Architekturunabhängige Quellcodeoptimierung durch Mustererkennung

Jacek Jakubowski

14. April 2002

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit der Programmoptimierung auf der Quellcodeebene. Die zugrunde liegende Methode umfaßt das Erkennen der optimierungswürdigen Codeabschnitte im Quelltext (Mustererkennung) und die anschließende Transformation des Codes in eine optimierte Form. Es wird eine Übersicht gegeben über verschiedene Stufen der Optimierung. Als Unterscheidungskriterium wird die Abstraktionsebene des zugrunde liegenden Modells angenommen. Ausgewählte Transformationen des Quellcodes werden detailliert vorgestellt. Ferner wird ein Tool vorgestellt, mit dem sich Quellcodemuster formulieren, sowie Quellcodetransformationen anwenden lassen. An einem realen Beispiel werden anschließend die Transformationen durchgeführt und bewertet.

Inhaltsverzeichnis

1 Einführung

Eine enorme Verbreitung der sog. eingebetteten Systeme *(embedded systems)* in unserem Alltag impliziert eine intensive Suche nach optimalen Methoden zu deren Entwicklung und Herstellung. In der letzten Dekade hat sich ein deutlicher Trend zum Einsatz von programmierbaren Prozessoren in den eingebetteten System entwickelt. An Stelle von anwendungsspezifischen, hochoptimierten ASIC's *(application specific integrated circuit)* werden zunehmend flexibel programmierbare Prozessoren eingesetzt. Die Änderung der Vorgaben bzw. Anforderungen an ein eingebettetes System erzwingen, beim Einsatz von programmierbaren Prozessoren, keine Neuentwicklung der Hardware. Es wird lediglich das Programm geändert, neu compiliert und wieder - z.B. als Firmware - geladen. Gleichzeitig bringen die Prozessoren den weiteren Vorteil der Wiederverwendbarkeit von vordefinierten Systemkomponenten mit sich *(core processors)*. Sie sind in größeren Systemen *(systems on a chip)* als bereits optimierte und getestete Module einsetzbar.

Damit ist bei der Entwicklung eines Systems der Aspekt der Softwareentwicklung in den Vordergrund gerückt. Die meisten Anwendungen für eingebettete Systeme waren und werden immer noch in Assembler programmiert [18]. Einerseits sind die Anwendungen von relativ geringer Komplexität. Es läuft meist nur eine Anwendung, die auch nur einen Algorithmus implementiert. Andererseits haben solche Systeme sehr hohe Anforderungen an das Zeitverhalten. Man denke nur an die digitale Signalbearbeitung *(DSP - digital signal prcessing)* oder andere Realzeitanwendungen wie MPEG-Encoder. Darüber hinaus müssen die Programme mit einem sehr knappen Speicherangebot auskommen. Der eigentliche Code, im ROM untergebracht, soll sich durch seine Kompaktheit auszeichnen. Die Zugriffe auf die Datenbereiche im RAM sollen den ggf. vorhandenen Cache optimal nutzen. Dies hat den Einatz von Assemblerprogrammierung gerechtfertigt, ja sogar nötig gemacht.

Die besten Ergebnisse werden immer noch mit aufwendiger manueller Optimierung durch den Entwickler erzielt. Um jedoch den ständig kürzer werdenden Entwicklungszyklen zu begegnen, wird immer häufiger eine Hochsprache (meistens C) eingesetzt. Leider sind die verfügbaren C-Compiler mit der Generierung des optimalen Codes, wie er durch die Assemblerprogrammierung entstehen kann, überfordert. Nicht selten übersteigt die Codegröße und die Geschwindigkeit (Zyklenanzahl) der mittels Compilern generier-

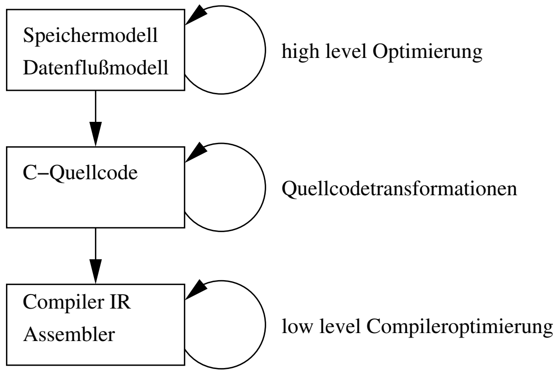


Abbildung 1: Optimierung im Designfluß

ten Programme um ein Mehrfaches die der direkt in Assembler codierten [20]. Große Zyklenzahl und schlechte Nutzung des Speichers (geringe Lokalität, große Datenbereiche) wirken direkt auf den Energieverbrauch eines Systems, welcher ein entscheidendes Kriterium für ein eingebettetes System darstellt. Aus diesem Grund richtet sich ein großer Teil aktueller Arbeiten auf die Methoden der Optimierung von Codegenerierung in den C-Compilern. Die wesentlichen Ziele sind einerseits die Effizienz - Garantie der Realzeiteigenschaft - und andererseits die Minimierung des Energiebedarfs. Dies sind jedoch Vorgaben, die oft zueinander im Widerspruch stehen. Ferner finden die Optimierungen in Compilern, im Gegensatz zu der Optimierung im Quellcode, auf einer niedrigeren Abstraktionsstufe statt. Mit zunehmender bzw. abnehmender Abstraktion in dem zugrunde liegenden Maschinenmodell nimmt die Architekturunabhängigkeit des Modells entsprechend zu bzw. ab. In Compilern werden Optimierungsmöglichkeiten genutzt, die sich aus den Eigenschaften der konkreten Zielarchitektur ergeben. Wie im Kapitel 2 kurz und ohne Anspruch auf Vollständigkeit vorgestellt wird, handelt es sich um Optimierungen, die nur sinnvoll auf der Assemblerebene durch den Compiler während der Codegenerierung durchgeführt werden können. Die Abbildung 1 veranschaulicht den Zusammenhang zwischen einer Entwicklungsstufe und der Optimierung.

Das Thema dieser Arbeit fußt auf der Annahme, daß es, komplementär zu den Möglichkeiten der Optimierung durch den Compiler, Optimierungen gibt, die erfolgreich auf der Ebene des Quellcodes angewendet werden können[1]. Die Optimierung auf der Quellcodeebene mag insofern unnötig er-

[1] Damit ist nicht der sog. gute Programmierstil gemeint. Denn egal, wie hoch die Ab-

scheinen, als daß der Zweck einer Hochsprache in der Ausdrucksstärke und der höheren Abstraktionsstufe liegt. Ein Programm soll einen Algorithmus korrekt beschreiben (kodieren). Weitere Effizienzgewinne sollen durch den Compiler erzielt werden. In der Arbeit wird gezeigt, daß eine methodische, nach Möglichkeit automatisierte Anwendung von Quellcodetransformationen ein großes Potential zur Verbesserung der Programmgüte beinhaltet. Durch die höhere Abstraktionsstufe auf der Quellcodeebene ergeben sich Möglichkeiten, globale Zusammenhänge innerhalb eines Programms zu analysieren und für die Optimierung auszunutzen.

1.1 Ziele der Arbeit

Als Ziel der Arbeit sollen, neben der Darstellung der Möglichkeiten und Methoden für die Quellcodeoptimierung, ausgewählte Quellcodetransformationen vorgestellt werden. Es soll untersucht werden, welches Optimierungspotential sich bei der Anwendung dieser Transformationen ergibt. Ferner sollen in einer passenden Form Muster formuliert werden, mit Hilfe derer die automatische Suche im Code ermöglicht wird. Für die Anwendung der Transformationen müssen bestimmte Bedingungen erfüllt sein, um die Korrektheit des Programms zu wahren. Das Prüfen dieser Bedingungen soll durch eine Analyse der als Muster identifizierten Codebereiche erfolgen. Es soll gezeigt werden, welche Analysen welche Fragen beantworten müssen. Die formale Darstellung der Muster wird in den jeweiligen Tools realisiert - wovon wir eines im Kapitel 5 vorstellen.

1.2 Begriffe

Die in dem Thema enthaltenen zentralen Begriffe der Arbeit sollen zunächst erläutert werden. Die Architekturunabhängigkeit der hier vorgestellten Optimierungen rührt ganz klar aus der Anwendung einer Hochsprache[2]. Ohne Annahmen über spezielle Architektureigenschaften, wie z.B. Multiprozessorsysteme, spezielle Prozessortypen, etc., werden wir uns ganz auf die Semantik

straktionssebene auch sei, es gibt immer Möglichkeiten, ein Problem besser oder schlechter im Programmcode auszudrücken. Es geht viel mehr um die systematische Anwendung von bestimmten Methoden auf den Quellcode fertiger Programme.

[2] Ob C wegen der vorhandenen Maschinennähe als Hochsprache im weitesten Sinne angesehen wird, sei erst mal dahin gestellt. Auf jeden Fall ist C (ANSI-C) standardisiert und für alle Plattformen bzw. Architekturen in Form von Compilern verfügbar. Diese Eigenschaften machen C für unsere Zwecke zur Hochsprache.

der Sprache ANSI-C konzentrieren. Die Annahmen über die Architektur beschränken sich auf das der Sprache C zugrunde liegende Maschinenmodell. Dieses beinhaltet eine CPU als Blackbox und den linearen Speicher. Die eigentliche Optimierung beruht auf dem Erkennen und Ersetzen von Codefragmenten durch anderen Code. Weiterhin kann zusätzlicher Code erzeugt bzw. zum Zwecke der Optimierung Code entfernt werden. Wir fassen diese Operationen unter den Begriff der Quellcodetransformation zusammen. Bei einer Quellcodetransformation darf selbstverständlich die Semantik des Programms nicht verändert werden. Die Transformationen werden in mehreren Zyklen vor der Compilierung durchgeführt. Die Programmteile, die durch Transformation bearbeitet werden können, entsprechen bestimmten Regeln. Durch „scharfes Hinsehen" ist es möglich, sie zu finden und anschließend die Transformation anzuwenden. Um den Prozeß weiter zu verbessern, wäre es sinnvoll, diesen Vorgang zu automatisieren. Die Methode hierzu kann z.B. die Mustererkennung sein. Die Ausdrücke, Anweisungen und Kontrollstrukturen in einem C-Programm können Muster enthalten oder auch in bestimmten Mustern vorkommen bzw. Muster bilden. Die Muster formal zu erfassen und sie zu beschreiben ist notwendig, um in einem Tool die Mustererkennung implementieren zu können.

Folgendes Beispiel veranschaulicht den von uns verwendeten Musterbegriff anhand der Codefragmente in C.

Ein Muster beschreibt den zu suchenden Quellcode, hier z.B. als eine Zuweisung eines arithmetischen Ausdruckes, in dem zwei Variablen miteinander verknüpft werden (s. Bsp. 1). Die Zuweisung kann von beliebigen Codesequenzen umgeben sein. Der Ausdruck ist seinerseits innerhalb eines Schleifenkörpers eingebettet.

Beispiel 1 *Quellcode, der als Muster beschrieben werden kann*

```
for(i=0; i<MAX; i++)
{...                        /* SEQ1 - Sequenz von bel. Anweisungen */
 ... = a + b;
 ...}                      /* SEQ2 - Sequenz von bel. Anweisungen */
```

Unter Mustererkennung verstehen wir demnach das Identifizieren von Codeabschnitten, die Kontrollstrukturen, Anweisungen oder Ausdrücke darstellen. In dem vorigen Beispiel ist es z.B. die Schleife mit genau diesen

Eigenschaften (Iterationsrichtung, Iterationsweite, Schrittweite), die in dem Schleifenkörper eine Anweisung von genau dieser Form enthält. Ein Muster soll darüber hinaus eine allgemeine Spezifizierung dieser Eigenschaften zulassen. In unserer Schleife würde z.B. die Festlegung des Musters auf eine Schrittweite von $+1$ auch ausbleiben können. Das Muster würde in dem Fall nur das Vorkommen einer Schrittaktion vorgeben.

Als Ergebnis der Suche sollen sowohl die Schleife als auch der gefundene Ausdruck (in unserem Beispiel die Zuweisung, die Schleife und die ggf. vorhandenen Codesequenzen SEQ1 und SEQ2) als eindeutig identifizierbare Entitäten (Codeobjekte)[3] verfügbar sein. Damit können weitere Eigenschaften dieser Strukturen analysiert werden, wie etwa die Frage nach der Iterationsweite der gefundenen Schleife. Ferner können die identifizierten Objekte durch Codetransformation manipuliert werden.

Die Transformation beschreibt, wie der Quellcode verändert wird. Ausgehend von den in der Mustererkennung identifizierten Objekten wie Schleife, if-Anweisung, Ausdruck, wird neuer Quellcode generiert. Die Schleife aus unserem Beispiel würde danach wie folgt aussehen können.

Beispiel 2 *Quellcode nach einer Transformation*

```
c = a+b;
for(i=0; i<MAX; i++)
{...                    /* SEQ1 - Sequenz von bel. Anweisungen */
 ... = c ;              /* vorher a+b */
 ...}                   /* SEQ2 - Sequenz von bel. Anweisungen */
```

Als Transformation verstehen wir das Verändern eines Codeobjekts, das Entfernen eines Codeobjekts oder auch das Einfügen neu generierter Codeobjekte. Nach der Transformation soll der neue Quellcode die gleiche Semantik wie der ursprüngliche besitzen. Derartige Codeveränderungen dürfen nicht die Programmkorrektheit verletzen. Da es schwierig wäre, in der Musterspezifikation komplexere Bedingungen an die Codeobjekte auszudrücken, ist ein weiterer Prozeß nach der Mustererkennung, der noch vor der Transformation liegt, notwendig. In einem solchen Schritt wird die Erfüllung bestimmter Bedingungen durch die Codeobjekte verifiziert.

[3] Nicht im Sinne der OO-Programmierung. Es handelt sich lediglich die Elemente der formalen Beschreibung eines Algorithmus (Schleife, Bedingung, Operation bzw. Sequenz von Operationen oder ein Ausdruck).

Die Bedingung an ein Muster stellt die korrekte Anwendung einer Transformation sicher.

Beispiel 3 *Bedingung an die Codeobjekte*

```
SEQ1 und SEQ2 dürfen keine Definitionen von a bzw. b enthalten.
```

Die Erfüllung dieser Bedingung durch die Codeobjekte SEQ1 und SEQ2 entscheidet über die Anwendung der Codetransformation. Es sei anzumerken, daß die Analysen, die zur Verifikation nötig sind, sehr aufwendig sein können. Es kann sich um Datenfußanalysen handeln, die ein Profiling des Programms erfordern. Sollten z.B. die beiden Sequenzen Funktionsaufrufe enthalten und wären **a** oder **b** Zeigervariablen, müßte die Suche nach den eventuell vorhandenen Definitionen von Inhalten in **a** und **b** im Code dieser Funktionen fortgeführt werden. Diese Suche kann sich als beliebig komplex erweisen.

1.3 Übersicht über die Arbeit

Im Kapitel 2 werden grob die Optimierungsmthoden bei der Codegenerierung in Compilern vorgestellt. Zu den wichtigsten Problemen zählen etwa die Registerallokation, die Addressgenerierung oder die Berechnung der Zugriffssequenzen. Das Kapitel 3 beschreibt Optimierungsmöglichkeiten im Quellcode, welche die Minimierung des Energieverbrauchs durch Berücksichtigung von Speicherhierarchien im Quellcode zum Ziel haben. Diese Methoden bilden den Hintergrund für eine Reihe von Quellcodetransformationen, die im Kapitel 4 vorgestellt werden, und deren Automatisierung mit Hilfe von Mustererkennungstechniken den zentralen Aspekt der vorliegenden Arbeit darstellt. Im Kapitel 5 wird ein Tool (*code transformation tool*; CTT) vorgestellt, das die Möglichkeit bietet, Muster, Transformationen und die Bedingungen dafür zu formulieren. Als praktische Aufgabe der Arbeit soll die Beschreibungssprache des CTT um Elemente erweitert werden, die die Anwendung der in Kapitel 4 vorgestellten Transformationen unterstützen. Diese Erweiterungen werden im Kapitel 6 erläutert. Zum Schluß sollen im Kapitel 7 die Ergebnisse der Arbeit vorgestellt werden.

2 Aspekte der Codeoptimierung für eingebettete Systeme

In den folgenden Abschnitten werden selektiv Optimierungsverfahren vorgestellt, wie sie in Compilern bei der Codegenerierung verwendet werden können. Es soll verdeutlicht werden, daß gewisse Optimierungsmethoden nur von einem Compiler sinnvoll eingesetzt werden können. Abhängig von der Abstraktionsstufe des verwendeten Modells werden wir in den folgenden Kapiteln derartige Optimierungstechniken als *low level* bzw. *medium level* bezeichnen.

Wir sehen allerdings davon ab, eine strikte Trennung zwischen der *low level, medium level* und der im Kapitel 3 vorgestellten *high level* Optimierung zu vollziehen. Viele Optimierungsmethoden - man nehme die *common subexpression elimination* CSE als Beispiel - können von einem Compiler wie auch in der Quellcodeoptimierung eingesetzt werden. Der wesentliche Unterschied besteht in der Abstraktionsstufe, auf der die Optimierung stattfindet. Damit hängt auch die Sicht auf die zugrundeliegende Architektur bzw. das Maschinenmodell zusammen. Diese Sicht entscheidet über die Möglichkeiten, bestimmte Optimierungen anwenden zu können.

Low level Optimierung hängt mit einer konkreten Architektur zusammen. Der Registersatz und der Befehlssatz sind die Hauptelemente der Maschinenmodelle. Es ist klar, daß eine lokale Umordnung von zwei Maschinenbefehlen zwecks besserer Ausnutzung der Prozessorpipeline am besten bei der Codegenerierung im Compiler direkt auf der Assemblerebene oder in einer architekturabhängigen Zwischendarstellung des Codes (*intermediate representation,* IR) stattfinden sollte. Typische *low level* Optimierungen sind z.B.:

- Registerallokation
- Codeauswahl
- Offset Assignment
- Codekompaktierung

Auf einer höheren Ebene arbeiten Optimierungen, deren Modelle nicht direkt aus der Beschreibung der Architektur bzw. der Befehlssätze stammen. Hierzu zählen verschiedene Graphendarstellungen für Sachverhalte wie Daten-oder Kontrollflüsse (CFG *conrol flow graph,* DFG *data flow graph*). Die Idee des *basic blocks* (BB) ist für diese Modelle ebenfalls grundlegend. Die Optimierungen, die mit diesen Modellen arbeiten sind z.B.:

- CSE
- Function Inlining
- Dead Code Elimination

Weitere Optimierungsschritte, die als *pass* im Compiler vorkommen (z.B. der GCC 3.x [7]) sind: *static single assignment* (SSA), *dead code elimination,* Schleifenoptimierung *(unrolling, code hoisting).* Es soll auch erwähnt werden, daß der GNU C-Compiler zwar viele Prozessoren für eingebettete Systeme als Plattform unterstützt, jedoch wenig von deren speziellen Eigenschaften bei der Codegenerierung bzw. Optimierung nutzt. Ebenso werden spezielle Aspekte der Optimierung, wie z.B. Energieverbrauch oder Realzeitverhalten, nicht explizit berücksichtigt.

2.1 Low Level Optimierungen

2.1.1 Modellierung

Als Modell für derartige *low level* Optimierung wird eine architekturabhängige IR (*intermediate representation*) verwendet. Als Beispiel sei hier die GeLIR Klassensammlung [17] erwähnt. Eine andere *intermediate representation*, die von dem GCC für die *low level* Repräsentation des Codes verwendet wird, heißt RTL (*register transfer language*). Diese Repräsentation modelliert den Zustand der Registermaschine und dessen Veränderung bei der Ausführung der Instruktionen.

Low Level Modell GeLIR

Für die Zwecke der *low level* Optimierung ist ein Modell sinnvoll, das die Struktur eines Maschinenprogramms widergibt und die Elemente einer Architektur abbildet. Wir stellen als Beispiel kurz die GeLIR C++ Klassensammlung vor. In dem GeLIR Modell wird die Programmstruktur durch die Klassenbeziehung 'enthalten in' realisiert, mit je einer Klassenhierarchie für die Programmdarstellung und für die Zielarchitektur. Die Hierarchie eines Programms in GeLIR Darstellung sieht wie folgt aus:

- **LirGeLIR** Klasse stellt ein ganzes Programm dar. Sie beinhaltet alle Funktionen des Programms wie auch die globale Symboltabelle.
- **LirFun** Klasse entspricht einer Funktion im Quellcode und enthält Basisblöcke. Sie enthält auch die lokale Symboltabelle.
- **LirBB** ist ein Codeblock der als ganzes ausgeführt wird. Dieser Block enthlält keine JUMP Instruktionen zum verlassen des Blockes.
- **LirMI** stellt eine Instruktion dar. Diese enthält u.U. mehrere parallel ausgeführte Operationen.
- **LirMO** ist eine Maschinenoperation wie ADD, LOAD oder JUMP.

Die Klassen der Architekturrepräsentation beschreiben:

- Operationen (Argumentenzahl, verwendete Register, Funktionseinheiten) - **LirOperation**
- Ressourcen (Register, Funktionseinheiten, Instruktionstypen) - **LirResources**
- Typen - **LirType**

RTL - Low Level Darstellung des GCC

Die RTL (*register transformation language)* Repräsentation des Codes wird von dem GCC für *low level* Optimierung und für die Generierung des Codes für die Zielmaschinensprache benutzt. Die Beschreibung der Zielarchitektur und des Befehlssatzes wird beim Compilieren des Compilers selbst erstellt. Die RTL kennt fünf Objektarten:

Ausdrücke sind die Grundobjekte der RTL Darstellung. Sie enthalten einen Typ, der angibt, welches Element von dem Ausdruck beschrieben wird (Konstante, Variable, Funktionsaufruf) und die Parameter des Ausdrucks.

Integers sind Zahlen in der Dezimaldarstellung.

Breite Integers sind Zahlen mit einer erweiterten Bitdarstellung.

Strings sind Zeichenketten wie in C; char *.

Vectors sind Arrays von Ausdrücken

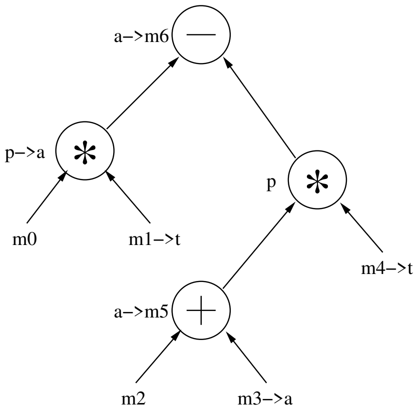
Beispiele für Ausdrücke in RTL

- *(const_int i)*: ist ein Ausdruck für eine Konstante mit dem Parameter i.
- *(const_string str)*: ist eine konstante Zeichenkette.
- *(reg:m n)*: bezeichnet ein Register. Wenn n klein ist, steht der Ausdruck für ein reales Register. Bei großen n wird zuerst ein Pseudoregister angenommen. Dieses wird dann bei der Registerzuordnung ggf. ersetzt.
- *(mem:m addr alias)*: Eine Speicherreferenz der Größe m ab Adresse *addr*.
- *(parallel [(set (reg:SI 1) (mem:SI (reg:SI 1))) (set (mem:SI (reg:SI 1)) (reg:SI 1))])*: beschreibt eine parallele Ausführung der Seiteneffekte. In diesem Fall werden die Werte des Registers 1 und der Speicherzelle, die R1 addressiert, vertauscht

RTL ist ein Beispiel für eine sehr spezialisierte und maschinenabhängige Programmdarstellung. Sie eignet sich nicht für den Einsatz außerhalb des Compilers, da sie schwierig zu parsen bzw. zu generieren ist.

