Diplomarbeit

Generische Low-Level Optimierungen für RISC-Architekturen

Lars Hornbach

Diplomarbeit
am Lehrstuhl 12
des Fachbereichs Informatik
der Universität Dortmund

28. November 2001

Betreuer:

Dipl.-Inform. Lars Wehmeyer
Prof. Dr. Peter Marwedel

Inhaltsverzeichnis

Kapitel 1

Einleitung

Elektronische Geräte spielen heute eine große Rolle in unserem Leben. Angefangen vom PC bis hin zu den kleinsten mobilen technischen Entwicklungen und Neuerungen, wie z.B. Mobiltelefone oder MP3 Spieler für die Hosentasche. Vor allem im Bereich der eingebetteten Systeme im Telekomunikationsbereich oder aber auch im KFZ-Bereich gewinnen leistungsfähige Prozessoren, die auch komplexe Aufgaben kostengünstig und mit vertretbarem Aufwand meistern, immer mehr an Bedeutung.

1.1 Motivation

Um die Entwicklungszeiten kurz zu halten, ist die Programmierung in einer Hochsprache wie C der Assemblerprogrammierung vorzuziehen. Die hohen Anforderungen an die Codequalität setzen dabei den Einsatz eines optimierenden Compilers voraus.
Diese Diplomarbeit beschreibt die Implementierung von Standardoptimierungen auf der Generic Low-Level Intermediate Representation *GeLIR*.

Es besteht ein ständiger Bedarf an Optimierungen, um die Ausführungsgeschwindigkeit von Applikationen zu erhöhen, die Reaktionszeit so kurz wie möglich zu halten oder die Standbyzeit zu verlängern. In der Regel bedeutet eine Erhöhung der Ausführungsgeschwindigkeit gleichzeitig eine Senkung des Energiebedarfs.

Natürlich spielt auch die Programmgröße eine Rolle: So kann es bei bestimmten Geschwindigkeitsoptimierungen durchaus passieren, daß zugunsten der Geschwindigkeit der Speicherplatzbedarf steigt. In so einem Fall muß entschieden werden, ob die durch die Optimierung gewonnene Geschwindigkeit noch wesentliche Vorteile bringt, wenn im Gegensatz dazu der Speicherplatzbedarf und damit auch die Kosten für die eingebauten Speicherchips steigen.

Insbesondere bei mobilen Geräten ist der Energieverbrauch ein wichtiger Aspekt, denn er entscheidet, wie lange das Gerät tatsächlich mobil einzusetzen ist. Ist etwa die Einsatzzeit eines Mobiltelefons zu gering bemessen, so ist es auf dem hart umkämpften Markt nicht konkurrenzfähig.

Auf der anderen Seite wächst der Bedarf an immer mehr Funktionalität, höherer Geschwindigkeit und weiterer Miniaturisierung. So ist ein Mobiltelefon schon lange nicht mehr alleine für das Telefonieren zuständig. Terminplaner, Spiele, Internetzugang, uvm. gehören heute oft zur Grundausstattung dazu. Auch wenn das Gerät nicht aktiv benutzt wird, verbraucht es Strom, um in ständiger Bereitschaft zu sein (Standby Modus). Alle diese Faktoren wirken sich unmittelbar auf den Energieverbrauch aus.

Grundsätzlich kann man unterscheiden zwischen Optimierungsansätzen auf Hard- und Softwareseite. Diese Diplomarbeit konzentriert sich auf Optimierungen der Software unter Berücksichtigung der Ausführungsgeschwindigkeit. Im allgemeinen läßt sich dadurch gleichzeitig eine Senkung des Energiebedarfs erreichen. Da nämlich die Leistung P in der vereinfachten Energieformel $E = P * t$ bei RISC Architekturen relativ konstant über alle Befehle ist, wird deutlich, daß bei einer Verkürzung der Zeit t auch der benötigte Energiebedarf sinkt.

Die Software-Entwicklung findet heutzutage fast ausschließlich in Hochsprachen, wie z.B. C/C++, Pascal, Basic oder Java statt. Der Code wird dadurch leichter auf andere Platformen portierbar und die Entwicklungszeiten verkürzen sich um ein Vielfaches.

Definition 1.1.1 *Ein* Compiler *ist ein Softwaresystem, das ein Hochsprachenprogramm in ein äquivalentes Programmm in Objektcode oder Maschinencode übersetzt, damit es auf einem Computer ausführbar ist. Möglich sind auch Übersetzungen von einer Hochsprache in eine andere, von einer Maschinensprache in eine andere oder von einer Hochsprache in eine Zwischensprache, usw. [MUC97]*

Der Compiler hat die Aufgabe, aus dem Hochsprachencode einen möglichst effizienten Maschinencode zu erzeugen. Die Codequalität kann dabei durch verschiedene Optimierungen beeinflußt und verbessert werden:

- Speicherzugriffe/Codegröße senken
- Laufzeit verringern
- Energieverbrauch senken
- Leistungsaufnahme reduzieren

Derzeit wird an der Universität Dortmund am Fachbereich Informatik, Lehrstuhl 12 der C Compiler *encc* (energy aware C-Compiler) entwickelt, der möglichst stromsparenden Code erzeugt. In eingebetten Systemen kommen oftmals energiesparende RISC (Reduced Instruction Set Computer) Prozessoren zum Einsatz. Für zwei Typen von RISC CPUs ist der *encc* Compiler bereits vorbereitet. Momentan kann Code für ARM [ARM01] (von Advanced RISC Machines Ltd.) und LEON [LEO01] (basierend auf der SPARC V8 Architektur [SPARC99]) Prozessoren erzeugt werden. Bei beiden CPUs handelt es sich um 32-Bit Prozessoren. Die ARM CPU ist insbsondere im mobilen Bereich sehr weit verbreitet. So läuft z.B. die aktuelle Version von Microsofts Windows CE Betriebssystem auf exklusiv mit ARM Prozessoren betriebenen, stiftbedienbaren Pocket PCs.

Die RISC Architektur am Beispiel der beiden CPUs ARM und LEON wird in Kapitel 2 vorgestellt.

1.2 Ziele dieser Diplomarbeit

Ziel dieser Diplomarbeit ist die Entwicklung generischer Low-Level Optimierungen. Der Vorteil der Generizität besteht in der möglichst hohen Wiederverwendbarkeit für andere Architekturen, wobei der Schwerpunkt dieser Arbeit auf RISC-Architekturen liegt. Diese Einschränkung bedeutet, daß z.B. keine heterogenen Datenpfade ausgenutzt werden. Architekturmerkmale, die sich bei Verwendung einer anderen Zielarchitektur ändern können, werden von den Optimierungen abgefragt und berücksichtigt. Dazu zählen z.B. die Anzahl der vorhandenen Register oder die Größe von als Immediate codierbaren Zahlen. Wenn eine Optimierung Befehle ändert, kann somit sichergestellt werden, daß der neue Code nicht mehr Register benötigt als gerade frei sind, um keinen unnötigen Spillcode zu erzeugen, der evtl. Verbesserungen der Optimierung wieder zunichte macht.

Im Gegensatz zu anderen Arbeiten über Optimierungen liegt ein Schwerpunkt dieser Arbeit auf der Implementierung und Untersuchung von Optimierungen auf einer neuen Datenstruktur, die auch eine Beschreibung der Zielarchitektur beinhaltet. Dazu werden zahlreiche Beispiele gegeben.

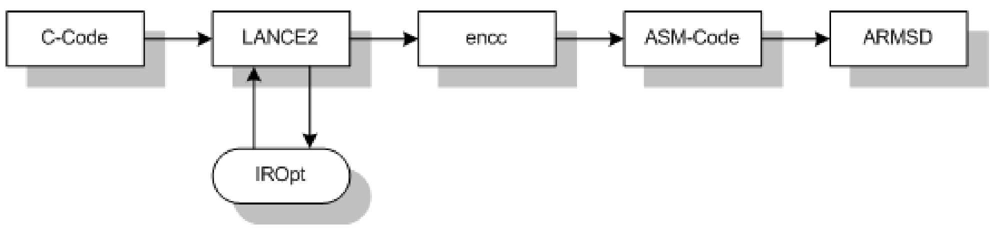

Abbildung 1.1: traditionelle C nach Assembler Übersetzung

Abbildung 1.1 skizziert beispielhaft den bisherigen Übersetzungsablauf eines C-Programms. Der C-Code wird mit Hilfe des Compiler Frontends *LANCE2* (siehe Kapitel 3) in eine Zwischencodedarstellung überführt, auf der architekturunabhängige Optimierungen ausgeführt werden. Diese Darstellung wird dann vom Compiler Backend *encc* in Assembler Code übersetzt und kann z.B. mit Hilfe des ARM Simulators *ARMSD* ausgeführt werden.

Definition 1.2.1 *Eine* Intermediate Representation (IR) *ist eine Zwischencodedarstellung eines Programms, die Ähnlichkeiten zu einer Maschinensprachendarstellung besitzt und üblicherweise im 3-Adress-Code ausgegeben wird.*

Definition 1.2.2 *Ein Compiler* Frontend *prüft das Quellprogramm einer Programmiersprache (hier C) auf syntaktische und semantische Korrektheit und generiert daraus eine Zwischencodedarstellung (IR = intermediate representation) des Programms. In diesem Stadium enthält die IR normalerweise noch keine hardwarespezifischen Informationen über den Zielprozessor.*

Definition 1.2.3 *Ein Compiler* Backend *bildet die durch das Fronend generierte Zwischencodedarstellung in ein Maschinenprogramm (z.B. Assembler, Objektcode oder Binärcode) des Zielprozessors ab. Dieser Prozess wird auch als Codegenerierung bezeichnet, mit dem Ziel der Generierung von semantisch äquivalentem und möglichst effizientem Maschinencode.*

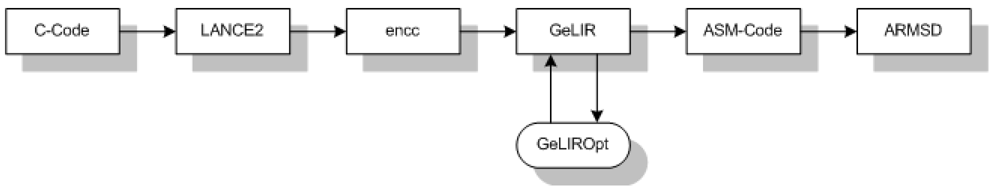

Abbildung 1.2: neue C nach Assembler Übersetzung

Abbildung 1.2 zeigt, daß die im Rahmen dieser Diplomarbeit entwickelten Optimierungen erst nach dem *encc* Backend, somit nach der Code Selektion und Register Allokation ansetzen. Durch die Funktionalität der *GeLIR* Datenstrukturen, neben der Programmdarstellung auch noch eine Architekturdarstellung zu verwalten, ergeben sich neue Möglichkeiten. So können die bei der Codegenerierung im Backend erzeugten zusätzlichen Informationen mit in die Optimierungen einfließen und Architekturinformationen berücksichtigt werden.

Stellvertretend für die große Anzahl verschiedener RISC CPUs knüpft diese Arbeit an die Arbeiten des Lehrstuhls 12 an und berücksichtigt die vom *encc* Compiler unterstützen ARM- und LEON-Architekturen. Da die ARM Unterstützung des Compilers zum Zeitpunkt dieser Arbeit am weitesten fortgeschritten ist, nimmt diese CPU auch den größten Platz dieser Diplomarbeit ein.

Optimierungen können auf High-Level oder Low-Level Ebene durchgeführt werden. Gegenstand dieser Diplomarbeit ist die Untersuchung und Implementierung von Low-Level Optimierungen. Bei der Entwicklung der Optimierungen wurde darauf geachtet, daß keine erneute Codeselektion und Registerallokation notwendig wird, da diese zum Zeitpunkt des Aufrufs der Low-Level Optimierungen bereits durchgeführt worden sind und die entsprechenden Algorithmen für RISC Prozessoren für die *GeLIR* nicht verfügbar sind.

Definition 1.2.4 Low-Level Optimierungen *arbeiten auf einer Low-Level IR, wie z.B. der* GeLIR *(siehe Kapitel 3.4). Sie werden im Designflow des* encc *Compilers nach der Codeselektion und der Registerallokation aufgerufen.*

Definition 1.2.5 High-Level Optimierungen *operieren auf einer High-Level IR. Diese Optimierungen werden vor der Codeselektion und der Registerallokation abgearbeitet. Eine High-Level Darstellung kommt in frühen Phasen des Compilierungsprozesses zum Einsatz.*

Der Bedarf an Low-Level Optimierungen ist auf jeden Fall gegeben. So können auf dieser Ebene architekturspezifische Besonderheiten ausgenutzt werden, die bisher auf High-Level Ebene nicht genutzt worden sind. Außerdem steht nach der Code Selektion detailliertere Wissen zur Verfügung. Die u.A. im Rahmen dieser Diplomarbeit implementierten Optimierungen Redundant Store Elimination (RSE) und Redundant Load Elimination (RLE) können überflüssige Load- und Store-Befehle, die erst nach Berücksichtigung von Spill Code in der Darstellung enthalten sind, entfernen. Die dafür notwendigen Informationen stehen erst nach Abschluß der Code Selektion und Register Allokation zur Verfügung.

Ein weiterer wesentlicher Punkt betrifft das "Aufräumen" nach anderen Optimierungen. So kann durch andere Optimierungen z.B. Dead Code erzeugt, der dann durch eine Low-Level Dead Code Elimination entfernt werden sollte, um Laufzeit einzusparen.

Das Hauptziel, nämlich die Implementierung von Low-Level Optimierungen zur Reduzierung der Ausführungsgeschwindigkeit und damit einhergehend evtl. auch eine Senkung des Energiebedarfs, wurde folgendermaßen erreicht:

- Anbindung der *GeLIR* an den *encc*, um eine geeignete Arbeitsumgebung zu schaffen
- Benötigte Datenflußanalyse zur Durchführung der Optimierungen einbinden
- Eigentliche Optimierungstechniken implementieren
- Eigenschaften der RISC Architektur ausnutzen, ohne den generischen Aspekt der Optimierungen zu vernachlässigen
- Test und Bewertung der erzielten Optimierungsmethoden

1.3 Einordnung in die Forschungsarbeit am Lehrstuhl 12

Es folgt eine kurze Auflistung einiger bisher am Lehrstuhl 12 durchgeführten bzw. in Arbeit befindlichen Forschungsarbeiten, die mit der Thematik dieser Diplomarbeit verwandt sind oder die gleiche Arbeitsumgebung einsetzen.

- **Schleifenoptimierungen zur Ausnutzung paralleler Rechenwerke von Prozessoren der M3-DSP Plattform [MH01]**

  Die Diplomarbeit von Martin Horst wurde parallel zu dieser Arbeit geschrieben und nutzt ebenfalls die *GeLIR* Datenstrukturen. Untersucht und implementiert wurde die Vektorisierung von Schleifen unter Ausnutzung von Hardware Pointer Registern und Zero Overhead Hardwareschleifen von M3-DSP Prozessoren. Desweiteren beschäftigt sich die Diplomarbeit mit Loop Unrolling für optimale Speicherzugriffe und Loop Peeling zur Unterstützung der Vektorisierung.
  Die Arbeit hatte praktischen Nutzen für diese Diplomarbeit, da die Datenflußanalyse in leicht modifizierter Weise auch für die Redundant Load/Store Elimination verwendet werden konnte.

- **Constraintbasierte Codegenerierung für eingebettete Prozessoren [BA00]**

  Steven Bashford verwendet in seinen Untersuchungen zur constraintbasierten Codegenerierung für eingebettete Prozessoren ebenfalls das *LANCE2* Compiler Frontend und transformiert die *LANCE2* IR Darstellung in eine *CoLIR* (Constrained based low-level intermediate representation) Darstellung, die als Vorläufer der *GeLIR* anzusehen ist. Ähnlich der *GeLIR* besitzt die *CoLIR* ein generisches Konzept zur Darstellung maschinenspezifischer Informationen.

- **XML-basierte generische Low-Level Zwischendarstellung für eingebettete Compiler (XeLIR) [MF01]**

  Auch die Diplomarbeit von Markus Fiesel wurde parallel zu dieser Arbeit geschrieben. Ausgehend von der *GeLIR* Datenstruktur wurde eine weitere generische Low-Level IR *XeLIR* entworfen, die auf der Basis von XML [XML01] u.a. ein persistentes Speichern und Wiederherstellen der *GeLIR* Datenstruktur ermöglicht. Zusätzlich wurde ein Programm zur patternbasierten Assemblercodegenerierung auf XML-Basis entwickelt, das bereits in dieser Diplomarbeit verwendet worden ist.
  Außerdem ist es möglich, gewisse Optimierungen direkt auf der XML-Darstellung durchzuführen. Die Beschreibungskomplexität z.B. einer Peep-Hole-Optimierung ist hierbei sehr gering.
  Desweiteren ging aus Markus Fiesels Arbeit der *GeLIR Simulator* hervor. Mit seiner Hilfe läßt sich etwa sicherstellen, daß eine Optimierung das Programm semantisch unverändert läßt.

1.4 Übersicht

Die vorliegende Diplomarbeit ist wie folgt aufgebaut:

Im anschließenden Kapitel 2 werden die RISC Prozessoren ARM7TDMI und LEON anhand ihrer wichtigsten Eigenschaften vorgestellt. Der ARM Prozessor ARM7TDMI wurde bei der Implementierung der Optimierungen besonders berücksichtigt.

Für die Low-Level Darstellung fiel die Wahl auf die ebenfalls am Lehrstuhl 12 entwickelte *GeLIR* (Generic Low-Level Intermediate Representation). Kapitel 3 gibt einen Überblick über die verwendete Arbeitsumgebung, insbesondere die *GeLIR* und die Anbindung des *encc* Compilers an die *GeLIR*.

In Kapitel 4 werden die nötigen Grundlagen der Datenflußanalysemethoden vermittelt und die bei den Load/Store Optimierungen zum Einsatz kommende Delta-Array-Datenflußanalyse vorgestellt.

Im darauffolgenden Kapitel 5 werden Low-Level Standard Optimierungen detailiert beschrieben. Es wird ein Überblick über die bekanntesten Standard Optimierungen gegeben und es werden architekturspezifische Details besprochen, die bei der Implementierung der Optimierungen vorteilhaft ausgenutzt werden können.

Die Load/Store Optimierungen aus Kapitel 6 verwenden die in Kapitel 4 vorgestellte Delta-Array-Datenflußanalysemethode.

Die erzielten Optimierungsergebnisse werden in Kapitel 7 ausführlich bewertet und analysiert.

Abschließend faßt Kapitel 8 die wichtigsten Aspekte dieser Diplomarbeit zusammen und gibt einen Ausblick auf mögliche weiterführende Arbeiten.

Im Rahmen dieser Diplomarbeit wurde u.a. auch die Architektur des ARM7TDMI Prozessors in die *GeLIR* übertragen. Andere Architekturen können analog eingebunden werden. Die Vorgehensweise ist in Anhang A dokumentiert.

Anhang B dokumentiert ab Seite 123 die komplette Implementierung der Optimierungen, gibt Tips zum praktischen Einsatz und weist auf mögliche Erweiterungen des Quellcodes hin.

Anhang C erläutert die Programmierrichtlinien, die bei der Implementierung befolgt wurden.

Kapitel 2

RISC-Architekturen

Im Rahmen dieser Diplomarbeit wurden verschiedene Low-Level Optimierungen für RISC Architekturen implementiert. Neben einigen Grundlagen zur RISC Architektur stellt dieses Kapitel die bisher vom *encc* Compiler unterstützen RISC Architekutren ARM7TDMI und LEON vor, da einige Architekturmerkmale für die Optimierungstechniken von Interesse sind und teilweise mit in die Implementierung eingeflossen sind.

2.1 Allgemeines

Folgende Merkmale zeichnen eine RISC Architektur im Gegensatz zu einer CISC (Complex Instruction Set Computer) Architekur aus: [MAR00]

- Reduzierter Befehlssatz
- CPI (Cycles per Instruction) Wert nahe 1 realisiert durch:

  Wenige, einfache Adressierungsarten

  Load/Store Architekur

  Feste Befehlswortlängen
- Optimierungen erfolgen in der Regel durch den Compiler

Durch den reduzierten Befehlssatz und der damit verbundenen wesentlich einfacheren Befehlsdekodierung erreichen RISC CPUs einen sehr hohen Instruktionsdurchsatz. Außerdem ermöglicht die Architektur eine gute Echtzeit-Interrupt-Reaktion auf meist kleinen, kosteneffektiven Chips.

2.2 ARM7TDMI

Die ARM7 Prozessorfamilie von Advanced RISC Machines Ltd. [ARM01] besteht aus einem Low Power 32-Bit RISC Prozessor. Die CPU ist relativ preiswert und ermöglicht die Ausführung stromsparender Anwendungen. Der Kern wurde nach und nach den Anforderungen des Marktes angepasst. Zur Zeit (Stand Mitte 2001) stehen folgende Prozessorkerne zur Verfügung [ARM01]:

- ARM7TDMI: Integer Kern
- ARM7TDMI-S: Synthesefähige Version des ARM7TDMI Kerns (z.B. in VHDL oder Verilog)
- ARM7EJ-S: Synthesefähige Verion mit DSP und Jazelle Erweiterungen zur Java Beschleunigung
- ARM720T: Kombinierter ARM7TDMI Kern mit 8K Cache und voller MMU (Memory Management Unit)

Die Wahl des geeigneten Kerns hängt von der gewünschten Geschwindigkeit (bis zu 130MIPs – bestimmt mit Dhrystone 2.1), dem Betriebssystem und den Speicheranforderungen ab. Für kleine Anwendungen (z.B. im mobilen Telekomunikatiosbereich, Kameras oder PDAs) bietet sich der ARM7TDMI Kern an.

Werden erweiterte DSP oder Java Fähigkeiten benötigt, greift man zum ARM7EJ und bei Systemen unter WindowsCE, Palm OS, Symbian OS, Linux oder Echtzeitbetriebssystemen eignet sich der ARM720T am besten.

Der im 0.18μm Prozess gefertigte ARM7TDMI bietet auf Grund seiner kleinen Größe ($<$0.53mm^2) und 3600 MIPs/W Strom-Effizienz die längste Batterielebenszeit bei Applikationen, die komplett auf den Chip gespeichert werden können.

Diese Diplomarbeit beschäftigt sich ausschließlich mit der ARM7TDMI Variante, die neben der LEON CPU auch vom *encc* Compiler des Lehrstuhls 12 unterstützt wird.

Die folgenden Kapitel beziehen sich, wenn nicht gesondert angegeben, generell auf den ARM7TDMI auch wenn z.B. nur von einer ARM CPU die Rede ist.

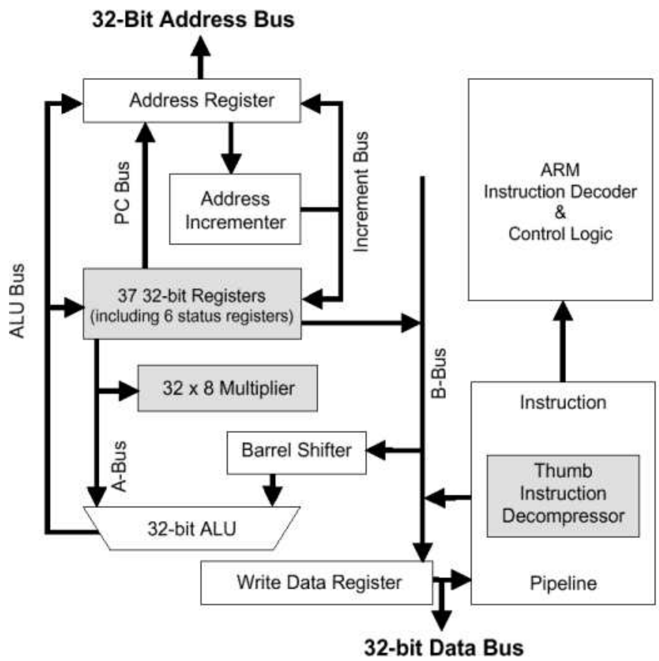

Abbildung 2.1: ARM7TDMI CPU [ATM99]

2.2.1 Eigenschaften des ARM7TDMI

Die folgenden Unterkapitel stellen den ARM7TDMI nach [ARM01] und [ARM99] dar. Die ARM CPU (siehe Abbildung 2.1) besitzt drei Funktionseinheiten: Eine ALU, einen Barrel-Shifter sowie eine Multipliziereinheit. Folgende Merkmale zeichnen den ARM7TDMI aus:

- 32-Bit RISC CPU, 1-130MIPs, kleine Die Größe
- 65-100 MHz Taktfrequenz
- Zwei Befehlssätze:
  32-Bit ARM Befehlssatz
  16-Bit THUMB (Low Power) Befehlssatz
- 37 32-Bit Register, davon 6 Status Register
  16 General Register (ARM Modus)
  8 General Register (THUMB Modus)
- 32-Bit ALU mit eigenständigem Barrel Shifter
- 32-Bit Multiplizierer
- 32-Bit Adress- und Datenbus

- Load/Store Architektur
- 3-stufige Instruktions-Pipeline
- Datentypen (im Speicher linear angeordnet):
  Byte, Halbwort, Wort
- Big Endian und Little Endian werden unterstützt
- Geringer Stromverbrauch: <0.25mW/MHz bei einem 0.18μm Fertigungsprozess
- Hohe Codedichte, vergleichbar mit 16-Bit Mikrokontrollern
- Verfügbar in 0.25μm, 0.18μm und 0.13μm
- Wird von zahlreichen (Echtzeit)-Betriebsystemen unterstützt
- Große Auswahl an Entwicklungswerkzeugen und Simulationsanwendungen
- Programmcode ist vorwärts-kompatibel mit ARM9, ARM9E und ARM10 CPUs, sowie mit Intel's StrongARM und XScale Produkten

Einige der genannten Komponenten sind in Abbildung 2.2 dargestellt.

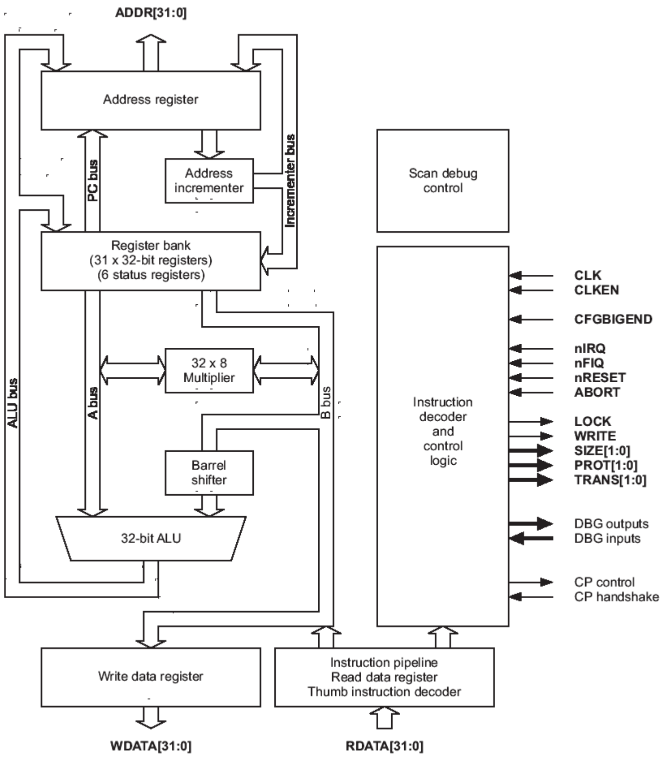

Abbildung 2.2: Datenpfad des ARM7TDMI

2.2.2 Das THUMB Konzept des ARM7TDMI

Der ARM besitzt neben seinem 32-Bit Befehlssatz einen weiteren 16-Bit Befehlssatz, den sog. Thumb Befehlssatz. Im Thumb Modus benötigt eine Instruktion nur 16-Bit statt 32-Bit im ARM Modus.

Der Thumb Befehlssatz umfasst eine Untermenge von 36 der meistgenutzen der insgesamt 80 Befehle aus dem ARM Modus. Beim Ausführen von Thumb Instruktionen erfolgt in der Dekodierphase ein automatisches Umsetzen in den ARM Befehlssatz innerhalb eines Taktzyklus. Für jede Thumb Instruktion gibt es eine gleichwertige 32-Bit ARM Instruktion, so daß ARM- und Thumb Modus problemlos zusammenarbeiten können.

Dieser reduzierte Befehlssatz erreicht eine höhere Speicherdichte im Vergleich zum normalen ARM Code, profitiert aber trotzdem noch von den Performance Vorteilen des ARM7TDMI im Gegensatz zu reinen 16-Bit Prozessoren mit 16-Bit Registern, da der Thumb Code auf den gleichen 32-Bit Registern arbeitet wie der reine ARM Code.

Gegenüber einer reinen 16-Bit Lösung ist der ARM im Thumb Modus somit in der Lage, 32-Bit Integer mit einem Befehl zu manipulieren. Der größere 32-Bit Adressraum kann effizient angesprochen werden. Außerdem können 32-Bit Shifter und ALU (Arithmetic logic unit), 32-Bit Register und 32-Bit Speicher Transfers genutzt werden. Ein Wechsel in und aus dem ARM Modus ist jederzeit möglich, so kann z.B. für zeitkritische Schleifen bei Interrupts oder für DSP Algorithmen der ARM Modus genutzt werden. [ARM95a]

Durch die auf 16-Bit reduzierte Befehlswortbreite im Thumb Modus sind einige Adressierungsarten aus Platzgründen im Gegensatz zum ARM Modus eingeschränkt.

Im Vergleich zum ARM Modus sinkt die Code Größe im Thumb Modus auf etwa 65%. Die Ausführungsgeschwindigkeit des Codes ist im Thumb Modus um etwa 160% höher als bei einer vergleichbaren CPU, die nur mit einem 16-Bit Speicherinterface ausgestattet ist. [ARM99]

2.2.3 Registerorganisation des ARM7TDMI

Die ARM CPU besitzt insgesamt 37 32-Bit Register, davon 31 allgemeine Register und 6 Status Register. Im ARM Modus sind 16 Register gleichzeitig sichtbar und je nach Prozessormodus 1-2 Status Register. Ein direkter Zugriff ist auf die 16 Register R0 bis R15 möglich. R15 speichert den Befehlszähler (PC), R14 wird in der Regel als Unterprogramm-Linkregister (LR) benutzt und R13 als Stack Pointer (SP). Die restlichen Register können allgemein für Daten und Adressen benutzt werden.

Der aktuelle Prozessorzustand wird in einem 17. Register, dem CPSR (Current Program Status Register) gespeichert.[ARM99]

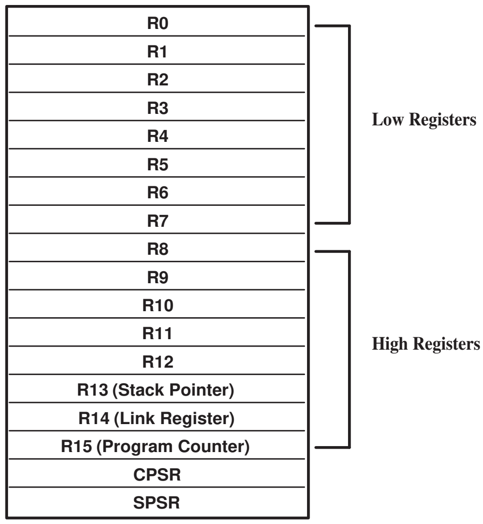

Abbildung 2.3: Registersatz im Thumb Modus [ARM95b]

Im Thumb Modus (siehe Abbildung 2.3) stehen nur 8 Register, R0 bis R7 (untere Registerbank), zur freien Verfügung. Nur einige Thumb Befehle können die Register R8 bis R12 (obere Registerbank) ansprechen, so daß die hohen Register als schnelle Zwischenspeicher genutzt werden können:

- Die MOV Instruktion kann Werte zwischen Low- und High-Registern (bzw. zwischen High- und Low-Registern) bewegen.
- Die CMP Instruktion kann Werte aus den hohen Registern mit niedrigen Registern vergleichen.
- Die ADD Instruktion ermöglicht die Addtion von Werten aus hohen Registern mit Werten aus niedrigen Registern.

Da für die hohen Register nur ein begrenzter Befehlsvorrat zur Verfügung steht, werden diese z.B. von dem kommerziellen Standardcompiler 'tcc' nicht genutzt. Mit Hilfe der in der *GeLIR* enthaltenen Modellierungstechniken ist es kein Problem, zukünftig auch die oberen Register mit zu benutzen. Der *encc* Compiler nutzt die hohen Register bereits als Vorstufe zum Spilling. Dies bietet sich an, da im hier berücksichtigten Thumb Modus lediglich 8 Register zur freien Verfügung nutzbar sind.

R13 speichert den Stack Pointer, R14 sichert die Rücksprungadresse bei Funktionsaufrufen und R15 nimmt den Program Counter auf. Dort wird auch die Rücksprungadresse bei Ausnahmebehandlungen gespeichert.

Außerdem stehen im Thumb Modus die beiden Status Register CPSR und SPSR (Saved Process Status Register) zur Speicherung diverser Zustands-Bits zur Verfügung.

Abbildung 2.4 zeigt die Beziehungen zwischen den Registern im ARM und Thumb Modus. Die hohen Register R8 bis R12 sind in der linken Seite der Darstellung nicht eingetragen, da der Großteil der Befehle im Thumb Modus keinen Zugriff auf diese Register hat.

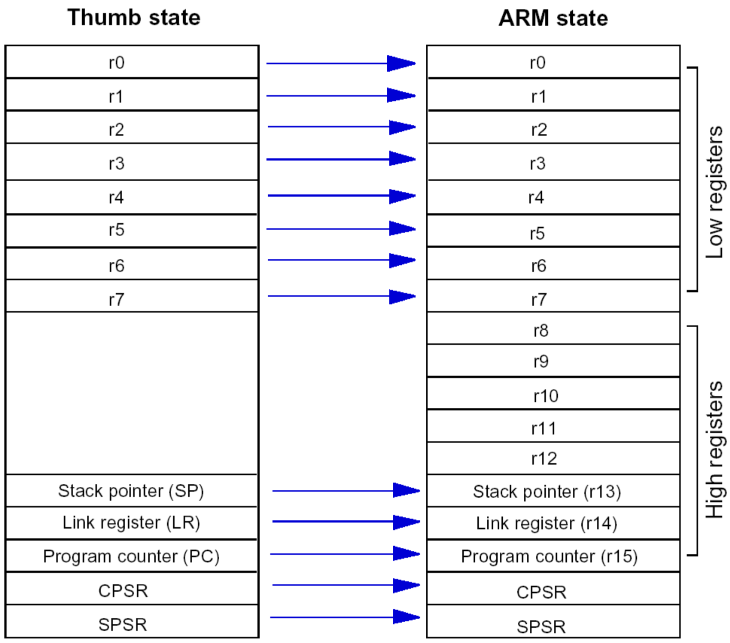

Abbildung 2.4: Beziehungen zwischen ARM- und Thumb Modus Registern [ARM99]

2.2.4 Der Scratch Pad Speicher des ARM7TDMI

Definition 2.2.1 *Bei einem Scratch Pad Speicher handelt es sich um einen schnellen Zwischenspeicher, ähnlich einem Cache. Im Gegensatz zum Cache-Speicher fehlt allerdings die Verwaltunglogik (z.B. Tag Memory und Vergleichs-schaltung). Der Programmierer bzw. Compiler muß sich selber um die Verwaltung des Scratch Pad Speichers kümmern, besitzt da durch aber aber auch alle Freiheiten im Umgang mit diesem Speicher. Man spricht auch von Compiler-Controlled Memory [CH98].*

Vorteilhaft für die Ausführungsgeschwindigkeit ist es, wenn möglichst viel Programmcode in den 4KB großen Scratch Pad On-Chip Speicher des ARM passt.

Der Scratch Pad Speicher des ARM wurde bereits in diversen Arbeiten der Universität Dortmund am Fachbereich Informatik, Lehrstuhl 12 untersucht. Christoph Zobiegala hat in seiner Diplomarbeit "Energieeinsparung durch compilergesteuerte Nutzung eines On-Chip-Speichers" [ZOB01] u.a. die statische Verwaltung des Scratch Pad Speichers untersucht.
Die dynamische Nutzung des Scratch Pad Speichers wird Nils Grunwald in seiner Diplomarbeit "Energieoptimierung durch dynamische Nutzung eines Scratchpad-Speichers" untersuchen.
Einen Vergleich zwischen dem Scratch Pad Speicher und einem Cache kann in der Arbeit "Vergleich von Caches und Scratch-Pad Speichern" [LEE01] von Bo-Sik Lee nachgelesen werden.

2.3 LEON

Die folgenden Abschnitte geben einen Überblick über den LEON Prozessor nach [LEO01].

Beim LEON Prozessor der schwedischen Firma Gaisler Research [GAIS] handelt es sich um das VHDL Modell eines SPARC V8 (Scaleable Processor ARChitecture Version 8) [SPARC99] konformem 32-Bit RISC Prozessors. Der Prozessor wurde ursprünglich von der European Space Agency (ESA) namentlich von Jiri Gaisler entwickelt. Das LEON Modell wird unter der GNU Public License (GPL) und der Lesser GNU Public License (LGPL) vertrieben, so daß der LEON Kern in einem eigenen Chip Design verwendet werden darf.

2.3.1 Eigenschaften der LEON CPU

Der LEON zeichnet sich durch folgende Eigenschaften aus:

- Getrennte Daten- und Instruktions-Caches
- Hardware Multiplizierer und Dividierer (optional)
- Interrupt Controller
- Zwei 24-Bit Zeitgeber
- Zwei UARTs
- Stromspar Funktionen
- Watchdog
- 16-Bit I/O Port
- Flexibler Speicher Controller

2.3.2 Aufbau der LEON CPU

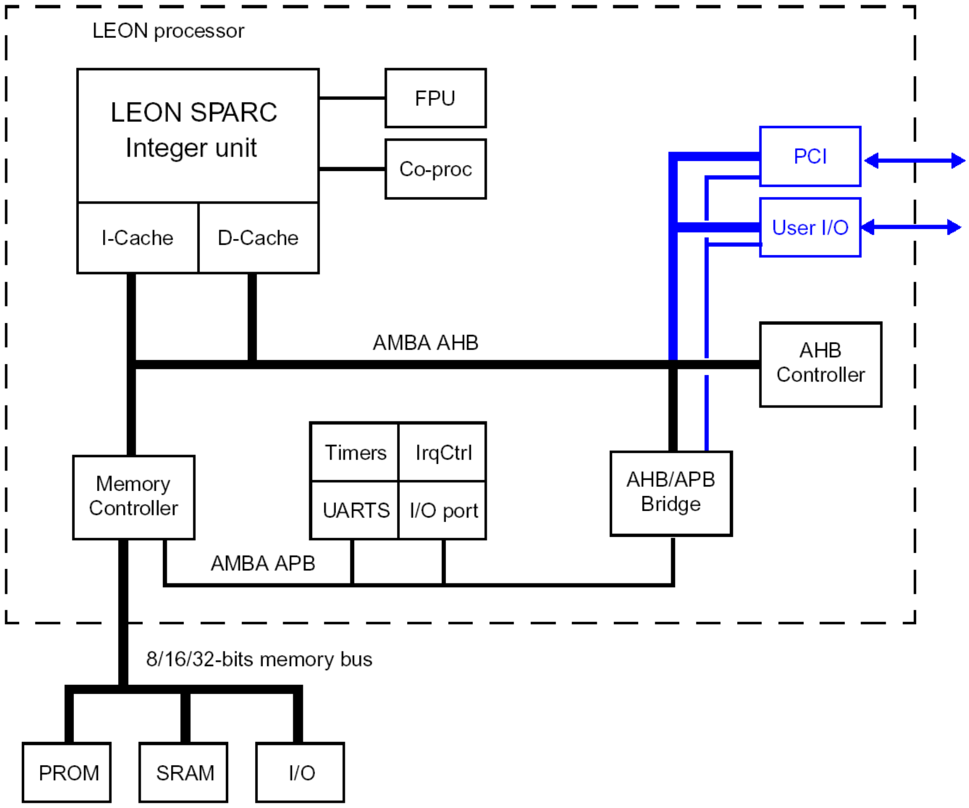

Abbildung 2.5: LEON Blockschaltbild [LEO01]

Abbildung 2.5 zeigt das Blockschaltbild des LEON mit folgenden Einheiten:

- **Integer Einheit**

  Die LEON Integer Einheit hält sich inklusive der Multiplizier- und Dividier-Instruktionen an den SPARC V8 Standard [SPARC99]. Die Anzahl der Registerfenster (siehe Unterkapitel 2.3.3) beträgt in der Grundeinstellung 8 und ist im Bereich zwischen 2 und 32 frei definierbar. Die Einheit arbeitet mit einer 5-stufigen Befehlspipeline: Instruction Fetch – Decode – Execute – Memory – Write.

  Zur Beschleunigung von DSP Algorithmen unterstützt der LEON optionale MAC-Instruktionen (Multiply ACumulate Operation).

- **Fließkomma Einheit und Koprozessor**

  Der LEON bringt keine eigene Fließkommaeinheit mit, besitzt aber eine Schnittstelle für einen FPU Kern und einen Coprozessor.

- **Cache Einheit**

  Der LEON stellt getrennte, direct-mapped Daten- und Instruktions-Caches zur Verfügung, die voneinander unabhängig zwischen einem und 64 KB Größe und 8-32 Bytes pro Cache Line konfiguriert werden können. Der Daten Cache arbeitet im Write-Through Verfahren.

- **Speicher Interface**

  Das Speicherinterface bietet über einen 8/16/32-Bit Datenbus (einstellbar) Zugang zu PROM, SRAM und Memory Mapped I/O Geräten. Der Speicher Controller kann einen Adressraum von 2GB dekodieren, wobei sich die ersten 512MB auf PROM Mapping, die weiteren 512MB auf das I/O Mapping und das letzte 1GB auf das RAM Mapping beziehen.

- **Timer/Watchdog**

  Die CPU besitzt zwei 24-Bit Zeitgeber und einen 24-Bit Watchdog

- **UARTs**

  Zwei 8-Bit UARTs zur seriellen Kommunikation, deren Baudrate individuell programmiert werden kann, stehen zur Verfügung.

- **Interrupt Controller**

  Der Interrupt Controller verwaltet 15 Interrupts von internen und externen Quellen. Mit einem optionalen zweiten Controller sind weitere 32 Interrupts verfügbar.

- **I/O Port**

  Hierbei handelt es sich um einen 16-Bit parallelen I/O Port, der als Eingang oder Ausgang programmiert werden kann.

