abschnitt vorgestellten Crossover-Varianten erfolgt in Abschnitt 3.8.1.

3.5.8 Mutation

Die Aufgabe der Mutation besteht in der Erzeugung von neuem Genmaterial oder in der Wiedergewinnung von Genmaterial, das im Verlaufe des Evolutionsprozesses verloren gegangen ist. Da durch das zuvor durchgeführte Crossover Nachkommen erzeugt werden können, die gegen Constraints verstoßen, wird die Mutation in Verbindung mit einer Korrektur durchgeführt. In Analogie zur Initialisierungsphase werden wiederum die Teilaufgaben CS, IA und RA durchgeführt, um eventuelle Verstöße gegen Ressource-Constraints aufzudecken und um bei Bedarf eine korrektheitserhaltende Mutation durchführen zu können. Im Grundsatz kann hier auf das in der Initialisierung durchgeführte Verfahren zurückgegriffen werden, wobei diesmal jedoch die aktuellen Genbelegungen eines zu mutierenden Individuums mitberücksichtigt werden. So kann einerseits anhand der aktuell zur Auswahl stehenden alternativen Ressourcen überprüft werden, ob ein bestimmtes Allel korrigiert werden muss und andererseits im Falle einer durchzuführenden Mutation eine probabilistische Auswahl aus der Menge der Alternativen erfolgen.

Ein möglicher mittels Crossover und Mutation erzeugter Nachkomme ist in Abb. 3.12 dargestellt. Gegenüber dem in Abb. 3.7 dargestellten Individuum werden nun weniger Datentransfers verwendet und anstelle der Multiplikation und der beiden Additionen zwei MAC-Operationen (s. MI4 und MI5) ausgeführt. Obwohl die Multiplikation (s. Gen 6) dazu doppelt ausgeführt werden muss, konnte die Anzahl der Kontrollschritte von sieben auf sechs reduziert werden.

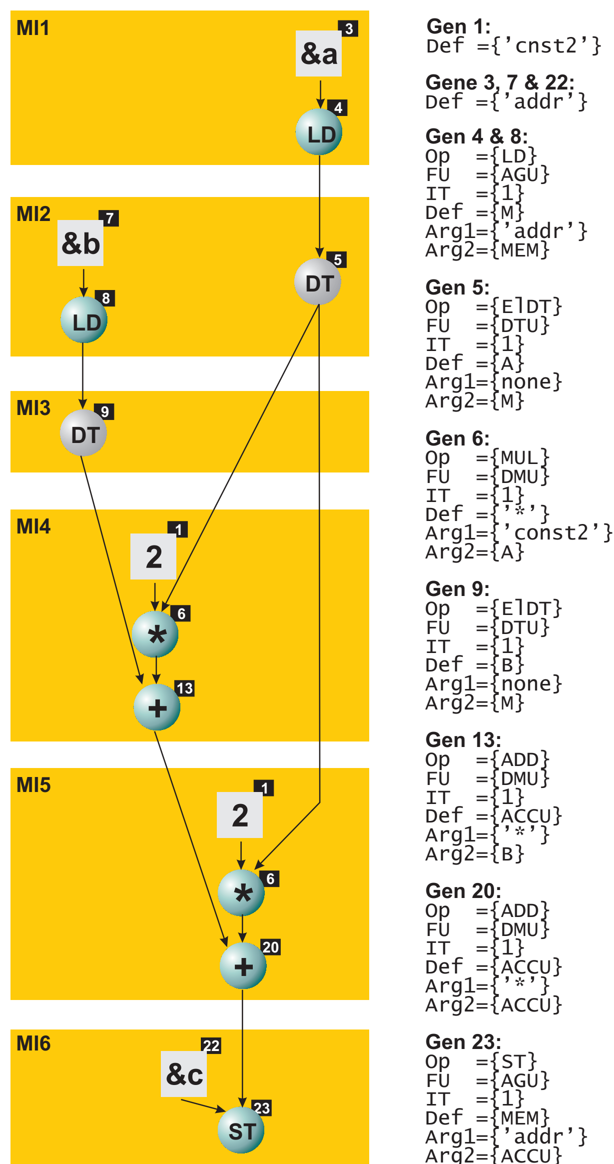


Abb. 3.12: Mittels Crossover und Mutation erzeugter Nachkomme

3.6 Adresscode-Generierung

Allgemein besteht das Ziel der Adresscode-Generierung darin, mit geringst möglichem Overhead (in Form von zusätzlichen Maschineninstruktionen oder Energiekosten) alle in einem Programm befindlichen Adressen von Speicherzugriffen zu berechnen. Dazu müssen in unserem Fall die folgenden Voraussetzungen erfüllt sein:

- Vor der Durchführung der Codegenerierung werden alle AMOs, die mit AGU-MOs umgesetzt werden sollen, markiert und bei der Durchführung der Teilaufgaben CS, IA und RA unberücksichtigt gelassen. AMOs, für die keine Umsetzung mit AGU-Anweisungen möglich ist, wie z.B. Subtraktionen oder Multiplikationen, müssen

dann im Datenpfad berechnet werden und verhindern damit die Ausnutzung der mit dem Einsatz von AGUs verbundenen Vorteile. Aus diesem Grund wird bereits vor Starten der Codegenerierung mittels einfacher algebraischer Transformationen versucht, nicht auf der AGU ausführbare Adressberechnungen durch solche zu ersetzen, die auf der AGU ausführbar sind. Zum Beispiel wird eine Subtraktion einer positiven Konstanten durch eine Addition mit einer negativen Konstanten ersetzt.

- Ebenfalls vor der Durchführung der Codegenerierung wird durch Einfügen von Datentransfers dafür gesorgt, dass alle erforderlichen Datentransfers von Registern des Datenpfades zu AGU-Registern (und umgekehrt) vorhanden sind. Hierdurch werden Registerkonflikte aufgrund der bereits durchgeführten Registerallokation vermieden. Dies spielt insbesondere für Adressberechnungen eine wichtige Rolle, die mindestens einen Adressbestandteil aus dem Gruppenspeicher laden müssen, da dieses Laden in jedem Fall die Verwendung von mehreren Registern des Datenpfades erfordert.

- Nach der Durchführung der Codegenerierung ist die Zugriffsreihenfolge auf den Speicher bekannt. Des Weiteren wurden allen verwendeten Variablen bereits konkrete Adressen zugewiesen.

3.6.1 Algorithmus zur Adresscode-Generierung

In Analogie zur Durchführung der Codegenerierung wird auch die Generierung des Adresscodes für jeden Basisblock separat durchgeführt. Zur Verdeutlichung der Vorgehensweise zur Adresscode-Generierung soll zunächst das in Abb. 3.13 einfache C-Programm dienen.

```
int A[32];
int B[32];
int a;

int main()
{
 for(i=0; i<10; i++)
  for(j=0; j<16; j++)
   {
    A[i+5] = A[i];
    A[i+j+2] = B[i+j+1] + A[i+2];
    B[6] = a;
   }
}
```

Abb. 3.13: Beispiel-Programm für Adresscode-Generierung

Das Programm besteht aus zwei ineinander geschachtelten Schleifen, deren innerste An-

weisungen eine Reihe von Zugriffen auf die Arrays A und B und einen Zugriff auf die globale Variable a enthalten. In Abb. 3.14 ist zusätzlich ein Speicherlayout für die beiden Arrays und die skalare Variable a angegeben, wie es vor der Generierung des Adresscodes vorliegen könnte.

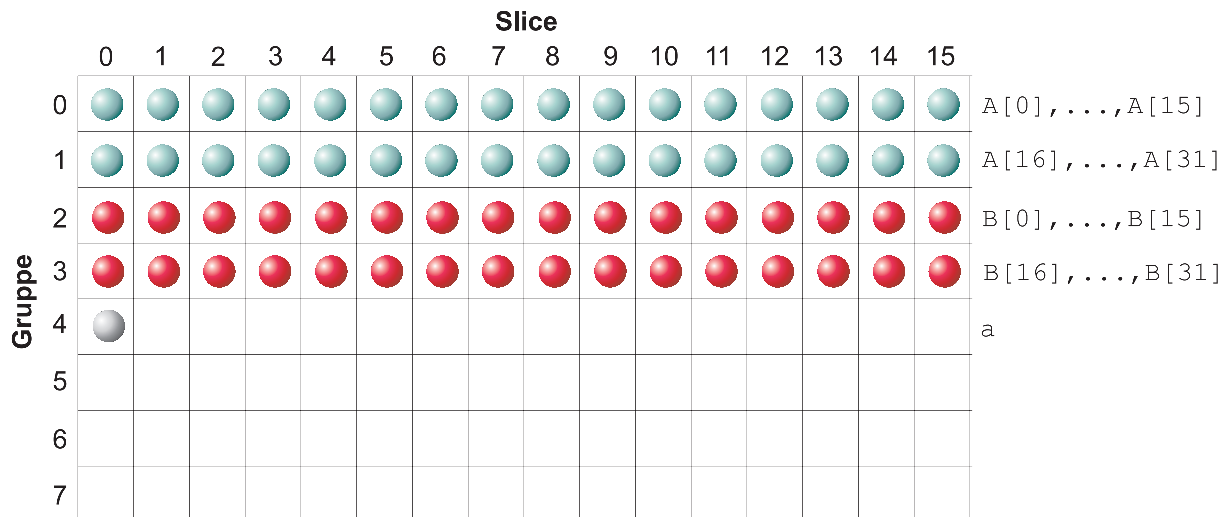


Abb. 3.14: Beispiel Adresscode-Generierung: Speicherlayout

Es ist zu erkennen, dass die Elemente der beiden Arrays jeweils zwei Gruppen des Gruppenspeichers beanspruchen, die aufeinander folgend im Gruppenspeicher angeordnet sind. Das Element `B[6]` ist z.B. in Gruppe 2 und Slice 6 abgelegt, was der Speicheradresse 38 entspricht. Nach der Durchführung der Phasen CS, IA und RA könnte sich in jeder Iteration ein Zugriff auf die Daten in der folgenden Reihenfolge ergeben:

`A[i]`, `A[i+5]`, `B[i+j+1]`, `A[i+2]`, `A[i+j+2]`, a, `B[6]`

Da für jeden der Datenzugriffe i.d.R. ein Speicherzugriff erforderlich ist, würde sich unter Berücksichtigung des gegebenen Speicherlayouts der in Abb. 3.15 dargestellte Graph ergeben.

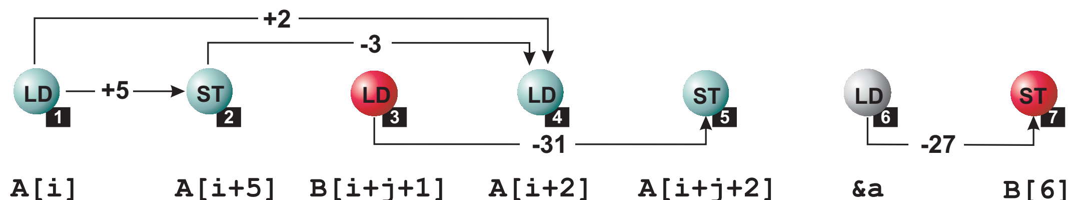


Abb. 3.15: Beispiel Adresscode-Generierung: Speicherzugriffs-Graph

Die Knoten des Graphen entsprechen den Speicherzugriffen und sind in der Reihenfolge ihres Zugriffs von links nach rechts angeordnet und mit Ordnungsnummern versehen.

Anhand der gewichteten Kanten kann nun erkannt werden, ob zwischen zwei bestimmten Knoten ein konstanter Adressoffset besteht und damit eine Ausnutzung von Auto-Inkrement- bzw. Auto-Modify-Anweisungen möglich ist. Es besteht z.B. zwischen den Knoten eins und zwei ein konstanter Adressoffset von fünf und zwischen den Knoten drei und fünf von -31. Da die Erzeugung des Graphen auf Basis der Ausführungsreihenfolge von Speicherzugriffen (einschließlich von Spills) durchgeführt wird, besteht offensichtlich eine große Abhängigkeit zur zuvor durchgeführten Codegenerierungs-Phase.

Anhand des nachfolgend angegebenen Pseudocode-Algorithmus wird nun der prinzipielle Ablauf der umgesetzten Adresscode-Generierung näher beschrieben:

Algorithmus 3.4 (Adresscode-Generierung)

```
(1) VerticalAddressAssignment(gelir);
    FOR EACH Funktion fun OF gelir DO
      FOR EACH Basisblock bb OF fun DO
(2)     mem_seq = InitMemSequence(bb);
(3)     mem_seq = InitOffsets(mem_seq);
(4)     mem_seq = AddressCodeGeneration(mem_seq);
      END;
(5)   fun = RedundantAddressCodeElimination(fun);
    END;
```

(1) `VerticalAddressAssignment(gelir)`
Die in den vorherigen Codegenerierungs-Phasen im Rahmen der horizontalen Adresszuweisung gebildeten Gruppen werden in diesem Schritt festen Adressen zugewiesen.

(2) `InitMemSequence(bb)`
In diesem Schritt werden zunächst alle in einem bestimmten Basisblock vorkommenden Speicherzugriffe entsprechend ihrer Zugriffsreihenfolge zu einer Speicherzugriffs-Sequenz angeordnet. Jeder Speicherzugriff wird also mit einem Knoten des Graphen assoziiert. Jedem Knoten wird entsprechend der Position des Speicherzugriffs in der Zugriffssequenz eine Ordnungsnummer zugewiesen (s. auch Abb. 3.15). Zur Ermittlung der Kantengewichte zwischen je zwei Knoten in Schritt 3 werden alle Adressbestandteile eines Speicherzugriffs bestimmt. Die konstanten Adressbestandteile, wie z.B. Basisadressen von Arrays und Arrayoffsets werden dann aufsummiert und bilden den *konstanten* Adressoffset. Alle anderen Adressbestandteile (wie z.B. Schleifen-Indexvariablen) stellen *variable* Adressbestandteile dar. Für den Speicherzugriff `B[i+j+1]` setzt sich der konstante Adressoffset aus der Basisadresse von Array B und der Konstante eins zusammen. Der variable Adressoffset ergibt sich

durch die aktuellen Werte der Schleifen-Indexvariablen i und j. Zusätzlich wird für jeden Knoten jeweils eine Menge von Adressregistern verwaltet, die potentiell zur Adressierung des Speichers verwendet werden können.

(3) `InitOffsets(`*mem_seq*`)`
Nachdem im vorherigen Schritt die Knoten des gerichteten Graphen erzeugt und initialisiert worden sind, werden in diesem Schritt gewichtete Kanten in den Graphen eingefügt. Es wird genau dann eine gerichtete Kante von Knoten n_i nach n_j eingefügt, wenn $i \leq j$ gilt und zwischen den beiden betroffenen Knoten entweder keine oder dieselben variablen Adressoffsets zur Adressierung benötigt werden. Das Kantengewicht einer eingefügten Kante ergibt sich dann aus der Differenz der jeweiligen konstanten Adressoffsets, so dass prinzipiell eine Adressierung mit Auto-Modify- oder Auto-Inkrement-Befehlen möglich ist.

(4) `AddressCodeGeneration(`*mem_seq*`)`
Nach der Erzeugung des Graphen wird, mittels des im Anschluss an diesen Pseudo-Algorithmus beschriebenen Verfahrens, für jeden Knoten der Adresscode bestimmt, der zur Adressierung des entsprechenden Speicherzugriffs erforderlich ist und in die GeLIR-Datenstrukturen eingefügt. Um den zu diesem Zeitpunkt noch vorhandenen ursprünglichen Adresscode zu eliminieren, wird abschließend eine Dead-Code-Elimination durchgeführt.

(5) `RedundantAddressCodeElimination(`*fun*`)`
Da die Adresscode-Generierung für jeden Basisblock separat durchgeführt wird, kann es vorkommen, dass in einen bestimmten Basisblock Anweisungen eingefügt werden, die bei einer mehr globaleren Betrachtung des Codes nicht erforderlich gewesen wären. So kann es z.B. sein, dass in jedem Basisblock das *PP*-Register immer mit derselben Seitenadresse initialisiert wird, obwohl in allen Vorgänger-Basisblöcken dieser Wert bereits gesetzt und seitdem nicht mehr verändert wurde. Wird in einem bestimmten Basisblock solch eine unnötige Initialisierung festgestellt, wird diese Anweisung ersatzlos aus dem Code gelöscht.

Zur Generierung des erforderlichen Adresscodes erfolgt eine Bearbeitung der Graphknoten entsprechend der Position in der Speicherzugriffs-Sequenz, also mit aufsteigenden Ordnungsnummern. Für jeden Graphknoten werden dabei die nachfolgend beschriebenen Schritte durchgeführt. Zur Einhaltung von Ressource-Constraints wird in jedem der beschriebenen Schritte bei einer Auswahl eines Adressregisters überprüft, ob dieses in der Menge der zur Verfügung stehenden Adressregister eines Knotens enthalten ist.

1.) Adressierung mittels Page-Pointer-Register.
Diese Adressierung ist nur dann möglich, wenn die Adresse, auf die zugegriffen werden soll, bereits zur Compilierungszeit bekannt ist, also keine variablen Adressoffsets

vorhanden sind. Bei Bedarf wird für eine Seitenadressierung das *PP*-Register mit der Adresse einer neuen Seite geladen, so dass eine Adressierung möglich wird.

Bei Erfolg: Stop

2.) Addressierung mittels Adressregister AR_0, AR_1, AR_2 oder AR_3.

a) Wähle ein Adressregister, das von einem über Kanten erreichbaren Vorgänger verwendet wurde und in der Zwischenzeit von keinem anderen Knoten wiederverwendet wurde. Vermerke beim Vorgängerknoten den Offset zum aktuellen Knoten, um bei der Durchführung des Speicherzugriffs des Vorgängers bereits die Adresse des aktuellen Knotens zu berechnen.

Bei Erfolg: Stop

b) Adressiere den aktuellen Knoten unabhängig von zuvor adressierten Knoten. Versuche ein aktuell nicht in Gebrauch befindliches Adressregister auszuwählen. Falls alle Adressregister bereits verwendet werden, d.h. potentiell noch Offsets zu späteren Knoten ausgenutzt werden können, wähle das Adressregister aus, das zum spätest möglichen Zeitpunkt wiederverwendet wird.

Da der eingefügte Adresscode nur für die gegebene Speicherzugriffs-Sequenz zu gültigem Assemblercode führt, muss bei der nachfolgend durchzuführenden Adresscode-Kompaktierung dafür gesorgt werden, dass die Reihenfolge der Speicherzugriffe erhalten bleibt. Aus diesem Grund werden zwischen je zwei aufeinander folgenden Speicherzugriffen Sequentialisierungskanten eingefügt, die eine entsprechende Ausführungsreihenfolge der Speicherzugriffe erzwingen. Diese Randbedingungen müssen bei der nachfolgend durchzuführenden Phase der Adresscode-Kompaktierung (s. Abschnitt 3.7) berücksichtigt werden.

3.6.2 Phasenkopplung mit Codegenerierung

Da die Durchführung der Codegenerierung bereits eine sehr komplexe Aufgabe darstellt, wird für Architekturen mit AGUs die Aufgabe der Adresscode-Generierung in einem separaten Optimierungsschritt durchgeführt. Allerdings besteht auch zwischen diesen Optimierungsphasen ein Phasenkopplungsproblem, da die Ergebnisse der Adresscode-Generierung zum einen von der Anzahl der Speicherzugriffe und zum anderen von der Ausführungsreihenfolge dieser Speicherzugriffe abhängen. So ist es möglich, dass es mehrere gleich gute Ergebnisse hinsichtlich der Codegenerierung gibt, allerdings mit unterschiedlichen Speicherzugriffs-Sequenzen. Es gilt also das Problem zu lösen, bereits bei der Durchführung der Codegenerierung die Auswirkungen der Adresscode-Generierung zu berücksichtigen.

Als Lösungsansatz für dieses Phasenkopplungsproblem kommen uns wieder die besonderen Eigenschaften des in Abschnitt 3.5 vorgestellten genetischen Codegenerators zugute. So ist es naheliegend, durch eine Erweiterung der Bewertungsfunktion der Individuen dieses Phasenkopplungsproblem zu berücksichtigen. Im Prinzip ergeben sich bei der Bewertung eines Individuums die folgenden beiden Möglichkeiten:

- Durchführung der Adresscode-Generierung und/oder der Kompaktierung für jedes Individuum. Dies würde eine exakte Bewertung der Individuen erlauben, allerdings insgesamt auch einen beträchtlichen Mehrbedarf an Rechenzeit erfordern. Es sind also Abschätzungsverfahren wünschenswert, die aufgrund von einfach ermittelbarer Kriterien eine grundsätzliche Differenzierung unterschiedlicher Codesequenzen zulassen.
- Berücksichtigung bereits vorhandener Informationen bzw. schnell ermittelbarer Informationen:
  - Vermeidung von MOs, für die potentiell Adresscode eingefügt werden muss. Dies können z.B. Speicherzugriffe sein oder MOs, die spezielle Hardware-Ressourcen wie z.B. *index_read* oder *index_write* (s. auch Seite 100 in Abschnitt 4.1.2) benutzen, für deren Verwendung häufig separater Adresscode eingefügt werden muss.
  - Vermeidung des Neuladens des *PP*-Registers aufgrund von aufeinander folgenden Zugriffen auf unterschiedliche Speicherseiten. Da ein Neuladen des *PP*-Registers in jedem Fall mit einem nicht parallelisierbaren Move-Befehl durchgeführt werden muss, können auch ohne explizite Durchführung der Adresscode-Kompaktierung die hierdurch entstehenden Kosten sehr genau abgeschätzt werden.

Aus Effizienzgründen wird im Codegenerator die zweite Variante realisiert.

3.7 Adresscode-Kompaktierung

Das Ziel dieses Schrittes liegt in der Kompaktierung des zuvor eingefügten Adresscodes. Da bereits eine Kompaktierung des restlichen Codes (des Datenpfades) vorliegt, besteht eine Möglichkeit darin, den zusätzlich eingefügten Adresscode ohne Änderung der bereits kompaktierten Befehle auf die bereits vorhandenen MIs aufzuteilen. Allerdings ist zu erwarten, dass das volle Optimierungspotential nur mit einer Neuanordnung der MOs zu MIs ausgeschöpft werden kann. Da diese Aufgabe im Wesentlichen der Aufgabe der bereits durchgeführten Codegenerierung entspricht, wird dazu der in Abschnitt 3.5 beschriebene

genetische Codegenerator GCG wiederverwendet. Als Unterschied zur Codegenerierung ist hier zu sehen, dass nun keine Auswahlmöglichkeiten der Ressourcen mehr vorhanden sind, da jede Menge bereits auf genau ein Element eingeschränkt wurde. Aufgrund des eingefügten Adresscodes müssen weitere Randbedingungen in Form von Sequentialisierungskanten beachtet werden.

3.8 Bewertung

Das zuvor in diesem Kapitel beschriebene Codegenerierungs-Verfahren auf Basis eines genetischen Optimierungsverfahrens ist unter Verwendung der Bibliothek *PGAPack* [Lev96], mit der die Entwicklung von genetischen Algorithmen unterstützt wird, umgesetzt worden. In diesem Abschnitt wird eine Bewertung der Qualität des Codegenerators anhand einiger ausgewählter Testroutinen für den speziellen Einstreifen-Modus (SISD-Modus) des M3-DSPs vorgenommen. Erweiterungen zur effektiven Ausnutzung von SIMD-Operationen und des Gruppenspeichers sind Bestandteil von Kapitel 4 und werden dort näher untersucht.

Im nachfolgenden Abschnitt wird zunächst auf die Einstellung der internen Parameter des GAs eingegangen. Dies umfasst insbesondere einen Vergleich der Qualität einiger Crossover-Operatoren. Danach erfolgt eine Bewertung des genetischen Codegenerators. Abschließend wird der Einfluss der Adresscode-Generierung auf die Codequalität betrachtet.

3.8.1 Einstellung der Parameter des genetischen Algorithmus

Das Konvergenzverhalten und die Qualität der Ergebnisse eines GAs hängen stark von der Wahl der internen Steuerungsparameter wie z.B. der Populationsgröße oder der Mutations-Wahrscheinlichkeit ab. Da eine vollständige Exploration der Belegung dieser Parameter mit Werten extrem zeitaufwändig ist, wurde anhand einer kleineren Auswahl von Programmen eine Einstellung dieser Parameter vorgenommen. Als gute Werte ergaben sich dabei die folgenden Parameter:

- Populationsgröße: 40
- Anzahl der in jeder Generation zu ersetzenden Individuen: 4
- Mutations-Wahrscheinlichkeit: 1/(Anzahl der Gene pro Individuum)
- Crossover-Wahrscheinlichkeit: 0,6
- Anzahl durchzuführender Generationen: (zwei- bis vier)-fache Anzahl von Genen

Mit diesen Parametern wurde das Konvergenzhalten des GAs bei Verwendung der Crossover-Operatoren `1-Punkt`, `2-Punkt`, `Uniform`[3], `1-Punkt-CS` und `2-Punkt-CS` untersucht. Die Anzahl der durchzuführenden Generationen wurde hier auf das 4-fache der Anzahl von Genen gesetzt. In den Abbildungen 3.16 und 3.17 ist der Optimierungsfortschritt bei Durchführung der Aufgaben CS, IA und RA für zwei selbstgeschriebene Testroutinen `dfg1` bzw. `dfg2` dargestellt, die jeweils durch einen Datenflussgraphen repräsentiert werden können (s. auch Seite 164 ff. in Anhang A). Als Maßstab dient die Anzahl der Maschineninstruktionen zu einem bestimmten Optimierungszeitpunkt (bzw. einer Generation).

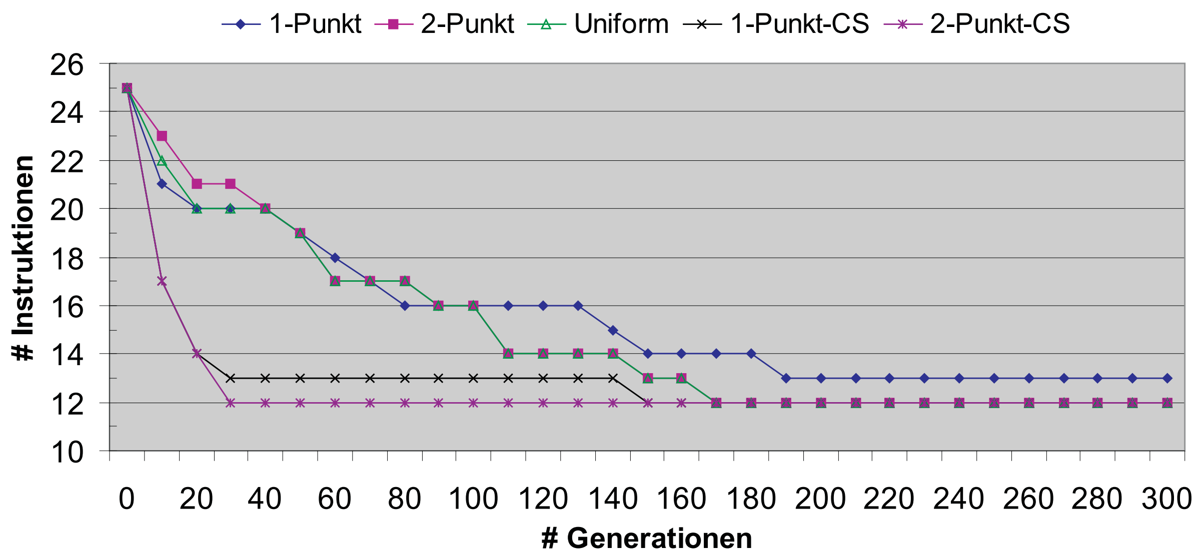


Abb. 3.16: Einfluss der Crossover-Operatoren auf die Konvergenz: `dfg1`-Routine

Bei beiden Testroutinen zeigt sich deutlich die Überlegenheit der speziellen Crossover-Varianten `1-Punkt-CS` und `2-Punkt-CS` gegenüber den Standard-Varianten. Bei beiden Testroutinen werden bei Verwendung der CS-Varianten bereits in der Anfangsphase der Optimierung sehr gute Lösungen gefunden. Des Weiteren führt bei beiden Testroutinen lediglich die Verwendung der CS-Varianten und des 2-Punkt-Crossovers zu den besten Ergebnissen. Bei Betrachtung der Ergebnisse für die `dfg2`-Routine fällt auf, dass das Uniform-Crossover im Vergleich zu den anderen Varianten deutlich schlechter abschneidet. Dies lässt sich damit begründen, dass beim Uniform-Crossover ohne Berücksichtigung der vorhandenen Ausführungsreihenfolge einzelne Gene der Individuen ausgetauscht werden, so dass die entstehenden Lösungen in der nachfolgenden Mutationsphase verstärkt korrigiert (bzw. mutiert) werden müssen.

[3]Ein Austausch der Allele erfolgte mit einer Wahrscheinlichkeit von 60%.

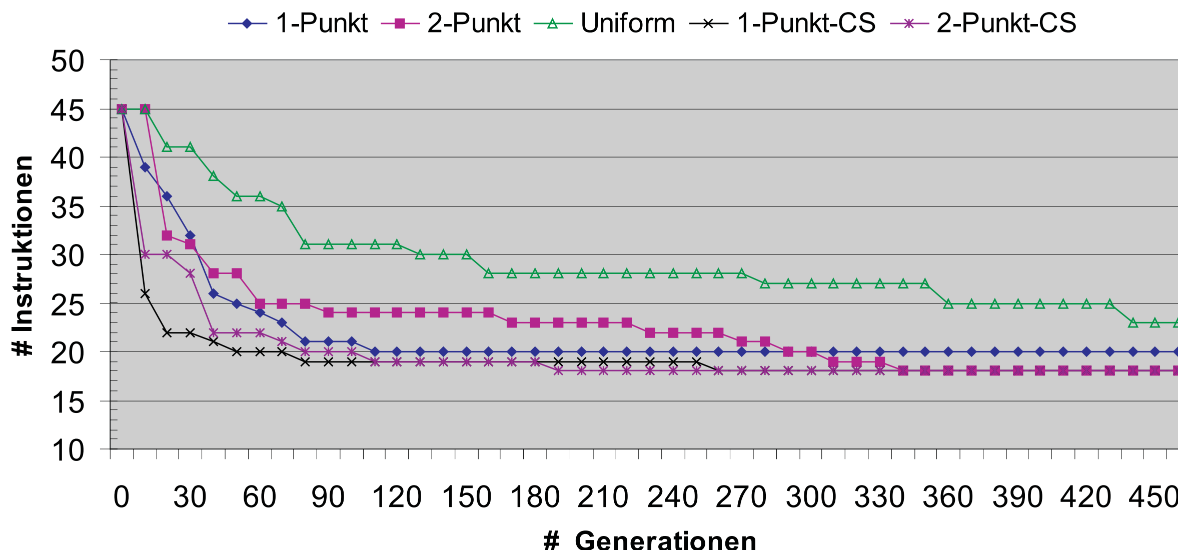


Abb. 3.17: Einfluss der Crossover-Operatoren auf die Konvergenz: `dfg2`-Routine

3.8.2 Genetischer Codegenerator

Um die Qualität des genetischen Codegenerierungs-Verfahrens zu beurteilen, beschränken wir uns in diesem Abschnitt zunächst auf die Verwendung von Testroutinen, die aus einem Basisblock bestehen, also jeweils durch einen Datenflussgraphen dargestellt werden können. Diese Routinen können z.B. Anweisungen der innersten Schleife eines Programms darstellen, so dass die Erzeugung von gutem Code für diesen Basisblock sich wesentlich auf die Codequalität des gesamten Programms auswirken würde.

In Tabelle 3.1 sind für die in diesem Abschnitt verwendeten Testroutinen einige charakteristische Merkmale aufgeführt. Die Routinen umfassen `cmultiply`, `cupdate` und `biquad` aus der DSPstone-Benchmarksuite [ZVSM94], einen Lattice-Filter (`lattice`) und die bereits im vorherigen Abschnitt betrachteten selbstgeschriebenen Routinen `dfg1` und `dfg2`.

Benchmark	#CSEs	#CSE-Verwend.	#Graphknoten			#Generationen		Laufzeit [s]
			IR	vor CG	vor ACK	CG	ACK	
cmultiply	4	8	21	57	34	171	102	22
cupdate	4	8	27	73	48	219	144	35
biquad	3	7	29	81	53	243	109	46
lattice	8	16	29	95	66	285	198	74
dfg1	4	9	23	75	43	225	129	41
dfg2	6	17	29	117	57	351	171	111

Tabelle 3.1: Charakteristische Merkmale der Testroutinen

In Spalte 2 wird für die Routinen die Anzahl der CSEs und in Spalte 3 die Anzahl

deren Verwendungen angegeben. Die Spalten 4 bis 6 geben Auskunft über die Anzahl der Graphknoten der initialen GeLIR-Darstellung (IR), vor Durchführung der Codegenerierung mit den Teilaufgaben CS, IA und RA (vor CG) und vor Durchführung der Adresscode-Kompaktierung (vor ACK). Es ist zu erkennen, dass sich die Anzahl der Graphknoten der initialen GeLIR-Darstellung durch das Einfügen der potentiell möglichen Datentransfers ungefähr um den Faktor 3 erhöht. Die Anzahl der Graphknoten ist vor der Durchführung der Adresscode-Kompaktierung geringer als vor der Codegenerierung, da viele Graphknoten aufgrund nicht erforderlicher Datentransferwege im Zuge der Codegenerierung eingespart werden konnten. Die Anzahl durchzuführender Generationen für die Codegenerierung (CG) und die Adresscode-Kompaktierung (ACK) ist in den Spalten 7 bzw. 8 angegeben und wurde für diese Routinen auf die dreifache Anzahl von Genen (bzw. Graphknoten) gesetzt. In der letzten Spalte werden die zur Compilierung und Simulation der jeweiligen Routinen erforderlichen Laufzeiten in Sekunden angegeben, wobei die Laufzeiten für das Front-End und die Simulation für diese Beispiele vernachlässigbar sind[4]. Hier ist auch zu beachten, dass in den meisten Fällen die beste Lösung bereits zu einem sehr frühen Optimierungszeitpunkt gefunden wurde. Es hat sich auch gezeigt (nicht in der Tabelle dargestellt), dass eine Bewertung von Individuen mit Hilfe unseres Energiekostenmodells zu keiner nennenswerten Erhöhung der Laufzeit führt.

Zur Beurteilung der Qualität des entwickelten genetischen Codegenerators werden die folgenden Codegenerierungs-Varianten betrachtet:

- `baum`
  Durchführung einer baumbasierten Codeselektion unter Entkopplung der Phasen zum Einfügen von Spillcode und der Codekompaktierung, wie es in herkömmlichen Compilern der Fall ist. Das alleinige Optimierungsziel ist die Minimierung der Ausführungszeit, es wird also keine explizite Energieoptimierung vorgenommen.

- `baum+phasen`
  Analoge Vorgehensweise wie bei `baum`, allerdings mit einer vollständigen Phasenkopplung.

- `graph+phasen`
  Analoge Vorgehensweise wie bei `baum+phasen`, allerdings wird statt einer baumbasierten eine graphbasierte Codeselektion durchgeführt.

- `graph+phasen+MaxEnergie`
  Analoge Vorgehensweise wie bei `graph+phasen`, allerdings wird für die Lösung mit der geringsten Ausführungzeit der Energieverbrauch maximiert, um das Optimierungspotential aufzuzeigen.

[4] Die angegebenen Laufzeiten beziehen sich auf einen AMD Athlon Prozessor mit einer Taktfrequenz von 1,34 GHz.

- `graph+phasen+MinEnergie`
  Analoge Vorgehensweise wie bei `graph+phasen`, allerdings wird als alleiniges Optimierungskriterium der Energieverbrauch minimiert.

In den Abbildungen 3.18 bis 3.20 werden die Ergebnisse der Codegenerierungs-Varianten hinsichtlich der erforderlichen Ausführungszeit, des Energieverbrauchs und der benötigten Speicherzugriffe gegenübergestellt. Alle Ergebnisse werden in Relation zu handgeneriertem Code (≙ 100%) gesetzt, der zum Vergleich einschließlich der C-Routinen in Anhang A aufgeführt ist. Da für den M3-DSP kein Referenz-Compiler verfügbar ist, muss auf einen Vergleich mit anderen Compilern leider verzichtet werden.