2.1.2 Codegenerierung

Eine markante Eigenschaft der Prozessoren in eingebetteten Systemen ist die Inhomogenität ihrer Registersätze. Oft gibt es Register mit Eigenschaften, die bestimmte Operationen besonders unterstützen, wie z.B. die Fast-Fourier-Transformation oder spezielle Addressierungsarten. Die optimale Auswahl der Register ist daher ein wichtiger Optimierungsansatz. Es existieren optimale Algorithmen für die Codeauswahl bei homogenen Registersätzen [19]. Die Codeauswahl und die Registerallokation bei einem inhomogenen Registersatz können unter bestimmten Bedingungen effizient berechnet werden. Dieses Problem rührt aus den speziellen Eigenschaften der Instruktionssätze in DSP's her. So erwarten manche Instruktionen ihre Operanden in bestimmten Registern. Andere benutzen für ihre Ergebnisse nur vorgegebene Register. Dies schränkt die Flexibilität der Registerauswahl während der Codegenerierung erheblich ein. Auch sind nicht alle Register mit allen



(a)

(b)		(c)		(d)	
lt	m1	lt	m4	lac	m3
mpy	m0	lac	m3	add	m2
pac		add	m2	sacl	m5
sacl	**m7**	sacl	m5	lt	m1
lac	m3	mpy	m5	mpy	m0
add	m2	lt	m1	pac	
sacl	m5	pac		lt	m4
lt	m4	**sacl**	**m7**	mpy	m5
mpy	m5	mpy	m0	spac	
lac	**m7**	pac		sacl	m6
spac		**lt**	**m7**		
sacl	m6	mpyk	1		
		spac			
		sacl	m6		

Abbildung 2: Codeauswahl für einen Ausdruck: (a) Ein Objektbaum nach der Zuordnung von Instruktionsmustern (aus dem Befehlssatz der TMS320C25 CPU) zu den Ausdrücken (b) postorder Codeauswahl (c) preorder Codeauswahl (d) optimale Codeauswahl.

anderen Registern im Datenpfad verbunden, d.h. es gibt keine Instruktionen, um zwischen diesen Registern Daten auszutauschen. Aus diesem Grund ist in vielen Fällen immer der Umweg über einen Hauptspeicherzugriff oder ein Zwischenregister notwendig. Es muß dann ggf. ein *spilling* durchgeführt werden, um die Daten im Zwischenregister zu erhalten.

Im Bild 2 wird beispielhaft dargestellt, wie drei mögliche Codesequenzen aus einem Ausdrucksbaum erzeugt werden. Für den Ausdrucksbaum (a) werden die Sequenzen (b) bzw. (c) nach einem Standardverfahren generiert. Dabei werden in einem Post- bzw. Preorderdurchlauf die Unterbäume verarbeitet, bevor für den Wurzelknoten der Code ausgewählt wird. In diesen beiden Sequenzen treten Zugriffe auf die Speicherzelle **m7**, über die die Speicherspills durchgeführt werden. Die Sequenz (d) wurde nach einem auf dem RTG *(register transfer graph)* Kriterium basierenden Algorithmus erzeugt, der die Sequenz frei von Speicherspills macht. Daher wird in (d) nicht mehr auf **m7** zugegriffen.

Definition 1 (RTG) *Ein RTG ist ein gerichteter Graph. Jeder Knoten repräsentiert möglichen Speicherort - Register, Speicher - im Datenpfad einer Instruktion. Eine Kante zwischen den Knoten r1 und r2 ist mit einer Instruktion markiert, die r1 als Quelloperanden und r2 als Zieloperanden hat.*

Definition 2 (RTG Eigenschaft) *Ein Instruktionssatz besitzt die RTG Eigenschaft, wenn: (a) Es für alle Knoten r_3, die als Ziel für eine Operation l dienen, die wiederum zwei Knoten r_1 und r_2 als Quelloperanden hat, mindestens einen Zyklus zwischen den Knoten r_1 und r_2 gibt. (b) Alle Zyklen zwischen den Knoten r_1 und r_2 enthalten einen Speicherknoten.*

Durch Umordnung der Instruktionssequenz *(instruction scheduling)* kann also die Anzahl der *spills* minimiert werden. Für Instruktionssätze oder ISAs *(instruction set architecture),* die das RTG Kriterium erfüllen, ist die Berechnung einer optimalen Instruktionsauswahl, die keine Speicherspills enthält, in Linearzeit möglich.

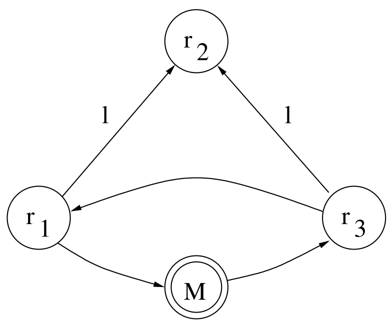


Abbildung 3: RTG Eigenschaft

Mit Hilfe eines RTG kann für einen Befehlssatz die Beziehung des Datenflusses zwischen den Registern modelliert werden. Die Knoten des Graphen repräsentieren die Register. Die doppelt umrandeten Knoten stellen Speicher bzw. gleichartige Registersätze dar. Schließlich entsprechen die gerichteten Kanten den Instruktionen. Wie im Bild 3 gezeigt wird, ist eine Kante **l** zwischen zwei Knoten $\mathbf{r}_1$ und $\mathbf{r}_3$ eine Instruktion, die $\mathbf{r}_1$ als Quelle (Operand) und $\mathbf{r}_3$ als Ziel (Register für das Ergebnis) hat. Der Speicherknoten fungiert als mögliche Zwischenablage für Daten während der Ausführung von konkurrierenden Instruktionen (sog. *allocation deadlock*). Die detaillierte Beschreibung des Algorithmus und dessen Optimalität wird in [2] gegeben.

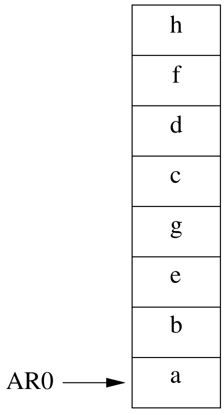
2.1.3 Offset Assignment

Eine Optimierung der Speicherzuordnung basiert auf der besonderen Eigenschaft der Adressregister in der AGU *(address generation unit)*, beim Zugriff ein automatisches In- bzw. Dekrementieren durchzuführen. Diese Eigenschaft ermöglicht einen sehr kompakten Code beim Zugriff auf sequentiell organisierte Datenstrukturen. Das Problem der Offsetzuordnung wird z.B. in [4] und in [15] behandelt.

Zu den grundlegenden Elementen einer Architektur, für die eine Opti-

```
e = a + b;
b = b + g;
c = c + d + f;
a = h – c;
c = b – c + a;
```

(a)



(b)

```
LDAR   AR0,&a    ; a
LOAD   *(AR0)+   ; b
ADD    *(AR0)+   ; e
STOR   *(AR0)–   ; b
LOAD   *(AR0)
ADAR   AR0,2     ; g
ADD    *(AR0)
SBAR   AR0,2     ; b
STOR   *(AR0)
ADAR   AR0,3     ; c
LOAD   *(AR0)+   ; d
ADD    *(AR0)+   ; f
ADD    *(AR0)
SBAR   AR0,2     ; c
STOR   *(AR0)
ADAR   AR0,3     ; h
LOAD   *(AR0)
SBAR   AR0,3     ; c
SUB    *(AR0)
SBAR   AR0,4
STOR   *(AR0)+   ; a
LOAD   *(AR0)    ; b
ADAR   AR0,3
SUB    *(AR0)    ; c
SBAR   AR0,4
ADD    *(AR0)    ; a
ADAR   *AR0,4
STOR   *(AR0)    ; c
```

(c)

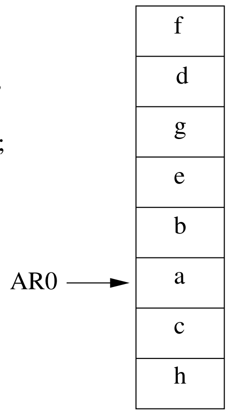
Abbildung 4: (a) Codesegment (b) Offsetzuordnung (c) Assemblercode

mierung der Offsetzuordnung durchgeführt wird, zählen die Adressregister, die Autoinkrementinstruktionen und die Offsetregister. Die Adressierung des Speichers erfolgt indirekt mit Hilfe der Adressregister, spezielle Instruktionen erlauben während des Zugriffs eine automatische Inkrementoperation auf den Adressregistern, die ohne zusätzliche Kosten, da parallel, ausgeführt wird. Ein beliebiger Offset der Inkrementoperation muß vorher in dem Offsetregister gesetzt werden. Es ist klar, daß die Anordnung der Variablen im Speicher sich direkt auf die Anzahl der Änderungen des Adressregisters auswirkt. Ziel der Optimierung ist es, eine Anordnung der Variablen im Speicher zu finden, bei der für eine vorgegeben Zugriffsreihenfolge, die Anzahl der Änderungen des Adressregisters minimal ist.

Als Beispiel betrachten wir ein Stück eines C Programms in der Abbildung 4. Wir setzen voraus, daß es sich um eine akkumulatorbasierte Maschine handelt. Jede Instruktion hat als Operanden den Akkumulator und eine über einen Register indirekt adressierte Speicherzelle. Der Adresregister $\mathbf{A}R0$ dient dem Speicherzugriff. Der Adressregister hat ferner die Eigenschaft, während einer Operation automatisch um eins inkrementiert bzw. dekrementiert werden zu können. Da der Wert der Inkrementperatin immer eins beträgt ist in diesem Fall kein zusätzlicher Offsetregister nötig.

```
e = a + b;
b = b + g;
c = c + d + f;
a = h – c;
c = b – c + a;
```

(a)



(b)

```
LDAR  AR0,&a    ; a
LOAD  *(AR0)+   ; b
ADD   *(AR0)+   ; e
STOR  *(AR0)–   ; b
LOAD  *(AR0)
ADAR  AR0,2     ; g
ADD   *(AR0)
SBAR  AR0,2     ; b
STOR  *(AR0)
SBAR  AR0,2     ; c
LOAD  *(AR0)
ADAR  AR0,5     ; d
ADD   *(AR0)+   ; f
ADD   *(AR0)
SBAR  AR0,6     ; c
STOR  *(AR0)–   ; h
LOAD  *(AR0)+   ; c
SUB   *(AR0)+   ; a
STOR  *(AR0)+   ; b
LOAD  *(AR0)
SBAR  AR0,2     ; c
SUB   *(AR0)+   ; a
ADD   *(AR0)+   ; c
STOR  *(AR0)
```

(c)

Abbildung 5: (a) Codesegment (b) optimale Offsetzuordnung (c) Assemblercode

Außerdem kann mit den **ADAD** bzw. **SBAD** Instruktionen der Wert des Adressregisters um einen beliebigen Wert rauf- oder runtergestzt werden. Die Reihenfolge der Zugriffe auf die Variablen ist durch den Quellcode (a) festgelegt. Das Problem besteht nun darin, die Anordnung der Variablen im Speicher (b) so zu gestalten, daß die Anzahl der **ADAD** und **SBAD** Oprationen im Assemblercode (c) minimiert wird. In dem erzeugten Assemblercode werden 10 **ADAD** bzw. **SBAD** Operationen verwendet. Eine optimale Speicherzuordnung für den gleichen C-Code wird im Bild 5(b) gezeigt. Im Assemblercode im Bild 5(c) sind nur noch sechs **ADAD** bzw. **SBAD** Instruktionen notwendig. Mit zusätzlichen Adressregistern wäre weitere Verbesserung möglich.

Zur Lösung dieser Probleme werden Graphenalgorithmen eingesetzt. Die Graphen, die als Zugriffsgraphen bezeichnet werden, beschreiben die Vorgänger- bzw. Nachfolgerbeziehung zwischen den Variablen in einer Zugriffssequenz. Es werden Heuristken eingesetzt, um die oft NP-vollständigen Graphenprobleme zu lösen. Basierend auf diesen Ansätzen werden in [11] heuristische Verfahren zur Nutzung von parallelen AGU's vorgestellt. Dabei wird das Problem der Registerzuordnung um die Verteilung der Variablen auf verschiedene Speicherbänke erweitert. In diesem Fall sind parallele Speicherzugriffe möglich. Eine Lösung, basierend auf dem Färbproblem eines Graphen,

```
(1) ACCU := u(n−1)
(2) TR := e(n−2)
(3) PR := TR * K2
(4) TR := e(n−1)
(5) e(n−2) := e(n−1)
(6) ACCU := ACCU + PR
(7) PR := TR * K1
(8) TR := e(n)
(9) e(n−1) := e(n)
(10) ACCU := ACCU + PR
(11) PR := TR * K0
(12) ACCU := ACCU + PR
(13) u(n) := ACCU
```

(a)

```
(1) ACCU := u(n−1)
(2) TR := e(n−2)
(3) PR := TR * K2
(4) e(n−2) := e(n−1) ||
    TR := e(n−1) ||
    ACCU := ACCU + PR
(5) PR := TR * K1
(6) e(n−1) := e(n) ||
    ACCU := ACCU + PR ||
    TR := e(n)
(7) PR := TR * K0
(8) ACCU := ACCU + PR
(9) u(n) := ACCU
```

(b)

Abbildung 6: Codekompaktierung: (a) vor der Optimierung (b) und danach mit parallel ausgeführen Instruktionen

ist in [16] zu finden.

2.1.4 Codekompaktierung

Bei der Codekompaktierung sollen Instruktionsordnungen gefunden werden, die die parallele Ausführung von Operationen ermöglichen (s. Abb. 6). Manche Instruktionen sind in mehreren Versionen vorhanden, die zusätzliche Seiteneffekte bewirken. Die Muliplikation - in Form der Instruktion *multiply accumulate* (MAC) - kann auch mit anschließender Summierung des Produkts im Akkumulator erfolgen.

In der Abbildung 6(a) ist ein Codeabschnitt mit 13 Instruktionen dargestellt. Durch eine Vertauschung der Instruktionen (4) und (5) bzw. (8), (9) und (10) können diese beiden Gruppen parallel ausgeführt werden. Es sind dann die Schritte (4) und (6) in der Abbildung 6(b), die die parallele Ausführung enthalten. Insgesmt sind in (b) nur noch neun Schritte erforderlich. Dabei wurde eine konkrete Eigenschaft in dem Instruktionssatz des TMS320C25 Prozessors ausgenutzt, zu dem der Codebeispiel gehört.

Um diese Optimierung anzuwenden, wird das IP (*integer programming)* Verfahren benutzt. Das IP bietet die Möglichkeit, eine Instruktionsordnung zu finden, die vorgegebene Bedingungen - z.B. obere Schranke für die Anzahl der Maschinenzyklen - erfüllt bzw. zu entscheiden, daß es keine solche Anordnung gibt [3]. Dazu wird für eine vorgegebene Codesequenz eine Formulierung durch ein IP Gleichungssystem erstellt. Diese enthält Entscheidungsvariablen für die Auswahl einer Operation in einem bestimmten Schritt mit den dazugehörigen Ressourcen. Ferner werden, in Form von Ungleichungen, die Bedingungen an die Reihenfolge der Instruktionen, wie z.B. die Datenabhängigkeiten, formuliert. Das System wird anschließend von einem IP-solver gelöst. Dieser berechnet alle Versionen der Instruktionssequenz und wählt eine, die die vorgegebenen Schranken erfüllt. Andernfalls entscheidet das Programm, daß es für die vorgegebene Bedingung keine passende Instruktionsanordnung gibt.

Da der IP-solver eine NP-vollständige Aufgabe lösen muß, lässt sich das Verfahren nur auf kleine, lokale Codesequenzen anwenden. Der Lösungsraum hängt einerseits von der Größe der Sequenz, andererseits von dem Instruktionssatz der Maschine ab. Mehr Restriktionen an die Ausführung von Instruktionen - wie vorgegebene Register oder Seiteneffekte - erzeugen mehr Bedingungen in der IP Repräsentation. Eine Anwendung von IP auf die Codekompaktierung mit Berücksichtigung der Ausführungszeit als Zyklenvorgabe ist in [12] zu finden.

2.2 Medium Level Optimierung

2.2.1 Modellierung

Mit *medium level* bezeichnen wir Optimierungen, die im Vergleich zu den *low level* Methoden in den verwendeten Modellen abstraktere Begriffe benutzen. Zwei grundlegende Elemente aller *medium level* Modelle sind der Kontrollflußgraph und der Basisblock.

Definition 3 *(Kontrollflußgraph CFG) Ein CFG ist ein Graph G=(N,E,s,e) mit Knotenmenge N und Kantenmenge E. Die Knoten bezeichnen die Instruktionen in einem Programm. s ist ein Startknoten und e ein Endknoten und sie bezeichnen die erste bzw. die letzte Instruktion. Ist n1 direkter Vorgängerknoten von n2 so ist* $(n1, n2) \in E$*. Das bedeutet, daß die Instruktion n2 direkt nach n1 ausgeführt werden kann. s hat keinen Vorgänger und e hat keinen Nachfolger.*

Der CFG beschreibt somit mögliche Ausführungspfade.

Definition 4 *(Basisblock BB) Ein Basisblock ist ein Pfad maximaler Länge in einem CFG. Jeder Knoten, bis auf den Start- und Endknoten, auf dem Pfad hat nur einen Nachfolger und nur einen Vorgänger. Der Startknoten darf mehrere Vorgänger und der Endknoten mehrere Nachfolger haben.*

Ein Basisblock beschreibt einen Ausführungspfad der vom Startknoten bis zum Endknoten komplett ausgeführt wird. Man kann einen Basisblock nicht zwischen dem Start- und Endknoten verlassen bzw. betreten.

Die Modellierungsmöglichkeiten in dem LANCE *frontend-system* basieren auf der Graphendarstellung bzw. dem Basisblock. Es beinhaltet eine IR-Darstellung, in der die Optimierungen, die auf Daten- und Kontrollflußgraphen beruhen, durchgeführt werden können. Die IR der SUIF Bibliothek ist ein anderes Beispiel einer abstrakten IR Darstellung, die schon zum Teil auf der Ebene des Quellcodes angesiedelt ist. Die Repräsentation des Codes als AST (*abstract syntax tree*) eröffnet vielfältige Möglichkeiten der Analyse und der Modellierung. Ferner erlaubt die SUIF IR mit dem Modell der ASTs die Darstellung ganzer Anweisungen einer Programmiersprache.

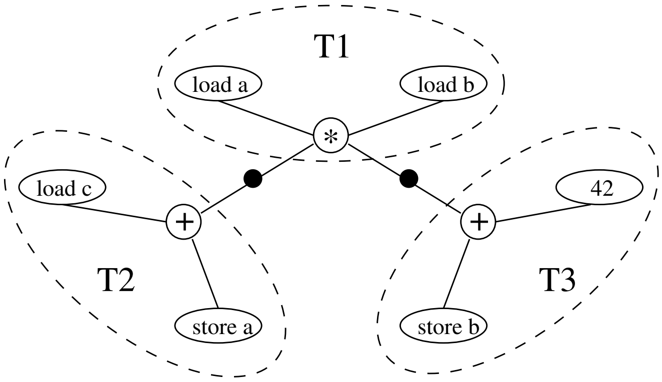
Die CSE Optimierung basiert auf Ausdrücken, die als Entität in dem Quellcode erkannt werden können. Relevant sind also nicht mehr die Maschineninstruktionen und Register, sondern z.B. Ausdrücke in einer C-Anweisung. Diese können als Instruktionsbäume modelliert werden. Die Inlining Optimierung ist ein anderers Beispiel für eine *medium level* Optimierung. Ein Graph, der die Beziehung 'Funktion F_1 ruft Funktion F_2 auf' darstellt und dabei die Codegröße der Funktionen speichert, ist hierfür ein geeignetes Modell.

2.2.2 Common Subexpression Elimination

Diese sehr populäre Optimierungsmethode wird von vielen Compilern für GP (*general purpose*) Architekturen bereits unterstützt. Es handelt sich hierbei um das Erkennen von identischen Unterausdrücken, die in einem Codeabschnitt den Charakter einer Konstanten haben. Die CSE gehört zu den Optimierungen, die auf verschiedenen Abstraktionsstufen in dem Entwicklungsprozeß angewendet werden können. Im Kapitel 4 stellen wir CSE als Quellcodeoptimierung dar. Dieser Abschnitt beschreibt den Einsatz von CSE auf der Compilerebene.

In der Abbildung 7(a) ist der Zusammenhang zwischen drei Instruktionen bei der Anwendung von CSE dargestellt. Das Ergebnis der Instruktion T1 wird als Operand in T2 und T3 benutzt. Um die wiederholte Berechnung von T1 einzusparen, wird das Ergebnis in der Variablen `temp` zwischengespeichert (Abb. 7(b)).

Die Anwendung von CSE bei der Codegenerierung geschieht innerhalb ei-



(a)

```
lt a
mpy b
spl temp
pac
add c
sacl a
lac temp
addk 42
sacl b
```

(b)

```
lt a
mpy b
pac
add c
sacl a
pac
addk 42
sacl b
```

(c)

Abbildung 7: Ein DFT für einen Ausdruck (a) Der Wert von CS zwischengespeichert im Speicher (b) und im Register (c)

nes Basisblockes. Da ein Basisblock immer am Stück ausgeführt wird - es gibt also keine Ein- und Ausgänge innerhalb des Blockes - und ein Basisblock aus einer überschaubaren Sequenz an Instruktionen besteht, ist die Datenflußanalyse eher unkompliziert. Der Basisblock kann hierfür durch einen DFG (*data flow graph*) repräsentiert werden. Im allgemeinen wird der gemeinsame Ausdruck in einer neuen Variablen abgelegt und in den folgenden Ausdrücken benutzt. Die zusätzlichen Speicherzugriffe zum Speichern und wiederholten Laden einer CSE Variablen sollten dabei unter einer Kostenfunktion günstiger sein, als die wiederholte Berechnung des Ausdrucks. Dies ist nicht immer der Fall. Hat z.B. die Optimierung des Energieverbrauchs die Priorität, ist unter Umständen ein einfacher arithmetischer Ausdruck, der nur Register benutzt, für die Energiebilanz günstiger als ein Speicherzugriff. Ein weiterer Seiteneffekt ist die Einführung zusätzlicher Variablen, die die Registerzuordnung erschwert und u.U. sogar zu einer Verschlechterung der Laufzeit führen kann. Dies zeigt, daß bestimmte Aspekte der gleichen Optimierung nur in einem passenden Modell berücksichtigt werden können.

Um den Konflikt zwischen Codegröße und Geschwindigkeit zu entschärfen, kann versucht werden, die Werte der CSE in den Registern zu halten. Die Auswahl der geeigneten Register bei einem heterogenen Registersatz mit vielen Einschränkungen hinsichtlich ihrer Benutzung (siehe 2.1.2) ist nicht trivial. Eine Lösung für das Problem mit Hilfe des *simulated annealing* Algorithmus wird in [13] vorgestellt. Der Algorithmus operiert auf einer Menge von DFTs (*data flow tree*) und sucht nach einer Anordnung der DFTs mit verschiedenen Ablageorten - also Registern - für die CSE Variablen. Die Situation im Bild 7 zeigt einen DFG mit drei DFTs. Die Bäume T2 und T3 benutzen eine CSE Variable - im Bild als schwarzer Punkt dargestellt, die in T1 berechnet wird. Wir nehmen (T1,T2,T3) als eine Instruktionsanordnung an. Wird nun die CSE Variable in einem Register (PR) gespeichert, so darf in dem Baum T2 das Register PR nicht verändert werden, weder als Zielregister noch infolge eines Seiteneffekts. Die Sequenz müßte in diesem Fall (T1,T3,T2) lauten.