- **AMBA Bus**

  LEON verfügt über eine vollständige Implementation des AMBA AHB und AMBA APB On-Chip Bus. Bei AMBA handelt es sich um einen offenen Standard [ARM01] einer On-Chip Bus Spezifikation für eingebettete Systeme. Der AMBA APB Bus dient dem Zugriff auf On-Chip Registern von Peripherie Funktionen (z.B. Timer), während der AMBA AHB für High-Speed Datentransfers verwendet wird. So werden z.B. Cache- und Memory Controller über den AHB Bus verbunden.

2.3.3 Registerorganisation des LEON

Eine Implementierung einer Integer Einheit nach SPARC V8, wie sie auch der LEON verwendet, kann zwischen 40 und 520 frei verwendbare 32-Bit Register besitzen. Die Register sind unterteilt in 8 globale Register und eine implementationsabhängige Anzahl von teilweise überlappenden 24-Register Mengen, die in 8 *In*-, 8 *Local*- und 8 *Lokale*-Register unterteilt sind. [SPARC99]

Zum Ansprechen der Register benutzt der LEON die Registerfenster-Technologie. So stehen einer Prozedur jederzeit 32 Register zur Verfügung, nämlich die 8 globalen Register und ein Registerfenster aus 24 Registern. Das Registerfenster umfasst 8 *In*- und 8 *Lokale*-Register einer Register Menge sowie 8 *In*-Register eines benachbarten Register Sets, die vom aktuellen Fenster als 8 *Out*-Register adressierbar sind. Tabelle 2.1 macht diese Umsetzung deutlich.

Registerfenster Adresse	Register Adresse
in[0] – in[7]	r[24] – r[31]
local[0] – local[7]	r[16] – r[23]
out[0] – out[7]	r[8] – r[15]
global[0] – global[7]	r[0] – r[7]

Tabelle 2.1: Fenster Adressierung

Die Anzahl der Register Mengen NWINDOWS variiert je nach Implementierung zwischen 2 und 32. Die gesamte Anzahl der Register setzt sich folgendermaßen zusammen:

$$\text{Anzahl Register} = 8 + \text{NWINDOWS} * 16$$

Hierbei steht die 8 für die globalen Register und die 16 für die (nicht-überlappende) Anzahl der Register eines Sets. Daraus folgt ein Minimum von 40 Registern und ein Maximum von 520 Registern.

Das aktuelle Registerfenster wird durch einen 5-Bit Zähler (CWP - Current Window Pointer) im Prozessor Status Register (PSR) verwaltet.

Jedes Registerfenster teilt seine *In*- und *Out*-Register mit zwei benachbarten Registerfenstern. So sind die *Out*-Register von CPW+1 als *In*-Register des aktuellen Registerfensters adressierbar. Abbildung 2.6 verdeutlicht das Verfahren für NWINDOW = 8.

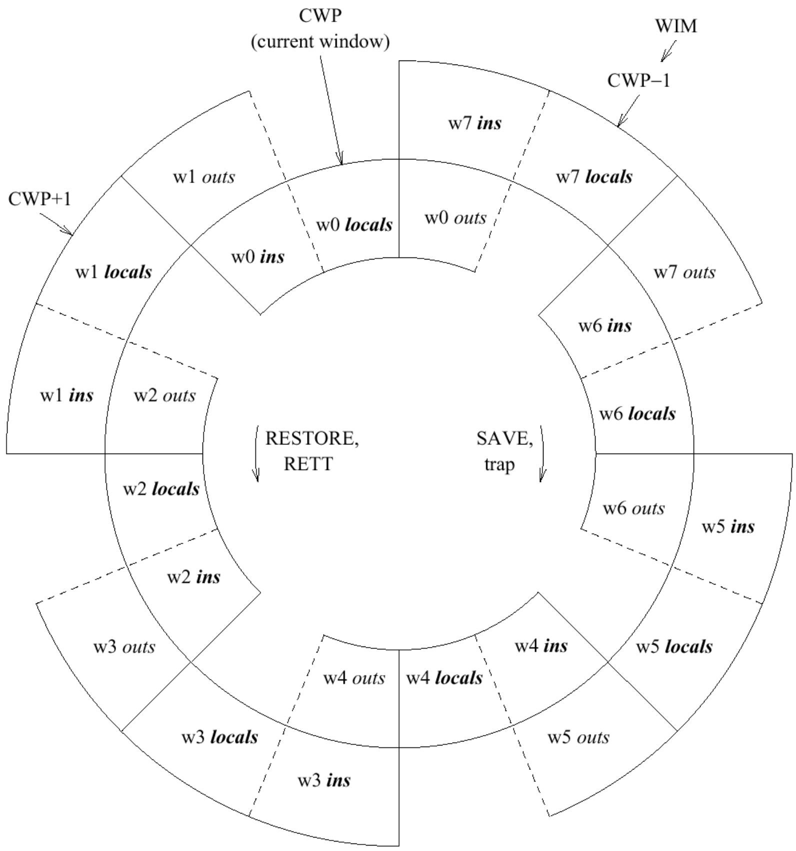

Abbildung 2.6: Registerfenster [SPARC99]

Die 8 globalen Register sind in Abbildung 2.6 nicht abgebildet.

2.3.4 Instruction Pipeline

Die LEON Integer Einheit benutzt eine Single-Instruction Pipeline bestehend aus fünf Stufen:

- **Instruction Fetch (FE)**

  Am Ende dieser Stufe ist eine Instruktion gültig. Die Instruktion wird direkt aus dem Instruktion Cache geholt, sofern dieser aktiviert ist. Ansonsten erfolgt die Befehlsholung durch den Memory Controller.

- **Decode (DE)**

  Die Instruktion wird dekodiert und die Operanden, die aus Register Files oder internen Datenbeipässen stammen können, gelesen. Außerdem erzeugt diese Stufe CALL– und Sprung-Adressen.

- **Execute (EX)**

  Im Falle einer ALU–, logischen oder Shift-Operation wird diese ausgeführt. Bei Speicheroperationen und Sprungbefehlen wird die benötigte Adresse generiert.

- **Memory (ME)**

  Hier findet ein Zugriff auf den Datencache bzw. Speicher statt.

- **Write (WR)**

  Das Ergebnis einer ALU–, logischen, Shift– oder Cache Leseoperation wird zurück in das Register File geschrieben.

2.3.5 Branch Delay Slot

Da die LEON CPU einen Branch Delay Slot der Länge 1 besitzt, muß dies bei der Assembler Code Erzeugung in geeigneter Weise berücksichtigt werden. Momentan fügt der *encc* Compiler einen *NOP* (No Operation Befehl) nach jeder Sprunganweisung ein.
Bei der Verwendung eines Branch Delay Slots wird während der Berechnung der Sprung-Adresse in der Pipeline bereits der dem Sprungbefehl folgende Assembler Befehl geholt und ausgeführt. Hierbei muß berücksichtigt werden, daß der folgende Befehl, unabhängig davon ob gesprungen wird oder nicht, ausgeführt wird. Da nicht sichergestellt ist, ob der entsprechende Sprungbefehl ausgeführt wird, dürfen nur Befehle geholt werden, die das Programmverhalten nicht verändern. Eine *NOP* Operation ist unkritisch, da diese keine Änderung des Prozessorzustands herbeiführt.
Hier könnte eine Optimierung ansetzen und aus dem Branch Befehl vorangegangen Code eine geeignete Assembler Instruktion heraussuchen und diese an Stelle des *NOP* Befehls einfügen.

Kapitel 3

Arbeitsumgebung

Dieses Kapitel stellt die Arbeitsumgebung, in der die Optimierungen entwickelt worden sind und die dabei benutzen Werkzeuge vor.

3.1 Verwendete Werkzeuge

Die im Rahmen dieser Diplomarbeit entwickelten Konvertierungen und Optimierungen fügen sich nahtlos in bereits bestehende Tools ein, nutzen deren Funktionalität und stellen neue Funktionen zur Verfügung.
Es wurden hauptsächlich folgende Werkzeuge verwendet:

- LANCE2 Compiler Frontend
- encc Compiler
- aiSee Visualisierungstool
- GeLIR Klassenbibliothek
- GeLIR Simulator
- XeLIR (XML basierte GeLIR-Darstellung zur Assemblerausgabe)

3.1.1 LANCE2

LANCE v2.0 (*LANCE2*) ist ein Compiler Frontend für die Programmiersprache ANSI C, deren Sprachstandard 1988 vom ANSI-Kommitee X3J11 veröffentlicht worden ist [KR88]. Die Abkürzung LANCE steht für "LS12 ANSI-C Environment". Er wurde an der Universität Dortmund im Fachbereich Informatik am Lehrstuhl 12 entwickelt und ist für die SUN Solaris Plattform (ab Version 2.8),

Linux (ab Kernel Version 2.2.16) und für Windows 2000 verfügbar.
Dieses Werkzeug übersetzt ANSI C Code in eine einfache maschinenunabhängige Zwischencodedarstellung (Intermediate Representation IR). Für die Generierung von Assembler Code, maschinenspezifische Optimierungen und das Linken von Code ist ein externes Compiler Backend notwendig. Die mitgelieferte C++ API bietet Zugriffsfunktionen für die IR und erlaubt beispielsweise die Entwicklung neuer Optimierungen auf IR Ebene oder die Anbindung externer Tools.
Die *LANCE2* IR ist architekturunabhängig und eignet sich aus diesem Grund als Basis für ANSI C Compiler für beliebige Architekturen. Durch ihre API wird zudem die Entwicklung architekturspezifischer Backends ermöglicht.

Das *LANCE2* System zeichnet sich durch folgende Merkmale aus [LEU01]:

- ANSI C 89 Frontend
- Eingebaute IR Optimierungen
- Bibliothek externer IR Optimierungswerkzeuge
- Ausführbare IR in Low-Level C Syntax als 3-Adress-Code
- C++ API Bibliothek inkl. Zugriffs- und Manipulationsfunktionen für die IR Darstellung
- Unterstützung für die Visualisierung von Kontroll-/Datenfkuß- und Aufruf-Graphen
- Interface zur Assembler Code Generierung

LANCE2 ist als Hilfsmittel für die Compilerentwicklung für eingebettete Systeme gedacht und wurde bereits in verschiedenen Froschungs- und Industrieprojekten verwendet, u.a. auch beim *encc* Compiler [LEU01].

3.1.2 encc

Der *encc* ist ein "energy aware C-Compiler", der zu Forschungszwecken im Fachbereich Informatik, am Lehrstuhl 12 der Universität Dortmund entwickelt wurde. Er ist für Solaris ab Version 2.8 und für Linux (getestet mit SuSE Linux ab Version 6.4) verfügbar.

Der Compiler besteht aus den folgenden Komponenten [SW01]:

- *LANCE2* Frontend
- Backend für den 16-Bit Thumb Befehlssatz des ARM7DTMI Prozessors
- Backend für den LEON 32-Bit Prozessor

- Mehrere Anpassungen an das originale ARM SDT 2.50 Kit
- Energie Profiler

Zur Umsetzung des C-Codes in eine 3-Adress-Code Zwischendarstellung verwendet der *encc* das *LANCE2* Frontend, das bereits Tools zur High-Level Optimierung auf dieser IR beinhaltet. Dazu zählen beispielsweise Jump Optimization, Dead Code Elimination und Common Subexpression Elimination.

Das *encc* Compiler Backend verwendet einen Tree Pattern Matching Algorithmus zur Durchführung der Code Selektion und eine heruristische, auf Graphfärbung basierende Methode zur Registerallokation. Anschließend werden die eingebauten Optimierungen des Backends durchgeführt, u.a. Instruction Scheduling, Register Pipelining und Memory Mapping Strategien. Im Rahmen dieser Arbeit wird das *encc* Compiler Backend um Low-Level Optimierungen erweitert.
Desweiteren beinhaltet der *encc* Compiler eine Datenbank mit Informationen über den Energieverbrauch jeder einzelnen Instruktion des ARM7TDMI Prozessors. Diese Daten wurden durch physikalische Messungen an der CPU gewonnen und erlauben die Auswertung von Optimierungen unter Energieverbrauchsgesichtspunkten. Diese Aufgabe übernimmt der Energie Profiler. Er berücksichtigt die tatsächlich ausgeführten Instruktionen und die Kosten der stattgefundenen Speicherzugriffe, die je nach Speichertyp (z.B. on-chip oder off-chip) unterschieden werden.
Außerdem kann der *encc* Compiler Code für die *LEON* CPU generieren, der ebenfalls unter dem Aspekt des Energieverbauchs untersucht werden kann.

Der Quelltext des zu übersetzenden Programms wird im C-Code an das *LANCE2* Compiler Frontend übergeben. Dieses erzeugt die *LANCE2* IR als Zwischencode, auf der bereits vorhandene High-Level Transformationen und Optimierungen durchgeführt werden. Die *LANCE2* IR wird dann vom *encc* Compiler in einen Kontrollflußgraphen überführt, um letztendlich Assembler Code für ARM oder LEON CPUs zu generieren.

Der Assembler Code für ARM CPUs kann beispielsweise mit Hilfe des ARM Assemblers und Linkers in ein ausführbares Executable transformiert werden, welches dann von dem ARM Simulator *ARMSD* (ARMulator) [ARM01] ausgeführt wird. Der Simulator liefert u.a. Daten über die Art und Anzahl der ausgeführten Instruktionen und über Zugriffe auf die verschiedenen Speicherarten. Ein ebenfalls an der Universität Dortmund, Fachbereich Informatik, Lehrstuhl 12 entwickelter Energieprofiler kann die Informationen des Simulators verarbeiten und mittels einer eigenen Datenbank, die Informationen über den Energieverbauch jedes Instruktionstyps und des Speicherzugriffs beinhaltet, eine Performance-Statistik ausgeben.
Weitere Details zum ARM Simulator liefert Kapitel 7.

Die Darstellung eines Programms als Kontrollflußgraph ist für Analyseaufgaben in Compilern am ehesten geeignet, da z.B. Informationen der Art, welches

Statement von welchem aus erreichbar ist, einfach abgefragt werden können. Der Quelltext oder eine rein textuelle IR Darstellung ist dafür nicht geeignet.

Definition 3.1.1 *Ein* Kontrollflußgraph *CFG (Control Flow Graph) ist ein gerichteter Graph* $G = (N, E, s, e)$ *mit Knotenmenge* N*, einer Menge von Kanten* $E \subseteq N \times N$*, einem Startknoten* s *und einem Endknoten* e *mit* $s, e \in N$*. Ist* n *direkter Vorgänger von* m*, so gilt* $(n.m) \in E$ *mit* n *ist Vorgänger von* m *und* m *ist Nachfolger von* n*. Jeder Knoten repräsentiert eine Instruktion und jede Kante einen Kontrollflußübergang zwischen zwei Instruktionen. Der Startknoten* s *hat keinen Eingang und der Endknoten* e *hat keinen Ausgang. Diese Eigenschaft kann durch Einfügen eines expliziten Endknotens realisiert werden. [BF99]*

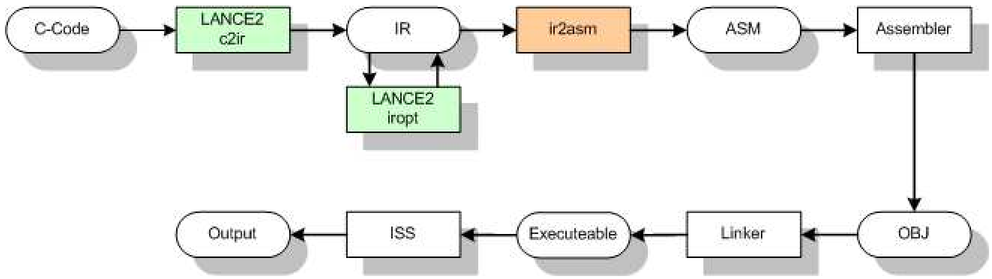

Abbildung 3.1: encc Ablaufdiagramm

Abbildung 3.1 zeigt ein Ablaufdiagramm des *encc* Compilers. Der C-Code wird durch das *LANCE2* Tool `c2ir` in eine Zwischencodedarstellung (IR) überführt, auf der optional einige *LANCE2* Optimierungen (`iropt`) angewendet werden können. Die IR Darstellung wird dann durch das `ir2asm` Modul, in dem die Hauptfunktionalität des *encc* Compilers steckt, in eine Assembler Darstellung transformiert. Der Assemblercode wird dann durch einen Assembler in einen Objektcode übersetzt und durch einen Linker in ein Executable überführt, das dann einem Instruction Set Simulator übergeben wird, um eine Ausgabe zu erzeugen.

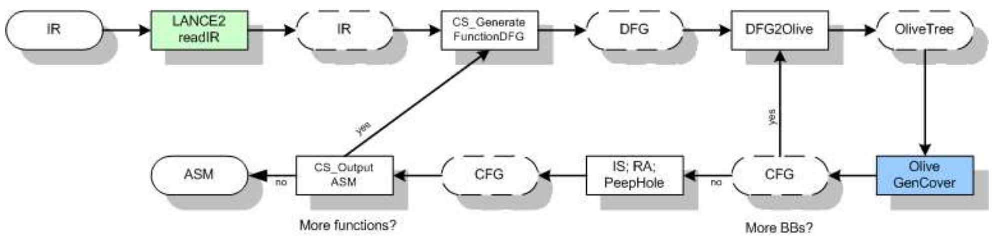

Abbildung 3.2: ir2asm Details aus encc Ablaufdiagramm

Der genaue interne Ablauf des *encc* Moduls `ir2asm` wird in Abbildung 3.2 dargestellt. Das *LANCE2* `readIR` Modul überführt die Dateidarstellung der

IR in eine IR Darstellung im Speicher. Ausgehend von dieser IR wird für jede Funktion des Quellprogramms ein Datenflußgraph (DFG) generiert.

Das Modul `DFG2Olive` erzeugt für jeden Basisblock des DFG einen OliveTree, für den durch das `GenCover` Modul eine passende Überdeckung in Form eines Kontrollflußgraphen (CFG) gefunden wird. Bei *Olive* handelt es sich um einen Code Generator Generator [FRASER92].
Anschließend wird auf die CFG Darstellung ein Instruction Scheduling (IS), eine Register Allokation (RA) und eine Peephole Optimierung durchgeführt, aus der eine optimierte CFG Darstellung hervorgeht, die letztendlich durch das `CS_Output` Modul in Assembler Code überführt wird.

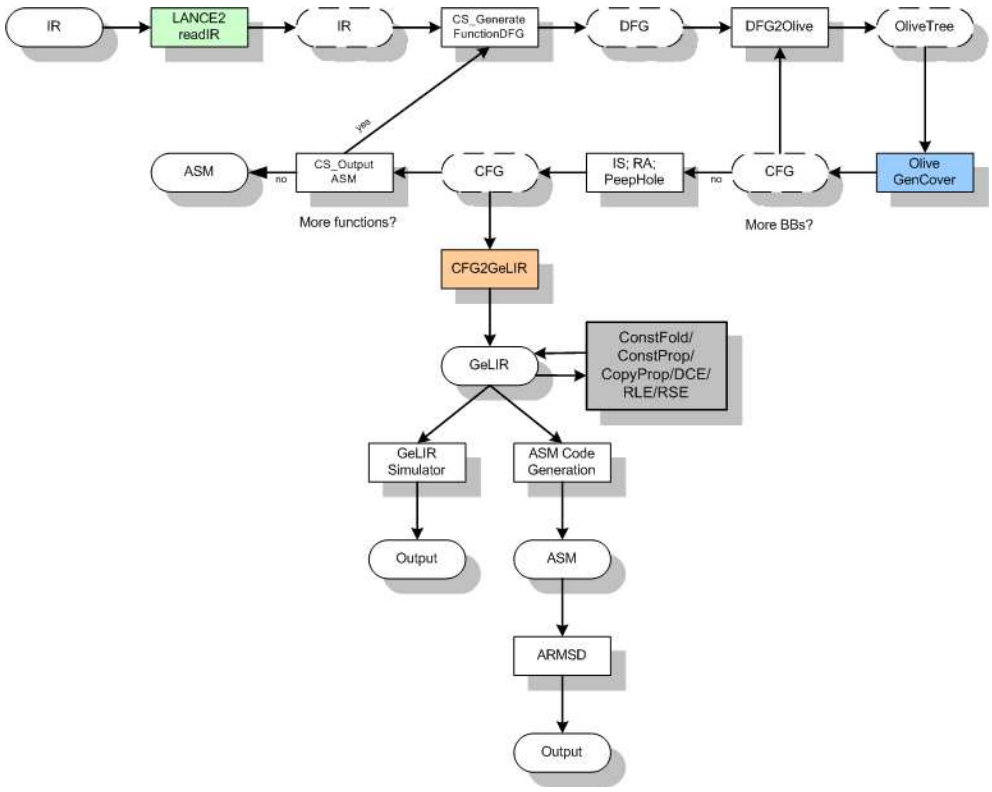

Abbildung 3.3: ir2asm Details aus encc Ablaufdiagramm inkl. GeLIR Unterstützung

Abbildung 3.3 zeigt das eben beschriebene `ir2asm` Modul inkl. der neu hinzugekommenen *GeLIR* Unterstützung. Zwischen den bestehenden *encc* Optimierungen und der Erzeugung der Assembler Darstellung wird die CFG Datenstruktur des *encc* abgegriffen und in die *GeLIR* Darstellung konvertiert. Auf dieser Datenstruktur können dann die ebenfalls im Rahmen dieser Diplomarbeit implementierten, generischen Low-Level Optimierungen operieren. Beim derzeitigen Entwicklungsstand kann die abstrakte *GeLIR* Darstellung auch durch den *GeLIR Simulator* simuliert werden.
Desweiteren besteht die Möglichkeit, Assemblercode in Textform aus der *GeLIR*

Darstellung herauszuschreiben. Das ist legitim, da es sich bei den derzeitigen Optimierungen nicht um "zerstörende" Optimierungen handelt, die eine erneute Code Selektion oder Registerallokation notwendig machen würden.
Zukünftig sind weitere Algorithmen denkbar, um die *GeLIR* Datenstruktur in eine Assemblerdarstellung zu überführen.

Nähere Informationen über `CFG2GeLIR`finden sich in Abschnitt 3.3.6.

3.1.3 aiSee

aiSee ist ein Graphvisualisierungstool, das für die graphische Darstellung von Datenstrukturen im Übersetzerbau entwickelt wurde. Die Eingabe für *aiSee* besteht aus einer Textdatei mit der Beschreibung des Graphen in GDL (Graph Description Language), aus der das Programm automatisch eine graphische Darstellung berechnet, die interaktiv erkundet werden kann [ABS00].

aiSee basiert auf dem 1991 an der Universität des Saarlandes entwickelten VCG (Visualization of Compiler Graphs) und wird seit 1998 von AbsInt Angewandte Informatik GmbH aus Saarbrücken weiterentwickelt und vertrieben [ABS01].

3.2 Übersicht der Erweiterungen

Die im Rahmen dieser Diplomarbeit hinzugekommenen Erweiterungen sind in Abbildung 3.4 mit dicken Linien gekennzeichnet. Zuerst findet eine Konvertierung des *encc* CFG in die *GeLIR* Darstellung statt. Die dafür benötigte Funktionalität wurde in der Klasse `CFG2GeLIR` zusammengefasst. Die genaue Vorgehensweise der Umwandlung wird in Abschnitt 3.6 erläutert. Implementationsbedingte Details finden sich in Anhang B.

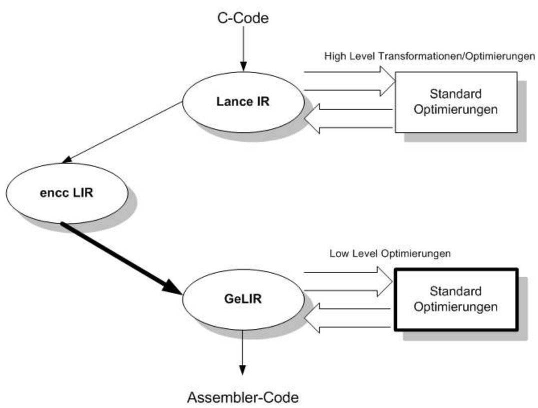

Abbildung 3.4: Einordnung der Erweiterungen

Um die Umsetzung des *encc* CFG in die *GeLIR* Darstellung graphisch zu verdeutlichen, sei vor der genauen Beschreibung der *GeLIR* Datenstruktur, folgendes kleines Beispielprogramm in der Programmiersprache C gegeben:

```
int a[10];

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

  for (i = 0; i < 10; i++)
  {
    a[i] = i * 2;
  }

  return 1;
}
}
```

Beispiel 3.2.1 *Beispielprogramm*

Den Kontrollflußgraphen des *encc* Compilers für obiges Beispielprogramm zeigt Abbildung 3.5, die mit dem Tool *aiSee* erstellt wurde.

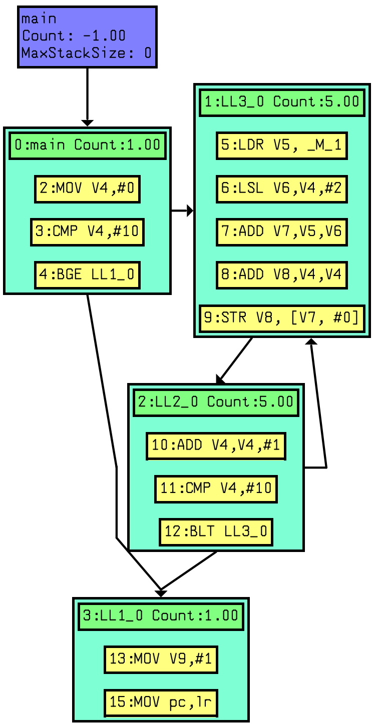

Abbildung 3.5: encc CFG Beispiel

Nach einem erfolgreichen Durchlauf der Konvertierungsroutinen der CFG2GeLIR Klasse erhält man die in Abbildung 3.6 gezeigte *GeLIR* Darstellung des Beispielcodes.

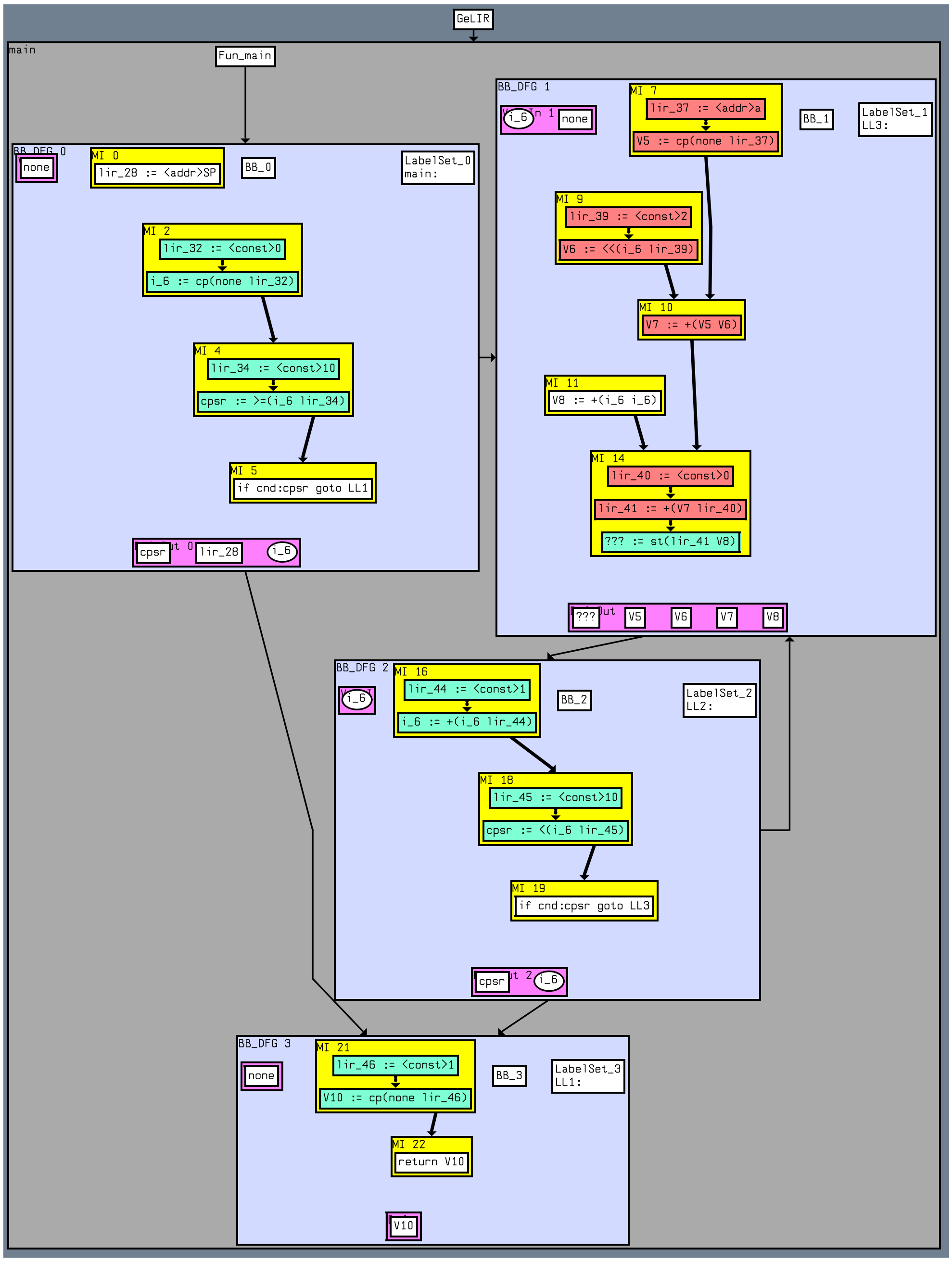

Abbildung 3.6: encc CFG Beispiel in GeLIR Darstellung

In der *GeLIR* Darstellung sind bereits mehr Details zu erkennen als in der *encc* CFG Darstellung. Der Programmcode wird durch in Maschineninstruktionen (MIs) eingebettete Maschinenoperationen (MOs) repräsentiert, die wiederum in Basisblöcke zusammengefaßt sind.

Definition 3.2.1 *Eine* Maschinenoperation *ist eine elementare Operation auf einem Prozessor, wobei die Operanden aus sequenziellen Komponenten gelesen werden und das Resultat in eine sequenzielle Komponente geschrieben wird. Eine Maschinenoperation ist an weitere Ressourcen gebunden, wie z.B. an Funktionseinheiten, auf denen die Operation ausgeführt wird [BA00].*

Definition 3.2.2 *Eine* Maschineninstruktion *ist eine Menge von parallel ausführbaren Maschinenoperationen auf einem Prozessor. Eine maximale Maschineninstruktion ist eine Menge von Maschinenoperationen, die sich mit keiner weiteren Maschinenoperation parallelisieren lässt. Der Instruktionssatz eines Prozessors lässt sich durch eine Menge von maximalen Maschineninstruktionen spezifizieren. Ein Instruktionstyp ist eine Menge maximaler Maschineninstruktionen mit gleichartiger Struktur [BA00].*

Zu jedem Basisblock werden automatisch die *VarsIn*- und *DefsOut*-Listen generiert. Diese Listen sind in Abbildung 3.6 jeweils am Anfang (VarsIn) und am Ende (DefsOut) eines Basisblocks erkennbar.

Definition 3.2.3 *Ein* Basisblock *(Instruktionsblock) ist ein Pfad von Maschineninstruktionen maximaler Länge, wobei höchstens der erste Knoten mehr als einen Eingang besitzt und nur der letzte Knoten mehr als einen Ausgang haben kann.*

Definition 3.2.4 *Eine* VarsIn-*Liste beinhaltet die Variablen, die im Basisblock an einer Stelle* p *gelesen werden und bis dahin nicht redefiniert worden sind. [FLW01]*

Definition 3.2.5 *Alle Variablen, deren Definitionen am Ende eines Basisblocks gültig sind, werden in der* DefsOut-*Liste gespeichert. [FLW01]*

Die eigentlichen Maschineninstruktionen innerhalb des Basisblocks werden in der *aiSee* Darstellung normalerweise weiß hinterlegt. Ist eine MI Bestandteil einer Adressberechnung von Load- oder Store-Befehlen, so wird dies durch ein spezielles Flag symbolisiert und die *aiSee* Darstellung generiert die MI mit einer roten Hintergrundfarbe (im Ausdruck typischerweise dunkelgrau schattiert). Angaben zu Größe, Typ und Gültigkeitsbereich der Symboltabellenelemente werden zusätzlich in grün angezeigt, sind aber in Abbildung 3.6 aus Gründen der Übersicht nicht enthalten. Mögliche MO Überdeckungen, d.h. Zuordnungen von MOs zu Prozessor-Ressourcen, werden ebenfalls mitverwaltet und können graphisch dargestellt werden.

3.3 Zwischencodedarstellung

Die Zwischencodedarstellung oder auch Intermediate Representation (IR) eines Quellprogramms wird vom Compiler Frontend (z.B. *LANCE2*) erzeugt. Ein Compiler Backend (z.B. *encc*) generiert dann eine tieferliegende IR oder den Zielcode aus dieser IR. Das Frontend ist in der Regel architekturunabhängig und die Details der Zielsprache sind weitgehend auf das Backend begrenzt. [ASU88]

Es ergeben sich folgende Vorteile bei der Verwendung einer Zwischencodedarstellung:

- **Unterstützung verschiedener Architekturen**

  Wenn mehrere Architekturen unterstützt werden sollen, braucht nur das Compiler Backend ausgetauscht bzw. zusätzlich an die neue Zielplattform angepasst werden.
  Dadurch läßt sich das $m \times n$-Compiler-Problem, wobei m der Anzahl der Sprachen und n der Anzahl der Architekuren entspricht, auf ein $m+n$-Compiler-Problem reduzieren.

- **Architekturunabhängige Optimierungen**

  Maschinenunabhänge Code Optimierungen können auf der Zwischencodedarstellung angewendet werden.

Gebräuchlich sind verschiedeneste Arten von Zwischensprachen, wie etwa abstrakte Syntaxbäume, Postfix-Notationen oder die 3-Adress-Code Darstellung. Wir verwenden hier allerdings nur die letztgenannte Darstellung in der Form: $x := y$ op z. Bei dieser, mit Assembler Code verwandten Darstellung, steht op für einen Operator und x, y und z sind Label, Konstanten oder vom Compiler erzeugte temporäre Werte. Ein 3-Adress-Code besteht also immer aus einem Ergebnis und zwei Operanden.

Vorteile des 3-Adress-Codes für die Entwicklung von Optimierungen sind die Auflösung von komplizierten arithmetischen Ausdrücken und geschachtelten Kontrollflußanweisungen. Der Beispielausdruck $w = x + y * z$ würde folgendermaßen aufgelöst:

$t_1 = y * z$

$w = x + t_1$

In der Literatur [MUC97] werden die verschienden Arten der Zwischencodedarstellung klassifiziert nach High-Level IR (HIR), Medium-Level IR (MIR) und Low-Level IR (LIR). Für die meisten Optimierungen eignen sich LIR und MIR. LIR insbesondere, wenn die Optimierungen auf Register- und Adress-Informationen angewiesen sind. HIR wird für Abhängigkeitsanalysen und einige darauf basierende Codetransformationen genutzt.

- **High-Level IR**

  Die High-Level IR Darstellung wird hauptsächlich in Preprozessoren vor der eigentlichen Compilierung angewandt, wobei sie anschließend wieder zurück in eine Hochsprache transformiert wird. Alternativ wird sie durch das Compiler Frontend erzeugt, in frühen Phasen des Compilierungsprozesses eingesetzt und meist kurze Zeit später in eine Medium- bzw- Low-Level IR übersetzt.

- **Medium-Level IR**

  Mit Hilfe einer Medium-Level IR können die meisten Eigenschaften der Quellsprache in einer sprachunabhängigen Art und Weise dargestellt werden. Sie dient als Basis für die Generierung von effizientem Maschinencode für eine oder mehrere Zielplattformen.

- **Low-Level IR**

  Operationen einer Low-Level IR können oftmals nahezu direkt Operationen der Zielarchitektur zugeordnet werden. Somit ist eine LIR in der Regel architekturabhängig. Abweichungen von dieser Regel bestehen in den Fällen, wo es mehrere Alternativen zur Erzeugung des effektivsten Codes gibt. Die richtige Alternative auszuwählen, ist dann Aufgabe der letzten Code Selektion im Compilierungsprozess.
  Die Low-Level IR bietet das größte Potential für mögliche Optimierungen.

Die im Rahmen dieser Diplomarbeit verwendete IR Darstellung *GeLIR* gehört zur Klasse der Low-Level IRs.

3.4 Generic Low-Level Intermediate Representation

Dieses Unterkapitel erläutert die wichtigsten Merkmale der objektorientierten *GeLIR* Klassenbibliothek nach [FLW01]. Die *GeLIR* (Generic Low-Level Intermediate Representation) nimmt einen großen Stellenwert dieser Diplomarbeit ein, da in ihr die eigentlichen Optimierungen entwickelt worden sind. Sie unterstützt durch ihren Aufbau die Portabilität der einzelnen Optimierungen auf verschiedene Architekturdarstellungen und dient der Programmdarstellung im Low-Level IR Format.

Die *GeLIR* kann die Programmdarstellung von der Architektur der Zielplattform abstrahieren und ermöglicht die Implementierung von architekturunabhängigen Optimierungen. Zusätzlich besteht die Möglichkeit, die Architektur einer Zielplattform zu beschreiben und über eine Schnittstelle anzusprechen, so daß auch architekturspezifische Merkmale, z.B. spezielle Register Files oder Funktionseinheiten, ausgenutzt werden können.

Insgesamt sind mit der *GeLIR* architekturunabhänge Programmdarstellungen genauso möglich wie architekturabhängige Programmdarstellungen in Verbindung mit der Darstellung von Informationen über den Zielprozessor.

3.4.1 Übersicht der GeLIR Klassenbibliothek

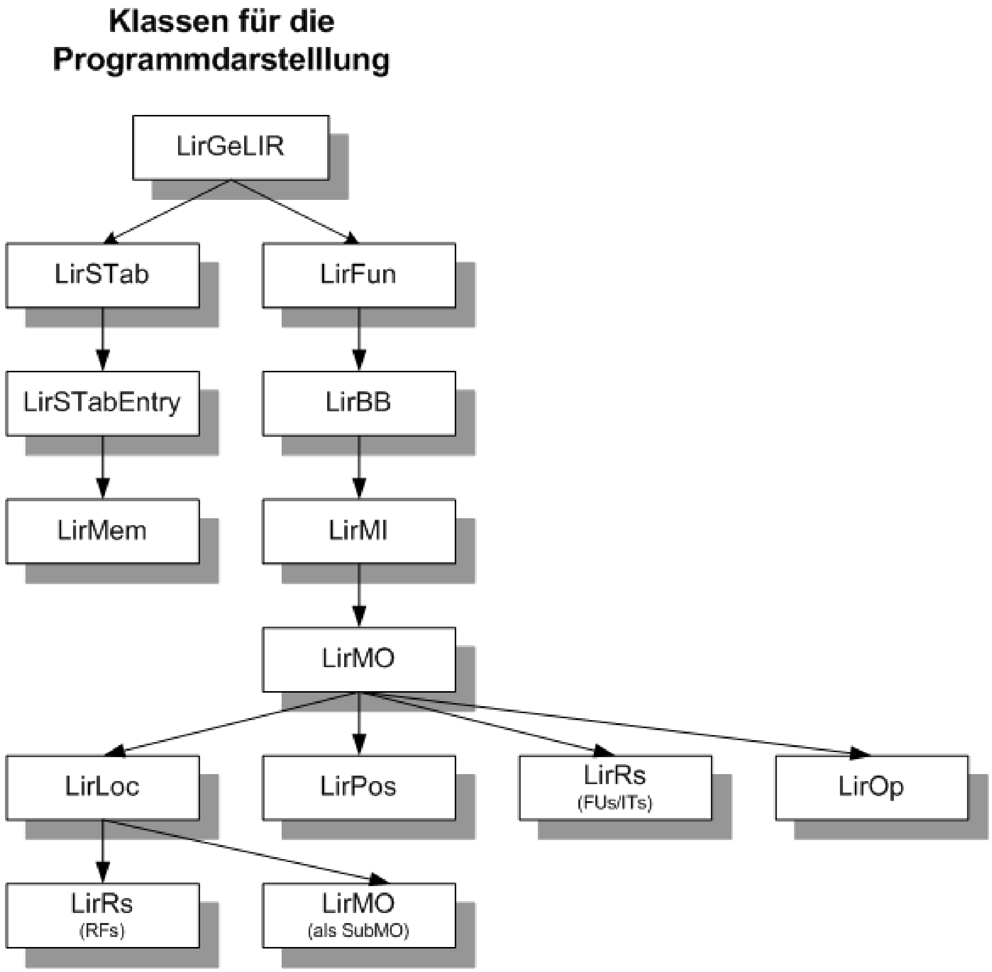

Abbildung 3.7: GeLIR Programmdarstellung - Klassen Übersicht

Abbildung 3.7 bietet eine Übersicht über die wichtigsten *GeLIR* Klassen. Die Pfeile zwischen den Klassen repräsentieren die Objektvererbung. Es gibt zwei große Hauptobjekte. im Objekt `LirGeLIR` wird die eigentliche Programmdarstellung verwaltet. Die Architekturbeschreibung der Zielplattform wird zusätzlich im Objekt `LirTarget` abgelegt, das in Abschnitt 3.3.4.3 beschrieben wird.

Durch diese Unterteilung sind eine abstrakte und eine architekturabhängige Programmdarstellung möglich. Die architekturunabhängige Darstellung besteht aus abstrakten Maschinenoperationen (MOs) und wird im `LirGeLIR` Objekt verwaltet.

Detailierte Informationen über Teile einer Maschinenoperation werden in den `LirLoc`, `LirOp` und `LirRs` Objekten gespeichert. Die Objekte `LirLoc` und `LirPos` werden für Kontextinformationen komplexer Maschinenoperationen verwendet. Ein Symboltabelleneintrag vom Objekttyp `LirSTabEntry` wird für jede Funktion, Variable, Konstante und jedes Label generiert. Diese Einträge werden dann entweder lokal zur Funktion oder global in eine entsprechende Symboltabelle `LirSTab` abgelegt. Über ein `LirMem`-Objekt kann jedes Symboltabellenelement bestimmte Speicherpositionen referenzieren.