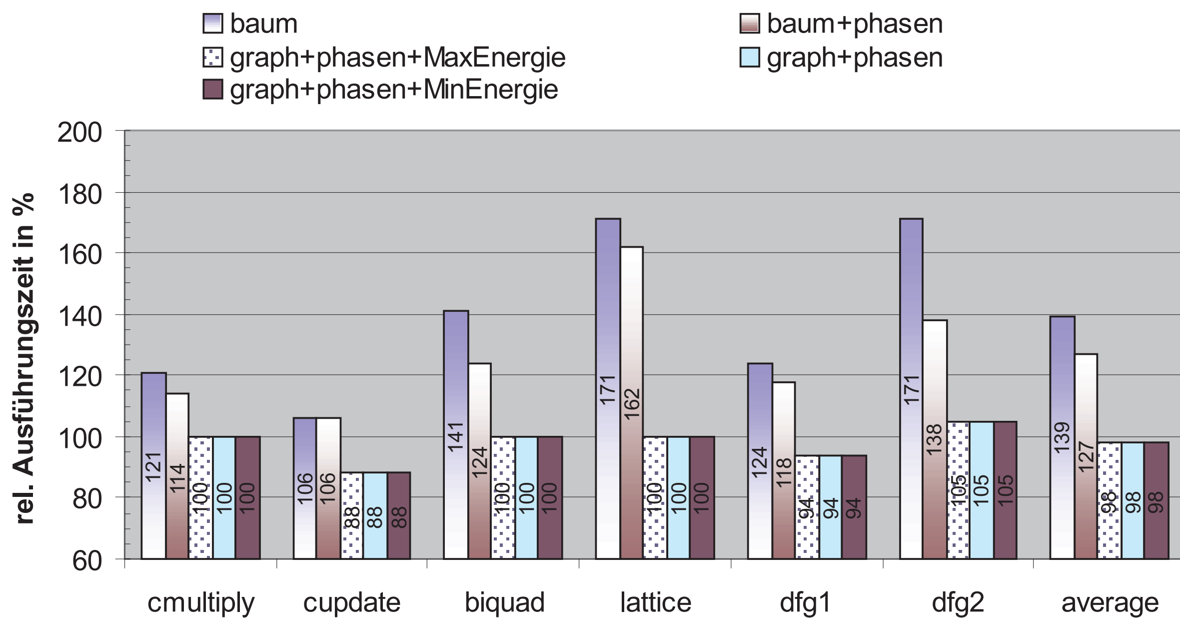


Abb. 3.18: Codegenerator Ergebnisse: Ausführungszeit (100% ≙ handgeneriertem Code)

In Abb. 3.18 zeigt sich, dass die unter Verwendung einer baumbasierten Codeselektion (`baum` und `baum+phasen`) generierten Programme bei allen Routinen zu den höchsten Ausführungszeiten führen. So beträgt der Overhead des reinen baumbasierten Verfahrens im Vergleich zum handgenerierten Code im Durchschnitt 39%. Es zeigt sich, dass eine integrierte Phasenkopplung diesen Overhead bereits deutlich auf 27% im Schnitt reduzieren kann. Bei der Betrachtung der Ausführungszeiten der graphbasierten Varianten ist erkennbar, dass diese an die Codequalität von handgeneriertem Code herankommen und diese im Fall der Routinen `cupdate` und `dfg1` sogar noch verbessern können.

Bei Betrachtung der Ergebnisse hinsichtlich des Energieverbrauchs in Abb. 3.19 ist ebenfalls ein wesentlich schlechteres Abschneiden der baumbasierten Verfahren gegenüber den anderen Verfahren zu erkennen. So weisen die beiden baumbasierten Verfahren im Vergleich zum handgenerierten Code einen Overhead von durchschnittlich 52% bzw. 44% auf.

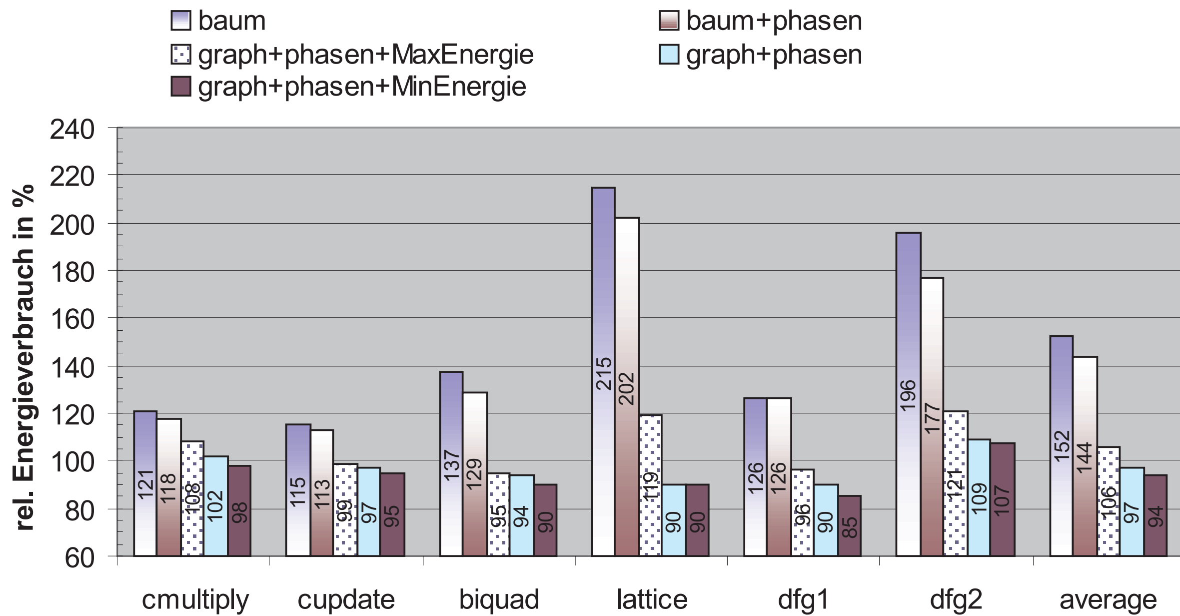


Abb. 3.19: Codegenerator Ergebnisse: Energieverbrauch (100% $\widehat{=}$ handgeneriertem Code)

Wie zu erwarten, liefert die Verwendung des Verfahrens `graph+phasen+MaxEnergie` den höchsten Energieverbrauch der Lösung mit der geringsten Ausführungszeit. Im Vergleich zu `graph+phasen+MinEnergie` wird hier sehr schön der Optimierungsspielraum deutlich, der im Durchschnitt 12 Prozentpunkte und für die `lattice`-Routine sogar 29 Prozentpunkte beträgt. Die Ergebnisse von `graph+phasen` liegen alle innerhalb dieses Bereichs. Interessant ist, dass sich durch die Anwendung der vom Compiler durchgeführten Energieoptimierung zum Teil deutliche Einsparungen (z.B. 15% für `dfg1`) im Vergleich zum handgenerierten Assemblercode ergeben. Dies zeigt sich auch am Beispiel der `biquad`-Routine, bei der trotz gleicher Ausführungszeit, der Energiebedarf des vom Compiler generierten Codes um 10% geringer ist.

In Abb. 3.20 zeigt sich, inwiefern sich die Verwendung der unterschiedlichen Verfahren auf die Anzahl der Speicherzugriffe auswirkt. Auffällig ist hier die extrem hohe Anzahl von Speicherzugriffen bei Anwendung einer baumbasierten Codeselektion, die als Hauptursache für die schlechte Codequalität dieser Verfahren angesehen werden kann. Dies trifft insbesondere für die Routinen `lattice` und `dfg2` zu, die eine vergleichsweise hohe Anzahl von CSEs und CSE-Verwendungen aufweisen. So sind für diese Routinen mehr als dreimal soviele Speicherzugriffe erforderlich als beim handgenerierten Code und den Varianten `graph+phasen` und `graph+phasen+MinEnergie`.

Fazit

Durch einen Vergleich der vom genetischen Codegenerator erzielten Codequalität mit handgeneriertem Assemblercode konnte die Effektivität dieses neuen Codegenerierungs-

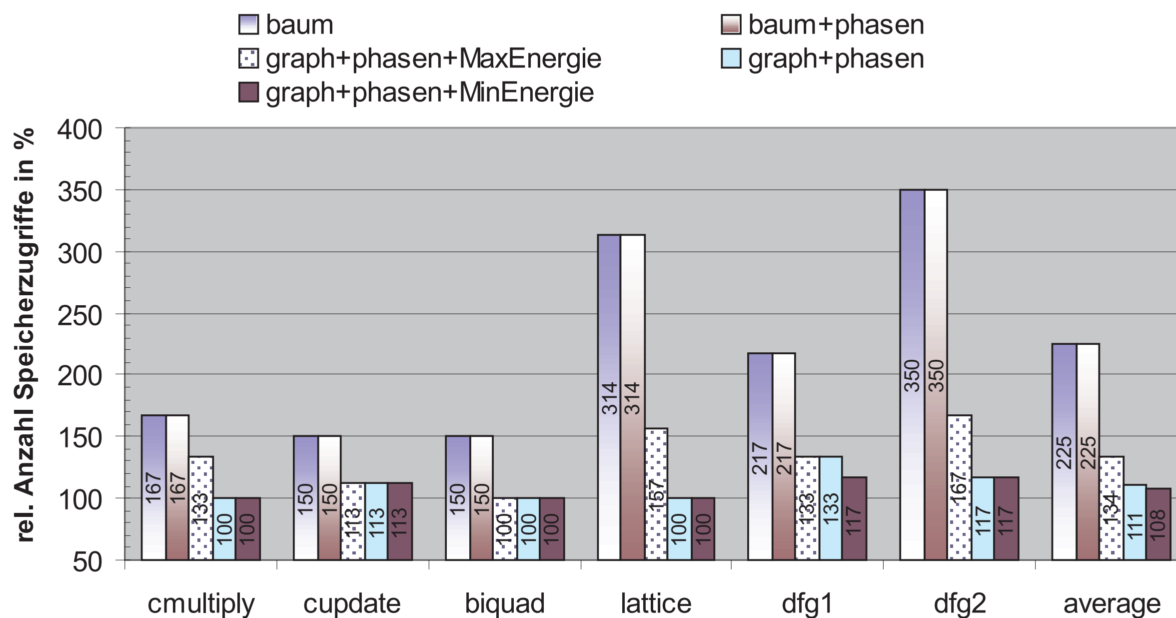


Abb. 3.20: Codegenerator Ergebnisse: Speicherzugriffe (100% $\widehat{=}$ handgeneriertem Code)

Verfahrens demonstriert werden. Als Hauptgründe für die gute Codequalität sind hier die Realisierung einer vollständigen Kopplung der Codegenerierungs-Phasen und die Durchführung einer graphbasierten Codeselektion anzusehen. Eine weitere Reduzierung des Energieverbrauchs konnte durch die zusätzliche Integration einer energieeffizienten Auswahl und Anordnung von Maschinenoperationen zu Maschineninstruktionen in den genetischen Codegenerator erzielt werden.

3.8.3 Adresscode-Generierung

Nachdem im vorherigen Abschnitt die Qualität des genetischen Codegenerators untersucht worden ist, soll nun der Einfluss der Adresscode-Generierung auf die Codequalität näher beleuchtet werden. In den Abbildungen 3.21 und 3.22 erfolgt dazu ein Vergleich der Ergebnisse hinsichtlich der Ausführungszeit und des Energieverbrauchs für die DSP-Routinen `n_real_update`, `n_real_update1x`, `chsign` und `antialias`. Die letzten beiden Routinen stellen dabei Sub-Routinen einer MP3-Applikation und `n_real_update1x` eine modifizierte Version der `n_real_updates`-Routine dar, bei der die innerste Schleife einmal abgerollt wurde. Da zur Durchführung der Adresscode-Generierung eine feste Speicherzugriffs-Reihenfolge gegeben sein muss und sich deswegen die Anzahl der Speicherzugriffe nicht mehr verändert, sind keine Ergebnisse bezüglich der Anzahl der Speicherzugriffe aufgeführt.

Mit `AGU` werden im Folgenden die Ergebnisse bezeichnet, bei denen die Adressberechnungen zwar auf der AGU ausgeführt worden sind, allerdings für jeden Speicherzugriff un-

abhängig. Dahingegen wird bei `AGUopt` die aktuell zu ermittelnde Adresse auf der Basis vorheriger Adressen, mit Hilfe von Auto-Modify- und Auto-Inkrement-Befehlen, berechnet. Bei `AGUopt+RACE` erfolgt zusätzlich eine globale (Basisblock-übergreifende) Eliminierung redundanter Setzungen des *PP*-Registers. Der Zusatz `MinEnergie` gibt wiederum an, dass eine Optimierung hinsichtlich des Energieverbrauchs vorgenommen wurde. Die generierten Ergebnisse werden in Relation zu den Resultaten bei Berechnung der Adressen im Datenpfad ($\widehat{=}$ 100%) gesetzt.

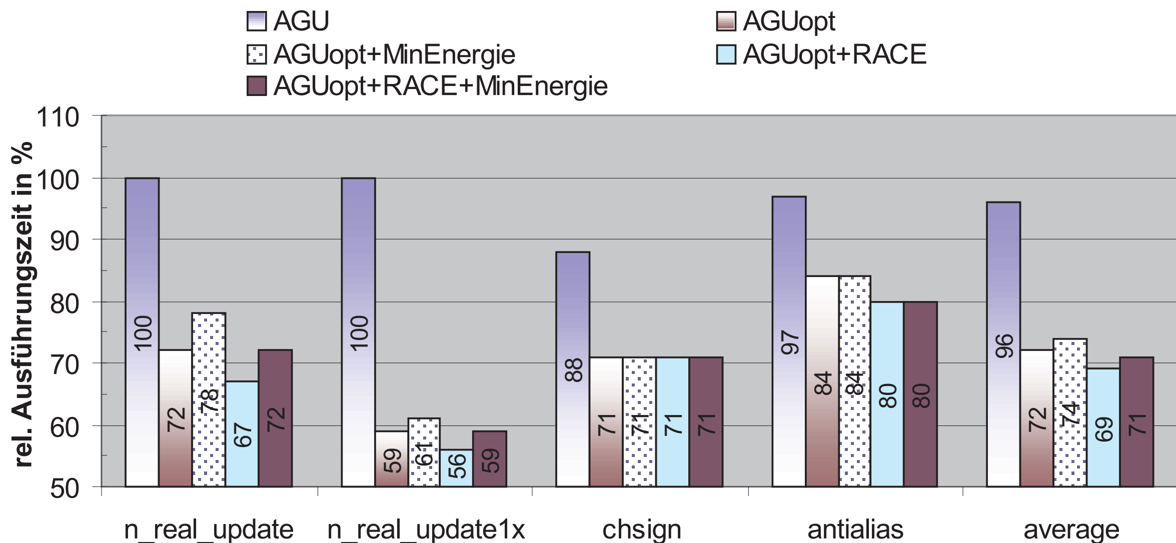


Abb. 3.21: Adresscode-Generierung Ergebnisse: Ausführungszeit (100% $\widehat{=}$ Berechnung der Adressen im Datenpfad).

Es zeigt sich, dass bereits mit der Einbeziehung der AGU eine geringe Verbesserung der Codequalität um durchschnittlich 4% hinsichtlich der Ausführungszeit (s. Abb. 3.21) und durchschnittlich 6% hinsichtlich des Energieverbrauchs (s. Abb. 3.22) für diese Routinen zu erzielen ist. Das lässt sich dadurch begründen, dass durch die Ausführung von Adressberechnungen auf der AGU anstatt im Datenpfad die durchzuführenden Berechnungen auf mehr parallel ansteuerbare Funktionseinheiten aufgeteilt werden. Durch die Ausnutzung der speziellen Adressgenerierungsbefehle kann die Codequalität hinsichtlich beider Optimierungskriterien um durchschnittlich ca. 30% wesentlich verringert werden. Die zusätzliche Durchführung einer Energieminimierung zeigt, dass nicht immer der schnellste Code auch der energieeffizienteste sein muss. So liegt die Ausführungszeit der `n_real_update`-Routine für `AGUopt+MinEnergie` zwar 8% ($\widehat{=}$ 6 Prozentpunkten) über der für `AGUopt`, führt allerdings zu einer Energieeinsparung von ca. 10% ($\widehat{=}$ 8 Prozentpunkten).

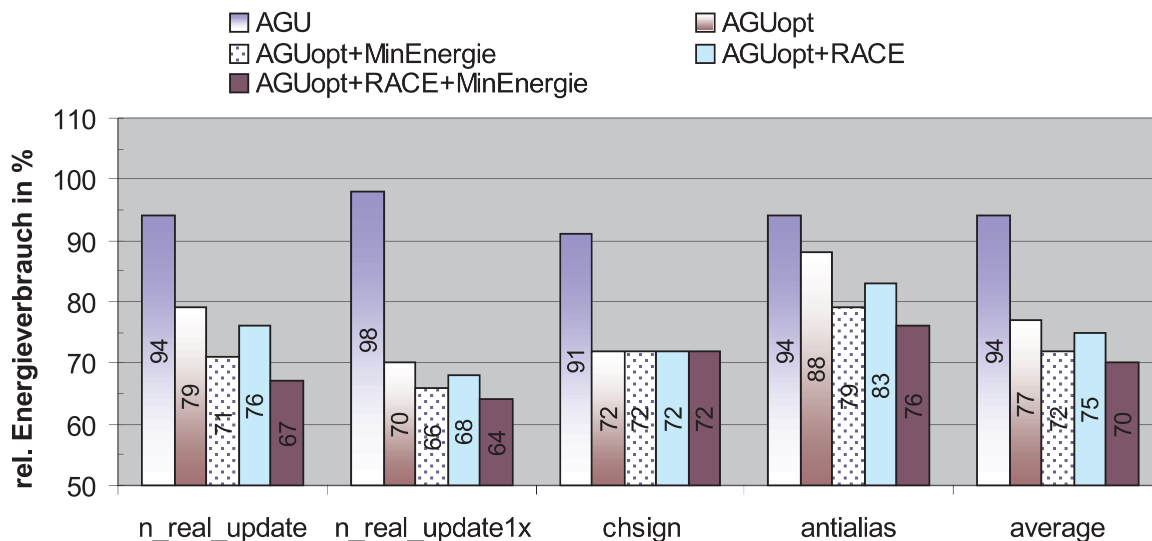


Abb. 3.22: Adresscode-Generierung Ergebnisse: Energieverbrauch (100% $\widehat{=}$ Berechnung der Adressen im Datenpfad).

3.8.4 Retargierbarkeit

Zur Demonstration der Retargierbarkeit des Codegenerators wurde neben dem Back-End für den M3-DSP auch ein Back-End für den ADSP2100 der ADSP210X-Familie [Dev91] entwickelt. Der betrachtete DSP-Prozessor enthält im Datenpfad drei Funktionseinheiten ALU, MAC und Shifter mit dedizierten Registerfiles. Des Weiteren ist der Speicher in zwei getrennte Speicherbänke partitioniert, auf die mittels zweier separater AGUs parallel zugegriffen werden kann. Eine parallele Ausführung von Datentransfers, Speicherzugriffen und Datenmanipulationen ist nur in eingeschränkter Art und Weise möglich.

Die Entwicklung des neuen Back-Ends wurde auf der Basis des für den M3-DSP bereits vorhandenen Codegenerators durchgeführt. Der Hauptaufwand entstand dabei in der Beschreibung der neuen Zielarchitektur. Aufgrund der generischen Implementierung des Codegenerators waren nur geringfügige Anpassungen des restlichen Codes erforderlich. Im Wesentlichen betrifft dies die Generierung alternativer Maschinenprogramme durch Spezifizierung der Anzahl einzufügender potentieller Datentransfers zwischen zwei Graphknoten und die Erzeugung einer initialen Überdeckung der Graphknoten mit Ressourcen. Da der ADSP2100 keine Seiten-indirekte Adressierung, wie beim M3-DSP mittels des *PP*-Registers möglich, gestattet, wurde bei der Adresscode-Generierung das *PP*-Register nicht in die Menge der zur Verfügung stehenden Adressregister eingefügt. Spezielle Funktionalität, insbesondere zur Handhabung des Gruppenspeichers des M3-DSPs, konnte durch Flags ausgeschaltet werden. Auf diese Weise war es innerhalb von zwei Tagen möglich, einen Codegenerator einschließlich eines Simulators für den ADSP zu generieren, der im Wesentlichen die gleiche Funktionalität besitzt, wie der hier beschriebene Codegenerator

für den M3-DSP. Es muss allerdings angemerkt werden, dass keine spezielle Optimierung zur Ausnutzung der beiden Speicherbänke implementiert worden ist, da es lediglich darum ging, die Retargierbarkeit zu untersuchen. Entsprechende Optimierungen können allerdings sehr modular in Analogie zu den in Kapitel 4 beschriebenen Erweiterungen für den M3-DSP vorgenommen werden. Eine Handhabung der in [HD98] beschriebenen VLIW-Architekturen, anhand derer der allgemein als retargierbar eingestufte Compiler AVIV getestet wurde, ist ebenfalls problemlos möglich.

Kapitel 4

SIMD-Optimierungen

Zur Erzielung einer möglichst hohen Ausführungsgeschwindigkeit von Anwendungen unterstützen digitale Signalprozessoren üblicherweise eine parallele Ausführung von Operationen auf Instruktionsebene (*feinkörnige Parallelität*). So können bei den Prozessoren der M3-Plattform u.a. eine Datenmanipulation, ein Datentransfer, ein Speicherzugriff und eine Schiebe-Operation in einem Prozessorzyklus simultan ausgeführt werden. Verglichen mit einer rein sequentiellen Ausführung eines Programms kann dies, bei entsprechender Unterstützung durch den Compiler, bereits zu einer drastischen Reduzierung der Ausführungszeit führen. Allerdings werden z.B. durch Kontrollflussverzweigungen und Datenabhängigkeiten in Programmen die Möglichkeiten zur Parallelisierung erheblich eingeschränkt, so dass das vorhandene Potential i.d.R. nur zu einem wesentlich geringeren Anteil ausgenutzt werden kann. Eine weitere Möglichkeit zur Erhöhung der Ausführungsgeschwindigkeit besteht in der Ausführung von SIMD-Operationen (*Vektorisierung*), wie sie auch von den M3-Prozessoren zur Verfügung gestellt werden. Das Besondere ist, dass mit einer Anweisung mehrere unterschiedliche Daten parallel verarbeitet werden. Im Falle des M3-DSPs ergibt sich so, in Verbindung mit der Möglichkeit der parallelen Ausnutzung der Funktionseinheiten, im Vergleich zu einer rein sequentiellen Ausführung weitreichendes Potential zur Erhöhung der Ausführungsgeschwindigkeit von Programmen. Da beim M3-DSP mit einer SIMD-Operation 16 Datenpfade parallel betrieben werden können und dabei im Vergleich zu einer SISD-Operation lediglich das vier- bis fünffache an Energie erforderlich ist, erscheint eine effektive Ausnutzung von SIMD-Operationen nicht nur hinsichtlich der Ausführungszeit sinnvoll, sondern ebenfalls unter Energie-Gesichtspunkten. Allerdings besteht das Problem, dass für eine entsprechende Compiler-Unterstützung zur Ausnutzung von SIMD-Funktionalität aufwändige Optimierungen und Analysen erforderlich sind.

Im folgenden Abschnitt wird zunächst eine kurze Einführung in die Problematik gegeben. Nach einer Übersicht über bestehende Verfahren in diesem Bereich werden Erweiterungen bezüglich der Architektur- und Programmdarstellung vorgestellt, die im Zusammenhang

mit den hier vorgestellten SIMD-Optimierungen erforderlich werden. Anschließend folgt eine Beschreibung der Optimierung zur effektiven Ausnutzung der parallelen Datenpfade und zur effektiven Ausnutzung der SIMD-Speicherzugriffe durch eine optimierte Anordnung von skalaren Variablen. Das Kapitel endet mit einer Bewertung dieser Optimierungen anhand einiger Testroutinen.

4.1 Einführung

Da eine Übersetzung von Programmen in effizienten Assemblercode allgemein eine äußerst komplexe Aufgabe darstellt, können vom Anwender in der Programmiersprache C beispielsweise Erweiterungen in Form von *Pragmas*, *Intrinsics* und *Inline-Assemblercode* vorgenommen werden: Pragmas sind Teil des ANSI C-Standards (s. z.B. [KR88]) und stellen direkt in das Quellprogramm eingefügte Anweisungen dar, die den Compiler bei der Übersetzung bestimmter Programmfragmente unterstützen sollen. Als einzige Anforderung wird vom C-Standard gefordert, dass diese keinen Einfluss auf das Ergebnis einer C-Anweisung haben dürfen. Im Prinzip bleibt es dem jeweiligen Compiler überlassen, ob dieser die zusätzlichen Informationen nutzt oder unberücksichtigt lässt. In unserem Fall könnten dies Anweisungen zur Umsetzung bestimmter Ausdrücke mit SIMD-Operationen sein, die sonst nicht vom Compiler als vektorisierbar erkannt werden.

Assembler-Intrinsics gestatten spezielle Code-Erweiterungen in C-Syntax, ohne dass sich der Benutzer mit der Assemblersprache auseinander setzen muss. Der Nachteil ist, dass für derart in den C-Code eingefügte „Funktionsaufrufe“, für jede Anweisung unabhängig, entsprechende Assembler-Anweisungen eingefügt werden. Ein direktes Einfügen von Assembler-Anweisungen (Inline-Assemblercode) kann auch vom Anwender direkt im Quellcode vorgenommen werden. Allerdings ergeben sich durch diese Vorgehensweisen eine Reihe von Nachteilen: So benötigt der Entwickler zur Programmierung ein tieferes Verständnis der zugrunde gelegten Zielarchitektur und des verwendeten Compilers, was die Programm-Entwicklung fehleranfälliger und zeitintensiver gegenüber einer automatisierten Compilierung macht. Des Weiteren ist der Quellcode bei der Verwendung von Assembler-Intrinsics und Inline-Assemblercode aufgrund von verwendeten compilerspezifischen Bibliotheksaufrufen nicht mehr auf andere Zielarchitekturen portierbar. Wünschenswert sind also Compiler, die in der Lage sind, SIMD-Operationen automatisch zu erkennen und effektiv auszunutzen.

Nachfolgend wird zunächst auf die Probleme einer automatisierten Ausnutzung von SIMD-Operationen durch Compiler eingegangen. Dazu werden in den folgenden Abschnitten zunächst allgemeine und danach die durch die besonderen Architektureigenschaften der M3-Prozessoren verursachten Problembereiche erläutert. Abschließend erfolgt eine Beschreibung der im Compiler erforderlichen Erweiterungen zur Ausnutzung der SIMD-

Funktionalität.

4.1.1 Allgemeine Problembereiche

Bei der Entwicklung von Optimierungen zur Ausnutzung von SIMD-Operationen stellt sich zunächst die Frage nach dem Zeitpunkt der Ausführung. Eine Integration in den Compilierungsprozess sollte zu einem Zeitpunkt erfolgen, zu dem die Erkennung potentieller SIMD-Ausführungen möglich ist und entschieden werden kann, ob diese wiederum zu gültigem Assemblercode führen. Dies setzt also einen ausreichenden Informationsgehalt des Zwischencodes voraus, der im günstigsten Fall ohne die Durchführung von aufwändigen Analysen ermittelbar ist. Sollen bestimmte Anweisungen als SIMD-Operationen umgesetzt werden, dürfen diese von nachfolgenden Compilierungsphasen nicht ungewollt wieder rückgängig gemacht werden. Es muss also dafür gesorgt werden, dass die erforderlichen Informationen in allen nachfolgend ausgeführten Compilierungsphasen verfügbar sind.

Bei der Ausnutzung von SIMD-Operationen hat die Lage (*Ausrichtung*) der Daten im Speicher einen großen Einfluss darauf, ob und mit welchem Aufwand eine Vektorisierung durchgeführt werden kann. So ist für eine Vektorisierung häufig das Einfügen zusätzlicher Datentransfers zum Packen und Entpacken der Vektorregister (oder beim M3: Gruppenregister) erforderlich. Aus dem Grund kann es bei einer ungünstigen Datenanordnung durchaus vorkommen, dass zwar die Ausführung einer SIMD-Operation möglich ist, allerdings der verursachte Overhead den Nutzen übersteigt und somit eine Ausführung der SIMD-Operation vermieden werden sollte. Grundsätzlich können hier die folgenden Fälle unterschieden werden:

- Eine Gruppe von Daten muss aus mehreren Gruppen zusammengesetzt werden.
- Die Anordnung der Daten innerhalb einer Gruppe muss verändert werden.
- Eine Kombination aus den beiden vorherigen Punkten.
- Ein Packen und Entpacken der Daten ist nicht erforderlich.

Der Optimalfall besteht offensichtlich darin, dass die Daten bereits so im Speicher abgelegt sind, wie sie auch verarbeitet werden sollen. Da dies allerdings nicht der Regelfall ist, sind hier Verfahren erforderlich, die eine geeignete Anordnung der Daten im Speicher bestimmen und in Abhängigkeit von dieser Datenanordnung die gegebenen Datentransfer-Modi effektiv ausnutzen.

4.1.2 M3-spezifische Problembereiche

In Ergänzung zum vorherigen Abschnitt sind bei den M3-Prozessoren aufgrund der vorhandenen Irregularitäten weitere Problembereiche zu berücksichtigen: Da zur Ausführung

von SISD-Operationen keine separate Funktionseinheit zur Abarbeitung vorgesehen ist, besitzt der Datenpfad 0 gegenüber den restlichen Datenpfaden eine Sonderfunktionalität, in dem neben der Ausführung von SIMD-Operationen auch die Ausführung von SISD-Operationen vorgesehen ist. Dadurch begründet besitzen die Elemente des Registerfiles vom Datenpfad 0 gegenüber denen der restlichen 15 Datenpfade ebenfalls eine Sonderfunktionalität, was eine korrekte Umsetzung der Codegenerierung erheblich erschwert. So werden bei einer SISD-Verarbeitung alle zu verarbeitenden Argumente entweder in den Eingangsregistern oder im Akkumulator des Datenpfades 0 erwartet. Um zusätzlich eine möglichst effektive Ausführung im SISD-Modus zu unterstützen, bestehen spezielle Datentransfer-Modi zwischen den Registern anderer Datenpfade und denen des Datenpfades 0.

Mit der Realisierung des Speichers der M3-Prozessoren als Gruppenspeicher wird eine große Speicherbandbreite zur Verfügung gestellt, die in Verbindung mit der Ausführung von SIMD-Operationen ein großes Potential zur Verbesserung der Codequalität bietet. Allerdings wird eine Konsistenzhaltung der Daten im SISD-Modus erheblich erschwert, da nur Zugriffe auf Gruppen (oder Partitionen) möglich sind. So muss zur Modifizierung eines einzelnen Wertes im Gruppenspeicher zunächst die entsprechende Gruppe in eines der Gruppenregisterfiles `A`, `B`, `C` oder `D` (s. auch Abb. 1.4 auf Seite 8) geladen werden. Erst dann kann der zu speichernde Wert an der entsprechenden Stelle im Registerfile modifiziert und die geänderte Gruppe wieder in den Gruppenspeicher zurückgeschrieben werden. Offensichtlich führt das im SISD-Modus erforderliche zusätzliche Laden der Gruppen im Vergleich zu herkömmlichen Speichern zu einem großen Overhead.

Zusätzliche Probleme treten bei der Verarbeitung von Daten auf, deren exakte Adresse mit statischen Analysen zur Übersetzungszeit nicht ermittelt werden kann. Wenn beispielsweise innerhalb einer Schleife ein fortlaufender Zugriff auf ein Array erfolgt, dann kann keine allgemeine Aussage darüber gemacht werden, in welchem Slice sich das aktuell adressierte Array-Element befindet. Da die Elemente eines Arrays i.d.R. an fortlaufenden Positionen im Speicher abgelegt werden, kann der entsprechende Slice aufgrund der Adresse *addr* des Array-Elements berechnet werden. Bei einer Gruppengröße von n-Elementen ergibt sich der Slice *slice* durch den Ausdruck: $slice = addr$ modulo n. Um diese Berechnungen nicht in Software realisieren zu müssen, ist die Verwendung von Befehlen vorgesehen, die Gebrauch von einer der beiden Hardware-Ressourcen *index_read* oder *index_write* machen. Beide Ressourcen haben gemein, dass sie die letzten vier Bit einer zuvor durchgeführten Adressberechnung enthalten:

- index_read-Ressource
  Diese Ressource wird gesetzt, wenn ein Lesezugriff auf den Gruppenspeicher mit Hilfe des Page-Pointer-Registers *PP* oder dem Adressregister P_0 erfolgt. Im Falle einer reinen Adressmodifikation (ohne Speicherzugriff), kann durch die Auswahl

eines entsprechenden Adressbefehls darauf Einfluss genommen werden, welche der beiden Index-Ressourcen gesetzt werden soll.

- index_write-Ressource
  In Analogie zum Setzen der index_read-Ressource wird diese Ressource bei einem Schreibzugriff mit Hilfe des Page-Pointer-Registers *PP* gesetzt oder bei einer Adressberechnung mit dem Adressregister P_1.

Unabhängig davon, ob lesend oder schreibend auf den Gruppenspeicher zugegriffen wird, hat eine Verwendung der Adressregister P_2 und P_3 keinen Einfluss auf die Index-Ressourcen.

4.1.3 Auswirkungen auf den Codegenerator

Aufgrund der in den beiden vorherigen Abschnitten beschriebenen Probleme bei der Ausnutzung von SIMD-Operationen sind in einigen Phasen des Codegenerators Erweiterungen erforderlich:

- Architekturspezifikation
  Für eine generische Handhabung von SIMD-Operationen ist eine Spezifikation der SIMD-Ausführungsmöglichkeiten erforderlich. Des Weiteren reicht aufgrund der zuvor beschriebenen Irregularitäten, im Gegensatz zur Modellierung von homogenen Registerfiles, die Angabe der Größe eines bestimmten Registerfiles nicht mehr aus, so dass eine Unterscheidung bestimmter Elemente eines Registerfiles erforderlich wird.

- Programmdarstellung
  Aufgrund der besonderen Eigenschaften des M3-Gruppenspeichers muss zwecks Wahrung der Datenkonsistenz im SISD-Modus sichergestellt werden, dass zu speichernde Daten in der richtigen Gruppe und im richtigen Slice abgelegt werden.

- Registerallokation
  Da jeder Speicherzugriff eine Gruppe von Daten betrifft, müssen anstelle des üblicherweise ausreichenden Spillens von einzelnen Registerelementen nun komplette Registerfiles (Gruppen von Daten) gespillt werden. Weitere Besonderheiten ergeben sich auch, wenn auf ein Element zugegriffen werden soll, dessen Position im Registerfile während der Übersetzungszeit nicht ermittelt werden kann, wie es bei der Verarbeitung von Arrays in Schleifen häufig vorkommt. Da in diesem Fall in der entsprechenden Maschinenoperation kein konkretes Registerelement angegeben werden kann, muss eine der Hardware-Ressourcen index_read oder index_write verwendet werden. Mit Durchführung der Registerallokation ist dann davon auszugehen,

dass potentiell *alle* Elemente des relevanten Registerfiles betroffen sind. Obwohl also z.B. bei einem Schreibzugriff mit Hilfe der index_write-Ressource nur in ein einzelnes Element des Registerfiles geschrieben wird, dürfen in diesem Registerfile erst dann wieder Modifikationen vorgenommen werden, wenn alle von der Anweisung datenabhängigen Operationen ausgeführt worden sind.

- Adresscode-Generierung
  Für alle Operationen, die mit Hilfe einer Index-Ressource auf eine Register-Ressource zugreifen, ist bei der Adresscode-Generierung darauf zu achten, dass die in Verbindung mit dem Einsatz der Index-Ressourcen verbundenen Randbedingungen eingehalten werden: Um ein entsprechendes Setzen der Index-Ressourcen zu gewährleisten, kann es deswegen unter gewissen Umständen erforderlich sein, zusätzliche *Dummy*-Adressbefehle einzufügen. Dies sind Adressbefehle, die normalerweise nicht zur Adressberechnung erforderlich gewesen wären, allerdings dafür sorgen, dass die entsprechende Index-Ressource vor deren Verwendung auf den richtigen Slice gesetzt wird. Durch Sequentialisierungskanten wird zusätzlich dafür gesorgt, dass bei der nachfolgend durchzuführenden Adresscode-Kompaktierung diese Ressource nicht vor der planmäßigen Verwendung neu gesetzt wird. Die sich in Verbindung mit der Verwendung der Index-Ressourcen ergebenden Randbedingungen bezüglich der einzusetzenden Adressregister werden direkt bei der Initialisierung des jeweiligen Knotens der Speicher-Zugriffssequenz berücksichtigt.