2.2.3 Function Inlining

Die *function inlining* Optimierung gehört zu den standardmäßigen Optimierungen in C Compilern. Hier wird nach heuristischen Methoden entschieden, ob es sinnvoll ist, den Aufruf einer Funktion an einer Stelle durch den Code der Funktion zu ersetzen. Das wichtigste Kriterium ist dabei die Größe der Funktion - als Anzahl der Instruktionen gemessen. Befindet der Compiler, daß der Funktionsaufruf aufwendiger wäre, als die Ausführung des Funktionskörpers an sich, wird die Optimierung angewendet. Für die eingebetteten Systeme ist allerdings der Aspekt der Codegröße bei dieser Optimierung ein wichtiger Faktor. Eine extensive *function inlining* Optimierung kann die Codegröße unakzeptabel anwachsen lassen. In der DSP Optimierung wird *function inlining* deswegen mit einem globalen Kostenmodell für die Codegröße angewendet. Eine Lösung in [14] benutzt einen Branch-and-Bound Algorithmus, um die Auswahl von Funktionen für die Anwendung von *function inlining* unter Berücksichtigung einer globalen Schranke für die Codegöße zu ermitteln. Beim darauffolgenden Profiling wird das dynamische Aufrufverhalten zwischen den Funktionen untersucht. Ferner werden die Größe jeder Funktion und die statischen Funktionsaufrufe zusammengestellt. Ein Vektor, dessen Belegung mit 0 (kein Inlining) bzw. 1 (Inlining) für jede Funktion gefüllt wird, stellt am Ende die Lösung dar.

2.2.4 Dead Code Elimination

Das Erkennen des sogenannten toten Codes gehört zu weiteren standardmäßigen Optimierungen in den C Compilern. Zum Teil wird das Vorkommen des toten Codes bereits beim Compilieren als Warnung oder Fehler gemeldet (*unused variable* bzw. *code has no effect*), wenn der tote Code schon im

Quellcode vorhanden ist. Als Analyse werden Probleme der Erreichbarkeit auf Kontrollflußgraphen gelöst. Ein Instruktionsknoten, der von dem Startknoten aus nicht erreicht werden kann, gilt als toter Code. Eine ungenutzte Variable wird entdeckt, wenn sie innerhalb ihres Gültigkeitsbereichs nicht referenziert wird.

Beispiel 4 *(a) Eine Zuweisungskette (b) nach copy propagation (c) und nach dead code elimination*

(a)	(b)	(c)
`a = x;`	`a = x;`	`a = x;`
`...; // CL1`	`...; // CL1`	`...; // CL1`
`b = a;`	`b = a;`	`...; // CL2`
`...; // CL2`	`...; // CL2`	`a = x;`
`c = b;`	`c = x;`	

Toter Code kann vor allem nach anderen Optimierungen entstehen. Durch *copy propagation* werden unter Umständen die Variablen innerhalb der Zuweisungskette überflüssig.

Im Beispiel 4(a) ist eine Zuweisungskette dargestellt. Eine Analyse ergibt, daß die Variable `b` nicht innerhalb von CL2 redefiniert wird. Daher kann die Variable `a` direkt an die Variable `c` Zugewiesen werden (Bsp. 4(b)). Wenn die Variable `b` in dem Codeblock CL2 nicht mehr referenziert wird, wird die Zuweisung `b = a` als toter Code erkannt und entfernt (Bsp. 4c).

2.3 Bewertung

Die beschriebenen *low level* Optimierungsmethoden beruhen auf vielen Architektureigenschaften, die in einem Maschinenmodell einer Hochsprache nicht verfügbar sind. Ausgangspunkt für diese Optimierungen ist entweder der Code in der IR-Darstellung des Compilers oder sogar der native Maschinencode der Zielarchitektur. Viele der Optimierungsprobleme sind NP-vollständig und werden in den Compilern für GP Architekturen gar nicht eingesetzt. Bei diesen Compilern ist eine kurze Compilierzeit meistens wichtiger als optimaler Code. Im Gegensatz dazu wird bei eingebetteten Systemen der Aufwand für eine qualitativ hochwertige Lösung in Kauf genommen.

Die *medium level* Optimierungen verwenden zwar abstrakte Modelle wie z.B. Datenflüsse, sie werden aber nur lokal, meistens im Basisblock oder zwischen unmittelbar benachbarten Basisblöcken, angewendet.

Es zeigt sich, wie im folgenden Kapitel zu sehen sein wird, daß es sinnvoll ist, die Abstraktionsebene der Optimierung in Richtung vom Maschinencode zum Quellcode hin zu verschieben. Obwohl dann die spezifischen Eigenschaften der eingebetteten Systeme nicht mehr sichtbar sind, gibt es viele Optimierungsmöglichkeiten, die auf die Anforderungen dieser Architekturen ausgerichtet sind - Energieverbrauch, Codedichte, Cacheausnutzung, optimale Speicheranordnung.

3 High Level DTSE Optimierung

3.1 Einführung

Nachdem wir im letzten Kapitel eine kurze Übersicht der Optimierungen in Compilern mit dem Schwerpunkt auf eingebette Systeme gegeben haben, wollen wir jetzt die Methoden vorstellen, die wir als *high level* bezeichnen. Im Gegensatz zu den *low level* Methoden wird *high level* Optimierung innerhalb des Programmiermodells einer Hochsprache angewendet. Da die Sprache C praktisch die einzige Hochsprache für die Programmierung von eingebetteten Systemen ist, bleibt sie der Gegenstand unserer Betrachtungen.

Die eigentliche Optimierung besteht im Umschreiben des Quellcodes unter Beibehaltung der Semantik und der Korrektheit. Üblicherweise werden die Optimierungsaspekte recht spät in den Entwicklungsprozeß einbezogen. Dies liegt zum Teil daran, daß die Optimierung die ursprüngliche Programmarchitektur unter Umständen völlig verändert. Es gibt z.B. Optimierungen, die das Speichermodell in einem Programm komplett umbauen. Die Möglichkeiten, eine derartige Optimierung durchzuführen, sind erst durch die Analyse eines vorhandenen Codes ersichtlich. Auf der anderen Seite wäre es recht aufwendig, ein System zu entwickeln und dabei mit einem sehr komplizierten Speichermodell zu arbeiten.

In den folgenden Abschnitten stellen wir einige *high level* Optimierungsmethoden vor. Es ist eine Sammlung von Methoden des DTSE Verfahrens (*data transfer and storage exploration)* [6]. Mit DTSE wird eine ganze Klasse von Methoden bezeichnet, die den Datentransport, zwischen der CPU und den verschiedenen Ebenen der Speicherhierarchie optimieren. Die Minimierung des Speicherverbrauchs ist ein weiterer Aspekt von DTSE. Die Hauptmotivation für die intensive Speicheroptimierung liegt in dem hohen

Energieverbrauch eines Zugriffs auf den off-chip Speicher, für eingebette Systeme ein entscheidendes Kriterium darstellt.

DTSE umfaßt vielfältige Methoden, die auch eine breite Basis an Modellen benutzen. Wir geben deswegen zuerst einen Überblick über die Modellierung in DTSE. Danach stellen wir ausgewählte Optimierungen vor, deren Ergebnise in Form von transformierten Quellcode des zu optimierenden Programms die Ausgangsbasis für die Quellcodeoptimierung durch Mustererkennung darstellen. Der mit DTSE optimierte Quellcode enthält demnach spezifische Codemuster, die weitere Optimierungen erlauben. Dieses Optimierungspotential ergibt sich aus folgenden Eigenschaften des Quellcodes.

1. Durch intensive Umstruktrierung der Programme entstehen viele verschachtelte Schleifen.

2. Es werden komplexe if-Bedingungen eingeführt.

3. Die Adressierung der Arrays wird sehr komplex. Die Indexberechnung enthält oft Modulo Operationen oder Integer Divisionen.

Auf diese Eigenschaften werden die im Kapitel 4 vorgestellten Optimierungen mit Hilfe von Mustern angewendet. Die im folgenden vorgestellten Modellierungsmethoden stellen die Grundlagen der DTSE Optimierungen dar.

3.1.1 Speichermodellierung mit Silage

Die Modellierung der Kosten umfasst die Bereiche des Speicherbedarfs eines Programmes und des Energiebedarfs für Speicherzugriffe. Hinzu kommen Methoden zur Abschätzung der benötigten Bandbreite für die Speicherzugriffe und des verbrauchten Platzes für den on-chip Speicher. Für die formale Darstellung wird die applikative Sprache Silage [10] benutzt. Silage erlaubt eine flexible Darstellung des Datenflusses ohne eine bestimme Ausführungsordnung des Codes vorauszusetzten - im Gegenteil z.B. zu einer Modellierung mit C.

Um die Speichergröße abzuschätzen, kann keine prozedurale Darstellung wie der C-Quellcode verwendet werden. Die Darstellung muß frei von den Annahmen über die Ausführungsreihenfolge sein. Der applikative Charakter des Speichermodells ist die Voraussetzung für die nötige Flexibilität bei dem Entwurf eines Speichermanagements. Eine C-Spezifikation erlaubt im allgemeinen keine vollständige Analyse des Datenflusses. Störend dabei sind z.B.

die Annahmen über die Ordnung von Zeilen und Spalten beim Zugriff auf mehrdimensionale Arrays. Die applikative Beschreibungssprache Silage wird daher zum Modellieren des Datenflusses eingesetzt. Im folgenden Beispiel wird zuerst das Array A initialisiert. In der Schleife j werden dann die aus A berechneten Werte im Array B gespeichert.

Beispiel 5 *Silage Programmcode*

```
#define W fix<16,4>
(k: 0 .. 9)::
   begin
        A[2*k] = W(0);     A[2*k+1] = W(1);
   end;
(i: 0 .. 9)::
   (j: 0 ..9)::   B[i][j] = A[i+j] + A[2*i+1];
```

Nehmen wir eine, und zwar die durch die Iterationen *k, i, j* vorgegebene, sequentielle Ausführung an. Die ersten 20 Speicherzellen müssen für das Array A in der Schleife k alloziert werden. Bei der Berechnung von B[i][j] kann aber B[i][0] in A[i] gespeichert werden, da A[i] ab diesem Zeitpunkt nicht mehr verwendet wird. Wir sagen, daß die Zahl A[i] *stirbt,* d.h. sie wird zum letzten Mal benutzt. In jedem der neun Durchläufe der Schleife i werden auf diese Weise neun neue Speicherzellen benötigt. Wir kommen also mit 101 Zellen aus.

Diese Kalkulation gilt nur bei der konkret vorausgesetzten Ausführungssequenz. Sie kann aber, wegen fehlender Annahmen, in Silage nicht erfüllt werden. Für parallele Ausführung ergeben sich vielfältige Möglichkeiten der Reihenfolge und Gruppierung. Selbst bei einer sequentiellen Berechnung aller B's gibt es 100! Möglichkeiten, durch B[i][j] durchzugehen.

Um dem Problem beizukommen, kann man den *lower-bound* bzw. *upper-bound* für den Speicherbedarf bestimmen unter der Annahme einer beliebigen aber sequentiellen Ausführungsordnung. Wir wissen, daß für die Speicherung der 100 B-Werte 100 Zellen notwendig sind. Gibt es also eine Anordnung, die nicht mehr als 100 Zellen benötigt? Im folgenden Beispiel ist eine derartige dargestellt.

Beispiel 6

```
(i: 0 .. 9)::
```

```
   (j: 0 ..9)::    B[i][j] = if(i!=j+1 & j!=i+1) -> A[i+j] + A[2*i+1];
(i: 0 .. 9)::
   (j: 0 ..9)::    B[i][j] = if(i==j+1)      -> A[i+j] + A[2*i+1];
(i: 0 .. 9)::
   (j: 0 ..9)::    B[i][j] = if(j==i+1)      -> A[i+j] + A[2*i+1];
```

Entscheidend für die Existenz einer Lösung ist die Verteilung des Verbrauchs der *sterbenden* Werte, d.h. der zum letzten Mal benutzten, auf die Operationen, in denen neue Werte gespeichert werden müssen. Leider gibt es keine effiziente algorithmische Lösung zur Erzeugung derartiger Konfigurationen. Ein Silage Programm kann auf einen gerichteten, azyklischen Graphen abgebildet werden. Das Problem der Berechnung einer Speicherverteilung in einem solchen Graphen ist NP-vollständig. Ähnliche Überlegungen kann man für d*ie upper-bound* durchführen. Es gibt auch hier keine effiziente Methode der Berechnung.

Für die Durchführung der einzelnen DTSE Optimierungsschritte ist die Ermittlung der Speicherschranken zu Beginn des Prozesses sehr wichtig. Insbesondere eine gute obere Schranke sollte zwecks Einschätzung der Ergebnisse vorhanden sein.

3.1.2 Geometrische Modellierung

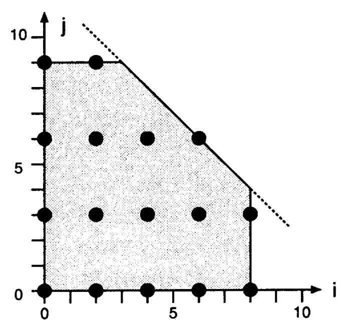


Abbildung 8: Iterationsdomäne zum Codebeispiel

Das geometrische Modellieren eines Programms beruht auf der Darstellung gewünschter Aspekte (z.B. Speicherhierarchie, Datenabhängigkeiten) mit Hilfe geometrischer Objekte und deren Eigenschaften. Eine Menge von Punkten mit ganzzahligen Indizes in einem ggf. mehrdimensionalen Koordinatensystem ist eine effektive Modellierungsart für Instruktionen innerhalb der Schleifen.

Beispiel 7

```
for ( i = 0; i < 10; i += 2 )
   for ( j = 0; j <= 10; j += 3 )
```

```
      if ( i + j <= 12 )
L1:           A[i][j] = f(B[i][j]);
```

Die Menge der Punkte im Bild 8 (*polytope*) wird als Iterationsdomäne bezeichnet (auch Iterationsraum genannt). Jeder einzelne Punkt steht für eine Operation (Label L1 im Code). Die Grenzen der Domäne ergeben sich aus den Iterationsgrenzen der Schleifen. Die „abgeschnitene" Ecke gibt die if-Bedingung wieder. Die Iterationsdomäne wird formal wie folgt beschrieben:

$$\begin{aligned}\mathrm{D}_1^{iter} = \{[i,j] \mid \exists\, \alpha, \beta;\ 0 \le i \le 9 \ \wedge\ 0 \le j \le 10 \ \wedge\ i+j \le 12 \ \wedge\ i = 2\alpha \\ \wedge\, j = 3\beta \ \wedge\ [i,j] \in Z^2 \ \wedge\ [\alpha,\beta] \in Z^2\}\end{aligned}$$

Die Symbole α und β stellen Hilfsdimensionen dar, um die Schrittweite 2 bzw. 3 als Bedingung zu modellieren. Es ergibt sich eine vierdimensionale geometrische Form (i, j, α, β), deren Projektion auf die Fläche i, j das Bild 8 ergibt.

3.2 DTSE Methoden

Im Folgenden skizzieren wir beispielhaft ausgewählte Verfahren der DTSE Optimierung, die auf den im vorigen Abschnitt vorgestellten Modellen basieren. Es sind Transformationen, welche die Lokalität der Speicherzugriffe erhöhen und die Speichertransfers minimieren. Die Vorgehensweise besteht in einer Umstrukturierung des Programms in folgende dreischichtige Hierarchie.

1. In der obersten Schicht sind die Systemfunktionen angesiedelt, die mit der eigentlichen Berechnung nicht zusammenhängt und für die Optimierung irrelevant ist (*main*-Funktion, globale Konstanten).

2. Die Schicht zwei enthält idealerweise alle für die Berechnung relevanten Codeabschnitte in einer Funktion zusammengefasst. Hierdurch wird der eigentliche Gewinn erzielt. Vorher unabhängige Schleifen werden im gemeinsamen Iterationsraum eingebettet.

3. Die Schicht drei, die *low level* Schicht, enthält z.B. die arithmetischen Operationen in der Indexberechnung der Arrayzugriffe oder lange Verkettungen von Bedingungen in if-Anweisungen. Diese Schicht ist nicht

der Gegenstand der DTSE Optimierung. Es sei anzumerken, daß die in der Arbeit vorgestellte Optimierung mit Mustererkennung den durch die Schicht drei erzeugten Code als Ausgangsbasis benutzt und auf diese Weise auf der DTSE Optimierung aufbaut. Durch Anwendung von DTSE entstehen vermehrt komplexe Index-Ausdrücke und verschachtelte Schleifen mit vielen if-Anweisungen auf den Iterationsvariablen.

Das DTSE arbeitet also in der Schicht 2 der obengenannten Programmstruktur. Um das Entstehen der für unsere Quellcodeoptimierung interessanten Code-Strukturen bei DTSE zu zeigen, wollen wir anhand kleiner Beispiele die Codetransformationen von DTSE erläutern.

Als Beispiel soll uns eine Anwendung aus der medizinischen Bildbearbeitung dienen. Es ist ein Algorithmus zur Konturgenerierung in tomografischen Aufnahmen (sog. *cavity detector*). Der Algorithmus enthält im Original drei aufeinander folgende Funktionen, die ein Bild als Eingabe nehmen und ein neues Bild als Ausgabe produzieren, das wiederum als Eingabe für den nächsten Schritt dient.

1. `function GaussBlur(img_in int[N][M], img_out int[N][M])`
2. `function ComputeEdges(img_in int[N][M], img_out int[N][M])`
3. `function detectRoots(img_in int[N][M], img_out int[N][M])`

Die Originalstruktur des *cavity-detector* Algorithnmus enthält die Funktionen, von denen jede durch ihre Eingabe iteriert und das Ergebnis in die Ausgabe schreibt. Die Bilddaten werden also mindestens drei mal komplett aus dem Speicher gelesen und zurückgeschrieben (s. Bsp. 8).

Beispiel 8 *Orginalstruktur des* cavity detector

```
void main() {
  unsigned char image_in[N][M], gxy[N][M], ce[N][M], image_out[N][M];
  // ...
  GaussBlur(image_in, gxy);
  ComputeEdges(gxy, ce);
  DetectRoots(ce, image_out);
```

3.3 Globale Schleifentransformationen

Durch Anwendung der globalen Schleifentransformationen [5] auf den *cavity-detecor wird* die komplette Berechnung in ein Nest aus zwei Schleifen verlagert, wie im folgenden Beispiel zu sehen ist.

Beispiel 9 *Funktion cav_ detect*

```
void cav_detect() {
   for ( y=0; y<M; y++)
      y_ce = y-GB-1;
      ...
      for( x=0; x<N, x++) {
         x_ce = x-GB-1;
         ...
         if (y >= 0 && y < M && x >= 0 && x < N) {
         if (y-1 >=  0)
             gsx[((y-1)%3)*3 + x%3] = ... // Gauss Bluring horizontal
         if (y-2 >= 0)
             gsy[((y-2)%3)*3 + x%3] = ... // Gauss Bluring vertical
         ...
         if (y_ce >= 0 && y_ce < M && x_ce >= 0 && x_ce < N)
            if (y_ce-1 >= 0)
               ce[((y_ce-1)%3)*3 + x_ce%3] = ... // Compute Edges
         ...
         image_out[(y_dr * N + x_dr) % 8704 + (y_dr * N + x_dr)
                   / 8704 * 16384 + 7680] = ... //  Detect Roots
   } }
```

In dem Code sind die vielen if-Anweisungen zu erkennen, die die Randbedingungen für die Anwendung der Funktionen prüfen. Sie stammen aus der Anwendung der Datenflußoptimierung (siehe 3.4 zum Prinzip der Datenflußoptimierung mit dem Polytope Modell). Zu sehen ist z.B. die Umwandlung der Bildpuffer in ein gemeinsames eindimensionales Array, auf dem alle Funktionen arbeiten - image_out[] ist das einzige Array, das das komplete Bild enthält. Die übrigen kleinen Arrays enthalten die Umgebung eines Pixels, in der alle Funktionen durchgeführt werden (im Bild 9 veranschaulicht), bevor am Ende der Schleife die Daten zurück an entsprechender Stelle im Bildarray geschrieben werden. Die komplexen Indexberechnungen rühren aus der lokalen Zugriffscharakteristik her. Bei der Bearbeitung eines Pixels wird auf alle

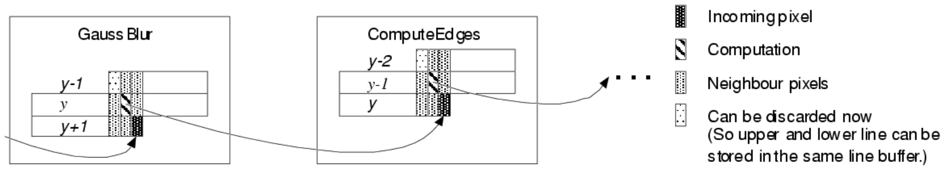


Abbildung 9: *cavity-detection* Algorithmus nach den Schleifentransformationen

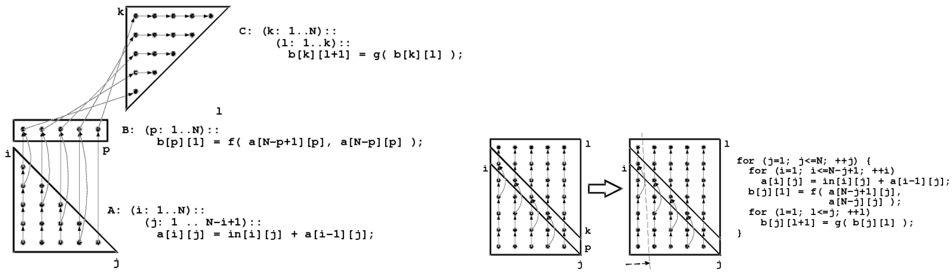


Abbildung 10: Schleifentransformation im Polytope Modell[5]

seine acht Nachbarpixel zugegriffen, und zwar auf einem eindimensionalen Array. Die Verarbeitung gleicht so einer art Softwarepipeline. Jede Iteration der Schleife bearbeitet benachbarte Pixel in jeweils nachfolgenden Stadien.

3.4 Prinzip der Anwendung des Polytope Modells

Eine Anwendung der Schleifentransformationen wird mit Hilfe eines Polytope Modells durchgeführt. Im Bild 10 sind drei Polytopes dargestellt, die aus den drei Schleifen A,B und C generiert wurden. Die Pfeile verdeutlichen den Datenfluß innerhalb der Schleifeniterationen und zwischen den Arrays in,a und b. Die neue Anordnung der Polytopes im rechten Teil des Bildes stellt die im zugehörigen Code gezeigte Transformation der drei Schleifen dar. Sie sind in einem Iterationsraum zusammengefasst. Die Pfeile in dem zusammenhängenden Rechteck zeigen, daß pro Iteration der äußersten Schleife der Datenpfad über alle drei Arrays durchlaufen wird. An den Übergängen zwischen in und a wie auch zwischen a und b können die Daten aus den Registern anstatt aus dem Hauptspeicher geholt werden.

3.5 Ergebnisse der DTSE Optimierung

Die systematische Anwendung der DTSE Methoden liefert für datenintensive Anwendungen gute Ergebnisse hinsichtlich des Energiebedarfs. Dies wird durch die konsequente Minimierung des Datentransfers zwischen dem off-chip Speicher, dem on-chip Speicher und den Registern erreicht. Die Optimierung richtet sich einerseits auf die absolute Speichergröße - durch die Wiederverwendung von Speicher - und andererseits auf die Zugriffscharakteristik auf den Speicher durch die Anpassung des Kontrollflusses an den gegebenen Datenfluß.

In der Regel wird das durch DTSE unter dem Aspekt der Speichernutzung optimierte Programm eine Verschlechterung der Laufzeit aufweisen. Folgende Eigenschaften sind für die DTSE-optimierten Programme charakteristisch:

- Es gibt mehr zusätzliche if-Anweisungen. Die Bedingungen sind lange Ketten von einfachen Relationen - oft Vergleiche einer Variablen mit einer Konstanten.

  ```
  if (x_in >= 0 && x_in < N && y_in >= GB && y_in <= M-1-GB)
  ```

  Das Beispiel stammt aus dem Code des *cavity-detectors.*

- Es gibt tief verschachtelte Schleifen, wie im folgenden Beispiel zu sehen ist.