3.4.2 Programmdarstellung in der GeLIR

Bei einer Programmdarstellung mittels den *GeLIR* Datenstrukturen handelt es sich um die abstrakte Darstellung des Programms. Der Anwender kann auf dieser Darstellung arbeiten, d.h. z.B. Optimierungen und Transformationen durchführen. Eine Simulation des dargestellten Programms auf einer abstrakten Maschine ist ebenfalls möglich.

Die Programmdarstellung wird vom gesamten Programm (`LirGeLIR`-Objekt) bis hinunter zu einer einzelnen Maschinenoperation (MO) abstrahiert. Siehe Abbildung 3.8.

- **LirGeLIR**

  Auf oberster Ebene gibt es genau ein `LirGeLIR`-Objekt, in dem das gesamte Programm abgelegt wird. Dort werden alle Funktionen als Liste von `LirFun`-Objekten sowie die Globale Symboltabelle im Objekt `LirSTab` verwaltet.

- **LirFun**

  Jede Funktion des Quellprogramms wird durch genau ein `LirFun`-Objekt dargestellt. Jedes `LirFun`-Objekt enthält wiederum mindestens einen Basisblock (BB) in einer Liste von `LirBB`-Objekten. Außerdem wird analog zur globalen Symboltabelle hier die lokale Symboltabelle der Funktion in Form eines `LirSTab`-Objektes gespeichert. Der globale Datenfluß zwischen den Funktionen ist in Form eines Kontrollflußgraphen im `LirFun`-Objekt abgelegt.

- **LirBB**

  Jedes `LirBB`-Objekt speichert einen Basisblock. Außerdem werden Informationen zu Daten-, Ausgabe- und Anti-Daten-Abhängigkeiten in einem Datenflußgraph gespeichert. Zusätzlich gibt es *VarsIn*- und *DefsOut*-Listen bestehend aus einzelnen MOs. Die Informationen dieser beiden Listen können für einfache Datenflußanalysen berücksichtigt werden.

- **LirMI**

  Eine vom Compiler erzeugte Maschineninstruktion kann aus mehreren Maschinenoperationen bestehen. Alle MOs einer MI werden parallel ausgeführt und sollten daher keine Datenabhängigkeiten besitzen. Mit Hilfe von Maschineninstruktionen lassen sich etwa Multiple Instructions Single Data (MISD) oder Multiple Instructions Multiple Data (MIMD) Befehle darstellen.

- **LirMO**

  Eine Maschinenoperation repräsentiert eine elementare, abstrakte *GeLIR* Operation, wie z.B. eine arithmetische oder logische Operation und bedingte oder unbedingte Sprünge. Durch Referenzen auf weitere MOs, sog. SubMOs, werden auch komplexe Maschinenoperationen (z.B. MAC Operationen) unterstützt.

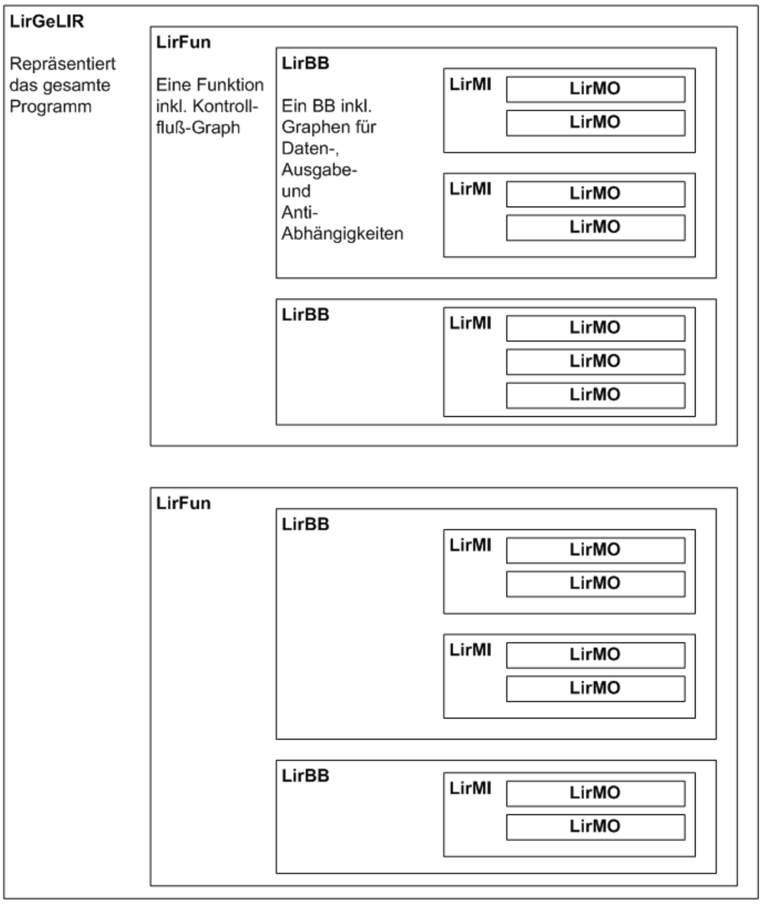

Abbildung 3.8: GeLIR Programm Darstellung

3.4.2.1 Maschinenoperationen

Eine Maschinenoperation repräsentiert eine abstrakte *GeLIR* Operation. Folgende Operationen bzw. Daten sind darstellbar:

- Konstanten
- Variablen
- Adressen
- Unäre / Binäre / n-äre Operationen
- (bedingte) Sprünge
- Returns (mit oder ohne Rückgabewert)

Abhängig von der jeweiligen Operation werden weitere Informationen in einer MO Klasse abgelegt. In der Regel sind dies die Definition, Argumente und der Operationstyp.

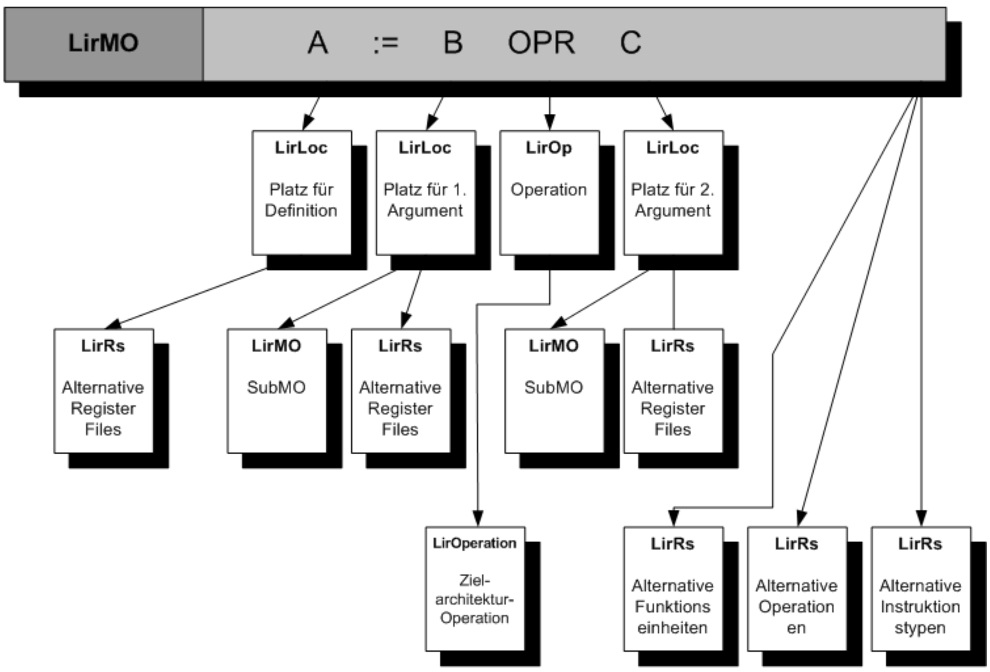

Abbildung 3.9: Darstellung einer binären MO

Abbildung 3.9 zeigt den Aufbau einer MO für eine binäre Operation. Der Typ der Operation wird von der MO in einem `LirOp`-Objekt abgelegt. Dessen abstrakte Operation kann einen Verweis auf eine konkrete Operation der Zielplattform haben, die in einem `LirOperation`-Objekt abgelegt ist. Die Definition und

Argumente werden ebenfalls in der MO verwaltet und in `LirLoc`-Objekten gespeichert. Komplexe Maschinenoperationen enthalten zudem Verweise auf weitere (Sub-)MO-Objekte. Desweiteren bietet die MO über die `LirLoc`-Objekte Zugriff auf die alternativen Register Files für die Definition und die Argumente und verwaltet evtl. Informationen über alternative Funktionseinheiten, Instruktionstypen und Operationen.

Bei unären bzw. n-ären MOs weicht die Darstellung leicht von der obigen ab, da dort nur ein Argument bzw. n Argumente verwaltet werden.

Eine Besonderheit gibt es bei Konstanten und Adressen. Statt der Argumente gibt es ein Containerobjekt `LirConst` bzw. `LirAddr` zur Speicherung der Informationen.

Definition 3.4.1 *Eine Komplexe Maschinenoperation besteht aus mehreren Maschinenoperationen und wird vom Prozessor in einem Taktzyklus abgearbeitet. Ein Beispiel dafür ist etwa die MAC Operation in DSPs.*

Die *GeLIR* Darstellung unterstützt die Abbildung komplexer MOs durch die kaskadenartige Anordnung von zwei oder mehr MOs. Eine SubMO wird durch eine Referenz von der darüberliegenden MO verwaltet, ist aber nach außen hin nicht in der graphischen Darstellung der MI sichtbar.

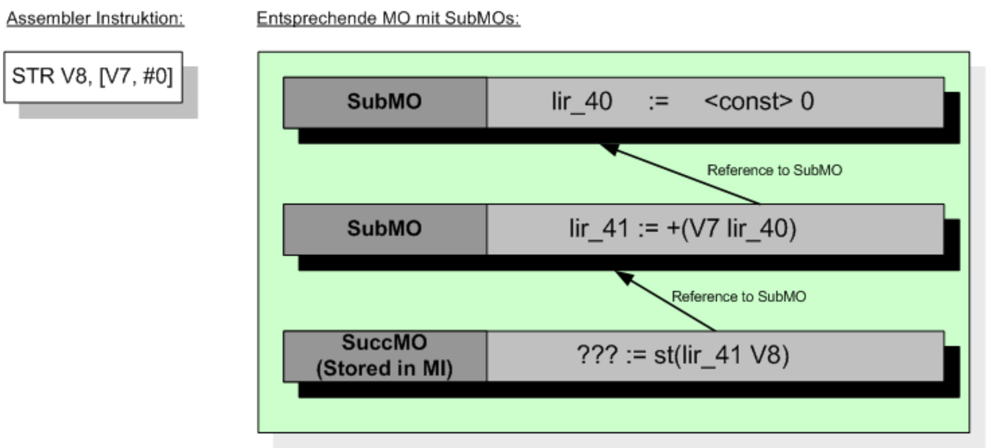

Abbildung 3.10: Darstellung einer SubMO

Abbildung 3.10 zeigt die Darstellung einer SubMO in *GeLIR* am Beispiel einer STR Operation des ARM, die der Wert in Register $V8$ an Adresse $V7 + \#0$ speichert. Die einzelen SubMOs werden dabei durch flüchtige Ressourcen miteinander verbunden.

Definition 3.4.2 Flüchtige Ressourcen *(Transitory resources) sind lesbare bzw. schreibbare Ressourcen, deren Inhalt nur innerhalb eines Instruktionszyklus* *gültig ist. Dies sind z.B. Signalleitungen zwischen Funktionseinheiten oder Register zum Zwischenspeichern eines Resultats einer Funktionseinheit, das noch im gleichen Instruktionszyklus von einer anderen Funktionseinheit gelesen wird [BA00].*

3.4.3 Architekturdarstellung in der GeLIR

Neben der abstrakten *GeLIR* Programmdarstellung gibt es eine optionale maschinennahe Architekturdarstellung. In ihr ist die konkrete Anbindung der Ressourcen der Zielplattform beschrieben. Diese Informationen sorgen für eine einheitliche Darstellung und ermöglichen eine Simulation oder alternativ die Durchführung einer Code Selektion.

Die Spezifikation von Ressourcen der Zielarchitektur kann in der *GeLIR* mit Hilfe des `LirTarget` Objekts erfasst werden. Durch dieses Merkmal eignet sich die *GeLIR* Darstellung zur Entwicklung generischer Optimierungen, die auch Zugriff auf prozessorspezifische Merkmale haben. Das können z.B. spezielle Funktionseinheiten oder der Aufbau der Register Files der CPU sein. Damit ist es für eine Klasse von Prozessoren möglich, prozessorunabhänge Optimierungen zu implementieren, die aber trotzdem auf bestimmte Architekturmerkmale einer Zielplattform zugreifen können.

Mit Hilfe des `LirTarget` Objektes werden architekturabhänge Targetoperationen und Ressourcen der Zielplattform dargestellt. Entsprechende Funktionen erlauben eine Abbildung der Targetoperationen auf die entsprechenden abstrakten *GeLIR* Maschinenoperationen.

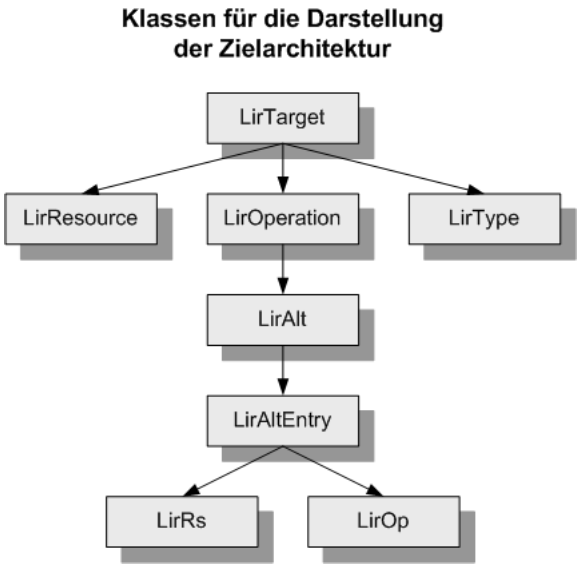

Abbildung 3.11: GeLIR Architekturdarstellung - Klassen Übersicht

Durch folgende *GeLIR*-Objekte wird die Zielarchitektur abgebildet:

- **LirTarget** Das `LirTarget`-Objekt bietet als Hauptobjekt Zugriff auf die darunterliegenden Informationen des `LirOperation`, `LirResource` und `LirType`-Objektes.
- **LirOperation** In den `LirOperation`-Objekten werden Informationen über spezifische Prozessorinstruktionen oder für allgemeine abstrakte LIR-Operationen abgelegt. Dazu zählen etwa die Anzahl der Argumente einer Operation, deren Typ und erlaubte Kombinationen von bestimmten Ressourcen. So kann hier beispielsweise festgelegt werden, auf welche Register eine bestimmte Operation zugreifen darf. Dies ist z.B. beim ARM Prozessor im Thumb Modus von Bedeutung, da dort nicht alle Befehle auf den hohen Registersatz zugreifen können.
- **LirResource** In einem `LirResource`-Objekt können Informationen über Register Files, Instruktionstypen oder Funktionseinheiten abgelegt werden.
- **LirType** Hier werden Informationen über die Typklasse (z.B. VOID, CHAR, INT, ...) und deren Größe in Bits oder Bytes abgelegt.

Für jede Operation können gültige Kombinationen der Targetressourcen angegeben werden. Diese werden in Objekten des Typs `LirAltEntry` abgelegt. Alle `LirAltEntry`-Objekte werden vom `LirAlt`-Objekt verwaltet. (Siehe Abbildung 3.11.)

Im Rahmen dieser Diplomarbeit wurden die Architekturmerkmale des ARM7TDMI in die *GeLIR* Darstellung übertragen. Details dazu finden sich in Anhang A ab Seite 115. Dort werden die notwendigen Schritte zur Abbildung der ARM7TDMI Architektur inkl. aller Register, Funktionseinheiten und Instruktionstypen dargestellt.

3.5 GeLIR Simulator

Zu der *GeLIR* Klassenbibliothek gibt es auch ein *GeLIR Utility* Paket mit einigen Tools. Dazu gehört u.a. auch der *GeLIR Simulator*, der als Debug Umgebung für Programme und Erweiterungen (z.B. Optimierungen) in der *GeLIR* Darstellung ausgelegt ist.

Der Simulator wendet das Verfahren der "Compiled Simulation" an. Dabei wird die interne *GeLIR* Datenstruktur in eine C Datei geschrieben und mit einer speziellen Umgebung versehen, die in der Lage ist, die Datenstruktur in einer Compiled Smulation auszuführen.
Die Debug-Umgebung unterstützt zwei verschiedene Operationsmodi: Abstrakte Simulation und maschinenabhängige Simulation. Je nach Simulationsmodus werden unterschiedliche Log Files generiert. Diese Log-Dateien beinhalten auf jeden Fall den Rückgabewert des simulierten Programms und optional im abstrakten Modus eine beliebige Anzahl weiterer Variablen, die dem Benutzer zur Beobachtung zur Verfügung stehen. Die Log Files werden abschließend von einem Skript verglichen um zu Entscheiden, ob die Simulation erfolgreich verlaufen ist. In dieser Arbeit wurde lediglich die abstrakte Simulation eingesetzt.

Der *GeLIR Simulator* besteht aus zwei Teilen. Einem Skript zur Validierung der Ergebnisse und einer C++ Klasse `CGeLIRSim`, die auf den *GeLIR* Datenstrukturen arbeitet und dafür sorgt, daß die benötigten C Files zur abstrakten oder maschinenabhängigen Simulation generiert werden.

Abbildung 3.12 gibt einen Überblick über die Arbeitsweise des *GeLIR* Simulators. Die Simulation bzw. Validierung läuft zweigleisig ab. Zum einen wird das ursprüngliche ANSI-C Programm mit Hilfe des GNU C Compilers übersetzt und ausgeführt, wobei die für den späteren Vergleich notwendigen Log Files erzeugt werden. Das ist auf der linken Seite der Abbildung 3.12 erkennbar. Zum anderen werden die für den Simulator benötigten Debug Project Files erzeugt und ebenfalls mit dem GNU Compiler übersetzt und dann ausgeführt, wobei wiederum Log Files mit dem Programmergbnis und evtl. weiterer Variablen erzeugt werden. Siehe dazu die rechte Seite von Abbildung 3.12.
Zur Generierung der benötigten Debug Project Files wird das ANSI-C Programm mit dem encc Compiler übersetzt. Intern wird dabei eine Datenstruktur erzeugt, die mit mit Hilfe der `CFG2GeLIR` Klasse in die *GeLIR* Datenstruktur transformiert wird. Mittels der `GeLIRSim` Klasse wird die Erzeugung der Debug Projects Files angestoßen.
Das Simulationsskript vergleicht die generierten Log Files und entscheidet Anhand der Gleichheit des Ergebnisses sowie der überwachten Variablen, ob die Simulation erfolgreich verlaufen ist.

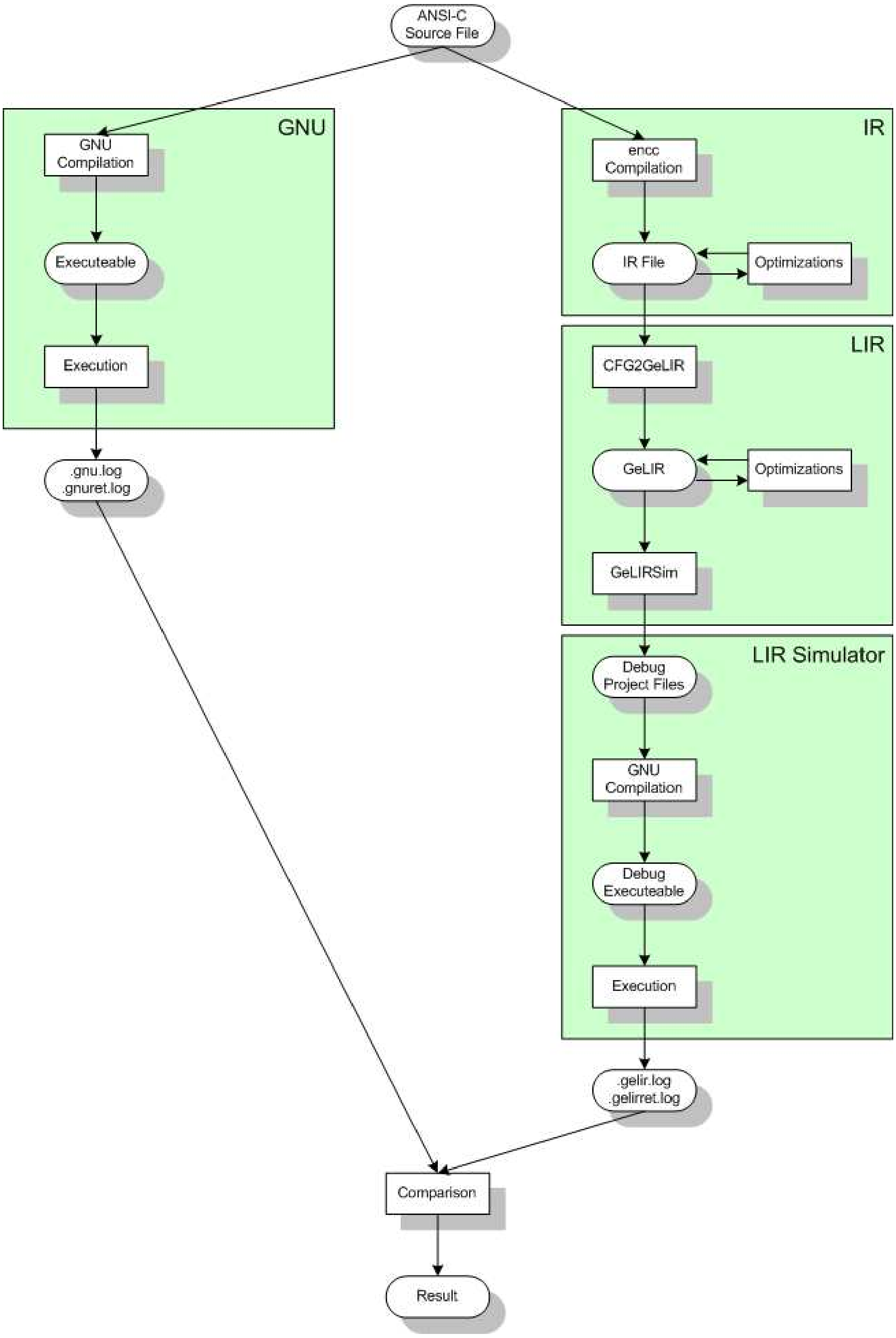

Abbildung 3.12: GeLIR Simulator Ablauf

3.5.1 Assemblercodegenerierung

Wie bereits eingangs in diesem Kapitel erwähnt, werden alle wesentlichen Informationen, die zur Assemblercodegenerierung notwendig sind und nach der Code Selektion und Register Allokation zur Verfügung stehen, mit in die *GeLIR* Datenstruktur übernommen.

Das von Markus Fiesel implementierte Tool *XeLIRPattern* [MF01] erlaubt die patternbasierte Assemblercodegenerierung auf XML-Basis.

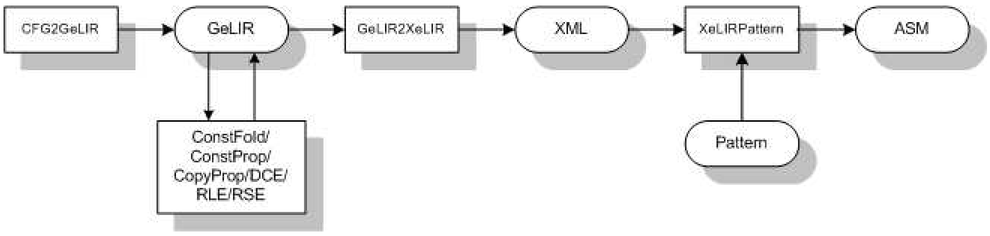

Abbildung 3.13: Ablaufdiagramm der Assemblercodegenerierung

Abbildung 3.13 zeigt den Ablauf der für die Assemblercodegenerierung notwendigen Schritte. Die *GeLIR* Darstellung wird mittels `CFG2GeLIR` erzeugt. Mit Hilfe des Moduls `GeLIR2XeLIR`, das ebenfalls im Rahmen der Diplomarbeit von Markus Fiesel [MF01] entstanden ist, kann die *GeLIR* Darstellung in der Dokumentenbeschreibungssprache XML (Extensible Markup Language) [XML01] herausgeschrieben werden.
Das Tool *XeLIRPattern* importiert diese XML Datei und kann in Verbindung mit einem geeigneten Pattern File den Assemblercode generieren. In der im Rahmen dieser Diplomarbeit entworfenen Pattern Datei ist beschrieben, wie die einzelnen *GeLIR* Maschinenoperationen mit realen Assemblerbefehlen der Zielarchitektur überdeckt werden können.

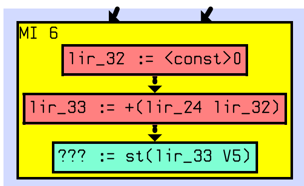

Abbildung 3.14: Beispiel eines Store-Befehls in GeLIR Darstellung

Abbildung 3.14 zeigt beispielhaft eine Store Operation in der *GeLIR* Darstellung. Nach der XML Generierung durch `GeLIR2XeLIR` findet sich die oben gezeigte Maschineninstruktion mit der Store Operation folgendermaßen in der XML Datei wieder:

```
<MI ID="MI-6">
  <!-- CONST lir_32 -->
  <!-- + ( lir_33, lir_24, lir_32 ) -->
  <!-- st ( ???, lir_33, V5 ) -->
  <MO class="ST"  ID="MO-8" class2="" op_name="st" op_sym="35" value_no="8">
      <OperationAlt IDREF="Op-141" op_alt_name="STR"/>
      <FUAlt IDREF="FU-799"/>
      <ITAlt IDREF="IT-513"/>
    <Def volatile="true" name="???" stab_id="1" >
        <DefAlt IDREF="RF-783"/>
    </Def>
    <Arg volatile="true" name="lir_33" stab_id="33" >
        <ArgAlt IDREF="RF-782"/>
      <MO class="OPR"  ID="MO-7" class2="ADDR_OPR" op_name="+" op_sym="2" value_no="7">
          <OperationAlt IDREF="Op-103" op_alt_name="ADD"/>
          <FUAlt IDREF="FU-799"/>
          <ITAlt IDREF="IT-513"/>
        <Def volatile="true" name="lir_33" stab_id="33" >
            <DefAlt IDREF="RF-782"/>
        </Def>
        <Arg volatile="true" name="lir_24" stab_id="24" >
            <ArgAlt IDREF="RF-794"/>
            <ArgIdx index="94" asmname="SP"/>
        </Arg>
        <Arg volatile="true" name="lir_32" stab_id="32" >
            <ArgAlt IDREF="RF-778"/>
          <MO class="CONST"  class2="ADDR_OPR" ID="MO-6" value_no="6">
            <Def volatile="true" name="lir_32" stab_id="32" >
                <DefAlt IDREF="RF-778"/>
            </Def>
              <Const type="INT" value="0" stab_id="31"/>
          </MO>
        </Arg>
      </MO>
    </Arg>
    <Arg volatile="true" name="V5" stab_id="10" >
        <ArgAlt IDREF="RF-792"/>
        <ArgIdx index="81" asmname="r0"/>
    </Arg>
  </MO>
</MI>
```

In der XML Darstellung sind alle Informationen der *GeLIR* Darstellung übernommen worden. Die Store MO inkl. der beiden Sub-MOs wird inkl. ihrer Typen, Argumente, Funktions- und Instruktionseinheiten beschrieben.

Für jede zu überdeckende Operation muß ein entsprechendes Pattern erzeugt werden, damit die benötigten Daten für die Assemblerausgabe aus der XML Darstellung generiert werden können. Mit Hilfe des Patterns werden nur die Daten herausgeschrieben, die auch wirklich benötigt werden. Es werden alle Pattern benötigt, die bei der Assemblerausgabe für die gewünschte Zielplattform vorkommen können. Das hier gezeigte Pattern dient der ARM Assemblercode Darstellung.

```
<Pattern>
  <MO class="ST" op_name="st">
    <OperationAlt/>
    <FUAlt/>
    <ITAlt/>
    <Def>
      <DefAlt/>
    </Def>
    <Arg>
      <ArgAlt/>
      <MO class="OPR" op_name="+">
        <OperationAlt/>
        <FUAlt/>
        <ITAlt/>
        <Def>
          <DefAlt/>
        </Def>
        <Arg>
          <ArgAlt/>
          <ArgIdx/>
        </Arg>
        <Arg>
```

```
          <ArgAlt/>
          <MO class="CONST">
            <Def>
              <DefAlt/>
            </Def>
            <Const/>
          </MO>
        </Arg>
      </MO>
    </Arg>
    <Arg>
      <ArgAlt/>
      <ArgIdx/>
    </Arg>
  </MO>
  <Instruction>
    <Print direct="\t"/>
    <Print direct="STR "/>
    <Print ref="Arg[2]/ArgIdx/@asmname"/>
    <Print direct=", ["/>
    <Print ref="Arg[1]/MO/Arg[1]/ArgIdx/@asmname"/>
    <Print direct=", #"/>
    <Print ref="Arg[1]/MO/Arg[2]/MO/Const/@value"/>
    <Print direct="]"/>
    <Print direct="\n"/>
  </Instruction>
</Pattern>
```

Nach Aufruf des Tools *XeLIRPattern* mit der XML-Datei und der Pattern-Datei als Parameter wird die Assemblerdatei ausgegeben. Für das o.g kurze Beispiel erhält man die Ausgabe:

```
STR r0, [SP, #0]
```

Templates für die Patterns werden mit dem Kommandozeilenparameter `-p` erzeugt. Damit werden alle in der Programmbeschreibung gefundenen Patterns ausgegeben. Wird eine schon bestehende Pattern-Datei angegeben, z.B. `XeLIRPattern -p ADR.c.xml ARM_Patterns.xml`, so werden zuerst die in `ARM_Patterns.xml` gefundenen Patterns ausgegeben, danach die nicht gefundenen.

3.6 Anbindung des encc an die GeLIR

Der ursprüngliche Compiler *encc* verwendet eigene Datenstrukturen, die als Vorarbeit in die *GeLIR* Darstellung transformiert werden müssen.
Hierbei ist insbesondere darauf zu achten, daß bereits vorhandene Informationen bzgl. der Codeselektion und Registeralokation mit in der *GeLIR* abgelegt werden. Diese Informationen werden auf der Low-Level Ebene berücksichtigt um aus der *GeLIR* Darstellung eine Assemblercodegenerierung zu ermöglichen.
Auch für die Simulation ist beispielsweise die Wiederherstellung von erweiterten Typinformationen relevant.

3.6.1 Übersicht

In der Klasse `CFG2GeLIR` findet sich die Implementierung zur Transformation der *encc* CFG Programmdarstellung in die *GeLIR* Datenstrukturen. Im Nachhinein stellte sich diese Konvertierung, die als Vorarbeit für die Optimierungen benötigt wird, als sehr arbeits- und zeitintensiv dar. Dieses hatte mehrere Gründe, u.a. erst im Laufe der Arbeit entstandene Dokumentation, teils gravierende Unterschiede in den verschiedenen Darstellungen und der sich ändernden Arbeitsumgebung, da die *GeLIR* Datenstrukturen zum Zeitpunkt der Implementierung noch nicht vollständig entwickelt waren.

Zugunsten der Entwicklung und Anbindung der eigentlichen Optimierungen unterstützt die momentan vorliegende `CFG2GeLIR` Klasse nur die ARM Plattform, da die ARM Unterstützung des *encc* zum Zeitpunkt der Erstellung dieser Diplomarbeit am weitesten fortgeschritten war. Die Unterstützung der LEON CPU wurde theoretisch durchgeführt.

Die Anpassung der `CFG2GeLIR` Funktionalität an weitere Architekturen, etwa der LEON CPU, stellt aber bis auf den benötigten Zeitaufwand keine großen Probleme mehr dar. Eine Konvertierung für die LEON Architektur kann analog zur in diesem Kapitel beschriebenen ARM Konvertierung stattfinden. Weitere Informationen für evtl. zukünftige Projekte gibt Kapitel 8. Anhang B beschreibt den Einsatz der `CFG2GeLIR` Klasse und die zugrundeliegenden Quellcodes.

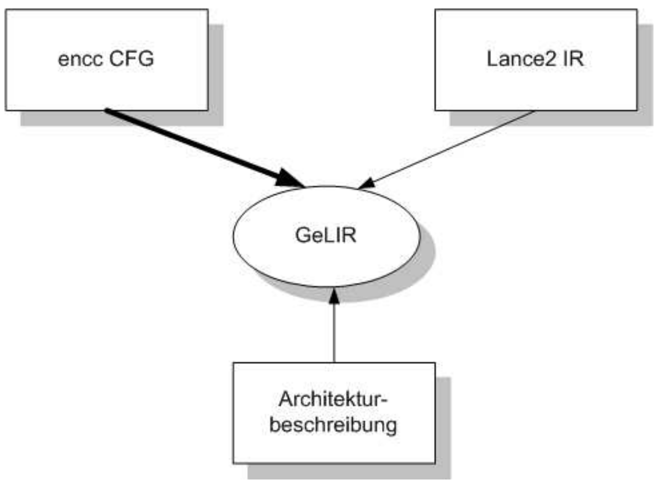

Abbildung 3.15: Schema der GeLIR Anbindung (grob)

Abbildung 3.15 skizziert die Elemente, die zur Konvertierung der *encc* Darstellung in die *GeLIR* Darstellung benötigt werden. Ein Großteil der Informationen konnte direkt aus dem *encc* CFG gewonnen werden. Einige Details sind aber bereits bei der Code Selektion verlorengegangen und waren nicht mehr direkt zu erreichen, z.B. die Typen der Rückgabewerte einer Funktion. Das liegt daran, daß in der durch den *encc* CFG ausgedrückten Assembler Darstellung keine

Typinformationen mehr ersichtlich sind. Informationen über die Funktionsrückgabetypen und die Anzahl der Funktionsargumente werden daher direkt aus der *LANCE2* IR Darstellung übernommen. Von der Assembler Darstellung des Programms wird eine Ebene zurück auf die Low-Level Darstellung transformiert. Schließlich wurde der *GeLIR* Darstellung noch eine Architektur Beschreibung der zugrundeliegenden Zielplattform übergeben.

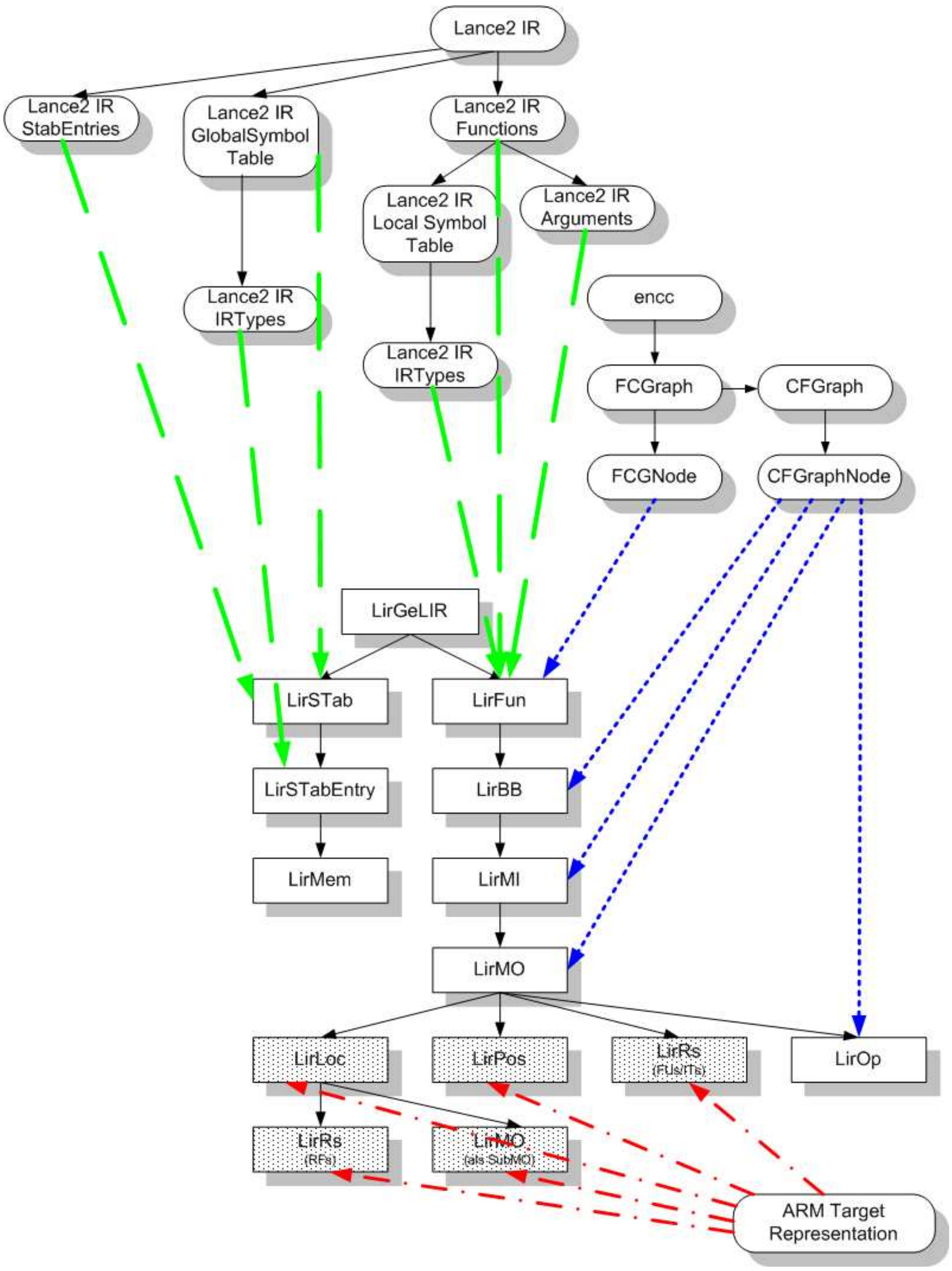

Abbildung 3.16: Schema der GeLIR Anbindung

In Abbildung 3.16 ist die *GeLIR* Anbindung auf *GeLIR* Klassenebene erkennbar. Aus Gründen der Übersicht konzentriert sich die Skizze dabei auf die wichtigsten Elemente. Die Informationen, die aus der *LANCE2* Darstellung gewonnen werden, werden durch gestrichelte Pfeile dargestellt, die in die entsprechenden *GeLIR* Klassen zeigen. Gepunktete Pfeile symbolisieren die Informationen, die aus dem Kontrollflußgraphen des *encc* Compilers gewonnen wurden. Informationen auf unterster *GeLIR* Ebene für die `LirMOs` wurden aus der ARM Target Beschreibung gewonnen. Dieses wird durch Punkt-Strich-Pfeile dargestellt.

3.6.2 Durchführung

Abbildung 3.17 zeigt exemplarisch einen Knoten aus einem CFG Graphen vor der Konvertierung in die *GeLIR* Darstellung (linke Seite der Abbildung) und nach der Konvertierung (rechte Seite).

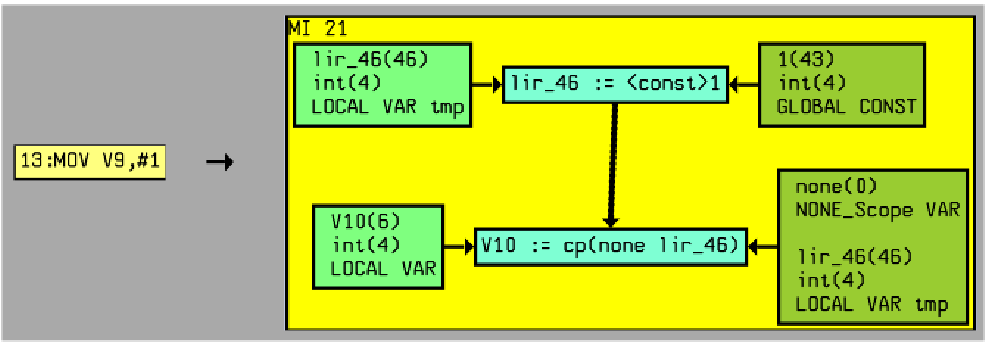

Abbildung 3.17: CFG → GeLIR

Ein kompletter `CFG2GeLIR` Durchgang läuft in folgenden Schritten ab:

1. **Initialisierung der ARM Target Beschreibung**

   Hierbei wird das `LirTarget`-Objekt mit der Beschreibung der ARM Architektur initalisiert. Darin enthalten sind der Aufbau aller Funktioneinheiten, Registersätze und der gesamte Instruktionssatz der CPU. Die Architekturbeschreibung wurde in eine separate Klasse ausgelagert, damit sich später problemlos neue Architekturen hinzufügen lassen. Weitere Details zur Architekturbeschreibung und deren Implementierung gibt Anhang A.

2. **Aufbau der globalen Symboltabelle**

   Die globale Symboltabelle wird aufgebaut, indem die globale *encc* Symboltabelle durchlaufen und in die *GeLIR* Darstellung übertragen wird.

3. **Erzeugung der Liste aller Funktionen**

Definition 3.6.1 *Ein* Function Call Graph *FCG ist die Vereinigung aller Kontrollflußgraphen CFG eines Programms.*

Der Function Call Graph `FCGraph` (FCG) des *encc* wird durchlaufen und anhand seiner Knoten werden entsprechende *GeLIR* `LirFun`-Objekte angelegt. Die Funktionsnamen werden beibehalten. Die Typen der Funktionsrückgabewerte und die Anzahl der Argumente sowie die Argumenttypeninformationen werden aus der *LANCE2* IR ermittelt, da diese Informationen nach der Codeselektion im *encc* nicht mehr ohne großen Aufwand ersichtlich sind.

4. **Aufbau der lokalen Symboltabellen für jede Funktionen**

   Sofern vorhanden werden die lokalen Symboltabellen für die einzelnen Funktionen erzeugt. Für jedes Symboltabellenelement wird dabei ein `LirSTabEntry`-Objekt entsprechenden Typs angelegt (Const, Addr, Label, Int, usw.). Die benötigten Informationen werden aus den *LANCE2* Symboltabellen übernommen.

5. **Aufbau der Liste aller Basisblöcke**

   In diesem Schritt werden zu jeder Funktion ihre zugehörigen Basisblöcke in die *GeLIR* eingefügt. Dazu wird der *encc* CFG durchlaufen und die entsprechenden Basisblockknoten werden herausgesucht.