Um die Techniken auch auf andere Prozessoren mit SIMD-Funktionalität anwenden zu können, gilt es also vor allem eine generische Beschreibung der SIMD-Operationen auf den GeLIR-Datenstrukturen zu ermöglichen. Wenn dies erreicht wird, können die für die M3-Prozessoren implementierten SIMD-Techniken im Grundsatz auch für andere SIMD-Prozessoren wiederverwendet werden. Ausnahmen stellen dabei mit Sicherheit einige spezielle Maßnahmen zur Handhabung des Gruppenspeichers dar. Allerdings werden bei der Entwicklung von Compilern immer wieder derartige Architektur-Besonderheiten zu berücksichtigen sein, die eine spezielle Anpassung bestehender Optimierungstechniken erfordern.

4.2 Bestehende Verfahren

Die automatisierte Ausnutzung von SIMD-Operationen kann grundsätzlich in Verfahren eingeteilt werden, die eine Vektorisierung von Schleifen vornehmen, oder innerhalb eines Basisblocks nach Anweisungen suchen, die zu SIMD-Operationen zusammengefasst werden können.

Die Grundidee der ersten Strategie besteht darin, vor der Durchführung der Codegenerierungs-Teilaufgaben CS, IA und RA nach Schleifen zu suchen und diese auf ihre

Vektorisierbarkeit hin zu untersuchen. In [KP93, MKC00, PSB01] werden dazu Verfahren auf Basis einer Mustererkennung vorgestellt. Diese suchen in einem ersten Schritt im Quellprogramm nach Programmfragmenten, deren Struktur mit einem der spezifizierten Muster übereinstimmt (*Idiom-Recognition*). Nach einer erfolgreichen Überprüfung weiterer Randbedingungen, werden dann die entsprechenden Programmfragmente unabhängig voneinander im Quellprogramm durch Funktionsaufrufe ersetzt. Die für diese Funktionsaufrufe optimierten Assembleranweisungen werden dann vom Compiler an den entsprechenden Stellen eingefügt. Um eine möglichst große Überdeckung von Programmfragmenten mit Mustern zu erzielen, werden zusätzlich Transformationen des Quellprogramms vorgenommen. Da z.B. in [MKC00] nur Schleifen mit einer Anweisung gehandhabt werden können, erfolgt dort eine Aufteilung von Schleifen mit mehr als einer Anweisung in mehrere einfachere Schleifen.

Die klassische Vorgehensweise zur Vektorisierung von Schleifen basiert darauf, zunächst durch die Anwendung von Schleifentransformationen, die in einem gegebenen Programm vorhandenen Schleifen in eine vektorisierbare Form zu bringen. Mit Hilfe des in [AK87] beschriebenen Verfahrens können unter Einhaltung der Datenabhängigkeiten vektorisierbare und nicht vektorisierbare Anweisungen unterschiedlichen Schleifen zugeordnet werden, so dass zumindest ein Teil der Anweisungen durch Vektoroperationen umgesetzt werden kann. Eine detaillierte Beschreibung dieser Vorgehensweise wird u.a. in [Zim90] anhand von Fortran 90-Programmen [Ada92] gegeben. Im Gegensatz zu C-Programmen besteht dabei die Möglichkeit, eine parallele Verarbeitung auf Arrays direkt in den Programmen auszudrücken. Auf diese Weise kann dem Codegenerator relativ einfach mitgeteilt werden, dass bestimmte Anweisungen vektorisiert werden können. Da im ANSI C-Standard keine derartigen Sprachkonstrukte vorgesehen sind, ist dieses bei der Übersetzung von C-Programmen nicht möglich.

In [SG00] wird ein vektorisierender C-Compiler für Intels *MMX-Befehle* (MMX = Multimedia Extension) des Pentium Prozessors vorgestellt. Nach der Erkennung von vektorisierbaren Schleifen fügt der auf Basis der SUIF-Entwicklungsumgebung implementierte Compiler inline Assemblercode für die entsprechenden Programmfragmente in das C-Programm ein. Mit Ausnutzung der MMX-Befehle ergibt sich hier eine Erhöhung der Ausführungsgeschwindigkeit bis zu einem Faktor von sechs. Um die Ergebnisse einer erfolgreichen Vektorisierung an den Codegenerator weiterzuleiten, werden in [DeV97, Kra00] zu vektorisierende Anweisungen in der Compiler-Zwischendarstellung entsprechend gekennzeichnet. In [DeV97] wird dabei am Beispiel des Torrent Vektorprozessors [ABI+95] ein vektorisierender SUIF-Compiler für traditionelle Vektorprozessoren vorgestellt. In [Kra00] erfolgt dies auf Basis der CoSy-Entwicklungsumgebung für den *Visual Instruktionssatz* (VIS) [KMT+95] der UltraSPARC-Prozessoren. Experimentelle Ergebnisse für einfache Schleifen zeigen hier, dass eine Steigerung der Ausführungsgeschwindigkeit bis zu einem Faktor von 4,8 möglich ist.

Diese Verfahren haben gemein, dass sie jeweils vor der Durchführung der Codegenerierung aufsetzen. Um die Anzahl der vektorisierbaren Schleifen zu erhöhen, werden in allen Verfahren zusätzlich Schleifentransformationen wie *Loop-Fission*, *Strip-Mining*, *Reduction-Recognition* oder *Scalar-Expansion* durchgeführt (s. z.B. [BGS94] für einen Überblick). Die nachfolgend beschriebenen Verfahren versuchen stattdessen SIMD-Ausführungsmöglichkeiten durch Zusammenfassen homogener Operationen im Zuge der Codegenerierung auf Basisblock-Ebene aufzuspüren.

In [Kra00, LA00] wird eine Ausnutzung von SIMD-Operationen in Verbindung mit der Codeselektion und der Instruktionsanordnung vorgeschlagen. Dabei wird zuvor durch n-faches Abrollen der Schleife die in einem Basisblock vorhandene Parallelität erhöht. Im Allgemeinen stellt hierbei die Anzahl der parallelen Einheiten ein gutes Maß für den Abrollfaktor dar. Danach werden gleichartige (homogene) Anweisungen, die zusammen als SIMD-Operation ausgeführt werden können, zu Gruppen zusammengefasst. Die in [Kra00] dokumentierten Ergebnisse bezüglich dieser Vorgehensweise ergaben gegenüber einer ebenso vorgenommenen Vektorisierung von Schleifen, keine Unterschiede hinsichtlich der Codequalität. Mit dem in [LA00] beschriebenen Verfahren zum Zusammenfassen homogener Operationen zu SIMD-Oprationen kann für den betrachteten Mikroprozessor die Ausführungsgeschwindigkeit für ein Benchmark um den Faktor 6,7 gesteigert werden und für die restlichen Benchmarks bis zu einem Faktor von 1,8. Die in [Leu00b] beschriebene Technik führt eine Ausnutzung von SIMD-Operationen in Verbindung mit der Codeselektion durch, indem alternative Überdeckungen von Bäumen mit Prozessorinstruktionen als lineares Gleichungssystem formuliert werden. Der Nachteil dieser Methode liegt allerdings in der zu erwartenden hohen Laufzeit zur Lösung von linearen Gleichungssystemen für eine große Anzahl paralleler Datenpfade. So wurde in [Leu00b] lediglich ein Abrollfaktor zwischen null und drei gewählt, wobei im Falle des M3-DSPs ein Abrollfaktor von 16 erforderlich wäre.

Die Vorteile der zuletzt genannten Verfahren liegen in der geringeren Komplexität der erforderlichen Analysen. Für den Fall, dass nur geringe Teile des entstandenen Zwischencodes mit SIMD-Operationen überdeckt werden können, besteht hier jedoch eine große Gefahr einer drastischen Erhöhung der Codegröße. Leider gibt es hier keine klaren Ausschluss-Kriterien für die eine oder andere Vorgehensweise, so dass eine Entscheidung, welche Methode für welche Architektur am geeignetsten ist, von Fall zu Fall entschieden werden sollte. Da sich diese Arbeit mit der Compilierung für DSPs beschäftigt, erscheint in Ergänzung zu den Techniken für traditionelle Vektorprozessoren und GPPs die Entwicklung von Verfahren erforderlich, mit denen die irregulären Architektureigenschaften von DSPs entsprechend berücksichtigt werden können. Dies betrifft auch Techniken zur Handhabung von DSP-Architekturen mit Gruppenspeichern, wie im Falle der M3-Prozessoren und des Media-Prozessors von MicroUnity.

4.3 Übersicht

Wie zuvor beschrieben gibt es zwei grundsätzliche Vorgehensweisen zur Ausnutzung von SIMD-Operationen. Aufgrund der weniger komplexen Analysen stellt die Methode des Zusammenfassens homogener Operationen eines Basisblocks zu SIMD-Operationen eine interessante Strategie dar. Durch eine entsprechende Erweiterung der Codeselektions-Phase kann eine solche Optimierung auf elegante Art und Weise in den bisherigen Codegenerierungs-Prozess integriert und dadurch mit den Aufgaben CS, IA und RA phasengekoppelt durchgeführt werden. Im Gegensatz zu anderen Optimierungsverfahren ergeben sich hier einige Vorteile: Da kein separater Optimierungsschritt erforderlich ist, kann eventuell vorhandener Overhead zum Ausrichten der Daten mit in die Bewertung einbezogen werden. Dies würde dazu führen, dass nur dann SIMD-Operationen ausgewählt werden, wenn der Nutzen den Overhead übersteigt. Aufgrund des umgesetzten graphbasierten Codeselektions-Verfahren besteht auch die Möglichkeit, (SIMD-)Ausführungsmöglichkeiten zwischen Bäumen zu erkennen und auszunutzen, ohne eine Erweiterung der bestehenden Codegenerierung vorzunehmen.

Trotz dieser Vorteile gegenüber herkömmlichen Codegenerierungs-Verfahren bietet sich im Falle der M3-Prozessoren jedoch eine Vektorisierung von Schleifen an, da ein 16-faches Abrollen einer zu vektorisierenden Schleife zu extrem großen Basisblöcken führt. Im Vergleich zu anderen Prozessoren würden dadurch die bereits genannten Nachteile des potentiellen Codegrößen-Overheads und der größeren Laufzeit der Optimierungen verstärkt ins Gewicht fallen. Für Prozessoren mit weniger parallelen Ausführungseinheiten und damit auch mit einem geringeren erforderlichen Abrollfaktor wäre die Umsetzung einer solchen Vorgehensweise allerdings wieder eine sehr interessante Alternative.

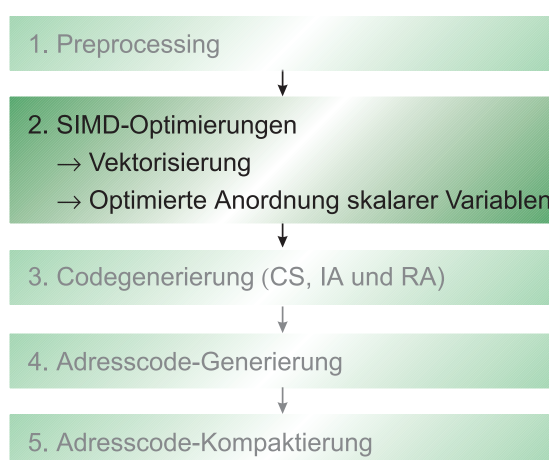


Abb. 4.1: Einordnung der SIMD-Optimierungen in das Back-End

In Ergänzung zu dem in Abschnitt 3.3 vorgestellten groben Ablauf der Optimie-

rungen im Back-End erfolgt in Abb. 4.1 eine Einordnung der entwickelten SIMD-Optimierungen. Dies betrifft zum einen die Durchführung einer Vektorisierung von Schleifen zur Ausnutzung der parallelen Datenpfade und zum anderen eine optimierte Anordnung von skalaren Variablen im Gruppenspeicher zur effektiven Ausnutzung von SIMD-Speicherzugriffen. Beide Optimierungen werden nach der Preprocessing-Phase und vor den Codegenerierungs-Phasen CS, IA und RA durchgeführt.

Bevor in den Abschnitten 4.6 und 4.7 die entwickelten SIMD-Optimierungen vorgestellt werden, erfolgt zunächst eine Beschreibung der erforderlichen Erweiterungen hinsichtlich der Architektur- und der Programmdarstellung.

4.4 Architekturdarstellung

Um eine generische Implementierung von SIMD-Optimierungen zu ermöglichen, ist eine geeignete Beschreibung der SIMD-Funktionalität essentiell. SIMD-Operationen haben die Eigenschaft, dass mit einer Operation mehrere gleichartige (homogene) Operationen parallel ausgeführt werden. Im Gegensatz zu einer SISD-Ausführung müssen daher statt *einem* Ergebnis eine Reihe von Ergebnissen gespeichert werden. Bei den Prozessoren der M3-Plattform betrifft das alle Elemente eines Registerfiles. Aufgrund der vorhandenen Irregularitäten ist bei der Architekturspezifikation eine Unterscheidung der einzelnen Elemente eines Registerfiles zwingend erforderlich. Die auf Seite 34 in Abb. 2.6 gegebene Spezifikation der Multiplikations-Operation des M3-DSPs muss nun dahingehend modifiziert werden, dass einzelne Funktionseinheiten und Registerelemente eines Registerfiles unterschieden werden können. Bis auf die zusätzliche Kennzeichnung der verwendbaren Register und Funktionseinheiten entsprechen die alternativen Ausführungsmöglichkeiten *LirAltEntry* 1 und *LirAltEntry* 2 in Abb. 4.2 der Spezifikation der SISD-Operation.

`Op =`{MUL0}` `FU =`{DMU0}` `IT =`{1}` `Def =`{ACCU0,'*'}` `Arg1=`{A0,B0,'CNST1','CNST2'}` `Arg2=`{A0,C0,D0,ACCU0}`	**LirAltEntry 1** **Attribute:** Exec-Time = 1 Latency = 1 Swapped-Args = false
`Op =`{MUL0}` `FU =`{DMU0}` `IT =`{1}` `Def =`{ACCU0,'*'}` `Arg1=`{A0,C0,D0,ACCU0}` `Arg2=`{A0,B0,'CNST1','CNST2'}`	**LirAltEntry 2** **Attribute:** Exec-Time = 1 Latency = 1 Swapped-Args = true
`Op =`{SIMD_MUL}` `FU =`{DMU}` `IT =`{1}` `Def =`{ACCU,'simd*'}` `Arg1=`{A,B,'CNST1','CNST2'}` `Arg2=`{A,C,D,ACCU}`	**LirAltEntry 3** **Attribute:** Exec-Time = 1 Latency = 1 Swapped-Args = false
`Op =`{SIMD_MUL}` `FU =`{DMU}` `IT =`{1}` `Def =`{ACCU,'simd*'}` `Arg1=`{A,C,D,ACCU}` `Arg2=`{A,B,'CNST1','CNST2'}`	**LirAltEntry 4** **Attribute:** Exec-Time = 1 Latency = 1 Swapped-Args = true

Abb. 4.2: M3-DSP Multiplikation einschließlich SIMD-Alternativen

LirAltEntry 3 und 4 stellen hingegen eine Erweiterung der Operations-Spezifikation um SIMD-Funktionalität dar. So wird z.B. mit `A` das gesamte Registerfile assoziiert, während

mit `A0` in *LirAltEntry* 1 und 2 ein konkretes Registerelement des Datenpfades 0 bezeichnet wird. Des Weiteren können mit Hilfe der flüchtigen Ressource `'simd*'`, analog zum SISD-Modus, MAC-Operationen modelliert werden.

Am Beispiel des Element-Datentransfers (`ElDT`) wird in Abb. 4.3 ferner die Sonderfunktionalität der Register des Datenpfades 0 verdeutlicht. Es ist zu erkennen, dass Datentransfers von allen Elementen der Registerfiles `A`, `B`, `D`, `Accu` und `M` nach `A0`, `B0` bzw. `C0` möglich sind, allerdings zu keinen weiteren Registerelementen anderer Datenpfade.

```
Op  ={ElDT}                              LirAltEntry 1
FU  ={DTU}
IT  ={1}
Def ={A0,B0,C0}
Arg1={-}                                 Attribute:
Arg2={A0,...,A15,B0,...,B15,             Exec-Time = 1
      D0,...,D15,M0,...,M15,             Latency = 1
      Accu0,...,Accu15}                  Swapped-Args = false
```

Abb. 4.3: M3-DSP Element-Datentransfer

Die Spezifikation einer Operation, die einen Datentransport eines Einzelwertes unter Verwendung der index_read-Ressource durchführt, ist in Abb. 4.4 dargestellt. Da die Position des zu transportierenden Wertes nicht bekannt ist, kann im Prinzip jedes Element eines bestimmten Registerfiles betroffen sein. Aus diesem Grund sind nicht alle einzelnen Registerelemente als Argument aufgeführt, sondern nur das Registerfile, aus dem der Wert gelesen werden soll. Bei der Durchführung der Registerallokation muss dies also entsprechend berücksichtigt werden, indem angenommen wird, dass aus allen Elementen des Registerfiles gelesen wird.

```
Op  ={ElDT_IndexRead}                    LirAltEntry 1
FU  ={DTU}
IT  ={1}                                 Attribute:
Def ={A0,B0,C0,D0}                       Exec-Time = 1
Arg1={-}                                 Latency = 1
Arg2={A,B,D,M,ACCU}                      Swapped-Args = false
```

Abb. 4.4: M3-DSP Element-Datentransfer mit Verwendung der index_read-Ressource

Zur Verdeutlichung der Auswirkungen der Spezifikationsänderung sind in Abb. 4.5 für zwei Knoten eines Datenflussgraphen die sich nun ergebenden alternativen Ressource-Mengen angegeben. Alle Ressource-Alternativen, die speziell zur Handhabung des Gruppenspeichers oder zur Ausnutzung der SIMD-Funktionalität zusätzlich benötigt werden, sind fett gedruckt. Es zeigt sich, dass für den Knoten 6 nun neben der SISD-Multiplikation `MUL0` zusätzlich die Auswahl einer entsprechenden SIMD-Operation `SIMD_MUL` möglich ist. Eine Einschränkung der Definitionsmenge auf die Ressource `'simd*'`, würde zu einer

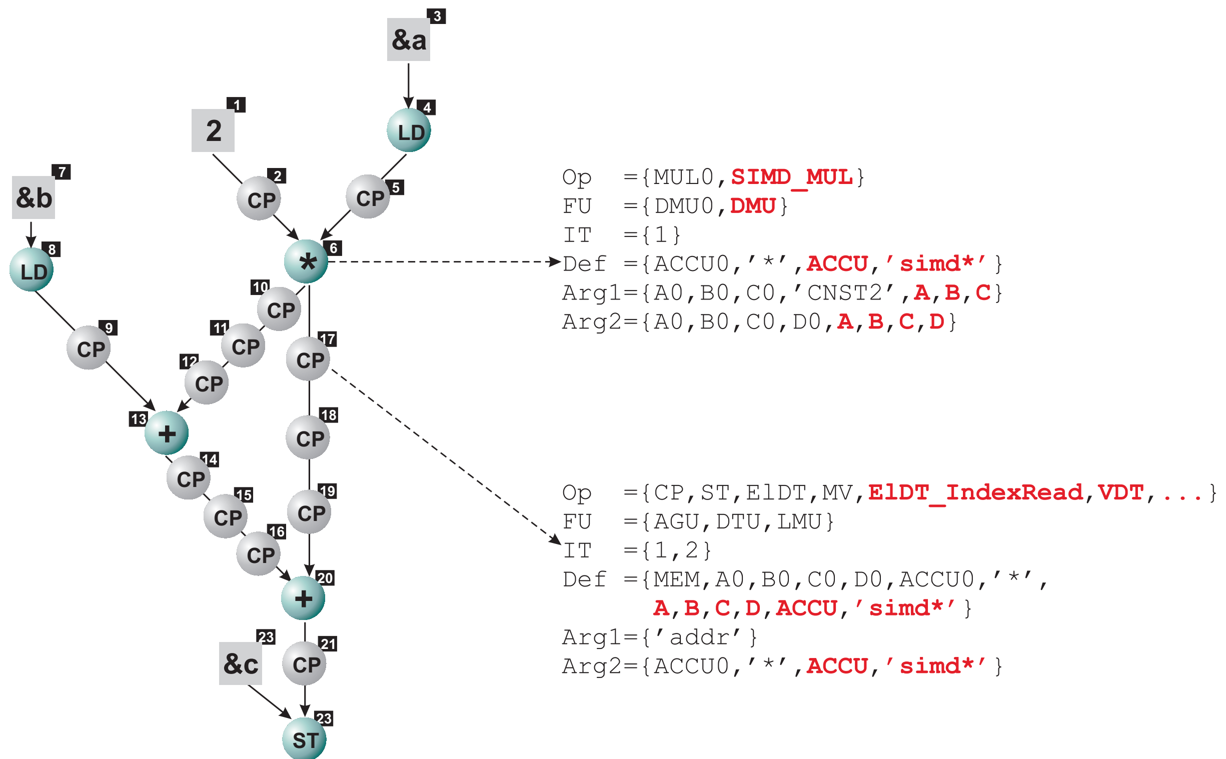


Abb. 4.5: Überdeckung eines Datenflussgraphen einschließlich SIMD-Operationen

Ausführung von zwei SIMD-MAC-Operationen führen, da die nachfolgenden Additionen (s. Knoten 13 und 20) diese Ressource automatisch als Argument zugewiesen bekommen würden. Die *CP*-Knoten 10, 11, 12, 17, 18 und 19 würden in diesem Fall jeweils mit einer abstrakten Copy-Operationen überdeckt werden und damit die flüchtige Ressource `'simd*'` weiterleiten, ohne einen zusätzlichen Prozessorzyklus zu beanspruchen.

4.5 Programmdarstellung

Der M3-Gruppenspeicher ermöglicht im SIMD-Modus eine effektive Versorgung der Datenpfade mit Daten. Da keine Einzelzugriffe auf Daten im Speicher möglich sind, ist allerdings bei einer Abarbeitung im SISD-Modus gegenüber herkömmlichen Speichern ein erhöhter Aufwand zur Wahrung der Datenkonsistenz erforderlich. So muss sichergestellt werden, dass zu speichernde Daten sowohl in der richtigen Gruppe als auch im richtigen Slice abgelegt werden. Das Prinzip dieser Vorgehensweise wird in den Abbildungen 4.6 a) und b) anhand des Speicherns einer Konstanten und des Ergebnisses einer Operation verdeutlicht.

Die zusätzlich einzufügenden Operationen zur Handhabung des Gruppenspeichers sind jeweils durch Fettdruck hervorgehoben. In beiden Fällen müssen zwei zusätzliche Anweisungen zum Laden der Gruppe der Variablen a in das Registerfile `A` ausgeführt werden, be-

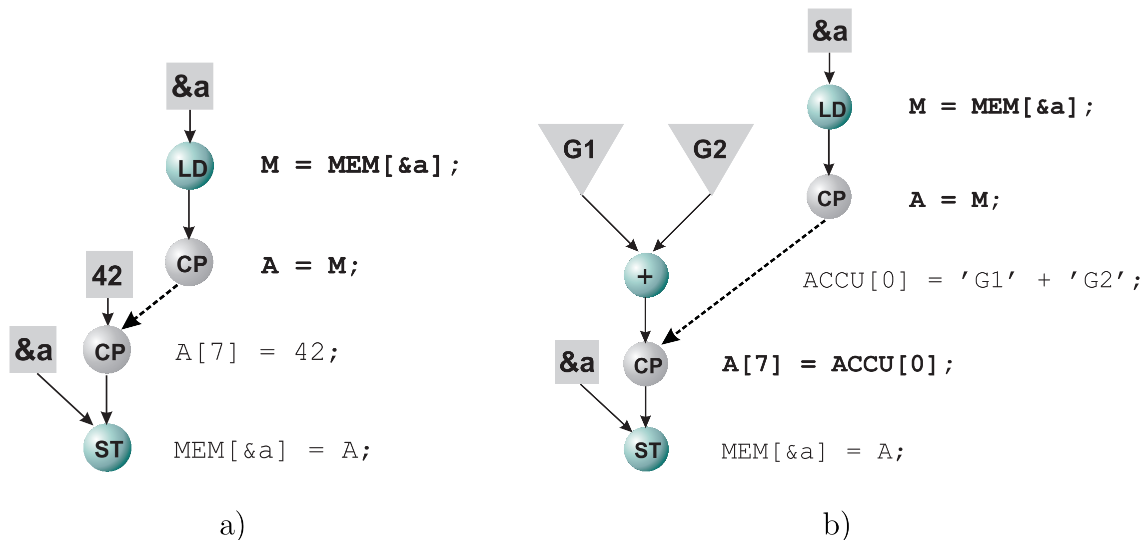


Abb. 4.6: Speichern von Daten unter Beibehaltung der Datenkonsistenz

vor der entsprechende Wert im Registerelement `A[7]` gesetzt werden kann. In Abb. 4.6 b) ist eine weitere Anweisung erforderlich, um das Ergebnis der Addition vom Akkumulator `ACCU[0]` in das Registerelement `A[7]` zu transportieren. Alle zum Laden der Gruppen erforderlichen abstrakten Maschinenoperationen werden in der Preprocessing-Phase des Back-Ends eingefügt. Dabei wird, wie in den Beispielen dargestellt, mit Hilfe von Ausgabeabhängigkeiten (s. gestrichelte Kanten) sichergestellt, dass die benötigte Gruppe vor der Modifikation in das richtige Registerfile geladen wird.

4.6 Vektorisierung von Schleifen

Aufgrund der zuvor genannten Nachteile, die durch ein häufiges Abrollen von Schleifen entstehen, basiert die hier entwickelte Technik zur Ausnutzung von SIMD-Operationen auf der Vektorisierung von Schleifen. Das Grundprinzip beruht darauf, zunächst alle in einem gegebenen Quellprogramm vorhandenen Schleifen auf ihre Vektorisierbarkeit hin zu überprüfen. Wird eine bestimmte Schleife als vektorisierbar eingestuft, werden entsprechende Einschränkungen bezüglich der zur Auswahl stehenden Operations-Alternativen vorgenommen. Die sich dadurch wiederum ergebenden weiteren Einschränkungen bezüglich anderer Ressourcen (wie Register) werden daraufhin automatisch durchgeführt. Dies hat den Vorteil, dass der Codegenerierung sehr präzise und gezielt Vorgaben gemacht werden können. Da nun prinzipiell mit einem Schleifendurchlauf mehrere Iterationen (beim M3-DSP: 16) simultan umgesetzt werden, verringert sich dadurch auch die Anzahl auszuführender Iterationen der vektorisierten Schleifen.

Zur Verdeutlichung dieser Vorgehensweise sei dazu das folgende C-Programm gegeben, wobei die Arrays A und B jeweils eine Größe von 1024 haben:

```
for(i=0; i<1024; i++)
 {
   x = A[i];
   x = x + 2;
   B[i] = x;
 }
```

Der linke Teil von Abb. 4.7 enthält den (etwas vereinfacht dargestellten) zum C-Code gehörigen GeLIR-Code, bei dem die *for*-Schleife in mehrere Teile aufgesplittet wurde. Dies betrifft Anweisungen zur Initialisierung und Aktualisierung der Schleifen-Indexvariablen i, zum Testen der Abbruchbedingung und zur Durchführung eines bedingten Sprungs an den Anfang der Schleife.

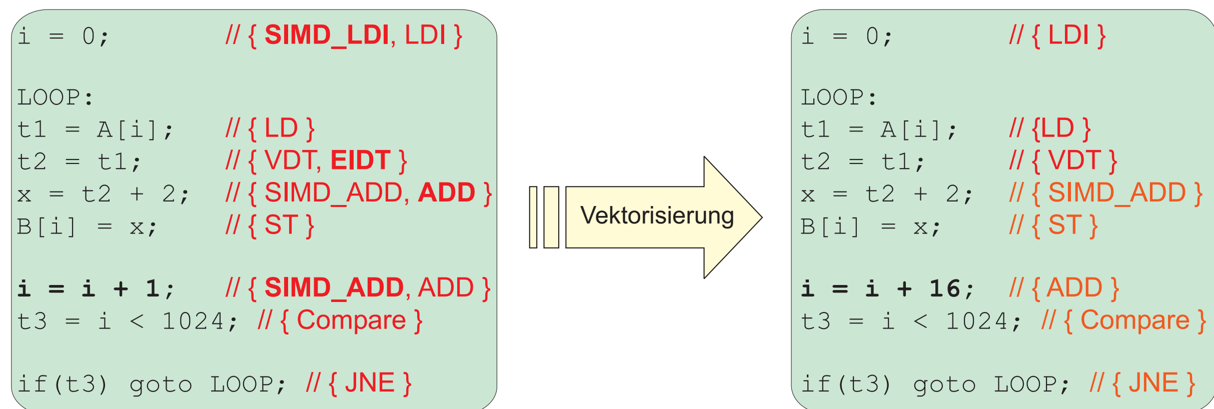


Abb. 4.7: Beispiel der Vektorisierung von Schleifen

Vorhandene alternative Auswahlmöglichkeiten bzgl. Operationen sind als Kommentare hinter den jeweiligen Code-Fragmenten angegeben. Mit der Durchführung der Analysen wurde erkannt, dass eine Vektorisierung dieser Schleife möglich ist, was zu dem im rechten Programmfragment dargestellten GeLIR-Code führt (relevante Änderungen sind durch Fettdruck hervorgehoben). Hier ist zu erkennen, dass die Schleifen-Indexvariable i, entsprechend der Anzahl paralleler Datenpfade, mit jedem Schleifendurchlauf nun um 16 statt um 1 erhöht wird. Da durch die vorgenommenen Einschränkungen mit jedem Speicherzugriff jeweils eine Gruppe von 16 Daten geladen und nachfolgend verarbeitet wird, führt dies dazu, dass statt der ursprünglich 1024 Schleifendurchläufe nun nur noch 64 erforderlich sind. Die Vektorisierung von Schleifen lässt sich weiterhin sehr gut mit bereits vorhandenen Optimierungen kombinieren, wie nachfolgend am Beispiel der Schleifenoptimierung zur Ausnutzung von *Zero-Overhead Hardware-Loops* (ZOLs) demonstriert wird.

Eine Realisierung von Schleifen durch Software ist in der Regel mit einem gewissen Overhead an Prozessorzyklen verbunden. Dieser ergibt sich durch zusätzlich erforderli-

che Prozessorzyklen zum Testen der Abbruchbedingung, zum Aktualisieren der Schleifen-Indexvariablen und durch nicht ausgenutzte Prozessorzyklen zur Umsetzung erforderlicher Sprungbefehle. Zur Reduzierung dieses Overheads wird von DSPs i.d.R. die Ausführung einer begrenzten Anzahl von Maschineninstruktionen in Zero-Overhead Hardware-Loops ermöglicht. Diese erlauben nach der Initialisierung eines speziellen Registers mit der Anzahl auszuführender Iterationen die Ausführung der eingebetteten Maschineninstruktionen ohne den sonst üblichen Schleifen-Overhead. So sind für das Testen der Abbruchbedingung und die Realisierung des Sprungbefehls zum Schleifenanfang keine zusätzlichen Prozessorzyklen erforderlich. Wird die Schleifen-Indexvariable lediglich als Schleifenzähler benötigt, kann des Weiteren auf die Aktualisierung dieser Variable verzichtet werden. Mit Hilfe der unteren und oberen Schleifengrenze und der Schrittweite der Schleife wird zunächst die Anzahl der auszuführenden Iterationen berechnet, um die Initialisierung eines entsprechenden Registers vornehmen zu können. Des Weiteren wird durch das Einfügen von zwei speziellen Maschinenoperationen die besondere Realisierung dieser Schleife gekennzeichnet. In Abb. 4.8 ist dies anhand des vektorisierten Beispielcodes aus Abb. 4.7 verdeutlicht.

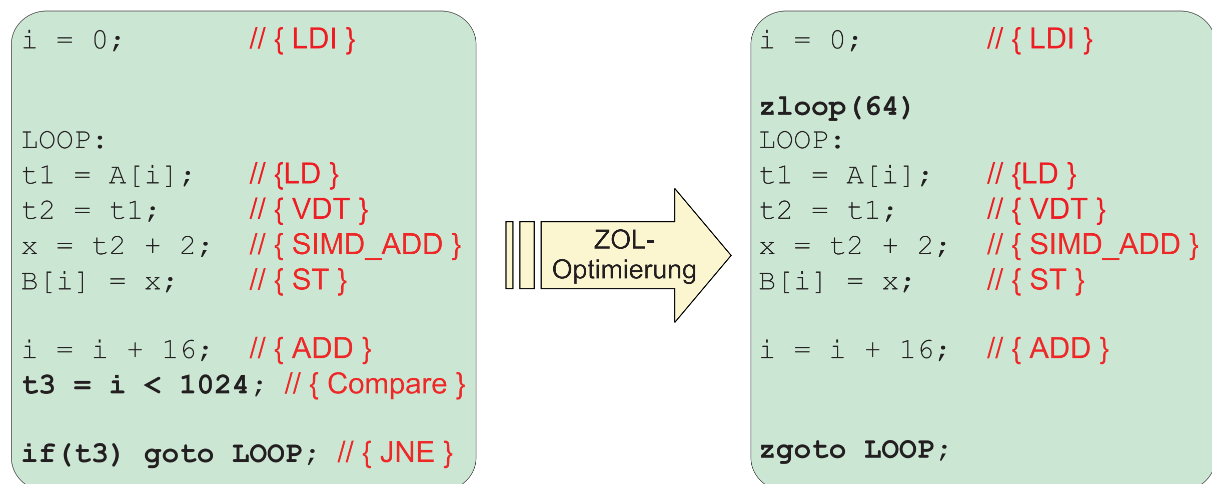


Abb. 4.8: Beispiel der Ausnutzung von Zero-Overhead Hardware-Loops

Zur Initialisierung der Hardwareschleife wird zum einen die Maschinenoperation `zloop` eingefügt, mit der eine Initialisierung der Hardwareschleife mit der Anzahl auszuführender Iterationen (hier 64) vorgenommen wird. Zum anderen wird die bedingte Sprunganweisung am Ende der Schleife durch die Maschinenoperation `zgoto` ersetzt. Diese symbolisiert, dass keine weiteren Instruktionen zum Testen der Schleifen-Abbruchbedingung und Aktualisieren des Schleifenzählers erforderlich sind. Nach einem erfolgreichen Einsetzen der speziellen Anweisungen wird abschließend eine *Dead-Code-Elimination* durchgeführt, die den redundant gewordenen Schleifencode entfernt. In Abbildung 4.8 betrifft dies lediglich den bedingten Sprungbefehl, da die Schleifen-Indexvariable noch zur Adressierung

der Arrays im Schleifenrumpf benötigt wird.

Wie sich an dem vorgestellten Beispiel erkennen lässt, stellt die ZOL-Optimierung eine sehr gute Ergänzung zur Vektorisierung dar. Weitere Informationen zu diesen Optimierungen können [Hor01b, LWDL02, LML02] entnommen werden.

Nach einem Überblick der einzelnen Schritte zur Vektorisierung von Schleifen wird im Folgenden detaillierter auf einige Aspekte der Umsetzung eingegangen.