Beispiel 10 *Code aus dem Motion Estimation Algorithmus (ME) nach DT-SE*

```
for(i=0; i<20; i++) {
   for(x=0;x<36;x++)
   for(y=0;y<49;y++) {
   xm=35; ym=48; min = 4080;
      for(v4x1=0;v4x1<9;v4x1++) {
         for(v4y1=0;v4y1<9;v4y1++) { dist=0;
            for(x4=0;x4<4;x4++) {
            for(y4=0;y4<4;y4++) {
```

- Die Arrayadressierung wird durch DTSE oft in eine nichtlineare Form transformiert. Sie enthält modulo- und integer-Divisionen. Der Berechnungsaufwand steigt dadurch besonders für die einfachen DSP Architekturen, die meist keine FP-Einheiten (*float point unit)* haben. Ein modulo-Befehl erzeugt dann eine große Sequenz an zusätzlichen Zyklen, da die Berechnung duch Aufruf einer Bibliotheksfunktion geschieht. Ein Beispiel, das wieder aus dem ME-Algorithmus stammt, zeigt dies.

Beispiel 11 *Arrayadressierung nach DTSE*

```
tmp[(x%464)*196+y%464+(x*196+y)/464 * 1024+496] = x;
```

Die obigen Codeeigenschaften machen weitere Optimierungen notwendig. Es sind allesamt Optimierungen, die vor allem im Quellcode durchgeführt werden müssen. Für die Optimierung durch einen Compiler ist der betreffende Gültigkeitsbereich (*scope)* entweder zu groß, wie im Falle der verschachtelten Schleifen, oder die Ausdrücke sind für den Compiler zu komplex. So müssten z.B. arithmetische Ausdrücke erst faktorisiert werden, um sie wie bei CSE durch ihre Faktoren zu ersetzen.

4 Ausgewählte Quellcodeoptimierungen

Wir haben im vorigen Kapitel eine Übersicht der Softwareoptimierung für die eingebetten Systeme gegeben. Abhängig von der Abstraktionsebene der verwendeten Modelle haben wir die Optimierung in Stufen eingeteilt. Die *low level* Optimierung arbeitet auf der modellhaften Darstellung einer konkreten Maschinenarchitektur. Die Optimierungen sind auf einen kleinen Codebereich beschränkt (z.B. ein Basisblock). Die im Kapitel 2 vorgestellten *low level* Optimierungen sind meistens in Form von konkreten Algorithmen verfügbar, dadurch sind die ausreichend formalisiert und somit für den Einsatz in Compilern gut geeignet. Ein Compiler stellt also ein Tool zur automatischen *low level* Optimierung dar.

Desweiteren haben wir *medium level* Optimierungen umrissen. Die dabei verwendeten Modelle sind geeignet, abstraktere Sachverhalte, wie Datenflüsse, zu beschreiben. Es sind Optimierungen, deren Anwendungsgebiet von der Abstraktionsstufe des verwendten Modells abhängt. So kann CSE innerhalb eines Basisblocks von dem Compiler durchgeführt werden, aber auch im Bereich einer ganzen C-Funktion auf der Quellcodeebene. Die Modellbildung für die *medium level* Optimierungen basiert auf gut formalisierbaren Konzepten. Für die Manipulation und Analyse der verschiedenen Graphenarten stehen - sehr oft auch effiziente - Algorithmen zur Verfügung. Diese Optimierungen werden daher in hochoptimierenden Compilern eingesetzt. Damit ist eine automatiserte *medium level* Optimierung möglich.

Schließlich haben wir die *high level* Optimierungen vorgestellt und uns dabei auf die DTSE Methodensammlung bezogen. Die Modelle der *high level*

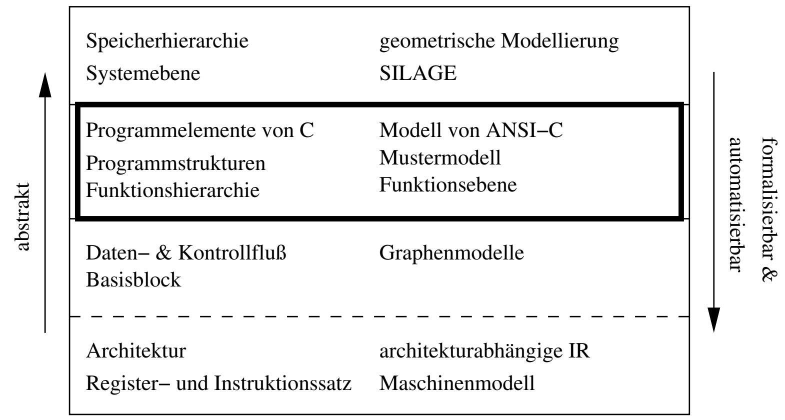


Abbildung 11: Abstraktionsstufen und Modellbildung

Optimierung beschreiben in einer formalen Weise die globalen Sachverhalte innerhalb eines Programms. Mit der Polyheder-Darstellung kann das Speichermodell eines komplexen Algorithmus formal erfasst und manipuliert werden. Die vorhandenen Algorithmen zur Analyse innerhalb des Modells sind allerdings oft sehr aufwendig - NP-vollständig - was deren Einsatz in Optimierungstools erschwert. Der *high level* Optimierungsprozess besteht aus mehreren Zyklen. Dabei werden Codetransformationen durchgeführt, die eine Optimierung relisieren. Danach wird das Programm compiliert ggf. getestet und mit einem Benchmark untersucht. Waren die Tests erfolgreich, kann die nächste Optimierung angewendet werden.

Nach der Phase der DTSE Optimierung ist das Programm hinsichtilich des Energieverbrauchs optimiert. Das Laufzeitverhalten ist in den meisten Fällen schlechter geworden. Der nachfolgende Schritt der Übersetzung kann auch bei einem hochoptimierenden Compiler die durch DTSE bedingten Verluste nicht wieder aufheben. Der Grund dafür liegt in dem noch größer gewordenen Abstand zwischen den Abstraktionsstufen der Modelle von DTSE und der des Compilers. Die abstrakten geometrischen Modelle der DTSE Optimierung erzeugen komplexe Programmstrukturen, die die Compileroptimierung nicht vollständig erfassen bzw. analysieren kann - z.B. arithmetische Transformationen der Modulo-Ausdrücke. Den Zusammenhang zwischen der Abstraktionsebene und der Modellierung zeigt die Abb. 11. Der stark umrandete Bereich zeigt die Einordnung dieser Arbeit in der Abstraktionshierarchie der Optimierungen.

Ein weiterer wesentlicher Unterschied zwischen der *high-level* DTSE Optimierung und der *low-level* Compilieroptimierung ist der Anwendungsbereich der jeweiligen Methoden. Die DTSE-Transformationen operieren auf dem Programm als ganzes und betrachten globale Zusammenhänge in dem Speichermodell. Der Compiler geht bei der Optimierung von dem Modell des Basisblockes aus. Analysiert werden meist nur die unmittelbar benachbarten Blöcke.

Die oben genannten Unterschiede implizieren den jetzigen Stand in der Verfügbarkeit automatischer Verfahren in der Optimierung. Während die Optimierung im Compiler durch effiziente Algorithmen realisiert werden kann, ist die DTSE Methode vorwiegend auf manuelle Durchführung angewiesen. Ebenso wird die Nachoptimierung des Quellcodes nach der Anwednung von DTSE von dem Entwickler manuell durchgeführt. Genau an dieser Stelle soll im folgenden Kapitel angesetzt werden. Wir versuchen, bestimmte Quellcodetransformationen formal zu analysieren, um sie mit Hilfe der Mustererkennung im Quellcode anwenden zu können.

4.1 Quellcodetransformationen

Eine Hochsprache ist für den Programmierer gedacht, um auf einer hohen Abstraktionsebene einen Algorithmus effizient kodieren zu können. Die intensive und systematische Anwendung der Optimierung im C-Quellcode wird durch das abstrakte Maschinenmodell einer Hochsprache nicht optimal unterstützt.

Quellcodeoptimierung wird erst durch den Einsatz von C für eingebettete Systeme intensiv erforscht. Gewöhnlich wird die Optimierung im Quellcode manuell durchgeführt. Das Erkennen von Programmelementen, die eine Transformation erlauben, ist im großen *scope* eines C-Programms nicht trivial. Meistens wird es durch „scharfes Hinsehen" des Entwicklers gemacht. Es gibt eine Reihe von Tools, die die nötige Analyse wie auch die Anwendung der Transformationen unterstützen. Das automatische analysieren eines Programms mit einem Tool nur anhand des C-Quellcodes ist allerdings aufwendig. Deswegen operieren die Tools nicht direkt auf dem C-Quellcode sondern auf einer geeigneten internen *high level* Darstellung - z.B. die SUIF-Darstellung [9]. Diese Darstellung ist eine Abbildung des C-Quellcodes, aus der sich nach der Optimierung der transformierte C-Quellcode wieder generieren läßt.

Wir stellen im weiteren eine Reihe von Optimierungen vor, die als Quellcodetransformationen modelliert werden können. Im Kapitel 5 stellen wir ein Tool zur automatischen C-Quellcodetransformation und das Modell der Quellcodemuster vor.

4.2 Konditionalausdrücke in Schleifen

Im folgenden wollen wir uns mit den Möglichkeiten der Optimierung von Konditionalausdrücken beschäftigen. Dabei werden wir uns auf bestimmte Konstellationen der Schleifen und if-Anweisungen konzentrieren. Da es sich um charakteristische Codestrukturen handelt, die nach Anwendung von DTSE auftreten, scheint dieses Vorgehen gerechtfertigt. Wir zeigen Quellcodetransformationen, die zum Ziel haben, die redundanten Auswertungen von Prädikaten in den if-Anweisungen zu eliminieren. Die allgemeine Form der von uns betrachteten Codestrukturen ist ein Nest von verschachtelten Schleifen. Innerhalb des Schleifenkörpers gibt es eine Sequenz von Konditionalausdrücken, die Bedingungen an die Wertebereiche der Iterationsvariablen formulieren. Wir werden für jedes betrachtete Beispiel die Anzahl $\#cond_i$ der ausgeführten Vergleichsoperationen auf der Iterationsvariablen jeweils vor und nach der Transformationen angeben. Mit $\#cond_i$ modellieren wir die Kostenfunktion des Programms.

4.2.1 Einfache Schleife

Der einfachste Fall besteht aus einer Schleife mit einer Bedingung an die Indexvariable im Schleifenkörper.

Beispiel 12 *Schleife mit einem Konditionalausdruck*

```
for ( i = 0; i < M; i++) {
   ... // stmt_seq_1
   if ( i < u1 ) { ...} // cond_seq_1
   ... // stmt_seq_2
}
```

Man kann für das Beispiel 12 die Anzahl der Ausführungen von Konditionaloperationen auf der Iterationsvariablen i ($cond_i$) mit $2*M+1$ angeben - Abbruchbedingung der Schleife und die if-Anweisung mit u1. Nun läßt sich überlegen, ob die Information über den Wert der i-Variablen, die bei der Auswertung der Bedingung gewonnen wird, für nachfolgende Iterationen erhalten werden kann. Wir wissen, daß die Variable i im Interval $I_i = [0 \dots M-1]$ mit der Schrittweite $+1$ iteriert. Das Interval I_i wird für die Ausführung des Blockes `cond_seq_1` auf ein Teilinterval $I_{u1} = [0 \dots u1-1] : 0 \leq u1 < M$ beschränkt. Durch die Transformation des Quellcodes in die Form wie im Beispiel 13 verringert sich die Anzahl der Auswertungen von $cond_i$.

Beispiel 13 *Transformierte Schleife mit einem Konditionalausdruck i<u1*

```
for ( i = 0; i < M; i++) {
   if ( i < u1 ) {
      for ( ;i < u1; i++) {
         ... // stmt_seq_1
         ... // cond_seq_1
         ... // stmt_seq_2
      }
   }
   ... // stmt_seq_1
   ... // stmt_seq_2
}
```

Für die $M - u1$ Iterationen, in denen $i < u1 \equiv false$ gilt, werden zwei $cond_i$ ausgewertet. Für die erste Iteration mit $i < u1 \equiv true$ werden drei $cond_i$ augewertet. Alle nachfolgenden $u1 - 1$ Iterationen mit $i < u1 \equiv true$ enthalten nur eine $cond_i$ - die Abbruchbedingung der inneren for-Schleife. Die Anzahl der Auswertungen von $cond_i$ beträgt somit

$$\#cond_i = (M - u1) * 2 + 3 + (u1 - 1) = 2 * M - u1 + 2$$

Für den Fall, daß die if-Bedingung $i > u1$ lautet, muß die Transformation anders aussehen. Die Abbruchbedingung der inneren Schleife ist gleich der der äußeren Schleife. Deswegen gibt es jetzt zusätzlich einen else-Block, der die Anweisungen `stmt_seq_1` und `stmt_seq_2` einschließt.

Beispiel 14 *Transformierte Schleife mit einem Konditionalausdruck und der Bedingung i>u1*

```
for ( i = 0; i < M; i++) {
   if ( i > u1 ) {
      for ( ;i < M; i++) {
         ... // stmt_seq_1
         ... // cond_seq_1
```

```
        ... // stmt_seq_2
      }
    }
    else {
    ... // stmt_seq_1
    ... // stmt_seq_2
    }
}
```

In den ersten $u1 + 1$ Iterationen gibt es 2 $cond_i$ Berechnungen. Ist $i = u1 + 1$, werden drei $cond_i$ Ausdrücke ausgeführt. Ab $i = u1 + 2$ fällt pro Iteration der inneren Schleife ($M - u1 - 2$) eine $cond_i$ Auswertung an, wie auch der letzte Test der Abbruchbedingung. Dies zusammen ergibt die Anzahl der $cond_i$ Berechnungen von

$$\#cond_i = (u1 + 1) * 2 + 3 + M - (u1 + 3) = M + u1 + 2$$

Durch ähnliche Analysen bekommt man die Ergebnisse für die Operatoren $\leq$ bzw $\geq$. Für $\geq$ ist $\#cond_i = M + u1 + 1$ und für $\leq$ $\#cond_i = 2 * M - u1 + 1$. Es ist jeweils eine Operation weniger, da die innere Schleife früher betreten wird.

Den letzten Fall in unserem einfachen Beispiel stellt der Operator $=$ dar. Das Einsparungspotential ist hierbei am größten, da die Anzahl der redundanten $cond_i$ Berechnung mit $M - 1$ am größten ist.

Beispiel 15 *Transformierte Schleife mit einem Konditionalausdruck und der Bedingung i==u1*

```
for ( i = 0; i < M; i++) {
   if ( i  <  u1 )
      for ( ;i < u1; i++) {
         ... // stmt_seq_1
         ... // stmt_seq_2
      }
   ... // stmt_seq_1
   ... // cond_seq_1
   ... // stmt_seq_2
   i++;
   for (;i<M; i++) {
   ... // stmt_seq_1
   ... // stmt_seq_2
```

```
    }
  }
```

Wir analysieren die $\#cond_i$ in dem Beispiel 15. Die äußere Schleife trägt nur zwei $cond_i$ bei. Die erste innere Schleife ebenfalls zwei - für die erste Iteration - und $u1-1$ für die nachfolgenden plus 1 für die Abbruchbedingung. Die zweite innere Schleife $M-(u1+1)$ für die Iterationen und eine für den Endtest. Dies ergibt zusammen

$$2+(2+(u1-1)+1)+(M-(u1+1)+1)=M+4$$

und ist konstant bei jeder Wahl der ursprünglichen Schranke $u1$.

Für den Fall, daß die Schleife rückwärts iteriert, gilt immer noch $\#cond_i = M+4$, wenn die Transformation entsprechend angepasst wird. Alle `i++` werden durch `i--` ersetzt. Zusätzlich werden alle Vergleiche von i mit u1 umgedreht und die Abbruchbedingung in der zweiten inneren Schleife an die äußere angepasst.

Die erzielte Verbesserung Δ_i läßt sich als Funktion von $u1$ darstellen. Für den Fall der $(=)$-Relation ist $\Delta_i = M-3$ konstant.

$$\begin{array}{llllll} Rel. <: & \Delta_i = & u1-1 & \qquad Rel. \leq: & \Delta_i = & u1 \\ Rel. >: & \Delta_i = & M-u1-1 & \qquad Rel. \geq: & \Delta_i = & M-u1 \end{array}$$

Anhand Δ_i läßt sich abschätzen, ob die Anwendung der Transformation einen Vorteil erbringt.

4.2.2 if-Anweisungen mit UND-Ketten

Wir betrachten jetzt Erweiterungen des einfachen Beispiels. Die erste Erweiterung führt eine komplexere Bedingung in der if-Anweisung ein. Wir setzen eine Kette von einfachen Bedingungen der Form $i \otimes const$ voraus. Die Relation $\otimes$ kann dabei ein Operator aus der Menge $O := \{<, <=, >, >=, ==\}$ sein. Die Verkettung erfolgt durch den logischen UND-Operator (&&).

Beispiel 16 *Schleife mit einer UND-Konditionalkette*

```
for ( i = 0; i<M; i++) {
   ... // stmt_seq_1
```

```
   if ( i > l1 &&  i < u1 && i < u2) { ... } // cond_seq_1
   ... // stmt_seq 2
}
```

Um die Anzahl von $cond_i$ auszurechnen, machen wir zuerst einige Annahmen über die Struktur der if-Anweisung:

1. Wir gehen von der realistischen Annahme aus, daß die Auswertung einer &&-Kette von Prädikaten nur bis zum ersten *false* liefernden Ausdruck durchgeführt wird. Dies entspricht der C-Semantik.

2. Ferner gehen wir von einer nach den Konstanten aufsteigenden Anordnung der Prädikate aus. Für das Beispiel 17 gilt somit $l1 < u1 < u2$.

Wir teilen das Intervall $[0 \ldots M-1]$ in drei Subintervalle und bestimmen für jedes von ihnen $\#cond_i^{Sub}$ und dann $\#cond_i$.

$$
\begin{aligned}
A: [0 \ldots l1] \quad \#cond_i^A = &\quad (l1+1)*2 \\
B: [l1+1 \ldots u1-1] \quad \#cond_i^B = &\quad (u1-l1-1)*4 \\
C: [u1 \ldots M-1] \quad \#cond_i^C = &\quad (M-u1)*3+1_{Endtest} \\
womit \quad \#cond_i = &\quad 3M+u1-2l1-1
\end{aligned}
$$

Nun transformieren wir den Code um.

Beispiel 17 *Schleife mit einer Konditionalkette*

```
for ( i = 0; i<M; i++) {
   if ( i > l1 &&  i < u1 && i < u2)
      for ( ; i < u1; i++) {
         ... // stmt_seq_1
         ... // cond_seq_1
         ... // stmt_seq_2
      }
   ... // stmt_seq_1
   ... // stmt_seq 2
}
```

und berechnen $\#cond_i$.

Für $\#cond_i^A$ ändert sich nichts. Im Intervall B sind jetzt fünf $cond_i$ für die erste Iteration der inneren Schleife und ein $cond_i$ für alle folgenden Iterationen sowie der Endtest. Also ist $\#cond_i^B = 5 + (u1 - l1 - 1)$. Der Block C hatte vorher drei $cond_i$ in jeder Iteration. Nun erfordert die erste Iteration keine Berechnung mehr, da sie bereits in dem Endtest der inneren Schleife erfolgt ist und somit ist $\#cond_i^C = (M - u1 - 1) * 3$.

$$\begin{aligned}
A : [0 \ldots l1] \quad \#cond_i^A &= \quad (l1 + 1) \times 2 \\
B : [l1 + 1 \ldots u1 - 1] \quad \#cond_i^B &= \quad 5 + (u1 - l1 - 1) \\
C : [u1 \ldots M - 1] \quad \#cond_i^C &= \quad (M - u1 - 1) \times 3 + 1_E ndtest \\
womit \quad \#cond_i &= \quad 3M - 2u1 + l1 + 4
\end{aligned}$$

Das Δ_i hängt jetzt mit der Größe des Subintervalls B, in dem alle Prädikate erfüllt sind, über folgende Beziehung zusammen

$$\#cond_{i_{orig}} - \#cond_{i_{opt}} = \Delta_i = 3u1 - 3l1 - 5$$

4.2.3 Sequenz von if-Anweisungen in verschachtelten Schleifen

Zuletzt betrachten wir einen allgemeinen Fall von mehreren eingebetteten Schleifen. Die innerste Schleife enthält eine Sequenz von if-Anweisungen, von denen jede als Bedingung eine UND-Kette von Konditionalausdrücken auf den Indexvariablen aller Schleifen enthält. Wir verzichten hier auf die detaillierte Berechnung der Anzahl von Auswertungen der Prädikate. Wir gehen davon aus, daß die Grenzen l,u,L und U im Interval $[0 \ldots M - 1]$ bzw. $[0 \ldots N - 1]$ liegen. Da wir jetzt nicht mehr eine festgelegte Ordnung für die Prädikate annehmen, können wir nicht eine exakte Anzahle der Auswertungen in den if-Anweisungen Angeben. Wir schätzen die Anzahl daher nur ab. Für $cond_i$ ist die obere Schranke der Auswertungen $\#cond_{i_O} = N*(M*6+1)$. Für $cond_j$ beträgt sie $\#cond_j = (N + 1) + M * 6$. Die Werte werden analog zu den obigen Beispielen ermittelt.

Beispiel 18 *Schleifennest mit einer Sequenz von Konditionalausdrücken*

```
for( j = 0; j < N; j++)
   for ( i = 0;  i < M; i++) {
```

```
        ... // stmt_seq_1
        if ( i > l11 && i < u11 && j<L11 && j>U11) { ...} // cond_seq_1
        ... // stmt_seq 2
        if ( i > l21 && i < u21 && j< L21 && j>U21) { ... } // cond_seq_2
        ... // stmt_seq 3
        if ( i > l31 && i < u31  && j< L31 && j>U31) { ... } // cond_seq_3
        ... // stmt_seq4
}
```

Für die Transformation des Besispiels 18 betrachten wir ein Intervall-Modell, in dem das Schnittinterval der Indexbereiche aller if-Bedingungen gefunden werden kann. Die Konditionalausdrücke der neuen if-Anweisung (COMMON-IF im Beispiel 19) werden aus den Minima bzw. Maxima der Intervallgrenzen der drei if-Anweisungen im Beispiels 18 gesetzt. $s\_i = \max\{l11, l21, l31\}$ und $S\_i = \min\{u11, u21, u31\}$. Die Grenzen s_j und S_j ergebn sich analog.

Beispiel 19 *Transformiertes Schleifennest mit einer Sequenz von Konditionalausdrücken*

```
for( j = 0; j < N; j++)
   for ( i = 0;  i < M; i++) {
      if( i > s_i && i < S_i && j < s_j && j > S_j) // COMMON-IF
         for(; i < S_i; i++) { // OPT-LOOP
         ... // stmt_seq_1
         ... // cond_seq_1
         ... // stmt_seq 2
         ... // cond_seq_2
         ... // stmt_seq 3
         ... // cond_seq_3
         ... // stmt_seq4
         }

      ... // stmt_seq_1
      if ( i > l11 && i < u11 && j<L11 && j>U11) { ...} // cond_seq_1
      ... // stmt_seq 2
      if ( i > l21 && i < u21 && j< L21 && j>U21) { ... } // cond_seq_2
      ... // stmt_seq 3
      if ( i > l31 && i < u31  && j< L31 && j>U31) { ... } // cond_seq_3
      ... // stmt_seq4
}
```

Der Iterationsraum der beiden Schleifen für i und j ist zweidimensional. In der if-Anweisung COMMON-IF wird ein Raum I aufgespannt von der Größe $(S\_i - s\_i - 2) \times (S\_j - s\_j - 2)$ (siehe Abbildung 12). Der Gewinn im optimierten Code ergibt sich aus der I-maligen Ausführung des Schleifenkörpers der neuen Schleife (OPT-LOOP im Beispiel 19) ohne if-Anweisungen. Entscheidend für die Anwendung dieser Optimierung ist die Größe von I.