6. **Generierung der Maschineninstruktionen**

   Für jede Maschineninstruktion eines Basisblocks wird der entsprechende Typ ermittelt, z.B. ob es sich um einen CALL/JMP/BRANCH/RET-Befehl oder eine Operation handelt und dann dazu passende Unterfunktionen zur Generierung der Maschinenoperationen aufgerufen, bevor letztendlich die eigentliche Maschineninstruktion eingefügt wird.

7. **Generierung der Maschinenoperationen**

   Die Generierung der Maschinenoperationen wurde im vorherigen Schritt angestoßen. Zunächst wird die entsprechende Maschinenoperationen als abstrakte, maschinenunabhängige *GeLIR* Operation eingefügt. Eine Addition wird beispielsweise nicht direkt auf das *ADD* Mnemonic der Zielarchitektur abgebildet, sondern als abstrakte *LIR_PLUS* Operation. Anhand des Adresstyps des aktuell betrachteten Statementknoten des *encc* CFG wird eine passende abstrakte Maschinenoperation generiert. Dieser Teil des `CFG2GeLIR` Konverters ist für den ARM Prozessor als Zielplattform bereits implementiert. Er kann für zukünftige Projekte leicht an andere CPUs angepasst werden.

   Abbildung 3.18 gibt einen Überblick über die bisher beschriebenen Stationen der Durchführung der Transformation.

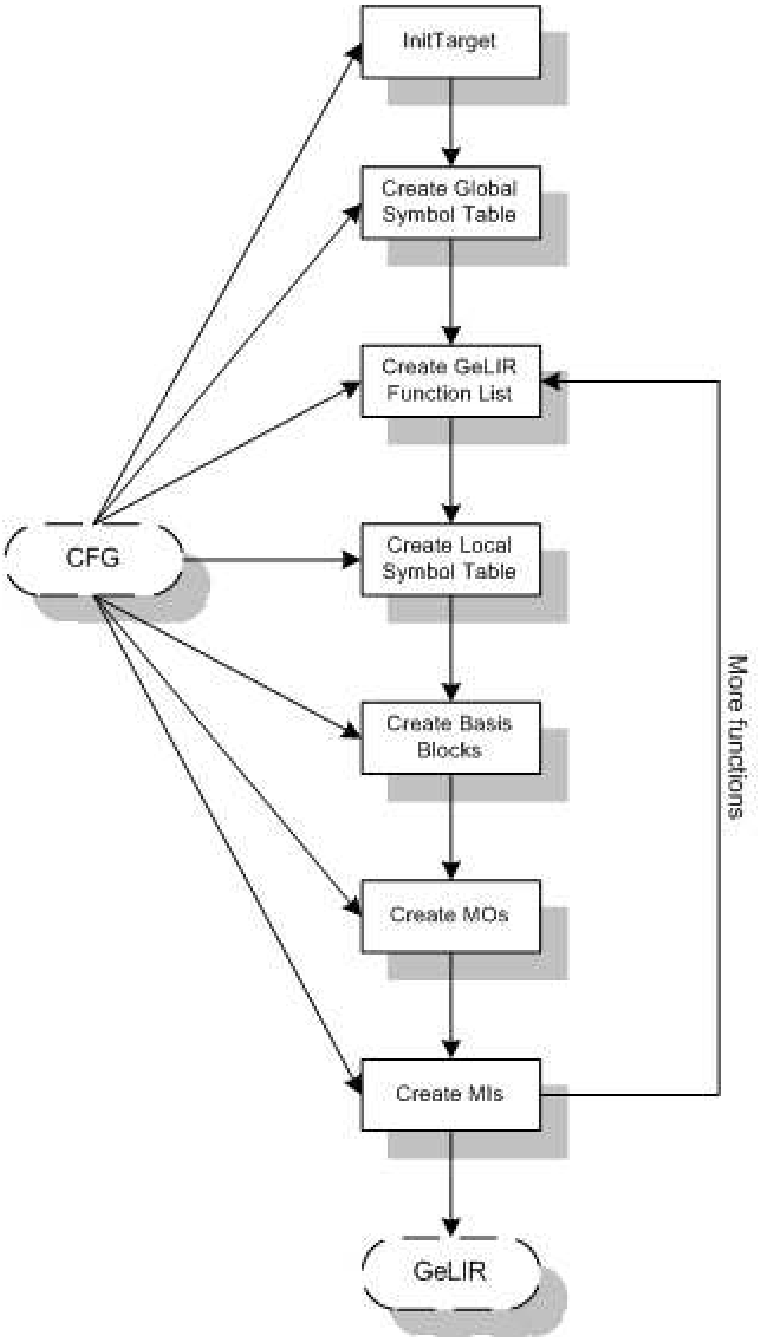

Abbildung 3.18: CFG2GeLIR Details der Durchführung

Tabelle 3.1 gibt eine Übersicht der vom *encc* unterstützten Adressierungsarten, die ebenfalls vom Konverter berücksichtigt werden. Dabei bezeichnet R ein Register, L ein Label, I einen Immediate und S den Stackpointer.

Um optional neben der abstrakten Darstellung auch auf eine reale Darstellung zugreifen zu können, werden für jede MO abschließend Informationen der Zielarchitektur gesetzt, und zwar die reale ARM Operation,

Adressierungsart	Befehlsklasse(n)
NONE	Return
R	Return value
L	Conditional branch
RI	Arithmetic ALU operations
RL	Load
RR	Logical ALU operations / MUL operation
RS	Arithmetic ALU operation
SI	Arithmetic ALU operation
SR	Arithmetic ALU operation
RRI	Arithmetic / Logical ALU operations
RRR	Arithmetic ALU operations
RSI	Arithmetic ALU operation
R_RI	Load / Store
R_RR	Load / Store
R_SI	Load / Store

Tabelle 3.1: In CFG2GeLIR unterstützte Adressierungsarten des ARM

die verwendete Funktionseinheit, der Instruktionstyp und die verwendeten Register.
Abbildung 3.19 zeigt die Maschinenoperationen, in der die entsprechenden Ressourcen bereits eingetragen worden sind, für den Assembler Befehl `ADD r3,r3,#1`.

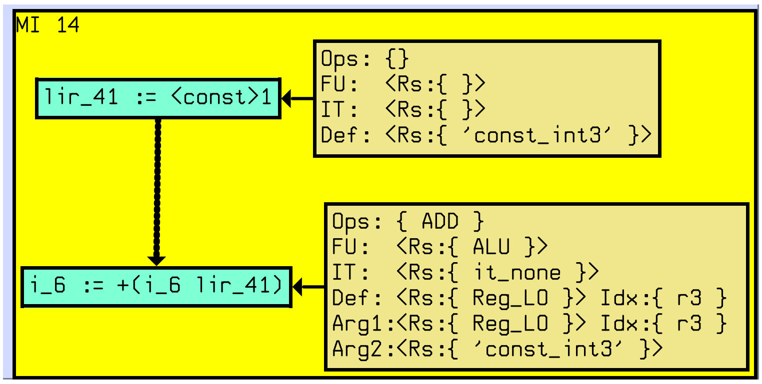

Abbildung 3.19: MO mit eingetragenen Ressourcen

Es wird darauf geachtet, daß die bereits vorliegenden Informationen aus der Codeselektion und der Registerallokation nicht verloren gehen, damit aus der *GeLIR* Darstellung jederzeit wieder eine Assemblerdarstellung generiert werden kann. So ist z.B. in der Abbildung erkennbar, daß die durch die Register Allokation ausgewählten Register `r3` des ADD-Befehls auch in der *GeLIR* Darstellung übernommen worden sind, indem der Index des entsprechenden Definitions- und des ersten Argument-Registers auf `r3` gesetzt werden.
Da die im Rahmen dieser Diplomarbeit implementierten Optimierungen keine neuen virtuellen Register anfordern und somit auch keinen neuen

Spillcode erzeugen, der den Registerdruck erhöhen würde, kann die Assembler Darstellung ohne eine erneute Code Selektion aus den *GeLIR* Datenstrukturen gewonnen werden.
Die dazu notwendigen Programmteile sind in seperate Funktionen ausgelagert und lassen sich leicht für andere Architekturen, z.B. die LEON CPU, erweitern.

8. **Aufräumarbeiten**

   Im Prinzip ist die eigentliche Konvertierung des *encc* Kontrollflußgraphen in die *GeLIR* an dieser Stelle abgeschlossen. Es folgen noch ein paar Aufräumarbeiten, bei denen u.a. alle überflüssigen Symboltabelleneinträge gelöscht werden, die durch das *LANCE2* Frontend eingefügt worden sind.

9. **Optimierungen durchführen**

   Optional werden die in den nachfolgenden Kapiteln beschriebenen Optimierungen in diesem Schritt durchgeführt.

10. **Evaluation durch Simulation**

    Auf Wunsch werden im letzten Schritt Simulationsdateien für den *GeLIR Simulator* generiert, damit die Korrektheit der *GeLIR* Programmdarstellung validiert werden kann.
    Simulationsergebnisse der im Rahmen dieser Diplomarbeit verwendeten Beispielprogrammme finden sich in Kapitel 7.

Hinweis 3.6.1 *Im Debug Modus erzeugt die* `CFG2GeLIR` *Klasse zwei .gdl Dateien* `CFG2GeLIR.ir.gdl` *und* `CFG2GeLIR.opt.ir.gdl`*, die z.B. mit* aiSee *angezeigt werden können. Der Inhalt dieser Dateien ist die grafische Darstellung des Benchmark-Programms im* GeLIR *Format vor und nach evtl. Optimierungen.*

3.6.3 Probleme und Besonderheiten

Bei der Umwandlung des *encc* CFG in die *GeLIR* Darstellung mussten u.A. folgende Besonderheiten beachtet werden:

- **Umsetzung komplexer Adressierungstypen**

  Befehle mit einfachen Adressierungsarten, z.B. mit Adressierungstyp `RR` (Register – Register) können direkt in eine entsprechende *GeLIR* Maschineninstruktion überführt werden. So wird z.B. aus dem Assemblerbefehl `MOV V6,V8`, der den Inhalt des virtuellen Registers `V8` in das virtuelle Register `V6` kopiert, in der *GeLIR* Ansicht ein Ausdruck der Art: `V6 := cp(none lir_39)`.
  Bei komplexeren Adressierungstypen, etwa der Form `R_RI` (Register – Register indirekt – Immediate) ist keine 1:1 Umsetzung von der Assemblerdarstellung in die *GeLIR* Darstellung möglich. In so einem Fall wird

eine Maschineninstruktion mit mehreren SubMOs erzeugt, die die Funktionalität des Assemblerbefehls abbildet. So werden z.B. für den Befehl `STR V8,[V4,#8]`, der den Wert von `V8` an Adresse `V4+8` speichert, folgende drei Maschinenoperationen innerhalb einer Maschineninstruktion erzeugt:

```
                   lir_22 = 8
                     |
         lir_23 = +(V4,lir_22)
          |
??? = st(lir_23, V8)
```

- **Einfärben der Adressberechnung bei LDR/STR Befehlen**

  Alle Maschineninstruktionen, die Teil einer Adressberechnung sind, werden gesondert gekennzeichnet. Dazu wurde eine rekursive Hilfsfunktion implementiert, die eine Einfärbung der entsprechenden Knoten vornimmt. Somit sind die Bestandteile einer Adressberechnung in der *aiSee* Darstellung unmittelbar erkennbar.

- **Verwendete Ressourcen ermitteln**

  Da die *GeLIR* Darstellung auf unterster Ebene nach der Code Selektion des *encc* Compilers erzeugt wird, sind bereits Ressourceninformationen der einzelnen Maschineninstruktionen in Form der verwendeten Register, Funktionseinheiten und Instruktionstypen vorhanden, die mit in die *GeLIR* Datenstruktur übernommen werden.
  Die verwendeten Funktionseinheiten und Instruktionstypen können dabei unmittelbar aus der Befehlsklasse der gerade betrachteten Maschineninstruktion ermittelt werden. Abschließend werden die in der bereits durchgeführten Registerallokation bestimmten Registerressourcen bestimmt und in die entsprechenden *GeLIR* Objekte abgelegt.

- **Besondere Behandlung des Stackpointers**

  Ein weiteres Problem stellte die korrekte Modellierung des Stackpointers (SP) dar, damit er in der abstrakten Darstellung und im *GeLIR Simulator* richtig abgebildet wird.
  Ein ADD Assembler Befehl am Anfang einer Funktion sorgt dafür, den nötigen Platz auf dem Stack bereitzustellen, indem der Stackpointer um die benötigte Anzahl der Elemente verringert wird. Der Befehl `ADD SP,#-8` bewirkt, daß auf dem Stack Platz für zwei 4-Byte Elemente geschaffen wird.
  In der *GeLIR* Darstellung wird der Stackpointer als Symboltabellenelement `SP` vom Typ Array definiert. Das Array wird als lokales Array angelegt, da für jede Funktion ein neuer Stackframe aufgebaut wird. Beim Aufruf einer Funktion mit einem `Add SP, #immediate` wird anstatt eine Add MO einzufügen, die Größe des `SP` Arrays auf den Absolutbetrag der Immediate Konstante gesetzt.

Kapitel 4

Datenflußanalysemethoden

Das Programmverhalten darf durch Optimierungen nicht willkürlich verändert werden, damit die Korrektheit des Programms bestehen bleibt. Aus diesem Grund sind bei vielen Optimierungen Vorarbeiten notwendig, um Informationen zu sammeln, welche Stellen im Code in geeigneter Weise abgeändert werden dürfen. Dazu zählen beispielsweise die Untersuchung von Verzweigungsbedingungen und die Abhängigkeiten bislang berechneter Werte, die durch Abhängigkeitsanalysen ermittelt werden.
Die in dieser Diplomarbeit verwendete *Delta-Array-Datenflußanalyse* zur Bestimmung der Datenabhängigkeiten bei den Optimierungen *Redundant Load Elimination* und *Redundant Store Elimination* wird in diesem Kapitel vorgestellt.
Die folgenden Ausführungen in diesem Kapitel basieren auf den Arbeiten von Martin Horst [MH01] und Björn Franke [BF99].

4.1 Einführung

Bei einer Datenflußanalyse werden Informationen über das Programm als Ganzes gesammelt und den einzelnen Basisblöcken des Kontrollflußgraphen zur Verfügung gestellt, um festzustellen, welche Variablenabhängigkeiten über alle möglichen Kontrollflußwege bestehen.

Mittlerweile gibt es eine große Menge an etablierten Datenflußanalysemethoden für skalare Variablen [ASU88], die die Grundlage vieler Optimierungen bilden, aber leider oftmals Zugriffe auf Arrays nicht berücksichtigen. Mit speziellen Array-Datenflußanalysemethoden gelingt es zusätzlich, Datenabhängigkeiten zwischen Array-Elementen in Schleifen zu analysieren und mit diesen Erkenntnissen geeignete Optimierungen zu entwickeln.

Als Grundlage für die im Rahmen dieser Diplomarbeit entwickelten Optimierungen *Redundant Load Elimination* und *Redundant Store Elimination* bot sich eine Array-Datenflußanalyse an, da sich mit deren Hilfe redundante Array-Zugriffe in Schleifen aufspüren lassen. Speicherzugriffe, insbesondere Array-Zugriffe, sind durch die damit verbundene Berechnung der benötigten Speicheradressen teurer im Zugriff als normale Registerzugriffe. Im Gegensatz zu einem Registerzugriff oder einem Zugriff auf eine skalare Variable erfordert ein Array-Zugriff einen zusätzlichen Speicherzugriff, was sich in Form von zusätzlichen Waitstates negativ auf die Latency und den Energieverbrauch auswirkt. Dieser Effekt wird innerhalb von Schleifen weiter verstärkt, so daß die Entfernung redundanter Speicherzugriffe durch eine Optimierung auf Basis einer Array-Datenflußanalyse die Kosten eines Programmablaufs deutlich verringern kann. Redundante Speicherzugriffe können z.B. auch während der Spillcodegenerierung des *encc* Compilers entstehen.

Aber auch die einfacheren Optimierungen, wie z.B. die *Constant Propagation* beruhen auf (einfachen) Datenflußanalysen, so daß hier vorgestellte Begriffe (etwa Verbände) auch dort zum Einsatz kommen.

In [BF99] werden vier Array-Datenfluß-Analysemethoden vorgestellt:

- **Delta-Array-Datenflußanalyse**

  Die *Delta-Array-Datenflußanalyse* bestimmt den Datenfluß in ähnlicher Weise wie iterative Verfahren zur Lösung des Datenflußproblems. Jeder Instruktion eines Schleifenkontrollflußgraphen wird eine Transferfunktion zugeordnet, die das Instruktionsverhalten statisch modelliert. Ein aus diesen Transferfunktionen entstehendes Gleichungssystem wird mit Hilfe eines iterativen Verfahrens gelöst.

- **Stretched-Loop Verfahren**

  Die Arbeitsweise des *Stretched-Loop Verfahren* ([BG96]) ist es, alle Array-Referenzen, die auf das gleiche Array-Element zeigen, gemeinsam zu betrachten, indem diese in Kongruenzklassen zusammengefasst werden. Dadurch ist es möglich, die Array-Abhängigkeiten auf der Ebene der einzelnen Knoten zu betrachten. Liegen die Referenzen außerhalb der Kongruenzklassen, so werden diese für die Analyse des einzelnen Knoten ignoriert, da sie irrelevant für die Bestimmung der Abhängigkeiten sind. Mehrere aufeinanderfolgende Iterationen, die die gesamte Lebensdauer eines Array-Elements umfassen, können gleichzeitig analysiert werden.
  Im Gegensatz zur *Delta-Array-Datenflußanalyse* werden Instruktionen nicht nur auf Iterationsebene, sondern auch auf Instruktionsebene analysiert. Dadurch steigt die Aproximationsgüte, was aber mit einer erhöhten Laufzeit verbunden ist.
  Insgesamt scheint das *Stretched-Loop Verfahren* etwas mächtiger als die *Delta-Array-Datenflußanalyse*, ist aber wesentlich komplexer zu implementieren.

- **Lazy Verfahren**

Das *Lazy Verfahren* [M93] arbeitet wie das *Stretched-Loop Verfahren* auf Instruktionsebene, um bessere Ergebnisse zu liefern. Desweiteren ist es nicht auf affine Indexterme beschränkt, sondern kann auf nicht-affinen Indextermen zumindest ein approximiertes Ergebnis liefern.
Bei diesem Verfahren werden ausgehend von Read-Instruktionen die im naheliegenden Instruktionsraum befindlichen Write-Instruktionen analysiert und diese Analyse dann im späteren Verlauf auf größere Distanzen ausgedehnt, um eine größere Präzision zu erreichen.
Vorteilhaft ist die genaue Analyse dieses Verfahrens und die Möglichkeit, nicht-affine Indexterme zu analysieren. Auf der anderen Seite besitzt das Verfahren eine hohe Laufzeit, ist nur mit großem Aufwand zu implementieren und nicht parametrisierbar.

- **Dynamic-Single-Assignment (DSA) Verfahren**

  Das *Dynamic-Single-Assignment* Verfahren nach [R91] unterscheidet sich von den o.g. Verfahren, da es eine speziell für die Analyse angepasste IR-Darstellung verwendet. Diese IR ist darauf spezialisiert, die Abhängigkeiten zwischen Instruktionen darzustellen.
  Das Verfahren ist in der Lage, auf affinen und nicht-affinen Indextermen zu arbeiten, es ist parametrisierbar und unterstützt beliebige Kontrollflußgraphen, ist also nicht auf bestimmte Schleifenklassen beschränkt. Desweiteren lassen sich mit diesem Verfahren speicherbasierte und wertebasierte Abhängigkeitsanalysen durchführen.
  Für unsere Zwecke ist das Verfahren nicht geeignet, da es eine eigene IR-Darstellung voraussetzt und nicht ohne weiteres auf der *GeLIR* Darstellung durchführbar ist. Ein weiterer wesentlicher Nachteil ist, daß das Verfahren u.U. nicht terminiert.

Die Wahl fiel auf die *Delta-Array Datenflußanalyse*, da diese im Gegensatz zu den anderen Verfahren folgende Vorteile aufweist:

- Geringe Komplexität des Verfahrens und damit ressourcenschonend
- Effiziente Laufzeit
- Terminiert immer
- Unterstützt eindimensionale und mehrdimensionale Arrays
- Parametrisierbar
- In gewissem Maße anpassbar und erweiterbar
- Eine generische Implementierung mit Unterstützung der *GeLIR* Datenstruktur liegt bereits vor und konnte für diese Diplomarbeit mit wenigen Anpassungen übernommen werden [MH01]

Insbesondere die letzten drei Punkte gaben den wesentlichen Ausschlag für den Einsatz der *Delta-Array-Datenflußanalyse*. Durch ihr hohes Maß der Parametrisierbarkeit konnte die Analyse erfolgreich für die Schleifenoptimierungen der M3-DSP Plattform [MH01] sowie für die *Redundant Load Elimination* und *Redundant Store Elimination* eingesetzt werden.

4.2 Grundlegende Begriffe

Bei einer Datenflußanalyse werden alle möglichen Kontrollflußwege einer Funktion untersucht. Aus diesem Grund werden die während der Analyse gewonnenen Informationen in einem Kontrollflußgraph abgelegt. Dieser enthält neben der korrekten Ausführungsreihenfolge aller Instruktionen auch alle möglichen Verzweigungen des Kontrollflußes.

Definition 4.2.1 *Ein Pfad π in einem Kontrollflußgraphen CFG(N, E, s, e) ist aus einer Folge von Kanten aufgebaut. π startet an einem Knoten n_1 und endet an einem Knoten n_k : $\pi = (n_1, n_2), \ldots, (n_{k-1}, n_k)$ mit $(n_i, n_{i+1}) \in E, i < k$.*

Um den Verwaltungsaufwand des Kontrollflußgraphen in Grenzen zu halten, werden alle Instruktionen, die hintereinander ausgeführt werden, zu Basisblöcken zusammengefasst.

4.2.1 Datenflußverbände

Definition 4.2.2 *Ein Verband L besteht aus einer Menge von Elementen, der Trägermenge, sowie zwei Operatoren $\sqcap$ (meet) und $\sqcup$ (join) mit folgenden Eigenschaften:*

1. *Abgeschlossenheit*
   $\forall x, y \in L : \exists z, w \in L : x \sqcap y = z \wedge x \sqcup y = w$
2. *Kommutativität*
   $\forall x, y \in L : x \sqcap y = y \sqcap x \wedge x \sqcup y = y \sqcup x$
3. *Assoziativität*
   $\forall x, y, z \in L : (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z) \wedge (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
4. *Supremum*
   $\exists \bot \in L : \forall x \in L : x \sqcap \bot = \bot$
   Das Element $\bot$ wird als bottom *bezeichnet.*
5. *Infimum*
   $\exists \top \in L : \forall x \in L : x \sqcup \top = \top$
   Das Element $\top$ wird als top *bezeichnet.*

Definition 4.2.3 *Ein Verband L ist ein* distributiver Verband, *wenn gilt:* $\forall x, y, z \in L : (x \sqcap y) \sqcup z = (x \sqcup z) \sqcap (y \sqcup z)$

Definition 4.2.4 *Die auf einem Verband L indizierte* Partielle Ordnung $(L, \sqsubseteq)$ *ist durch* $\forall x, y \in L : x \sqsubseteq y \iff x \sqcap y = x$ *definiert. Alternativ ist diese Definition auch mit dem* $\sqcup$ *Operator möglich, wobei die Operatoren* $\sqsubset$, $\sqsupset$ *und* $\sqsupseteq$ *analog definiert sind.*

Definition 4.2.5 *Die Höhe eines Verbandes L ist definiert als:* $height(L) = max\{n | \exists x_1, x_2, \ldots x_n : \bot \sqsubset x_1 \sqsubset x_2 \sqsubset \ldots \sqsubset x_n \sqsubset \top\}$

4.2.2 Transferfunktionen

Mit Hilfe der Datenflußverbände werden Eigenschaften von Datenabhängigkeiten modelliert. Da sich diese Eigenschaften während des Programmablaufs verändern können, muß dies in geeigneter Weise berücksichtigt werden.
Dazu werden Transferfunktionen verwendet, die die durch Instruktionen herbeigeführten Veränderungen des Programmstatus (z.B. aktuelle Variablenbelegung) auf Basis der Datenflußverbände modellieren.

Transferfunktionen sind Funktionen $f : L \rightarrow L$, die der Konten–/Kantenmenge des Kontrollflußgraphen CFG(N, E, s, e) zugeordnet werden.
Es gilt: $tf : M \rightarrow (L \rightarrow L)$, mit $M = \{N, E\}$.

Bei Array-Elementen ist es möglich, daß gleiche Bezeichner auf verschiedene Array-Elemente zeigen, z.B. zeigt der Bezeichner $a[i]$ in den verschiedenen Iterationen auf unterschiedliche Speicherbereiche.
Genauso können verschiedene Bezeichner auf gleiche Array-Elemente zeigen. So zeigt etwa $a[i]$ in Iteration 2 und $a[i+1]$ in Iteration 1 auf das gleiche Element. Die Transferfunktionen müssen so ausgelegt sein, daß sie mit diesen Fällen zurecht kommen.

4.2.3 Schleifenkontrollflußgraph

Ein *Schleifenkontrollflußgraph* ist ein erweiterter Kontrollflußgraph mit folgenden Eigenschaften:

Definition 4.2.6 *Ein* Schleifenkontrollflußgraph *(Loop Control Flow Graph, LCFG) FG = (N,E) mit Knotenmenge N und Kantenmenge E stellt den Kontrollfluß innerhalb eines Schleifenkörpers dar. Dabei bezeichen die Knoten N Instruktionen, wobei jeder Knoten N nur jeweils eine Instruktion enthalten darf. Die möglichen Kontrollflußübergänge im Graphen werden durch die Kantenmenge E dargestellt. Schleifenkontrollflußgraphen sind im Gegensatz zu einem Kontrollflußgraphen um einen ausgezeichneten Knoten Exit erweitert, welcher den expliziten Übergang zur nächsten Iteration ausdrückt. Alle Kanten, die den*

Schleifenkörper verlassem, führen zum Exit Knoten und alle ausgehenden Kanten des Exit Knotens führen zum Beginn der Schleife.

Einzelne Knoten des *Schleifenkontrollflußgraphen* enthalten Instruktionen einer Schleife, deren Beeinflussung der Daten durch die o.g. Transferfunktionen beschrieben wird.

4.3 Delta-Array-Datenflußanalyse

Dieser Abschnitt erläutert die wichtigsten Faktoren zum Verständnis der von Duesterwald, Gupta und Soffa vorgestellten *Delta-Array-Datenflußanalyse* [DGS] und beschreibt grob die Arbeitsweise der Datenflußanalyse.
Tiefergehende Details können in [DGS], [BF99] und [BG96] nachgelesen werden. Die Beschreibung auf Basis einer realen Implementierung der *Delta-Array-Datenflußanalyse* kann in [MH01] gefunden werden.

Die *Delta-Array-Datenflußanalyse* unterstützt verschiedene Optimierungen, u.A. Load–/Store-Optimierungen und Schleifenoptimierungen (Loop Unrolling).
Die zu analysierende Funktion muß folgende Voraussetzungen erfüllen:

- **Strukturierte Schleifen**

  Strukturierte Schleifen sind Schleifen, die die *Single-Entry/Single-Exit* Eigenschaft besitzen. Eine Schleife besitzt die *Single-Entry* Eigenschaft, wenn es einen Knoten n gibt, über den alle Pfade von einem Startknoten s zu den Schleifenknoten führen müssen.
  Gibt es einen Knoten n, über den alle Pfade aus der Schleife zu einem Endknoten e führen, so besitzt die Schleife die *Single-Exit* Eigenschaft.

- **Affine Indexfunktionen**

  Affine Indexfunktionen sind lineare Funktionen der Form $a * i + b$, wobei a und b konstant sind.

Trifft eine dieser Eigenschaften nicht zu, kann die Analyse nicht angewendet werden. Nicht-affine Indexfunktionen treten in der Praxis jedoch nur sehr selten auf.

Mit Hilfe der *Delta-Array-Datenflußanalyse* kann zum einen abgefragt werden, ob eine Variable überschrieben wird oder nicht. Desweiteren kann die maximale Anzahl der Iterationen ermittelt werden, nachdenen eine Variable überschrieben wird.
Die Berechnung von maximalen Iterationsdistanzen zu allen Knoten eines Schleifenkontrollflußgraphen ist daher der Kern der Analyse.

Definition 4.3.1 *Die* Iterationsdistanz *beschreibt die Anzahl der Iterationen x, über deren Dauer die Lösung an einem Knoten n gültig ist.*

4.3.1 Verwendeter Datenflußverband

Die *Delta-Array-Datenflußanalyse* dient der Ermittlung der maximalen Iterationsdistanzen δ. Dieses wird durch den verwendeten Datenflußverband mathematisch umgesetzt.
Bei skalaren Datenflußanalysen wird der binäre Verband zur Berechnung der Iterationsdistanzen benutzt. Dieser wird bei der *Delta-Array-Datenflußanalyse* zu einem mehrwertigen, linearen und streng monoton steigenden Verband L mit $L = \{\bot, 0, 1, 2, \ldots, \top = UB - 1\}$ verallgemeinert.
UB steht für die maximale Iterationsanzahl der Schleife. Der Infimum Wert $\bot$ bezeichnet die Ungültigkeit einer Eigenschaft. In diesem Fall bedeutet das, daß der Wert seine Gültigkeit unmittelbar verliert. Das Supremum $\top$ kennzeichnet die Gültigkeit eines Wertes über alle Iterationen hinweg.
Die *Delta-Array-Datenflußanalyse* ordnet jeder Referenz ein Verbandelement zu.

4.3.2 Verwendete Operatoren

Die *Delta-Array-Datenflußanalyse* benötigt eine Minimum Funktion zur Realisierung des $\wedge$-Verbandoperators, eine Maximum Funktion für den $\vee$-Operator und einen Inkrement Operator, der den Übergang von einer Iteration in die nächste beschreibt. Mit Hilfe der $\wedge$– und $\vee$-Operatoren wird die Veränderung der Iterationsdistanz modelliert, die beim Aufeinandertreffen der verschiedenen Kontrollflußpfade entstehen können.

Min Funktion:

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

Max Funktion:

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

Inkrement-Operator:

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

4.3.3 Transferfunktionen

Das Verhalten der Instruktionen wird mit Hilfe der Transferfunktionen abstrahiert, wobei unterschieden wird, ob eine Instruktion die Gültigkeit einer Datenflußeigenschaft erzeugt oder vernichtet. Aus diesem Grund sind den Knoten n des Schleifenkontrollflußgraphen LCFG erzeugende und vernichtende Array-Referenzen zugeordnet, aus denen sich die Transferfunktionen f_n konstruieren lassen.
Die Transferfunktionen werden dahingehend klassifiziert, ob sie eine Iterationsdistanz erstmalig erzeugen, ob sie die Iterationsdistanz erhalten oder ob sie zur Modellierung eines Iterationsüberganges dienen.

- **Erzeugungsfunktionen**

  Eine Erzeugungsfunktion beschreibt den beginnenden Geltungsbereich einer Referenz d in einem Knoten n des Schleifenkontrollflußgraphen LCFG. Zu Beginn erhält die Eigenschaft ihre anfängliche Gültigkeit mit der Iterationsdistanz Null.

  Formal gilt für den Fall, daß in Knoten n die Referenz d mit der Distanz 0 erzeugt wird:

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

- **Erhaltungsfunktionen**

  Mittels einer Erhaltungsfunktion wird die Gültigkeit von Werten zwischen zwei abhängigen Referenzen dargestellt. Sie dienen der Anpassung der Iterationsdistanzen innerhalb des Kontrollflußes. Dabei können die Iterationsdistanzen unverändert bleiben, verringert werden oder vollständig zurückgesetzt werden.
  Wenn der Kontrollfluß einen Knoten im Kontrollflußgraphen durchläuft, ist die maximale neue Iterationsdistanz zu bestimmen, da Instanzen innerhalb des aktuell betrachteten Knotens vorherige Definitionen vernichten können.

  Die Erhaltungsfunktion ist formal folgendermaßen definiert:

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

  Die Konstante p_n^d wird mittels einer Fallunterscheidung bestimmt. Hierbei ist d die Definition, die den Knoten n passiert.

- **Exit-Funktionen**

  Mit Hilfe der Exit-Funktion wird der Übergang in die nächste Iteration der Schleife modelliert. Dabei wird die Induktionsvariable um eins inkrementiert. Dadurch werden die Iterationsdistanzen eintreffender gültiger Eigenschaften erhöht. Die Exit-Funktions ist wie folgt definiert:

$f^d_{exit}(x) = x++$

4.3.4 Iteratives Lösungsverfahren

Abschließend, nachdem für jeden Knoten des LCFG eine entsprechende Transferfunktion aufgestellt worden ist, wird im letzten Schritt der Datenfluß bestimmt. Dieser wird mit Hilfe eines Datenfluß-Gleichungssystems modelliert.
Dazu werden für Knoten n der Schleife die beiden Vektoren $IN[n] = \{x_1, \ldots, x_m\}$ und $OUT[n] = \{y_1, \ldots, y_m\}$ gebildet, mit m =Anzahl der Knoten innerhalb des LCFG.
Die maximale Iterationsdistanz zwischen zwei Knoten n und i bei Erreichen des Knotens n wird dabei durch x_i dargestellt und die bei Verlassen des Knoten n durch y_i.
Wenn nun $IN[n,d] = x_d$ die maximalen Iterationsdistanzen x_d und y_d zwischen den Knoten n und d beim Erreichen des Knotens n und analog $OUT[n,d] = y_d$ beim Verlassen des Knoten n bezeichnen, kann das Datenfluß-Gleichungssystem folgendermaßen initialisiert werden:

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

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

Bei Erscheinen einer Definition in einem Knoten wird deren Gültigkeit in OUT durch $\top$ überschätzt und ansonsten durch die Minimum Funktion ($\wedge$) unverändert durch den Knoten transportiert.
Alle Werte des ersten Knoten in IN werden durch $\bot$ unterschätzt, da den ersten Knoten des Schleifenkörpers noch keine Definitionen erreichen. Bei allen anderen Knoten wird mit der minimalen Iterationsdistanz aller Vorgängerknoten $pred(n)$ gerechnet.

Nach Durchführung der Initialisierung wird das Datenfluß-Gleichungssystem gelöst. Dazu wird ein iteratives Verfahren angewendet, das so lange ausgeführt wird, bis sich das Gleichungssystem stabilisiert, d.h. in der i-ten Iteration das Ergebnis mit $i-1$-ten Iteration übereinstimmt.
Die Iterationsschritte nach der Initialisierung sehen formal folgendermaßen aus:

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

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

4.3.5 Parametrisierung

Die Parametrisierung erlaubt die Anwendung des Verfahrens an verschiedene Datenflußprobleme durch minimale Anpassung des Verfahrenablaufs. Durch die Wahl verschiedener Parameter lassen sich Vorwärts– und Rückwärtsanalysen durchführen sowie Must– und May-Probleme (s.u.) behandeln.

Bei der Vorwärtsanalyse wird der Datenfluß vorwärts transportiert und analog bei der Rückwärtsanalyse rückwärts betrachet. Um mit der *Delta-Array Datenflußanalyse* Rückwärtsprobleme zu bearbeiten, wird der Schleifenkontrollflußgraph LCFG, auf dem die Analyse operiert, invertiert. Dadurch fließt der Datenfluß rückwärts und positive und negative Iterationsdistanzen kehren sich um, was durch eine kleine Anpassung bei der Aufstellung der Erhaltungsfunktionen berücksichtigt werden muß [BF99].

Abhängig von der späteren Verwendung der gesammelten Datenflußeigenschaften, interessiert man sich für Datenflußeigenschaften, die an einem Knoten n gelten können (may), oder an n gelten müssen ($must$).
Bisher wurde das Verfahren zur Analyse von *Must*-Problemen beschrieben. Für den Einsatz der *Delta-Array Datenflußanalyse* für *May*-Probleme müssen die partielle Ordnung des Verbandes L, dessen Operatoren, die Erhaltungsfunktionen und die Initialisierung des Datenfluß-Gleichungssystems angepasst werden.

Die Ordnung des Verbandes L muß für die *May*-Analyse umgekehrt werden, so daß gilt $L = \{\perp = UB - 1, \ldots, 0, \top\}$.

Ab jetzt bedeutet $\top$ die Gültigkeit über keine Distanz und $\perp$ die Gültigkeit über alle Distanzen. Aus diesem Grund müssen die Operatoren $\wedge$ und $\vee$ des Verbands L angepaßt werden, indem die jeweiligen Operatoren vertauscht werden, so daß gilt: $\wedge = max$ und $\vee = min$.

Bei einem *May*-Problem verliert eine Definition erst ihre Gültigkeit, wenn dies eindeutig nachgewiesen ist. Daher wird die Lösung im Gegensatz zum *Must*-Problem überschätzt. Beim *Must*-Problem kam es bereits bei einem potentiellen Konflikt zu einer Unterschätzung der Lösung.
Aus dem Grund wird die Erhaltungsfunktion, insbesondere die Bestimmung der Konstante p_n^d, angepasst. Durch die Überschätzung der Lösung wird ausgedrückt, daß Instanzen von d solange erhalten bleiben, bis diese mit Sicherheit vernichtet worden sind. Details zur Berechnung der Konstanten finden sich in [BF99].

Abschließend muß nur noch die Initialisierung des Datenfluß-Gleichungssystems geändert werden um der Veränderung der partiellen Ordnung des Verbandes L gerecht zu werden. Dazu werden alle Werte der IN und OUT Vektoren mit dem Wert $\perp$ vorbelegt und somit wird vor dem Lösen des Datenfluß-Gleichungssystems die Lösung überschätzt.

Kapitel 5

Standardoptimierungen

Dieses Kapitel gibt eine Übersicht über gängige Low-Level Standard Optimierungen die im Rahmen dieser Diplomarbeit auf den *GeLIR* Datenstrukturen implementiert worden sind. Jede Optimierung wird dazu mit Codebeispielen ausführlich vorgestellt.

5.1 Einführung

Im Rahmen dieser Diplomarbeit wurden folgende Standardoptimierungen auf der Low-Level IR *GeLIR* implementiert:

- Constant Folding
- Constant Propagation
- Copy Propagation
- Dead Code Elimination

Die Wahl fiel auf die o.g. Optimierungen, da es sich hierbei um oft benötigte Standardoptimierungen handelt. Diese Optimierungen bilden die Basis für weitere, aufwendigere Optimierungen auf den *GeLIR* Datenstrukturen, die wiederum Potential für diese Optimierungen nach sich ziehen können. Insbesondere komplexere Optimierungen wie die ab Kapitel 6 vorgestellte *Redundant Load/Store Elimination* modifizieren den Programmcode u.U. so, daß die hier besprochenen Standardoptimierungen anschließend zum Einsatz kommen sollten.
Auch parallel zu dieser Diplomarbeit entwickelte *GeLIR* Optimierungen, z.B. die Ausnutzung von *Zero Overhead Hardware Loops*, profitieren beispielsweise von der *Dead Code Elimination*. Die umfangreicheren Optimierungen sind in der

Regel auf die Analyse und Optimierung des Programmcodes nach bestimmten Gesichtspunkten beschränkt und verlassen sich darauf, daß die gängigen Standardoptimierungen den Code in weiteren Durchläufen erneut bearbeiten.
Das für den *encc* Compiler verwendete *LANCE2* Frontend beherrscht bereits einen Großteil dieser Optimierungen. Allerdings sind diese auch auf der *GeLIR* Darstellung notwendig, damit in späteren Phasen optimiert werden kann, da es bisher keinerlei auf der *GeLIR* arbeitenden Optimierungen gab.
Die im Rahmen dieser Diplomarbeit implementierten generischen Optimierungen bilden darüber hinaus den Ausgangspunkt für architekturspezifische Optimierungen.

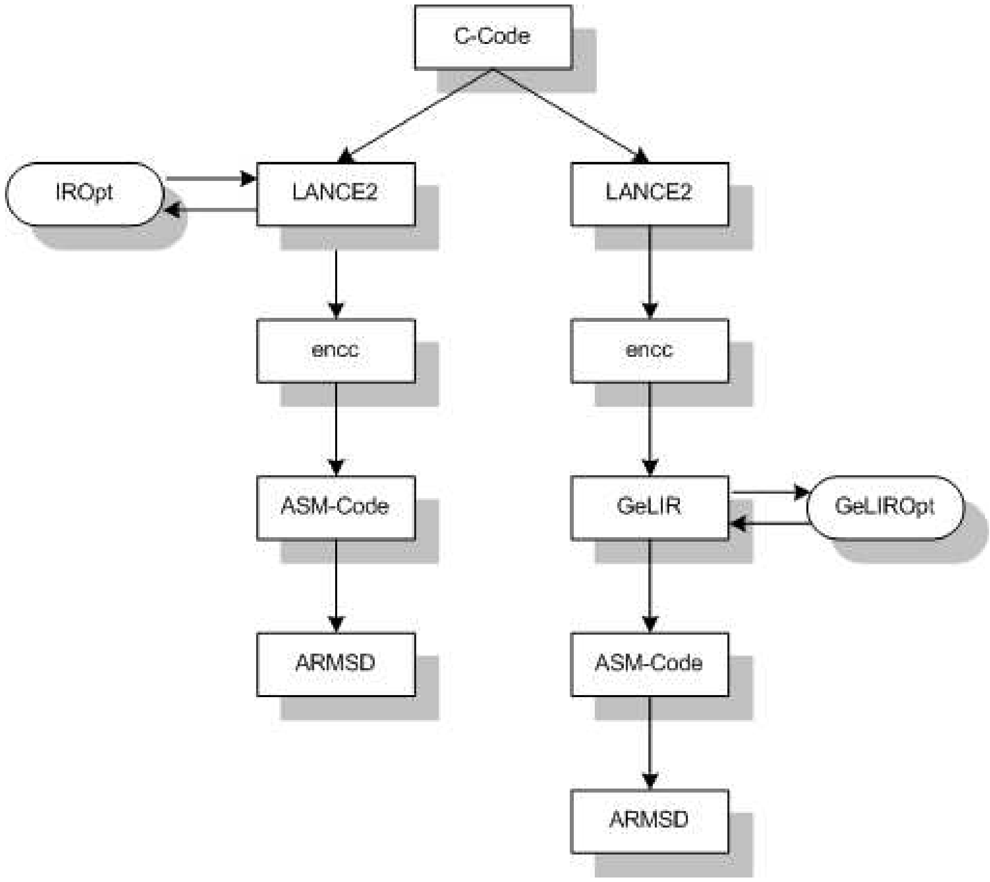

Abbildung 5.1: Einordnung der Optimierungen

Abbildung 5.1 zeigt die Einordnung der Optimierungen. Die linke Seite der Grafik stellt den bisherigen Weg dar. Hier wird der C-Code durch das *LANCE2* Compiler Frontend in die IR-Darstellung übersetzt auf der architekturunabhängige Standardoptimierungen angewendet werden. Das *encc* Compiler Backend erzeugt dann aus der optimierten IR-Darstellung den Assemblercode, der mit Hilfe des ARMulators *ARMSD* analysiert werden kann.
Diese Diplomarbeit bestreitet einen neuen Weg, der in der rechten Seite von Abbildung 5.1 dargestellt wird. Das *LANCE2* Frontend wird hier lediglich zur Er-

zeugung der IR-Darstellung verwendet, die dann wiederum vom *encc* Backend übersetzt wird. Dort greift der bereits in Kapitel 3 vorgestellte CFG2GeLIR Konverter ein und transformiert die interne *encc* Darstellung des Kontrollflußgraphen in die *GeLIR* Darstellung, auf der dann die Standardoptimierungen operieren.
Da in dieser Phase bereits zusätzlich alle Informationen der Codegenerierung und Registerallokation vorliegen, und diese mit in die *GeLIR* Darstellung übernommen worden sind, können die Optimierungen nun architekturspezifische Details ausnutzen. Die konkreten Ansätze dazu werden im Anschluß an jede Optimierung vorgestellt.
Desweiteren ist es möglich, aus der *GeLIR* Darstellung Assemblercode zu generieren, da alle nötigen Informationen dazu vorliegen.