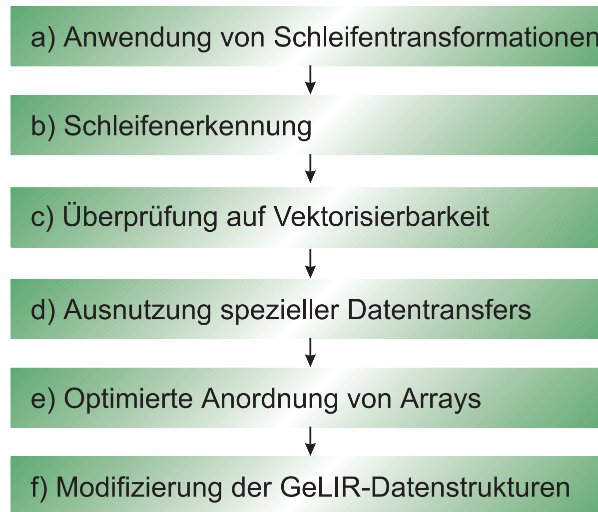


Abb. 4.9: Teilschritte zur Vektorisierung von Schleifen

a) Anwendung von Schleifentransformationen
Häufig liegen die in einer Anwendung vorhandenen Schleifen nicht in der Form vor, die eine Vektorisierung zulässt. Aus diesem Grund werden vor der Vektorisierung Schleifentransformationen durchgeführt, mit dem Ziel, die vorhandenen Schleifen in eine vektorisierbare Darstellung zu transformieren. Da eine Implementierung dieser Schleifentransformationen nicht zu den Zielsetzungen dieser Arbeit gehört, wurden die erforderlichen Transformationen manuell durchgeführt. Das Ergebnis stellt wiederum gültigen C-Code dar, der nach wie vor auf andere Architekturen portierbar ist. In Abschnitt 4.6.1 werden einige gebräuchliche Schleifentransformationen vorgestellt.

b) Schleifenerkennung
In diesem Schritt wird zunächst nach potentiell vektorisierbaren Schleifen gesucht. Aufgrund der Komplexität der in einem späteren Schritt noch durchzuführenden Datenflussanalyse werden hier lediglich Schleifen betrachtet, die genau einen Einsprungs- und Austrittspunkt (*Single-Entry Single-Exit*) besitzen. Um eine solche Schleife im nachfolgenden Schritt auf ihre Vektorisierbarkeit hin untersuchen zu

können, werden außerdem die untere und obere Schleifengrenze sowie die Schleifen-Indexvariable ermittelt. Im Falle einer Vektorisierung werden diese Informationen benötigt, um eine entsprechende Anpassung der Anzahl durchzuführender Schleifendurchläufe vornehmen zu können. Für eine Beschreibung der eingesetzten Analysetechniken soll an dieser Stelle auf die Diplomarbeit von Horst [Hor01b] verwiesen werden.

c) Überprüfung auf Vektorisierbarkeit
Zur Feststellung, ob eine bestimmte Schleife vektorisierbar ist, müssen eine Reihe von Analysen durchgeführt werden, auf die in Abschnitt 4.6.2 kurz eingegangen wird. Eine detaillierte Beschreibung der Analysetechniken kann der Diplomarbeit von Horst [Hor01b] entnommen werden.

d) Ausnutzung spezieller Datentransfers
Um bei einer ungünstigen Anordnung der Daten im Speicher den zur Ausrichtung der Daten erforderlichen Overhead so gering wie möglich zu halten, ist die Verwendung spezieller Datentransfer-Modi des Verbindungsnetzwerkes essentiell. Detaillierte Informationen hierzu befinden sich Abschnitt 4.6.3.

e) Optimierte Anordnung von Arrays
Häufig ist eine Vektorisierung von Schleifen nur bei einer entsprechenden Ausrichtung der Daten im Gruppenspeicher möglich. Da die Verarbeitung von Daten innerhalb von Schleifen häufig auf Arrays basiert, wird in diesem Schritt versucht, die verwendeten Arrays in einer geeigneten Form im Gruppenspeicher abzulegen. Eine genauere Beschreibung erfolgt in Abschnitt 4.6.4.

f) Modifizierung der GeLIR-Datenstrukturen
Wird eine Schleife als vektorisierbar eingestuft, werden in diesem letzten Schritt die entsprechenden Modifikationen auf den GeLIR-Datenstrukturen vorgenommen. Dies umfasst zunächst eine entsprechende Einschränkung der Operations-Alternativen für Graphknoten, die SIMD-Operationen ausführen sollen. Zusätzlich wird die Schrittweite der Schleife entsprechend der Anzahl parallel ausgeführter Iterationen in einem Schleifendurchlauf angepasst. Alle durch die Vektorisierung nicht benötigten Operationen zur Handhabung des Gruppenspeichers im SISD-Modus werden ebenfalls aus den GeLIR-Datenstrukturen gelöscht und müssen in der nachfolgend durchzuführenden Codegenerierung nicht betrachtet werden.

4.6.1 Unterstützende Schleifentransformationen

Mit der Anwendung von Schleifentransformationen wird häufig bezweckt, den gegebenen Programmcode derart zu modifizieren, dass nachfolgende Optimierungen effektiver

arbeiten, bzw. überhaupt erst durchgeführt werden können. Dies trifft auch in unserem Fall auf die Vektorisierung zu, bei der durch die Anwendung von Schleifentransformationen die Anzahl der vektorisierbaren Schleifen erhöht werden soll. Im Folgenden werden aus diesem Grund die häufig verwendeten Schleifentransformationen *Loop-Unswitching*, *Loop-Interchange*, *Loop-Split* und *Reduction-Recognition* vorgestellt und deren Arbeitsweise anhand eines Beispiels verdeutlicht.

Als Ausgangspunkt betrachten wir das in Abb. 4.10 dargestellte Programmfragment `original`, das schrittweise durch Anwendung von Schleifentransformationen in eine vektorisierbare Form gebracht werden soll.

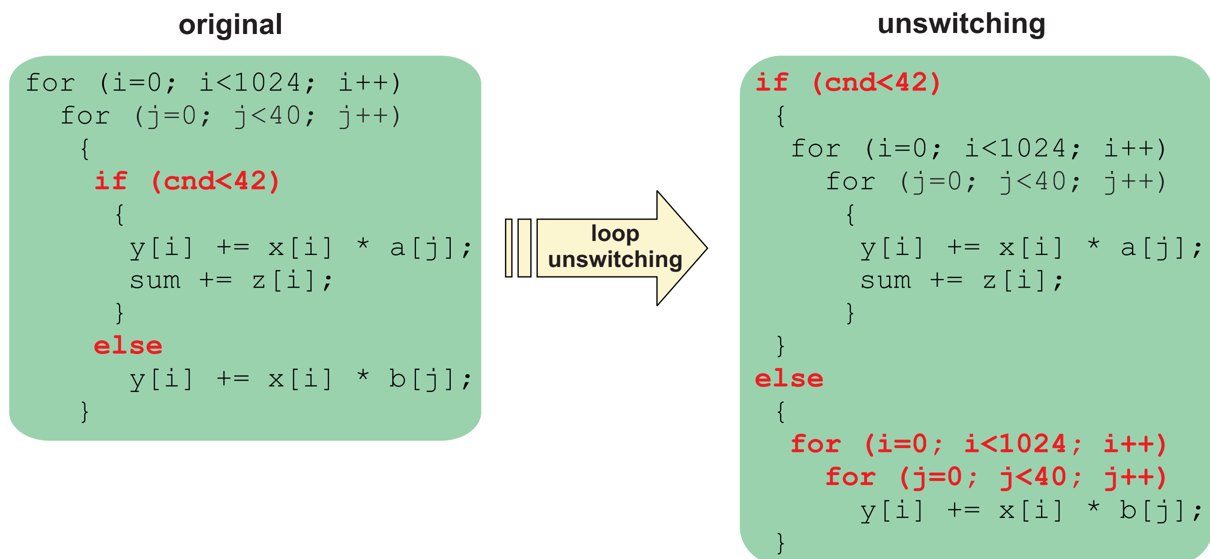


Abb. 4.10: Beispiel zur Anwendung von Loop-Unswitching

Wie zu erkennen ist, enthält das Programm in der innersten der beiden ineinander geschachtelten Schleifen eine *if*-Anweisung. Grundsätzlich besteht durchaus die Möglichkeit, derartige Konstrukte zu vektorisieren, was allerdings aufgrund der Verzweigungen des Kontrollflusses im Schleifenrumpf aufwändigere Analysen erfordert. In solchen Fällen bietet sich daher die Anwendung der Schleifentransformation *Loop-Unswitching* an, mit der die in einer Schleife enthaltenen *if*-Anweisungen aus der Schleife „herausgezogen“ werden. Dies darf allerdings nur dann vorgenommen werden, wenn die zu testende Bedingung *schleifeninvariant*, also unabhängig von den in der Schleife enthaltenen Anweisungen, ist. In Abb. 4.10 wird dies dadurch erreicht, indem die *if*-Anweisung in der innersten Schleife nach außen verschoben wird und der Schleifen-Kontrollcode dupliziert wird (s. Programmfragment `unswitching`). Dadurch wird die zuvor im Schleifenrumpf vorhandene *if*-Anweisung statt 40960-mal ($1024 \times 40 = 40960$) nun nur noch einmal ausgeführt, so dass ebenfalls positive Auswirkungen hinsichtlich der Ausführungszeit zu erwarten sind.

Leider lässt sich auch das erhaltene Programm noch nicht vektorisieren, weil mit `y[i]`

zweimal eine Zuweisung an ein Array-Element vorgenommen wird, dessen Adresse nicht in Abhängigkeit zur Schleifen-Indexvariablen der innersten Schleife steht. Mit Hilfe der Schleifentransformation *Loop-Interchange* bietet es sich darum an, die Position der Schleifen zu vertauschen. Das Ergebnis dieser Schleifentransformation, angewendet auf das Programmfragment `unswitching`, führt zu dem in Abb. 4.11 abgebildeten Programmcode `interchange`.

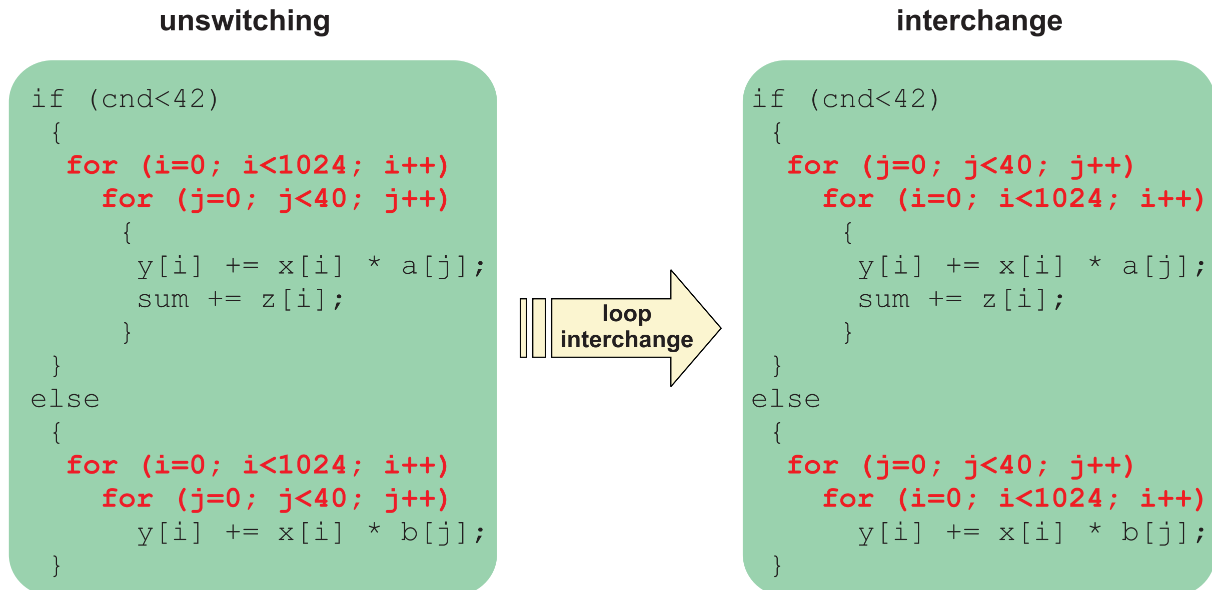


Abb. 4.11: Beispiel zur Anwendung von Loop-Interchange

Wie zu erkennen ist, bewirkt diese Transformation, dass nach der Anwendung jeweils in der innersten Schleife die Schleifen-Indexvariable i modifiziert wird. Dadurch kann nun zumindest die im *else*-Zweig des Programms enthaltene innerste Schleife mit der Anweisung `y[i] += x[i] * b[j]` vektorisiert werden. Dabei werden in jedem Schleifendurchlauf jeweils Gruppen von Daten der Arrays `x` und `y` aus dem Gruppenspeicher geladen. Der Array-Zugriff `b[j]` wird als Einzelwert erkannt und muss zunächst mittels eines Broadcast-Befehls – durchführbar innerhalb eines Prozessorzyklus – in ein komplettes Registerfile geschrieben werden, bevor eine Weiterverarbeitung erfolgen darf. Leider ist nach wie vor keine Vektorisierung der innersten Schleife des *if*-Zweigs möglich, weil mit der Anweisung `sum += z[i]` eine Zuweisung an eine skalare Variable vorhanden ist, die keine Vektorisierung zulässt.

Sind also in einer Schleife Anweisungen vorhanden, die nicht vektorisiert werden können, kann durch die Anwendung der Schleifentransformation *Loop-Split* eine Schleife derart in mehrere Schleifen aufgeteilt werden, dass zumindest ein Teil der Anweisungen vektorisiert werden kann. Im Gegensatz zur Schleifentransformation *Loop-Unswitching* führt dies nicht unittelbar zu einer geringeren Anzahl auszuführender Anweisungen, da alle Anweisungen nach wie vor mit derselben Häufigkeit ausgeführt werden. In dem in Abb. 4.12 gegebenen

Beispiel kann, nach der Durchführung des Loop-Splits auf die im *if*-Zweig enthaltene Schleife, die obere Schleife des Programmfragments `split` vektorisiert werden.

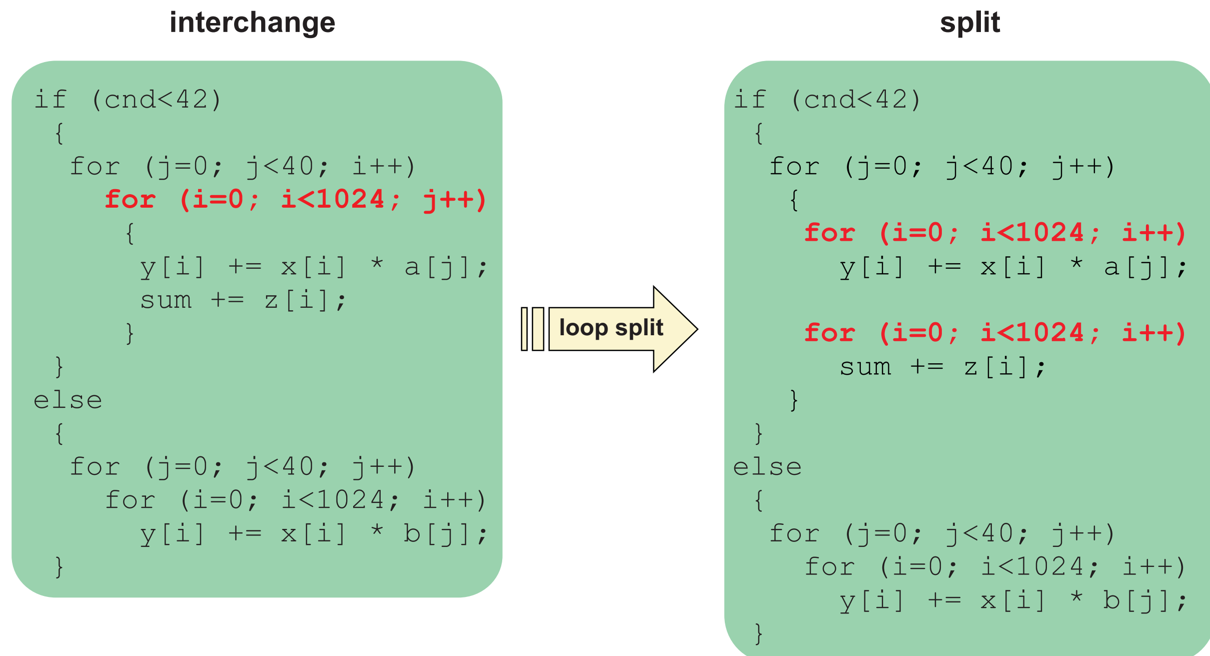


Abb. 4.12: Beispiel zur Anwendung von Loop-Split

Die nach der Anwendung von Loop-Split erhaltene zweite Schleife stellt eine *Reduktion* eines Arrays auf einen skalaren Wert dar. Zur Handhabung solcher Reduktionen kann durch die Anwendung der Transformation *Reduction-Recognition* wiederum ein Teil dieser Schleife vektorisiert werden. Dazu wird wie im Programmfragment `reduction_recognition` in Abb. 4.13 veranschaulicht, ein temporäres Array `tsum` eingeführt.

Das ursprüngliche Array `z` wird in einem ersten Schritt zunächst auf dieses kleinere Array reduziert. In einem abschließenden Schritt wird dann das erhaltene Array, dessen Größe o.B.d.A. der Anzahl der parallelen Datenpfade entspricht, letztendlich auf den skalaren Wert reduziert. Der Vorteil ist, dass nun zwar immer noch eine Schleife vorhanden ist, die nicht vektorisiert werden kann, diese allerdings nur noch 16-mal statt zuvor 1024-mal durchlaufen werden musss.

Die hier vorgestellten Schleifentransformationen stellen lediglich eine kleine Auswahl dar. Weitere Informationen zu diesem Thema können u.a. [BGS94] entnommen werden.

4.6.2 Überprüfung auf Vektorisierbarkeit

Bevor die Vektorisierung einer bestimmten Schleife vorgenommen werden kann, ist die Durchführung einer Reihe von Analysen erforderlich, auf die im Folgenden kurz eingegangen wird (vgl. [Hor01b]).

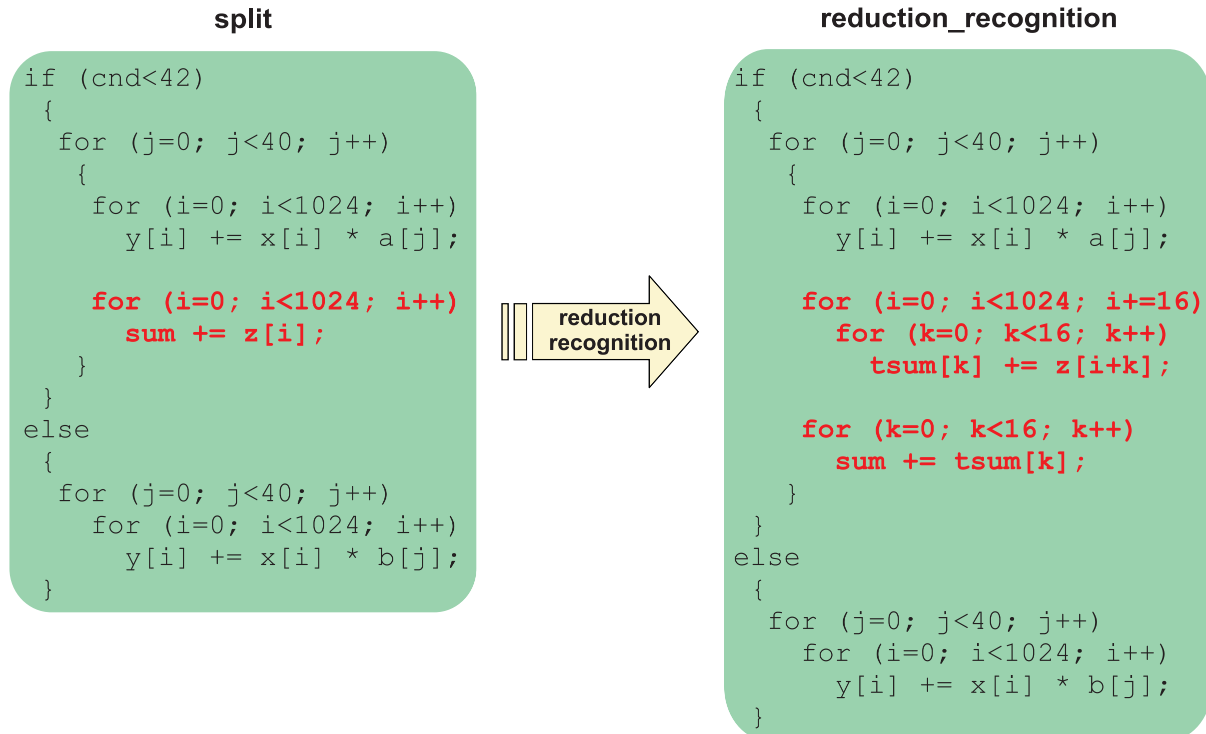


Abb. 4.13: Beispiel zur Anwendung von Reduction-Recognition

1. Überprüfung der ermittelten Schleifengrenzen und der Schrittweite.
   Eine Vektorisierung ist potentiell möglich, wenn die untere und obere Schleifengrenze durch die Anzahl der parallelen Verarbeitungseinheiten (beim M3-DSP: 16) teilbar ist und die Schrittweite 1 beträgt. Ist die Schrittweite z.B. 2, so darf nur jedes zweite Element eines bestimmten Arrays verändert werden, obwohl auf allen Datenpfaden simultan gearbeitet wird. Als Lösung bietet sich hier eine Maskierung der Datenpfade an, durch die entsprechende Datenpfade abgeschaltet werden können.

2. Erkennen von iterationsübergreifenden Datenabhängigkeiten.
   Im Gegensatz zu einer üblicherweise vorhandenen Datenabhängigkeitsanalyse auf Basisblock-Ebene, müssen zur Gewährleistung einer semantisch korrekten Vektorisierung Datenabhängigkeiten über Schleifengrenzen hinweg analysiert werden. Wird z.B. ein Wert eines Arrays in der i-ten Iteration verändert und in der $i+x$-ten Iteration verwendet, so ist eine Vektorisierung nicht möglich, wenn x kleiner als die Anzahl der parallelen Datenpfade ist, da in diesem Fall nicht gesichert ist, dass der zu verarbeitende Wert rechtzeitig geschrieben wurde. Offensichtlich spielt bei der Entscheidung, ob eine Vektorisierung hinsichtlich dieses Kriteriums möglich ist, die Laufrichtung (also das Vorzeichen der Schleifen-Schrittweite) eine wichtige Rolle und wird deswegen mit berücksichtigt. Des Weiteren wird in diesem Schritt getestet, ob die zu verarbeitenden Daten in zusammenhängenden Speicherbereichen (Gruppen) vorlie-

gen, oder vor der Verarbeitung zu einer neuen Gruppe zusammengesetzt (gepackt) werden müssen. Die Bereitstellung dieser Informationen erfordert die Durchführung einer Array-Datenflussanalyse. Dazu wird die von Duesterwald [DGS93] vorgestellte δ-Array-Datenflussanalyse eingesetzt, mit der eine iterationsübergreifende Analyse von Abhängigkeiten zwischen Array-Zugriffen in Schleifen möglich ist.

3. Analyse der Array-Indexterme.
   Aufgrund der Art und Weise in der auf ein bestimmtes Array zugegriffen wird, muss entschieden werden, ob diese vom Codegenerator unterstützt werden. Dies betrifft insbesondere das Packen und Entpacken von Daten. Hier kommt es dann im Wesentlichen darauf an, in wieweit der Codegenerator derartige Zugriffe unterstützt.

4. Zuweisungen an skalare Variablen.
   Bei der Verwendung von skalaren Variablen ist darauf zu achten, dass nur Zuweisungen an Hilfsvariablen vorgenommen werden. Bei Variablen, die noch außerhalb der Schleife Verwendung finden, ist eine Zuordnung eines Vektors zu einer skalaren Variablen (Reduktion) erforderlich. Als Beispiel kann hier die Berechnung der Vektorsumme gesehen werden, bei der alle Werte eines Arrays in einer skalaren Variablen aufsummiert werden (s. `sum` in Abb. 4.13 für ein Beispiel).

5. Überprüfung weiterer Abhängigkeiten.
   Neben den bereits erwähnten Bedingungen können in einer Schleife vorhandene Kontrollflussabhängigkeiten und Ein- und Ausgabeanweisungen eine Vektorisierung verhindern. Ergeben sich z.B. in Abhängigkeit der Kontrollflusswege unterschiedliche Zugriffe auf die Arrays, erschwert dies eine korrekte Vektorisierung immens und wird deswegen in unserem Fall nicht durchgeführt. Durch die Anwendung von Schleifentransformationen kann in solchen Fällen evtl. noch eine Vektorisierung erreicht werden.

Alle hier beschriebenen Analysen sind auf den GeLIR-Datenstrukturen implementiert und können auch zur Entwicklung anderer Back-Ends eingesetzt werden.

4.6.3 Ausnutzung spezieller Datentransfers

Unter der Annahme, dass in jeder Iteration die Schleifen-Indexvariable i das erste Element einer Gruppe bezeichnet, können Schleifenausdrücke wie `A[i] = B[i]` bei einem Speicherlayout, wie es in Abb. 4.14 gegeben ist, problemlos vektorisiert werden. Wie anhand des Speicherlayout zu erkennen ist, können dem Array `A` dann in drei Schleifendurchläufen gruppenweise die entsprechenden Daten von Array `B` zugewiesen werden. Die in der ersten Iteration relevanten Daten sind entsprechend gekennzeichnet.

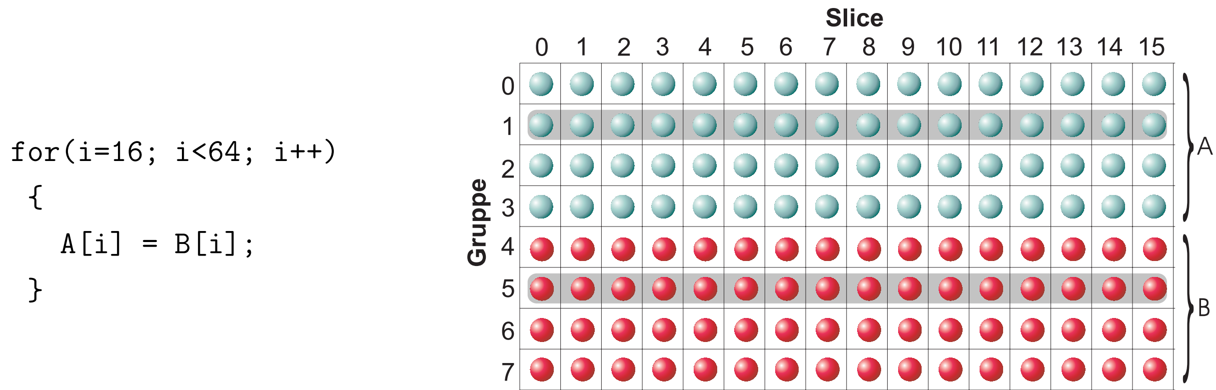


Abb. 4.14: Beispiel einer gruppenweisen Verarbeitung von Daten

Anders sieht es allerdings bei Schleifen mit Ausdrücken aus, bei denen die Daten nicht gruppenweise verarbeitet werden können, sondern wie im Beispiel von Abb. 4.15 zunächst aus mehreren Gruppen (hier zwei) zusammengesetzt werden müssen.

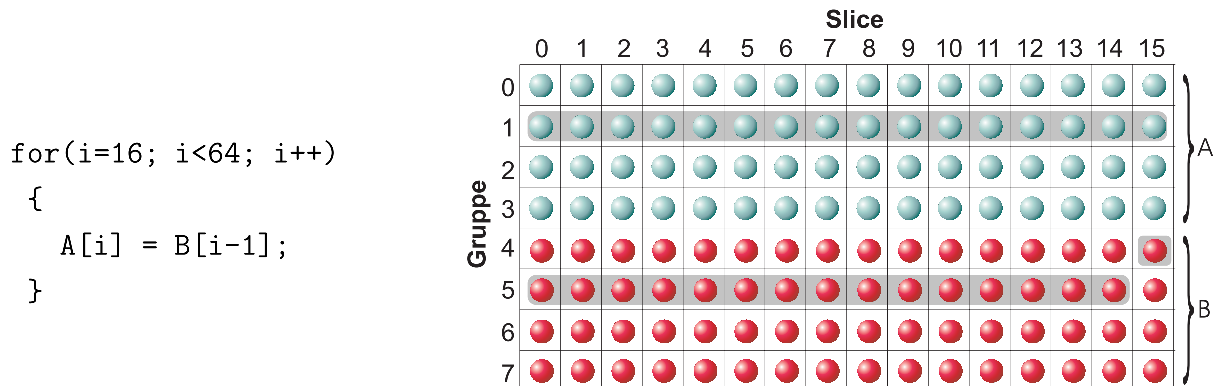


Abb. 4.15: Beispiel zur Verarbeitung von zusammengesetzten Gruppen

Um in einem solchen Fall dennoch eine Vektorisierung zu ermöglichen, wird zunächst die fünfte Gruppe in ein Registerfile geladen. Danach werden alle Daten in dem entsprechenden Registerfile mittels eines Zurich-Zip-Datentransfers [LBSL97, DF02] in einem Prozessorzyklus zyklisch um eine Position nach rechts verschoben und das noch fehlende Einzelelement der Gruppe vier an die erste Position des Registerfiles nachgeladen. Die dazu erforderlichen Operationen werden in den GeLIR-Code eingefügt.

Wenn n die Anzahl der parallelen Datenpfade, $cnst$ eine beliebige Konstante und i eine Schleifen-Indexvariable darstellt, können grundsätzlich die folgenden Ausdrücke vektorisiert werden:

- $A[i] = B[i]$
- $A[i] = B[i \;+/-\; n{*}cnst]$

- $A[i] = B[i \; +/- \; cnst]$
- $A[i] = B[cnst]$
- $A[i] = cnst$

Zu beachten ist, dass die Ausdrücke auf der rechten Seite der Gleichungen durch arithmetische Operatoren, für die eine Ausführung als SIMD-Operation möglich ist, beliebig miteinander kombiniert werden können. Dies bedeutet, dass z.B. auch Ausdrücke wie

$$A[i] = B[i] * C[i-1] + 42;$$

vektorisierbar sind.

4.6.4 Optimierte Anordnung von Arrays

Die Effektivität der Ausnutzung von SIMD-Operationen ist stark abhängig von der Lage der Daten im Speicher. So sind im günstigsten Fall die Daten bereits so im Speicher angeordnet, dass keine zusätzlichen Datentransfers erforderlich sind, um die Daten in den Gruppenregisterfiles geeignet anzuordnen. Im ungünstigsten Fall kann eine ungeeignete Anordnung der Daten die Ausführung von SIMD-Operationen verhindern. Gründe hierfür liegen u.a. im ungewollten Überschreiben von Daten oder in zu hohen Kosten zum Anordnen der zu verarbeitenden Daten.

Während skalare Variablen beliebig im Speicher angeordnet werden können, ist dies bei komplexen Datentypen wie Arrays nicht ohne weiteres möglich. Bei diesen wird im Allgemeinen davon ausgegangen, dass aufeinander folgende Array-Elemente auch in aufeinander folgenden Speicherzellen abgelegt werden. So müssten bei einer wahlfreien Anordnung der Elemente ebenfalls die entsprechenden Array-Zugriffsfunktionen angepasst werden. Da diese allerdings im Prinzip beliebig komplex werden können, stellt dies für den allgemeinen Fall eine nicht lösbare Aufgabe dar. So würde bereits eine Verteilung der Elemente eines Arrays, auf das in einer einfachen Schleife mittels einer Schleifen-Indexvariablen zugegriffen wird, eine äußerst schwierige Aufgabe darstellen.

Als ausschlaggebendes Kriterium für die Anordnung eines Arrays dienen die bei der Schleifenerkennung ermittelten Werte für die untere und obere Grenze (i_l und i_u) einer Schleifen-Indexvariablen i. Anhand dieser Werte werden bei n parallelen Datenpfaden die folgenden Fälle unterschieden, wobei o.B.d.A. jeweils davon ausgegangen wird, dass die Größe der verwendeten Arrays mit der Anzahl der Schleifendurchläufe übereinstimmt.

- $(i_l \bmod n) = 0 \;\; \wedge \;\; (i_u \bmod n) = 0$ Wenn die untere und obere Schleifengrenze ohne Rest durch die Anzahl der Datenpfade n teilbar ist, ist keine Bestimmung eines speziellen Speicherlayouts der Arrays erforderlich, da die verwendeten Arrays

die Gruppen des Gruppenspeichers vollständig auffüllen. Im Beispiel von Abbildung 4.16 werden bei einer Vektorisierung der gegebenen Schleife in jeder Iteration jeweils die Array-Elemente von genau einer Gruppe verarbeitet.

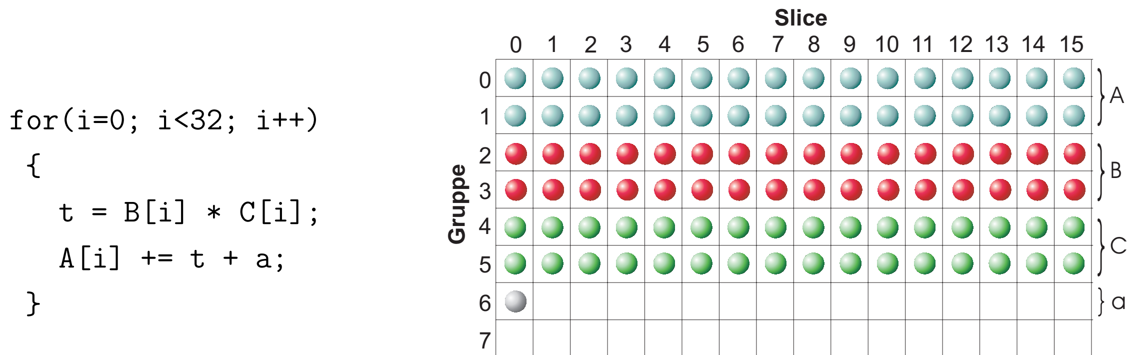


Abb. 4.16: Einfaches Speicherlayout

- $(i_l \bmod n) = 0 \;\wedge\; (i_u \bmod n) \neq 0$

  Ist die obere Schleifengrenze i_u nicht ohne Rest durch die Anzahl der Datenpfade teilbar, so darf nur dann eine Vektorisierung erfolgen, wenn durch eine geeignete Ausrichtung der Arrays sichergestellt ist, dass keine noch benötigten Daten bei Verarbeitung der letzten Gruppe überschrieben werden. In Abbildung 4.17 wird dies berücksichtigt, indem für die Arrays A, B und C die verbliebenen Speicherplätze der letzten Gruppe jeweils unbenutzt bleiben.

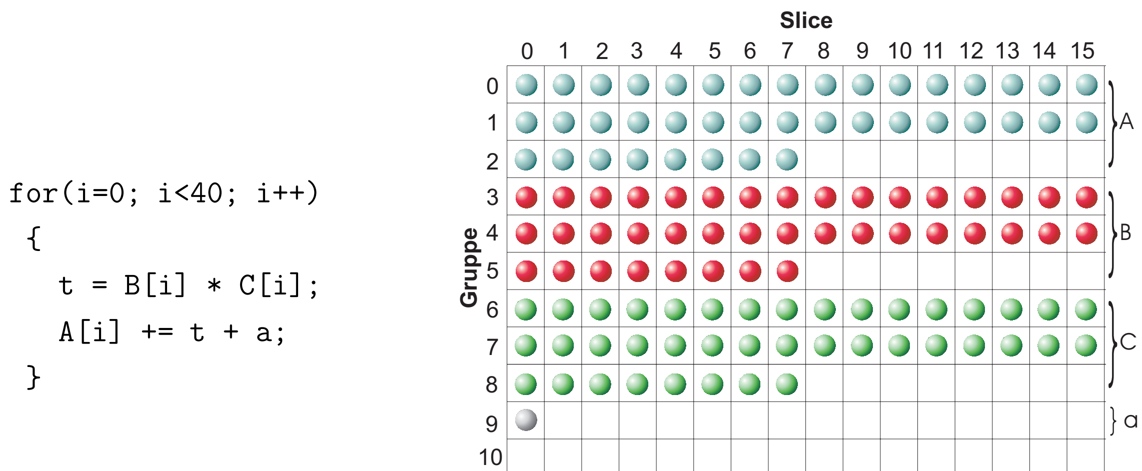


Abb. 4.17: Speicherlayout mit einfacher Ausrichtung der Arrays

Obwohl bei einer Vektorisierung der Schleife in der letzten (dritten) Iteration zwar auf 16 Datenpfaden gerechnet und eine vollständige Gruppe in den Speicher geschrieben wird, treten bei dieser Anordnung keine ungewollten Datenverluste auf.

- $(i_l \bmod n) \neq 0 \;\wedge\; (i_u \bmod n) \neq 0$

  Wenn die untere Schleifengrenze nicht ohne Rest durch die Anzahl der Datenpfade teilbar ist, kann in diesem Fall bei einer einfachen Ausrichtung der Arrays nicht ohne weiteres eine Vektorisierung durchgeführt werden (s. auch Abb. 4.18). Da die untere Schleifengrenze gleich zehn ist, müssten bei einer Vektorisierung der Schleife in der ersten Iteration die Ergebnisse für `A[10]` bis `A[25]` berechnet werden. Dazu wäre allerdings bei einer einfachen Ausrichtung der Arrays, wie in Abb. 4.17, ein großer Overhead zum Packen und Entpacken der zu verarbeitenden und speichernden Gruppen erforderlich. Durch eine Expansion der Arrays wie in Abb. 4.18 dargestellt, kann dieser Overhead vermieden werden, wenn zusätzlich die Schleifen-Indexvariablen in jeder Iteration statt um 16, um zehn erhöht wird.