4.2.4 Bewertung

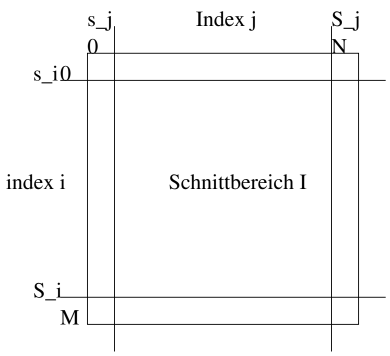


Abbildung 12: Iterationsraum zweier eingebetteter Schleifen im Beispiel 19

Die vorgestellten Transformationen nutzen die Wiederverwendung der Information über die Erfüllbarkeit von Prädikaten in if-Anweisungen. Unter der Prämisse, daß die if-Anweisungen zu *true* evaluieren, werden in Schleifen Codebereiche ausgeführt, ohne die if-Anweisungen selbst berechnen zu müssen. Es soll klar werden, daß das Ziel der Optimierung nicht die komplette Eliminierung der if-Anweisungen sein soll. Dies wäre mit dem Abrollen der Schleife über alle Schnittintervalle der if-Bedingungen möglich. Bei einer konsequenten Anwendung des Abrollens tritt jedoch eine massive Aufblähung des Codes auf.

Das Schema des Speicherzugriffs nach der Anwendung von DTSE entspricht einem mehrdimensionalen Iterationsraum (wie im Bild 12). Dabei stellt jede Schleife im Nest eine Dimension dar. Die Intervalle der Indexvariablen, die die UND-Ketten erfüllen bzw. nicht erfüllen, ergeben Regelmäßigkeiten innerhalb des Iterationsraumes. Die Randbereiche, wie sie im Bild 12 zu sehen sind, entsprechen Indexintervallen, für die bestimmte if-Anweisungen Randbedingungen abfragen und sicherstellen. Der mittlere Schnittbereich erfüllt mit seinen Koordinaten alle if-Bedingungen. Es scheint berechtigt, nach den maximalen Schnittbereichen in dem Iterationsraum zu suchen. Die Anwendung der Optimierung auf die Randbereiche ist zwar möglich, bringt aber, neben dem Nachteil des duplizierten Codes, bei den wenigen Iterationen kaum Gewinn.

Die Transformation ist zwar auf beliebig tief verschachtelte Schleifen anwendbar, jedoch wird die Einsparung mit steigender Nesttiefe immer kleiner. Der Anteil des gesuchten Schnittintervalls an dem gesamten n-dimensionalen Iterationsraum wird mit größerem n kleiner.

Die Analysen, die vor der Anwendung dieser Optimierung durchgeführt werden müssen, beschränken sich auf die Struktur der Konditionalausdrücke und auf die Ermittlung der Größe des gemeisamen Schnittbereichs. Es muß sichergestellt werden, daß es sich um Bedingungen der Form $i_x \otimes const$ handelt. Dabei ist i_x die Iterationsvariable der x-ten Schleife im Schleifennest. Der Operator $\otimes$ ist ein C-Vergleichsoperator und *const* eine Zahl. Die Struktur der Kette von Konditionen muß sich auf die &&-Operatoren beschränken. Somit ist die Analyse auf einen lokalen Bereich beschränkt.

Die Verkettung von Prädikaten mit dem ||-Operator ergibt im allgemeinen keinen zusammenhängenden Schnittbereich, und muß daher ganz anders Modelliert werden. Die Techniken hierzu sollen in dieser Arbeit nicht behandelt werden.

4.3 Eliminierung der Modulo Ausdrücke

Die typischerweise vorhandene nichtlineare Adressierung, die als Folge der DTSE Transformationen entsteht, kann durch Quellcodetransformationen der NOSR-Optimierung (*non-liear operator strength reduction)* eliminiert werden. Wir gehen davon aus, daß die nach DTSE vorhandenen modulo Operationen durch Aufrufe von Bibliotheksfunktionen berechnet werden, was für die eingebetteten Systeme mit Sicherheit zutreffend ist. Auch für Prozessoren, die die modulo Instruktion direkt unterstützen, kann diese Transformation bei vielen gleichen modulo-Ausdrücken einen Vorteil bringen. Durch die Transformation wird zwar ein Mehraufwand an Instruktionen eingeführt, der jedoch aus einer überschaubaren Anzahl von billigeren Operationen besteht. Die Transformationen führen dabei neue Zählvariablen ein, die entsprechend weitergezählt bzw. zurückgesetzt werden, um den repetitiven Charakter der modulo Operation nachzubilden [8]. Wir nehmen in allen nachfolgenden Beispielen an, daß das Modul $i > 1$ ist. Außerdem gehen wir noch von einer einfachen Form des Moduls aus, in der es immer als Konstante auftritt.

Im Bild 13 ist das Zugriffsschema einer Arrayadressierung dargestellt. In dem Intervall $[0 \ldots 19]$ wird auf die ersten vier Elemente des Arrays B zugegriffen. Nach der Einführung der Zählvariablen *tmp* entfällt die modu-

```
for (i=0; i<20; i++)
      B[i mod 4];

tmp=0;
for (i=0; i<20; i++) {
      if(tmp >= 4)
         tmp -=4;
      B[tmp];
      tmp ++;
}
```

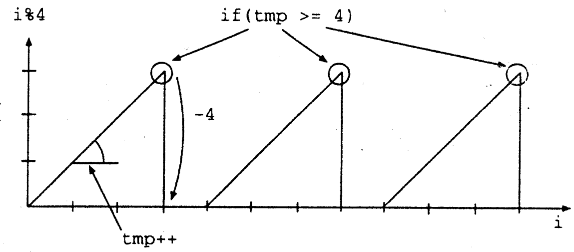


Abbildung 13: Abschnittssweise linearer Indexzugriff

lo Berechnung komplett. Die if-Anweisung vor dem Arrayzugriff und der folgende Inkrementbefehl bilden den zyklischen Zugriffscharakter nach. Die Initialisierung der *tmp* Variablen ergibt sich aus dem Startwert der Schleife und wird auf $tmp = 0 \bmod 4 = 0$ gesetzt. Der Inkrementbefehl entspricht der Schrittweite der Schleife in der Form $step \bmod 4$.

Wir wollen im folgenden die Struktur der Schleife variieren und die Transformation entsprechend anpassen. Zuerst verändern wir die untere Grenze der Schleife und setzen den Startwert auf $i < 0$.

Beispiel 20 *Schleife mit negativem Startwert (a) und die zugehörige Transformation (b)*

```
for(i=-10; i<10; i++)
   B[i % 4]...;
```

(a)

```
tmp = -10 % 4;
for(i=-10; i<10; i++)
   if(i<0) {if(tmp > 0) tmp-=4;}
   else {if(tmp >= 4) tmp-=4;}
   B[tmp]...;
   tmp++;}}
```

(b)

Das Zugriffsschema sieht nicht mehr über dem ganzen Schleifenintervall gleich aus. Für negative i-Werte ist das Zackenbild (Abb. 14) am Koordinatenursprung gespiegelt. Diese „Unstetigkeit" innerhalb des Schemas wird in der Transformation durch die zusätzliche if-Anweisung abgefragt.

Als nächstes drehen wir die Iterationsrichtung der Schleife herum und versuchen, wieder eine passende Transformation zu finden.

Beispiel 21 *Schleife mit positivem Startwert und negativer Schrittweite (a) und die zugehörige Transformation (b)*

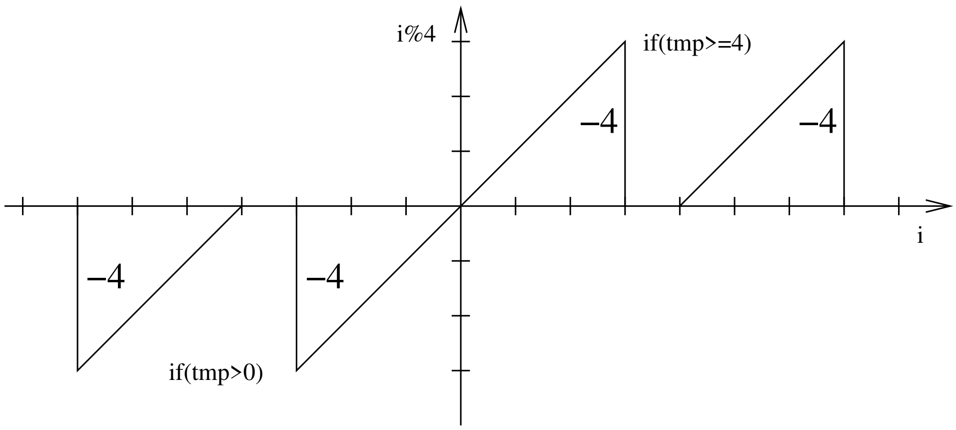


Abbildung 14: Arrayzugriff für negative und positive Indexvariablen

```
for(i=10; i>-10; i--)
   B[i % 4]...;
```

(a)

```
tmp = 10 % 4;
for(i=10; i>-10; i--)
   if(i<0) {if(tmp <=-4) tmp+=4;}
   else {if(tmp < 0) tmp+=4;}
   B[tmp]...;
   tmp--;}}
```

(b)

Die neue Zählvariable *tmp* wird jetzt entsprechend der Schleifenrichtung heruntergezählt. Das Zurücksetzten geschieht diesmal durch die Addition des modulo Wertes.

Wir werden jetzt die Schrittweite der Schleife variieren und auf $|step| \neq 1$ setzen. Um die *tmp*-Variable richtig hochzählen zu können, muß sie um den konstanten Wert $step \bmod 4$ geändert werden. Für den Fall $|step| \neq mod$ kann die Transformation bis auf das Inkrementieren der *tmp*-Variablen unverändert übernommen werden. Der Audruck $step \bmod 4$ sollte vom Compiler als Konstante wegoptimiert. Andernfalls muß der Ausdruck als Initialisierung einer `const` Variablen dienen.

```
...;
tmp += step % 4;
...;
```

Den Fall $|step| = mod$ betrachten wir getrennt. Im Beispiel 22(a) ist der Ausdruck `i % 3` konstat für alle $i < 0$ bzw. $i > 0$. Der Wechsel vom negativen Indexbereich in den positiven kann mit der bisherigen Transformation nicht

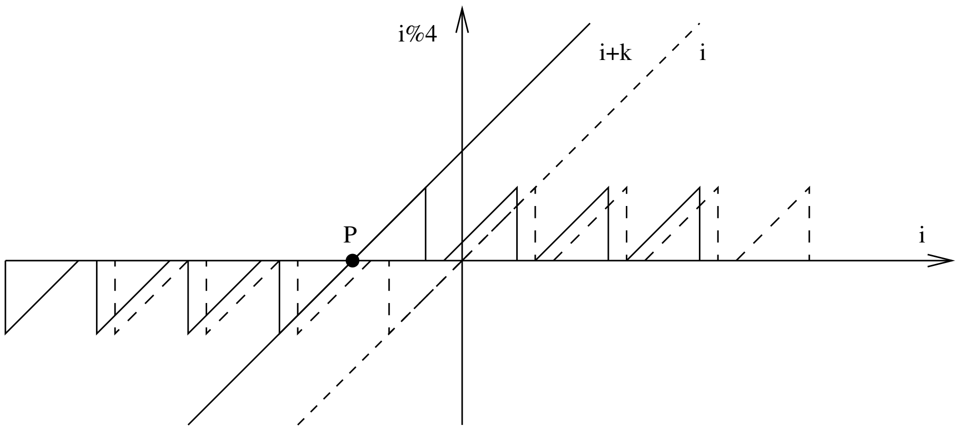


Abbildung 15: Nichtlineares Zugriffsschema. Für i%c (gestrichelt) und (i+k)%c (durchgezogen)

korrekt behandelt werden. Es ist eine Transformation nötig, die die *tmp*-Variable für $i > 0$ neu initialisiert. Alternativ ist eine zusätzliche Variable *tmp2* möglich, die vor der Schleife initialisiert werden kann. Sie enthält den Wert der modulo Operation nach dem Vorzeichenwechsel. In diesem Fall muß eine if-Anweisung die Verwendung von *tmp* oder *tmp2* entscheiden (Bsp. 22(b)).

Beispiel 22 *Die Schrittweite ist gleich dem Modul (a); tmp bzw. tmp2 für den negativen bzw. den positiven Bereich (b)*

```
for(i=-10; i<10; i+=3) {
   ...;
   B[i % 3]...;
   ...;
}
```

(a)

```
tmp=-10%3;
      /* Wert für i>0 */
tmp2=((-10 - (-10/3)*3)+3)%3;
for(i=-10; i<10; i+=3) {
   if(i<0) B[tmp]...;
   else B[tmp2]...;
}
```

(b)

Als letzten Fall betrachten wir einen modulo-Ausdruck mit erweiterter Form des Dividenden. Zu der Variablen in dem modulo Ausdruck wird eine Konstante k addiert (Bsp. 24(a)). Die Änderung in der Transformation besteht in der angepassten Initialisierung der Zählvariablen *tmp* (Bsp. 24(b)). Wechselt zusätzlich beim Durchiterieren das Vorzeichen der Indexvariablen, so muß auch die Abfrage des Vorzeichenwechsels modifiziert werden.

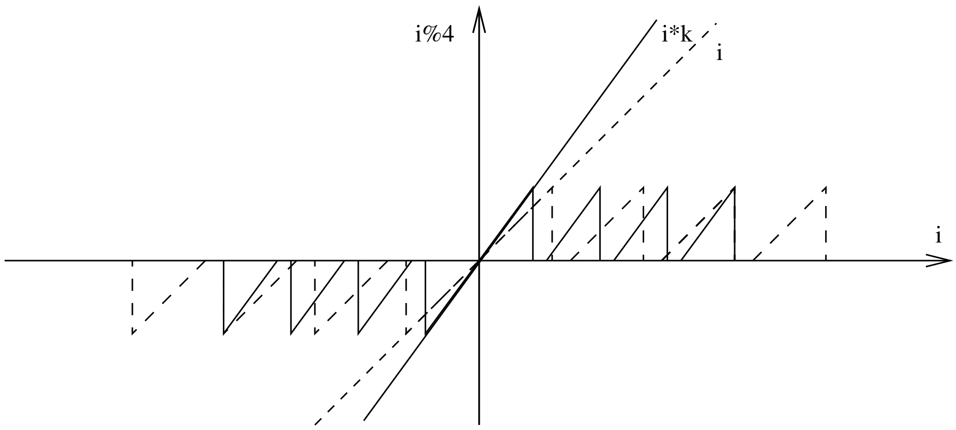


Abbildung 16: Nichtlineares Zugriffsschema. Für i%c (gestrichelt) und (i*k)%c (durchgezogen)

Beispiel 23 *Modulooperation für (i+k) (a) und die Transformation (b)*

```
for(i=-10; i<10; i++) {
   ...;
   B[(i+k)%4]...;
   ...;
}
```

```
tmp=(-10+k)%4; // Zählvariable
for(i=-10; i<10; i++) {
   if((i+k) < 0)
      {if(tmp <=-4) tmp+=4;}
   else
      {if(tmp < 0) tmp+=4;}
   B[tmp]...; tmp++;}
```

Das Bild 15 zeigt die Änderung des Zugriffsschemas beim Addieren einer Konstante. Es veranschaulicht die modifizerte Initialisierung von *tmp* und den Vorzeichenwechsel am Punkt P. Die Zackenform des Zugriffsschemas wurde um k verschoben. Der Fall, in dem die Addition durch eine Multiplikation ersetzt wird $(i * k)$, kann durch eine Stauchung des Musters veranschaulicht werden (Abb. 16). In diesem Fall muß neben der Initialisierung der *tmp* Variablen auch die neue Schrittweite, anstelle des Vorzeichenwechsels, angepasst werden.

Beispiel 24 *Modulooperation für (i*k) (a) und die Transformation (b)*

```
for(i=1; i<100; i++) {
   ...;
   B[(i*k)%4]...;
   ...;
}
```

```
tmp = (1 * k) % 4; //  Zählvariable
tmp_step = (1 * k) % 4; // Schrittweite
for(i=1; i<100; i++)
   ...;
   B[tmp]...;
   tmp+=tmp_step;
```

4.3.1 Generisches Modell der Indexersetzung

In dem folgenden Abschnitt wollen wir den formalen Hintergrund der NOSR Optimierung, wie es in [8] zu finden ist, kurz vorstellen. Das Modell erlaubt die Transformation für beliebig tief verschachtelte Schleifenstrukturen. Es ist auf die modulo- und integer-Division anwendbar, und erhält die ursprüngliche Schleifenstruktur.

Die Adressierung der Arrays, die eine affine Abhängigkeit von den umschliessenden Iterationsvariablen besitzt, lässt sich in einer kanonischen Form darstellen.

Beispiel 25 *Kanonische Darstellung des Arrayzugriffs in dem post-DTSE Code*

```
for (iter1 = init1; iter1 < stop1; iter1 += step1)
   for (iter2 = d21*iter1 + d22; iter2 < stop2; iter2 += step2)
   ...
      for (iterN = dN1*iter1 + dN2*iter2 + ... + dNN;
           iterN < stopN; iterN += stepN)
         A [(c0 + c1*iter1 + c2*iter2 + ... + cN*iterN) % val
           +(c0 + c1*iter1 + c2*iter2 + ... + cN*iterN) / val];
```

Beispiel 26 *Kanonische Darstellung des Arrayzugriffs nach der NOSR Optimierung*

```
for (iter1 = init1; iter1 < stop1; iter1 += step1) {
   if (modPtr1 >= val) { modPtr1 -= val; divPtr1++; }
   modPtr2 = modPtr1;
   divPtr2 = divPtr1;
   for (iter2 = d21*iter1 + d22; iter2 < stop2; iter2 += step2) {
      if (modPtr2 >= val) { modPtr2 -= val; divPtr2++; }
      modPtr3 = modPtr2;
      divPtr3 = divPtr2;
      ...
         for (iterN = dN1*iter1 + dN2*iter2 + ... + dNN;
              iterN < stopN; iterN += stepN) {
            if (modPtrN >= val) { modPtrN -= val; divPtrN++; }
            A [modPtrN + divPtrN];
            modPtrN += mod_incrN;
            divPtrN += div_incrN;
         }
```

Die konsequente Anwendung der Eliminierung von Operatoren / und % ergibt das Codeschema in Beispiel 26. Die Werte $modPtr_1$, $divPtr_1$, mod_incr_i sowie div_incr_i können nach folgenden Formeln zur Compilezeit berechnet werden.

$$
\begin{aligned}
modPtr_1 &= k \,\%\, val \\
divPtr_1 &= k \,/\, val
\end{aligned}
$$

wobei $k = (c_0 + c_1 init_1 + \ldots + c_N init_N)$ und

$$
\begin{aligned}
init_2 &= d_{21} init_1 + d_{22} \\
&\vdots \\
init_N &= d_{N1} init_1 + d_{N2} init_2 + \ldots + d_{NN}
\end{aligned}
$$

Die Inkrementwerte werden nach folgenden Folrmeln berechnet

$$
\begin{aligned}
mod\_incr_i &= base_i \,\%\, val \\
div\_incr_i &= base_i \,/\, val
\end{aligned}
$$

wobei

$$\forall i \in [1, N] : base_i = (\sum_{j=0}^{N-1} c_{i+j} \times E_{j,i}) \times step_i$$

und

$$
\begin{aligned}
\forall j \in]0, N-i] : \quad E_{j,i} &= \sum_{k=0}^{j-1} d_{i+j,i+k} \times E_{k,i} \\
j = 0 : \quad E_{0,i} &= 1
\end{aligned}
$$

Dabei sind $init_1, c_i, d_{i,i}$und $step_i$ bei der Compilierung bekannten Werte.

4.3.2 Reduktion der Modulo-Ausdrücke

Die ACM Optimierung (*algebraic cost minimisation)* hat die Minimierung der Anzahl vorhandener modulo-Ausdrücke zum Ziel. Da NOSR (4.3) zusätzlichen Berechnungsaufwand erzeugt und die endgültige Registerallozierung durch Einführung neuer Variablen erschwert, ist es sinnvoll, vor der eigentlichen Transformation die Anzahl an vorhandenen modulo Ausdrücken zu minimieren.

Die Idee dieser ACM Optimierung besteht in dem Erkennen gemeinsamer Faktoren der modulo Ausdrücke. Es werden ausgewählte Ausdrücke durch arithmetische Transformationen erweitert bzw. umgeformt, so daß sie aus anderen modulo Operationen bestehen. Dies erzeugt zunächst mehr modulo Ausdrücke, wobei jedoch weniger unterschiedliche Ausdrücke vorkommen. Diese wenigen unterschiedlichen Operationen werden andernorts vorausberechnet und mit CSE in den erweiterten Ausdrücken eingesetzt.

Beispiel 27 *Codeausschnitt des* cavity-detector mit vier modulo Ausdrücken

```
for (x=...) {
   array1[x % 3];       // M1
   ...
   array1[(x + 4) % 3];   // M2
   ...
   array2[(x + 1) % 3];   // M3
   ...
   array2[(x + 2) % 3];   // M4
   ...}
```

Wir lassen die Ausdrücke M1 und M3 unverändert und versuchen nur, M2 und M4 zu transformieren.

$$\begin{aligned}(x+4)\,\%\,3 &= (x\,\%\,3 + 4\,\%\,3)\,\%\,3 \\ &= (x\,\%\,3 + 1)\,\%\,3 \\ &= (x\,\%\,3 + 1\,\%\,3)\,\%\,3 \\ &= (x+1)\,\%\,3\end{aligned}$$

Damit läßt sich $(x+4)\%3$ (M2) durch $(x+1)\%3$ (M3) ausdrücken. Die

Transformation von M4 ist eine modulo Erweiterung.

$$(x + 2)\,\%\,3 = 3 - x\,\%\,3 - (x + 1)\,\%\,3$$

Somit läßt sich M4 durch M1 und M3 darstellen. Nach der anschließenden Anwendung von CSE haben wir nur noch zwei modulo Berechnungen.

Beispiel 28 *Codeausschnitt des* cavity-detector nach ACM und CSE Anwendung mit zwei modulo Ausdrücken)

```
for (x=...) {
   cse1 = x % 3;
   cse2 = (x +1) % 3;
   array1[cse1];      // M1
   ...
   array1[cse2];   // M2
   ...
   array2[cse2];   // M3
   ...
   array2[3-cse1-cse2];   // M4
   ...}
```

Es ist klar, daß die Transformationen der modulo Ausdrücke, wie wir sie gezeigt haben, nicht von einem Compiler während der CSE Optimierung entdeckt werden können. Es gibt allerdings Sonderfälle, die auch von einem Compiler bewältigt werden könnten. Wenn die modulo Konstante eine 2-er Potenz ist, läßt sich die Berechnung durch eine billigere Maskieroperation ersetzen.

Beispiel 29 *Äquivalente C-Ausdrücke bei Modulodivision mit einer 2-er potenz*

```
x % m; // m ist eine 2-er potenz   ===>   x & (m-1); // bitweise AND
```

Die arithmetischen Optimierungen erfordern keine aufwendigen Analysen im Daten- bzw. Kontrollfluß des Programms. Sie haben einen sehr lokalen Charakter. Eine Anwendung als Quellcodetransformation ist ebenfalls problemlos möglich.

4.3.3 Bewertung

Die Transformationen der modulo Ausdrücke zielen darauf ab, die modulo Operationen gänzlich zu eliminieren. Sie erzeugen dabei zusätzlichen Aufwand - neue Variablen, if-Anweisungen, Zuweisungen und Inkrementbefehle. Mit arithmetischen Transformationen der ACM Optimierung wird zuerst die Anzahl der modulo Ausdrücke minimiert. Die ACM Optimierung wird nicht durch die Mustererkennung erfaßt. Sie kann daher nicht mit den von uns vorgestellten Methoden automatisiert werden. Danach wird auf die restlichen modulo Operationen die eigentliche Transformation angewendet. Für den aus dieser Optimierung resultierenden Gewinn ist die Tatsache, daß die Berechnung von modulo Instruktionen auf der Zielarchitektur durch Bibliotheksfunktionen stattfindet, entscheidend.