Die folgenden Unterkapitel stellen die Optimierungen vor und berücksichtigen dabei die architekturunabhängigen sowie die architekturabhängigen Aspekte und geben einen Überblick über den Ablauf der einzelnen Optimierungen inkl. aller relevanten implementationsbedingten Details.
Informationen zur Einbindung und Konfiguration der Optimierungen in eigene Projekte finden sich in Anhang B.
Ein zusammenfassendes Beispiel am Ende dieses Kapitels macht das Zusammenspiel der einzelnen Optimierungen deutlich.

5.2 Constant Folding

Das *Constant Folding* bezieht sich auf die Auswertung von Werten, die als konstant bekannt sind, zur Übersetzungszeit durch den Compiler. *Constant Folding* wird in der Literatur auch als *Constant-expression evaluation* bezeichnet [MUC97].

Hierbei handelt es sich um eine datenflußunabhängige Optimierung, da immer nur ein einzelner Ausdruck betrachtet wird. Wenn der Algorithmus alle Operanden einer Operation als konstant erkennt, so wird die Operation durch den berechneten Wert ersetzt. Solche Ausdrücke treten auf Quellcodeebene eher selten auf, sind aber auf der Zwischencodeebene als Folge anderer Optimierungen und innerhalb von Adressierungsausdrücken recht häufig anzutreffen.

Boolesche Werte und Integer Konstanten können problemlos gefaltet werden. Fließkommazahlen müssen gesondert berücksichtigt werden. Hier spielt die Fließkommaarithmetik der Zielarchitektur eine Rolle, die sich nicht von der Architektur der Compilerplattform unterscheiden darf.

```
Original Code:

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

  for (i = 0; i < 10; i++)
  {
    j = 2 + 3 * 4 - 2;
  }

  return j;
}
```

```
Nach Constant Folding:

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

  for (i = 0; i < 10; i++)
  {
    j = 12;
  }

  return j;
}
```

Beispiel 5.2.1 *Constant Folding*

Wie auch im abschließenden Beispiel am Ende dieses Kapitels erkennbar sein wird, bietet es sich zur Effektivitätssteigerung an, den *Constant Folding* Algorithmus in Kombination mit einer datenflußabhängigen Optimierung, etwa der *Constant Propagation*, durchzuführen.

Abbildung 5.2 zeigt den Ausschnitt eines Beispielprogramms vor der Durchführung der *Constant Folding* Optimierung. Interessant sind die MIs 9 und 11. Dort findet die Subtraktion `14 - 2` statt, deren Ergebnis in `V7` abgelegt und später verwendet wird. Hier liegt Optimierungspotential für das Constant Folding.

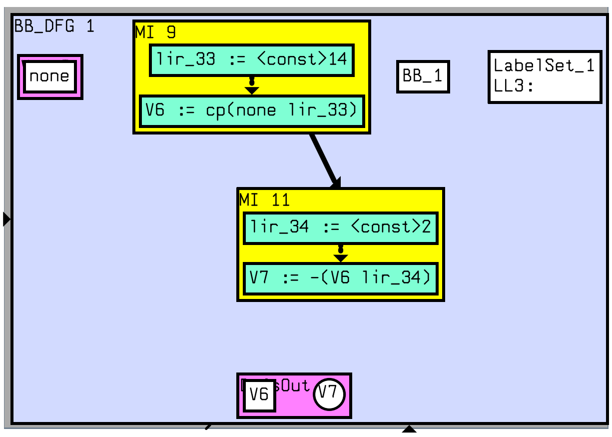

Abbildung 5.2: Codeausschnitt vor Constant Folding Optimierung

Nach dem Constant Folding ist in Abbildung 5.3 erkennbar, daß die Subtraktion durch eine direkte Zuweisung des korrekten Ergebnisses 12 in `V7` ersetzt worden ist. MI 9, die vorher Teil der Subtraktion war, befindet sich noch als nutzloser

Code in der *GeLIR* Darstellung. Eine anschließende *Dead Code Elimination* würde diese überflüssige MI entfernen.

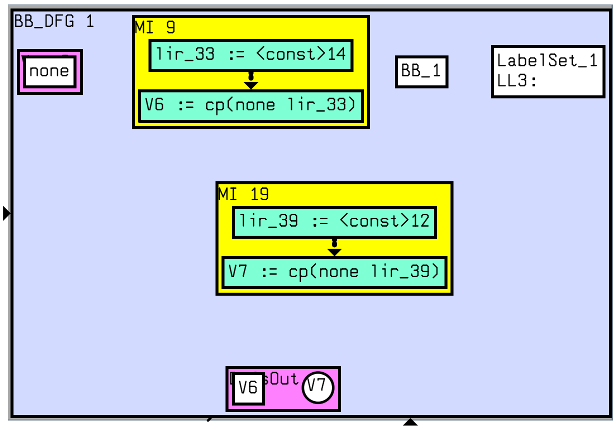

Abbildung 5.3: Codeausschnitt nach Constant Folding Optimierung

5.2.1 Architekturspezifisches

Handelt es sich bei den Operanden eines Ausdrucks um boolesche Werte, so ist das *Constant Folding* auf jeden Fall praktizierbar. Bei Integer Werten gibt es Ausnahmen, wenn bei der Berechnung Laufzeitfehler (z.B. Division durch Null) oder Überläufe entstehen können. In diesen Fällen verzichtet man in der Regel auf das *Constant Folding*. Überläufe stellen kein Problem dar, wenn die Arithmetik der Zielmaschine mit der Arithmetik des Compilers identisch ist bzw. berücksichtigt wird.

Komplizierter ist die Situation bei Fließkommawerten, da die Fließkommaarithmetik auf der Zielplattform und die des Compilers gleich sein oder entsprechend berücksichtigt werden muß, um hier korrekte Werte zu erhalten. So muß die Genauigkeit auf beiden Systemen übereinstimmen, um Rechenfehler zu vermeiden. Desweiteren gibt es nach dem ANSI/IEEE-54 Standard (Kodierung der Fließkommawerte mit 32-/64- und 80-Bit Genauigkeit) wesentlich mehr Ausnahmebehandlungen zu beachten als bei einer Integerarithemtik.
Gibt es Unterschiede zwischen Compiler- und Zielarchitektur, so muß die Fließkommaarithmetik geeignet simuliert werden oder das *Constant Folding* muß auf die Faltung von Boolean und Integer Werten beschränkt werden.

Bei der Behandlung von Fließkommawerten zeigt sich die Stärke der *GeLIR* Datenstrukturen. Da die *GeLIR* eine architekturspezifische Darstellung unterstützt, läßt sich auch ein *Constant Folding* auf Fließkommazahlen durchführen, wenn entsprechende Informationen über die Fließkommadarstellung in der Target Beschreibung hinterlegt sind. Das ginge bei vollkommen architekturunabhängigen Implementierungen, wie z.B. *LANCE2*, nicht.

In Folge des *Constant Folding* kann die Wiederverwendung der Ergebnisregister der gefalteten Konstanten zur Senkung des Registerdrucks beitragen, da ein Register eingespart werden kann, wenn z.B. vor der Optimierung zwei Konstanten je ein Register belegt haben, die dann durch *Constant Folding* zu einer Konstanten zusammengefaltet werden konnten.

5.2.2 Ablauf der Analyse

Beim *Constant Folding* werden alle Basisblöcke einer Funktion für jede Operation durchlaufen und es wird überprüft, ob es sich bei der jeweiligen Operation um eine arithmetische oder logische Operation handelt. Wenn dies der Fall ist, muß überprüft werden, ob alle Argumente der Operation Konstanten sind. Ist diese Bedingung erfüllt, werden die konstanten Argumente in geeigneter Weise zusammengefasst und die Operation wird durch eine einfache Wertzuweisung in Form einer Copy-Operation ersetzt.

5.3 Constant Propagation

Bei der *Constant Propagation* werden als konstant erkannte Variablenwerte direkt an ihrer Verwendungsstelle eingesetzt. Dadurch läßt sich evtl. ein Register einsparen, wenn der entsprechende Variablenwert als Immidiate kodierbar ist.

Die *Constant Propagation* Technik kann in der Compiler Optimierung zu verschiedenen Zwecken eingesetzt werden: [WZ91].

- Ausdrücke, die zur Übersetzungszeit ausgewertet worden sind, brauchen nicht mehr während der Ausführungszeit ausgewertet zu werden. Das spart insbesondere in inneren Schleifen Zeit.
- Code, der nie ausgeführt wird, kann gelöscht werden. Solch unerreichbarer Code wird entdeckt, wenn bedingte Sprünge identifiziert werden, die immer den gleichen Sprungpfad nehmen.
- Durch die Erkennung von Pfaden, die nicht ausgeführt werden, läßt sich der Kontrollfluß des Programms vereinfachen.

```
Original Code:                    Nach Constant Propagation:

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

  n = 32;                           n = 32;
  c = 10;                           c = 10;

  for (i = 1; i < n; i++)           for (i = 1; i < 32; i++)
  {                                 {
    b = 5 + c;                        b = 5 + 10;
  }                                 }

  return b;                         return b;
}                                 }
```

Beispiel 5.3.1 *Constant Propagation*

Abbildung 5.4 zeigt den Ausschnitt eines Beispielprogramms vor der Durchführung der *Constant Propagation* Optimierung. Dort ist zu erkennen, daß die Konstanten der MI 3 und MI 7 nicht als Immediatewerte direkt im Vergleichsbefehl der MI 7 verwendet werden, sondern erst durch potentiell teure Speichertransferoperationen, die in der *GeLIR* Darstellung mit `cp` gekennzeichnet sind, in entsprechende Register-Ressourcen kopiert werden. Die Terme, die die Konstanten beinhalten, werden allerdings nicht ausgewertet. Dies geschieht erst durch einen *Constant Folding* Durchlauf.

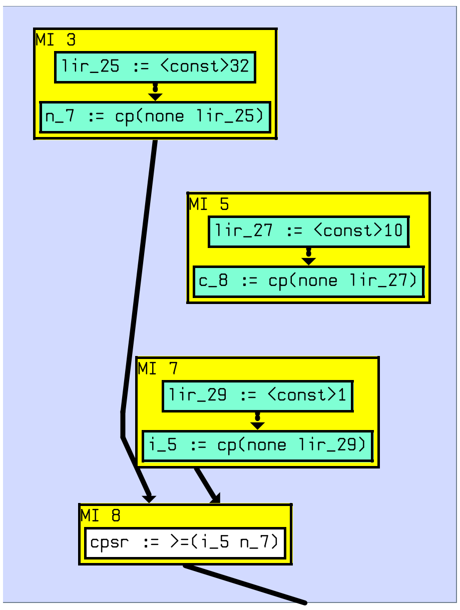

Abbildung 5.4: Codeausschnitt vor Constant Propagation Optimierung

Abbildung 5.5 zeigt das obige Beispiel nach der *Constant Propagation.* Von den beiden Konstanten wurde nur eine als Immediate in dem Vergleichbefehl propagiert, da die zugrundeliegende Zielarchitektur (hier ARM7TMDI) keinen Vergleichsbefehl mit zwei Immediate Werten akzeptiert. Optional kann die Berücksichtigung der Zielarchitektur abgeschaltet werden. In diesem Fall würden beide Konstanten als Immediate in den Vergleichsbefehl eingesetzt. Im Anschluß bietet sich erneut eine *Dead Code Elimination* an, um die überflüssigen MIs (in der Abbildung nicht dargestellt) zu entfernen.

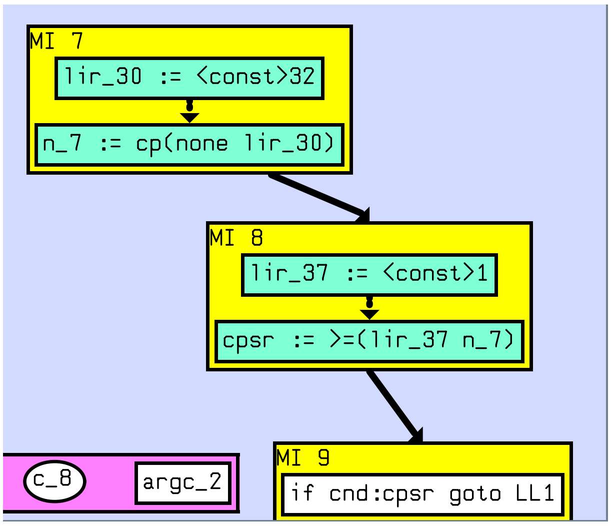

Abbildung 5.5: Codeausschnitt nach Constant Propagation Optimierung

Constant Propagation ermöglicht weitere Optimierungen, z.B. die Dead Code Elimination. Dadurch ergibt sich eine Einsparung von Registern und Instruktionen, wenn möglichst viele Konstanten als Immediate Werte codiert werden können.

Das *Constant Propagation* Problem an sich ist nicht entscheidbar [KU77]. Für bestimmte Instanzen des Problems existieren aber effiziente Algorithmen. Im folgenden werden vier dieser Algorithmen kurz vorgestellt. Jeder dieser vier Algorithmen zur Durchführung der *Constant Propagation* arbeitet "konservativ", d.h. evtl. werden nicht alle Konstanten gefunden, aber die Konstanten, die gefunden werden, sind über alle möglichen Kontrollflüsse konstant.
Der im Rahmen dieser Diplomarbeit implementiere *Constant Propagation* Algorithmus lehnt sich an das *Simple Constant Propagation* Verfahren an.

- **Simple Constant Propagation**

  Zu jedem Basisblock des Datenflußgraphen werden zwei Listen verwaltet. In der ersten Liste wird für jede Variable der Wert gespeichert, den sie beim Eintritt in den Basisblock hat. Die andere Liste enthält den Wert beim Verlassen des Basisblocks.

Der *Simple Constant Propagation* Algorithmus berücksichtigt alle möglichen Ausführungspfade, unabhängig davon, ob sie auch wirklich ausgeführt werden und entlang jedes Pfades wird immer nur ein Wert pro Variable mitgeführt.
Detailierte Informationen zu dem Algorithmus lassen sich in [KIL77] nachlesen.

- **Conditional Constant Propagation**

  Im Gegensatz zum *Simple Constant Propagation* werden beim *Conditional Constant Propagation* Variablendefinitionen in Programmzweigen ignoriert, die nie ausgeführt werden. Durch diese Eigenschaft kann der Algorithmus eine größere Anzahl von Konstanten finden und gleichzeitig eine Elimination von unerreichbarem Code durchführen.
  Weitere Informationen finden sich in [WZ91].

- **Sparse Simple Constant Propagation**

  Die Ergebnisse eines *Sparse Simple Constant Propagation* Algorithmus sind vergleichbar mit denen der *Simple Constant Propagation*. Allerdings erreicht *Sparse Simple Constant Propagation* eine höhere Ausführungsgeschwindigkeit durch Ausnutzung der SSA-Form.
  Im Gegensatz zu einer gängigen IR Darstellung, werden bei Programmen in *Static Single-Assignment* (SSA) Form, die verwendeten Werte von ihren Speicherplätzen getrennt dargestellt um evtl. effizientere Analysen und Optimierungen des Programmverhaltens zu bekommen. Dazu wird verlangt, daß jede Variable maximal einmal definiert werden darf, da so auch zusammenhängende Verwendungen der gleichen Variable aufgelöst werden können, da sie zu verschiedenen Variablen werden.
  Die Transformation eines Programms in SSA-Form wird mittels einer ϕ-Funktion durchgeführt. Weitergehende Details zur SSA-Form finden sich in [AG98] und [MUC97].
  Eine ausführliche Beschreibung des *Sparse Simple Constant Propagation* Algorithmus kann in [MUC97] nachgelesen werden.

- **Sparse Conditional Constant Propagation**

  Sparse Conditional Constant Propagation basiert auf *Conditional Constant Propagation* und benutzt zur Beschleunigung ebenfalls die SSA-Form.
  Details können in [WZ91] nachgelesen werden.

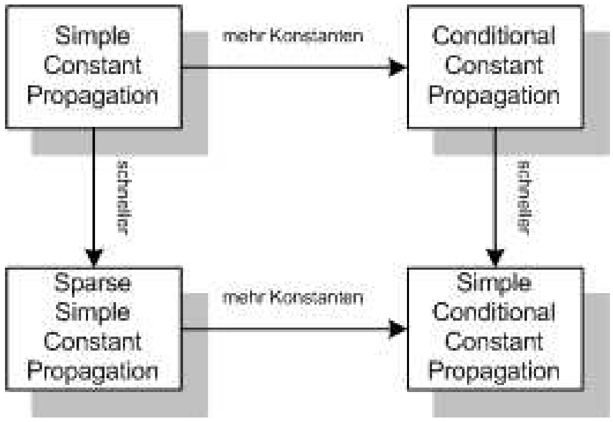

Abbildung 5.6: Beziehungen zwischen den vier Constant Propagation Algorithmen

5.3.1 Architekturspezifisches

Die Optimierung *Constant Propagation* ist architekturunabhängig implementiert worden, damit sie auch zukünftig mit möglichst vielen Zielarchitekturen, die evtl. noch an die *GeLIR* angebunden werden, nutzbar ist. Über ein Flag in der *GeLIR* Konfigurationsdatei (siehe Anhang B) kann ein Modus aktiviert werden, der architekturspezifische Besonderheiten ausnutzt.
So können etwa RISC Befehle ausgenutzt werden, die eine (u.U. kleine) Integer Konstante als Operanden zulassen. Das wird erreicht, indem die max. Befehlswortlänge der Zielarchitektur berücksichtigt wird. So unterstützt die ARM CPU z.B. max. 5-Bit unsigned Konstanten bei Shift Operationen im THUMB Modus, die LEON CPU in der Regel hingegen 13-Bit sign-ext. Konstanten.

```
Vorher:                 Nach Ausnutzung der Immediate Befehle:
MOV r0,#7               ADD r1,r1,#7
ADD r1, r1, r0
```

Beispiel 5.3.2 *Ausnutzung eines Immediate Befehls*

Ist der architekturspezifische Modus aktiviert, so kann die max. Größe der Immediate Konstanten, die in der Target Beschreibung abgelegt ist, berücksichtigt werden. Somit wird die *Constant Propagation* nur für Instruktionen angewendet, deren Immediate eine vorgegebene Grenze nicht überschreitet, um somit möglichst effizienten Code zu erzeugen.
Auf der Ebene der abstrakten Operationen kann ebenfalls die max. Größe der Immediate Konstanten für jede Befehlsklasse berücksichtigt werden. Dazu werden wie in Anhang B beschrieben, die nötigen Informationen aus einer Konfigurationsdatei eingelesen.

Außerdem werden im architekturspezifischen Modus alle möglichen Überdeckungen der zu modifizierenden Maschinenoperation vor der Propagierung der entsprechenden Konstanten berücksichtigt. Dadurch wird vermieden, daß Immediate Werte in Maschinenoperationen eingesetzt werden, die sich dann später nicht auf eine reale Assemblerinstruktion der Zielarchitektur abbilden lassen und durch eine erneute Code Selektion rückgängig gemacht werden müssten. Dieses Merkmal wurde bereits im obigen Beispiel angewendet.

Die *Constant Propagation* unterstützt indirekt ein weiteres Merkmal von RISC Architekturen: Einige RISCs haben einen Adressierungsmodus, der die Summe von einem Register und einer (kleinen) Konstante benutzt, aber nicht die Summe von zwei Registern. Wenn mindestens eines der Register durch eine Konstante ersetzt werden kann, können die abstrakten Instruktionen direkt auf eine reale RISC Instruktion abgebildet werden.

5.3.2 Ablauf der Analyse

Im Rahmen dieser Diplomarbeit wurde die Variante der *Simple Constant Propagation* implementiert. Die *Sparse Simple/Conditional Constant Propagation* kam nicht in Frage, da die Kontrollflußdarstellung der *GeLIR* in eine SSA-Form transformiert werden müsste, um die Optimierung durchzuführen. Dieser Aufwand wäre nicht angemessen gewesen, da es bei der Implementierung der hier betrachteten Optimierungen nicht so sehr auf die Laufzeit ankommt, solange die Güte der Optimierungsergebnisse nicht darunter leidet.

Bei der Analyse kommt eine Verband-(Lattice-) Datenstruktur zum Einsatz, wie sie bereits in Kapitel 4 beschrieben worden ist.
Die Ausgabe eines *Constant Propagation* Algorithmus ist eine Zuweisung von Verbandelementwerten für eine Variable zu jedem Knoten des Kontrollflußgraphen. Das höchste Verbandelement ist $\top$, das niedrigste Element ist $\bot$ und alle Elemente zwischen $\top$ und $\bot$ sind konstant. Es gibt eine unendliche Anzahl dieser konstanten Verbandelemente, passend zu jeder Konstanten i.
Jedem Programmknoten sind Zellen zugeteilt, sogenannte $LatticeCells$ oder Verbandzellen, die aus Verbandelementen bestehen. Die Werte der Verbandelemente ändern sich, während der Algorithmus abgearbeitet wird. Die Ergebnisse und Operanden von Ausdrücken sind mit den $LatticeCells$ verknüpft.
Wird am Ende der *Constant Propagation* einer $LatticeCell$ eine Konstante zugewiesen, so bedeutet dies, daß der entsprechende Operand oder das Ergebnis auf allen möglichen Ausführungspfaden immer den gleichen Wert besitzt, wenn der der $LatticeCell$ zugehörige Knoten verlassen wird. Eine Zuweisung des $\bot$ Elements bedeutet, daß ein konstanter Wert nicht garantiert werden kann und die Zuweisung des $\top$ Elements bedeutet, daß die Variable bis jetzt als unbekannte Konstante erkannt worden ist. Nach Terminierung des *Constant Propagation* Algorithmus sind alle $LatticeCells$ der ausführbaren Knoten entweder $\bot$ oder konstant.

Der Algorithmus beginnt mit der optimistischen Annahme, bei der alle Verbandselemente mit $\top$ vorinitialisiert werden, also davon ausgegangen wird, daß die entsprechenden Variablen konstant sind. Die Variablen des Startknotens werden mit $\perp$ initialisiert, d.h. die Variable ist nicht konstant, sofern die Werte der entsprechenden Variablen nicht bekannt sind.

Ist eine Variable innerhalb eines Basisblocks nicht Ziel einer Zuweisung, so wird der Wert der Variablen durch die Ausführung des Basisblocks nicht verändert. Eine Zuweisung würde den Zustand des Verbandelements der Variablen im Basisblock verändern und kann den Wert der Verbandelemente der anderen Basisblöcke ebenfalls beeinflussen, so daß diese Basisblöcke dann neu ausgewertet werden müssen.
Verwendet die rechte Seite der Zuweisung eine Variable mit dem Verbandselement $\perp$, so bekommt das Verbandelement der Variablen ebenfalls den Wert $\perp$, die Variable ist also nicht konstant. Wenn die rechte Seite der Zuweisung kostant ist, so wird auch das Verbandselement auf diesen konstanten Wert gesetzt.

Beim *Simple Constant Propagation* Algorithmus werden in jedem Basisblock im Kontrollflußgraph pro Variable zwei $LatticeCells$ mit dem Wert der Variable verwaltet. Zum einen wird der Wert der Variable gespeichert, den sie beim Eintritt in den Basisblock hat (*VarsIn*-Liste) und zum anderen der Wert, der beim Verlassen des Basisblocks gültig war (*DefsOut*-Liste). Der Vorgang des Besuchens eines jeden Knoten K beinhaltet die Untersuchung jeder $LatticeCell$ an dem entsprechenden Knoten.

Zu Beginn ist eine Arbeitsliste mit dem Startknoten initialisiert. Ein Knoten K wird aus der Arbeitsliste gewählt, entfernt und untersucht. Der Wert des Verbandelements der *VarsIn*-Liste von K wird durch Verknüpfung mit den *Defs-Out*-Listen der Vorgängerknoten bestimmt und ausgewertet. Dazu wird der $\sqcap$-Operator verwendet, der wie folgt definiert ist:

Sei v_i ein Verbandelement der Konstante i:

$v_i \sqcap \top = v_i$
$v_i \sqcap \perp = \perp$
$v_i \sqcap v_j = v_i, \forall i = j$
$v_i \sqcap v_j = \top, \forall i \neq j$

Enthält K eine Zuweisung, so können sich die Werte der $DefsOut$-Listen ändern, so daß in diesem Fall alle Nachfolger von K neu ausgewertet werden müssen. Dazu werden sie in die Arbeitsliste eingetragen. Ändern sich keine Verbandelemente, so werden keine Knoten in die Arbeitsliste eingefügt. Das Verfahren wiederholt sich bis die Arbeitsliste leer ist.

Der Algorithmus berücksichtigt alle Ausführungspfade, unabhängig davon, ob sie wirklich durchlaufen werden und findet alle konstanten Variablen. Dazu wird überprüft, ob zwischen der ersten Definition einer Variablen und deren Gebrauch sich die Variable auf den Kontrollflußwegen ändert.
Ändert sicht die Variable nicht, so wird das entsprechende Argument der Ma-

schinenoperation, die die Variable benutzt, durch den Wert der Variable ersetzt, die sie bei ihrer Definition hatte.

Es gibt ein der *Constant Propagation* ähnliches Verfahren, daß nicht nur für Konstanten, sondern auch für Kopien von Variablen geeignet ist, nämlich die *Copy Propagation*.

5.4 Copy Propagation

Bei der *Copy Propagation* werden Kopien von Variablen-Werten durch die Benutzung der ursprünglichen Variablen ersetzt.

Satz 5.4.1 *Wenn es Zuweisungen der Form $f \leftarrow g$ gibt, kann jede Verwendung von f durch g ersetzt werden, wenn sich g zwischenzeitlich nicht ändert. Gelingt es, alle Referenzen auf f zu ersetzen, so ist $f \leftarrow g$ sowie f selbst überflüssig.*

```
Original Code:                      Nach Copy Propagation:

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

  for (i = 1; i < 10; i++)            for (i = 1; i < 10; i++)
  {                                   {
    z = i;                              z = i;
    y = z + 14;                         y = i + 14;
    y = i + z;                          y = i + i;
  }                                   }

  return i;                           return i;
}                                   }
```

Beispiel 5.4.1 *Copy Propagation*

Im obigen Beispiel ist `z` eine Kopie des Wertes von `i`. Da `i` zwischenzeitlich nicht verändert wird, können die Vorkommen von `z` durch `i` ersetzt werden, so daß die Anweisung `z = i` überflüssig wird. Abbildung 5.7 zeigt einen Ausschnitt der *GeLIR* Darstellung des obigen Beispiels vor der Durchführung der *Copy Propagation*.

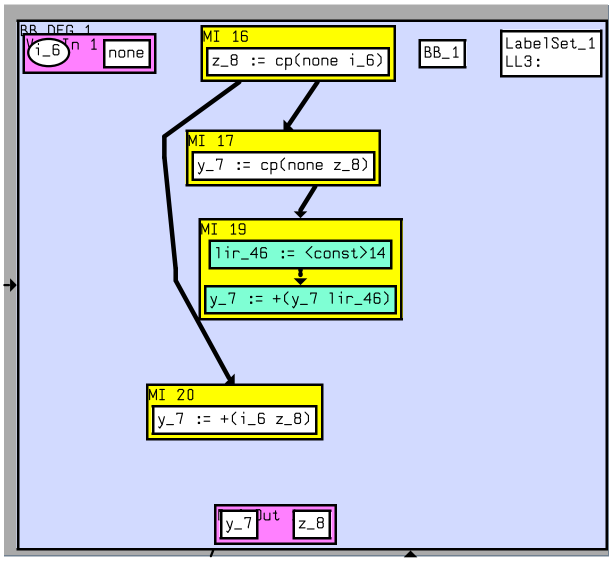

Abbildung 5.7: Codeausschnitt vor Copy Propagation Optimierung

In Abbildung 5.8 ist die *GeLIR* Darstellung nach dem Optimierungslauf zu sehen. In den MIs 19 und 20 wurden die Verweise auf `z` durch die entsprechenden Verweise auf `i` ersetzt. Dadurch wird die Zuweisung `z = i`, die durch die MIs 16 und 17 dargestellt wird, überflüssig und kann durch einen anschließenden *Dead Code Elimination* Durchlauf gelöscht werden, da `z` innerhalb der Schleife nicht mehr verwendet wird.

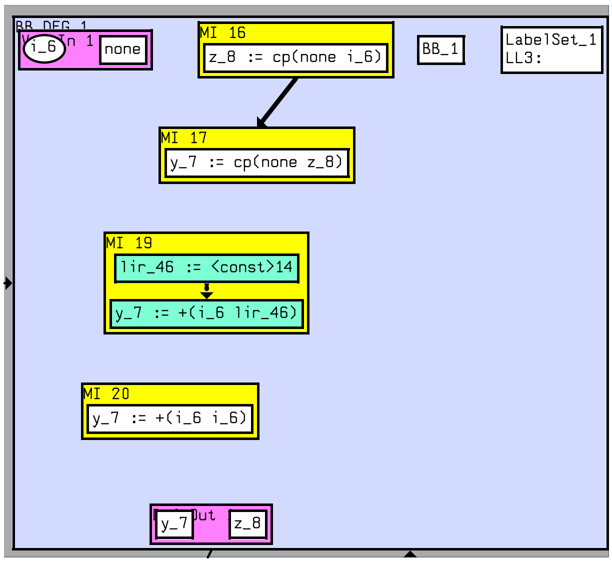

Abbildung 5.8: Codeausschnitt vor Copy Propagation Optimierung

Die *Copy Propagation* trägt indirekt zur Reduzierung der Codegröße bei und sorgt für eine Verringerung des Registerdrucks, wenn dies nicht bereits bei der Register Allokation im Rahmen des Coalescing [MUC97] erkannt worden ist.

Da jedoch nicht davon ausgegangen werden kann, daß jede Register Allokation auch ein Coalescing durchführt, kann in so einem Fall die *Copy Propagation* Optimierung zum Einsatz kommen.

5.4.1 Architekturspezifisches

Auch bei der *Copy Propagation* kann man architekturspezifische Merkmale ausnutzen. Bei einigen Befehlen gibt es Abhängigkeiten der Ressourcen von bestimmten Registern, so sind z.B. nicht alle Register als Operanden für eine Multiplikationsanweisung erlaubt. In so einem Fall darf die *Copy Propagation* nur auf Befehle durchgeführt werden, deren Register "kompatibel" sind.
Durch den RISC typischen homogenen Registersatz des ARM Prozessors treten diese Fälle in der Praxis selten auf. Eine Ausnahme bilden die hohen Register des ARM Befehlssatzes, die im THUMB Modus nur bei bestimmten Operationen zulässig sind.
Auch der LEON besitzt Register, die nur eingeschränkt nutzbar sind. So wird z.B. bei der MAC-Operation des LEON ein sog. `asr18` Register zur Berechnung benutzt, auf das nur mit den beiden Befehlen `rdasr` lesend und `wrasr` schreibend zugegriffen werden kann.

In der *GeLIR* können die Ressourcen zusammen mit den Befehlen modelliert werden. Dadurch ist ein Abgleich der erlaubten Ressourcen für jeden Befehl möglich. Details dazu können in [FLW01] nachgelesen werden.

5.4.2 Ablauf der Analyse

Ablauf und Implementierung der Analyse orientieren sich an [MUC97]. *Copy Propagation* ist eng verwand mit dem Verfahren *Register Coalescing*, kann aber auf einer beliebigen IR angewendet werden. Beim *Register Coalescing* kommt ein Interferenzgraph zur Analyse zum Einsatz, wohingegen die *Copy Propagation* eine einfache Datenflußanalyse anwendet, um festzustellen, ob die sie auf eine Kopieranweisung angewendet werden darf.
Die Analyse wird in zwei Phasen, lokal und global, aufgeteilt. Die erste Phase arbeitet auf Basisblockebene und die zweite Phase über den gesamten Kontrollflußgraphen. Um eine bessere Laufzeit zu erzielen, wird eine Liste mit allen verfügbaren Copy-Anweisungen geführt.
Zur Durchführung der globalen *Copy Propagation* wird zuerst eine einfache Datenflußanalyse angewendet. Dazu werden zwei Mengen `COPY(i)` und `KILL(i)` benötigt. Die `COPY(i)` Menge beinhaltet Quadrupel (u, v, i, p) mit einer Kopieranweisung $u \leftarrow v$, der Basisblocknummer i in der die Anweisung auftritt und weder u noch v später zugewiesen werden, sowie die Position p der Maschinenoperation. Die Menge `KILL(i)` beinhaltet Kopieranweisungen die in Basisblock i ungültig werden. Es ist wiederum ein Quadrupel (u, v, j, p) mit der Kopieranweisung $u \leftarrow v$ die in Basisblock $j \neq i$ an Position p auftaucht.

Die Datenflußgleichungen `IN(i)` und `OUT(i)`, die die für die *Copy Propagation* verfügbaren Kopieranweisungen am Eingang bzw. Ausgang von Basisblock i repräsentieren, sind folgendermaßen definiert:

$$IN(i) = \bigcap_{j \in Pred(i)} OUT(j)$$

Eine Kopieranweisung ist am Anfang eines Basisblocks i verfügbar, wenn sie an allen Ausgängen der Vorgängerbasisblöcke von Basisblock i verfügbar ist. Aus diesem Grund wird der Schnittmengenoperator zur Verbindung der Pfade benutzt.

$$OUT(i) = COPY(i) \cup (IN(i) - KILL(i))$$

Eine Kopieranweisung ist am Ausgang eines Basisblocks i verfügbar, wenn sie in der Menge `COPY(i)` enthalten ist oder wenn sie am Anfang von Basisblock i verfügbar ist und nicht von Basisblock i vernichtet wird.

Initialisierung:

$$IN(entry) = \emptyset$$

$$IN(i) = \bigcup_i COPY(i), \forall i \neq entry$$

Die globale *Copy Propagation* läuft dann in folgenden zwei Schritten ab:

1. Für jeden Basisblock B wird die Menge der verfügbaren Kopieranweisungen ACP folgendermaßen gesetzt:

   $ACP = a \in Var \times Var, \exists w \in \text{integer mit } (a_1, a_2, B, w) \in IN(B)$

2. Die lokale *Copy Propagation* wird für jeden Basisblock B durchgeführt. Dabei werden alle Instruktionen des Basisblocks durchlaufen und die Operanden, bei denen es sich um Kopien handelt, mit Einträgen aus der ACP Menge ersetzt.

5.5 Dead Code Elimination

Wie an den vorhergegangenen Beispielen erkennbar ist, entsteht Dead Code oftmals durch andere Optimierungen. Eine *Dead Code Elimination* ist also unabdingbar und sollte in Wechselwirkung mit den anderen Optimierungen aufgerufen werden. Natürlich kann inaktiver Code auch bereits im Quellprogramm enthalten sein. Er wird selten bewußt eingeführt, entsteht aber häufig als Nebenprodukt verschiedenster Optimierungen oder durch Makroexpansion.

Satz 5.5.1 *Eine Instruktion ist "dead", wenn sie nur Werte berechnet, die auf allen von dieser Instruktion ausgehenden Pfaden nicht benutzt werden.*

Bei der *Dead Code Elimination* muß der globale Datenfluß berücksichtigt werden. Dieser beinhaltet die nötigen globalen Informationen, in welcher Weise eine Funktion oder ein größeres Programmsegment, seine Daten manipuliert. Informationen über den Datenfluß können mit Hilfe einer Datenflußanalyse ermittelt werden. Für die *Dead Code Elimination* reicht eine einfache Datenflußanalyse, wie sie in der *GeLIR* mittels der *VarsIn*- und *DefsOut*-Listen realisiert und in Kapitel 3.4 beschrieben ist. Komplexere Optimierungen greifen auf spezialisierte Datenflußanalysemethoden zurück, wie z.B. die in Kapitel 4 erläuterte *Delta-Array-Datenflußanalyse*.

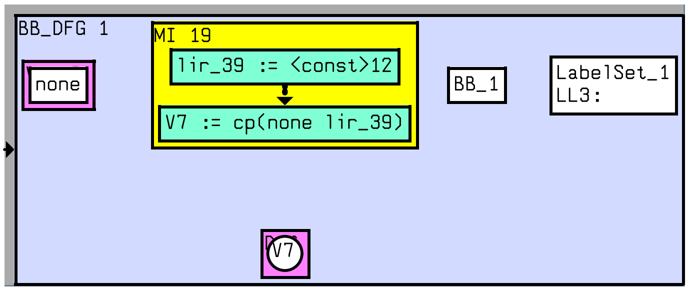

Abbildung 5.9: Codeausschnitt nach Dead Code Elimination

Abbildung 5.9 greift den Kontrollflußgraphausschnitt aus Abbildung 5.3 nach der *Constant Folding* Optimierung erneut auf und zeigt das Ergebnis nach Anwendung der *Dead Code Elimination*. Die in Abbildung 5.3 bereits als überflüssig erkannte `MI 9` wurde entfernt.

5.5.1 Architekturspezifisches

Auch bei der *Dead Code Elimination* können architekturspezifische Besonderheiten ausgenutzt werden. Bestimmte Maschinenoperationen sehen auf dem ersten Blick aus wie nicht benötigter Code, ändern aber beispielsweise den Maschinenzustand der Zielarchitektur, z.B. ein Befehl, der den Cache des LEON Prozessors abschaltet. Solche Maschinenoperationen müssen erkannt und in geeigneter Weise behandelt werden. Die *GeLIR* stellt die nötige Funktionalität hierfür bereit, da bei einer Maschinenoperation zusätzlich ein Flag gesetzt werden kann, daß es sich keinesfalls um Dead Code handelt. Diese Eigenschaft wird von der *Dead Code Elimination* abgefragt.
Desweiteren berücksichtigt die im Rahmen dieser Diplomarbeit implementierte *Dead Code Elimination* in der *GeLIR* modellierte Maschinenoperationen zur Unterstützung von Zero Overhead Hardware Loops. Dazu stellt die *GeLIR* die beiden MO Klassen `ZLOOP` und `ZJMP` bereit. Erstere dient der Modellierung der Schleifeninitialisierung. Mit der zweiten Klasse wird der Vorgang, bei dem der Zähler erniedrigt wird und der Sprung, solange der Zähler nicht Null ist, abgebildet. Diese beiden Vorgänge werden in einer realen Architektur mit Zero

Overhead Hardware Loops Unterstützung automatisch von der Hardware ausgeführt.
Die hier berücksichtigten ARM und LEON Prozessoren unterstützen keine Zero Overhead Hardware Loops. Eine Unterstützung schien aber trotzdem sinnvoll, da die *Dead Code Elimination* auch für andere Architekturen eingesetzt wird, wie z.B. für Schleifenoptimierungen auf der M3-DSP Plattform, die u.A. auch Zero Overhead Hardware Loops unterstützt [MH01].

5.5.2 Ablauf der Analyse

Die *Dead Code Elimination* Implementierung arbeitet basisblockorientiert. Dazu wird zuerst eine Menge `setDeadDefsOut` bestimmt, deren Dead Code Elemente aus der globalen *DefsOut*-Liste des Basisblocks bestimmt wird. Dafür muß überprüft werden, ob es einen Nachfolgerbasisblock gibt, der das *DefsOut* Element benutzt und damit die Dead Code Eigenschaft vernichtet.
Außerdem muß überprüft werden, daß keine "none-dead" MO auf dem Pfad zwischen *VarsIn* und *DefsOut* liegt. Dieser Fall kann auftreten, wenn es eine Schleife vom Basisblockende zum Anfang des gleichen Basisblocks gibt.
Im nachfolgenden Schritt wird für jede Maschinenoperation überprüft, ob es sich um eine Dead Code MO handelt. Wenn dem so ist, wird sie in eine Liste eingefügt. Nach Abschluß der Analyse werden alle MOs dieser Liste gelöscht.
Zu Beginn wird jede Maschinenoperation als Dead Code markiert. Davon ausgenommen werden MOs, die ein entsprechendes "NotDeadCode"-Flag gesetzt haben, oder folgende Eigenschaften erfüllen:

- Die Maschinenoperation gehört zur Klasse der ST, JMP, CJMP, ZJMP, ZLOOP, CALL oder RET Operationen.
- Die Maschinenoperation besitzt einen Nachfolger, der nicht Mitglied der Klasse LirMO::DEFOUT ist.
- Die Maschinenoperation hat einen Nachfolger der Klasse LirMO::DEFOUT und ist Teil des globalen Datenflusses.
- Die Maschinenoperation wird indirekt über einen Pointerzugriff angesprochen.

Alle anderen MOs werden in die Dead Code Liste eingefügt.

5.6 Zusammenfassendes Beispiel

Abschließend ein kleines Beispiel, um das Zusammenspiel der einzelnen Optimierungen zu zeigen. Die Programmdarstellung wurde bewußt in ANSI C

Notation gewählt, da eine Darstellung auf *GeLIR* Ebene schnell unübersichtlich wird und in diesem Fall keinen wesentlichen Mehrwert zum Verständnis der Optimierungen liefert.