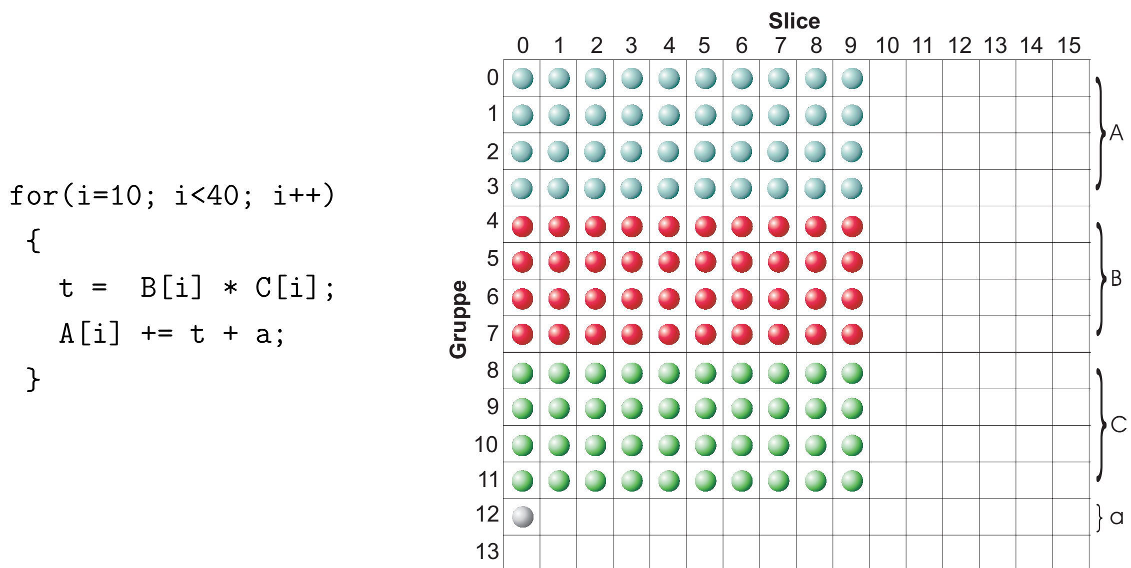


Abb. 4.18: Speicherlayout mit optimierter Ausrichtung der Arrays

  Aufgrund der Lücken im Speicher besteht allerdings nun das Problem, dass bei SISD-Zugriffen außerhalb dieser Schleife eine Anpassung der Zugriffsfunktion durchgeführt werden muss, um nach wie vor den Zugriff auf das richtige Array-Element zu gewährleisten.

4.7 Optimierte Anordnung skalarer Variablen

Der Onchip-Gruppenspeicher der Prozessoren der M3-Plattform ermöglicht eine effiziente Versorgung der parallelen Datenpfade mit Daten, indem mit jedem Speicherzugriff jeweils eine Gruppe von Daten parallel geladen wird. Wenn die Parallelität der Datenpfade allerdings nicht ausgenutzt werden kann, ist es erforderlich, im SISD-Modus nacheinander auf einzelne (in Registern vorliegende) Daten zuzugreifen. Befinden sich nun aufeinander

folgend zu verarbeitende Daten in unterschiedlichen Gruppen, muss jeweils die entsprechende Gruppe aus dem Speicher geladen werden, um auf den benötigten Wert zugreifen zu können. Das in Abb. 4.19 gegebene Beispiel verdeutlicht, dass dies einen beträchtlichen Overhead an Speicherzugriffen bedeutet, der sich durch eine geschickte Gruppierung der Variablen erheblich reduzieren lässt.

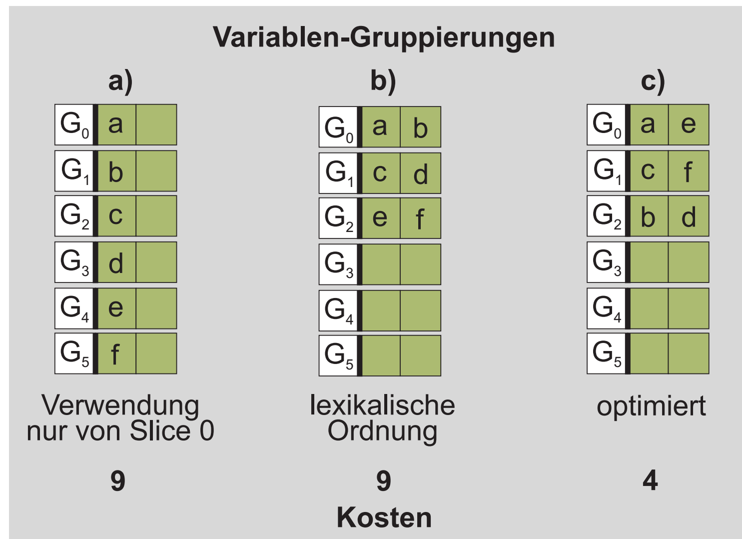


Abb. 4.19: Auswirkungen unterschiedlicher Variablen-Gruppierungen

Für eine Variablen-Zugriffssequenz $S_V = (a, e, a, c, f, c, b, d, a)$ sind in Abb. 4.19 drei unterschiedliche Variablen-Gruppierungen a) bis c) mit resultierenden Kosten gegeben[1]. Die Ermittlung der Kosten erfolgte in diesem Beispiel, indem für jeden Zugriff auf eine Variable, die sich in einer anderen Gruppe als die vorherige befindet, die Gesamtkosten um eins erhöht wurden. Dabei sollte berücksichtigt werden, dass der Zugriff auf die erste Variable a ebenfalls einen Speicherzugriff verursacht. Für die Variablen-Gruppierungen a) und b) ergeben sich demnach Gesamtkosten in Höhe von neun Speicherzugriffen, da mit jedem Variablenzugriff der Sequenz eine andere Gruppe betroffen ist. Die optimierte Gruppierung c) führt hier zu einer drastischen Reduzierung der Kosten auf vier Speicherzugriffe, weil mehrfach auf Variablen einer Gruppe zugegriffen wird, die sich bereits in einem Registerfile befindet.

Offensichtlich besteht bei einer Abarbeitung im SISD-Modus ein enger Zusammenhang zwischen der Lage der skalaren Variablen im Gruppenspeicher und der Anzahl auszuführender Speicherzugriffe. Das Ziel der hier beschriebenen Optimierung besteht deswegen in der Ermittlung einer optimierten Anordnung der in einem Programm verwendeten skalaren Variablen zu Gruppen, so dass die Anzahl der erforderlichen energieintensiven

[1] In diesem Beispiel nehmen wir der Einfachheit halber eine Architektur mit einer Gruppenbreite von zwei an.

Speicherzugriffe reduziert wird und dadurch bedingt weniger Operationen ausgeführt werden müssen. Neben einer Reduzierung der Ausführungszeit ist ebenfalls auch eine Reduzierung des Energieverbrauchs zu erwarten. Da bei Prozessoren mit mehreren verteilten Speicherbänken ebenfalls eine Aufteilung von Variablen auf die einzelnen Speicher vorgenommen werden kann, ist eine entsprechende Verwendung der im Folgenden beschriebenen Techniken nicht nur auf Prozessoren mit Gruppenspeichern wie denen der M3-Prozessoren beschränkt.

Im folgenden Abschnitt wird zunächst eine formale Grundlage des zu lösenden Optimierungsproblems geschaffen. Danach erfolgt eine Beschreibung des Lösungsansatzes und der Integration in den Compilierungsprozess.

4.7.1 Problemdefinition

Üblicherweise besteht die Aufgabe der Adresszuweisung in der Zuweisung der verwendeten Variablen zu Adressen im Speicher. Aufgrund des Gruppenspeichers ist in unserem Fall allerdings eine weitere Phase der Adresszuweisung erforderlich. So wird die Aufgabe der Adresszuweisung, wie in Abb. 4.20 verdeutlicht, in die *horizontale* und *vertikale* Adresszuweisung unterteilt.

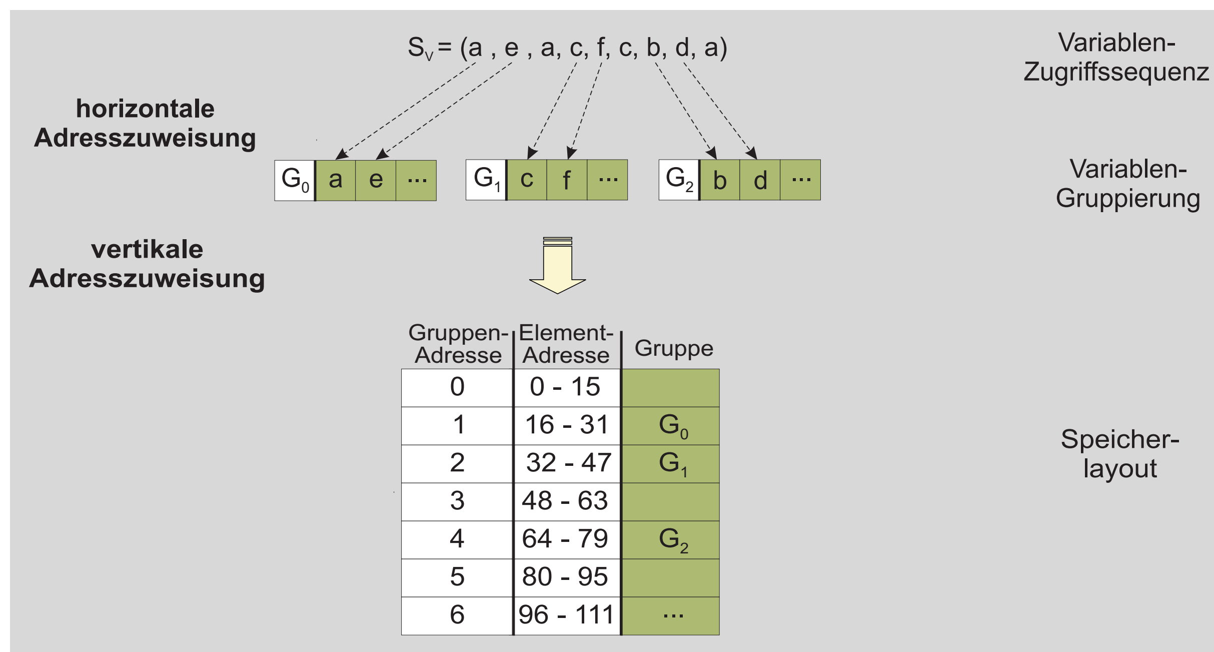


Abb. 4.20: Horizontale und vertikale Adresszuweisung

Die Aufgabe der horizontalen Adresszuweisung besteht in der Zuordnung aller verwendeten Variablen zu Gruppen. Erst danach werden im Zuge der vertikalen Adresszuweisung, mittels des in Abschnitt 3.6 beschriebenen Verfahrens, den sich ergebenden Gruppen Adressen zugewiesen.

Wie wir zuvor gesehen haben, sind zur Ermittlung einer geeigneten Anordnung der Variablen zu Gruppen, Informationen über die Zugriffsreihenfolge von Variablen erforderlich, die der Optimierung in Form einer Variablen-Zugriffssequenz zur Verfügung gestellt werden müssen:

Definition 4.1 (Variablen-Zugriffssequenz) *Wenn* $V = \{v_1, v_2, \ldots, v_n\}$ *eine Menge von skalaren Variablen darstellt, besteht eine Variablen-Zugriffssequenz* $S_V = (s_1, s_2, \ldots, s_m)$ *aus einer Sequenz von Variablen* $s \in V$*, auf die während einer Programmausführung in dieser zeitlichen Reihenfolge lesend oder/und schreibend zugegriffen wird.*

In der in Abb. 4.20 dargestellten Variablen-Zugriffssequenz erfolgen Zugriffe auf die Variablen a, b, c, d und e. Das Ergebnis der horizontalen Adresszuweisung könnte in der dargestellten Zuweisung der Variablen zu den Gruppen G_0, G_1 und G_2 bestehen. Da zu Beginn der Sequenz auf die Variablen a und e zweimal aufeinander folgend zugegriffen wird, erscheint die Zuweisung dieser Variablen zur selben Gruppe eine gute Wahl. In diesem Fall ist es also möglich, einmal die Gruppe G_0 aus dem Speicher in ein Registerfile zu laden und die danach folgenden Zugriffe – ohne erneuten Speicherzugriff – direkt auf dem Registerfile durchzuführen. Dazu bietet es sich an, Variablen, die in einer Variablen-Zugriffssequenz an aufeinander folgenden Positionen stehen (also *Nachbarn* sind), möglichst denselben Gruppen zuzuweisen.

Definition 4.2 (Nachbar) *Zwei Variablen* v *und* w *sind genau dann Nachbarn in der Variablen-Zugriffssequenz* $S_V = (s_1, s_2, \ldots, s_n)$*, wenn es mindestens ein* $i \in \{1, \ldots, n-1\}$ *in* S_V *gibt, für das* s_i *und* s_{i+1} *Zugriffe auf die Variablen* v *und* w *darstellen. Die Reihenfolge der Zugriffe spielt dabei keine Rolle.*

Aus dieser Definition ergibt sich unmittelbar der Begriff der *Nachbarschaftsbeziehung*.

Definition 4.3 (Nachbarschaftsbeziehung) *Zwischen zwei Variablen* v *und* w *einer Variablen-Zugriffssequenz* S_V *besteht eine Nachbarschaftsbeziehung, wenn diese Nachbarn sind.*

In Abhängigkeit davon, ob zwei Variablen derselben Gruppe zugewiesen werden, besteht zwischen diesen Variablen entweder eine *erfüllte* oder eine *unerfüllte Nachbarschaftsbeziehung.*

Definition 4.4 (Erfüllte und unerfüllte Nachbarschaftsbeziehung) *Wenn zwischen zwei Variablen* v *und* w *eine Nachbarschaftsbeziehung existiert und diese derselben Gruppe angehören, besteht zwischen den Variablen* v *und* w *eine erfüllte Nachbarschaftsbeziehung. Gehören die Variablen* v *und* w *unterschiedlichen Gruppen an, besteht eine unerfüllte Nachbarschaftsbeziehung.*

In Analogie zu [LDK+95] verwenden wir die folgende Graph-Repräsentation der Variablen-Zugriffssequenz:

Definition 4.5 (Variablen-Zugriffsgraph) *Ein Variablen-Zugriffsgraph* $G_V = (V, E)$ *ist ein ungerichteter Graph mit einer Menge von Knoten* $V = \{v_1, v_2, \ldots, v_n\}$*, auf die in einer Variablen-Zugriffssequenz* S_V *zugegriffen wird. Die Kantenmenge* E *enthält genau dann eine Kante* e_{ij} *zwischen zwei Knoten* v_i *und* v_j*, wenn zwischen diesen Variablen eine Nachbarschaftsbeziehung bezüglich* S_V *besteht. Das Gewicht* w_{ij} *einer Kante* e_{ij} *ergibt sich durch die Anzahl der aufeinander folgenden Zugriffe der Variablen* v_i *und* v_j *in* S_V*.*

Zusätzlich wird die Menge der Kanten des Variablen-Zugriffsgraphen in *externe* und *interne* Kanten eingeteilt:

Definition 4.6 (Externe und interne Kanten) *Eine Kante* e_{ij} *zwischen zwei Knoten* v_i *und* v_j *wird als externe Kante bezeichnet, wenn zwischen den Knoten* v_i *und* v_j *eine Nachbarschaftsbeziehung besteht und diese Knoten unterschiedlichen Gruppen angehören. Eine Kante wird stattdessen als interne Kante bezeichnet, wenn diese Knoten derselben Gruppe zugewiesen wurden.*

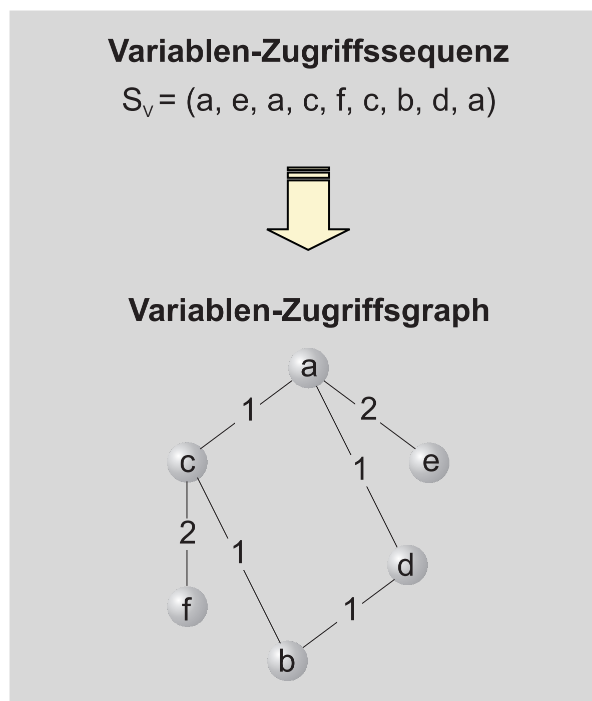


Abb. 4.21: Darstellung einer Variablen-Zugriffssequenz als Variablen-Zugriffsgraph

In Abb. 4.21 ist für die bereits betrachtete Variablen-Zugriffssequenz der zugehörige Variablen-Zugriffsgraph abgebildet. Nachdem in diesem Abschnitt die erforderlichen Grundlagen geschaffen worden sind, wird im nächsten Abschnitt der verfolgte Lösungsansatz beschrieben.

4.7.2 Lösungsansatz

Da Speicherzugriffe immer auf Gruppen bezogen sind, entspricht die Aufgabe der horizontalen Adresszuweisung einer *Partitionierung* der Variablen. Offensichtlich kann die Anzahl der Speicherzugriffe dadurch minimiert werden, indem durch eine geeignete Aufteilung (oder Partitionierung) der Variablen zu Gruppen, die Summe der Kantengewichte der externen Kanten minimiert wird (*Graph-Partitionierungsproblem*). Dabei gilt es, die folgenden Randbedingungen zu beachten:

- Die Anzahl der Variablen, die einer Partition zugeordnet werden dürfen, ist durch die Gruppenbreite beschränkt.
- Die Anzahl der resultierenden Gruppen ist unbekannt, da die Gruppen nicht zwangsläufig voll ausgelastet sein müssen.
- Jede Variable muss genau einer Gruppe zugewiesen werden.
- Die Position einer Variablen innerhalb einer Gruppe spielt keine Rolle.

In Abb. 4.22 sind für drei mögliche Graph-Partitionierungen die dazugehörigen Variablen-Gruppierungen angegeben. Offensichtlich stellen die Partitionen a) und b) das Ergebnis schlechter Partionierungsverfahren dar, da in keinem dieser Fälle eine Nachbarschaftsbeziehung ausgenutzt wird und dadurch nur externe Kanten (siehe fett gedruckte Kanten) existieren. Wie zu erkennen ist, entsprechen die aus den Partitionierungen resultierenden Variablen-Gruppierungen denen aus Abb. 4.19.

Da die Durchführung einer derartigen Partitionierung die Lösung eines NP-harten Optimierungsproblems bedeutet [GJ79], wurden im Rahmen der Diplomarbeit von Kottmann [Kot00] einige Partitionierungsverfahren implementiert und hinsichtlich ihrer Güte gegenübergestellt. Darunter befinden sich einfache heuristische Partitionierungsverfahren, der häufig zur Partitionierung verwendete *Kernighan-Lin-Algorithmus* [KL70] und ein Partitionierungsverfahren auf Basis eines genetischen Algorithmus. Die Heuristiken stellen dabei polynomielle Verfahren auf Basis eines Kruskal-Algorithmus [Kru56] dar, die allerdings im Vergleich zu den anderen Verfahren häufig unbefriedigende Ergebnisse liefern. Der Kernighan-Lin-Algorithmus führte hingegen bereits zu sehr guten Ergebnissen, wobei jedoch keine polynomielle Laufzeit dieses Verfahrens garantiert werden kann. Eine Validierung hat ergeben, dass insbesondere für größere Variablenmengen durch das genetische Partitionierungsverfahren bessere Ergebnisse erzielt werden. Die Einsparungen liegen dabei zwischen 10% und 35% gegenüber den Kruskal-Varianten und zwischen 3% und 10% gegenüber dem Kernighan-Lin-Algorithmus. Da eine Validierung dieser Verfahren klare Vorteile zugunsten des genetische Partitionierungsverfahrens ergab, wird die

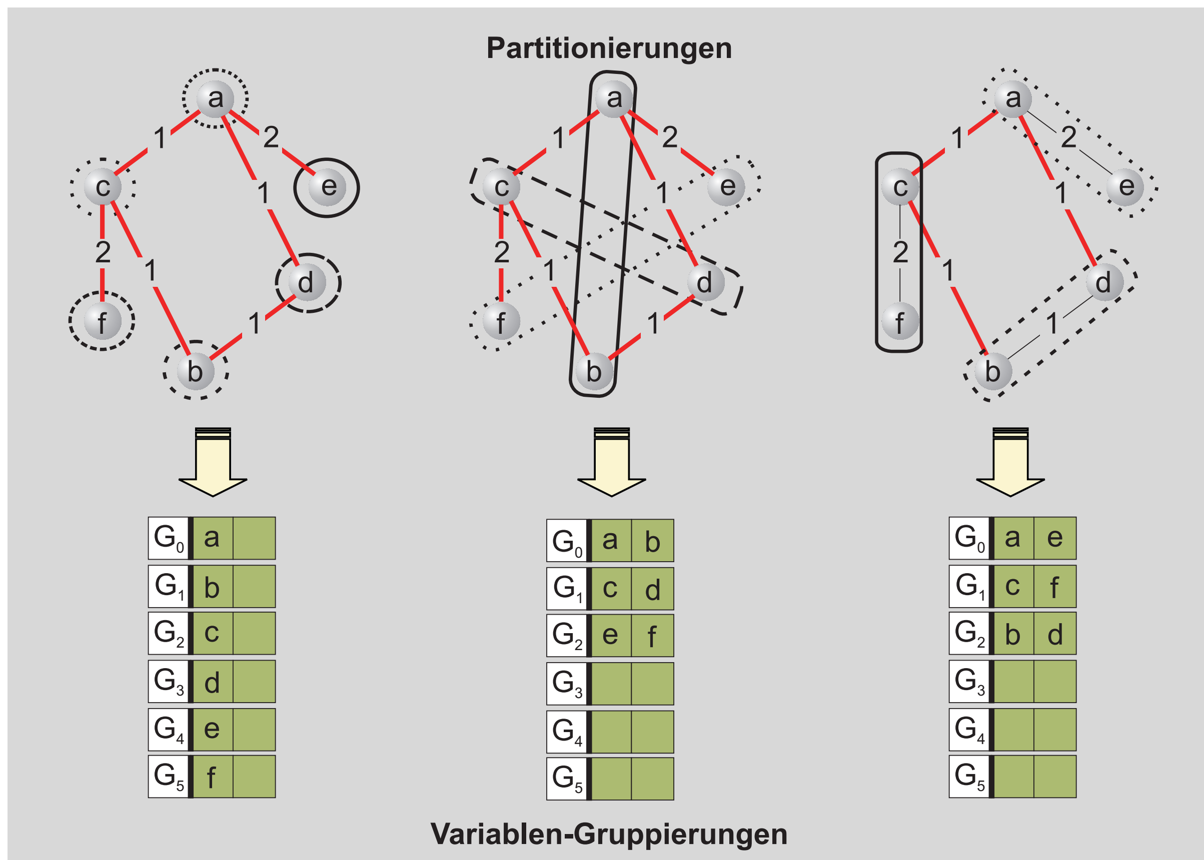


Abb. 4.22: Graph-Partitionierungen und resultierende Variablen-Gruppierungen

Aufteilung der Variablen zu Gruppen in diesem Compiler auf Basis des genetischen Algorithmus durchgeführt. Detaillierte Informationen zur Realisierung des genetischen Partitionierungsverfahrens können [Kot00, LKB+01] entnommen werden.

4.7.3 Integration in den Compilierungsprozess

Da die Zugriffsreihenfolge der Variablen stark vom generierten Code abhängt, ist vor Durchführung der Optimierung ein Codegenierungs-Durchlauf erforderlich, bei dem die Teilaufgaben der Codegenerierung CS, IA und RA durchgeführt werden. Hierbei wird davon ausgegangen, dass in jede Gruppe nur eine Variable aufgenommen werden kann. Das sich hieraus ergebende beste Individuum wird ermittelt und gespeichert. Zusätzlich werden von dieser Lösung zwei Variablen-Zugriffssequenzen erzeugt, in denen (Speicher-)Zugriffe auf globale und lokale Variablen getrennt voneinander aufgeführt werden. Diese Unterscheidung ist erforderlich, da lokale und globale Variablen unterschiedlichen Speicherbereichen und damit nicht denselben Gruppen zugewiesen werden dürfen. Des Weiteren ergibt sich bei der Partitionierung der lokalen Variablen gegenüber den globalen Variablen zusätzliches Optimierungspotential. So kann aufgrund der begrenzten Lebensdauer

lokaler Variablen derselbe Speicherplatz mehreren Variablen zugewiesen werden, sofern sich ihre Lebensbereiche nicht überschneiden. Mit Hilfe des Partitionierungsverfahrens wird für diese Variablenzugriffssequenzen eine optimierte Variablen-Gruppierung ermittelt. Diese dienen dann als Eingabe für einen erneuten Codegenerierungs-Durchlauf, der in der Initialisierungsphase mit Lösungen der zuvor ermittelten besten Lösung initialisiert wird.

4.8 Bewertung

In diesem Abschnitt erfolgt eine Bewertung der zuvor vorgestellten Optimierungen anhand einiger Testroutinen. Dazu werden im folgenden Abschnitt zunächst Ergebnisse der Vektorisierung, in Verbindung mit der Optimierung zur Ausnutzung der Zero-Overhead Hardware-Loops, vorgestellt. Des Weiteren wird der Einfluss von Schleifentransformationen auf die Codequalität anhand des in Abschnitt 4.6.1 betrachteten Beispielprogramms demonstriert. Danach erfolgt eine Bewertung der Optimierung für eine effektive Ausnutzung der SIMD-Speicherzugriffe im SISD-Modus.

4.8.1 Vektorisierung

Zur Beurteilung der Effektivität der Optimierung zur Vektorisierung von Schleifen werden in diesem Abschnitt die Ergebnisse einiger Compiler-Varianten gegenübergestellt. Dabei soll durch das Ein- bzw. Ausschalten von Optimierungen ein direkter Vergleich der Auswirkungen der jeweiligen Optimierungen auf die Codequalität ermöglicht werden. Im einzelnen werden die Einflüsse der folgenden Optimierungen untersucht, die alle die volle Breite des Gruppenspeichers ausnutzen:

- `SISD`
  Durchführung einer Codegenerierung im Einstreifen-Modus.
- `SIMD`
  Durchführung der Optimierung zur Vektorisierung von Schleifen.
- `seq`
  Statt der standardmäßig durchgeführten Kompaktierung von Maschinenoperationen (s. Kapitel 3) wird lediglich sequentieller Code erzeugt. Hiermit soll zum einen der Nutzen einer Kompaktierung gegenüber einer rein sequentiellen Codeerzeugung aufgezeigt werden und zum anderen auch das Potential zur Verbesserung der Codequalität durch eine SIMD-Ausführung gegenüber dem einer Kompaktierung.

- `ZOL`
  Hiermit werden Verfahren gekennzeichnet, die zusätzlich die Optimierung zur Ausnutzung von Zero-Overhead Hardware-Loops ausführen. Auch hier soll durch einen Vergleich mit den entsprechenden Varianten ohne Berücksichtigung dieser Optimierung, das Potential zur Verbesserung aufgezeigt werden. Des Weiteren soll gezeigt werden, dass diese Optimierung eine sehr gute Ergänzung zur Vektorisierung von Schleifen darstellt.

Bei der mit `SISD+ZOL` bezeichneten Compiler-Variante wird also eine Codegenerierung im Einstreifen-Modus unter Ausnutzung von Zero-Overhead Hardware-Loops durchgeführt, wobei zusätzlich eine Kompaktierung der Maschinenoperationen erfolgt. Da in diesem Abschnitt in erster Linie Aussagen über die Effektivität der SIMD-Optimierung zur Ausnutzung der parallelen Datenpfade gemacht werden sollen, werden alle Ergebnisse in Relation zu der Variante `SISD+ZOL` ($\widehat{=}$ 100%) gesetzt. Als Testroutinen dienen dabei die folgenden drei Schleifen:

```
for(i=0; i<1024; i++)
 {
   A[i] = B[i] + 2;
 }
```

Schleife 1

```
for(i=0; i<1024; i++)
 {
   t = B[i] * C[i];
   A[i] += t + a;
 }
```

Schleife 2

```
for(i=0; i<1022; i++)
 {
   t = B[i+1] * C[i+2];
   A[i] += t + 2;
 }
```

Schleife 3

In den Abbildungen 4.23 bis 4.25 werden Ergebnisse bezüglich Ausführungszeit, Energieverbrauch und durchschnittlicher Leistungsaufnahme[2] für diese drei Testroutinen vorgestellt. In Abb. 4.23 sind zunächst die Ergebnisse bezüglich der Ausführungszeit dargestellt.

Es zeigt sich, dass durch einen Verzicht auf eine Kompaktierung und die Ausnutzung von Zero-Overhead Hardware-Loops (s. `SISD+seq`) die Ausführungszeit gegenüber der Variante `SISD+ZOL` im Durchschnitt um 97% höher ist. Bei einer zusätzlichen Berücksichtigung von Zero-Overhead Hardware-Loops (s. `SISD+ZOL+seq`) können bereits deutliche Verbesserungen der Codequalität erzielt werden. Der Anteil der durch eine Kompaktierung erzielten Verbesserungen beträgt für diese Testroutinen im Durchschnitt 52% (s. `SISD+ZOL+seq`). Die Durchführung einer Vektorisierung der Schleifen (s. `SIMD`) ohne eine Ausnutzung von Zero-Overhead Hardware-Loops führt bereits zu einer drastischen Reduzierung der Ausführungszeit auf durchschnittlich 9%. Mit einer zusätzlichen Umsetzung der Optimierung der Zero-Overhead Hardware-Loops kann die Ausführungszeit für

[2] Im Gegensatz zum Energieverbrauch wird bei der Leistungsaufnahme die Anzahl der Prozessorzyklen des Programms nicht mitberücksichtigt.

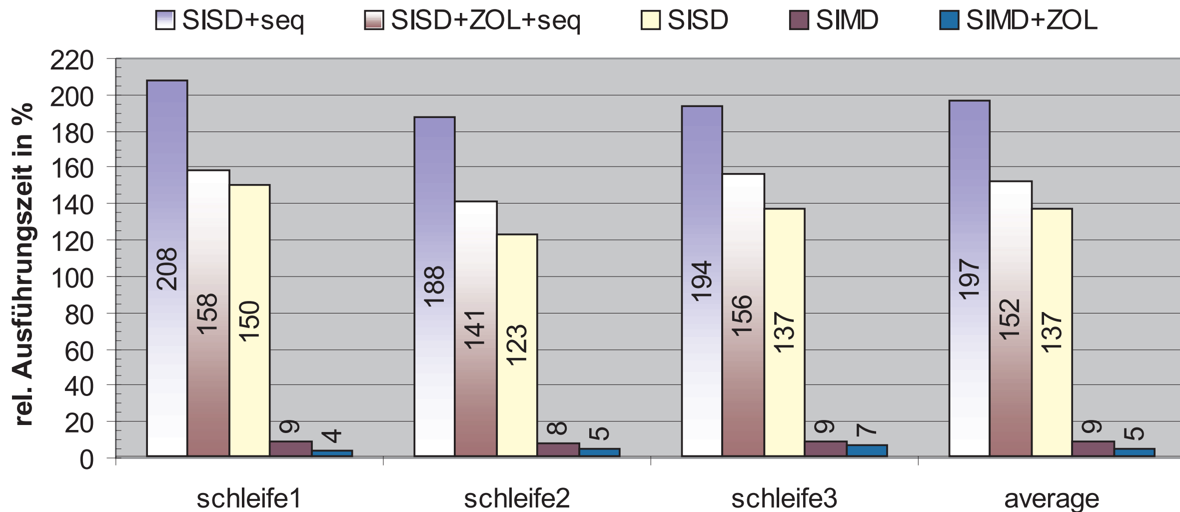


Abb. 4.23: Vektorisierung Ergebnisse: Ausführungszeit (100% $\widehat{=}$ `SISD+ZOL`)

diese Testroutinen letztlich auf durchschnittlich 5% reduziert werden, was einer Erhöhung der Ausführungsgeschwindigkeit um den Faktor 20 entspricht.

Die in Abb. 4.24 dargestellten Ergebnisse bezüglich des Energieverbrauchs weisen dieselbe Tendenz auf, wie die zuvor dargestellten. Allerdings ist hier zu beobachten, dass die erzielten Einsparungen im Vergleich zur Reduzierung der Ausführungszeit geringer ausfallen. Beispielsweise beträgt der durchschnittliche Overhead bei einer sequentiellen Ausführung (s. Variante `SISD+ZOL+seq`) nun lediglich 31% statt 52% bezüglich der Ausführungszeit.

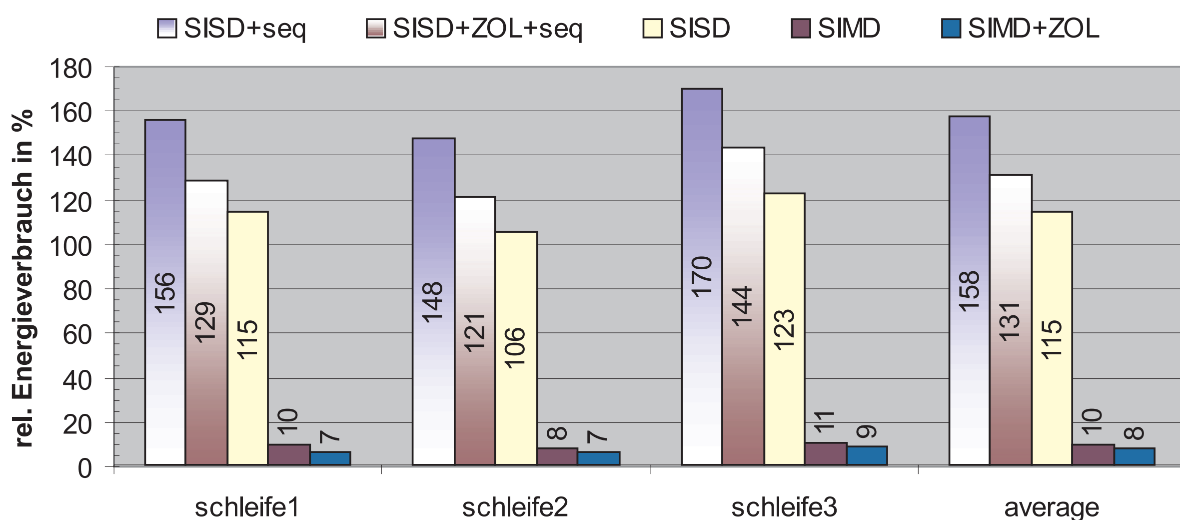


Abb. 4.24: Vektorisierung Ergebnisse: Energieverbrauch (100% $\widehat{=}$ `SISD+ZOL`)

Ähnliches lässt sich bei Betrachtung der SIMD-Varianten beobachten. Die geringeren Einsparungen hinsichtlich des Energieverbrauchs lassen sich dadurch erklären, dass zur Ausnutzung der Parallelität mit der Ausführung einer Maschineninstruktion im Vergleich zu einer SISD-Ausführung mehr Energie aufgewendet werden muss. So können mit einer SIMD-Operation 16 Operationen in einem Zyklus ausgeführt werden, was ein Nutzenverhältnis zwischen SISD und SIMD von 1:16 bedeutet. Bezüglich des Energieverbrauchs ist das Nutzenverhältnis aufgrund des vier- bis fünfmal höheren Energiebedarfs bei Ausführung einer SIMD-Operation jedoch mit ca. 1:4 wesentlich geringer. Dies belegen auch die in Abb. 4.25 dargestellten Ergebnisse bezüglich der durchschnittlichen Leistungsaufnahme. Wie zu erwarten zeigt sich, dass bei der Ausführung von SIMD-Operationen die deutlich höchste durchschnittliche Leistungsaufnahme zu verzeichnen ist. Da z.B. die durchschnittliche Leistungsaufnahme der Variante `SIMD+ZOL` im Vergleich zu der Referenz-Variante `SISD+ZOL` um durchschnittlich 47% höher ist, müssen die Reduzierungen des Energieverbrauchs gegenüber denen der Ausführungszeit auch geringer ausfallen.