Für den von uns betrachteten durch DTSE erzeugten Code ist vor allem der einfache Fall einer Schleife mit den Grenzen $[0 \ldots N]$ und einer Schrittweite von 1 relevant. Die Struktur des Moduls in den modulo Ausdrücken beschränkt sich in den meisten Fällen der von uns betrachteten Codebeispiele auf die Formen i, $i + k$ und $i * k$, für die wir Transformationen vorgestellt haben. Die Transformationen lassen sich leicht anwenden und benötigen einfache Analysen. Diese sollten für eine automatisierte Anwendung der Transformationen sicherstellen, daß es sich um modulo Berechnungen auf der Indexvariablen der Schleife handelt. Ferner sollten die oben angesprochenen Bedingungen an das Modul bzw. Schrittweite der Schleife geprüft werden. Schließlich sollte die Struktrur der modulo Ausdrücke erkannt werden können.

4.4 Common Subexpression Elimination

Die CSE (*common subexpression elimination*) Optimierung ist eine Standardmethode in der Compileroptimierung [1]. Sie ist effizient und gut erforscht. Auch die speziellen Aspekte der eingebetteten Systeme im Bezug auf CSE wurden untersucht [13]. Die Idee der CSE Optimierung besteht im Zwischenspeichern der vorausberechneten Werte für deren spätere Wiederverwendung. Nach dem wohlbekannten Prinzip - Geschwindigkeit gegen Platz - geht mit der Anwedung von CSE ein erhöter Speicherverbrauch einher. Wir wollen im folgenden die Möglichkeiten der CSE-Optimierung auf der Quellcodeebene aufzeigen. Das allgemeine Prinzip ist im Beispiel 30 dargestellt. In Anlehnung an dieses Beispiel legen wir einige Begriffe fest:

CSE-Ausdruck ist der durch eine neue Variable zu ersetzende Ausdruck; `b*i`

CSE-Variable bezeichnet die neu eingeführte Variable, in der der Wert des CSE-Ausdruckes zwischengespeichert wird; `tmp`

CSE-Komponenten sind die Variablen, Konstanten und Funktionsaufrufe, aus denen der CSE-Ausdruck besteht; `b, i`

Beispiel 30 *CSE über die Grenzen der Basisblöcke hinweg*

```
a = b*i; // ANW1
...; // SEQ1
if(...) {
   ...; // SEQ2
   c = b*i; } // ANW2
```

(a)

```
tmp = b*i;
a = tmp; // ANW1
...; // SEQ1
if(...) {
   ...; // SEQ2
   c = tmp; // ANW2}
```

(b)

4.4.1 Analyse

Im Gegensatz zu den bisher vorgestellten Quellcodetransformationen sind an CSE strengere Bedingungen geknüpft, die mit dem Datenfluß des Programms zusammenhängen. Ohne eine aufwendige, profiling-basierte Datenflußanalyse des Programms (die wir bei unserem Ansatz der Mustererkennung vermeiden wollen) sind wir auf Informationen angewiesen, die man alleine aus dem Betrachten des Quellcodes gewinnen kann. Das Sicherstellen der Korrektheit bei einer globalen CSE Anwendung - über die Grenzen eines Basisblocks hinweg - ist dabei kein triviales Problem. Die Analyse, die nur auf der Betrachtung und Interpretation des Code beruht, muß u.a. folgende Eigenschaften prüfen:

- Eine CSE-Ausdruck darf seinen Wert nicht ändern. Der Codeblock zwischen zwei Referenzen der CSE-Variablen darf keine Definitionen von CSE-Komponenten enthalten.

- Der CSE-Ausdruck darf keine Funktionsaufrufe enthalten. Es ist anderenfalls ohne weitere Analyse der Funktion nicht klar ob, die Funktion die CSE-Komponenten ändert.

- Die CSE-Komponenten dürfen keine Zeiger sein. Man kann nicht ohne einer umfaßenden Analyse alle Referenzen auf eine Speicherlokation ermitteln.
- Wenn ein CSE-Ausdruck eine bestimmte Arrayreferenz, z.B. `arr[i]` enthält, darf in dem darauffolgenden Code überhaupt kein Schreibzugriff auf den Aray erfolgen. Wenn es einen Schreibzugriff, z.B. auf `arr[x]` gäbe, müsste durch eine Analyse geprüft werden, ob `x!=i` ist. Da x ein beliebiger Audruck sein kann, wäre eine derartige Analyse sehr komplex.

4.4.2 Transformation

Das größte Optimerungspotential liegt sicherlich innerhalb der Schleifen. Wir können also nach Unterausdrücken suchen, die z.B. die Iterationsvariable der Schleife enthalten. Damit haben wir eine strukturelle Vorgabe, die sich als Muster modellieren läßt (Bsp. 31).

Beispiel 31 *vor der CSE Optimierung in einem Schleifenkörper*

```
for( i=...) {
   A[i+4] = ...; // EXP1
   ...; // SEQ1
   ...=(i+4) * f(b*(i+4)); // EXP2
   ...; // SEQ2
   ...= b*(i+4); // EXP3
   B[i+4]=...}// EXP4
```

Wir nehmen an, das Muster beschreibt eine Schleife mit einer Iterationsvariablen i und einer Anweisungsliste l im Schleifenkörper. Das Muster passt zu dem Code, wenn innerhalb der Liste l die Variable i in einer Operation mit einem anderen Operanden - Variable oder Konstante - auftritt und bis zum Ende des Schleifenblockes der gleiche Ausdruck mindestens noch einmal vorkommt. Die Anwendung einer Transformation, so sie durch die Prüfung der Bedingungen an SEQ1 und SEQ2 erlaubt ist, reduziert den Ausdruck $i + 4$ (Bsp. 32).

Beispiel 32 *nach der CSE Optimierung in einem Schleifenkörper*

```
for( i=...) {
   int tmp=i+4;
```

```
   A[tmp] = ...; // EXP1
   ...; // SEQ1
   ...=tmp * f(b*tmp); // EXP2
   ...; // SEQ2
   ...= b*tmp; // EXP3
   B[tmp]=...}// EXP4
```

Da die neue *tmp* Variable von uns erzeugt wurde, stellt sie ein bekanntes strukturelles Element dar, das uns einen neuen Ansatzpunkt zur weiteren Anwendung der Transformation bietet. Durch Einführung der Variablen `tmp2` kann der Ausdruck `b*tmp` ebenfalls reduziert werden (Bsp. 33).

Beispiel 33 *Wiederholte Anwendung von CSE in einem Schleifenkörper*

```
for( i=...) {
   int tmp=i+4;
   int tmp2=b*tmp;
   A[tmp] = ...; // EXP1
   ...; // SEQ1
   ...=tmp * f(tmp2); // EXP2
   ...; // SEQ2
   ...= tmp2; // EXP3
   B[tmp]=...}// EXP4
```

4.4.3 Bewertung

Mit der CSE Optimierung auf der Quellcodeebene können in einem weiten Codebereich Unterausdrücke reduziert werden. Die Analyse des Codes muß potentielle Korrektheitsverletzungen durch die Transformation anhand der Abschätzung von möglichen Seiteneffekten verhindern. Es ist eine sehr defensive Analyseart, die wir im Rahmen der Mustererkennung anwenden. Wenn sich alleine aus der Semantikanalyse des Quellcodes eine potenzielle Verletzung der CSE Bedingungen ergibt, wird die Transformation nicht durchgeführt.

Eine iterative Anwendung der CSE Transformation kann auch komplexere Ausdrücke reduzieren. Da diese Art der CSE Transformationen in einem Ausdrucksbaum (s.Abb. 17) einer Anweisung immer nur den kleinsten Unterbaum, in dem die gesuchte Variable vorkommt, betrachtet, erzeugt sie bei mehrmaliger Anwendung u.U. viele neue Variablen. Eine Ausweitung der Suche auf Ausdrücke mit mehreren Operationen würde dies verhindern, gleich-

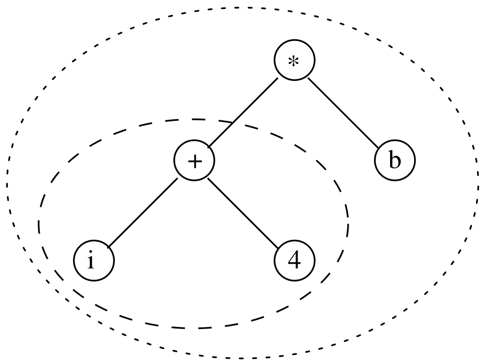


Abbildung 17: Ausdrucksbaum der Anweisung (i+4)*b

zeitig aber die Anzahl gefundener Unterausdrücke verkleinern. In diesem Fall müßte die Menge der reduzierten Unterausdrücke gegen deren Komplexitäten abgewogen werden.

Die einmalige Anwendung der CSE Optimierung mit Hilfe von Mustererkennung im Quellcode kann nicht eine vollständige Transformation des Quellcodes, in der alle vorkommenden Teilausdrücke ersetzt werden, durchführen. Die Musterspezifikation legt eine Erwartung an die Struktur des Quellcodes fest. Es wird z.B. in dem Muster spezifiziert, daß wir eine Schleife erwarten, innerhalb derer nach Ausdrücken mit der Indexvariablen gesucht werden soll. Eine Formuliereung in der Art

1. *Finde alle Unterausdrücke in der Funktion f(),*
2. *Prüfe alle Bedingungen,*
3. *Ersetze alle in 1. gefundenen Unterausdrücke, die in den größeren Ausdücken vorkommen, durch neue CSE-Variablen,*

ist dagegen viel zu weit gefasst. Die mustergestützte Quellcodetransformation stellt lediglich einen Prozeßschritt dar, der von einer höheren Instanz, z.B. einer Entwicklungsumgebung oder einem Expertensystem, als eine atomare Aktion eingesetzt werden kann. Durch wiederholte Anwendung einer Transformation mit ggf. neuen Mustern können einzelne Schritte automatisch durchgeführt werden (vergl. hierzu Bsp. 32 und Bsp. 33). Der gesamte Optimierungsprozeß muß aber von der übergeordneten Instanz kontrolliert werden. Vor diesem Hintergrund betrachten wir die CSE Optimierung immer im Rahmen einer vorgegebenen Programmstruktur.

5 Mustererkennung & Code Transformation Tool (CTT)

Im Kapitel 4 haben wir ausgewählte Quellcodeoptimierungen vorgestellt, die sich durch bestimmte Transformationen realisieren lassen. Dabei haben wir die Optimierungen analysiert, ohne konkrete Modellierungen durch Quellcodemuster anzugeben. In diesem Kapitel wollen wir ein Werkzeug vorstellen (*code transformation tool* (CTT)), das eine Darstellung und Manipulation des Codes mit Hilfe von Mustern erlaubt.

Da das CTT auf der SUIF-Bibliothek [9] des SUIF Compiler Systems (*stanford university internal format* (SUIF)) basiert, geben wir nach der allgemeinen Einführung in CTT eine kurze Übersicht über die SUIF Bibliothek. Danach stellen wir die Transformationssprache von CTT vor, mit der wir konkrete Quellcodemuster formulieren werden. Zum Schluß dieses Kapitels geben wir Beispiele von Quellcodetransformationen an.

5.1 CTT

Das CTT - entwickelt an der TU Delft (NL) im Rahmen des Projekts MOVE - ist ein Werkzeug, das automatische Quellcode-zu-Quellcode Transformationen von ANSI-C Programmen ermöglicht. CTT besitzt eine Transformationssprache, mit der sich Muster, Bedingungen und Transformationen formulieren lassen. Diese Sprache besteht aus sogenannten Metaelementen, die als C-Makros realisert sind. Als Schnittstelle besitzt CTT eine graphische Oberfläche (QT) und die eine *transformation engine* in Form des Programms ctt. Die Arbeitsweise von ctt läßt sich in drei Phasen unterteilen:

1. Zuordnen von Quellcodemustern zu den Quellcodeelementen - *pattern matching*

2. Verifikation der vorgegebenen Bedingunen - *condition checking*

3. Anwendung der Transformationen und die Generiereung des neuen Quellcodes - *transformation*

Entsprechend der obigen Aufzählung werden wir im folgenden die funktionalen Einheiten von ctt als *matcher, conditioner* und *transformer* bezeichnen (Abb. 18).

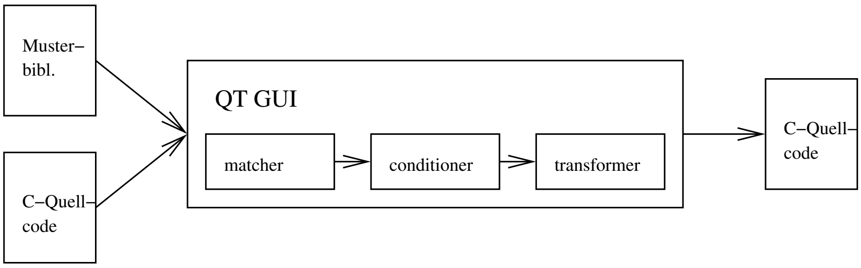


Abbildung 18: Architektur des CTT

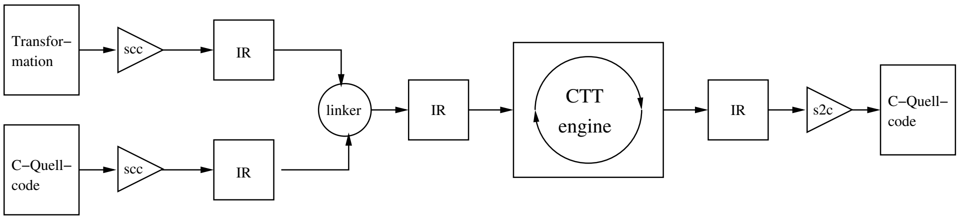


Abbildung 19: Transformationsablauf in ctt

Alle drei Verarbeitungsphasen arbeiten auf Basis einer IR Darstellung der SUIF-Bibliothek. Mit dieser *high level* IR ist eine detaillierte Darstellung der Programm- und Strukturelemente möglich. Nach einer Übersetzungsphase, die die Musterdatei und die Quellcodedatei in die IR Darstellung überführt, werden die SUIF-Objektdateien von einem Binder (*suif linker*) in ein SUIF Programm integriert. In einem gemeinsamen Namensraum können dann z.B. die Variablen den Mustern zugeordnet werden bzw. neue Symbole in den Namensraum des Programms eingefügt werden. Das Prüfen der Bedingungen durch den *conditioner* teilt sich in die statische Analyse der Eigenschaften von Programmelementen und in die dynamische Analyse des Datenflußes auf. Die Datenflußanalyse wird allerdings von externen Tools mittels Laufzeitanalysen (*profiling*) durchgeführt und stellt somit keinen integralen Bestandteil von CTT dar. Nach der erfolgten Transformation wird mit einem Generator (*s2c*) das Programm aus der IR Darstellung zurück in C überführt. Der Ablauf wird in Abb. 19 veranschaulicht.

5.2 SUIF Repräsentation

Das SUIF Compilersystem, das an der Universität von Stanford entwickelt wurde, ist eine Umgebung für die Entwicklung von Compilern. Durch den modularen Aufbau der einzelnen Arbeitsphasen des Compilers lassen sich leicht alternative bzw. neue Implementierungen hinzufügen. Die Kommunikation zwischen den einzelnen Arbeitsphasen ist durch den Dateizugriff realisiert, wodurch sich eine sehr lose Kopplung ergibt.

Die IR von SUIF - implementiert von der SUIF Bibliothek in der objektorientierten Weise - ist eine Coderepräsentation, die *low level* ebenso wie *high level* Repräsentationen des Programmcodes unterstützt. So ist z.B. eine Sequenz von abstrakten Maschineninstruktionen als Liste darstellbar oder aber eine - in Form eines Ausdrucksbaumes - komplexe arithmetische Anweisung. Ebenso sind die Elemente einer Hochsprache wie Schleifen, if-Anweisungen oder ganze Funktionen mit ihren Attributen modellierbar.

Ein Programm in der SUIF Darstellung besteht aus einer Hierarchie von Knoten, die entsprechende Komplexitätsstufen repräsentieren. Die Objekte einer höheren Stufe enthalten die Objekte der untergeordneten Stufe.

5.2.1 Dateiebene

Die oberste Ebene bilden die Objekte *file set* und *file set entry*. Die Klasse *file set* stellt eine Menge von Quellcodedateien dar. Sie enthält neben der Menge von einzelnen Dateien eine globale Symboltabelle, deren Gültigkeitsbereich sich auf alle Dateien erstreckt. Die Klasse *file set entry* enthält eine private Symboltabelle, die nur innerhalb der Datei bekannt ist, sowie eine Liste von den in der Datei definierten Prozeduren. Dank dieser Konstruktion lassen sich zwischenprozedurale Analysen durchführen. Als Operationen bieten diese Klassen das Einlesen und Wegschreiben der Programmdarstellung von bzw. in die Dateien. Ferner wird, durch Iteratoren der Symboltabellen, der Zugriff auf die Prozeduren realisiert.

5.2.2 Prozedurebene

Eine Datei enthält eine Liste von Prozedurknoten. Zur Darstellung des Prozedurcodes stehen die *high level* Repräsentation in Form von Bäumen, die besser für strukturelle Analysen von z.B. Datenflüssen geeignet sind, und die *low level* Repräsentation als Listen, die für skalare Analysen und Codegenerierung prädestiniert ist, zur Verfügung. Der AST (*abstract syntax tree*) stellt die Struktur des Codes in Form von o.g. Bäumen und Listen dar. Knoten wie Blöcke, Schleifen und if-Anweisungen enthalten Listen von Anweisungsknoten, an denen Ausdrucksbäume als Blätter hängen. Es gibt folgende Knoten, die einen AST bilden:

- Blockknoten: Sie stellen z.B. den C-Codebereich zwischen den {} Klammern dar. Blockknoten enthalten eine Symboltabelle und die Liste von untergeordneten AST-Knoten. Eine Sonderform des Blockknotens ist der Prozedurknoten. Er enthält einige zusätzliche Methoden und eine andere Symboltabelle. Der Prozedurknoten stellt die Wurzel eines AST's dar.

- Konditionalknoten: Diese Knoten stellen die if-Anweisungen dar. Sie enthalten drei Listen von AST-Knoten für den *if-header,* den *then*-Teil und den *else*-Teil.

- Schleifenknoten: Es gibt zwei Arten von Schleifenknoten. Den *for node* für Schleifen mit skalaren Iteratoren und einigen zusätzlichen Bedingungen und die *loop nodes,* die allgemeine „*do-while*" Schleifen abbilden. Schleifenknoten besitzen zwei Listen von AST-Knoten, eine für den Code der Abbruchbedingung und eine für den Schleifenkörper.

- Instruktionsknoten: Diese als Blätter eines AST-Baumes fungierenden Objekte beinhalten einen Ausdrucksbaum bzw. eine einzelne Instruktion. Es kann in einem Instruktionsknoten zwischen einer Listen- und einer Baumdarstellung des Ausdruckes geschaltet werden.

5.2.3 Instruktionsebene

Der Instruktionknoten ist die Wurzel eines Ausdrucksbaums, der eine einzelne Instruktion darstellt. Im einfachsten Fall handelt es sich um eine RISC-artige Maschineninstruktion. Aber auch komplexere Operationen wie Arrayzugriffe, switch-Anweisungen oder Funktionsaufrufe können als Einzelinstruktionen vorkommen. Im allgemeinen Fall enthält ein Instruktionsknoten einen Ausdrucksbaum, alternativ eine Instruktionsliste.

Der wesentliche Unterschied zwischen der Baum- und der Listendarstellung besteht in der Zugriffsmethode auf die einzelnen Elemente wie Instruktionen, Operanden und Symbole. Während die Elemente einer Liste jeweils einen AST-Knoten darstellen, wodurch der sequentielle Zugriff die totale Ausführungsordnung wiederspiegelt, ist bei einem Ausdrucksbaum nur der Wurzelknoten - sprich die erste Instruktion - ein AST-Knoten. Alle weiteren Knoten können nur mit Hilfe von Zeigern auf die Operanden der jeweiligen

Instruktionen erreicht werden. Ein Ausdrucksbaum repräsentiert somit eine partielle Ausführungsordnung auf den Instruktionen.

Beispiel 34 *Eine Anweisung (a) in C (b) als Ausdrucksbaum und (c) als flache Instruktionsliste.*

```
j = ( i + (--b)) % 4;                                     (a)

sub t:g4 (i.32) main.b = main.b, e1                       (b)
   e1: ldc t:g4 (i.32) 1
rem t:g4 (i.32) main.j = e1, e2
   e1: add t:g4 (i.32) main.i, main.b
   e2: ldc t:g4 (i.32) 4

ldc t:g4 (i.32) nd#8 = 1                                  (c)
sub t:g4 (i.32) main.b = main.b, nd#8
add t:g4 (i.32) nd#10 = main.i, main.b
ldc t:g4 (i.32) nd#11 = 4
rem t:g4 (i.32) main.j = nd#10, nd#11
```

In dem Beispiel 34 *(b)* ist eine Anweisung als ein Ausdrucksbaum dargestellt. Die Quelloperanden der Instruktion `rem` sind Zeiger auf weitere Instruktionen - `add` und `ldc`. Der Zieloperand ist das Symbol der Variablen `j`. Die Instruktion `add` nimmt wiederum zwei Variablensymbole als Quelloperanden, während die Ladeinstruktion `ldc` eine Konstante, die Zahl 4, als Quelloperanden hat. Die Typbezeichnungen hinter den Instruktionsnamen beziehen sich auf die Zieloperanden. Der Typ `t:g4` bezeichnet dabei eine 32-bit breite Integerzahl. Die Instruktionsliste in 34 *(c)* zeigt die Ausführungsreihenfolge, die die Berechnung des Ausdruckes in *(a)* darstellt. Es werden automatisch temporäre Variablen erzeugt, die die Ergebnisse vorausgehender Instruktionen speichern.

Zwischen den beiden Darstellungsarten kann mit den Methoden `flatten` bzw. `cvt_to_tree` umgeschaltet werden. Es muß jedoch klar sein, daß die beiden Darstellungen nicht äquivalent sind. Die Ausführungordnung einer Liste ist eine totale. Wenn man z.B. in der Liste, die vorher aus einer Baumdarstellung erzeugt wurde, eine Instruktion aus anderen Ausdrucksbäumen einfügt und dann zurück in die Baumdarstellung wechselt, wird SUIF in der Regel automatisch eine neue temporäre Variablen einfügen müssen, um die Semantik zu erhalten. Bei der partiellen Ausführungsordnung eines Baumes ist ein enfaches Verschieben von Ausdrucksknoten zwischen den Unterbäumen nicht möglich. Ein Ausdrucksunterbaum, z.B. für einen Operanden einer Instruktion, muß vollständig ausgewertet werden, eher die Auswertung des Unterbaums für den nächsten Operanden beginnnt. Oder anders ausgedrückt, es kann während der Auswertung, z.B des linken, Unterbaumes keine Sprünge in den rechten Unterbaum geben.

5.2.4 Symbolebene

Für die Beschreibung der Typen steht ein C ähnliches Typensystem zur Verfügung. Die Definitionen von Symbolen und deren Typen befinden sich in Symboltabellen, die ebenfalls in einer Hierarchie angeordnet sind. Analog zu der Blockstruktur gibt es eine globale Symboltabelle, die zu dem *file set* Objekt gehört. Jede Symboltabelle enthält Zeiger auf die Symboltabellen der untergeordneten Elemente der Blockhierarchie. Erwartungsgemäß behalten die Symbole einer übergeordneten Tabelle ihre Gültigkeit in den hierarchisch tiefer liegenden Blöcken.