1. Constant Folding

   ```
   Original Code:

   n = 50 + 14;
   c = 1023 * 2;
   for (i = 1; i < n; i++)
   {
     r = i;
     a[r-1] = a[r];
     a[r]   = a[r] + c;
   }
   ```

   ```
   Nach Constant Folding:

   n = 64;
   c = 2046;
   for (i = 1; i < n; i++)
   {
     r = i;
     a[r-1] = a[r];
     a[r]   = a[r] + c;
   }
   ```

   Die Addition und die Multiplikation werden beim *Constant Folding* durch den Compiler ausgewertet.

2. Constant Propagation

   ```
   Original Code:

   n = 64;
   c = 2046;
   for (i = 1; i < n; i++)
   {
     r = i;
     a[r-1] = a[r];
     a[r]   = a[r] + c;
   }
   ```

   ```
   Nach Constant Propagation:

   n = 64;
   c = 2046;
   for (i = 1; i < 64; i++}
   {
     r = i;
     a[r-1] = a[r];
     a[r]   = a[r] + 2046;
   }
   ```

   Die durch das *Constant Folding* zusammengefalteten Konstanten `n` und `c` werden durch die *Constant Propagation* Optimierung direkt an ihrer Verwendungstelle eingesetzt. Im Beispiel erkennbar an der Abfrage des Schleifenzählers `i` und am zweiten Summanden der Addition.

3. Copy Propagation

   ```
   Original Code:

   n = 64;
   c = 2046;
   for (i = 1; i < 64; i++)
   {
     r = i;
   ```

   ```
   Nach Copy Propagation:

   n = 64;
   c = 2046;
   for (i = 1; i < 64; i++}
   {
     r = i;
   ```

```
  a[r-1] = a[r];                  a[i-1] = a[i];
  a[r]   = a[r] + 2046;           a[i]   = a[i] + 2046;
}                               }
```

`r` ist eine Kopie von `i`. Da `i` zwischenzeitlich nicht verändert wird, kann der Zugriff des Array `a` auch über `r` erfolgen.

4. Dead Code Elimination

```
Original Code:                  Nach Dead Code Elimination:

n = 64;
c = 2046;
for (i = 1; i < 64; i++)        for (i = 1; i < 64; i++}
{                               {
  r = i;
  a[i-1] = a[i];                  a[i-1] = a[i];
  a[i]   = a[i] + 2046;           a[i]   = a[i] + 2046;
}                               }
```

Durch die *Dead Code Elimination* werden die überflüssigen Zuweisungen an `n`, `c` und `r` entfernt.

Kapitel 6

Low-Level Load/Store Optimierungen

Auf Basis der in Kapitel 4 vorgestellten *Delta-Array-Datenflußanalyse* wurden die beiden Optimierungen *Redundant Load Elimination* und *Redundant Store Elimination* auf *GeLIR* Ebene implementiert.
Dieses Kapitel gibt eine Übersicht dieser Optimierungen, stellt sie mittels Codebeispielen vor und zeigt, wie die *Delta-Array-Datenflußanalyse* die Erkennung und Entfernung redundanter Loads und Stores ermöglicht.

6.1 Einleitendes Beispiel

Aufbauend auf das Beispiel aus Kapitel 5 läßt sich die *Redundant Load Elimination* Optimierung anwenden:

```
Original Code:

for (i = 1; i < 64; i++)
{

  a[i-1] = a[i];
  a[i]   = a[i] + 2046;
}
```

```
Nach Redundant Load Elimination:

for (i = 1; i < 64; i++}
{
  temp   = a[i];
  a[i-1] = temp;
  a[i]   = temp + 2046;
}
```

Im Originalcode muß `a[i]` zweimal geladen werden. Im optimierten Fall nur noch einmal, da der Inhalt von `a[i]` in einem Register `temp` zwischengespeichert wird.

Hier wird bereits der Vorteil der *Redundant Load Elimination* deutlich. Durch das Ersetzen einer redundanten Speicherleseoperation durch eine in der Regel schnellere Registerkopieroperation wird die Ausführungsgeschwindigkeit erhöht.

Allgemein sind Speicherzugriffoperationen redundant, wenn der von ihnen referenzierte Wert bereits verfügbar ist.

Definition 6.1.1 *Eine Lade-Operation LDR r,a[i] an einer Stelle p in einem Programm ist* partiell/total redundant, *wenn entlang einiger/aller Pfade zu p der Wert von a[i] schon in einem Register r' verfügbar ist [BF99].*

Definition 6.1.2 *Eine Schreib-Operation STR r,a[i] an einer Stelle p in einem Programm ist* partiell/total redundant, *wenn entlang einiger/aller Pfade von p aus eine weitere Definition des gleichen Array-Elementes folgt, ohne daß zwischenzeitlich ein Gebrauch stattgefunden hat [BF99].*

6.2 Redundant Load Elimination

Mit Hilfe der *Redundant Load Elimination* lassen sich redundante (Array)-Speicherlesezugriffe innerhalb von Schleifen beseitigen. Dieses Unterkapitel beschreibt das Verfahren der *Redundant Load Elimination* und bietet Informationen über die Einbindung der *Delta-Array-Datenflußanalyse* von [MH01]. Weitergehende Details lassen sich in [BG96], [BF99] oder [RS00] nachlesen.

```
Original Code:


for (i = 1; i < 64; i++)
{
  x = a[i-1];
  ...
  y = a[i];

}
```

```
Nach Redundant Load Elimination:

t = a[0];
for (i = 1; i < 64; i++}
{
  x = t;
  ...
  t = a[i];
  y = t;
}
```

Beispiel 6.2.1 *Redundant Load Elimination*

Im o.g. Programmausschnitt ist ein allgemeines Beispiel für die *Redundant Load Elimination* aufgezeigt. Dort erkennt man, daß neben der Entfernung eines überflüssigen Speicherlesezugriffs auch eine Erweiterung des Schleifenprologs notwendig war.

Der Speicherlesezugriff auf `a[i-1]` innerhalb der Schleife konnte entfernt werden, da auf die gleiche Speicherzelle bereits eine Iteration vorher über `a[i]` zugegriffen worden ist. Es reicht aus, diesen Wert in ein temporäres Register zwischenzuspeichern. Desweiteren muß ein Schleifenprolog eingefügt werden, damit die temporäre Variable bei der ersten Iteration sinnvoll vordefiniert ist, um das Programmverhalten nicht zu verändern. Das liegt darin begründet, da der erste Gebrauch von `t` innerhalb des Schleifenkörpers vor der entsprechenden Definition liegt.

Bei vorhandenem Optimierungspotential kann die *Redundant Load Elimination* die Anzahl der benötigten Taktzyklen für die Programmausführung senken, da eine Speicherleseoperation, wie sie zum Zugriff auf das Array-Element notwendig ist, oft mehr Taktzyklen benötigt als eine `MOV` Operation. Insgesamt ist eine Geschwindigkeitssteigerung durch den Ersatz einer redundanten Speicherleseoperation durch eine schnellere Registerkopieroperation und dem Wegfall der benötigten Adreßberechnung für den Speicherzugriff zu erwarten.
Auf der anderen Seite kann die Programmgröße zunehmen, wenn ein Schleifenprolog eingefügt werden muß. Außerdem wird eine temporäre Variable eingefügt.
Außerdem besteht durch die lange Lebenszeit des eingefügten Registers `t` die Gefahr, daß u.U. Spillcode erzeugt wird.

Anschließend folgt ein Beispiel für die *Redundant Load Elimination* auf Basis der *GeLIR* Datenstruktur:

```
Original Code:                    Nach Redundant Load Elimination:

int a[21];                        int a[21];
int ret;                          int ret;

int main(int argc, char** argv)   int main(int argc, char** argv)
{                                 {
int x,y,z;                        int x,y,z;

  for (z = 0; z < 20; z++)          for (z = 0; z < 20; z++)
  {                                 {
    x = a[z+1];                       x = a[z+1];
                                      t = x;
    x = x + x;                        x = x + x;

    y = a[z+1];                       y = t;

    ret = y;                          ret = y;
  }                                 }

  return ret;                       return ret;
}                                 }
```

Beispiel 6.2.2 *Redundant Load Elimination*

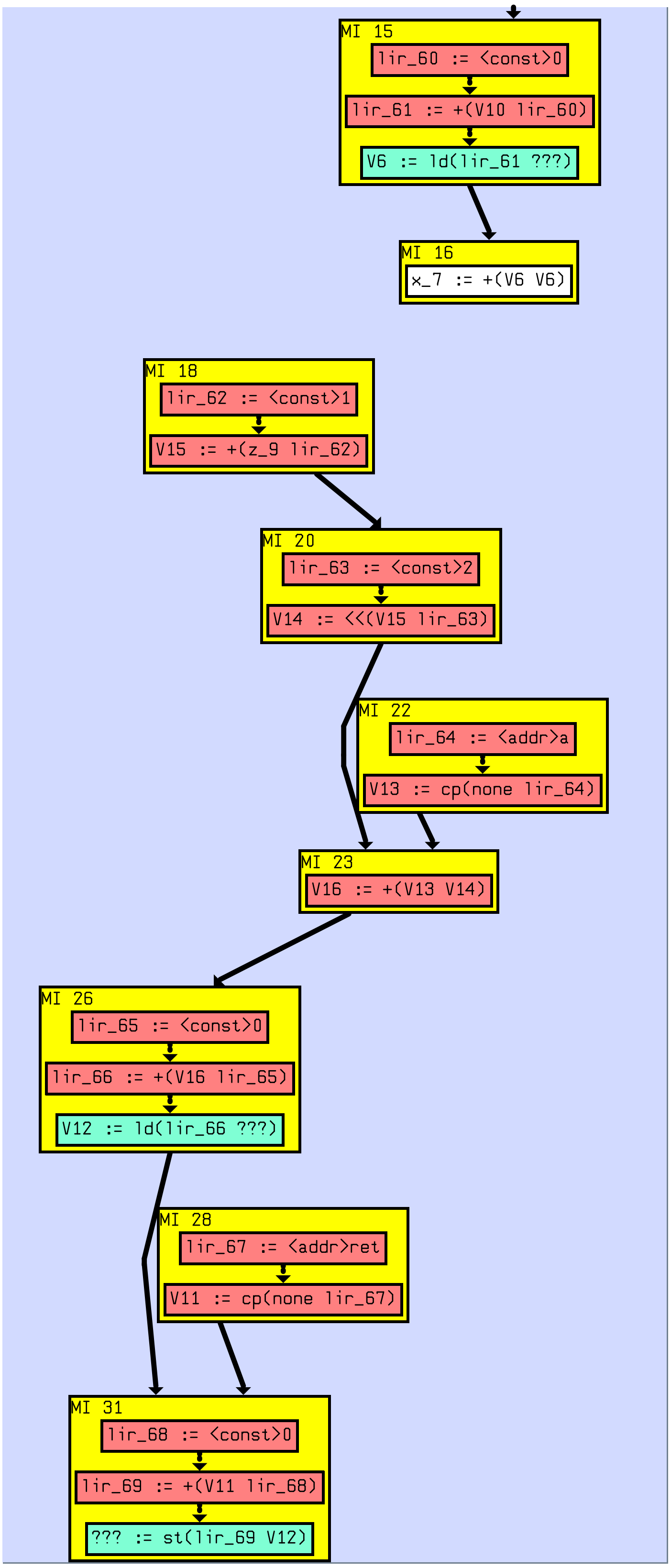

Abbildung 6.1: Codeausschnitt vor RLE Optimierung

Abbildung 6.1 zeigt einen Ausschnitt der *GeLIR* Darstellung des o.g. Beispiels mit Optimierungspotential für die *Redundant Load Elimination*. `MI 15` zeigt den Load Befehl für die Zeile `x = a[z+1]` des C-Programms. Die komplette Adressberechnung ist aus Platzgründen nicht vollständig sichtbar. `MI 18` bis `MI 26` zeigen den erneuten Speicherlesezugriff auf `a[z+1]`. Die entsprechende Zeile des C-Programms lautet `y = a[z+1]`.

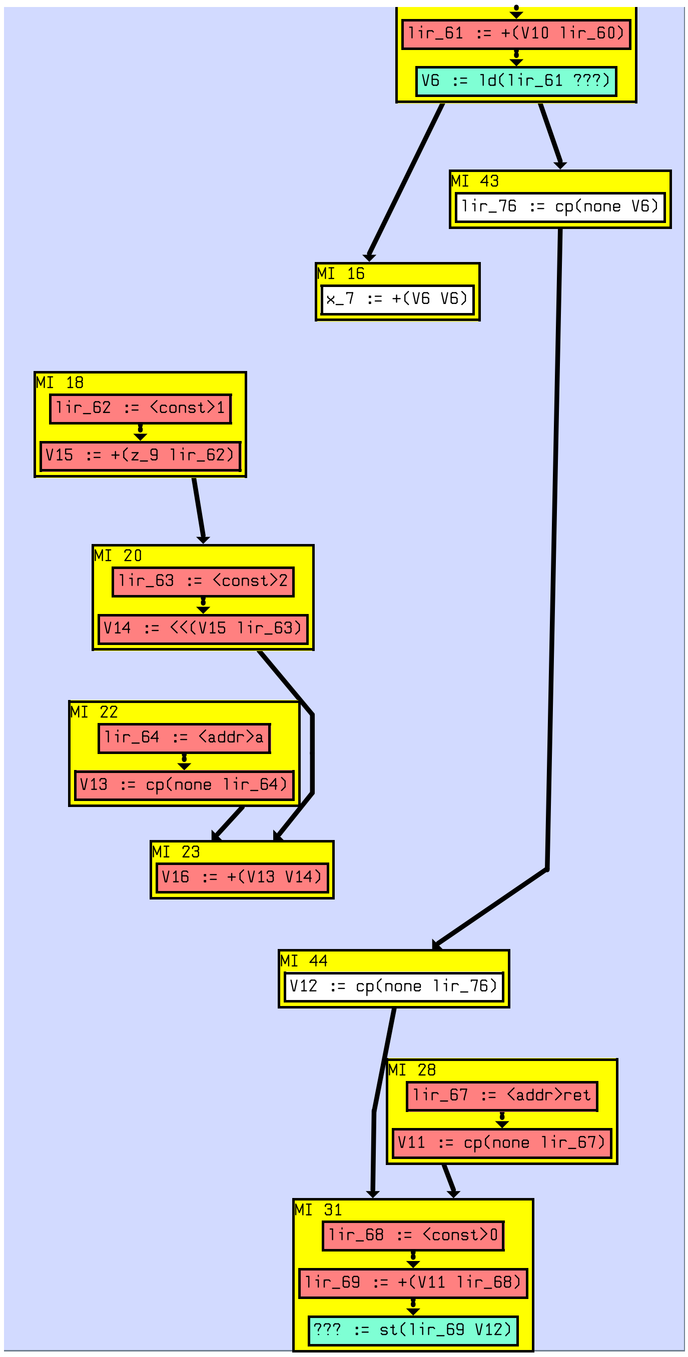

Abbildung 6.2: Codeausschnitt nach RLE Optimierung

In Abbildung 6.2 ist der Codeausschnitt nach der Durchführung der *Redundant Load Elimination* zu sehen. Deutlich zu sehen ist die neu eingefügte Copy-Maschinenoperation in `MI 43`, die den Wert des vorangegangenen Array-Zugriffs zwischenspeichert. Die zweite Speicherleseoperation auf das Array-Element ist durch eine weitere Copy-Maschineninstruktion ersetzt worden, durch die das vorher gesicherte Array-Element nun zurückkopiert wird.
Die vorher für die Adressberechnung des zweiten Array-Zugriffs nötigen Maschineninstruktionen sind nun nicht mehr notwendig und lassen sich durch die *Dead Code Elimination* aus dem Code entfernen.

6.2.1 Architekturspezifisches

Da bei der Durchführung der Optimierung die Speicherzugriffe durch einfache Kopieroperationen ersetzt werden, könnten diese die hohen Register des ARM Thumb Befehlssatzes sinnvoll verwenden und so dazu beitragen, daß der Registerdruck innerhalb der Schleife evtl. sinkt und ein Spilling, also das Aus- und spätere wieder Einlagern des Wertes, vermieden wird. Damit ist die Spillinggefahr gebannt, wenn die Anzahl der redundanten Loads innerhalb einer Schleife kleiner oder gleich der zur Verfügung stehenden hohen Register ist. Liegt kein allgemeines Register mehr zum Speichern des Wertes bis zum erneuten Gebrauch vor, kann dies zum Spilling führen. In diesem Fall sollte auf die Durchführung der *Redundant Load Elimination* verzichtet werden.
Architekturen ohne Datencache profitieren im besonderen Maße von der *Redundant Load Elimination*, da deren in der Regel ausgelasteter Speicherbus durch die Elimination überflüssiger Speicherzugriffe entlastet wird.

6.2.2 Ablauf der Analyse

Die im Rahmen dieser Diplomarbeit implementierte *Redundant Load Elimination* basiert auf der im Kapitel 4 vorgestellten *Delta-Array-Datenflußanalyse* von [MH01].
Redundant Load Elimination berechnet zu allen Knoten die Distanzen, für die ein Wert über mehrere Iterationen verfügbar ist. Dies geschieht iterativ mit der *Delta-Array-Datenflußanalyse*, wobei die Distanzen den Elementen des Datenflußverbands entsprechen. Danach werden redundante Loads mit einer Iterationsdistanz von maximal 1 eliminiert. Für größere Iterationsdistanzen greift man auf das Verfahren *Register Pipelining* [SS00] zurück.

Die *Delta-Array-Datenflußanalyse* wird dabei als *Vorwärts*-Analyse und der zugrundeliegende Datenflußverband als *Must*-Verband parametrisiert.
Die Vorwärtsrichtung bei der Analyse wurde gewählt, da bei erstmaliger Referenzierung eines Array-Elementes dessen Wert für die folgenden Knoten verfügbar ist und bei einer Redefinition die Verfügbarkeitseigenschaft vernichtet wird.

Die Analyse der δ-available values dient der Erkennung der Werte vorangegangener Referenzen, die nach δ Iterationen noch verfügbar sind.

Definition 6.2.1 *Ein Wert x eines Array-Elementes a ist bei einem Knoten n_2* δ-available, *wenn dessen Definition an einem Knoten n_1 erfolgt ist und es entlang aller Pfade von n_1 nach n_2 über δ Iterationen hinweg zu keiner Redefinition von a gekommen ist.*

Da es sich bei δ-available values um ein Must-Problem handelt, wird der Must-Verband als Datenflußverband gewählt.

Nach geeigneter Initialisierung liefert ein Aufruf der *Delta-Array-Datenflußanalyse* eine Matrix zurück. Diese Matrix beinhaltet die Iterationsdistanzen zwischen zwei Speicherzugriffen, die mit Hilfe der entsprechenden *GeLIR* Maschinenoperationen adressiert werden können.
Der Zustand, daß bei der *Redundant Load Elimination* ein Wert seine Gültigkeit verliert, wenn er beschrieben wird, wird in der Matrix mit dem $\perp$-Symbol ausgedrückt. Im Falle des $\perp$-Symbols in der Matrix ist eine *Redundant Load Elimination* für dieses Element nicht möglich.

Nach Aufruf der *Delta-Array-Datenflußanalyse* untersucht die *Redundant Load Elimination* die zurückgegebene Matrix daraufhin, ob eine Referenz r bei einem Knoten n *δ-available* ist. Dazu wird die Matrix in x-Richtung (nX) und y-Richtung (nY) durchlaufen und auf folgende Eigenschaften hin untersucht:

1. $nX \neq nY$, da es sich ansonsten um die gleiche Operation handelt.
2. Iterationsdistanz ≤ 1?
3. Wert des Verbandelements muß $\neq \perp$ sein, da sich sonst ein Schreibzugriff zwischen zwei Leseoperationen befindet.

Sind diese Bedingungen erfüllt, kann mit der Optimierung begonnen werden. Im einfachsten Fall ist die Iterationsdistanz 0, d.h. daß alle Array-Zugriffe auf das gleiche Element verweisen. Dieser Fall wurde im obigen Beispiel gezeigt. Dort wird der erste Speicherlesezugriff in eine temporäre Variable zwischengespeichert und die folgenden Array-Lesezugriffe auf das gleiche Array-Element können durch einen Lesezugriff auf diese temporäre Variable ersetzt werden.

In C als "volatile" definierte Variablen sollten von der *Redundant Load Elimination* ausgenommen werden, um das Programmverhalten nicht zu verändern. So können beispielsweise zwei hintereinanderstehende Loads, die auf die gleiche Speicherzelle verweisen, durchaus unterschiedliche Ergebnisse liefern, wenn sie auf einen als volatile definierten Memory Mapped I/O Bereich zeigen, in dem zwischenzeitlich extern geschrieben worden ist.
Die Maschinenoperationen der *GeLIR* Darstellung besitzen ein `IsVolatile`

Flag, das zu diesem Zweck von der *Redundant Load Elimination* abgefragt werden kann.

Im Fall eines redundanten Loads mit der Iterationsdistanz 1 sind zwei Fälle zu unterscheiden, die über ein Flag der Matrix abgefragt werden können:

1. Die redundante Load-Operation liegt im Schleifenkörper vor der Referenz:

```
Original Code:


for (z = 0; z < 20; z++)
{
  x = a[z];
  ..

  y = a[z-1];
}
```

```
Nach RLE:

t = a[1];
for (z = 0; z < 20; z++)
{
  x = t;
  ..
  t = a[z-1];
  y = t;
}
```

Beispiel 6.2.3 *RLE mit Iterationsdistanz 1 - Fall 1*

2. Die redundante Load-Operation liegt im Schleifenkörper nach der Referenz.

```
Original Code:


for (z = 0; z < 20; z++)
{
  x = a[z-1];


  ..
  y = a[z];

}
```

```
Nach RLE:

t2 = a[1];
for (z = 0; z < 20; z++)
{
  t1 = a[z-1];
  x  = t;
  ..
  y  = t2;
  t2 = t;
}
```

Beispiel 6.2.4 *RLE mit Iterationsdistanz 1 - Fall 2*

Hier wird durch eine zweite temporäre Variable `t2` der Initialwert in die Schleife hineingebracht und in jeder Iteration aktualisiert `t2 = t`.

6.3 Redundant Store Elimination

Die *Redundant Store Elimination* kann redundante (Array)-Schreibzugriffe innerhalb von Schleifen beseitigen. Dieses Unterkapitel beschreibt das Verfahren der *Redundant Store Elimination*, das ebenfalls unter Verwendung der *Delta-Array-Datenflußanalyse* von [MH01] implementiert worden ist. Weitergehende Details lassen wiederum in [BG96], [BF99] oder [RS00] nachlesen.

```
Original Code:                   Nach Redundant Store Elimination:

for (i = 1; i <= 64; i++)        for (i = 1; i <= 64 - 1; i++}
{                                {
  a[i-1] = x;                      a[i-1] = x;
  ...                              ...
  a[i] = y;
}                                }
                                 a[64-1] = x;
                                 ...
                                 a[64] = y;
```

Beispiel 6.3.1 *Redundant Store Elimination*

Obiger Programmausschnitt zeigt ein allgemeines Beispiel vor und nach der Anwendung der *Redundant Store Elimination.* Neben der Entferung des überflüssigen Schreibzugriffs innerhalb der Schleife war das Einfügen eines Schleifenepilogs notwendig, da das *1-redundante Store* `a[i]` in allen Iterationen bis auf die letzte Iteration redundant ist. Dieser Sonderfall der letzten Iteration wird im Schleifenepilog behandelt. Dazu wird der Iterationsbereich der Schleife um eins veringert und der ursprüngliche Schleifenkörper als Epilog der Schleife angefügt, wobei jedes Vorkommen der Induktionsvariable durch den Wert der oberen Iterationsgrenze ersetzt wird.

Im Gegensatz zur *Redundant Load Elimination* müssen hier keine zusätzlichen Register eingefügt werden. Der Vorteil dieser Optimierung ist die Verkleinerung des Schleifenkörpers durch die Elimination redundanter Array- Schreiboperationen.
Nachteilig ist die evtl. Vergrößerung des Codeumfangs durch den eingefügten Epilog. Außerdem können Schleifen mit nicht-affinen Ausdrücken nicht behandelt werden. Diese Einschränkung ist aber bereits durch die *Delta-Array-Datenflußanalyse* gegeben.

Auch bei der *Redundant Store Elimination* sollten in C als "volatile" definierte Variablen ausgenommen werden, um das Programmverhalten nicht zu verändern.
Die Maschinenoperationen der *GeLIR* Darstellung besitzen ein `IsVolatile` Flag, das zu diesem Zweck von der *Redundant Store Elimination* abgefragt werden kann.

Anschließend ein weiteres Beispiel für die *Redundant Store Elimination* auf Basis der *GeLIR* Datenstruktur:

```
Original Code:

int a[20];
int ret;

int main(int argc, char** argv)
{
int x,y,z;

  for (z = 0; z < 20; z++)
  {
    a[z] = x;

    y = z * 2;

    a[z] = y;

    ret = z;
  }

  return ret;
}
```

```
Nach Redundant Store Elimination:

int a[20];
int ret;

int main(int argc, char** argv)
{
int x,y,z;

  for (z = 0; z < 20; z++)
  {


    y = z * 2;

    a[z] = y;

    ret = z;
  }

  return ret;
}
```

Beispiel 6.3.2 *Redundant Store Elimination*

Abbildung 6.3 zeigt einen Ausschnitt der *GeLIR* Darstellung des o.g. Beispiels mit Optimierungspotential für die *Redundant Store Elimination.* Der dargestellte Basisblock zeigt den Schleifenkörper des o.g. Beispielcodes. `MI 11` und `MI 18` sind darin die beiden Speicherzugriffe auf `a[z]`. Der erste Speicherzugriff, also `MI 11`, ist redundant, da er innerhalb der Schleife unmittelbar durch einen anderen Wert überschrieben und zwischenzeitlich nicht verändert wird. Aus der Grafik ist nicht direkt ersichtlich, daß der erste Store Befehl redundant ist. Ein Durchlauf der *Delta-Array-Datenflußanalyse* ermittelt, daß die beiden temporärem Variablen `lir_51` und `lir_54` auf den gleichen Speicherbereich zeigen.

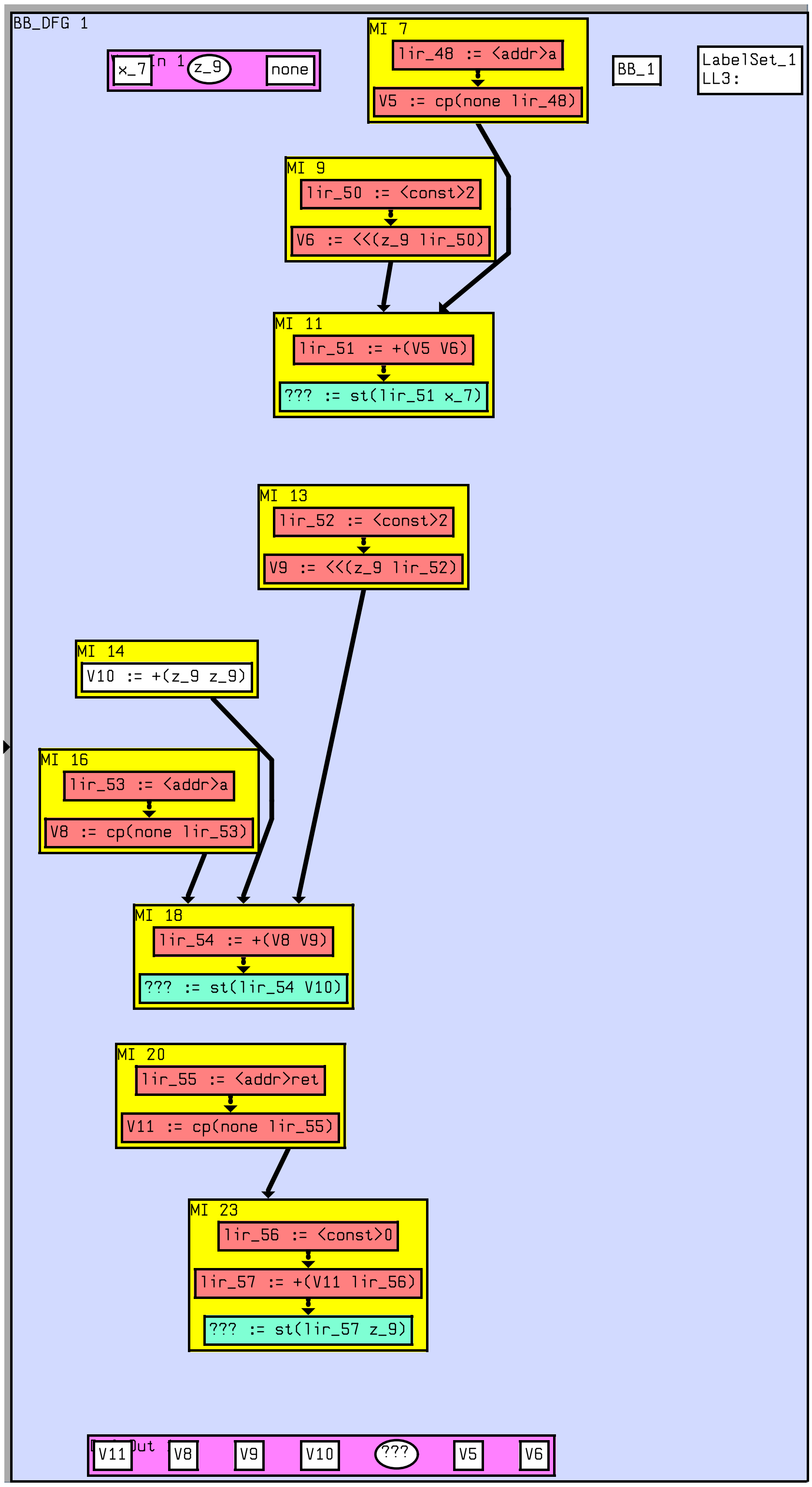

Abbildung 6.3: Codeausschnitt vor RSE Optimierung

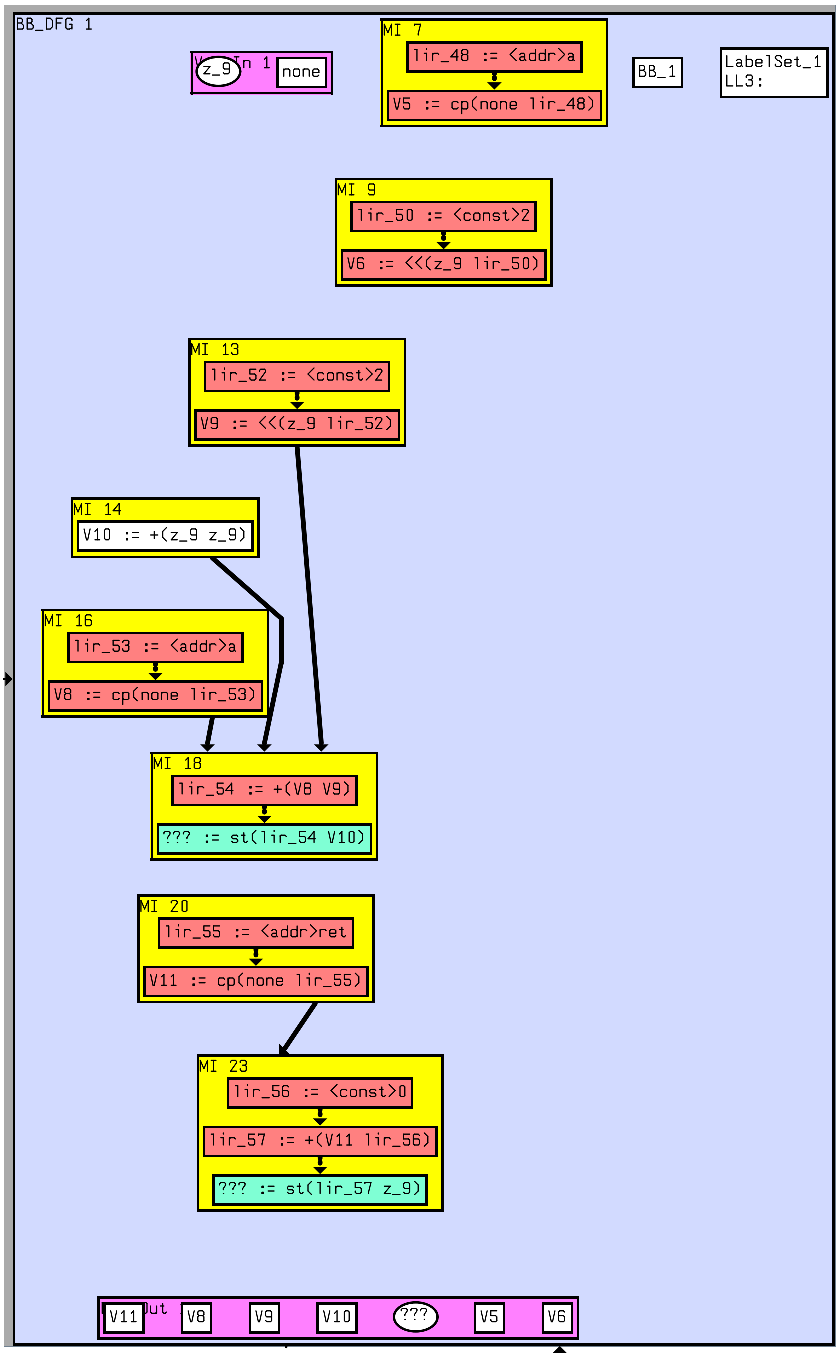

Abbildung 6.4: Codeausschnitt nach RSE Optimierung

Abbildung 6.4 zeigt den Basisblock nach der Durchführung der *Redundant Store Elimination*. Der erste, redundante Speicherzugriff `MI 11` wurde entfernt. Die für die Adressberechnung des Array-Zugriffs nötigen Maschineninstruktionen sind nicht mehr notwendig und stehen als überflüssiger Code (`MI 7` und `MI 9`) im Basisblock. Sie lassen sich durch einen *Dead Code Elimination* Durchlauf aus dem Code entfernen.

6.3.1 Architekturspezifisches

An architekturspezifischen Besonderheiten könnten beispielsweise bei der LEON CPU dessen Cache Register und die Memory Mapped I/O Bereiche durch eine Analyse über den Speicher berücksichtigt werden.
Desweiteren kann die Größe des Instruktionscache des LEON berücksichtigt werden. Wird der bei der *Redundant Store Elimination* einzufügende Schleifenepilog zu groß, kann es passieren, daß die Schleife vor der Optimierung evtl. schneller ausgeführt werden konnte, wenn sie sich komplett im Instruktionscache befand. Ein zu großer Schleifenepilog kann aber verhindern, daß die Routine nach wie vor in den Instruktionscache passt. In diesem Fall kann z.B. der Schleifenepilog selber in eine Schleife eingebettet werden, um den Programmcode zu verkürzen.

6.3.2 Ablauf der Analyse

Zur Beseitigung von redundanten Schreibzugriffen auf Array-Elementen innerhalb von Schleifen wird mit einer geeignet parametrisierten *Delta-Array-Datenflußanalyse* ermittelt, welche Definitionen bis zu welchem Knoten ohne anschließenden Gebrauch erfolgten. Mit einer *Delta-Busy-Analyse* lassen sich die Definitionen ohne anschließenden Gebrauch finden.

Definition 6.3.1 *Eine Store-Operation s in einem Knoten n_i ist δ-*busy*, wenn sie entlang aller Kontrollflußpfade zu einem Knoten n_j ausgeführt wird, ohne daß das gespeicherte Array-Element entlang dieser Pfade über δ Iterationen hinweg verwendet wird.*

Definition 6.3.2 *Eine Store-Operation s_i in einem Knoten n_k ist genau dann* δ-redundant*, wenn es eine weitere Store-Operation s_j in einem Knoten n_l im Schleifenkörper gibt und s_i im Knoten n_l* δ-busy *ist.*

Zur Parametrisierung der *Delta-Array-Datenflußanalyse* wird die Rückwärts-Arbeitsrichtung gewählt, da die Suche von späteren Stores hin zu früheren führt. Dazu wird auf einem umgekehrten Kontrollflußgraph gearbeitet, dessen Kantenrichtungen umgedreht sind.
Bei der Bestimmung, ob ein Knoten *delta-busy* ist, handelt es sich um ein Must-Problem, so daß der Must-Verband für die Parametrisierung der *Delta-Array-Datenflußanalyse* eingesetzt werden muß.

Bei der *Redundant Store Elimination* darf zwischen zwei Werten kein lesender Zugriff erfolgen. Deswegen werden alle Distanzen der lesenden Speicherzugriffe mit dem $\perp$-Symbol initialisiert. Im Falle des $\perp$-Symbols ist eine *Redundant Store Elimination* nicht möglich.
Durch eine Definition wird die zu untersuchende Eigenschaft *δ-busy* während der Analyse erzeugt. Ein Gebrauch vernichtet diese Eigenschaft.

Nach Aufruf der *Delta-Array-Datenflußanalyse* untersucht die *Redundant Store Elimination* die zurückgegebene Matrix analog zum Vorgehen bei der *Redundant Load Elimination*. Dazu wird die Matrix ebenfalls in x-Richtung (nX) und y-Richtung (nY) durchlaufen und auf folgende Eigenschaften hin untersucht:

1. $nX \neq nY$, da es sich ansonsten um die gleiche Operation handelt.
2. Iterationsdistanz ≤ 1?
3. Wert des Verbandelements muß $\neq \perp$ sein, da sich sonst ein Schreibzugriff zwischen zwei Leseoperationen befindet.

Sind diese Bedingungen erfüllt, kann mit der Optimierung begonnen werden. Wenn eine δ-*redundante* Operation erkannt wurde, wird die entsprechende Maschinenoperation innerhalb des Schleifenkörpers aus der *GeLIR* Darstellung entfernt. Für eine Iterationsdistanz von $\delta = 0$ wäre die Optimierung damit beendet.
Andernfalls wird die Iterationsgrenze der Schleife um δ vermindert und ein Schleifenepilog mit δ-facher Aneinanderreihung des Schleifenkörpers aufgebaut, wobei jedes Vorkommen der Induktionsvariable innerhalb des Schleifenkörpers geeignet ersetzt wird. In der ersten Kopie des Schleifenkörpers wird die Induktionsvariable durch $Iterationsgrenze - \delta$ ersetzt. Dann durch $Iterationsgrenze - \delta + 1$ bis schließlich in der letzten Kopie der Wert der $Iterationsgrenze$ eingesetzt wird. Es ist zu beachten, daß der redundante Store in Schleifenepilog erhalten bleiben muß.
Beinhaltet eine Schleife mehrere redundante Store-Operationen mit unterschiedlichen Iterationsdistanzen, so wird der größte δ-Wert als Optimierungsgrundlage verwendet.
Bei einem großen δ-Wert bzw. einem umfangreichen Schleifenkörper ist es sinnvoll, eine Epilog-Schleife zu erzeugen und auf die Aneinanderreihung der Schleifenkörperkopien zu verzichten.
Diese Eigenschaft wird von der momentanen Implementierung nicht unterstützt.

Kapitel 7

Bewertung der Optimierungsergebnisse

In diesem Kapitel werden die Ergebnisse der im Rahmen dieser Diplomarbeit implementierten Optimierungen vorgestellt und mit bereits vorhandenen alternativen Optimierungswegen verglichen.
Außerdem werden die zur Validierung des Konverters `CFG2GeLIR` und der Optimierungen selbst notwendigen Schritte und Hilfsprogramme erläutert.

7.1 Vorgehensweise

Die Bestimmung der Ergebnisse erfolgte durch die Ausführung von Optimierungen auf verschiedenen Ebenen: High-Level Optimierungen, reine Backend Optimierungen sowie *GeLIR* Optimierungen. Diese Optimierungen wurden auf eine Reihe von Benchmark- und Testprogrammen angewendet.
Die Überprüfung der semantischen Korrektheit erfolgte mit Hilfe des *GeLIR Simulators*. Werte über die Ausführungsgeschwindigkeit des mit Hilfe von *XeLIR* generierten optimierten Assemblercodes und des Energieverbauchs lieferte letztendlich der *ARMulator* in Verbindung mit dem *encc Energy Profiler*.

7.1.1 Überblick

Insgesamt stehen drei mögliche Optimierungsebenen zur Verfügung.

1. Optimierungen auf der High-Level IR von *LANCE2*
2. Optimierungen im Backend des *encc* Compilers
3. Optimierungen auf der Low-Level IR *GeLIR*

Zum Testen wurden jeweils Kombinationen der drei vorgestellten Optimierungsklassen aktiviert und die Optimierungsergebnisse verglichen. Abbildung 7.1 zeigt den Einsatzort der Optimierungsklassen im Arbeitsablauf.

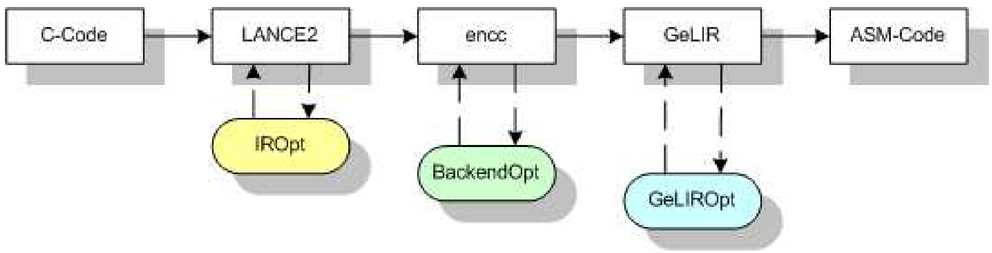

Abbildung 7.1: Übersicht der drei möglichen Optimierungsebenen

Beim Vergleich der "Optimierungspakete" sollte beachtet werden, daß es Unterschiede zwischen den einzelnen Optimierungen gibt. Die *LANCE2* Optimierungen werden seit Jahren gepflegt und intensiv getestet, während die *GeLIR* Optimierungen naturgemäß noch recht jung sind. Die *encc* Backend Optimierungen entstanden teilweise "automatisch" aus den verwendeten Algorithmen. Einige Verbesserungen sind jedoch auch auf zusätzliche Regeln in der Olive-Grammatik (siehe Abschnitt 3.1.2) zurückzuführen. Folgende Optimierungen werden im einzelnen angewendet:

- ***LANCE2* Frontend**
  - Constant Folding
  - Constant Propagation
  - Copy Propagation
  - Dead Code Elimination
  - Common Subexpression Elimination
  - Jump Optimization
- ***encc* Backend**
  - Constant Folding

    Constant Folding wird vom *encc* Compiler während der Code Selektion durchgeführt.
  - Constant Propagation

    Constant Propagation wird bereits zu Beginn bei der Generierung der Datenflußbäume von *encc* durchgeführt.
  - Copy Propagation

    Während der Register Allokation im Rahmen des Coalescing wird die *Copy Propagation* durchgeführt.