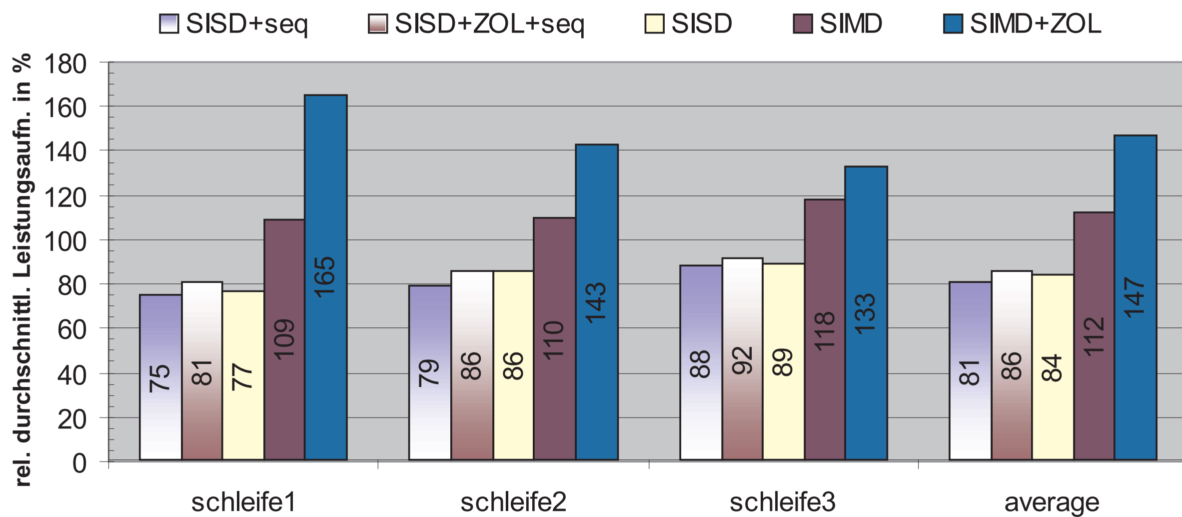


Abb. 4.25: Vektorisierung Ergebnisse: Durchschnittliche Leistungsaufnahme (100% $\widehat{=}$ `SISD+ZOL`)

Um den möglichen Einfluss von Schleifentransformationen auf die Codequalität aufzuzeigen, werden in Abb. 4.26 Ergebnisse für die in Abschnitt 4.6.1 vorgestellten Schleifentransformationen präsentiert. Dabei wurde die Compiler-Variante `SIMD+ZOL` auf das in Abschnitt 4.6.1 angegebene Originalprogramm `original` sowie auf die durch Schleifentransformationen entstandenen Programme `unswitching`, `interchange`, `split` und `reduction_recognition` angewendet. Alle in Abb. 4.26 dargestellten Ergebnisse sind in Relation zum ursprünglichen Programm `original` gesetzt.

Wie zu erwarten, wird durch die Anwendung der Schleifentransformation *Loop-Unswitching* eine Reduzierung der Ausführungszeit erzielt, da die in der innersten Schleife

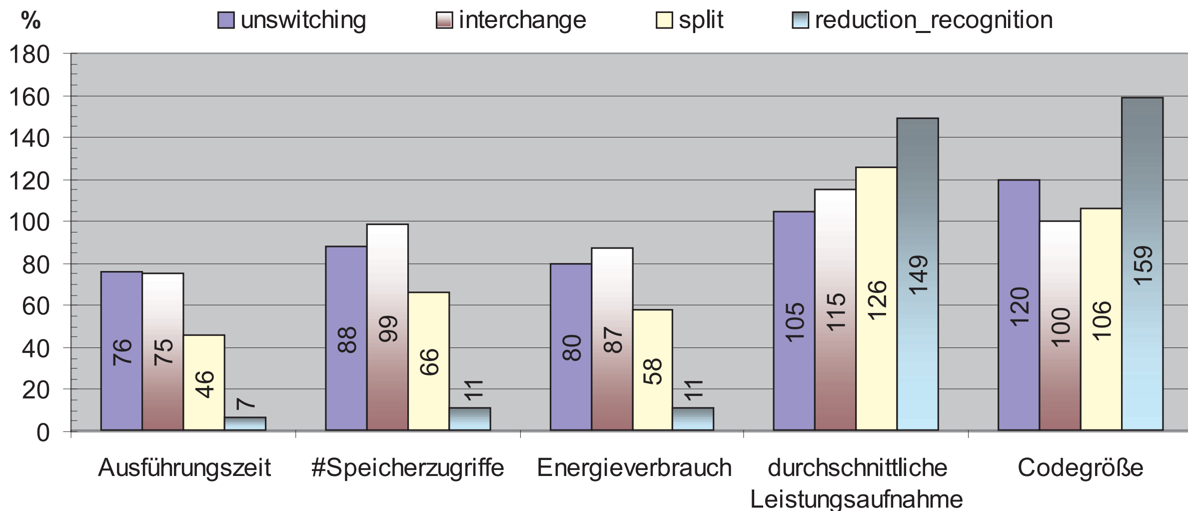


Abb. 4.26: Schleifentransformationen zur effektiven Ausnutzung von SIMD-Operationen (100% ≙ `original`)

vorhandene *if*-Anweisung statt 40960-mal nun nur noch einmal ausgeführt werden muss (s. Programm `unswitching` in Abb. 4.10). Dies wirkt sich ebenfalls positiv auf die Anzahl der Speicherzugriffe und den Energieverbrauch aus. Während lediglich eine geringe Erhöhung der durchschnittlichen Leistungsaufnahme zu verzeichnen ist, erhöht sich die Codegröße durch diese Optimierung um 20%. Mit der Durchführung der Schleifentransformation *Loop-Interchange* kann im Programm `interchange` (s. Abb. 4.11) die Schleife im *else*-Zweig vektorisiert werden. Dies führt zwar gegenüber dem Programm `unswitching` zu einer Reduzierung der Codegröße, schlägt sich aber nur geringfügig in einer geringeren Laufzeit nieder, da bei der verwendeten Initialisierung der Variablen *cnd* das Testen der *if*-Anweisung den Wert *true* liefert und somit der *else*-Zweig des Programms nicht ausgeführt wird.

Durch die Aufteilung der Schleife des *if*-Zweigs mittels *Loop-Split* in das Programm `split` (s. Abb. 4.12) kann nun die Ausführungszeit gegenüber der vorherigen Version um 39% (≙ 29 Prozentpunkte) weiter gesenkt werden, da nun ein weiterer Teil der Anweisungen vektorisiert werden kann. Dies wirkt sich ebenso positiv auf die Anzahl der Speicherzugriffe und den Energieverbrauch aus, der nun bereits um 42% gegenüber dem Originalprogramm geringer ist. Durch die zusätzliche Schleife, für die nun Code erzeugt werden muss, ist wiederum ein Anstieg der Codegröße zu verzeichnen. Aufgrund der verstärkten Ausführung von SIMD-Operationen führt dies zu einem Anstieg der durchschnittlichen Leistungsaufnahme um nun insgesamt 26%. Mit der Umsetzung von *Reduction-Recognition* kann wiederum ein Teil der Anweisungen vektorisiert werden. So ist nun insgesamt eine Reduzierung der Ausführungszeit auf 7% erreicht. Auch die Anzahl der Speicherzugriffe und der Energieverbrauch weisen mit nunmehr 11% gegenüber den ursprünglichen Werten auf

eine drastische Verbesserung der Codequalität hin. Die durchschnittliche Leistungsaufnahme ist durch die intensive Ausführung von SIMD-Operationen auf 49% über die des Originalprogramms angestiegen. Durch das Einfügen von zwei zusätzlichen Schleifen ist die Codegröße um insgesamt 59% angestiegen.

Fazit
Die dargestellten Ergebnisse zeigen, dass die im Compiler umgesetzten Optimierungen zur Kompaktierung und Ausnutzung von Zero-Overhead Hardware-Loops bereits zu einer drastischen Verbesserung der Codequalität beitragen. Der größte Anteil der hier aufgezeigten Verbesserungen wurde mit der Vektorisierung von Schleifen erzielt. Allerdings enthielten die hier betrachteten Testroutinen jeweils eine Schleife, die ohne weitere Schleifentransformationen vektorisiert werden konnte. Ergebnisse für einige gängige Schleifentransformationen zeigten, dass sich die Anwendung von Schleifentransformationen positiv auf den Anteil der vektorisierbaren Anweisungen auswirken und zu beträchtlichen Code-Verbesserung führen kann. Wie anhand der beispielhaft durchgeführten Schleifentransformationen allerdings auch festgestellt wurde, ist damit auch ein gewisses Risiko verbunden. So führte die Anwendung der Schleifentransformationen für die hier betrachtete Testroutine zu einer Reduzierung der Ausführungszeit um 93% und des Energieverbrauchs um 90%, bei einer gleichzeitigen Erhöhung der Codegröße um 59%. Offensichtlich besteht in diesem Fall ein Trade-Off zwischen der Durchführung von Performance- und Energieoptimierungen auf der einen Seite und einer Reduzierung der Codegröße auf der anderen Seite.

4.8.2 Anordnung skalarer Variablen

In diesem Abschnitt werden experimentelle Ergebnisse für die Optimierung zur Bestimmung einer geeigneten Anordnung von skalaren Variablen innerhalb des Gruppenspeichers vorgestellt. Da die Breite des Gruppenspeichers entscheidenden Einfluss auf die Partitionierung der Variablen zu Gruppen hat, werden M3-Architekturen mit unterschiedlichen Gruppen- bzw. Partitionsgrößen betrachtet. Gekennzeichnet werden diese mit `XXslices`, wobei `XX` für die entsprechende Partitionsgröße steht. Zur Beurteilung der Effektivität dieser Optimierung werden die Ergebnisse des Codegenerators unter Verwendung des genetischen Partitionierungsverfahrens – gekennzeichnet mit dem Zusatz `opt` – den Ergebnissen bei einer unoptimierten Anordnung gegenübergestellt. Dabei werden bei dieser Anordnung keine Profiling-Informationen ausgenutzt und die Variablen entsprechend ihres Vorkommens im unoptimierten IR-Zwischencode angeordnet.

Als Testroutinen dienen hier die beiden DSP-Routinen `fft` und `dct`. Vor Durchführung der Codegenerierung wurden in beiden Routinen die enthaltenen Schleifen abgerollt und die vorhandenen Array-Zugriffe durch Zugriffe auf skalare Variablen ersetzt (*Array-Skalarisierung*). Dies hat nun den Vorteil, dass die resultierenden skalaren Variablen

beliebig im Gruppenspeicher angeordnet werden können. In den Abbildungen 4.27 und 4.28 werden Ergebnisse bezüglich der Anzahl erforderlicher Speicherzugriffe und der Ausführungszeit vorgestellt. Dabei werden für jede M3-Architektur die Ergebnisse des einfachen und des genetischen Partitionierungsverfahrens nebeneinander dargestellt und die prozentualen Veränderungen der beiden Partitionierungsverfahren für eine bestimmte Gruppenbreite angegeben (hervorgehoben durch schwarzen Kasten). Alle Ergebnisse werden in Relation zu einer M3-Architektur mit einer Gruppenbreite von eins (`1slice`) gesetzt, für die keine spezielle Partitionierung erforderlich ist.

Wie in Abb. 4.27 zu erkennen ist, führt die Verwendung des genetischen Partitionierungsverfahrens zu erheblichen Verbesserungen der Anzahl von Speicherzugriffen gegenüber der Verwendung des einfachen Partitionierungsverfahren. So werden für die FFT-Routine bis zu 50% und für die DCT-Routine bis zu 66% an Speicherzugriffen eingespart. Im Durchschnitt sind dies zwischen 39% für eine Architektur mit einer Gruppenbreite von vier und 61% für eine Gruppenbreite von zwölf.

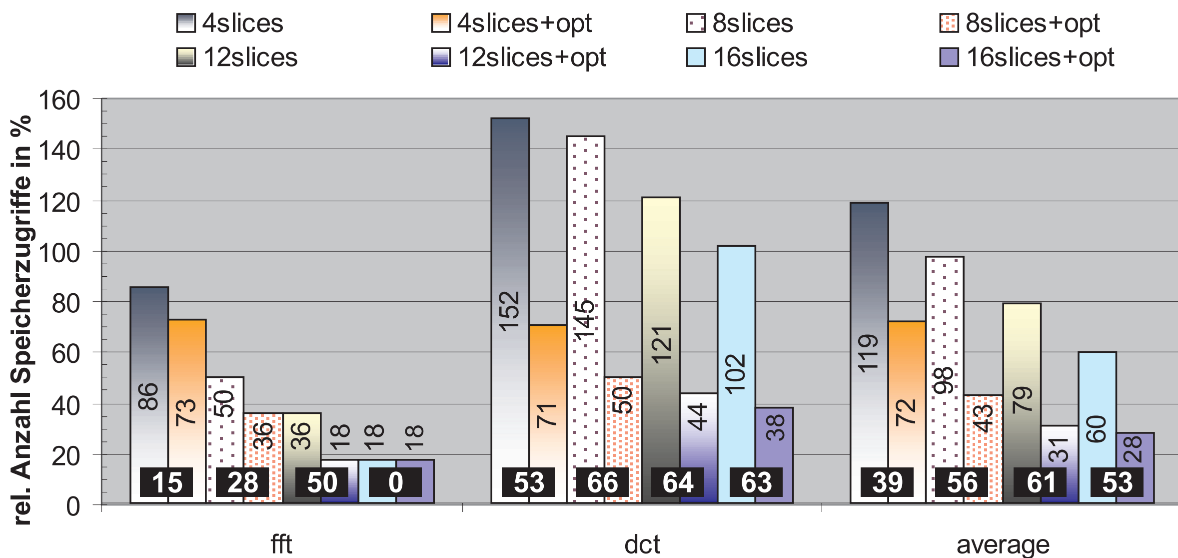


Abb. 4.27: Anordnung skalarer Variablen: Speicherzugriffe (100% $\widehat{=}$ `1slice`)

Im Vergleich der Architektur-Varianten mit einer Gruppenbreite größer als eins zeigt sich, dass durch eine günstige Anordnung der Variablen sehr viele energieintensive Speicherzugriffe eingespart werden können. Im Durchschnitt liegen die Einsparungen zwischen 28% für eine Gruppenbreite von vier und 72% für eine Gruppenbreite von 16. In Abb. 4.28 ist dargestellt, wie sich diese Reduzierung der Anzahl von Speicherzugriffen auf die Ausführungszeit auswirkt.

Es zeigt sich, dass die Einsparungen hinsichtlich der Ausführungszeit sowohl im Vergleich der Partitionierungsverfahren untereinander als auch im Vergleich der M3-Architekturen mit unterschiedlichen Gruppenbreiten geringer ausfallen. Die Ursachen sind darin zu se-

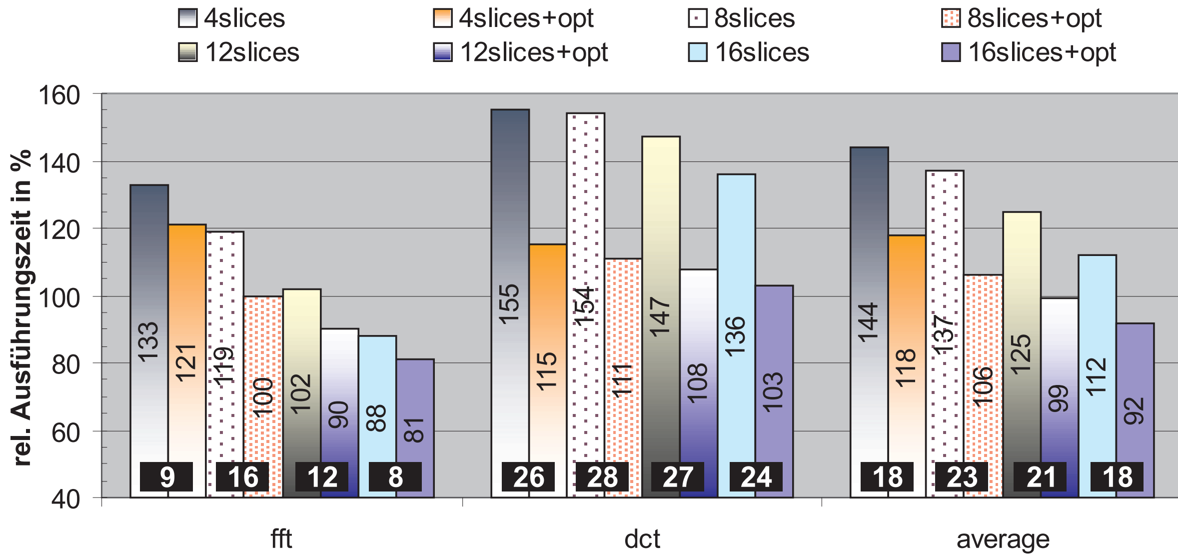


Abb. 4.28: Anordnung skalarer Variablen: Ausführungszeit (100% $\widehat{=}$ `1slice`)

hen, dass das unmittelbare Ziel dieser Optimierung in einer Reduzierung der Anzahl von Speicherzugriffen besteht, die zumindest teilweise durch Datentransfers ersetzt worden sind. Dennoch führt der Einsatz des genetischen Partitionierungsverfahrens im Vergleich zu einer einfachen Partitionierung noch zu einer deutlichen Reduzierung der Ausführungszeit. Bei Betrachtung der Ergebnisse wird auch der durch den Gruppenspeicher entstandene Overhead gegenüber einem herkömmlichen Speicher mit einer Gruppengröße von eins deutlich. So kann erst ab einer Gruppenbreite von zwölf, bei Verwendung des genetischen Partitionierungsverfahrens, der Overhead im Durchschnitt wettgemacht werden.

Fazit

Die vorgestellten Ergebnisse zeigen, dass zur Handhabung des Gruppenspeichers im SISD-Modus ein beträchtlicher Overhead entsteht, der durch eine geeignete Anordnung von Variablen zu Gruppen erheblich reduziert werden kann. Die hier vorgestellten Ergebnisse demonstrieren die Effektivität der entwickelten Optimierung, die mit kleineren Modifikationen ebenfalls für Prozessoren mit verteilten Speicherbänken, wie den Prozessoren der ADSP210X-Familie [Dev91], eingesetzt werden kann.

Kapitel 5

Experimentelle Ergebnisse

In Ergänzung zu den bereits vorgenommenen Bewertungen der Compilertechniken anhand von Testroutinen werden in diesem Kapitel Ergebnisse für eine Reihe von Benchmarks vorgestellt. Dazu werden im folgenden Abschnitt zunächst die betrachteten Benchmarks vorgestellt, anhand derer in Abschnitt 5.2 eine Bewertung der entwickelten Compilertechniken vorgenommen wird. In Abschnitt 5.3 folgt dann ein Vergleich des vom Compiler generierten Assemblercodes mit handgeneriertem Code. Nach einem Systemvergleich des M3-DSPs und des TMS320C6201 von Texas Instruments in Abschnitt 5.4 werden abschließend die Ergebnisse einer HW/SW-Exploration vorgestellt.

5.1 Betrachtete Benchmarks

Eine Übersicht der betrachteten Benchmarks einschließlich einiger charakteristischer Merkmale ist in Tabelle 5.1 gegeben. Dies umfasst die Benchmarks `n_real_up` (n_real_updates), `lms` und `dot_prod` (dot_product) aus der DSPstone-Benchmarksuite [ZVSM94], bei denen die Größe der zu verarbeitenden Arrays jeweils auf 1000 Elemente festgesetzt wurde. Des Weiteren werden mit den Benchmarks `fir`, `cmultiply` (complex_multiply), `hamming` (hamming_window), `biquad` und `lattice` typische DSP-Routinen betrachtet. Um die Anwendbarkeit der entwickelten Techniken auch für größere Programme zu zeigen, werden zusätzlich Ergebnisse für eine MP3-Anwendung `mp3` (MP3 = MPEG 1 Layer III) vorgestellt[1]. Da zur Durchführung einer Vektorisierung der Benchmarks teilweise die Anwendung von Schleifentransformationen erforderlich ist und sich dadurch bedingt Unterschiede bei der Compilierung ergeben, werden zu jedem Benchmark die charakteristischen Merkmale für eine SISD- und SIMD-Codegenerierung angegeben. Eine entsprechende Kennzeichnung erfolgt in der zweiten Spalte von Tabelle 5.1 mit SISD bzw. SIMD. Als charakterische Merkmale zu diesen Benchmarks ist in

[1] Die Benchmarks `fir`, `cmultiply`, `hamming`, `biquad`, `lattice` und `mp3` wurden von den Entwicklern des M3-DSPs der TU Dresden zur Verfügung gestellt.

den Spalten drei bis fünf die Anzahl der gemeinsamen Teilausdrücke (#CSEs) und deren Verwendungen (#CSE-Verw.) sowie die Anzahl der Basisblöcke (#BBs) angegeben. Es ist zu erkennen, dass sich durch die Anwendung der Schleifentransformationen die Anzahl der Basisblöcke teilweise mehr als verdoppelt. So erhöht sich z.B. die Anzahl der Basisblöcke für die Benchmarks `lms` und `fir` jeweils von 5 auf 11. Die Spalten sechs bis acht geben Auskunft über die Anzahl der Graphknoten der initialen GeLIR-Darstellung (IR), vor der Durchführung der Codegenerierung (vor CG) und vor der Durchführung der Adresscode-Kompaktierung (vor ACK). Stellvertretend für alle in den nachfolgenden Abschnitten verwendeten Compiler-Varianten werden in der letzten Spalte exemplarisch die Compilierungszeiten für diese Benchmarks angegeben[2]. Dabei wurden im genetischen Codegenerator die auf Seite 86 in Abschnitt 3.8.1 angegebenen Parametereinstellungen verwendet und die Anzahl durchzuführender Generationen auf die zweifache Anzahl von Genen gesetzt.

Benchmark	Modus	#CSEs	#CSE-Verw.	#BBs	#Graphknoten			Laufzeit [s]
					IR	vor CG	vor ACK	
n_real_up	SISD	1	2	3	28	61	58	23
	SIMD	1	2	3	28	58	49	25
lms	SISD	8	16	5	80	195	191	223
	SIMD	6	12	11	113	217	256	114
dot_prod	SISD	1	2	3	25	53	51	24
	SIMD	3	6	7	57	96	115	33
fir	SISD	2	4	5	53	114	120	50
	SIMD	5	10	11	99	189	204	70
cmultiply	SISD	2	6	3	42	104	84	77
	SIMD	2	6	3	42	98	90	63
hamming	SISD	5	10	7	63	107	137	120
	SIMD	3	6	7	62	104	118	37
biquad	SISD	2	4	5	81	167	128	194
	SIMD	5	10	11	168	391	312	258
lattice	SISD	6	13	9	137	312	321	245
	SIMD	10	20	19	247	523	507	249
mp3	SISD	27	54	56	722	1338	1806	1448
	SIMD	31	62	64	794	1457	1966	1583

Tabelle 5.1: Charakteristische Merkmale der betrachteten Benchmarks

[2]Die angegebenen Laufzeiten beziehen sich auf einen Intel Pentium 4-Prozessor mit 2,66 GHz Taktfrequenz.

5.2 Bewertung der Compilertechniken

Um die Auswirkungen des genetischen Codegenerators und der Vektorisierung auf die Codequalität beurteilen zu können, werden in diesem Abschnitt die folgenden Compiler-Varianten betrachtet:

- `SISD+baum`
  Durchführung einer baumbasierten Codeselektion unter Entkopplung der Phasen zum Einfügen von Spillcode und der Codekompaktierung, wie es in herkömmlichen Compilern der Fall ist. Es wird keine Vektorisierung von Schleifen durchgeführt, aber dennoch eine Gruppenspeicherbreite von 16 Elementen zugrunde gelegt.

- `SISD+baum+phasen`
  Analoge Vorgehensweise wie bei `SISD+baum`, allerdings mit einer vollständigen Phasenkopplung.

- `SISD+graph+phasen`
  Analoge Vorgehensweise wie bei `SISD+baum+phasen`, allerdings wird statt einer baumbasierten eine graphbasierte Codeselektion durchgeführt.

- `SISD+graph+phasen+1slice`
  Analoge Vorgehensweise wie bei `SISD+graph+phasen`, allerdings wird ein Gruppenspeicher mit einer Breite von eins zugrunde gelegt. Im Vergleich mit der Variante `SISD+graph+phasen`, bei der eine Gruppenspeicherbreite von 16 Elementen angenommen wird, soll hiermit der durch den Gruppenspeicher verursachte Overhead zur Wahrung der Datenkonsistenz aufgezeigt werden.

- `SIMD+graph+phasen`
  Analoge Vorgehensweise wie bei `SISD+graph+phasen`, allerdings werden SIMD-Operationen ausgenutzt.

Zusätzlich erfolgt bei allen Varianten eine Ausnutzung von Zero-Overhead Hardware-Loops. In den Abbildungen 5.1 bis 5.4 werden zunächst die Ergebnisse für die zuvor aufgeführten Benchmarks vorgestellt. Eine Darstellung der Ergebnisse für die MP3-Anwendung erfolgt anschließend. Um die Auswirkungen der Optimierungen gegenüber den üblicherweise in Compilern eingesetzten Verfahren zu demonstrieren, werden alle Ergebnisse in Relation zum Verfahren `SISD+baum` ($\widehat{=}$ 100%) gesetzt. Mit `average` wird jeweils der Durchschnitt über alle Benchmarks für jedes Verfahren angegeben.

In Abbildung 5.1 werden zunächst die Ergebnisse der oben aufgeführten Compiler-Varianten bezüglich der Ausführungszeit gegenübergestellt. Es zeigt sich, dass das Verfahren `SISD+baum`, wie es in herkömmlichen Compilern Anwendung findet, in allen

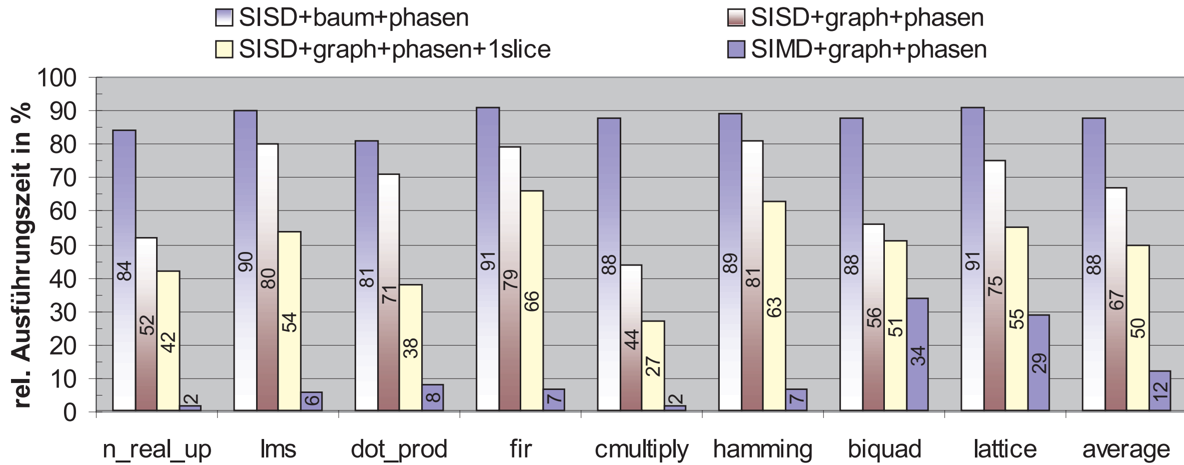


Abb. 5.1: Ergebnisse bzgl. Ausführungszeit (100% $\widehat{=}$ `SISD+baum`)

Fällen zu der geringsten Codequalität führt. Mit der Durchführung einer Phasenkopplung kann die Ausführungszeit für diese Benchmarks bereits um durchschnittlich 12% reduziert werden. Eine weitere Verbesserung der Codequalität um durchschnittlich 33% ergibt sich durch die Umsetzung einer graphbasierten Codeselektion. Die Ergebnisse von `SISD+graph+phasen+1slice` zeigen im Vergleich zu `SISD+graph+phasen` deutlich den Overhead zur Handhabung des Gruppenspeichers im SISD-Modus auf. Die besten vom Compiler generierten Ergebnisse werden bei der Ausnutzung von SIMD-Operationen erzielt. So kann die Ausführungszeit dieser Benchmarks im Durchschnitt um 88%, und im Falle der Benchmarks `n_real_up` und `cmultiply` sogar um 97,5% drastisch reduziert werden. Dies entspricht einer Erhöhung der Ausführungsgeschwindigkeit um den Faktor 40.

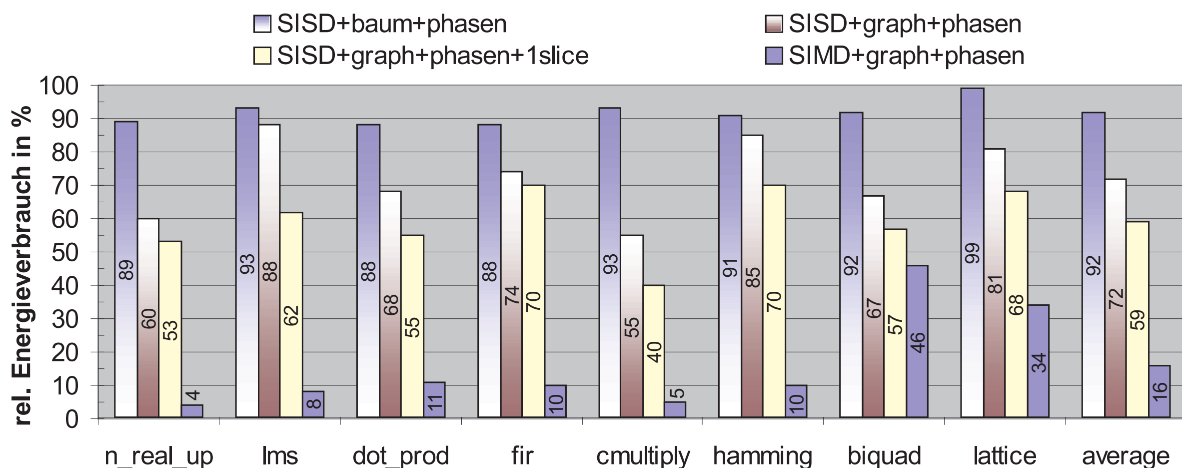


Abb. 5.2: Ergebnisse bzgl. Energieverbrauch (100% $\widehat{=}$ `SISD+baum`)

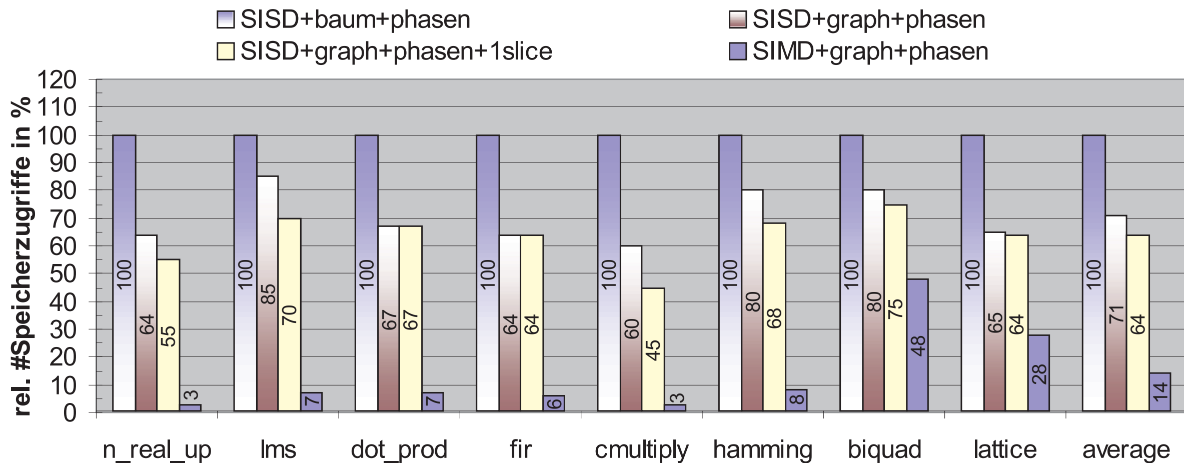


Abb. 5.3: Ergebnisse bzgl. der Anzahl von Speicherzugriffe (100% $\widehat{=}$ `SISD+baum`)

Die in den Abbildungen 5.2 und 5.3 dargestellten Ergebnisse bezüglich des Energieverbrauchs und der Anzahl erforderlicher Speicherzugriffe weisen den gleichen Trend auf. Die Einsparungen hinsichtlich des Energieverbrauchs fallen aufgrund des höheren Energiebedarfs von SIMD-Operationen im Vergleich zu dem von SISD-Operationen zwar etwas geringer aus, betragen aber immer noch durchschnittlich 84%.

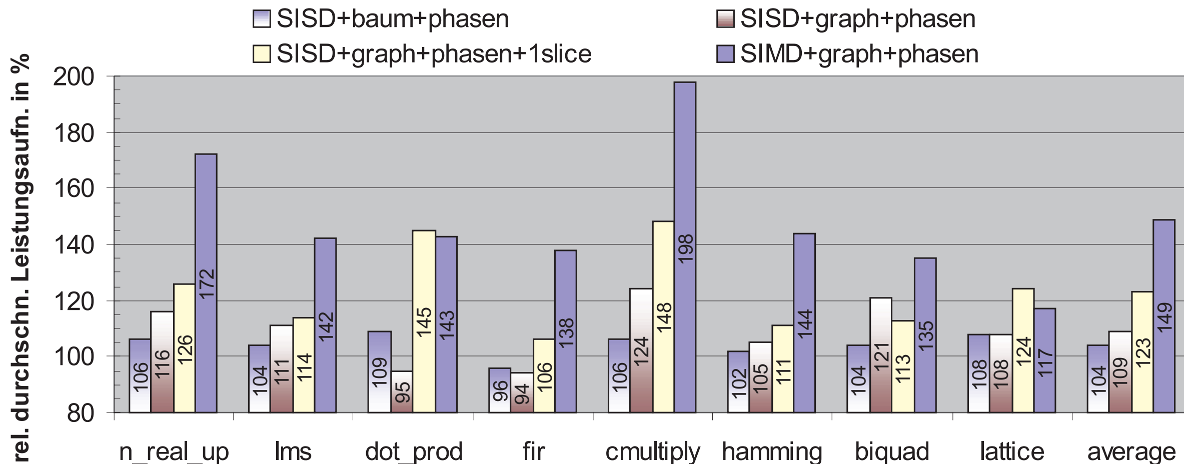


Abb. 5.4: Ergebnisse bzgl. der durchschnittlichen Leistungsaufnahme (100% $\widehat{=}$ `SISD+baum`)

In Abbildung 5.4 werden die Ergebnisse bezüglich der durchschnittlichen Leistungsaufnahme vorgestellt. Wie erwartet zeigt sich hier, dass bei einer verstärkten Ausführung von SIMD-Operationen die durchschnittliche Leistungsaufnahme am höchsten ist. So ergibt sich im Vergleich mit der Compiler-Variante `SISD+baum` bei einer Ausnutzung von SIMD-Operationen im Durchschnitt eine um 49% höhere durchschnittliche Leistungsaufnahme.

Für die `cmultiply`-Routine fällt auf, dass mit jeder ausgeführten Maschineninstruktion im SIMD-Modus zwar fast das doppelte an Leistung aufgewendet wird, der Energieverbrauch aufgrund der erheblich geringeren Ausführungszeit dennoch um 95% geringer ist.