Die Symbole werden eindeutig mit einer ID bzw. einem Namen bezeichnet. Es gibt drei Arten von Symbolen, die entsprechend Variablen, Marken und Prozeduren identifizeren. Variablensymbole können in jeder Symboltabelle definiert werden. Sie beschreiben den Namen, den Typ und andere Eigenschaften - wie Speicherbelegung - einer Variablen. Die Markensymbole sind nur in Prozeduren zulässig und sind zur Stellenmarkierug im Code vorgesehen. Die Prozedursymbole schließlich werden nur in den globalen Symboltabellen definiert. Sie beschreiben eine Prozedur und ihren Typ. Außerdem beinhalten sie einen Zeiger auf den Codeblock der Prozedur.

5.2.5 Annotationen

Die Erweiterungsfähigkeit des SUIF Compilersystems wird durch sogenannte *annotations* sichergestellt. Es sind Objekte, die beliebige benutzerdefinierte Strukturen aufnehmen können, die dann einem SUIF Objekt zugeordnet werden können. Auf diese Weise können Informationen zwischen verschiedenen Verarbeitungsphasen des Compilers ausgetauscht werden. Die Annotationen speichern entweder eine Liste von einfachen Werten oder sie enthalten Zeiger auf komplexere Strukturen. Außerdem können sie in der Datei gespeichert und wieder eingelesen werden.

5.3 Die Transformationssprache von CTT

Wir haben bisher die Struktur des CTT Systems vorgestellt. Im folgenden geben wir eine Einführung in die bereits erwähnte Transformationssprache. Eine Musterdatei enthält C-Quellcode, der um sogenannte Metaelemente erweitert ist. Die Metaelemente sind übliche C-Makros, die bei der Übersetzung mit *scc* die eigentlichen Muster in der IR Darstellung ergeben.

Eine Transformation wird in einer Datei beschrieben. Die Datei enthält drei Quellcodeblöcke, die jeweils dem Muster, den Bedingungen und der Transformation entsprechen.

```
PATTERN
{ ... } // Musterbeschreibung
CONDITIONS
{ ... } // Bedingungen
TRANSFORMATION
{ ... } // Spez. der Transformationen
```

Diese Metaelemente dürfen in einer Datei nur einmal vorkommen. Der Block **CONDITIONS** kann auch ganz entfallen. Darüberhinaus gibt es noch zwei Metaelemente **NAME** und **DESCR**, welche die Datei benennen bzw. beschreiben. Bevor wir weitere Metaelemente kennen lernen, werden wir, um eine Begriffsgrundlage zu schaffen, das Modell der Quellcodemuster einführen.

5.3.1 Das Modell der Quellcodemuster

Eine formale Beschreibung der Struktur und der Elemente eines C-Quellcodes kann in Form von Mustern erfolgen. Die Grundlage der Musterformulierung bilden die Strukturelemente der Sprache C innerhalb einer Funktion. Ein Muster ist eine Menge von Musterelementen, die folgende Objekte darstellen:

- Variablen
- Zuweisungsanweisungen
- Listen von Anweisungen
- if-Anweisungen

- Konditionalausdrücke; Ausdrücke in if-Anweisungen, die einen Wahrheitswert liefern
- Funktionsaufrufe
- Schleifen
- Schleifengrenzen; spezielle Konditionalausdrücke
- Schleifenzähler; Anweisungen, die das Fortzählen einer for-Schleife realisieren
- allgemaine Ausdrücke; z.B. arithmetische (Unter-) Ausdrücke

Die Musterelemente sollten den Instanzen der oben aufgezählten Objekte zugeordnet werden, sofern die Instanzen mit ihren Attributen dem Muster entsprechen. Ein Musterelement für die Codeobjekte läßt ihr Vorkommen und deren Eigenschaften spezifizieren. Ein allgemeines Muster beispielsweise, das jede C-Funktion beschreibt, muß lediglich eine Liste von Anweisungen beschreiben. Ein Muster, das eine for-Schleife mit den Grenzen zwischen 0 und 100 beschreibt, paßt auf alle Funktionen, die mit einer Anweisungsliste beginnen - auch mit for-Schleifen, die andere Grenzen aufweisen - gefolgt von einer for-Schleife mit den genannten Grenzen und ggf. mit einer weiteren Anweisungsliste enden. Dabei zählt der Schleifenkörper zu der Schleife als deren Attribut.

Im Mustermodell des CTT wird eine enge strukturelle Vorgabe für die for-Schleifen gemacht. Ohne jetzt auf die Details einzugehen, wollen wir Schleifen, die vom CTT als for-Schleifen dargestellt werden, *wohl geformte* Schleifen nennen. Alle anderen Schleifen werden durch die allgemeine *do-while* Schleife repräsentiert.

5.3.2 Variablen

Das Metaelement **VAR** beschreibt eine einfache Variable oder auch ein Array beliebigen Typs. In den Mustern werden die Variablen ohne Typ spezifiziert, um allgemeine Muster formulieren zu können. Ebenso ist die Arraygröße irrelevant für die Musterbeschreibung, was durch die Verwendung des Metaelements **DONT_CARE** ausgedrückt wird.

Beispiel 35 *Das Metaelement VAR deklariert Variablen*

```
EINGABE                              MUSTER

int x, arr[100];                     VAR i, a[DONT_CARE];
```

Die Mustererkennung bindet die Variable `x` an `i`. Ab dieser Stelle ist in der weiteren Musterbeschreibung mit dem Namen `i` die Quellcodevariable `x` gemeint. Da die Bindung über den Namen geschieht, wird jedes Vorkommen des Musterelements als gleiche Variable erkannt. Es ist allerdings in C erlaubt, gleiche Namen für verschiedene Variablen zu benutzen, wenn sie in verschiedenen Blöcken definiert sind. Aus diesem Grund sind die globalen Variablen im Muster als *static* zu deklarieren, um nicht mit globalen Variablen im Eingangsquellcode zu kollidieren.

5.3.3 Ausdrücke

Audrücke sind Teile von Anweisungen. Es stehen zwei Metaelemente zur Verfügung, um Ausdrücke zu spezifizieren. Das Element **EXPR** steht für beliebige Ausdrücke. Das **STEP_EXPR** Element bezeichnet den Ausdruck zum Aktualisieren der Zählervariablen einer for-Schleife. Um die Ausdrücke zu unterscheiden, wird für diese eine Nummer vergeben.

Beispiel 36 *Zuordnung der Ausdrücke zu einem EXPR-Metaelement*

```
EINGABE                              MUSTER

a = 2*(b+arr[i%9])-4.3;              x = 2*(EXPR(2))-4.3;
c = b+arr[i%9] + a;                  y = EXPR(2) + EXPR(1);
a = 0;                               EXPR(1) = 0;
```

Der Ausdruck `b+arr[i%9]` im Beispiel 36 wird als EXPR(2) und die Variable `a` als EXPR(1) erkannt. Die gesamte Eingabe paßt zum Muster, da in der zweiten Zeile ebenfalls EXPR(1) bzw EXPR(2) im richtigen Kontext vorkommen. Die Reihenfolge der Bindung von Ausdrücken an die EXPR Elemente ist beliebig. Die Variable `a` wird erst in der zweiten Zeile mit EXPR(1) verbunden. Eine Zeile wie z.B. `c = b+arr[i%9]-1` paßt nicht mehr zu der zweiten Zeile im Muster, womit die ganze Eingabe das Muster nicht erfüllt ist. Die EXPR Elemente können sowohl auf der linken (*left-hand-side expression*) als auch auf der rechten (*right-hand-side expression*) Seite einer Zuweisung stehen.

Beispiel 37 *STEP_EXPR wird an i+1 gebunden*

```
EINGABE                        MUSTER

for( i=0; i<100; i++) {        for(x=0; x<100; STEP_EXPR(x,1)) {
   c = f(b);                      EXPR(2) = f(b);
   A[i] = c;                      A[x] = EXPR(2);
}                              }
```

Das Element STEP_EXPR(x,1) wird an die rechte Seite der Zuweisung in `i++` gebunden. Der Ausdruck `i++` wird nämlich in der SUIF Darstellung als `i=i+1` repräsentiert. Die linke Seite wird als Attribut der for-Schleife in dem AST-Knoten gespeichert. Neben der Nummer des Ausdruckes gibt es einen weiteren Parameter, `x`, der sicherstellt, daß die Schleifenvariable in dem Ausdruck vorkommt. Die Schleife im Beispiel 37 paßt zu dem Muster für eine for-Schleife weil sie *wohl geformt* ist. Dies bedeutet, daß der Zähler - ein Skalar - im Schleifenkopf initialisiert wird, die Abbruchbedingung ein Konditionalausdruck, der die Indexvariable mittels eines logischen Operators vergliechen wird und das Fortzählen eine Zuweisung an die Iteratorvariable impliziert. Ferner wird die Zählervariable nicht in dem Schleifenkörper verändert.

5.3.4 Konditionalausdrücke

Speziell für die Formulierung der Konditionalausdrücke gibt es weitere Metaelemente **COND** und **BOUND**. Das Element COND beschreibt die Bedingungen in den if-Anweisungen und den while-Schleifen. In den for-Schleifen muß das Element BOUND verwendet werden.

Beispiel 38 *Konditionalausdrücke in if-, while- und for-Anweisungen.*

```
EINGABE                     MUSTER

while( a<(x+2) ) {          while(COND(1)) {
   for(i=0; i<10; i++) {       for(x=0; BOUND(x,2); STEP_EXPR(x,1)) {
      if(i<5)                     if(COND(3))
         c = f(b);                   EXPR(2) = f(b);
      A[i] = c;                   A[x] = EXPR(2);
   }                           }
...;                        }
}
```

Wie bei den allgemeinen Ausdrücken, wird bei den Konditionalausdrücken für beide Metaelemente ein gemeinsamer Nummernbereich benutzt. Ebenso gibt es für die for-Schleife das Element BOUND, das die Bindung der Abbruchbedingung an die Iterationsvariable erzwingt.

5.3.5 Anweisungen und Anweisungslisten

Eine sinnvolle [4] Anweisung in C kann kann eine Zuweisung sein, eine Schleife, eine if-Anweisung oder ein Funktionsaufruf. Mit dem Metaelement **STMT** wird eine Anweisung dargestellt. Eine Folge von Anweisungen kann mit dem Element **STMTLIST** erfasst werden.

Beispiel 39 *Anweisungen und Anweisungslisten*

```
EINGABE                       MUSTER

a = 10;                       a = 10;
for(i=0; i<10; i++) {         STMT(1);
   if(i<5)                    STMTLIST(1);
      c = f(b);               if(COND(1)) {
}                                STMTLIST(2);
A[i] = c;                     }
B[i] = c*2;
if(a==0) { ... } // mind. eine Anweisung
```

Die Anweisung `a=10` wird als solche erkannt. Die for-Schleife wird dann an STMT(1) gebunden. Zwei nachfolgende Zuweisungen `A[i]...`, `B[i]...` werden mit STMTLIST(1) verknüpft. Sie stellen danach insofern eine Einheit dar, als daß sie wieder erkannt werden können, wenn sie in genau der gleichen Konstellation auftreten. Die letzte if-Anweisung enthält eine weitere Anweisungsliste STMTLIST(2).

5.3.6 Funktionsaufrufe

Die Funktionsaufrufe werden als Anweisungen von einem speziellen Metaelement **PROCCALL** dargestellt. Mit diesem Element läßt sich der Aufruf einer Funktion an einer Stelle im Code beschreiben. Ferner kann durch ein weiteres Muster die aufgerufene Funktion spezifiziert werden.

Beispiel 40 *Funktionsaufruf und Funktionsmuster.*

[4] Ein Ausdruck wie z.B. a+2; kann zwar auch als Anweisung im Code vorkommen, er stellt aber keine Aktion dar. Es wird bei dessen Auführung nichts verändert.

```
EINGABE
...;
ret = calculate(a, b, c);
...;
int calculate(int _a,
              float _b,
              int _c) {
   if(_a > 10)
      return _b * _c;
   else return _c;
}
```

```
MUSTER
static void func_muster(int dummy,
            VAR x, VAR y, VAR z);
...
r = PROCCALL((1, x, y, z),
             func_muster);
...
int func_muster(int dummy, VAR k,
                VAR l, VAR m){
   if(COND(1))
      return l*m;
   else STMT(1);
}
```

Das Element PROCCALL((1,x,y,z),func_muster) ist ein Funktionsaufrufmuster mit der Nummer 1 und zwar für eine Funktion, die drei Argumente nimmt und durch das Muster `func_muster` spezifiziert wird. Es ist zwingend notwendig, die Deklaration der Funktion im Muster als `static` anzugeben, um die SUIF Datei mit anderen Mustern konfliktfrei binden zu können. Die Definition von `func_mster` bekommt als ersten Parameter ein Zusatzargument für die Nummer des PROCCALL Elementes. Die formalen Parameter _a, _b, und _c in der Definition der `calculate` Funktion werden an die Musterparameter k, l und m gebunden. Die aktuellen Parameter a, b und c in dem Aufruf von `calculate` dagegen an x, y und z.

Es ist möglich, die PROCCALL Elemente innerhalb der Funktionsmuster zu plazieren.

5.3.7 Weitere Metaelemente

Das Metaelement **MARK** setzt im Muster eine Markierung an der unmittelbar folgenden Anweisung, so daß diese Anweisung in dem TRANSFORMATION bzw. CONDITIONS Block mit dem gleichen MARK Element referenziert werden kann. Damit kann eine Anweisung als ganzes beschrieben werden, was ansonsten durch das STMT Element passiert, und man zusätzlich weitere Bedingungen an die innere Struktur der Anweisung stellen kann. Benutzt man z.B. für eine for-Schleife das STMT Element, kann nicht mehr mit COND die Bedingung referenziert werden. Ein anderes Beispiel ist eine Anweisung, in der man einen Ausdruck spezifizieren möchte. Um auf die Anweisung in dem CONDITIONS Block als ganzen zugreifen zu können, muß sie mit dem MARK Element markiert werden. Auf die im Beispiel 41(a) mit MARK(1) markierte if-Bedingung, kann später innerhalb des TRANSFORMATION Blockes mit MARK(1) zugegriefen werden. Ebenso kann (Bsp. 41(b)) die Zeile `a=EXPR(1)+3;`, die mit dem MARK(1) Element markiert wird, ähnlich wie mit dem STMT(1) Element im weiteren Mustercode referenziert werden.

Beispiel 41 *if-Anweisung markieren mit MARK (a) Anweisung mit einem konkretem Ausdruck (b)*

```
MARK(1);
if(COND(1)) {
   STMTLIST(1);
}
```

(a)

```
MARK(1);
a = EXPR(1) + 3;
```

(b)

Ein weiteres spezielles Element ist **FORCE_BINDING**. Es erlaubt eine explizite Bindung einer bekannten Variablen an ein VAR Element. Ohne FORCE_BINDING im Beispiel 42 wird `x=sin(i)` an `a=sin(i)` gebunden. Will man jedoch die beiden Anweisungslisten an der Stelle `b=sin(i)` trennen, muß x an b explizit gebunden werden.

Beispiel 42 *Explizite Bindung mit FORCE_BINDING*

```
EINGABE
...;
a = sin(i);
b = sin(i);
...;
```

```
MUSTER
VAR x;
FORCE_BINDING("x","b");
STMTLIST(1);
x = sin(i);
STMTLIST(2);
```

5.3.8 Metaelemente in Transformationen

Außer den bisher vorgestellten Elementen, mit denen auf die gebundenen Codeobjekte Bezug genommen werden kann, gibt es Metaelemente, die nur im TRANSFORMATION Block angewendet werden. Es gibt drei Formen von Metaelementen zur Erzeugung neuer Variablen. Die Beispiele zeigen ihre Benutzung. Das Element VAR muß ebenfalls noch vor jedem CREATE_VAR_FROM_x angegeben werden.

CREATE_VAR_FROM_PROC("x",1) erzeugt eine neue Variable x vom Typ des Rückgabewertes der Funktion, die an PROCCALL(1) gebunden ist.

CREATE_VAR_FROM_STRING(”x”,”int”) erzeugt eine neue Variable x. Der Typ wird explizit als Zeichenkette in der C Notation angegeben. In dem Beispiel also `int`.

CREATE VAR FROM VAR(”x”,”y”) erzeugt eine neue Variable vom gleichen Typ wie die Variable y.

5.4 Bedingungen im CONDITIONS-Block

Wie bereits erwähnt, gibt es einen Block CONDITIONS, in dem Bedingungen an die Quellcodestrukturen formuliert werden, die an die Metaelemente im PATTERN Block gebunden sind. Nachdem also der *matcher* die Muster in dem Code erkannt hat, und bevor der *transformer* den neuen Code generiert, muß der *conditioner* die Bedingungen prüfen.

5.4.1 Boolesche Ausdrücke

Dies sind Funktionen, die logische Verknüpfungen der Bedingungen erlauben. Der Ausdruck **not(<bedingung>)** negiert eine Bedingung. Das Makro **and(<bedingung>, <bedingung>)** drückt eine UND Verknüpfung aus und **or(<bedingung>, <bedingung>)** entsprechend die ODER Verknüpfung. Als **<bedingung>** können unter anderem auch **not**, **or** und **and** Ausdrücke vorkommen, wodurch sich komplexe Bedingungen formulieren lassen.

```
CONDITIONS { and(expr_is_var(1), expr_is_var(2)); }
```

5.4.2 Strukturelle Bedingungen

Es gibt eine Reihe von Funktionen, die als Argument die Nummer eines Codeobjekts nehmen und eine festgelegte Eigenschaft dieses Objektes testen.

expr_is_proccall(n) bzw. stmt_is_proccall(n) sind als Bedingung erfüllt, wenn das an EXPR(n) bzw. STMT(n) gebundene Objekt ein Funktionsaufruf ist.

expr_is_constant(n) ist erfüllt, wenn EXPR(n) eine Konstante ist.

bound_is_constant(n) ist erfüllt, wenn die Bedingung in BOUND(x, n) ein Vergleich zwischen x und einer Konstanten ist.

expr_is_var(n) ist erfüllt, wenn EPR(n) eine Variablenreferenz darstellt.

stmt(list)_has_no_return(n) ist erfüllt, wenn STMTLIST(n) bzw. STMT(n) keinen `return` Befehl enthält. Da STMT(n) auch an eine if- oder for-Anweisung - mit den dazugehörigen Blöcken - gebunden sein kann, steht STMT nicht unbedingt nur für eine Anweisung.

stmt(list)_has_no_unsafe_jumps(n) ist erfüllt, wenn in den Anweisungen von STMT(n) bzw. STMTLIST(n) keine `break`, `continue` oder `goto` Befehle vorkommen.

5.4.3 Datenflußbedingungen

Da die Datenflußanalysen in CTT profilinigbasiert sind und zusätzliche externe Tools benötigen, betrachten wir sie nicht näher. Vollständigkeitshalber geben wir jeoch ein Beispiel an, um die Verwendung zu veranschaulichen.

```
CONDITONS { not(dep("anti between expr 1 and stmt 1")); }
```

Die Funktion `dep` wertet den Ausdruck aus, der nach einer Grammatik die Datenflußabhängigkeiten beschreibt. In diesem Fall soll keine Antidatenabhängigkeit zwischen `expr 1` und `stmt 1` bestehen.

5.5 Quellcodetransformationen

Nachdem wir die Transformationssprache von CTT vorgestellt haben, wollen wir einige Quellcodetransformationen durchführen. Wir werden Transformationen für die im Kapitel 4 vorgestellten Optimierungen angeben. Dabei wollen wir untersuchen, welche Erweiterungen in der Transformationssprache ggf. notwendig wären.

5.5.1 Konditionalausdrücke in Schleifen

Wir erinnern uns an die Optimierung der Konditionalausdrücke innerhalb von verschachtelten Schleifen, die über mehrdimensionale Arrays iterieren. Wir werden im weiteren die Transformationen an einem vereinfachten, d.h. eher unrealistischem, Codebeispiel durchführen, um die Vorgehensweise besser veranschaulichen zu können.

Beispiel 43 *Zwei Konditionalausdrücke mit Schnittbereich*

```
 for(a=0; a < 1000; a++) {
   if(a>70 && a<900) { cnt1++; }
   if(a>4 && a<800) { cnt2++; }
}
```

Wir formulieren jetzt ein Muster, mit dem sich der Code aus Beispiel 43 erkennen und in eine optimierte Form transformieren läßt.

Beispiel 44 *Transformation der Konditionalausdrücke*

```
   PATTERN {
   VAR x;
   for(x=EXPR(1); BOUND(x,1); STEP_EXPR(x,2)) {
      if(COND(1)) {
         STMTLIST(1);
      }
      if(COND(2)) {
         STMTLIST(2);
      }
   }
TRANSFORMATION {
VAR x;
for(x=EXPR(1); BOUND(x,1); STEP_EXPR(x,2)) {
   if(COND(1) && COND(2))
      for(i=i; a<800; STEP_EXPR(x,2)) {
         STMTLIST(1);
         STMTLIST(2);
      }
   if(COND(1)) { STMTLIST(1); }
   if(COND(2)) { STMTLIST(2); }
}}
```

Nach dem Transformationslauf erhalten wir folgenden automatisch generierten Code.

Beispiel 45 *Konditionalausdrücke nach der Transformation*

```
for (a = 0; a < 1000; a++) {
   if (70 < a && a < 900 && 4 < a && a < 800) {
      for (a = a; a < 800; a++) {
         cnt1++;
         cnt2++;
      }
```

```
    }
    if (70 < a && a < 900) { cnt1++; }
    if (4 < a && a < 800) { cnt2++; }
}
```

Nun haben wir zwar die Optimierung angewendet, jedoch mußten wir dabei die Abbruchbedingung der neuen inneren Schleife manuell einsetzen. Außerdem enthält die verkettete if-Bedingung alle Konditionalausdrücke, obwohl an dieser Stelle die Grenzen - bestehend aus der maximalen unteren bzw. minimalen oberen Schranke - des gemeinsamen Schnittbereichs ausreichend gewesen wären (s. Bsp. 45).

Ohne die Kenntnis der Konstanten in den Konditionalausdrücken läßt sich diese Transformation nicht automatisch durchführen. Die eigentliche Erkennung des Schnittbereichs zwischen den beiden Konditionalausdrücken und entsprechende Generierung der neuen Schleifengrenze kann nur durch eine Erweiterung um neue Metaelemente erreicht werden. Wir werden im Kapitel 6 eine entsprechende Erweiterung vorstellen.

Ein weiteres Problem der Transformation im Beispiel 44 ist die Spezifikation der Bedingungen der if-Anweisungen mittels der COND Elemente. Die Transformation ist nur dann korrekt, wenn COND an Vergleiche der Schleifenvariablen mit Konstanten bzw. UND Verkettungen solcher Vergleiche gebunden ist. Auch hierfür sollte ein neues Element eingeführt werden, das nur derartige Konditionalausdrücke als passend erkennt.

Um das Muster so allgemein wie möglich zu halten, wären zwischen den von uns vorgegebenen Metaelementen weitere STMTLIST Ausdrücke nötig für das Erkennen der gesuchten Struktur innerhalb von beliebigen Codesequenzen. Da STMTLIST auch leere Listen zulässt, würde so ein Muster auch zu dem Beispiel 43 passen. Dieser Vorgehensweise sind allerdings enge Grenzen gesetzt. So muß die Struktur der Schleifenverschachtelung explizit angegeben werden. Auch die erwartete Sequenz der if-Anweisungen muß angegeben werden. Dies stellt eine grundsätzliche Einschränkung der Transformationssprache dar, durch die eine Spezifikation von unbestimmten Objektsequenzen oder Objektverschachtelungen nicht möglich ist.

Ein allgemeineres Muster, das ein dreidimensionales Zugriffsschema spezifiziert sieht wie im Beispiel 46 aus.