- ***GeLIR* Ebene**
  - Constant Folding
  - Constant Propagation
  - Copy Propagation
  - Dead Code Elimination
  - Redundant Load Elimination
  - Redundant Store Elimination

Zum Erzeugen der Ergebnisse werden fünf Kombinationen aus den drei möglichen Optimierungsebenen gebildet. Zuerst sind alle Optimierungen deaktiviert. Dann folgen drei Durchläufe, in denen je eine der drei möglichen Optimierungsebenen eingeschaltet ist. Der fünfte Durchlauf erfolgt mit eingeschalteten Backend– und *LANCE2*-Optimierungen, da dies bisher als Standardweg zur Codegenerierung mit dem *encc* Compiler eingesetzt wurde.
Die *GeLIR* Optimierungen lassen sich komfortabel durch eine Konfigurationsdatei ein- und ausschalten. Um die Backend Optimierungen zu deaktivieren, müssen die entsprechenden Olive-Regeln, sowie Optimierungsaufrufe auskommentiert werden. Die Durchführung der *LANCE2* Optimierungen läßt sich im Aufrufskript `genasm` ausschalten.

7.1.2 Benchmarkprogramme

Folgende Beispielprogramme kamen zum Einsatz, um die Ergebnisse der neu implementierten Low-Level Optimierungen zu ermitteln:

- Bubble Sort Algorithmus
- Heap Sort Algorithmus
- Insertion Sort Algorithmus
- Selection Sort Algorithmus
- Dot Product aus der DSP-Stone Benchmark Suite [DSP]

7.2 Validierung der Optimierungen

Die Validierung der Optimierungsergebnisse lief zweistufig ab. Dazu kamen die beiden Tools *GeLIR Simulator* und *ARMulator* zum Einsatz. Der *GeLIR Simulator* arbeitet auf der abstrakten Programmdarstellung der *GeLIR*.
Der *ARMulator* führt den generierten Assemblercode auf einer simulierten Architekturumgebung aus und generiert die Simulationsergebnisse.

7.2.1 GeLIR Simulator

Mit Hilfe des in Kapitel 3.5 beschriebenen *GeLIR Simulators* läßt sich die semantische Äquivalenz des Programms vor und nach der Optimierung überprüfen und die Ausführungsgeschwindigkeit als ungefähre Anzahl der Zyklen ermitteln.

Die semantische Korrektheit wird durch die Ausgabe des Simulators validiert. Dazu muß das zu überprüfende Benchmarkprogramm leicht modifiziert werden. Gegeben sei folgendes, kurzes Beispielprogramm:

```
int ret;
int a[4];

int main(int argc, char** argv)
{
  a[1] = 11;
  a[2] = 9;
  a[3] = a[1] + a[2] * 4;

  ret = a[3];

  return ret;
}
```

Für den *GeLIR Simulator* sind folgende Anpassungen vorzunehmen:

```
#ifdef GELIRAUTOGEN
#include "autogen.h"
#endif

int ret;
int a[4];

int main(int argc, char** argv)
{
  a[1] = 11;
  a[2] = 9;
  a[3] = a[1] + a[2] * 4;

  ret = a[3];

#ifdef GELIRAUTOGEN
  string filename;
  filename = string(argv[0]) + string(".gnu.log");

  CAutoGen out(filename);
  out.Output("ret", ret);
  out.OutputArray("a", a, 0, 3);
  out.Close();

  filename = string(argv[0]) + string(".gnuret.log");
  out.Open(filename);
  out.Output("ret", ret);
  out.Close();
#endif

  return ret;
}
```

Durch Verwendung von Compilerdirektiven und bedingter Übersetzung ist sichergestellt, daß sich die Semantik durch diese Modifikation nicht ändert. Es wird lediglich Hilfscode zum Erzeugen der für den Simulator notwendigen Log-Files zum Vergleich der Ergebnisse hinzugefügt. Mit Hilfe der `out` Klasse lassen sich beliebige Programmvariablen beobachten. So werden beispielweise in unserem Beispiel der Rückgabewert `ret` und das Array `a` ausgegeben.

7.2.2 ARMulator

Mit Hilfe des *ARMulators* wird der wie in Kapitel 3.5.1 beschrieben erzeugte Assemblercode unter realen Bedingungen auf einem "virtuellen" ARM7TDMI mit 33MHz simuliert.
Als Ergebnis erzeugt der *ARMulator* ein Trace-File, das der *Energy Profiler* des *encc* Compilers als Eingabe verwenden kann, um daraus einen Report zu erzeugen. Dieser Report liefert als Ergebnis die Anzahl der benötigten Zyklen für die Programmabarbeitung, die Programmgröße und Werte über den Energiebedarf.

Der aus der XML Darstellung erzeugte Assemblercode ist reiner Assemblercode. Dieser muß noch leicht modifiziert werden, damit er vom ARM Assembler übersetzt werden kann. Dazu werden assemblerspezifische Symbole für den Assembler und Linker hinzugefügt.

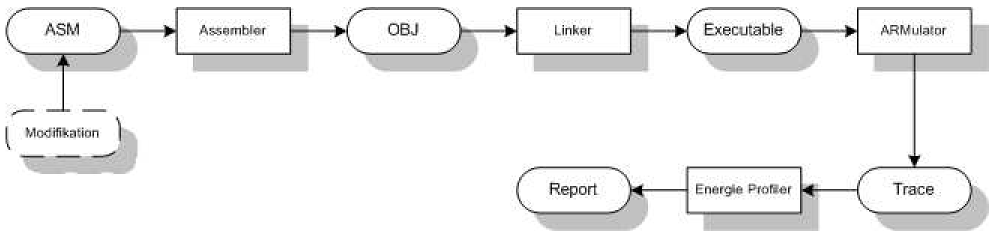

Abbildung 7.2: Ablauf zur Erzeugung von Geschwindigkeits- und Energiewerten

Abbildung 7.2 zeigt die notwendigen Arbeitsschritte. Der von *XelirPattern* (siehe Abschnitt 3.5.1) erzeugte Assemblercode wird nach den nötigen Modifikationen durch den ARM Assembler assembliert und schließlich zu einem ausführbaren Programm gelinkt, aus dem der *ARMulator* das Trace-File für den *Energy Profiler* erzeugt. Aus dem resultierenden Report des Profilers wurden die Ergebnisse des nachfolgenden Abschnitts bestimmt.

7.3 Ergebnisse

Die folgenden Tabellen und Diagramme zeigen die Ergebnisse der möglichen Optimierungswege für die durchgeführten Benchmarkprogramme. Zum Vergleich sind die Ergebnisse bei einem Durchlauf ohne Optimierungen mit angegeben.

Benchmark	unoptimiert	LANCE2	encc	LANCE2 & encc	GeLIR
Bubble Sort	488708	429211	428214	358213	428414
Heap Sort	111350	118216	99633	84378	86469
Insertion Sort	285900	296601	254996	254297	255000
Selection Sort	198190	205215	180466	178379	181060
Dot Product	14766	12358	11162	10754	13564

Tabelle 7.1: Änderung der Taktzyklen bei den Optimierungsdurchläufen

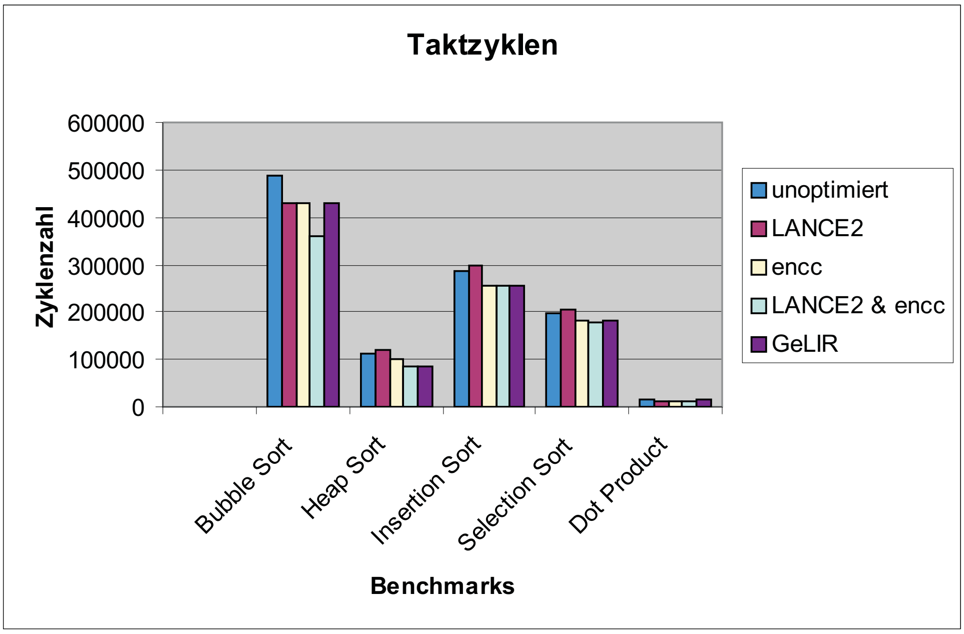

Abbildung 7.3: Übersicht der benötigten Zyklen

Auffallend ist, daß die *LANCE2* Optimierungen für sich betrachtet wenig Taktzyklen einsparen und teilweise sogar mehr Zyklen benötigen als der nicht optimierte Code. In Verbindung mit den Optimierungen im Backend wird hingegen die beste Laufzeit erreicht. Dies zeigt, daß High-Level Optimierungen immer im Zusammenhang mit Optimierungen in späteren Phasen gesehen werden sollten, und in diesem Bereich sinnvolle Vorarbeit leisten.
Die *encc* und *GeLIR* Optimierungen liefern relativ ähnliche Resultate. Bis auf das Heap Sort Beispiel sind die *encc* Ergebnisse etwas besser.

Benchmark	unoptimiert	LANCE2	encc	LANCE2 & encc	GeLIR
Bubble Sort	7388.766	6511.827	6597.033	5547.408	6600.321
Heap Sort	1655.007	1779.504	1493.422	1266.755	1294.544
Insertion Sort	4304.030	4480.201	3899.876	3889.388	3899.941
Selection Sort	2909.279	3029.849	2674.898	2643.615	2684.585
Dot Product	222.755	187.229	169.572	162.878	200.780

Tabelle 7.2: Änderung der Energie (10ˆ6 J) bei den Optimierungsdurchläufen

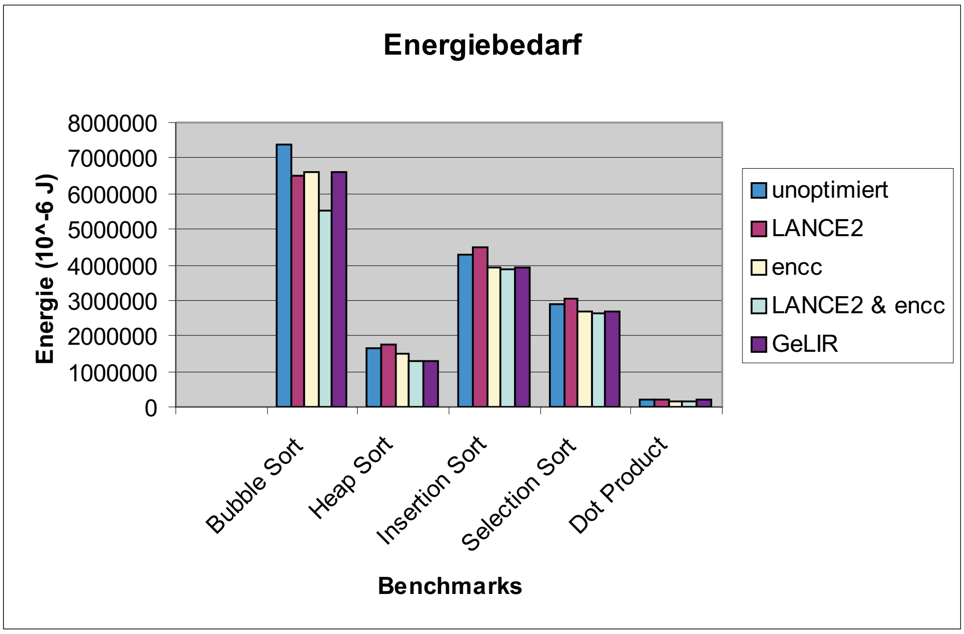

Abbildung 7.4: Übersicht des Energieverbrauchs

Da der Energieverbrauch linear mit der Zeit gekoppelt ist, ergeben sich hier sehr ähnliche Verhältnisse wie bei der reinen Ausführungszeit. Die mit den *GeLIR* Optimierungen erreichten Ergebnisse reichen zwar nicht an die gemeinsame Anwendung von High-Level- und Low-Level-Optimierungen heran, erreichen aber dennoch eine deutliche Effizienzsteigerung des Codes.

Benchmark	unoptimiert	LANCE2	encc	LANCE2 & encc	GeLIR
Bubble Sort	132889	128134	112691	102688	112791
Heap Sort	33258	33719	28932	25823	25408
Insertion Sort	82905	88249	72602	72500	72604
Selection Sort	59977	64331	53937	53636	54234
Dot Product	4217	3213	2715	2511	3516

Tabelle 7.3: Änderung der Codegröße (Anzahl der Instruktionen) bei den Optimierungsdurchläufen

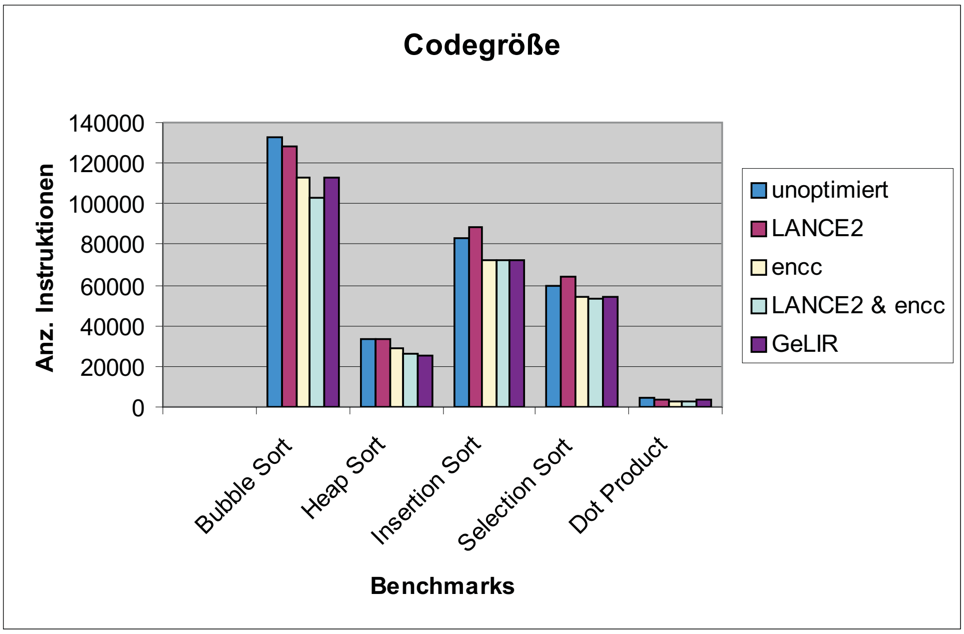

Abbildung 7.5: Übersicht der Codegröße

In Bezug auf die Codegröße, wichtige Kenngröße für mobile Systeme mit (teurem) Speicher, ergibt sich ein ähnliches Bild: Die *GeLIR* Optimierungen kommen durchaus in den Bereich, der auch mit reinen Backend-Optimierungen erreicht wird.

Im folgenden werden die durch die Ergebniswerte ausgedrückten Veränderungen der Optimierungen bzgl. der benötigten Zyklen, der Anzahl der Instruktionen und des Energiebedarfs, prozentual dargestellt.

Benchmark	LANCE2	encc	LANCE2 & encc	GeLIR
Bubble Sort	12.17%	12.38%	26.70%	12.34%
Heap Sort	-6.17%	10.52%	24.22%	22.34%
Insertion Sort	-3.74%	10.81%	11.05%	10.81%
Selection Sort	-3.54%	8.94%	10.00%	8.64%
Dot Product	16.31%	24.41%	27.17%	8.14%
Durchschnitt	3.01%	13.41%	19.83%	12.45%

Tabelle 7.4: Prozentuale Verbesserung der Zyklenzahl

Benchmark	LANCE2	encc	LANCE2 & encc	GeLIR
Bubble Sort	3.58%	15.20%	22.73%	15.12%
Heap Sort	-1.39%	13.01%	22.36%	23.60%
Insertion Sort	-6.45%	12.43%	12.55%	12.43%
Selection Sort	-7.26%	10.07%	10.57%	9.58%
Dot Product	23.81%	35.62%	40.46%	16.62%
Durchschnitt	2.46%	17.26%	21.73%	15.47%

Tabelle 7.5: Prozentuale Verbesserung der Codegröße

Benchmark	LANCE2	encc	LANCE2 & encc	GeLIR
Bubble Sort	11.87%	10.72%	24.92%	10.67%
Heap Sort	-7.52%	9.76%	23.46%	21.78%
Insertion Sort	-4.09%	9.39%	9.63%	9.39%
Selection Sort	-4.14%	8.06%	9.13%	7.72%
Dot Product	15.95%	23.88%	26.88%	9.87%
Durchschnitt	2.41%	12.36%	18.81%	11.89%

Tabelle 7.6: Prozentuale Verbesserung des Energiebedarfs

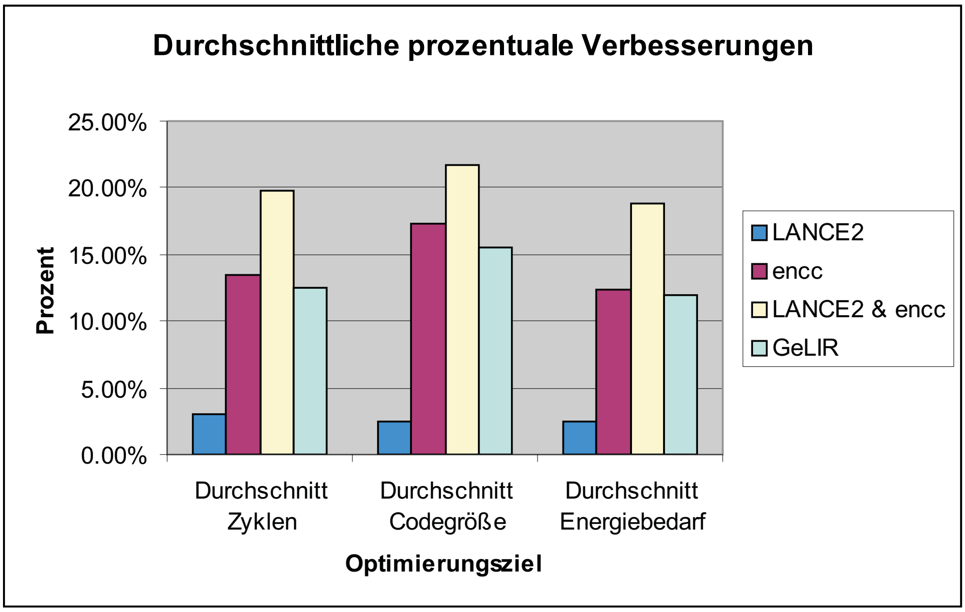

Abbildung 7.6: Prozentuale Verbesserungen

Insgesamt konnte die Codequalität durch die *GeLIR* Optimierungen im Durchschnitt um 11.89% verbessert werden und liegt damit fast gleichauf mit den *encc* Optimierungen, die den Code um 12.36% optimieren. Die *LANCE2* Optimierungen bringen für sich betrachetet nur einen Gewinn von 2.41%, erreichen aber in Verbindung mit den Backend-Optimierungen des *encc* Compilers mit 18.81% das beste Resultat.

7.4 Bewertung

In dieser Arbeit wurde gezeigt, daß Optimierungen auf Basis der *GeLIR* Datenstrukturen durchaus einen Vorteil bieten: Zum einen besteht die Möglichkeit, auf der abstrakten Ebene High-Level-Informationen zu sammeln und Optimierungen, die z.B. Datenflußanalysen voraussetzen, mit relativ wenig Aufwand zu implementieren. Andererseits können zusätzlich dazu architekturabhängige Informationen, die in der Target-Beschreibung der *GeLIR* Datenstruktur abgelegt sind, berücksichtigt und ausgenutzt werden. Wurden Code Selektion und Register Allokation bereits durchgeführt, so können die dadurch gewonnenen Erkenntnisse über Ressourcen-Zuordnungen ebenfalls in die Optimierungen einfließen.

Bei Betrachtung der o.g. Ergebnisse der Optimierungen wird deutlich, daß die Kombination aus *LANCE2* und *encc* Backend Optimierungen die besten Resultate liefert. Dies ist nicht weiter verwunderlich, da die *LANCE2* Optimierungen in jahrelanger Forschungs- und Entwicklungsarbeit gepflegt wurden und auch in kommerziellen Projekten Verwendung finden. Die Backend-Optimierungen des *encc* wurden im Laufe der Zeit immer wieder erweitert und an die Bedürfnisse angepasst. Sie gehen in hohem Maße auf die spezielle Struktur des von *LANCE2* generierten Codes ein. Dadurch ist die erzielte Codequalität verglichen mit den neu eingeführten *GeLIR* Optimierungen höher.

Für die Konzeption und Implementierung der *GeLIR* Optimierungen stand ein begrenzter Zeitrahmen zur Verfügung, wobei ein Großteil der Entwicklungsarbeit für die exemplarische Anbindung der *GeLIR* Datenstrukturen an den vom *encc* Compiler generierten Kontrollflußgraphen investiert werden musste.

Die ebenfalls beispielhaft existierende Anbindung des *LANCE2* Frontends an die *GeLIR* verspricht einen einfacheren Einstieg in die Verwendung der neuen LIR. Derzeit existieren weder eine Code Selektion noch Register Allokation für den ARM Prozessor auf Basis der *GeLIR*. Somit gibt es keinen objektiven Weg, die Güte von Low Level Optimierungen zu bewerten, da hierfür in der Regel ein vollständiges Backend benötigt wird, das ausführbaren Assemblercode generieren kann. Aus diesem Grund mußten die notwendigen Informationen zur Codegenerierung aus der *encc* Darstellung übernommen werden.

Es ist davon auszugehen, daß mit Hilfe der *GeLIR* Darstellung weiteres Optimierungspotential ausgeschöpft werden kann. Dafür notwendige Grundlagen wurden in dieser Diplomarbeit dargelegt. Die implementierten und in diesem Dokument vorgestellten Optimierungen stellen einen sinnvollen Grundstock an Optimierungstechniken auf Basis der *GeLIR* Datenstruktur dar. Nach Bedarf lassen sich zukünftig aufbauend auf diesen Optimierungen weitere Algorithmen hinzufügen.

Kapitel 8

Zusammenfassung und Ausblick

Dieses Kapitel fasst die Ergebnisse dieser Diplomarbeit kurz zusammen und erörtert mögliche Erweiterungen und Änderungen, die im Rahmen zukünftiger Forschungsarbeiten aufgenommen werden können.

8.1 Zusammenfassung

Diese Diplomarbeit ist die erste, die sich mit der Implementierung von Low-Level Optimierungen auf der *GeLIR* Datenstruktur befasst. Ein wichtiger Grundstock an immer wieder benötigten Standardoptimierungen wurde implementiert und mit bewährten Optimierungen verglichen.
Es hat sich gezeigt, daß auch komplexere Optimierungen auf Basis einer umfangreichen Datenflußanalyse mit vertretbarem Aufwand für die *GeLIR* entwickelt werden können. Dazu wurden beispielhaft die Optimierungen *Redundant Load Elimination* sowie *Redundant Store Elimination* herangezogen.

Um die Low-Level Optimierungen unter "realen" Bedingungen zu testen, wurde eine Anbindung an den an der Universität Dortmund am Fachbereich Informatik, Lehrstuhl 12 entwickelten *encc* Compiler an die *GeLIR* Umgebung realisiert. So war ein direkter Vergleich zwischen den bisher eingesetzten High-End Optimierungen im *LANCE2* Frontend und den teilweise im Compiler Backend enthaltenen Optimierungen mit den neu entwickelten *GeLIR* Optimierungen möglich.

Die Beschreibung der Arbeitsumgebung, insbesondere die Vorstellung der *GeLIR* nimmt einen großen Stellenwert in dieser Diplomarbeit ein, da sie zum ersten Mal Grundlage einer Diplomarbeit war. Die *GeLIR* Datenstruktur ist noch recht jung, was zum einen problematisch war, da sie sich in ständiger

Entwicklung befand und aktuelle Änderungen während der Entstehung der Diplomarbeit berücksichtigt werden mussten. Auf der anderen Seite war es dank der *GeLIR*-Funktionalität möglich, die Optimierungen generisch zu entwickeln, so daß die Optimierungen nicht auf eine Prozessorarchitektur beschränkt sind. Gleichzeitig können jedoch auch architekturabhängige Gesichtspunkte, die sich positiv auf den Ablauf der Optimierungen auswirken können, berücksichtigt werden.

Es mag im ersten Moment nicht sinnvoll erscheinen, bekannte Standardoptimierungen, die zumindest teilweise bereits im verwendeten Compiler vorhanden sind, erneut zu implementieren. Es sollte allerdings berücksichtigt werden, daß die *GeLIR* zukünftig auch an andere Compiler angebunden werden kann, die diese Optimierungen u.U. noch nicht beinhalten. In dem Fall kann auf jeden Fall von der Wiederverwendbarkeit der *GeLIR* Optimierungen profitiert werden. Ein weiterer Aspekt, der auch bereits in dieser Arbeit angesprochen wurde, ist der Bedarf an erneuten Optimierungsdurchläufen, z.B. der *Dead Code Elimination*, nach anderen Optimierungen.

Desweiteren stehen den Optimierungen auf unterster Ebene durch die Code Selektion und Register Allokation gewonnene Informationen zur Verfügung, die auf höherer Ebene nicht vorhanden sind. Diese Details können bei der Entwicklung der Optimierungen berücksichtigt werden.

Die durchgeführten Messungen haben gezeigt, daß sich auf Basis der *GeLIR* Datenstruktur implementierte Optimierungen mit den bisher verwendeten und seit Jahren gepflegten Optimierungen vergleichen lassen. Bestehende Optimierungen werden nicht übertroffen, aber es konnte klar gemacht werden, daß mit Hife der *GeLIR* weiteres Optimierungspotential ausgeschöpft werden kann. Der dazu notwendige Grundstock an gängigen Low-Level Optimierungen wurde für die *GeLIR* umgesetzt und läßt sich zukünftig bei Bedarf erweitern und verbessern.

8.2 Mögliche Erweiterungen

Diese Diplomarbeit kann als Ausgangspunkt für weitere Untersuchungen genutzt werden. Dazu ergibt sich an mehreren Stellen der Bedarf an möglichen Erweiterungen. Folgende Punkte sollen als Anregungen dienen, wo und warum bestimmte Ansätze dieser Diplomarbeit erweitert werden können.

8.2.1 Der Konverter CFG2GeLIR

In seiner jetzigen Implementierung ist der Konverter auf ARM Architekturen im Thumb Modus spezialisiert. Da der zugrundeliegende *encc* Compiler momentan erweitert wird, um auch effizienten Code für LEON CPUs zu erzeugen,

könnte auch der Konverter dahingehend modifiziert werden.
Zur Unterstützung des LEON müsste zum einen die Architekturbeschreibung des LEON analog zur Vorgehensweise in Anhang A eingegeben werden und zum anderen die eigentliche Konverterklasse `CFG2GeLIR` modifiziert werden.
Dazu sind hauptsächlich die Programmstellen anzupassen, an denen die Ressourcen der Zielarchitektur in die *GeLIR* Darstellung eingefügt werden. Desweiteren müssen die möglichen Typen und Adressierungsarten der einzelnen Assemblerbefehle berücksichtigt werden.
Architekturabhängige Optimierungen für den LEON könnten z.B. dessen Branch Delay Slot ausnutzen.

8.2.2 Die Optimierungen

Die Optimierungen sind größtenteils nach den beschriebenen Standardverfahren implementiert worden. Die eigentliche Arbeitsgeschwindigkeit der Optimierungen hatte keine Priorität, so daß in diesem Punkt mit Sicherheit nachgebessert werden könnte.
Außerdem gibt es oftmals verschiedene Verfahren und Ansätze eine bestimmte Optimierung zu implementieren, so daß u.U. bessere Ergebnisse erzielt werden können, wenn weitere Zeit in die Untersuchung der möglichen Optimierungsstrategien gesteckt wird.

Zu den einzelnen Optimierungen wurden in den jeweiligen Kapiteln bereits Vorschläge gemacht, was beispielsweise an architekturspezifischen Besonderheiten ausgenutzt werden könnte. Einige Dinge davon sind auch bereits verwirklicht, andere könnten noch implementiert und verfeinert werden. Dazu ein kurzer Überblick über die möglichen Erweiterungen und vorhandenen Schwachstellen der einzelnen Optimierungen:

- **Constant Folding**

  Die jetzige *Constant Folding* Implementierung unterstützt aus den in Kapitel 5.2 genannten Gründen keine Fließkommawerte. Das wurde bis jetzt auch nicht benötigt, da die ARM Codegenerierung des *encc* Compilers bis jetzt auch keine Fließkommawerte unterstützt. Wenn die *GeLIR* Optimierung in anderen Compiler Backends eingesetzt wird, kann eine Fließkommaunterstützung sinnvoll werden.

- **Constant Propagation**

  Die *Constant Propagation* Optimierung kann dahingehend erweitert werden, daß unerreichbarer Code entdeckt und eliminiert wird, um den Kontrollflußgraph zu vereinfachen. Dazu müssten bedingte Sprünge identifiziert werden, die immer den gleichen Sprungpfad wählen. Dieses Verfahren ist als *Conditional Constant Propagation* bekannt. Derzeit implementiert ist das Verfahren *Simple Constant Propagation*.

- **Copy Propagation**

  Eingeschränkt nutzbare Register des LEON und die hohen Register des ARM, auf die nur mit bestimmten Befehlen zugegriffen werden kann, können im architekturabhängigen Teil der Optimierung wie in Kapitel 5.4.1 beschrieben, berücksichtigt werden.

- **Dead Code Elimination**

  Momentan werden Maschinenoperationen, die über einen Pointer angesprochen werden, von der *Dead Code Elimination* nicht weiter berücksichtigt, und verbleiben möglicherweise im Code, auch wenn diese überflüssig sind. Dazu könnte analysiert werden, auf welche Speicherbereiche die Pointer zeigen und anhand der Analyse entschieden werden, ob die Pointerzugriffe dem "Dead Code" zuzuordnen sind.

- **Redundant Load Elimination**

  Die derzeitige Implementation verwendet im architekturabhängigen Modus die hohen Register des ARM Befehlssatzes für die temporäre Zwischenspeicherung von Werten und kann daher nicht beliebig viele redundante Load-Befehle eliminieren, da sonst die Gefahr des Spillens besteht und der Nutzen der Optimierung in Frage gestellt wird.

- **Redundant Store Elimination**

  Die in Abschnitt 6.3.1 genannten Ideen könnten zusätzlich implementiert werden. Dazu zählt z.B. die Konvertierung des Schleifenepilogs in eine eigene Schleife, wenn eine bestimmte Größenordnung überschritten worden ist.

Je nach Einsatzzweck ist es sinnvoll auch noch weitere Optimierungen zu implementieren, so könnte z.B. der Branch Delay Slot des LEON Prozessors für eine Optimierung verwendet werden.

8.2.3 Code Selektion/Register Allokation auf Basis der GeLIR

Zum Zeitpunkt, als diese Diplomarbeit geschrieben wurde, gab es keine Code Selektion und Register Allokation, die auf den *GeLIR* Datenstrukturen arbeitet. Bei Optimierungen, die keine neuen Maschinenoperationen in den Code einfügen, wie es bei dem Großteil der hier behandelten Optimierungen der Fall ist, ist dies kein echtes Problem. Sobald allerdings neuer Code eingefügt werden muß, z.B. im Rahmen der *Redundant Load Elimination* im Schleifenprolog, wäre eine Code Selektion und Register Allokation auf *GeLIR* Basis wünschenswert.
Die Generierung von Assemblercode war im Rahmen dieser Diplomarbeit noch relativ einfach zu lösen, da bereits eine Code Selektion und Registerallokation vorangegangen waren und deren Informationen weiterverwendet werden konnte. Allerdings musste bei den Optimierungen darauf geachtet werden, daß nicht

beliebige neue Register eingefügt werden können.
Wenn zukünftig Codeselektion und Registerallokation fUr den ARM auf Basis der *GeLIR* exisitieren, wäre ein Einstieg direkt von der LANCE2-IR in *GeLIR* möglich. Dazu kann die bereits existierende Klasse `lance2gelir` verwendet werden. Ausgehend von der abstrakten Darstellung des C Programms müssen dann geeignete Ressourcen-Zuordnungen gefunden werden, um ausführbaren Code zu erzeugen.

Anhang A

Darstellung des ARM in GeLIR

Bevor die IR Darstellung in geeigneter Weise in die *GeLIR* Darstellung (hier mittels `CFG2GeLIR`) transformiert wird, müssen die Informationen für eine Zielarchitektur in die architekturspezifischen Datenstrukturen der *GeLIR* überführt werden.
Dieser Anhang stellt die nötigen Schritte für die ARM Target Architekturanbindung an die *GeLIR* beispielhaft dar. Eine allgemeiner gefasste Einleitung zu dem Thema gibt [FLW01].

A.1 Übersicht

Untenstehendes Beispiel ist ein Ausschnitt der wichtigsten Architekturmerkmale aus der Target Implementierung für den ARM7TDMI Prozessor. Zuerst werden die Maschineninstruktionstypen definiert, die zur besseren Übersicht unterteilt werden in Grundtypen, komplexe Typen (zusammengesetzte Typen) und ARM-spezifische Typen.
Anschließend werden die Typen für flüchtige Ressourcen, Speicherressourcen, Registerressourcen, Funktionseinheiten und Maschineninstruktionen definiert. Letztere sind nach den Funktionseinheiten ALU, MUL (Multiplizierer) und BSH (Barrel Shifter) gruppiert.
Eine Besonderheit in der *GeLIR* Darstellung stellen die flüchtigen Ressourcen dar. Folgendes Beispiel illustriert den Einsatz einer flüchtigen Ressource. $V4 = 17$ wird in der *GeLIR* Darstellung beispielsweise zu:

```
lir_53 := <const>17
                 \
  V4 := cp(none lir_53)
```

Beispiel A.1.1 *Flüchtige Ressource*

Dabei wird die Konstante 17 über die flüchtige Ressource *lir_53* in das virtuelle Register $V4$ kopiert. Die flüchtigen Ressourcen dienen nur der Modellierung und tauchen in der realen Programmdarstellung nicht auf.

```
/////////////////////////////////////////////
// types of machine instructions
/////////////////////////////////////////////

enum LirTypeKEY
{
  // basic types
  LIR_TYPE_S = (LIR_TYPE_NO + 1),   // short int
  LIR_TYPE_I,                       // int
  LIR_TYPE_L,                       // long int
  LIR_TYPE_UC,                      // unsigned char int
  LIR_TYPE_US,                      // unsigned short int
  LIR_TYPE_UI,                      // unsigned int
  LIR_TYPE_UL,                      // unsigned long int

  // complex types
  LIR_FUN_TYPE_NONE,                // no def, no nothing
  LIR_FUN_TYPE_NONE_I,              // no def = int
  LIR_FUN_TYPE_I_I,                 // int = int
  LIR_FUN_TYPE_I_I_I,               // int = (int X int)
  LIR_FUN_TYPE_I_I_I_I,             // int = (int X int X int)

  // ARM
  LIR_TYPE_INT3,                    //  3-bit value
  LIR_TYPE_INT5,                    //  5-bit value
  LIR_TYPE_INT6,                    //  6-bit value
  LIR_TYPE_INT7,                    //  7-bit value
  LIR_TYPE_INT8,                    //  8-bit value
  LIR_TYPE_INT10                    // 10-bit value
};

/////////////////////////////////////////////
// resources like register, function units
/////////////////////////////////////////////

enum LirResourceKEY
{
  // transitory ('fluechtige') resources
  TR_MUL = (LIR_RESOURCE_NO + 1),   // register
  TR_CONST_0,                       // 0
  TR_CONST_1,                       // 1
  TR_CONST_2,                       // 2
  TR_CONST_INT3,
  TR_CONST_INT5,
  TR_CONST_INT6,
  TR_CONST_INT7,
  TR_CONST_INT8,
  TR_CONST_INT10,
  TR_CONST_C,                       // [-128, 127]
  TR_CONST_S,                       // [-128, 127]
  TR_CONST_I,                       // [-32768, 32767]
  TR_ADDR,                          // [-32768, 32767]

  // memory resource
  RF_MEM,

  // register ressources
  RF_LO,                            // R0 - R7 Lo registers
  RF_HI,                            // R8 - R12 Hi registers
  RF_SP,                            // Stack Pointer
  RF_LR,                            // Link Register
  RF_PC,                            // Program Counter
  RF_CPSR,                          // Current Program Status Register
  RF_SPSR,                          // Saved Process Status Register

  // function unit ressources
  FU_ALU,
  FU_MUL,
  FU_BSH
};
```

```
//////////////////////////////////////////////////////////////////////
//
// machine instructions
//
// use definition numbers greater than ABS_OP_DEF_NO in order to avoid
// overlapping ranges with abstract machine operations
//////////////////////////////////////////////////////////////////////

enum LirOperationKEY
{
  /////////////////////////
  // ALU-Instructions
  /////////////////////////
  OP_NONE             = (LIR_OPERATION_NO + 1),
  OP_ALU_ADC,           // Add with Carry
  OP_ALU_ADD,           // Add
  .
  .
  OP_ALU_RETVOID,       // return (void) for GeLIR modelling
  OP_ALU_RETVAL,        // return value for GeLIR modelling

  /////////////////////////
  // MUL-Instructions
  /////////////////////////
  OP_MUL_MUL,           // Multiply


  /////////////////////////
  // BSH-Instructions
  /////////////////////////
  OP_BSH_ASR,           // Arithmetic Shift Right
  .
  .
  OP_BSH_ROR            // Rotate Right
};
```

A.2 Target Initialisierung

Über die Methode `InitTarget()` aus der Klasse `ARM_Target` wird die Targetbeschreibung initialisiert, so daß z.B. die *GeLIR* Optimierungen darauf zugreifen können.