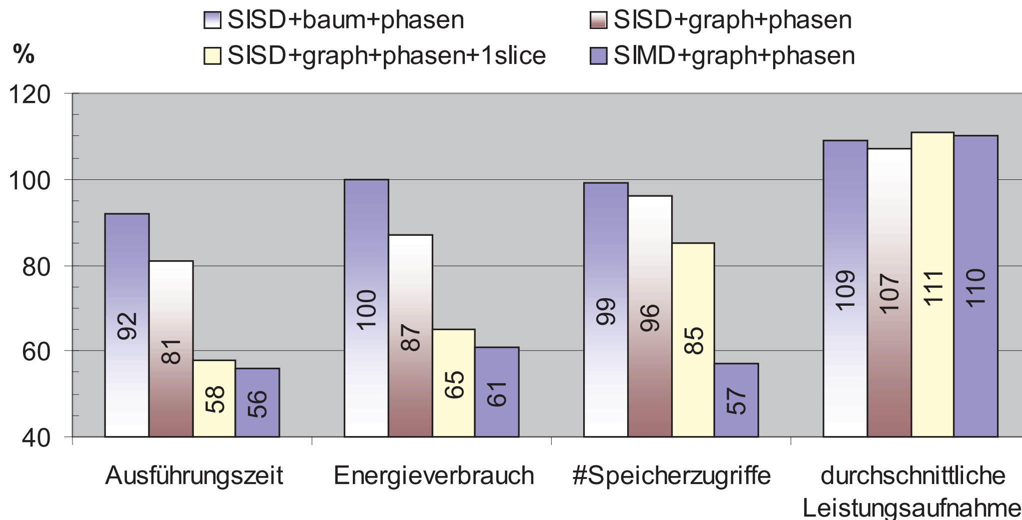


Abb. 5.5: Ergebnisse für eine MP3-Anwendung (100% $\widehat{=}$ `SISD+baum`)

In Abb. 5.5 sind die Ergebnisse für die MP3-Anwendung bezüglich Ausführungszeit, Energieverbrauch, Anzahl Speicherzugriffe und durchschnittlicher Leistungsaufnahme dargestellt. Es zeigt sich auch hier, dass der Einsatz der entwickelten Compilertechniken zu einer drastischen Verbesserung der Codequalität führt. So ergeben sich bezüglich der Ausführungszeit Verbesserungen um 44%, bezüglich des Energieverbrauchs um 39% und bezüglich der Anzahl Speicherzugriffe um 43%. Aufgrund des verhältnismäßig geringen Anteils vektorisierbarer Schleifen variiert die durchschnittliche Leistungsaufnahme bei allen Varianten nur geringfügig. Die in Tabelle 5.1 angegebenen Laufzeiten des Compilers für diese Anwendungen belegen die Anwendbarkeit der entwickelten Techniken auch für größere Programme. So werden für die Übersetzung der MP3-Anwendung mit 63 Basisblöcken 1583 Sekunden benötigt, was im Bereich der DSP-Compilierung eine akzeptable Zeitspanne darstellt. Im Vergleich zu den schnelleren traditionellen Übersetzungsverfahren sind die Laufzeiten aufgrund des zugrunde gelegten genetischen Optimierungsverfahrens zwar relativ hoch, allerdings ist die erzielte Codequalität auch wesentlich besser. Zusätzlich wird dem Anwender wesentlich mehr Flexiblität eingeräumt, da z.B. die Möglichkeit besteht, in der Entwicklungsphase einer Anwendung den genetischen Codegenerator zu Testzwecken mit einer geringen Anzahl von Generationen zu starten. Die erzielte Codequalität wird dadurch zwar i.d.R. unter den Möglichkeiten bleiben, dies hat allerdings den Vorteil, dass das Ergebnis bereits nach sehr kurzer Zeit vorliegt. Bei Bedarf wird dem Anwender auch die Möglichkeit gegeben, durch eine Erhöhung der Anzahl durchzuführender Generationen, besseren Assemblercode zu generieren, indem der Compiler „über Nacht“

gestartet wird.

5.3 Vergleich mit handgeneriertem Assemblercode

Um einen Eindruck der erzielten Codequalität des Compilers zu bekommen, wird in diesem Abschnitt zusätzlich ein Vergleich des vom Compiler generierten Assemblercodes für die Benchmarks `fir`, `cmultiply` und `hamming` mit handgeneriertem Code durchgeführt.

Im Falle des `cmultiply`-Benchmarks hat sich gezeigt, dass der Compiler an die Codequalität des Handassembler-Codes herankommt. So beträgt der Overhead bezüglich der Ausführungszeit 16% und bezüglich des Energieverbrauchs lediglich 2%. Für die `fir`- und `hamming`-Routinen hingegen beträgt der Overhead bezüglich der Ausführungszeit ca. 400% bzw. 480% und bezüglich des Energieverbrauchs ungefähr 330% bzw. 600%.

Auffällig ist, dass die auf Seite 88 in Abschnitt 3.8.2 durchgeführte Bewertung des genetischen Codegenerators im Durchschnitt keinen Overhead gegenüber handgeneriertem Code ergab. Da die dort betrachteten Routinen jeweils aus einzelnen Basisblöcken bestehen, liegt die Vermutung nahe, dass der noch vorhandene Overhead vornehmlich auf Basisblock-übergreifende Einflüsse zurückzuführen ist. So könnte z.B. durch ein Halten von Werten in Registern über Basisblock-Grenzen hinweg vor allem in Schleifen die Codequalität deutlich verbessert werden. Allerdings sind derartige Optimierungen aufgrund der stark irregulären Architektur des M3-DSPs nicht ohne weiteres umsetzbar. Wünschenswert wäre deswegen vor allem eine Reihe von homogenen Registerfiles, in denen Basisblock-übergreifend Werte bzw. Gruppen von Werten zwischengespeichert werden können.

Ein weiteres Defizit gegenüber dem handgenerierten Code lässt sich durch die mangelnde Analysefähigkeit von zu übersetzenden Programmen erklären. So können bereits für einen Menschen „einfach“ durchzuführende Analysen von Arrayzugriffen, den Compiler vor große Probleme stellen. Insbesondere die Programmiersprache C bietet dem Anwender zahlreiche Möglichkeiten, Zugriffsfunktionen auf Arrays zu implementieren, die vom Compiler nicht mehr analysiert werden können und somit eine Vektorisierung verhindern. Im Falle des `hamming`-Benchmarks war eine Vektorisierung durch den Compiler nur durch ein optimiertes Speicherlayout möglich. Die zu verarbeitenden Arrays wurden dabei so im Speicher abgelegt, dass lediglich zehn von 16 Elementen einer Gruppe ausgenutzt wurden. Die Verarbeitung erfolgte dann zwar auf allen 16 Datenpfaden, wobei allerdings 10 Datenpfade ausreichend gewesen wären. Im Gegensatz dazu konnten durch die Generierung eines speziellen Speicherlayouts beim Handassemblercode 16 Datenpfade ausgenutzt werden, wodurch sich der Overhead des vom Compiler generierten Codes im Vergleich zum handgenerierten Code erklären lässt.

5.4 Systemvergleich

Um einen Eindruck der Effizienz des Gesamtsystems bestehend aus M3-DSP und dem entwickelten Compiler zu bekommen, werden in diesem Abschnitt die Ergebnisse eines Systemvergleichs vorgestellt. Dazu erfolgt eine Gegenüberstellung der erzielten Codequalität des M3-Systems mit einem System, bestehend aus dem TMS320C6201-Prozessor von Texas Instruments (TI) und dem dazugehörigen TI-Compiler [Tex99]. Die Taktfrequenz beträgt für den TI-Prozessor 133 MHz und in Analogie zu den in [WFL+99] veröffentlichten Ergebnissen 100 MHz für den M3-DSP. Es werden wiederum die Ergebnisse mehrerer Varianten des M3-Compilers betrachtet. In Abb. 5.6 sind die Ergebnisse für die Compiler-Varianten `M3+SISD+graph+phasen`, `M3+SISD+graph+phasen+1slice` und `M3+SIMD+graph+phasen` in Relation zu denen des TI-Compilers gesetzt.

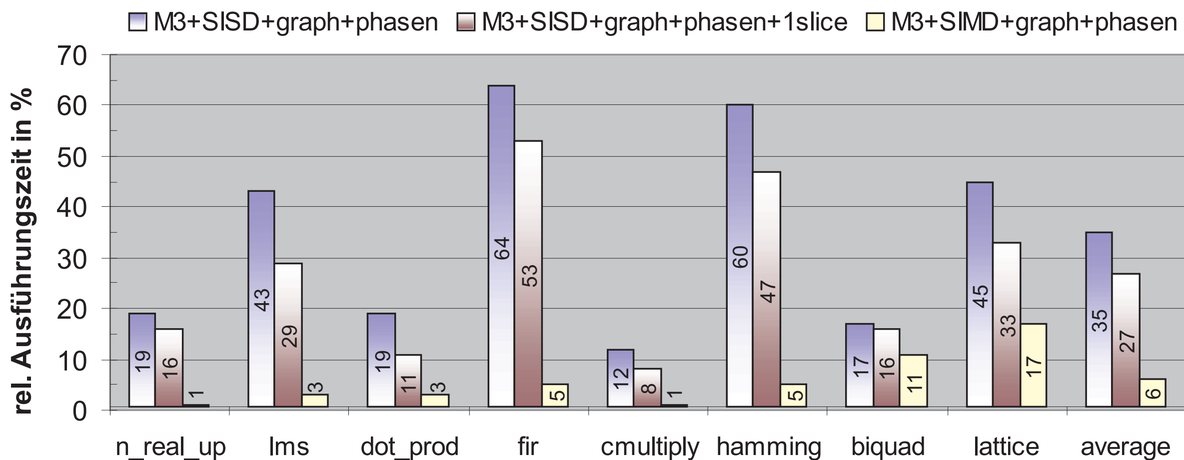


Abb. 5.6: Systemvergleich bzgl. Ausführungszeit (100% $\widehat{=}$ TI TMS320C6201)

Es zeigt sich, dass in allen Fällen das M3-System im Vergleich zum TI-System die geringsten Ausführungszeiten aufweist. Die größten Unterschiede in der Ausführungsgeschwindigkeit sind jeweils für das `cmultiply`-Benchmark zu verzeichnen. So liegt bereits mit der Compiler-Variante `M3+SISD+graph+phasen` die Ausführungsgeschwindigkeit um 88% über der des TI-Systems. Im Durchschnitt über alle Benchmarks sind dies 65%. Für die M3-Variante `M3+SISD+graph+phasen+1slice` ist aufgrund des fehlenden Overheads zur Handhabung des M3-Gruppenspeichers eine weitere Verbesserung gegenüber dem TI-System zu verzeichnen. Die besten Resultate ergeben sich wie erwartet für die M3-Variante `M3+SIMD+graph+phasen`, bei der eine Vektorisierung durchgeführt worden ist. Für die `cmultiply`- und `n_real_updates`-Benchmarks betragen die Ausführungszeiten lediglich 1% im Vergleich zu denen des TI-Systems. Für den Durchschnitt über alle betrachteten Benchmarks ergibt sich eine um 94% geringere Ausführungszeit des M3-Systems.

5.5 HW/SW-Exploration

Der in Abschnitt 5.3 aufgezeigte Overhead des Compilers gegenüber handgeneriertem Assemblercode beruht zu einem großen Anteil auf den äußerst irregulären Architektureigenschaften der M3-Prozessoren. Aus diesem Grund werden im Folgenden Ergebnisse einer in Zusammenarbeit mit den Entwicklern des M3-DSPs durchgeführten HW/SW-Exploration vorgestellt. Hierdurch sollen Erkenntnisse gewonnen werden, die es ermöglichen, ein energieeffizienteres Gesamtsystem bestehend aus Prozessor und Compiler zu entwickeln. Vor allem soll die Architektur compilerfreundlicher und damit auch energieeffizienter gestaltet werden. Dazu werden die folgenden Architekturvarianten betrachtet:

- `M3-DSP`
  Aktuelle Architektur des M3-DSPs mit 16 parallelen Datenpfaden.

- `XXslices`
  Zugrunde gelegt wird hier die Architektur des M3-DSPs, wobei allerdings die Anzahl der parallelen Datenpfade variiert wird. Eine Architektur mit 10 Datenpfaden wird demnach mit `10slices` bezeichnet.

- `scalar`
  Analog zu `M3-DSP`, allerdings wird der Befehlssatz um eine Maschinenoperation `ScalarReduction` erweitert, mit der die Werte eines Gruppenregisters baumartig entsprechend einer angegebenen Operation (z.B. Addition) verknüpft und das Ergebnis im Akkumulator des Datenpfades 0 ablegt werden (*Skalarreduktion*). Zusätzlich kann ein Registerelement angegeben werden, dessen Inhalt ebenfalls mitberücksichtigt wird. Die Ausführung einer solchen Operation auf einer Architektur mit 16 Datenpfaden benötigt 4 Taktzyklen zum Verknüpfen der Inhalte des angegebenen Gruppenregisters, zuzüglich einem extra Taktzyklus für die Verknüpfung mit dem Einzelregister.

- `accu`
  Analog zu `M3-DSP`, allerdings wird ein zusätzlicher Akkumulator in jeden Datenpfad eingefügt. Das Ergebnis einer MAC-Operation kann nun wahlweise in einen der beiden Akkumulatoren geschrieben und von dort aus weiterverarbeitet werden. Mit dieser Architekturmodifikation soll z.B. dem Compiler eine effizientere Umsetzung einer komplexen Multiplikation ermöglicht werden.

- `MACsplit+accu`
  Analog zu `accu`, allerdings wird zusätzlich in jedem Datenpfad die vorhandene MAC-Einheit in einen parallel ansteuerbaren Addierer und Multiplizierer aufgeteilt. Die Akkumulatoren sind in diesem Fall den Funktionseinheiten fest zugeordnet. In Hinblick auf den Energieverbrauch hat eine solche Aufsplittung den Vorteil, dass bei

einer einfachen Operation (Addition oder Multiplikation) kein Anlegen des entsprechenden neutralen Elementes der auszuführenden Operation erforderlich ist. Des Weiteren entfällt bei der Ausführung einer Multiplikation gefolgt von einer Subtraktion das energieintensive Invertieren des Subtrahenden. Bei dieser Variante müssen MAC-Operationen zwar als einzelne Operationen (Multiplikation und Addition) hintereinander ausgeführt werden, allerdings können zu einem Zeitpunkt mehr datenunabhängige Operationen parallel ausgeführt werden.

- `4homogRF`
  Analog zu `M3-DSP`, jedoch werden zusätzlich zu den bereits vorhandenen Registerfiles vier homogene Registerfiles berücksichtigt.

In den Abbildungen 5.7 und 5.8 werden die Auswirkungen der einzelnen Architekturmodifikationen auf die Ausführungszeit vorgestellt, indem die Veränderungen der Codequalität mit der Codequalität für den M3-DSP in Relation gestellt werden. Da für diese Architekturvarianten keine separaten Energiekostenmodelle vorliegen, wird auf die Angabe von entsprechenden Energiewerten verzichtet. Allerdings können teilweise trotzdem durchaus Rückschlüsse auf den Energieverbrauch gezogen werden, weil in der Regel eine Reduzierung der Ausführungszeit auch zu einer Reduzierung des Energieverbrauchs führt.

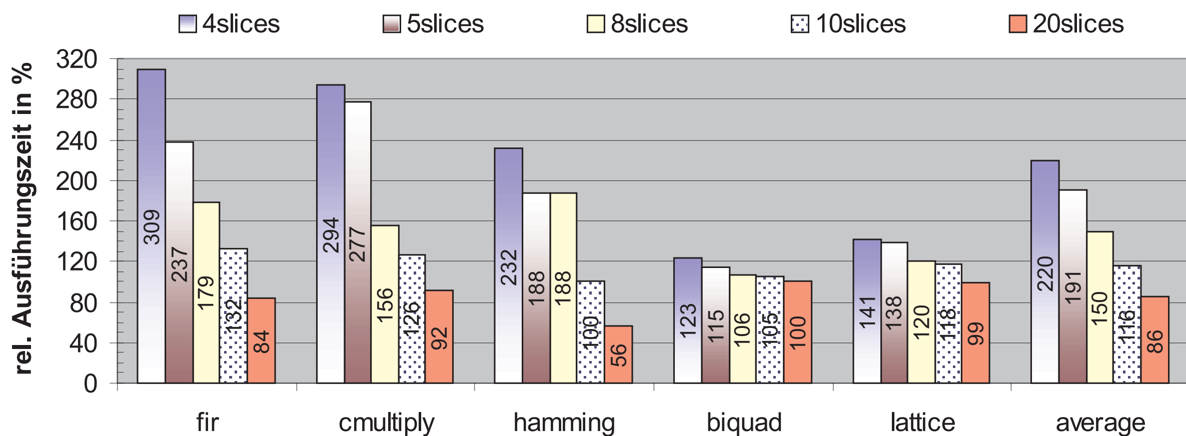


Abb. 5.7: Auswirkungen der Anzahl von Datenpfaden auf die Ausführungszeit (100% $\hat{=}$ `M3-DSP`)

In Abbildung 5.7 sind Ergebnisse für fünf Benchmarks bei Variation der Anzahl von Datenpfaden dargestellt. Wie zu erwarten, verringert sich die erforderliche Ausführungszeit bei einer steigenden Anzahl von Datenpfaden. Dies ist insbesondere bei den Benchmarks **`fir`**, **`cmultiply`** und auch **`hamming`** zu beobachten. Allerdings führt eine Erhöhung der Anzahl von Datenpfaden von z.B. zehn auf 16 zu keiner Verbesserung bei dem **`hamming`**-Benchmark, da aufgrund der speziellen Anordnung der zu verarbeitenden Arrays nur auf

zehn von 16 Datenpfaden sinnvolle Berechnungen ausgeführt werden. Aufgrund des kleineren Gruppenspeichers und der geringeren Anzahl von Datenpfaden ist bezüglich des Energieverbrauchs bei der Architektur mit zehn Datenpfaden jedoch mit einem geringeren Energieverbrauch zu rechnen. Die Auswirkungen auf die `biquad`- und `lattice`-Benchmarks sind relativ gering, da in beiden Fällen keine vollständige Vektorisierung möglich war, so dass sich eine Veränderung der Anzahl von Datenpfaden jeweils nur auf einen Teil der Anwendung auswirkt.

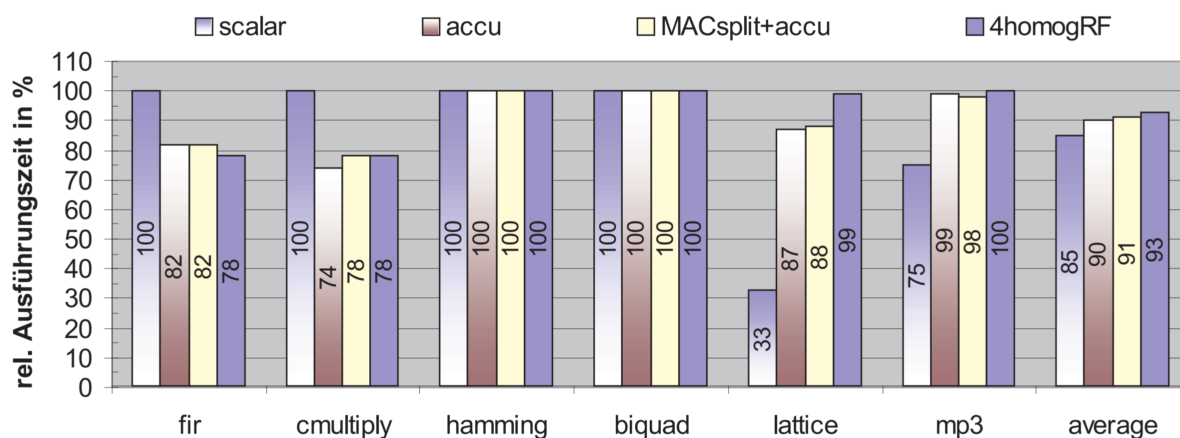


Abb. 5.8: Auswirkungen weiterer Architekturänderungen auf die Ausführungszeit (100% $\widehat{=}$ `M3-DSP`)

Abschließend werden in der Abbildung 5.8 die Auswirkungen der übrigen Architekturänderungen für die in Abb. 5.7 betrachteten Benchmarks und die MP3-Applikation vorgestellt. Die Anwendung einer Skalarreduzierung war lediglich für die Benchmarks `lattice` und `mp3` möglich und führte dort zu einer Reduzierung der Ausführungszeit von 67% bzw. 25%. Die Einführung eines zusätzlichen Akkumulators erscheint ebenfalls sinnvoll. So konnte die Ausführungszeit im Durchschnitt um 10% reduziert werden. Eine Aufsplittung der MAC-Einheit führt für diese Benchmarks trotz Wegfall der MAC-Ausführungsmöglichkeiten zu einer Reduzierung der Ausführungszeit von durchschnittlich 9%. Dies zeigt, dass der Compiler sehr gut in der Lage ist, die größere Parallelität auszunutzen. Die Berücksichtigung von vier zusätzlichen homogenen Registerfiles wirkt sich ebenfalls positiv aus. Es ist allerdings zu erwarten, dass durch den Einsatz von Basisblock-übergreifenden Optimierungen die Codequalität noch um einiges verbessert werden kann. Leider sind aufgrund der vorgenommenen Architekturänderungen keine konkreten Aussagen über den letztendlich resultierenden Energieverbrauch möglich.

Kapitel 6

Zusammenfassung

Mit dem Einsatz von Prozessoren in eingebetteten Systemen wird dem Entwickler durch die Verwendung von Software ein hohes Maß an Flexibilität eingeräumt. So wird im Gegensatz zur Verwendung von anwendungsspezifischer Hardware auch bei Spezifikationsänderungen in späten Entwicklungsphasen i.d.R. keine kosten- und zeitintensive Neuentwicklung der verwendeten Hardware erforderlich. Um den Anforderungen bezüglich einer hohen Verarbeitungsgeschwindigkeit, einer geringen Chipgröße und in zunehmendem Maße auch einem geringen Energieverbrauch zu genügen, werden häufig digitale Signalprozessoren (DSPs) zur Datenverarbeitung eingesetzt. Leider stellt die manuelle Überführung einer Anwendung in Assemblercode des Zielprozessors eine äußerst zeitaufwändige und fehlerträchtige Aufgabe dar. Aus diesem Grund werden Compiler benötigt, die in der Lage sind, eine gegebene Anwendung in effizienten Assemblercode zu überführen. Im Vergleich zu General-Purpose Prozessoren (GPPs) weisen DSPs spezielle Architekturmerkmale auf, die von herkömmlichen Compilertechniken nur unzureichend oder gar nicht ausgenutzt werden. Das Ziel dieser Arbeit bestand darum in der Entwicklung neuer Compilertechniken für DSPs, um die durch Compiler generierte Codequalität insbesondere hinsichtlich der Ausführungszeit und des Energiebedarfs zu verbessern. Aufgrund der charakteristischen Merkmale von DSPs, werden an die Compiler dabei die folgenden Anforderungen gestellt:

- Ausführung von komplexen Operationen (z.B. MAC-Operationen) in einem Taktzyklus.
- Unterstützung einer (häufig eingeschränkten) Parallelität auf Instruktionsebene und bei Bedarf von SIMD-Operationen.
- Ausnutzung von Adressgenerierungseinheiten, mit denen die Durchführung von Adressberechnungen parallel zu weiteren Operationen des Datenpfades möglich ist.
- Handhabung der irregulären Datentransferwege zwischen Registern.

- Falls erforderlich, Aufteilung der Daten auf mehrere Speicherbänke zur Erhöhung der Speicherbandbreite.
- Ausführung von einer oder mehreren Maschineninstruktionen in Zero-Overhead Hardware-Loops ohne den üblichen Schleifen-Overhead.

In den nachfolgenden Abschnitten wird nach Themen geordnet eine kompakte Darstellung dieser Arbeit gegeben. Dies betrifft im folgenden Abschnitt zunächst die neue Compiler-Zwischendarstellung GeLIR. In Abschnitt 6.2 werden dann die wesentlichen Architekturmerkmale der in dieser Arbeit betrachteten Zielarchitektur einschließlich des Energiekostenmodells zusammengefasst. Die wesentlichen Eigenschaften des entwickelten genetischen Codegenerators und der SIMD-Optimierungen werden in den Abschnitten 6.3 und 6.4 beschrieben. Abschließend wird eine Konklusion dieser Arbeit gegeben.

6.1 Compiler-Zwischendarstellung (GeLIR)

Aufgrund der mangelnden Möglichkeiten bestehender Zwischendarstellungen (IRs) zur Handhabung der irregulären DSP-Architektureigenschaften, wurde in dieser Arbeit eine neue Compiler-Zwischendarstellung GeLIR (*Generic Low-Level IR*) präsentiert. Diese stellt eine Weiterentwicklung der von Bashford entwickelten constraintbasierten Zwischendarstellung CoLIR (*Constraint based Low-Level IR*) dar und dient sowohl auf der maschinenunabhängigen als auch auf der maschinenabhängigen Ebene als allgemeines Austauschformat zwischen den einzelnen Optimierungen. Basierend auf den Konzepten von CoLIR, wurde mit GeLIR eine Compiler-Zwischendarstellung geschaffen, mit der neben einer maschinenunabhängigen Darstellung des Quellprogramms auch die Möglichkeit der Darstellung von alternativen Maschinenprogrammen besteht. Dazu können unabhängig von der Programmdarstellung prozessorspezifische Merkmale in generischer Form abgelegt werden. Insbesondere durch die Möglichkeit, die für DSPs typischen irregulären Datentransferwege und parallelen Ausführungsmöglichkeiten von Operationen spezifizieren zu können, wird die Implementierung von Compilertechniken für eine breite Klasse von Prozessoren ermöglicht.

Neben einigen einfachen Analysen und Optimierungen wird auf den GeLIR-Datenstrukturen eine δ-Array-Datenflussanalyse zur Ermittlung von Abhängigkeiten zwischen Arrayzugriffen, eine Schleifenanalyse zur Ermittlung der in einer Schleife ausgeführten Basisblöcke sowie eine generische Schleifenoptimierung zur Ausnutzung von Zero-Overhead Hardware-Loops zur Verfügung gestellt.

Zur Validierung der auf den GeLIR-Datenstrukturen ausgeführten Transformationen und Optimierungen besteht des Weiteren die Möglichkeit der Simulation einer gegebenen GeLIR-Darstellung auf unterschiedlichen Abstraktionsebenen. Grundsätzlich wird hier

zwischen einer maschinenunabhängigen Simulation und einer maschinenabhängigen Simulation unterschieden. Die Generierung des Simulators erfolgt automatisch auf Basis der gegebenen Programm- und Architekturdarstellung und vereinfacht damit erheblich die Berücksichtigung von Architekturänderungen. Bei Vorliegen eines entsprechenden Energiekostenmodells können im Gegensatz zu herkömmlichen Simulatoren auch Informationen über den Energieverbrauch des simulierten Assemblerprogramms generiert werden.

6.2 Zielarchitektur und Energiekostenmodell

Zur Demonstration der Anwendbarkeit der entwickelten Compilertechniken dienten die Prozessoren der skalierbaren M3-Plattform. Wesentliche Merkmale dieser Prozessoren sind eine Reihe von Datenpfaden, auf denen sowohl eine Abarbeitung nach dem SIMD-Prinzip (SIMD = *Single Instruction Multiple Data*) als auch in einem speziellen Einstreifen-Modus nach dem SISD-Prinzip (SISD = *Single Instruction Single Data*) möglich ist. Aufgrund des verwendeten Gruppenspeichers wird eine hohe Speicherbandbreite zur Verfügung gestellt, die im SIMD-Modus eine effektive Versorgung der Datenpfade mit Daten erlaubt. Im speziellen wurde der M3-DSP betrachtet, der eine Instanz dieser Prozessor-Plattform mit 16 Datenpfaden darstellt. Da der M3-DSP alle DSP-typischen Charakteristika aufweist und zusätzlich noch eine SIMD-Ausführung von Operationen unterstützt, stellt dieser Prozessor zur Demonstration der entwickelten Compilertechniken eine geeignete Beispielarchitektur dar.

Um neben der Performance auch den Energiebedarf von Programmen durch den Compiler optimieren zu können, wurde ein Energiekostenmodell auf Instruktionsebene für den M3-DSP vorgestellt, das eine Bewertung beliebiger Befehlssequenzen hinsichtlich des Energieverbrauchs im Codegenerator und Simulator erlaubt. Eine Validierung des Energiekostenmodells ergab eine Abweichung von weniger als 2% im Vergleich zur Messung während der Ausführung auf dem M3-DSP. Bei Betrachtung der einzelnen Energiedaten für den M3-DSP wurde deutlich, dass sich für eine Minimierung des Energieverbrauchs neben der Entwicklung von Optimierungen zur Verringerung der Ausführungszeit insbesondere Techniken lohnen, die

- zu einer Reduzierung von Speicherzugriffen führen,
- eine geschickte Auswahl und Anordnung von Maschinenoperationen zu Maschineninstruktionen vornehmen und
- die vorhandenen Datenpfade sinnvoll ausnutzen.

Zur Umsetzung dieser Ziele wurden neben einem neuen Codegenerierungs-Verfahren auf Basis eines genetischen Algorithmus auch SIMD-Optimierungen vorgestellt.

6.3 Genetischer Codegenerator (GCG)

Aufgrund der irregulären DSP-Architektureigenschaften sind im Vergleich zu GPPs besonders starke Abhängigkeiten zwischen den einzelnen Codegenerierungs-Phasen der Codeselektion, Instruktionsanordnung und Registerallokation vorhanden. Da eine separate Betrachtung der Teilprobleme zu potentiell ineffizientem Assemblercode führt, besteht ein großer Bedarf an phasengekoppelten Optimierungsverfahren, die diese Teilprobleme simultan lösen. Aus diesem Grund ist dem in dieser Arbeit vorgestellten Codegenerator ein Optimierungsverfahren auf Basis eines genetischen Algorithmus zugrunde gelegt. Eine sehr wichtige Eigenschaft genetischer Algorithmen ist es, dass geeignetes Genmaterial bevorzugt in nachfolgende Generationen übernommen wird. Ungünstige Entscheidungen, die in einer frühen Optimierungsphase (Generation) gemacht wurden, können so (im Sinne einer Phasenkopplung) revidiert werden. Im Gegensatz zu herkömmlichen Techniken ist mit diesem Verfahren eine vollständige Phasenkopplung der Teilaufgaben der Codeselektion, Instruktionsanordnung (einschließlich Kompaktierung) und Registerallokation möglich, was insbesondere für die hier betrachteten irregulären DSP-Prozessoren eine wichtige Eigenschaft darstellt. Aufgrund des integrierten Energiekostenmodells ist der Codegenerator in der Lage, eine energieeffiziente Auswahl und Anordnung von Instruktionen, mit dem Ziel der Minimierung des Energieverbrauchs einer Anwendung durchzuführen. Zusätzlich werden bereits die Auswirkungen der Adresscode-Generierung mitberücksichtigt. Des Weiteren wird anstelle der von herkömmlichen Codegeneratoren üblicherweise durchgeführten baumbasierten Codeselektion eine graphbasierte Codeselektion realisiert, was zu einer drastischen Reduzierung von energieintensiven Speicherzugriffen, dem Energieverbrauch und auch der Ausführungszeit führt.

Ergebnisse für Testroutinen bestehend aus einem Basisblock zeigen, dass der Codegenerator mit dem per Hand generierten Assemblercode konkurrieren kann. Im Durchschnitt konnte die Ausführungszeit für diese Routinen sogar um 2% verringert werden. Ergebnisse bezüglich des Energieverbrauchs zeigen, dass bei einer zusätzlich im genetischen Codegenerator durchgeführten Energieoptimierung der Energieverbrauch gegenüber dem handgenerierten Code im Durchschnitt um 6% und im Einzelfall sogar bis zu 15% reduziert werden konnte, ohne eine Verschlechterung der Ausführungszeit hinnehmen zu müssen.

Die Realisierung einer vollständigen Phasenkopplung in Verbindung mit einer graphbasierten Codeselektion erfordert im Vergleich zu Standardverfahren zwar längere Optimierungszeiten, führt allerdings auch zu effizienterem Assemblercode. So ergaben sich für die betrachteten Benchmarks durchschnittliche Verbesserungen von 33% bezüglich der Ausführungszeit, bei einer gleichzeitigen Reduzierung des Energieverbrauchs um 28%.

Eine ebenfalls vorgestellte Technik zur Adresscode-Generierung basiert auf einer schnellen Heuristik, mit der alle architekturspezifischen Randbedingungen berücksichtigt wer-

den können. Die durch diese Optimierung generierten Anweisungen zur Adressierung des Speichers werden zunächst ohne Berücksichtigung von parallelen Ausführungsmöglichkeiten in den bereits vorhandenen GeLIR-Code eingefügt. Die endgültige Zuweisung zu Maschineninstruktionen erfolgt dann in einer abschließend durchgeführten Adresscode-Kompaktierung unter Verwendung des genetischen Codegenerators. Ergebnisse für einige Testroutinen zeigten, dass durch eine Ausnutzung der Adressgenerierungseinheit und deren Spezialbefehle die Codequalität um durchschnittlich 31% bezüglich der Ausführungszeit und 30% bezüglich des Energieverbrauchs reduziert werden konnte.

6.4 SIMD-Optimierungen

Zur Einhaltung von Echtzeitbedingungen unterstützt der M3-DSP neben den üblichen parallelen Ausführungsmöglichkeiten auf Instruktionsebene auch die Ausführung von SIMD-Operationen. Aus diesem Grund wurden des Weiteren Optimierungsverfahren zur effektiven Ausnutzung der SIMD-Datenpfade und der SIMD-Speicherzugriffe vorgestellt.

Vektorisierung von Schleifen
Die beschriebene Vorgehensweise zur Ausnutzung von SIMD-Operationen basiert auf der klassischen Technik zur Vektorisierung von Schleifen auf der Basis von Fortran 90-Programmen. Im Gegensatz zu C-Programmen besteht dabei die Möglichkeit, eine parallele Verarbeitung auf Arrays direkt in den Programmen auszudrücken. Auf diese Weise kann nachfolgenden Compilerphasen relativ einfach mitgeteilt werden, dass bestimmte Anweisungen vektorisiert werden können. In Ergänzung zu den bereits entwickelten Techniken für traditionelle Vektorprozessoren und GPPs war die Entwicklung von Techniken erforderlich, mit denen die irregulären Architektureigenschaften von DSPs entsprechend berücksichtigt werden können. Dies umfasste dabei die folgenden Punkte:

- Spezifikation der erforderlichen SIMD-Funktionalität auf den GeLIR-Datenstrukturen einschließlich der erforderlichen Differenzierung unterschiedlicher Registerelemente eines Registerfiles.
- Weiterreichung von Informationen bezüglich der Vektorisierung bestimmter Anweisungen an nachfolgende Compilerphasen.
- Entwicklung einer Technik zur Handhabung von DSP-Architekturen mit Gruppenspeichern, wie sie bei den M3-Prozessoren und dem Media-Prozessor von MicroUnity verwendet werden.
- Entwicklung eines Verfahrens, mit dem spezielle SIMD-Datentransfers effektiv ausgenutzt werden können.

- Ermittlung einer optimierten Anordnung von Arrays im Gruppenspeicher, um die Anzahl der vektorisierbaren Schleifen zu erhöhen.

Die Spezifikation der SIMD-Funktionalität auf den GeLIR-Datenstrukturen hat dabei zum einen den Vorteil, dass mit der Möglichkeit zur Darstellung alternativer Maschinenprogramme präzise und gezielt Vorgaben an nachfolgende Compilierungsphasen gemacht werden können. Zum anderen besteht damit auch die Möglichkeit, die entwickelten Techniken in anderen Compilern wiederzuverwenden. Bei Kenntnis von entsprechenden Energieverbrauchswerten für einen Prozessor kann ebenfalls das Energiekostenmodell wiederverwendet werden.

Mit der Integration des Energiekostenmodells in den Codegenerator und den Simulator wird mit dieser Arbeit erstmalig das Potential von SIMD-Operationen hinsichtlich der energieeffizienten Ausführung von DSP-Programmen compilerunterstützt untersucht. Bei der Betrachtung des Energiekostenmodells für den M3-DSP wurde ersichtlich, dass für die Ausführung einer SIMD-Operation gegenüber einer entsprechenden SISD-Operation das vier- bis fünffache an Energie benötigt wird. Ergebnisse für eine Reihe von Testroutinen und Benchmarks haben jedoch gezeigt, dass dieser Overhead wieder mehr als ausgeglichen werden kann.

Für die betrachteten Benchmarks wurden Verbesserungen bezüglich der Ausführungszeit von bis zu 96% und bezüglich des Energieverbrauchs von bis zu 93% festgestellt. Im Durchschnitt lagen die Verbesserungen bei 82% bzw. 78%. Anhand weiterer Ergebnisse konnte gezeigt werden, dass sich diese Optimierung sehr gut mit einer Ausnutzung von Zero-Overhead Hardware-Loops kombinieren lässt.

Optimierte Anordnung von skalaren Variablen

Neben der großen Parallelität, die durch die Datenpfade zur Verfügung gestellt wird, kann als Grund für die drastischen Verbesserungen die hohe Speicherbandbreite des M3-DSPs angesehen werden. So wird mit der Realisierung des On-Chip-Speichers als Gruppenspeicher eine effektive Versorgung der Datenpfade mit Daten ermöglicht. Kann diese Parallelität der Datenpfade allerdings nicht ausgenutzt werden, ist es erforderlich, nacheinander auf einzelne (in Registern vorliegende) Daten zuzugreifen. Wenn sich nun aufeinander folgend zu verarbeitende Daten jeweils in unterschiedlichen Gruppen befinden, muss bei jeder Verwendung ein erneuter Speicherzugriff auf eine Gruppe durchgeführt werden. Es hat sich gezeigt, dass sich die Anzahl der Speicherzugriffe durch eine geschickte Gruppierung der Variablen erheblich verringern lässt und indirekt dadurch auch die Ausführungszeit reduziert werden kann. Es konnte gezeigt werden, dass sich das zu lösende Optimierungsproblem auf ein Partitionierungsproblem abbilden lässt. So führten gute Lösungen für dieses Partitionierungsproblem auch zu besserem Assemblercode. Mit dem Einsatz eines genetischen Partitionierungsverfahrens konnte so die Anzahl auszuführender Speicherzugriffe um bis zu 66% gegenüber einer unoptimierten Anordnung der Variablen reduziert

werden. Dadurch ergaben sich für diese Routine Verbesserungen der Ausführungszeit in Höhe von 28%.