Die Target Initialisierung läuft in vier Schritten ab:

1. Typspezifikation
2. Ressourcenspezifikation (Registersätze, Funktionseinheiten und Instruktionstypen)
3. Spezifikation der Operationen
4. Einfügen der Alternativen für abstrakte Maschinenoperationen

A.2.1 Typspezifikation

In diesem Abschnitt werden einfache und komplexe Typen für Variablen und n-stellige Operationen spezifiziert. Eine genaue Funktions- und Parameterbeschreibung kann der *GeLIR* Interface Dokumentation entnommen werden [FLW01].

```
void ARM_Target::InsertTargetType(LirTypeKEY type_key,
                LirType::TypeClass type_class,
                unsigned int size, bool b, const string& type_name)
{..}

//////////////////////////////////////////////////////////////////////
///////////// specification of all types ///////////////////////////////
//////////////////////////////////////////////////////////////////////

void ARM_Target::InsertTargetTypes()
{
  // basic types
  // 1-Byte Integer
  InsertTargetType(LIR_TYPE_S,  LirType::INT, 1, true, "type_s");
  .
  .

  // 3-Bit Integer
  InsertTargetType(LIR_TYPE_INT3, LirType::INT, 3, false, "type_int3");
  .
  .

  // function types
  // fun : int = int x int
  InsertTargetType(LIR_FUN_TYPE_I_I_I, LirType::FUN, LIR_TYPE_I, bin_vec,
    "type_i_i_i");
  .
  .
}
```

A.2.2 Ressourcenspezifikation

In diesem Abschnitt werden die Registerressourcen, Speicherressourcen und Funktionseinheiten in die Targetbeschreibung eingefügt. Bei der ARM Architekturbeschreibung reicht eine Speicherressource aus, da hier ein homogener, linear organisierter Speicher abgebildet wird.
Instruktionstypen könnten an dieser Stelle ebenfalls definiert werden. Davon wird aber hier kein Gebrauch gemacht. Durch die Instruktionstypen kann in der *GeLIR* Darstellung die mögliche Parallelausführung bestimmter Instruktionen modelliert werden. Da der ARM Prozessor allerdings kein Instruction Level Parallelism unterstützt, kann hier auf diese Funktionalität verzichtet werden.

```
void ARM_Target::InsertTargetResource(KEY rs_class, LirResourceKEY rs_key,
                   LirResource::ResourceClass res_class,
                   const string& rs_name, SIZE size, int low, int high,
                   LirTypeKEY type_key)
{..}

//------------------------------------------------------------------------

void ARM_Target::InsertTargetResources()
{
  // insert memory resource into target
  InsertTargetResource(RF_Rs, RF_MEM,  mem, "Mem",      1, 70, 70, LIR_TYPE_I);

  // insert register files into target
  InsertTargetResource(RF_Rs, RF_R0,   reg, "Reg_R0",   1, 71, 71, LIR_TYPE_I);
  .
  .

  // insert transitory resources into target
  InsertTargetResource(RF_Rs, TR_CONST_INT3, trans, "'const_int3'", 1, 5005,
    5005, (LirTypeKEY) LIR_TYPE_NONE);
  .
  .

  // insert function units into target
  InsertTargetResource(FU_Rs, FU_ALU, none, "ALU", 1, 6000, 6000,
    (LirTypeKEY) LIR_TYPE_NONE);
  .
  .
}
```

A.2.3 Operationenspezifikation

Nachdem die Typen spezifiziert und die Ressourcen eingefügt worden sind, müssen die Operationen ebenfalls spezifiziert werden. Um die Übersicht zu wahren, wird hier beispielhaft eine Variationen des ADD Befehls dargestellt. Alle anderen Operationen der Zielarchitektur werden analog eingefügt.
Mit Hilfe der Operationenspezifikation lassen sich die verschiedenen (Adressierungs)-Arten eines Befehls abbilden. Im Falle des ADD Befehls wären das z.B.:

```
ADD Rd, Rs, Rn
ADD Rd, Rs, #3bit_Imm
ADD Rd, Hs
ADD Hd, Rs
ADD Hd, Hs
ADD Rd, #8bit_Imm
ADD SP, #+/-7bit_Imm
```

```
void ARM_ALU::InsertAdd()
{
  // fun : int = int x int x int
  LirType* type = LirTarget::FindType(LIR_FUN_TYPE_I_I_I);
  LirAlt* alt = new LirAlt();

  // ADD Rd, Rs, Rn

  // generate default alternatives
  LirAltEntry* alt_entry1 = new LirAltEntry(type->Arity());
  alt_entry1 -> InsertOperation(OP_ALU_ADD);
  alt_entry1 -> InsertFU(FU_ALU);
  alt_entry1 -> InsertIT(LIR_IT_NONE);
  alt_entry1 -> InsertDef(RF_LO);
  alt_entry1 -> InsertDef(TR_ADDR);
  alt_entry1 -> InsertArg(RF_LO, 1);
  alt_entry1 -> InsertArg(RF_LO, 2);

  LirAltEntry* alt_entry2 = new LirAltEntry(type->Arity());
  alt_entry2 -> InsertOperation(OP_ALU_ADD);
  alt_entry2 -> InsertFU(FU_ALU);
  alt_entry2 -> InsertIT(LIR_IT_NONE);
  alt_entry2 -> InsertDef(RF_LO);
  alt_entry2 -> InsertDef(TR_ADDR);
  alt_entry2 -> InsertArg(RF_LO, 1);
  alt_entry2 -> InsertArg(RF_LO, 2);
  alt_entry2 -> SetSwappedArgs(true);

  // insert alt_entries into alt
  alt -> InsertAltEntry(alt_entry1);
  alt -> InsertAltEntry(alt_entry2);

  ...
  ...

  // insert operation into target
  ARM_Target::InsertTargetOperation(OP_ALU_ADD, "ADD", LIR_FUN_TYPE_I_I_I,
    true, alt);
}
```

A.2.4 MO Alternativen

Abschließend werden die Alternativen für die abstrakten Maschinenoperationen eingefügt. Dabei wird in der Target Darstellung jeder abstrakten *GeLIR* Operation (z.B. vom Typ *LIR_PLUS*) eine (oder mehrere) alternative Operation(en) der Zielarchitekur zugeteilt (z.B. vom Typ *OP_ALU_ADD*). Auch hier wieder aus Gründen der Übersicht nur ein exemplarischer Ausschnitt am Beispiel der Addition.

```
void ARM_AMO::InsertAlternative()
{
  InsertAltLirMult();
  InsertAltLirPlus();
  ...
  ...
  InsertAltLirLd();
  InsertAltLirSt();
}

//--------------------------------------------------------------------------

void ARM_AMO::InsertAltLirPlus()
{
  LirAlt* alt = LirTarget::FindOperation(OP_ALU_ADD)->Alt()->Copy();
  LirTarget::FindOperation(LIR_PLUS)->SetAlt(alt);
}
```

A.2.5 GeLIR Darstellung

Abbildung 1.1 zeigt den ersten Basisblock von folgendem Beispielprogramm in der *GeLIR* Darstellung nach der Target Initialisierung. Neben der reinen Programmdarstellung sind die alternativen Ressourcen für jede MO erkennbar.

```
int main(void)
{
int a[80];
int b[64];
int c[64];
int i, lc;

  for (i = 0; i < 80; i++) {
    a[i] = 0;
  }

  for (i = 0; i < 64; i++) {
    b[i] = i;
    c[i] = i;
  }

  for (lc = 0; lc < 64; lc++) {
    a[lc] = b[lc] + c[lc];
  }
}
```

Beispiel A.2.1 *Beispielprogramm*

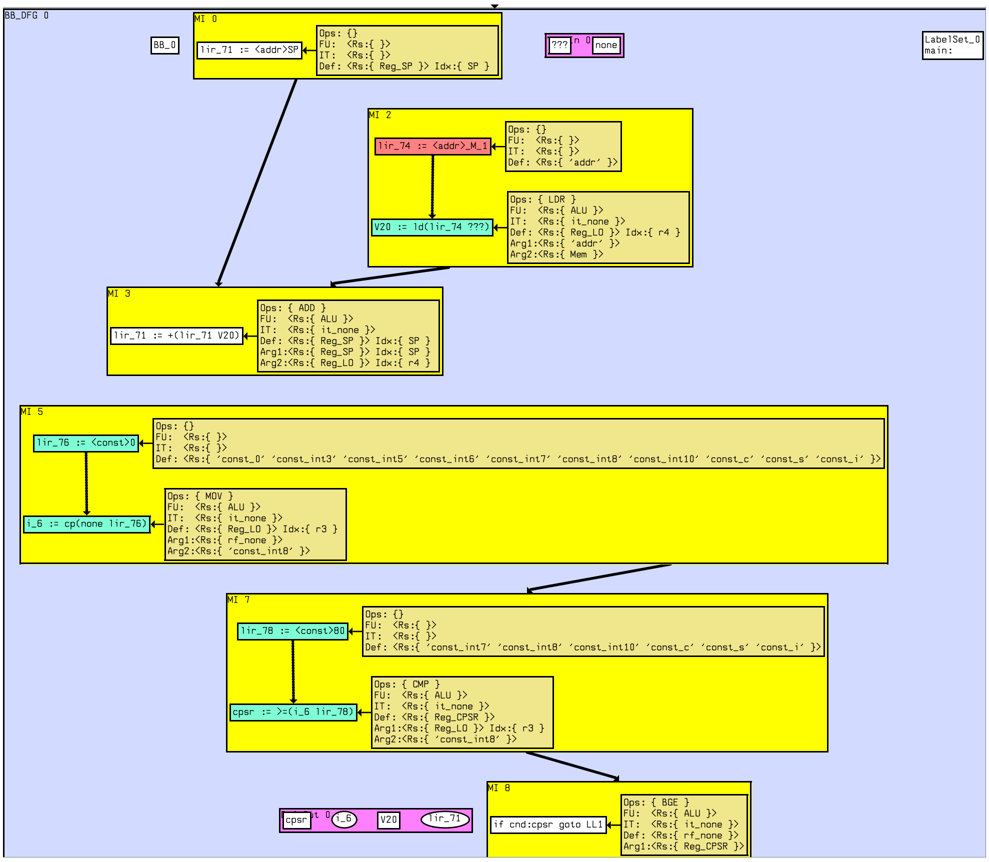

Abbildung 1.1: GeLIR Darstellung inkl. alternative Ressourcen

Anhang B

Dokumentation der Implementierungen

Dieser Anhang erläutert, welche Werkzeuge für eine lauffähige Arbeitsumgebung installiert werden müssen. Außerdem werden die Änderungen und neuen Implementierungen, die im Rahmen dieser Diplomarbeit entwickelt wurden, erklärt. Dazu zählen der Aufbau der Erweiterungen sowie die nötigen Grundlagen, um den Konverter und die Optimierungen in eigenen Projekten einzusetzen.

B.1 Installation der Arbeitsumgebung

Eine lauffähige Arbeitsumgebung unter SunOS oder Linux setzt den *encc* Compiler, das *Lance2* Frontend, die *GeLIR* und die *GeLIR Utilities* voraus. Als weiteres Werkzeug empfiehlt es sich, *aiSee* zur Visualisierung der Datenstrukturen zu installieren.

Es bietet sich an, die einzelnen Tools in jeweils eigenständigen Verzeichnissen zu entpacken/installieren, die dann unter einem Projektverzeichnis zusammengefasst werden. Eine typische Installation könnte etwa folgende Verzeichnisstruktur aufweisen, wobei der Verzeichnisname des *encc* Compilers aus historischen Gründen "ARM12CC" ist.

Im folgenden wird die Installation auf einem gängigen Unix System beschrieben. Es wird vorausgesetzt, daß das System als solches erwartungsgemäß aufgesetzt und mit den nötigen Standardtools (GCC, TCL, STL, etc.) versehen ist, sowie die am Lehrstuhl 12 entwickelten Softwarepakete *ARM12CC*, *GELIR* und *LANCE2* bereitstehen.
Auf einem Linux System muß die Umgebungsvariable `export OS=Linux` und auf einem SunOS System unter Solaris `export OS=SunOS` gesetzt sein, damit die Makefiles korrekt ausgeführt werden können.

Die folgenden Installationsanweisungen sind in dieser Form nur für Solaris Rechner des Lehrstuhl 12 gültig. Desweiteren muß man zur Verwendung der Versionsverwaltungssoftware "cvs" Mitglied in der Gruppe "lowpower" sein.

```
/Projects
|
|-- ARM12CC
|
|-- GELIR
|
|-- GELIR_ARM
|
|-- GELIR_UTIL
|
|-- LANCE2
|
|-- UTIL
|
\-- Tools
 |
  \-- aiSee
```

B.1.1 Installation des LANCE2 Frontends

Das *LANCE2* Frontend mit den beiden Befehlen `module add arm12cc` und `cvs checkout LANCE2` im Projektverzeichnis installieren und folgende Umgebungsvariablen setzen:

```
export LANCE2_ROOT=~/Projects/LANCE2
export LANCE2_LIB=$LANCE2_ROOT/LIB/$OS
export LANCE2_INCL=$LANCE2_ROOT/INCL
export LANCE2_CONFIG=$ARM12CC_ROOT/config.arm
```

Da die für den *encc* Compiler notwendigen *LANCE2* Headerdateien und Bibliotheken mittlerweile auch im *encc* Compiler Paket abgelegt werden, ist eine manuelle Installation des *LANCE2* Paketes nicht zwingend notwendig. Einzig die Umgebungsvariablen müssen gesetzt werden.

B.1.2 Installation der GELIR Umgebung

Die *GeLIR* Klassenbibliothek wird mit den Kommandos `module add GELIR` und `cvs checkout GELIR` im Projektverzeichnis installiert und mit dem Befehl

`make` übersetzt.
Folgende Umgebungsvariablen werden für den Einsatz der *GeLIR* benötigt:

```
export GELIR_HOME=~/Projects/GELIR
export GELIR_LIB=$GELIR_HOME/LIB/$OS
export GELIR_INCL=$GELIR_HOME/INCL
```

Eine Dokumentation zur *GeLIR* kann optional mit `make gelir_doc` erzeugt werden.

Die *GeLIR* Klassenbibliothek dient als Grundlage für die Optimierungen und stellt die nötigen Datenstrukturen zur Darstellung eines Programms in einer Low-Level IR bereit. Details finden sich in Kapitel 3.4.

B.1.3 Installation der GELIR_UTIL Erweiterung

Voraussetzung: `GeLIR`

Um die *GeLIR Utilities* zu installieren, müssen die Kommandos `module add GELIR` und `cvs checkout GELIR_UTIL` im Projektverzeichnis ausgeführt werden und die Quellcodes anschließend mit `make` übersetzt werden.

Folgende Umgebungsvariable muß gesetzt sein:

```
export GELIR_UTIL_HOME=~/Projects/GELIR_UTIL
```

Die *GeLIR Utilities* beinhalten Tools, die auf der *GeLIR* arbeiten. Dazu zählen beispielsweise die *GeLIR Optimierungen* oder der *GeLIR Simulator*.

B.1.4 Installation der GELIR_ARM Erweiterung

Voraussetzungen: `GeLIR`, `GELIR_UTIL`, `LANCE2` und `UTIL`

Die ARM Erweiterung zur *GeLIR* wird mit `module add arm12cc` und anschließendem `cvs checkout GELIR_ARM` im Projektverzeichnis installiert und wird für den *encc* Compiler mit *GeLIR* Unterstützung benötigt.

Folgende Umgebungsvariable muß gesetzt sein:

```
export GELIR_ARM_HOME=~/Projects/GELIR_ARM
```

Durch die *GeLIR ARM* Erweiterung kann die interne Datenstruktur des *encc* Compilers in eine *GeLIR* Datenstruktur übertragen werden, damit die *GeLIR* Optimierungen auch für den *encc* Compiler zur Verfügung stehen. Für weitere Details zu der ARM Erweiterung siehe Kapitel 3.6

B.1.5 Installation des UTIL Pakets

Voraussetzungen: keine

Das UTIL Paket wird mit den Kommandos `module add GELIR` und `cvs checkout UTIL` im Projektverzeichnis installiert.

Folgende Umgebungsvariable muß gesetzt sein:

```
export UTIL_HOME=~/Projects/UTIL
```

Im UTIL Paket sind einige Erweiterungen zu finden, die auch unabhängig von der *GeLIR* genutzt werden können, z.B. Grundroutienen für eine Delta Array Datenflußanalyse, deren Funktionalität in die *GeLIR* Optimierungen RSE und RLE eingeflossen ist.
Die hier vorliegende *Delta-Array-Datenflußanalyse* kann mittels Templateklassen auf beliebigen Datenstrukturen operieren, z.B. der *GeLIR*. Sie muß vor der Benutzung lediglich in geeigneter Weise initialisiert werden. Details dazu finden sich in [MH01].

B.1.6 Installation des encc Compilers

Voraussetzungen: `LANCE2`
Optional: `GELIR, GELIR_ARM` und `GELIR_UTIL` (für GeLIR Unterstützung und Low-Level Optimierungen)

Üblicherweise kann der Compiler mit den beiden Kommandos `module add arm12cc` und `cvs checkout ARM12CC` im Projektverzeichnis installiert werden. Mit `make gelir` wird er inkl. *GeLIR* Unterstützung übersetzt bzw. nur mit `make` ohne *GeLIR* Unterstützung.

Im `ARM12CC` Verzeichnis befindet sich der *encc* Compiler. Über das TCL-Skript `encc` aus dem `ARM12CC/BIN` Verzeichnis läßt sich eine GUI für den Compiler aufrufen.
Damit der Compiler funktioniert, müssen folgende Umgebungsvariablen gesetzt werden:

```
export ARM12CC_ROOT=~/Projects/ARM12CC
export PATH=$PATH:$ARM12CC_ROOT/BIN/$OS:$ARM12CC_ROOT/BIN
```

Der *encc* Compiler ist die Hauptanwendung. Er dient der Codeerzeugung für ARM- und LEON-Systeme. Details zum Compiler finden sich in Kapitel 3.4.3.

Wird der *encc* Compiler mittels `make gelir` übersetzt, werden rekursiv alle anderen Makefiles aufgerufen, so daß das komplette System übersetzt wird.

B.2 Der Konverter CFG2GeLIR

Dieser Abschnitt soll dem Anwender des `CFG2GeLIR` Konvertes einen Überblick über die enthaltenen Dateien und deren Einsatz vermitteln.

Folgende Dateien gehören zum Konverter `CFG2GeLIR`:

```
.
.
|-- GELIR_ARM
|   |
|   |-- INCL
|   |   |
|   |   |-- CFG2GeLIR.h     /* Konverter CFG2GeLIR - Header Datei    */
|   |   |
|   |   |                   /* Architekturbeschreibung des ARM7TDMI  */
|   |   |                   /* Headerdateien:                        */
|   |   |-- alu_arm.h       /* ALU                                   */
|   |   |-- amo_arm.h       /* Alternativen für abstrakte MOs        */
|   |   |-- bsh_arm.h       /* Barrel Shifter                        */
|   |   |-- defs_arm.h      /* Typen Definitionen                    */
|   |   |-- isa_arm.h       /* Hilfsfunktionen                       */
|   |   |-- mul_arm.h       /* Multiplizierer                        */
|   |   \-- target_arm.h    /* Hauptklasse                           */
|   |
|   \-- SRC
|       |
|       |-- CFG2GeLIR.cpp   /* Konverter CFG2GeLIR C++ Datei */
|       |
|       |                   /* Architekturbeschreibung des ARM7TDMI  */
|       |                   /* C++ Dateien:                          */
|       |-- alu_arm.cpp     /* ALU                                   */
|       |-- amo_arm.cpp     /* Alternativen für abstrakte MOs        */
|       |-- bsh_arm.cpp     /* Barrel Shifter                        */
|       |-- isa_arm.cpp     /* Hilfsfunktionen                       */
|       |-- mul_arm.cpp     /* Multiplizierer                        */
|       \-- target_arm.cpp  /* Hauptklasse                           */
.
.
```

Die Datei `CFG2GeLIR.h` muß eingebunden werden, wenn der Konverter benutzt werden soll. Standardmäßig ist diese Datei in der Datei `cselect.h` des *encc* Compilers eingebunden, wenn dieser mit der Option `make gelir` übersetzt worden ist, und wird dort nach der Code Selektion und Register Allokation aufgerufen.
Die Klasse `CCFG2GeLIR` stellt nach außen hin folgende Funktionen zur Verfügung:

```
class CCFG2GeLIR
{
public:
  CCFG2GeLIR(FCGraph* pFunCallGraph, SymTable* pGlobalSymTable,
             IntermediateRepresentation* pLance2IR);

  LirGeLIR* GetGeLIR();
}
```

Der Konstruktor `CCFG2GeLIR` wird mit dem *encc* Function Call Graph, der globalen *encc* Symboltabelle und der *LANCE2* IR als Parameter aufgerufen und konvertiert die *encc* Datenstrukturen in das *GeLIR* Format. Diese *GeLIR* Darstellung läßt sich mit der Funktion `GetGeLIR()` als Zeiger auf ein `LirGeLIR` Objekt zur weiteren Verarbeitung zurückgeben.

Die restlichen .h und .cpp Dateien beinhalten die Architekturbeschreibung des ARM. Sie werden automatisch vom Konverter benutzt und brauchen nicht seperat eingebunden zu werden. In ihnen ist das Mapping encc → abstrakte Operationen → Target Operationen (inkl. ARM Ressourcen) implementiert. Details dazu finden sich in Kapitel 2.3.4.

B.3 Die Optimierungen

B.3.1 Übersicht

Die *GeLIR* Optimierungen befinden sich im `GELIR_UTIL` Verzeichnis:

```
.
.
|-- GELIR_UTIL
|   |
|   |-- INCL                           /* Header Dateien:             */
|   |   |
|   |   |-- GeLIR_ConstantFolding.h      /* Constant Folding            */
|   |   |-- GeLIR_ConstantPropagation.h  /* Constant Propagation        */
|   |   |-- GeLIR_CopyPropagation.h      /* Copy Propagation            */
|   |   |-- GeLIR_DeadCodeElimination.h  /* Dead Code Elimination       */
|   |   |-- GeLIR_DeltaAnalysis.h        /* Delta Array Analyse Init.   */
|   |   |-- GeLIR_Opt.h                  /* Hauptklasse                 */
|   |   |-- GeLIR_RLE.h                  /* Redundant Load Elimination  */
|   |   \-- GeLIR_RSE.h                  /* Redundant Store Elimination */
|   |
|   \-- SRC                            /* C++ Dateien:                */
|       |
|       |-- GeLIR_ConstantFolding.cpp    /* Constant Folding            */
|       |-- GeLIR_ConstantPropagation.cpp/* Constant Propagation        */
|       |-- GeLIR_CopyPropagation.cpp    /* Copy Propagation            */
|       |-- GeLIR_DeadCodeElimination.cpp/* Dead Code Elimination       */
|       |-- GeLIR_DeltaAnalysis.cpp      /* Delta Array Analyse Init.   */
|       |-- GeLIR_Opt.cpp                /* Hauptklasse                 */
|       |-- GeLIR_RLE.cpp                /* Redundant Load Elimination  */
|       \-- GeLIR_RSE.cpp                /* Redundant Store Elimination */
.
.
```

B.3.2 Einbindung aller Optimierungen

Es bietet sich an, die Optimierungen über die Hauptklasse `CGeLIR_Opt` zu integrieren, da diese dann über die Konfigurationsdatei `gelir_config.arm` bzw. `gelir_config.leon` konfiguriert und gezielt ein- und ausgeschaltet werden können. Die aktivierten Optimierungen werden dabei in einer sinnvollen Reihenfolge so lange aufgerufen, bis das Optimierungspotential ausgeschöpft ist.

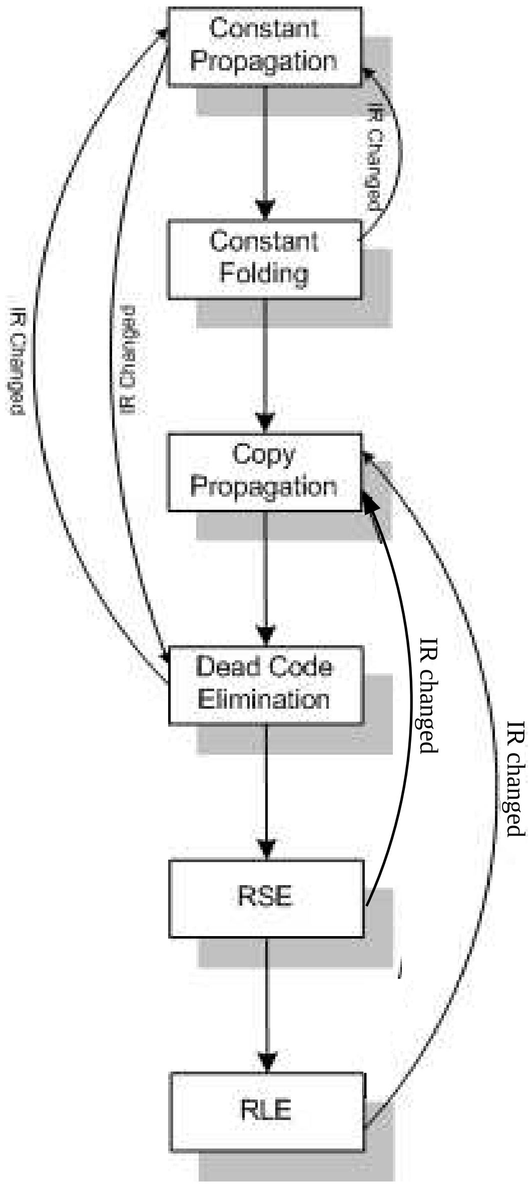

Abbildung 2.1: Aufrufstrategie der Optimierungen

Bei der Aufrufreihenfolge der Optimierungen wird die in Abbildung 2.1 dargelegte Strategie verfolgt. Jede Optimierung für sich wird so lange durchgeführt, bis keine Änderung der *GeLIR* Darstellung durch die entsprechende Optimierung mehr möglich ist und es wird dann zur nächsten Optimierung gesprungen. Das wird in der Abbildung 2.1 durch die dickeren, senkrechten Pfeile symbolisiert.

Konnte der Code durch die *Constant Propagation* optimiert werden, werden unabhängig von der weiteren Aufruffolge ebenfalls *Constant Folding* und *Dead Code Elimination* zum (evtl.) erneuten Aufruf vorgemerkt. Dies ist sinnvoll, da durch die *Constant Propagation* eingesetzte Konstanten u.U. durch ein *Con-*

stant Folding zusammengefasst werden können. Der Aufruf der *Dead Code Elimination* dient dem Löschen von Zuweisungen, die duch die *Constant Propagation* überfüssig geworden sind.

Nach erfolgreichem *Constant Folding* wird die Optimierung *Constant Propagation* zum erneuten Start vorgemerkt, da aus dem Zusammenfalten von Gleichungen neue Konstanten hervorgegangen sein könnten, die mittels *Copy Propagation* an geeigneter Stelle eingesetzt werden können.

Da sich nach durchgeführter *Copy Propagation* nicht mehr benötigter Code ansammeln kann, wird im Erfolgsfall der Start der *Dead Code Elimination* neu angestoßen.

Nach der *Dead Code Elimination*, wird aus o.g. Gründen ebenfalls ein erneuter Aufruf der *Constant-* und *Copy Propagation* eingeleitet.

Redundant Store Elimination und *Redundant Load Elimination* lösen im Erfolgsfall erneut einen Aufruf der *Copy Propagation* mit anschließender *Dead Code Elimination* aus, um überflüssige Operationen, die z.B. zu einer Adressberechnung eines Arrayzugriffs gehört haben, zu eleminieren.

Das Verfahren bricht ab, wenn keine der Optimierungen neue Änderungen an der *GeLIR* Darstellung mehr vornimmt und achtet darauf, daß sich die Optimierungen nicht endlos aufrufen.

Für die Nutzung der Optimierungen muß lediglich die Header Datei `GeLIR_Opt.h` eingebunden und die einzige Zugriffsfunktion der Klasse

```
void CGeLIR_Opt::Optimize(LirGeLIR* pGeLIR, string strIRFileName="")
```

an geeigneter Stelle aufgerufen werden. Standardmäßig wird `Optimize` in der jetzigen Implementierung innerhalb der `CFG2GeLIR` Klasse nach Erzeugung der *GeLIR* Datenstruktur aufgerufen.
Im Prinzip kann die Klasse an jeder Stelle aufgerufen werden, an der die *GeLIR* Datenstruktur zur Verfügung steht, da diese als einzig zwingender Parameter übergeben werden muß. Optional kann zusätzlich noch der Dateiname der *GeLIR* Darstellung zugrundeliegenden .IR Datei mit übergeben werden, um daraus die nötigen Dateien für den *GeLIR Simulator* zu erzeugen. Fehlt dieser Parameter, so werden keine Dateien für den Simulator geschrieben.
Durch das optionale Anlegen der Simulationsdateien unmittelbar nach der Optimierung, können die Auswirkungen der Optimierungen bei Bedarf mit Hilfe des *Gelir Simulators* getestet werden. Diese Funktionalität ist bei den Optimierungen nicht notwendig und daher standardmäßig abgeschaltet, wurde aber während der Implementierung zu Testzwecken verwendet und in der Implementierung belassen.

B.3.3 Konfigurationsdatei

Die Konfigurationsdatei zu den Optimierungen befindet sich im `GELIR/CONFIG/` Verzeichnis. Die Einstellungen sind optional. Fehlt die Konfiguratiosdatei, so werden sinnvolle Voreinstellungen verwendet. Die Konfigurationsdatei hat folgenden Aufbau (Auszug der für die Optimierungen relevanten Optionen):

```
####################################################################
## Configuration section for low-level GeLIR optimizations (general) ##
####################################################################

# Syntax: option = on|off

# Write log file geliropt.log
GeLIR_Opt_LogFile = on

# Write AISee .gdl files before and after optimizations
GeLIR_Opt_WriteAISee = on

# Write Gelir Simulator files after optimizations
GeLIR_Create_Sim_Files = off

# Directory for .log, .gdl and gelirsim files
# Empty equals current directory
GeLIR_Opt_Output_Dir =

# Display more details while running optimizations
GeLIR_Opt_Verbose = on

# Dead Code Elimination
GeLIR_Opt_DCE = on

# Constant Folding
GeLIR_Opt_ConstFold = on

# Constant Propagation
GeLIR_Opt_ConstProp = on

# Copy Propagation
GeLIR_Opt_CopyProp = on

# Redundant Store Elimination
GeLIR_Opt_RSE = on

# Redundant Load Elimination
GeLIR_Opt_RLE = on

# Use CPU specific optimization enhancements
GeLIR_CPU_Specific = off
```

Der Syntax ist selbsterklärend. Die Optionen werden mit dem Parameter `on` eingeschaltet und mit `off` abgeschaltet. Eine Ausnahme bildet lediglich die Option `GeLIR_Opt_Output_Dir`. Hier kann optional ein Verzeichnisname in Unix Notation angegeben werden, wenn die ausgegebenen Dateien nicht im aktuellen Verzeichnis abgelegt werden sollen.
Die Optimierungen arbeiten in ihrer Grundeinstellung völlig architekturunabhängig. Mit der letztgenannten Option `GeLIR_CPU_Specific` können allerdings architekturspezifische Erweiterungen aktiviert werden. Derzeit sind diese Erweiterungen für die Optimierungen *Constant Folding* und *Constant Propagation* implementiert und durch folgende Optionen in der Konfigurationsdatei (am Beispiel ARM) steuerbar:

```
#########################################################
## Configuration section for GeLIR Constant Propagation ##
#########################################################

# Max immeditate bit size of LIR_COPY operation, set 0 to ignore
GeLIR_COPY_IMMEDIATE = 8

# Max immeditate bit size of LIR_PLUS operation, set 0 to ignore
GeLIR_PLUS_IMMEDIATE = 3

# Max immeditate bit size of LIR_UNARY_PLUS operation, set 0 to ignore
GeLIR_UNARY_PLUS_IMMEDIATE = 8

# Max immeditate bit size of LIR_MINUS operation, set 0 to ignore
GeLIR_MINUS_IMMEDIATE = 3

# Max immeditate bit size of LIR_UNARY_MINUS operation, set 0 to ignore
GeLIR_UNARY_MINUS_IMMEDIATE = 8

# Max immeditate bit size of LIR_CMP operation, set 0 to ignore
GeLIR_CMP_IMMEDIATE = 8

# Max immeditate bit size of LIR_SHL operation, set 0 to ignore
GeLIR_SHL_IMMEDIATE = 5

# Max immeditate bit size of LIR_SHR operation, set 0 to ignore
GeLIR_SHR_IMMEDIATE = 5

# Max immeditate bit size of LIR_LD operation, set 0 to ignore
GeLIR_LD_IMMEDIATE = 7

# Max immeditate bit size of LIR_ST operation, set 0 to ignore
GeLIR_ST_IMMEDIATE = 7
```

Mit Hilfe der Optionen kann die maximale Größe als Bitanzahl für die einzelnen Operationsklassen festgelegt werden. `GeLIR_COPY_IMMEDIATE = 8` bedeutet z.B., daß nur Konstanten in eine `COPY` Operation propagiert werden, wenn die Größe der Konstanten ≤ 8 Bit beträgt.
Wird als Größe 0 angegeben, so wird die entsprechende Schranke ignoriert.
Im oben gezeigten Auszug der Konfigurationsdatei sind die maximal zulässigen Obergrenzen der Immediate Werte für die ARM Architektur festgelegt. Wenn größere Konstanten eingesetzt würden, könnten diese Befehle nicht direkt auf einen ARM Assembler Befehl abgebildet werden. Die Code Selektion würde diese "großen" Konstanten dann wieder unter Verwendung zusätzlicher Befehle aufteilen und der Geschwindigkeitsvorteil durch die Optimierung wäre hinfällig.

```
#####################################################
## Configuration section for GeLIR Constant Folding ##
#####################################################

# Max. number of inserted statements for CPU specific constant folding
# to avoid LDR. Set to zero to ignore.
GeLIR_CPU_ConstFoldDepth = 3
```

Beim *Constant Folding* kann es passieren, daß die gefaltete Konstante zu groß ist um als Immediate Wert im entsprechenden Assemberbefehl der Zielarchitektur dargestellt zu werden. Soll z.B. der Wert 255 in das Register `V0` kopiert werden, kann dies im ARM Thumb Modus durch ein simplen `MOV V0, 255` Befehl erreicht werden. Bei größeren Werten muß allerdings auf einen `LDR` Befehl zurückgegriffen werden, der zusätzliche teure Speicherzugriffe benötigt.
Der architekturabhängige Teil des *Constant Folding* versucht die teureren `LDR` Befehle durch günstigere Befehle (im Sinne des Resourcenverbrauchs) zu ersetzen. So wird beispielsweise der `LDR` Befehl durch einen `MOV` Befehl mit anschließender arithmetischer Opertion(en) ersetzt um das gleiche Ergebnis zu erhalten.

Mit der Option `GeLIR_CPU_ConstFoldDepth` kann die maximale Anzahl der Befehle festgelegt werden, die an Stelle des `LDR` Befehls eingesetzt werden dürfen. Im o.g. Beispiel wird ein `LDR` Befehl durch maximal drei "preiswertere" Befehle ersetzt. Kann der Befehl nicht durch maximal drei andere Befehle dargestellt werden, so wird das *Constant Folding* für diesen Befehl nicht durchgeführt.
Mit dieser optionalen Begrenzung kann eine sinnvole Obergrenze für die jeweilige Zielarchitektur angegeben werden, da es sinnlos ist ein `LDR` Befehl durch zu viele andere Befehle zu ersetzen, die den Geschwindigkeitsvorteil wiederum zu nichte machen.
Diese Funktionalität des *Constant Folding* ist beispielhaft für einige Fälle implementiert und kann problemlos erweitert werden.

B.3.4 Einbindung einzelner Optimierungen

Die einzelnen Optimierungen können alle unabhängig von der Klasse `CGeLIR_Opt` auf *GeLIR* Datenstrukturen angewendet werden. Die passenden Parameter aus der Konfigurationsdatei werden weiterhin ausgewertet.
Es muß nur die zur jeweiligen Optimierung passende Header Datei eingebunden werden und die statische `Optimize` Funktion der entsprechenden Klasse aufgerufen werden.

Die `Optimize` Funktionen können folgende Parameter haben:

Parameter	Aktion
LirGeLIR* pGeLIR	Gültige LirGeLIR Datenstruktur
LirFun* pGeLIRFun	LirFun Objekt. Wenn dieser Parameter vorhanden ist, kann die entsprechende Optimierung direkt für eine bestimmte Funktion ausgeführt werden.
LirBB* pLIRBB	LirBB Objekt. Analog zum LirFun Objekt, wird die Optimierung auf dem angegeben Basisblock durchgeführt.
bool bWriteLogFile	Optionales Anlegen eines Log Files mit Informationen über den Optimierungsverlauf. Standardmäßig abgeschaltet.
bool bShowMessages	Optional werden Meldungen über den Optimierungsfortschritt auf der Standard Output Konsole ausgegeben. Standardmäßig eingeschaltet.
bool bCPUSpecific	Architekturabhängige Erweiterungen für die jeweilige Optimierung einschalten. Standardmäßig abgeschaltet.

Tabelle B.1: Funktionsparameter der Optimierungen

Mittels `Klassenname::Optimize(...)` werden die entsprechenden Optimierungen gestartet. Ein *Constant Folding* Durchlauf über die gesamte *GeLIR* Datenstruktur, würde beispielsweise wie folgt aufgerufen:

```
CGeLIRConstantFolding::Optimize(pGeLIR)
```

In der Regel bieten die Zugriffsfunktionen der Optimierungsklassen die Möglichkeit, auch einzelne Funktionen oder Basisblöcke gezielt zu optimieren. Das ist sinnvoll, wenn man z.B. bewußt nur einen Basisblock modifiziert hat und eben diesen optimieren möchte.
Alle Optimierungen geben einen Boolean Wert zurück. Ist dieser 1, so konnte die entsprechende Optimierung den *GeLIR* Code modifizieren. Bei der Rückgabe einer 0 konnte nichts optimiert werden.
Für weitere Details zum Aufrufen der Optimierungen empfiehlt sich ein Blick in die entsprechenden Header Dateien.

Anhang C

Programmierrichtlinien

Um den Programmcode der Implementierungen übersichtlich und lesbarer zu gestalten, wurden neben den obligatorischen Kommentaren die im Folgenden erläuterten Programmierrichtlinien angewendet, die sich nach der ungarischen Notation richten.

- **E**indeutige Typendefinitionen

Prefix	Typ
T	Typen
C	Klassen
E	Enumeration

Tabelle C.1: Typendefinitionen

Klassen werden durch ein dem Namen vorangestelltes `C` gekennzeichnet, z.B. `class CGeLIR_Opt`.

- **V**ariablenlokalität

Prefix	Typ
m_	Member Variable einer Klasse
g_	Globale Variable
ohne Prefix	Lokale Variablen in Funktionen

Tabelle C.2: Variablenlokalität

Globale Variablen innerhalb einer Klasse, sog. Member Variablen, werden z.B. durch ein vorangestelltes `m_` gekennzeichnet, z.B. `LirFun* m_pGeLIRFun`.

- **V**ariablentypen

Im o.g. Beispiel `LirFun* m_pGeLIRFun` wurde demnach ein `p` vorangestellt, da es sich um einen Zeiger auf ein `LirFun` Objekt handelt.

Prefix	Typ
b	Boolean
p	Pointer
c	Konstante Variable
f	Float
d	Double Float
i	Integer
ui	Unsigned Integer
l	Long Integer
ul	Unsigned Long Integer
sz	String Zero Terminated
str	String Klasse
it	Iterator

Tabelle C.3: Variablentypen

- **D**ebugcode

  Debugausgaben sind in entsprechende Debug Makros eingebettet und können über eine Konfigurationsdatei gezielt für die jeweiligen Funktionen ein- und ausgeschaltet werden.

- **E**indeutige Sprachbezeichnung

  Variablennamen und Kommentare wurden ausschließlich in englischer Sprache verfasst.

- **G**leichmäßige Einrückung

  Jede weitere Verschachtelungstiefe wurde jeweils durch zwei Leerzeichen eingerückt.

Tabellenverzeichnis

Literaturverzeichnis

[ABS00] Saarland University and AbsInt Angewandte Informatik GmbH: *aiSee Graph Visualization User Documentation*, Version 2.00, 1. September 2000

[ABS01] Saarland University and AbsInt Angewandte Informatik GmbH: *AbsInt: Advanced Compiler Technology for Embedded Systems*, 2001, http://http://www.absint.de

[AG98] A.W. Appel, M. Ginsburg: *Modern Compiler Implementation in C*, Cambridge University Press, Cambridge, United Kingdom, 1998

[ARM01] Advanced RISC Machines Ltd (ARM): *ARM - The Architecture for the Digital World*, 2001, URL: http://www.arm.com

[ARM94] Advanced RISC Machines Ltd (ARM): *ARM7 Data Sheet*, Dokument-Nr: ARM DDI 0020C, Dezember 1994

[ARM95a] Advanced RISC Machines Ltd (ARM): *ARM7TDMI Data Sheet*, Dokument-Nr: ARM DDI 0029E, 1995

[ARM95b] Advanced RISC Machines Ltd (ARM): *An Introduction to Thumb*, Version 2.0, März 1995

[ARM99] Advanced RISC Machines Ltd (ARM): *ARM7TDMI Technical Reference Manual*, Dokument-Nr: ARM DDI 0084E, März 1999

[ATM99] Advanced RISC Machines Ltd (ARM): *ATMEL: ARM7TDMI Embedded RISC Microcontroller*, 1999

[ASU88] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman: *Compilerbau*, Addison-Wesley (Deutschland) GmbH, 1988

[BA00] Steven Bashford: *Constraintbasierte Codegenerierung für eingebettete Prozessoren*, Dissertation, Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, Technische Informatik, 2000

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

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

[BGS94] D. Bacon, L. Graham, O. Sharp: *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys, Vol. 26, No. 4:345-420, December 1994

[CH98] Keith D. Cooper and Timothy J. Harvey: *Compiler-Controlled Memory*, Eighth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, October 1998

[DGS] Evely Duesterwald, Rajiv Gupta, Mary Lou Soffa: *A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations*, Proc. ACM SIGPLAN Conference on Programming Languages Design and Implementation, pages 68-77, June 1993

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

[FLW01] Markus Fiesel, Markus Lorenz und Lars Wehmeyer: *GeLIR Generic Low-Level Intermediate Representation - Introduction and Interface*, Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, Technische Informatik, 2001, URL: http://ls12-www.cs.uni-dortmund.de/research/gelir/

[FRASER92] C.W. Fraser, D.R. Hanson, T.A. Proebsting: *Engineering a Simple, Efficient Code Generator Generator*, ACM Letters on Programming Languages and Systems, vol. 1, no. 3, pages 213-226, June 2000

[GAIS] Gaisler Research *Gaisler Research Web Page*, URL: http://www.gaisler.com

[KIL77] G.A. Kildall: *A unified approach to global program optimization*, Conference Recording of the First ACM Symposium on Principles of Programming Languages, pages 194-206, October 1973

[KND96] D.J. Kolson, A. Nicolau, N. Dutt: *Elimination of Redundant Memory Traffic in High-Level Seynthesis*, Computer-Aided Design of Integrated Circuits and Systems, Vol. 15, No. 11, November 1996

[KU77] J.B. Kam, J.D. Ullman: *Monotone data flow analysis frameworks*, Acta Inf. 7, 305-317, 1977

[KR88] Brian W. Kernighan, Dennis M. Ritchie: *The C programming language, Second Edition*, Prentice Hall, Englewood Cliffs N.Y., 1988

[LEE01] Bo-Sik Lee: *Vergleich von Caches und Scratch-Pad Speichern* Diplomarbeit, Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, Technische Informatik, 2001

[LEO01] Gaisler Research: *The LEON Processor User's Manual*, Gaisler Research, Version 2.3.3, May 2001, URL: http://www.gaisler.com

[LEU01] Rainer Leupers: *LANCE - Retargetable C compiler*, Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, URL: http://ls12-www.cs.uni-dortmund.de/lance/

[M93] V. Maslov: *Lazy Array Data-Flow Dependency Analysis*, Technical Report CS-TR-3110.1, University of Maryland, Collage Park CS, July 1993

[MAR00] P. Marwedel: *Skript zur Vorlesung Rechnerarchitektur*, Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, Technische Informatik, April 2000

[MF01] Markus Fiesel: *XML-basierte generische Low-Level Zwischendarstellung für eingebettete Compiler (XeLIR)* Diplomarbeit (noch nicht veröffentlicht), Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, Technische Informatik, 2001

[MH01] Martin Horst: *Schleifenoptimierungen zur Ausnutzung paralleler Rechenwerke von Prozessoren der M3-DSP Plattform* Diplomarbeit (noch nicht veröffentlicht), Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, Technische Informatik, 2001

[MUC97] Steven S. Muchnick: *Advanced Compiler Design Implementation*, Morgan Kaufmann Publishers, Juli 1997

[R91] B. R. Rau: *Data Flow and Dependency Analysis for Instruction Level Parallelism* Lecture Notes in Computer Science, Vol. 589, p. 236-250, 1991

[RS00] Rüdiger Schwarz: *Reduktion des Energiebedarfs von Programmen für den ARM-Prozessor durch Registerpipelining* Diplomarbeit, Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, Technische Informatik, 2000

[SPARC99] SPARC International, Inc.: *The SPARC Architecture Manual*, Revision SAV080SI9308, 1999

[SS00] S. Steinke, R. Schwarz, L. Wehmeyer, P.Marwedel: *Low Power Code Generation for a RISC Processor by Register Pipelining*, Technical Report, Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, Technische Informatik, August 2000

[SW01] P. Marwedel, S. Steinke, L. Wehmeyer: *encc Home*, Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, URL: http://ls12-www.cs.uni-dortmund.de/research/encc

[WZ91] Mark N. Wegman und F. Kenneth Zadeck: *Constant Propagation with Conditional Branches*, IBM T.J. Watson Research Center, ACM Transactions on Programming Languages and Systems, Vol. 13, No. 2, April 1991, Pages 181-210

[XML01] World Wide Web Consortium W3C: *Extensible Markup Language (XML)*, World Wide Web Consortium, URL: http://www.w3.org/XML/

[ZOB01] Christoph Zobiegala: *Energieeinsparung durch compilergesteuerte Nutzung eines On-Chip-Speichers* Diplomarbeit, Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, Technische Informatik, 2001