6.5 Konklusion

In dieser Arbeit wurden neue Compilertechniken für DSPs präsentiert, die deren typische Hardwareeigenschaften effektiv ausnutzen. Anhand von Ergebnissen für eine Reihe von Testroutinen, Benchmarks und einer MP3-Anwendung konnte gezeigt werden, dass hiermit der Overhead herkömmlicher Compilertechniken bezüglich der Performance und des Energieverbrauchs drastisch reduziert werden kann. Mit dem entwickelten Codegenerator auf Basis eines genetischen Algorithmus wird dabei ein neues phasengekoppeltes Verfahren zur Codegenerierung vorgestellt, das auch eine graphbasierte Codeselektion erlaubt. Aufgrund eines integrierten Energiekostenmodells ist der genetische Codegenerator des Weiteren in der Lage, den Energieverbrauch von Programmen zu minimieren.

Ein Vergleich der Codequalität des Compilers mit handgeneriertem Assemblercode zeigte, dass für Anwendungen ohne Kontrollfluss der Compiler in der Lage ist, die Codequalität von handgeneriertem Assemblercode zu erzielen und sogar noch zu übertreffen. Bei der Betrachtung von Benchmarks mit Kontrollfluss und einer Ausnutzung der parallelen SIMD-Datenpfade konnte für ein Benchmark nahezu die Codequalität des handgenerierten Assemblercodes erzielt werden. Für zwei weitere Benchmarks war jedoch ein größerer Overhead zu verzeichnen. Als Gründe dafür konnten die nur eingeschränkten Analysemöglichkeiten des Compilers und mangelnde Basisblock-übergreifende Optimierungen ausgemacht werden. Hieraus ergeben sich Ansätze für weitere Arbeiten in diesem Bereich. So ist zu erwarten, dass bereits durch eine globale Optimierung, die das Halten von Werten in Registern über Basisblock-Grenzen auch für irreguläre Registersätze erlaubt, die Codequalität erheblich verbessert werden kann.

Mit einer Gegenüberstellung der von Compilern generierten Codequalität für den M3-DSPs und den TMS320C6201 von Texas Instruments konnte des Weiteren die Effizienz des Systems bestehend aus M3-Architektur und Compiler demonstriert werden. Leider war hier aufgrund fehlender Energieverbrauchswerte für das TI-System lediglich ein Vergleich bezüglich der Ausführungszeit möglich.

Mit der Integration des Energiekostenmodells in den Codegenerator und den Simulator konnte mit dieser Arbeit erstmalig das Potential von SIMD-Operationen hinsichtlich der energieeffizienten Ausführung von DSP-Programmen compilerunterstützt untersucht werden. Durch die beispielhafte Implementierung der Techniken für die M3-Prozessoren und die Retargierung des genetischen Codegenerators auf einen weiteren DSP wurde die Anwendbarkeit für reale Prozessoren gezeigt.

Anhang A

Referenzcode

In diesem Kapitel werden die in Abschnitt 3.8.2 verwendeten Quellprogramme und die dazugehörigen manuell erzeugten Assemblerprogramme angegeben, um dem Leser eine bessere Beurteilung des vom Compiler generierten Assemblercodes zu ermöglichen. Dabei war es das Hauptziel, den Assemblercode nachvollziehbar zu halten und ist deswegen vereinfacht als Pseudo-Assemblercode dargestellt. Da sich die Generierung des Assemblercodes auf eine M3-DSP-Architektur mit einem Slice bezieht, werden Elemente eines Registerfiles nicht explizit mit der entsprechenden Slice-Nummer angegeben. Vereinfacht dargestellt können dabei die folgenden elementaren Befehle mit den angegebenen Ressourcen verwendet werden, wobei nur eine parallele Ausführung von Befehlen aus den Klassen AGU, DTU und DMU möglich ist[1]:

AGU: Lese- bzw. Schreibzugriff auf die Adresse a des On-Chip-Speichers:

```
M = MEM[&a];
MEM[&a] = {A,B,ACCU};
```

DTU: Datentransfers zwischen zwei Registern:

```
{A,B,C,D} = M;
{A,B,C} = {A,B,D,ACCU};
```

DMU: Durchführung einer MAC-Operation[2].

```
ACCU = {A,B,ACCU,0} {+,-} {A,B,1,2} * {A,C,D,ACCU};
```

LMI: Datenmove zwischen zwei Registern:

```
{A,B,C,D,ACCU} = {A,B,C,D,ACCU};
```

[1] Die dargestellten Befehle stellen lediglich eine Teilmenge des Befehlssatzes dar, die zur Umsetzung der Quellprogramme erforderlich sind.

[2] Die Umsetzung einer Multiplikation bzw. Addition erfolgt durch Anlegen des entsprechenden neutralen Elementes (Null bzw. Eins), auf dessen Angabe im Assemblercode zugunsten einer besseren Übersicht verzichtet wurde.

A.1 Testroutine complex_multiply

A.1.1 Quellprogramm

```
int cr, ci, br, bi, ar, ai;
int main()
{
 cr = ar * br - ai * bi;
 ci = ar * bi + ai * br;
 return 0;
}
```

A.1.2 Handgeschriebener Pseudo-Assemblercode

```
AGU.reset();
M = MEM[&ar];
M = MEM[&br] || A = M;
M = MEM[&bi] || C = M;
M = MEM[&ai] || B = M || ACCU = A * C;
D = M;
ACCU = ACCU - B * D;
MEM[&cr] = ACCU || A = D || ACCU =A * B;
ACCU = ACCU + A * C;
MEM[&ci] = ACCU;
push 0;
```

A.2 Testroutine complex_update

A.2.1 Quellprogramm

```
int a0, a1, b0, b1, c0, c1, d0, d1;
int main()
{
 d0 = c0 + a0 * b0;
 d0 = d0 - a1 * b1;
 d1 = c1 + a1 * b0;
 d1 = d1 + a0 * b1;
 return 0;
}
```

A.2.2 Handgeschriebener Pseudo-Assemblercode

```
AGU.reset();
M = MEM[&a0];
M = MEM[&b0] || D = M;
M = MEM[&c0] || B = M;
M = MEM[&a1] || A = M;
M = MEM[&b1] || A = M || ACCU = A + B * D;
B =ACCU || ACCU = B * A;
M =MEM[&c1] || C = M;
B =ACCU || ACCU = B - A * C;
MEM[&d0] = ACCU || A = M;
A = C || ACCU = A + B;
ACCU = ACCU + A * D;
MEM[&d1] = ACCU;
push 0;
```

A.3 Testroutine biquad_one_section

A.3.1 Quellprogramm

```
int x, w1, w2, b0, b1, b2, a1, a2;
int main()
{
 int y, w;
 w = x - a1 * w1;
 w -= a2 * w2;
 y = b0 * w;
 y += b1 * w1;
 y += b2 * w2;
 w2 = w1;
 w1 = w;
 return 0;
}
```

A.3.2 Handgeschriebener Pseudo-Assemblercode

```
AGU.reset();
M = MEM[&w1];
M = MEM[&a1] || C = M;
M = MEM[&x] || A = M;
M = MEM[&w2] || B = M;
M = MEM[&a2] || D = M || ACCU = B - A * C;
M = MEM[&b0] || A = M;
M = MEM[&b1] || A = M || ACCU = ACCU - A * D;
MEM[&w1] = ACCU || A = M || ACCU = A * ACCU;
M = MEM[&b2] || B = D || ACCU = ACCU + A * C;
B = C;
MEM[&w2] = B || A = M;
ACCU = ACCU + A * D;
push ACCU;
```

A.4 Testroutine lattice2

A.4.1 Quellprogramm

```
int x, y, y81, y61;
int main()
{
 int y2,y3,y4,y5,y6,y7,y8,y9;
 y2 = (y81 + x) * 1;
 y3 = y2 + x;
 y5 = (y3 + y61) * 2;
 y6 = y3 + y5;
 y8 = y5 + y61;
 y9 = ((y2 + y81) * 3) + (y8 * 4);
 y = y9 + (y6 * 5);
 y81 = y8;
 y61 = y6;
 return 0;
}
```

A.4.2 Handgeschriebener Pseudo-Assemblercode

```
AGU.reset();
M = MEM[&x];
M = MEM[&y81] || C = M;
M = MEM[&y61] || A = M;
M = MEM[&y81] || D = M || ACCU = A + C;
A = ACCU || ACCU = ACCU + C;
B = ACCU || ACCU = ACCU + D;
ACCU = 2 * ACCU;
C = ACCU || ACCU = ACCU + D;
MEM[&y81] = ACCU || C = ACCU || ACCU = B + C;
MEM[&y61] = ACCU || B = M;
A = ACCU || ACCU = A + B;
B = 3;
B = 4 || ACCU = B * ACCU;
C = 5 || ACCU = ACCU + B * C;
ACCU = ACCU + A * C;
MEM[&y] = ACCU;
push 0;
```

A.5 Testroutine dfg1

A.5.1 Quellprogramm

```
int a, b, c, d;
int main()
{
 int t_a, t_b, t_c;
 t_a = c - d * (a - b);
 t_b = t_a + b * d * t_a;
 t_c = c + (e - t_a * t_b);
 c = t_c;
 return 0;
}
```

A.5.2 Handgeschriebener Pseudo-Assemblercode

```
AGU.reset();
M = MEM[&a];
M = MEM[&b] || A = M;
M = MEM[&d] || C = M;
M = MEM[&c] || A = M || ACCU = A - C;
M = MEM[&e] || B=M;
ACCU = B - A*ACCU;
A = ACCU || ACCU = A * C;
ACCU = A + A * ACCU;
A = M || ACCU = A * ACCU;
ACCU = A - ACCU;
ACCU = B + ACCU;
MEM[&c] = ACCU;
push 0;
```

A.6 Testroutine dfg2

A.6.1 Quellprogramm

```
int a, b, c, d, e;
int main()
{
 int t_a, t_b, t_c, t_d;
 t_a = a + b * c - d * (a - b);
 t_b = t_a + b * d * t_a;
 t_c = t_b * c + d * (e - t_a * t_b);
 t_d = t_b * t_c + d;
 d = t_d;
 return 0;
}
```

A.6.2 Handgeschriebener Pseudo-Assemblercode

```
AGU.reset();
M = MEM[&a];
M = MEM[&b] || B = M;
M = MEM[&c] || A = M;
M = MEM[&d] || D = M || ACCU = B - A;
C = ACCU || ACCU = B + A * D;
M = MEM[&e] || B = M;
ACCU = ACCU - B * C;
A = ACCU || ACCU = A * ACCU;
ACCU = A + B * ACCU;
C = ACCU || ACCU = A * ACCU;
A = M;
A = C || ACCU = A - ACCU;
ACCU = B * ACCU;
ACCU = ACCU + A * D;
ACCU =B + A * ACCU;
MEM[&d] = ACCU;
push 0;
```

Literaturverzeichnis

[ABI+95] K. Asanovic, J. Beck, B. IZZbrissou, B.E.D. Kingsbury, N. Morgan, and J. Wawrzynek. The T0 Vector Microprocessor. In *Proceedings of Hot Chips VII*, pages 187–196, August 1995.

[Ada92] J. Adams. *Fortran 90 Handbook*. McGraw Hill, 1992.

[AGT89] A.V. Aho, M. Ganapathi, and S.W.K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4), October 1989.

[aiS] aiSee. http://www.aisee.com.

[AK87] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(4), October 1987.

[AM95] G. Araujo and S. Malik. Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, pages 36–41, 1995.

[AML96] G. Araujo, S. Malik, and M. Lee. Using Register Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures. In *Proceedings of the Design Automation Conference (DAC)*, 1996.

[AOC02] G. Araujo, G. Ottini, and M. Cintra. Global Array Reference Allocation. *Transactions on Design Automation for Electronic Systems (TODAES)*, 7(2), April 2002.

[ASU77] A.V. Aho, R. Sethi, and J.D. Ullman. Code Generation for Expressions with Common Subexpressions. *Journal of the ACM*, 24(1), January 1977.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Bä96] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

[Bak74] K.R. Baker. *Introduction to Sequencing and Scheduling*. Wiley, 1974.

[Bar92] D. H. Bartley. Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes. In *Software Practice and Experience*, volume 22(2), pages 101–110, February 1992.

[Bas95] S. Bashford. Code Generation Techniques for Irregular Architectures. Technical Report 596, Lehrstuhl Informatik XII, University of Dortmund, November 1995.

[Bas01] S. Bashford. *Constraintbasierte Codegenerierung für eingebettete Prozessoren*. PhD thesis, Universität Dortmund, http://eldorado.uni-dortmund.de:8080/FB4/ls12/forschung/2001/Bashford, 2001.

[Bea91] S.J. Beaty. *Instruction Scheduling Using Genetic Algorithms*. PhD thesis, Department of Mechanical Engineering, Colorado State University, Fort Collins, Colorado, USA, 1991.

[BGS94] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4), December 1994.

[BL99] S. Bashford and R. Leupers. Constraint driven Code Selection for Fixed-Point DSPs. In *Proceedings of the Design Automation Conference (DAC)*, 1999.

[CAC+81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markenstein. Register Allocation via Coloring. *Computer Languages*, 6(1), January 1981.

[CoS] CoSy. http://www.ace.nl.

[Dev91] Analog Devices. *ADSP-2001 User's Manual*, 1991.

[DeV97] D.J. DeVries. *A Vectorizing SUIF Compiler*. PhD thesis, University of Toronto, June 1997.

[DF02] T. Dräger and G. Fettweis. Energy Savings with Appropriate Interconnection Networks in Parallel DSP. In *Proceedings of the Workshop zum DFG-Verbundprojekt „Grundlagen und Verfahren verlustarmer Informationsverarbeitung (VIVA)“*, pages 35–42, Chemnitz, Germany, March 2002.

[DGS93] E. Duesterwald, R. Gupta, and M. Soffa. A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations. In *Proceedings of the Conference on Programming Languages Design and Implementation (PLDI)*, pages 68–77, Albuquerque, New Mexico, June 1993.

[EB98] J. Eyre and J. Bier. DSP Processors Hit the Mainstream. *IEEE Computer*, August 1998.

[Ert99] M.A. Ertl. Optimal Code Selection in DAGs. In *Proceedings of the Symposium on the Principles of Programming Languages (POPL)*, pages 242–249, January 1999.

[ESL89] H. Emmelmann, F.W. Schröer, and R. Landwehr. BEG – A Generator for Efficient Back Ends. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 227–237, New York, USA, 1989.

[Fal02] H. Falk. Control Flow Optimization by Loop Nest splitting at the Source Code Level. Technical Report 773, Universität Dortmund, Lehrstuhl Informatik XII, October 2002.

[FHP92] C.W. Fraser, D.R. Hanson, and T.A. Proebsting. Engineering a Simple, Efficient Code-Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3), September 1992.

[Fie01] M. Fiesel. XML-basierte generische Zwischendarstellung für Compiler. Master's thesis, Universität Dortmund, Lehrstuhl Informatik XII, 2001.

[FM03] H. Falk and P. Marwedel. Control Flow driven Splitting of Loop Nests at the Source Code Level. In *Proceedings of the Design Automation and Test Conference in Europe (DATE)*, Munich, Germany, March 2003. (to appear).

[Frö01] S. Fröhlich. *Codegenerierung für Signalprozessoren mit Hilfe genetischer Algorithmen*. PhD thesis, Technische Universität Wien, April 2001.

[FWD+98] G. Fettweis, M. Weiss, W. Drescher, U. Walther, F. Engel, and S. Kobayashi. Breaking new grounds over 3000 MOPS: A broadband mobile multimedia modem DSP. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT)*, pages 1547–1551, Toronto, Canada, September 1998.

[Geb97] C. H. Gebotys. An Efficient Model for DSP Code Generation: Performance, Code Size, Estimated Energy. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, pages 41–47, Antwerp, Belgium, September 1997.

[GeL] GeLIR. http://ls12-www.cs.uni-dortmund.de/research/gelir/.

[GFO97] A. Gierlinger, R. Forsyth, and E. Ofner. Gepard - A Parameterizable DSP Core for ASICs. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT)*, San Diego, California, USA, 1997.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Company, 1979.

[Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[Gro90] G.F. Grohoski. Machine Organization of the IBM RISC System/6000 Processor. *IBM Journal of Research and Development*, 34(1), 1990.

[Han96] C. Hansen. MicroUnity's MediaProcessor Architecture. *IEEE Micro*, 16(4), August 1996.

[Han99] S. Hanono. *Aviv: A Retargetable Code Generator for Embedded Processors*. PhD thesis, Massachusetts Institute of Technology, 1999.

[HD98] S. Hanono and S. Devadas. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *Proceedings of the Design Automation Conference (DAC)*, pages 510–515, San Francisco, California, USA, June 1998.

[HKN+01] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr. A Novel Methodology for the Design of Application-Specific Instruction-Set Processors (ASIPs) Using a Machine Description Language. *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 20, 2001.

[Hol92] J.H. Holland. *Adaption in Natural and Artificial Systems*. MIT Press, 1992.

[Hor01a] L. Hornbach. Generische Low-Level Optimierungen für RISC-Architekturen. Master's thesis, Universität Dortmund, Lehrstuhl Informatik XII, 2001.

[Hor01b] M. Horst. Schleifenoptimierungen zur Ausnutzung paralleler Rechenwerke von Prozessoren der M3-DSP Plattform. Master's thesis, Universität Dortmund, Lehrstuhl Informatik XII, 2001.

[Int] Moore's Law. http://www.intel.com/research/silicon/mooreslaw.htm.

[JP01] S. Jung and Y. Paek. The Very Portable Optimizer for Digital Signal Processors. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 84–92, Atlanta, Georgia, USA, November 2001.

[KB02] W. Kantschik and W. Banzhaf. Linear-Graph GP – A new GP Structure. In *Proceedings of the European Conference on Genetic Programming (EuroGP)*, pages 83–92, Kinsale, Ireland, April 2002.

[KL70] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49, 1970.

[KL98] D. Kästner and M. Langenbach. Integer Linear Programming vs. Graph-Based Methods in Code Generation. Technical Report A/01/98., Universität des Saarlandes, 1998.

[KMT+95] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner. The Visual Instruction Set (VIS) in UltraSPARC. In *Proceedings of IEEE COMPCON*, pages 462–469, San Francisco, California, USA, March 1995.

[Kot00] D. Kottmann. Adreßzuweisung für den M3-DSP. Master's thesis, Universität Dortmund, Lehrstuhl Informatik XII, 2000.

[KP93] C.W. Kessler and W.J. Paul. Automatic Parallelization by Pattern Matching. In *Proceedings of the International Conference of Parallel Computing (ACPC)*, pages 166–181, Gmunden, Austria, October 1993.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[Kra00] A. Krall. Compilation Techniques for Multimedia Extensions. In *International Journal of Parallel Programming*, volume 28, pages 347–361, 2000.

[Kru56] J.B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1), 1956.

[Käs00] D. Kästner. PROPAN: A Retargetable System for Postpass Optimisations and Analyses. In *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 63–80, Vancouver, California, USA, June 2000.

[Käs01] D. Kästner. *Retargetable Postpass Optimisation by Integer Linear Programming*. PhD thesis, Universität des Saarlandes, 2001.

[LA86] C. F. Lin and J. B. Anderson. M-algorithm Decoding of Channel Convolutional Codes. In *Proceedings of the Princeton Conference of Information Science and Systems*, pages 362–366, Princeton, Great Britain, March 1986.

[LA00] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 145–156, Vancouver, Canada, June 2000.

[LBSL97] P. Lapsley, J. Bier, A. Shoham, and E. Lee. *DSP Processor Fundamentals*. Wiley, 1997.

[LD98] R. Leupers and F. David. A Uniform Optimization Technique for Offset Assignment Problems. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, pages 3–8, Hsinchu, Taiwan, December 1998.

[LDK+95] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage Assignment to Decrease Code Size. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 186–195, La Jolla, California, USA, 1995.

[LDKT95] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction Selection Using Binate Covering for Code Size Optimization. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 1995.

[LDL+01] M. Lorenz, T. Dräger, R. Leupers, P. Marwedel, and G.P. Fettweis. Low-Energy DSP Code Generation Using a Genetic Algorithm. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 431–437, Austin, Texas, USA, September 2001.

[Leu98] R. Leupers. Optimized Array Index Computation in DSP Programs. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 87–92, Yokohama, Japan, February 1998.

[Leu99] R. Leupers. Schneller Code statt schnelle Compiler. *Elektronik*, 22, 1999.

[Leu00a] R. Leupers. *Code Optimization Techniques for Embedded Processors*. Kluwer Academic Publishers, 2000.

[Leu00b] R. Leupers. Code Selection for Media Processors with SIMD Instructions. In *Proceedings of the Design Automation and Test Conference in Europe (DATE)*, pages 4–8, Paris, France, March 2000.

[Leu00c] R. Leupers. Register Allocation for Common Subexpression in DSP Data Paths. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2000.

[Lev96] D. Levine. Users Guide to the PGAPack Parallel Genetic Algorithm Library. Technical Report ANL-95/18, Argonne National Laboratory, January 1996.

[LKB+01] M. Lorenz, D. Kottmann, S. Bashford, R. Leupers, and P. Marwedel. Optimized Address Assignment for DSPs with SIMD Memory Accesses. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 415–420, Yokohama, Japan, January 2001.

[LLHT00] C. Lee, J.K. Lee, T.T. Hwang, and S.-C. Tsai. Compiler Optimization on Instruction Scheduling for Low Power. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, pages 55–60, Madrid, Spain, September 2000.

[LM96] R. Leupers and P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 109–112, San Jose, California, USA, November 1996.

[LM01] R. Leupers and P. Marwedel. *Retargetable Compiler Technology for Embedded Systems.* Kluwer Academic Publishers, 2001.

[LML02] M. Lorenz, P. Marwedel, and R. Leupers. Energiebewusste Compilierung für Digitale Signalprozessoren. In *Proceedings of the Workshop zum DFG-Verbundprojekt „Grundlagen und Verfahren verlustarmer Informationsverarbeitung VIVA“*, pages 76–83, Chemnitz, Germany, March 2002.

[LTMF95] M. T.-C. Lee, V. Tiwari, S. Malik, and M. Fujita. Power Analysis and Low-Power Scheduling Techniques for Embedded DSP Software. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, pages 110–115, Cannes, France, September 1995.

[LWDL02] M. Lorenz, L. Wehmeyer, T. Dräger, and R. Leupers. Energy aware Compilation for DSPs with SIMD Instructions. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems and Software and Compilers for Embedded Systems (LCTES/SCOPES)*, pages 94–101, Berlin, Germany, June 2002.

[Mar97] P. Marwedel. Compilers for Embedded Processors. In *Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI)*, pages 201–208, Osaka, Japan, December 1997.

[MB02] P. Marwedel and L. Benini. Low-Power/Low-Energy Embedded Software. Tutorial at Design, Automation and Test in Europe (DATE), Paris, March 2002.

[MG95] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[MKC00] R. Manniesing, I. Karkowski, and H. Corporaal. Automatic SIMD Parallelization of Embedded Applications Based on Pattern Recognition. In *Proceedings of the International Euro-Par Conference*, pages 349–356, Munich, Germany, August 2000.

[Moo65] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[Mot86] Motorola. *DSP56000, Digital Signal Processor, User's Manual*, 1986.

[MPS98] E. Macii, M. Pedram, and F. Somenzi. High-Level Power Modeling, Estimation, and Optimization. In *Transactions on CAD of ICs and Systems*. IEEE, November 1998.

[Muc97] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[Nis97] V. Nissen. *Einführung in Evolutionäre Algorithmen*. Vieweg, 1997.

[NN94] S. Novack and A. Nicolau. Mutation Scheduling: A Unified Approach to Compiling for Fine–Grain Parallelism. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 892 of *LNCS*, pages 16–30. Springer–Verlag, Ithaca, New York, USA, August 1994.

[Pol97] R. Poli. Evolution of Graph-Like Programs with Parallel Distributed Genetic Programming. In *Proceedings of the International Conference on Genetic Algorithms (ICGA)*, pages 346–353, East Lansing, Michigan, USA, July 1997.

[PSB01] G. Pokam, J. Simonnet, and F. Bodin. A Retargetable Preprocessor for Multimedia Instructions. In *Proceedings of the Workshop on Compilers for Parallel Computers (CPC)*, Edingburgh, Scotland, June 2001.

[RF98a] A. Römer and G.P. Fettweis. Code Generation for Processors with VLIW Architecture. In *Workshop on System Design Automation (SDA)*, pages 31–35, Dresden, Germany, March 1998.

[RF98b] A. Römer and G.P. Fettweis. Neuer Ansatz für die Code-Generierung mit Hilfe des Viterbi-Algorithmus. In *Proceedings of DSP Deutschland*, pages 78–86, Munich, Germany, October 1998.

[RP96] J.M. Rabaey and M. Pedram, editors. *Low Power Design Methodologies*. Kluwer Academic Publishers, 1996.

[SBT00] A. Sama, M. Balakrishnan, and J.F.M. Theeuwen. Speeding up Power Estimation of Embedded Software. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 191–196, Rapallo, Italy, July 2000.

[SG00] N. Sreraman and R. Govindarajan. A Vectorizing Compiler for Multimedia Extensions. *International Journal of Parallel Programming*, 28(4), 2000.

[SKWM01] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In *Proceedings of the International Workshop Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Yverdon-Les-Bains, Switzerland, September 2001.

[SMM+91] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Whitley. A Comparison of Genetic Sequencing Operators. In *Proceedings of the International Conference on Genetic Algorithms (ICGA)*, pages 69–76, San Mateo, California, USA, 1991. Morgan Kaufman.

[SPA] SPAM. http://www.ee.princeton.edu/spam/.

[SS99] G. Sinevriotis and T. Stouraitis. Power Analysis of the ARM 7 Embedded Microprocessor. In *Proceedings of the International Workshop Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 261–270, Kos Island, Greece, October 1999.

[Sta] StarCore. http://www.starcore-dsp.com/starcore.html.

[STD94] C.-L. Su, C.-Y. Tsui, and A.M. Despain. Low Power Architecture Design and Compilation Techniques for High-Performance Processors. In *Proceedings of IEEE COMPCON*, pages 489–498, February 1994.

[Sti99] A. Stiller. Prozessorgeflüster. *CT*, 25, 1999.

[SUI] SUIF. http://suif.stanford.edu/suif/.

[Tex99] *TMS320C6000 CPU and Instruction Set Reference Guide*. Texas Instruments, 1999.

[Tig] TigerShark. http://www.analog-devices.com.

[Tji93] S. Tjiang. An Olive Twig. Technical report, Synopsys Inc., 1993.

[TMW94a] V. Tiwari, S. Malik, and A. Wolfe. Compilation Techniques for Low Energy: An Overview. In *Proceedings of the Symposium on Low Power Electronics*, pages 38–39, San Diego, California, USA, October 1994.

[TMW94b] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: A First Step towards Software Power Minimization. In *IEEE Transactions on VLSI Systems*, pages 437–445, December 1994.

[Tri] Trimaran. http://www.trimaran.org/.

[WFL+99] M.H. Weiss, G. P. Fettweis, M. Lorenz, R. Leupers, and P. Marwedel. Toolumgebung für plattformbasierte DSPs der nächsten Generation. In *Proceedings of DSP Deutschland*, pages 175–184, Munich, Germany, September 1999.

[WG97] B. Wess and M. Gotschlich. Optimal DSP Memory Layout Generation as a Quadratic Assignment Problem. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, volume 3, pages 1712–1715, Hong Kong, June 1997.

[WGHB94] T. Wilson, G. Grewal, B. Halley, and D. Banerji. An Integrated Approach to Retargetable Code Generation. In *Proceedings of the International Symposium on High-Level Synthesis*, pages 70–75, Niagra-on-the-Lake, Ontario, Canada, May 1994.

[WM95] R. Wilhelm and D. Maurer. *Compiler Design*. Addison Wesley, 1995.

[WSS91] D. Whitley, T. Starkweather, and D. Shaner. The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination. In L. David, editor, *The Handbook of Genetic Algorithms*, pages 350–372, Van Nostrand Reinhold, New York, USA, 1991.

[ZDT99] E. Zitzler, K. Deb, and L. Thiele. Comparison of Multiobjective Evolutionary Algorithms: Empirical Results (Revised Version). Technical Report 70, Institut für Technische Informatik und Kommunikationsnetze, ETH Zürich, December 1999.

[Zim90] H. Zima. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.

[ZVSM94] V. Zivojnovic, J.M. Velarde, C. Schläger, and H. Meyr. DSPstone - A DSP-oriented Benchmarking Methodology. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT)*, pages 715–720, Dallas, Texas, USA, October 1994.

[ZW99] T. Zeitlhofer and B. Wess. Operation Scheduling for Parallel Functional Units Using Genetic Algorithms. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 1997–2000, Phoenix, Arizona, USA, March 1999.

Index

δ-Array-Datenflussanalyse, 40, 119

Adresscode-Generierung, 5, 51, 56, 60, 80, 102
Adresscode-Kompaktierung, 60, 85, 86
Adressgenerierungseinheit, 5, 51
Adressoffset
  konstant, 83
  variable, 83
Adresspointer-Register, 51
Adresszuweisung, 57, 125
  horizontale, 62, 125
  vertikale, 60, 125
Aggregation, 66
AGU, 5, 51
aiSee, 41
ALAP, 69
Allel, 63, 68
alternative Ausführungsmöglichkeit, 107
Antiabhängigkeit, 25
Architekturdarstellung, 32, 101, 107
Array-Skalarisierung, 136
ASAP, 69
ASIC, 2
Ausgabeabhängigkeit, 25
Auto-Dekrement, 51
Auto-Inkrement, 51, 83
Auto-Modify, 52, 83
AVIV, 55

Back-End, 3
Basisblock, 22
Baumgrammatik, 48
Bewertung von Individuen, 75
Bypass, 70, 72

Chromosom, 63, 67
COCOON, 28
Codegenerator, 101
  genetischer, 62, 64
Codegenerator-Generator, 27, 48
Codegenerierung, 4, 47, 53, 59
Codeselektion, 4, 50
  baumbasiert, 16, 48, 71
  graphbasiert, 48, 71
Cold-Scheduling, 58
CoLIR, 28
Compiler, 3
Compilierungsprozess, 3
Constant-Folding, 3
Constraint-Programmierung, 54
Constraintpropagierung, 30, 37, 39, 61
Copy-Propagation, 3
CoSy, 26
Crossover, 63, 77
  CS-, 78
  Einpunkt-, 77
  Uniform-, 77
  Zweipunkt-, 77
CSE, 16, 48

Datenflussabhängigkeit, 25
Datenflussgraph, 16
Dead-Code-Elimination, 3, 72, 112
Delay-Line, 6
DFG, 16
digitale Signalverarbeitung, 5
Dominanz, 65

Drei-Adressbefehl, 23
DSP, 2, 7, 47, 97
DSP-Plattform, 7
Dummy-Adressbefehl, 102

eingebetteter Prozessor, 2
eingebettetes System, 2
Einstreifen-Modus, 7
ELCOR, 27
Element-Datentransfer, 108
Energiegruppe, 10
Energiekosten
  Basis-, 10
  Overhead-, 10
Energiekostenmodell, 9
Energieoptimierung, 13, 57

feinkörnige Parallelität, 97
FIR-Filter, 5
Fortran 90, 103
Front-End, 3
Funktionseinheit, 33

ganzzahlig lineare Programmierung, 53
GCG, 62, 64
GeLIR, 29
Gen, 63, 67
General-Offset-Assignment, 52
Generic Low-Level IR, 29
genetische Programmierung, 79
genetischer Algorithmus, 56, 63
  Parameter, 87
GOA, 52
GPP, 2, 16, 47
Gruppe, 7
Gruppenregister, 99
Gruppenregisterfile, 8, 100
Gruppenspeicher, 8, 100, 109

Hamilton-Pfad, 57
Hamming-Distanz, 58
HW/SW-Exploration, 147

Idiom-Recognition, 103
ILP, 17
Individuum, 63
Initialisierung eines Individuums, 68
Inline-Assemblercode, 98
Instruktionsanordnung, 4, 50
Instruktionstyp, 25, 33
Intrinsic, 98
IR, 3

Kante
  externe, 127
  interne, 127
Kantenkonsistenz, 37, 39, 61
Kernighan-Lin-Algorithmus, 128
Knotenkonsistenz, 37, 38, 61
Kompaktierung, 4
kompilierte Simulation, 44
Kontrolldatenflussgraph, 41
Kontrollflussgraph, 22, 41

LANCE, 27
lebendige Variable, 4
LIR, 4
List-Scheduling, 55, 56, 68
  probabilistisch, 64, 69
Low-Level IR, 4

M-Algorithmus, 55
M3-DSP, 8
M3-Plattform, 7
MAC-Operation, 6, 108
Maschineninstruktion, 4, 25
Maschineninstruktionstyp, 25
Maschinenoperation, 4, 24
  abstrakte, 22
  faktorisierte, 24, 33
  komplexe, 24
  partielle, 24

zgoto, 112
zloop, 112
Maschinenprogramm
alternatives, 35
Middle-End, 3
MMX-Befehl, 103
Mobilität, 69
Modify-Register, 52
Moore'sches Gesetz, 1
Mustererkennung, 103
Mutation, 63, 79

Nachbar, 126
Nachbarschaftsbeziehung, 126
erfüllte, 127
unerfüllte, 127
Nicht-Dominanz, 65

Page-Pointer-Adressierung, 52
Page-Pointer-Register, 52, 84
Pareto-Optimalität, 65
Partitionierung, 127
Partitionierungsverfahren, 128
genetisches, 128
Peephole-Optimierung, 5
PGAPack, 87
Phasenkopplung, 49, 85
Phasenkopplungsproblem, 5, 85
Meta-, 51, 62
Population, 63
Pragma, 98
Programmdarstellung, 30, 101, 109
PROPAN, 28

Registerallokation, 4, 50, 101
Registerbindung, 4
Registerfile, 33
Registervergabe, 4
Ressource, 33
flüchtige, 24, 34, 108
index_read, 86, 100, 108
index_write, 86, 101
sequentielle, 23
Retargierbarkeit, 95

Scheduling
horizontal, 58
Mutation, 55
vertikal, 58
Schleifenanalyse, 40
Schleifenerkennung, 113
Schleifentransformation, 103, 113, 114
Loop-Interchange, 116
Loop-Split, 116
Loop-Unswitching, 115
Reduction-Recognition, 117
Selektion, 63, 76
Sequentialisierungskante, 85, 87, 102
SIMD, 7
SIMD-MAC-Operation, 109
SIMD-Operation, 97
SIMD-Speicherzugriff, 106
Simple-Offset-Assignment, 52
Simulation, 43
maschinenabhängig, 43
maschinenunabhängig, 43
Single-Entry Single-Exit, 113
SISD, 7
Skalarreduktion, 147
Slice, 7
SOA, 52
SoC, 2
SPAM, 28
Speicherlayout, 119
Spillcode, 5
Split-Node-Graph, 55
Standardoptimierung, 3
SUIF, 26
High-Level, 26
Low-Level, 26
Systems-on-Chip, 2

Systemvergleich, 146

Target Description Language, 28
TDL, 28
Tie-Break-Heuristik, 57
Time-to-Market, 2
Trade-Off, 135
Tree-Pattern-Matcher, 48, 53
Trellis, 54
Trimaran, 27
TWIF, 28
Typ
  abstrakter, 33
  realer, 33

Value-Nummer, 72
Variablen-Gruppierung, 124
Variablen-Zugriffsgraph, 127
Variablen-Zugriffssequenz, 124, 126
Vektordatentransfer, 8
Vektorisierung, 97, 99, 102, 110
Vektorregister, 99
Very Long Instruction Word, 8
Very Portable Optimizer, 27
VLIW, 8
VPO, 27

XeLIR, 30, 42

Zephyr, 27
Zero-Overhead Hardware-Loop, 7, 40, 111
Zurich-Zip-Datentransfer, 9, 120
Zwischendarstellung, 21, 25