Beispiel 46 *Transformation der Konditionalausdrücke in einer dreifach verschachtelten Schleifengruppe*

```
PATTERN {
VAR x,y,z;
STMTLIST(10);
for(x=EXPR(1); BOUND(x,1); STEP_EXPR(x,2)) {
   for(y=EXPR(3); BOUND(y,2); STEP_EXPR(y,4)) {
      for(z=EXPR(5); BOUND(z,3); STEP_EXPR(z,6)) {
         STMTLIST(11);
         if(COND(1))
            { STMTLIST(1); }
         STMTLIST(12);
         if(COND(2))
            { STMTLIST(2); }
        STMTLIST(13);
}  }  }
STMTLIST(14);}
TRANSFORMATION {
VAR x,y,z;
STMTLIST(10);
for(x=EXPR(1); BOUND(x,1); STEP_EXPR(x,2)) {
   for(y=EXPR(3); BOUND(y,2); STEP_EXPR(y,4)) {
      for(z=EXPR(5); BOUND(z,3); STEP_EXPR(z,6)) {
         if(COND(1) && COND(2)) { /* Schnittbereich für 3 Dim. */
            for(z=z; BOUND(z,3); STEP_EXPR(z,6)) {
               STMTLIST(11);
               STMTLIST(1);
               STMTLIST(12);
               STMTLIST(2);
               STMTLIST(13);
         }  }
         STMTLIST(11); // ELSE BEGIN
         if(COND(1))
            { STMTLIST(1); }
         STMTLIST(12);
         if(COND(2))
            { STMTLIST(2); }
         STMTLIST(13);  // ELSE END
}  }  }
STMTLIST(14);}
```

Wenn `COND(1)` und `COND(2)` eine UND Verkettung von Vergleichen zwischen x, y oder z und einer Konstanten darstellen, dann drückt `COND(1) && COND(2)` die Bedingung für den Schnittbereich im Iterationsraum mit den Dimensionen x, y, z aus. Da wir noch keine neuen Metaelemente kennen, gilt die Transformation aus Beispiel 46 mit den schon dargestellten Einschränkungen. So gilt diese Transformation nur, wenn der Iterationsbereich über der Variablen z die größere obere Schranke hat als der durch `COND(1) && COND(2)` definierte. Im anderen Fall muß der Anweisungsblock zwischen //ELSE BEGIN und //ELSE END in einen `else`-Block eingeschlossen werden. Dies verhindert, daß diese Anweisungsliste nach der letztet Iteration der innersten Schliefe, die dann auch die letzte Iteration der z-Schleife wäre, zweimal ausgeführt werden würde.

5.5.2 Eliminierung der Modulo Ausdrücke

Die im Absatz 4.3 vorgestellte Transformation zum Eliminieren der modulo Ausdrücke kann nur für einen konkret angegebenen modulo Ausdruck durchgeführt werden (s. Bsp. 47). Außerdem sind nur Schleifen mit Schrittweite 1 erlaubt, da man ansonsten die neue Schrittvariable nicht korrekt initialisieren kann. Dies begründet sich darin, daß es keine Möglichkeit gibt, aus dem Element STEP_EXPR den Wert der Schrittweite zu extrahieren.

Beispiel 47 *Musterbeschreibung für die modulo Optimierung*

```
PATTERN {
VAR x, arr[DONT_CARE];
for(x=EXPR(1); BOUND(x,1); x++)
  {STMTLIST(10);
   EXPR(3)=arr[x % EXPR(4)];
   STMTLIST(11);
   }
}
```

```
TRANSFORMATION {
VAR x,xv,arr[DONT_CARE];
CREATE_VAR_FROM_VAR("xv","x");
xv=EXPR(1) % EXPR(4)
for(x=EXPR(1); BOUND(x,1); x++)
  {STMTLIST(10);
   if(xv >= EXPR(4)) xv -= EXPR(4);
   EXPR(3)=arr[xv];
   xv++;
   STMTLIST(11);
  }
}
```

Als Bedingung in dem CONDITIONS Block muß auf jeden Fall sichergestellt werden, daß `EXPR(4)` und `EXPR(1)` eine Konstante darstellen.

```
CONDITIONS { and(expr_is_constant(1),expr_is_constant(4)); }
```

Eine Erweiterung der Transformationssprache in Bezug auf diese Optimierung sollte vor allem folgende Möglichkeiten bieten:

1. Erkennen von modulo Audrücken an einer beliebigen Stelle innerhalb einer Anweisung. Dabei sollen die Ausdrücke auf die im Abschnitt 4.3 vorgestellten Formen beschränkt werden: `i%c, (i+b)%c, (i*b)%c`.

2. Zugriff auf das Modul der %-Operation in dem erkannten Ausdruck.

3. Zugriff auf den Wert der Schrittweite in dem Ausdruck von STEP_EXPR.

Wenn man davon ausgeht, daß die Transformationen innerhalb von *wohl geformten* Schleifen durchgeführt werden und die modulo Ausdrücke Operationen auf den Indexvariablen sind, könnte man die erste Erweiterung auf die STMTLIST Elemente ausdehnen. Da die Struktureigenschaft der Schleife die Änderung der Indexvariablen im Schleifenkörper verbietet, könnten alle gleichen modulo Ausdrücke ersetzt werden, ohne daß zusätzliche Analysen wie in der CSE Optimierung nötig wären.

5.5.3 Common Subexpression Elimination

Die CSE Optimierung stellt im Vergleich zu den bisher gezeigten Transformationen die größten Ansprüche an die Analyse des Codes. Da das EXPR Metaelement keinerlei explizite Angaben über die Eigenschaften eines Ausdruckes zuläßt und außerdem die Angabe der EXPR Elemente eine konkrete Struktur vorgibt, ist die CSE Transformation nicht als ein allgemeines Muster formulierbar.

Eine Einschränkung der Ausdrücke auf eine klar definierte Menge ist der erste Schritt, um die CSE Optimierung mit Hilfe eines Musters zu erfassen. Ferner muß ein Mechanismus zur Verfügung gestellt werden, um eine Such- und Ersetzoperation in einer Codesequenz durchführen zu können. Da das STMTLIST Element eine nicht näher spezifizierte Anweisungssequenz darstellt, ist es ein guter Kanditat für eine Erweiterung. Folgendes Muster sollte danach formuliert werden können:

```
PATTERN {                      TRANSFORMATION {
 EXPR(1) = EXPR(2);             CSE=EXPR(2);
 STMTLIST(1);                   EXPR(1)=CSE;
}                              /*STMTLIST mit CSE <-> EXPR(2) */
                                STMTLIST(1);
                               }
```

Ein Analyse im CONDITIONS Block muß ggf. `STMTLIST(1)` für die Anwendung von CSE prüfen. Auch hierfür ist eine Erweiterung der Transformationssprache notwendig.

6 Erweiterungen des CTT

Im vorigen Kapitel haben wir versucht, die Transformationen für ausgewählte Optimierungen in der Transformationssprache des CTT zu formulieren.

Dabei haben wir erkannt, daß ohne bestimmten Erweiterungen, die Ausdrucksmöglichkeiten für Musterdarstellung sehr beschränkt sind. Wir werden im folgenden die im Rahmen dieser Arbeit implementierten Erweiterungen vorstellen. Daneben wollen wir konkrete Vorschläge für den weiteren Funktionsausbau diskutieren.

6.1 Neue Metaelemente

Um die Möglichkeiten der Mustererkennung in der Transformationssprache des CTT zu erweitern, wurde eine Reihe neuer Metaelemente implementiert. Wir stellen sie jeweils im Kontext der Quellcodeoptimierung vor, für die diese Elemente eine Musterformulierung ermöglichen sollen, dar.

6.1.1 Konditionalausdrücke in Schleifen

COND_VAR

Um die Erkennung der &&-Ketten von Vergleichsoperationen zu ermöglichen, wird ein Metaelement COND_VAR eingeführt. Das Element erkennt logische Vergleichsoperationen zwischen einer Iterationsvariablen und einer Konstanten. Die Nummer des COND_VAR Elements identifiziert eine if-Bedingung. Der zweite Parameter ist die erwartete Indexvariable einer umschließenden Schleife. Folgendes Muster bindet `COND_VAR(1,x)` an die Bedingung der if-Anweisung im Eingangscode:

```
for(x=EXPR(1); BOUND(x,1);
     STEP_EXPR(x,2))
   if(COND_VAR(1,x)) ...;
```

```
for(i=0; i<100; i++)
   if(i<20) ...;
```

Die Bedingung in der if-Anweisung kann eine beliebige &&-Verketteung von Vergleichen sein. Wenn das Metaelement COND_VAR(1,x) lautet, muß es in der Kette mindestens einen Vergleich geben, der die Variable x als Operanden hat. Alle Vergleiche, die nicht x enthalten, müssen Vergleiche anderer Indexvariablen sein. Es dürfen also keine Vergleiche vorkommen, die nicht mindestens eine Indexvariable als Operanden haben. Das Beispiel 48 veranschaulicht diesen Sachverhalt.

Beispiel 48 *Verwendung des COND_VAR Elementes. C1 paßt zum Muster. C2 und C3 aber nicht.*

```
VAR x,y,v;

for(y=EXPR(1); BOUND(y,1);
     STEP_EXPR(y,2)) {
   for(x=EXPR(3); BOUND(x,2);
        STEP_EXPR(x,4)) {
      if(COND_VAR(x,1)) ...;
      if(COND_VAR(y,2)) ...;
      if(COND_VAR(x,3)) ...;
```

```
int i,j,k;

for(j=0; j<100; j++) {
   for(i=0; i<200; i++) {
      if(i>5 && i<80 && j<2) ...; C1
      if(i<90) ...; C2
      if(i>2 && k>1) ...; C3
```

Der linke Teil zeigt eine Musterbeschreibung. Es werden drei if-Anweisungen erwartet, die &&-Ketten von Vergleichen als Bedingung haben. Die Elemente mit der Nummer 1 und 3 müssen mindestens eine Bedingung für die Indexvariable der inneren Schleife beinhalten. Dies ist eine notwendige und hinreichende Bedingung, um dem Muster zu entsprechen. Das Element mit der Nummer 2 ist analog mit der äußeren Schleife verbunden. Im rechten Teil des Beispiels 48 sind drei if-Anweisungen aufgeführt. Die Anweisung C1 paßt zum Element COND_VAR(x,1), da sie nur Vergleiche mit Indexvariablen und mindestens einen Vergleich mit der Variablen *i*, die im Muster an x gebunden ist, enthält. Die Anweisung C2 enthält zwar einen Indexvergleich, es gibt aber keinen Vergleich mit der Variablen *j*, die an *y* gebunden ist. C2 paßt also nicht zum Muster. Und schließlich schlägt auch die Erkennung der Anweisung C3 fehl. Sie enthält den richtigen Vergleich mit der Indexvariablen *i*, hat aber darüber hinaus einen Vergleich mit der Variablen *k*, die keine Indexvariable der umschließenden Schleifen ist.

COND_EXPR

Nachdem mit dem Metaelement COND_VAR die &&-Ketten in den if-Anweisungen erkannt wurden, muß eine neue &&-Kette für die Einstiegsbedingung der neuen Schleife gebildet werden. Wir führen hier das COND_EXPR Metaelement ein. Dieses Element wird nur in dem TRANSFORMATION Block verwendet. Es bekommt als Argument den Namen einer Indexvariablen. Der generierte C-Code ist die &&-Kette von Bedingungen, die den gemeinsamen Iterationsbereich aller if-Anweisungen bzgl. dieser Indexvariablen definieren. Die Transformation

```
if(COND_EXPR(x))      C-Quellcode --> if(i>2 && i<99)
```

würde stattfinden, wenn x an i gebunden ist und die Bedingungen `i>2`, `i<99` den gemeinsamen Iterationsbereich der Indexvariablen i über allen if-Bedingungen definieren, die den COND_VAR Elementen zugeordnet sind.

BOUND_VAR

Um die Abbruchbedingung der gemeinsamen inneren Schleife zu ermitteln, definieren wir das Metaelement BOUND_VAR. Auch dieses Element ist nur für den TRANSFORMATION Block bestimmt. Das Argument gibt die Variable an, für die die kleinste obere Schranke aus allen COND_VAR Bedingungen ermittelt werden soll. Es liefert im Moment das Minimum aller Konstanten aus den $<$-Operationen mit der entsprechenden Indexvariablen. Es funktioniert nur für den Fall vorwärts iterierenden Schleifen.

```
if(COND_EXPR(x))       C-Quellcode -->   if(i>2 && i<99)
   for(x=x; BOUND_VAR(x); x++) ...;        for(i=i;i<99;i++) ...;
```

Eine zusätzliche Funktion **cond_expr_has_intersect()** für den CONDITIONS Block untersucht den gemeinsamen Iterationsbereich aller if-Bedingungen an die gleiche Indexvariable. Für den Fall, daß es keinen Schnittberech gibt, wird die Transformation nicht angewendet. Eine mögliche Erweiterung wäre die Prüfung gegen einen Schwellenwert für die Größe des Schnittbereichs. Obige Funktion wäre die richtige Stelle für derartige Tests.

Mit den neu eingeführten Metaelementen können wir jetzt eine Quellcodetransformation für die Optimierung der Konditionalausdrücke in Schleifen durchführen (s. Bsp. 49).

Beispiel 49 *Zwei eingebettete Schleifen mit if-Bedingungen an die Indexvariablen*

```
EINGANGSCODE

for(j=0; j<100; j++) {
   for(k=0; k<1000; k++) {
      if(k>5 && k<900) ...;
      if(k>10 && k<995) ...;
      if(j>3 && j<90 && k<800) ...;
      if(j>5 && j<99) ...;
}  }
```

```
PATTERN {
VAR x,y;
for(x=EXPR(1); BOUND(x,1);x++) {
   for(y=EXPR(3); BOUND(y,2);y++) {
      if(COND_VAR(1,x)) ...;//STLS1
      if(COND_VAR(2,x)) ...;//STLS2
      if(COND_VAR(3,y)) ...;//STLS3
      if(COND_VAR(4,y)) ...;//STLS4
     }
}}
```

```
AUSGANGSCODE

for(j=0; j<100; j++) {
   for(k=0; k<1000; k++) {
      if(k>10 && k<800 &&
         j>5 && j<90) {
         for(k=k;k<800;k++) {
            ...; // STLS1
            ...; // STLS2
            ...; // STLS3
            ...; // STLS4
         }
      }
      if(k>5 && k<900) ...;
      if(k>10 && k<995) ...;
      if(j>3 && j<90 && k<800) ...;
      if(j>5 && j<99) ...;
}  }
```

```
CONDITIONS {
   cond_expr_has_intersect();
}
TRANSFORMATION
VAR x,y;
for(x=EXPR(1); BOUND(x,1);x++) {
   for(y=EXPR(3); BOUND(y,2);y++ {
      if(COND_EXPR(x) &&
         COND_EXPR(y)) {
         for(y=y; y<BOUND_VAR(y);y++) {
            ...; // STLS1
            ...; // STLS2
            ...; // STLS3
            ...; // STLS4
         }
      }
      if(COND_VAR(1,x)) ...; //STLS1
      if(COND_VAR(2,x)) ...; //STLS2
      if(COND_VAR(3,y)) ...; //STLS3
      if(COND_VAR(4,y)) ...; //STLS4
     }
}}
```

Da es sich nur um Vorwärtsschleifen handelt, haben wir die Zählanweisungen mit `x++, y++` explizit angegeben. Die Initialisierung der Iterationsvariablen `y=y` ist für die Korrektheit zwar unnötig, sie muß aber aufgrund der Strukturvorgabe für for-Schleifen der SUIF Darstellung trotzdem eingefügt werden.

6.1.2 Eliminierung der Modulo Ausdrücke

Um allgemeine Muster für die Ausdrücke mit %-Operatoren formulieren zu können, führen wir ein Metaelement **STMT_REM(n,x)** ein. Es erkennt eine einzelne Anweisung, in der ein modulo Operator vorkommt. Es ist unerheblich, ob der Operator in einem LHS (*left hand side*) oder RHS (*right hand side*) Ausdruck steht. Das erste Argument ist wieder die eindeutige Nummer, die aus dem Nummernkreis der STMT Elemente stammt. Das zweite Argument ist die Variable, die als Modul in der %-Operation steht. Folgendes Beispiel zeigt die Bedeutung von STMT_REM, wenn x an i gebunden ist.

```
STMT_REM(1,x); paßt auf --> a=i%3; oder a[i%10]=c; oder a=f(i%4);
```

Das Element STMT_REM bindet nur das erste Vorkommen eines modulo Ausdrucks. Es hat in dem PATTERN Block eine andere Wirkung als im TRANSFORMATION Block. Während im ersten nur das Erkennen des Ausdrucks geschieht, wird im zweiten auch die eigentliche Ersetzung der modulo Operation durch die Zählvariable durchgeführt.

Da das Modul der %-Operation in der Transformation verwendet wird, brauchen wir eine Möglichkeit, diese Konstante aus dem Ausdruck extrahieren zu können. Hierzu dient als weiteres Element, **MODUL(x)**. Es wird nur in dem TRANSFORMATION Block eingesetzt. Als Parameter nimmt es die Nummer der Anweisung, die an STMT_REM(x) gebunden war.

Eine ähnliche Funktion hat ein ebenfalls neues Element **STEP_VAL(x)**. Dessen Einführung war zwar durch die modulo Transformation motiviert, es hat aber einen allgemeinen Charakter und kann bei allen Transformationen, die mit for-Schleifen umgehen, eingesetz werden. Es liefert einen skalaren Wert der Schrittweite in einem Ausdruck des STEP_EXPR(x,i) Elements. Voraussetzung für die Korrektheit bei der Verwendung von STEP_VAL in dieser Transformation ist, daß die Schrittweite eine Konstante und die Operation eine Inkrement bzw. Dekrement Instruktion ist. Ist die Schrittweite in STEP_EXPR durch einen beliebigen Ausdruck gegeben, darf die Transformation nicht ohne weitere Analyse von STEP_EXPR angewendet werden. Es gibt keine Funktionen des CONDITIONS Blocks, die diese Bedingungen prüfen würden. Sie müssen daher beim Erkennen des Musters zugesichert werden.

Im Beispiel 50 wird eine komplette modulo Transformation durchgeführt. Hierbei ist ein besonderes Augenmerk auf die Verwendung von `STMT_REM(1,x)` im TRANSFORMATION Block zu werfen. Das Element gibt die zugehörige Anweisung aus und modifiziert diese dabei. Es ist eine Art eines aktiven Elements, das bisher in der Transformationssprache nicht vorhanden war.

Beispiel 50 *Ersetzung eines Moduloausdrucks durch Zählvariable*

```
EINGANGSCODE

int i,arr[10];
for(i=0; i<20; i++) {
   ...;
   arr[i%9]=...;
   ...;
}}
```

```
PATTERN {

VAR x;
for(x=EXPR(1); BOUND(x,1);
     STEP_EXPR(x,2)) {
   STMTLIST(1);
   STMT_REM(1,x);
   STMTLIST(2);
}}
```

```
AUSGANGSCODE

int i,arr[10];
int iv,ivs;

iv = 0;
ivs = 1;
for(i=0; i<20; i++) {
   ...;
   if(9 <= iv) { iv = iv - 9; }
   arr[iv] = ...;
   iv = iv + ivs;
   ...;
}}
```

```
TRANSFORMATION {

VAR x,iv,ivs;
CREATE_VAR_FROM_VAR("iv","x");
CREATE_VAR_FROM_VAR("ivs","x");

iv = EXPR(1) % MODUL(1);
ivs = STEP_VAL(2) % MODUL(1);

for(x=EXPR(1); BOUND(x,1);
     STEP_EXPR(x,2)) {
   STMTLIST(1);
   if(iv >= MODUL(1))
      iv-=MODUL(1);
   STMT_REM(1,x);
   iv += ivs;
   STMTLIST(2);
}}
```

Die neu eingeführte Variable *ivs* enthält einen konstanten Wert. Eine Erweiterung des Trasnformationsmechanismus könnte den Wert ausrechnen, und an die Stelle der *ivs* Variablen einsetzen. Auf der anderen Seite werden solche Konstellationen von nachfolgenden Compileroptimierungen erkannt und wegoptimiert. Die Optimierungen *constant folding* und *copy propagation* könnten hier angewendet werden.

6.1.3 CSE

Die Methode für die Implementierung der CSE Optimierung ähnelt der für die Eliminierung der Modulo Ausdrücke. Es wird zuerst ein Ausdruck mit einer vorgegebenen Variablen erkannt. Hierfür führen wir ein neues Element **EXPR_VAR(n,x)** ein. Neben der Nummer, die zu dem Nummernkreis der EXPR Elemente gehört, wird als Parameter die Variable übergeben, die in dem gesuchten Ausdruck vorkommen muß. Dabei muß die Unterscheidung, ob es sich um einen *LHS* oder *RHS* Ausdruck handelt, anhand einer entsprechenden Plazierung des Elements gemacht werden. Eine CSE Ersetzung auf der linken Seite ist z.B. für Anweisungen wie `arr[a*b]=...;` möglich.

```
(1) EXPR(1)=EXPR_VAR(2,x);  passt zu -> a=b*i; oder a=(2+b)*(i+f());
(2) a[EXPR_VAR(1,x)]=EXPR(2); passt zu -> a[(2+b)%i+2]=f();
```

In den obigen Beispielen ist x an i gebunden. Dabei wird diejenige Operation ermittelt, die als den Operanden die Variable i hat. Im Beispiel (1) lautet der Ausdruck `b*i` bzw. `i+f()`. Im Beispiel (2) `(2+b)%i`.

Um nun die eigentliche Ersetzung des erkannten Ausdruckes durchzuführen, wird das STMTLIST Element zu einem aktiven Element erweitert. In dem TRANSFORMATION Block eingesetzt, prüft es, ob es bisher einen passenden Ausdruck zu einem EXPR_VAR Element gibt. Ist dies der Fall, ersetzt STMTLIST in jeder zu seiner Liste gehörenden Anweisung den Ausdruck von EXPR_VAR durch eine neue CSE-Variable. Das Element EXPR ersetzt den als Unterausdruck vorkommenden EXPR_VAR Ausdruck mit der CSE_Variablen.

Das EXPR_VAR Element liefert im TRANSFORMATION Block nur den kleinsten Unterausdruck mit der Operation, für die die gesuchten Variable einen Operanden darstellt.

Beispiel 51 *Transformation für die CSE Optimierung*

```
EINGANGSCODE
int a;

for(i=0; i<1000; i++) {
   ...;
   j=a/(3+k) + f();
   arr[a/(3+k)]=j;
   l=j-(l+a/(3+k));
   a+=1;
   j=a/(3+k);
}
```

```
PATTERN {
VAR y,x;

for(x=EXPR(1); BOUND(x,1);
     STEP_EXPR(x,2)) {
   STMTLIST(1);
   EXPR(3)=EXPR_VAR(4,y);
   STMTLIST(2);
}}
```

```
AUSGANGSCODE
 int a,cse;

for(i=0; i<1000; i++) {
   ...;
   cse=a/(3+k);
   j=cse + f();
   arr[cse]=j;
   l=j-(l+cse);
   a+=1;
   j=a/(3+k);
}
```

```
TRANSFORMATION {
VAR x,y,CSE;
CREATE_VAR_FROM_VAR("CSE","y");

for(x=EXPR(1); BOUND(x,1);
     STEP_EXPR(x,2)) {
   STMTLIST(1);
   CSE=EXPR_VAR(4,y);
   EXPR(3)=EXPR(4);
   STMTLIST(2);
}}
```

Die CSE Variable wird mit dem Unterausdruck initialisiert. Das Element `EXPR(4)` liefert dagegen den kompletten Ausdruck, wie er im PATTERN Block von EXPR_VAR erkannt wurde, wobei die CSE Variable den Unterausdruck innerhalb von `EXPR(4)` ersetzt. Das nachfolgende `STMTLIST(2)` Element führt auf die gleiche Weise in jeder Anweisung die Ersetzung durch. Verändert eine der Anweisungen eine in dem CSE-Ausdruck vorkommende Variable, so wird ab diesem Punkt keine Ersetzung mehr durchgeführt. Die Prüfung der Bedingung für die CSE Anwendung findet nicht in dem CONDITIONS Block statt. Da die Analyse, wie auch die Transformation einen sehr lokalen Charakter haben, ist es aus Effizenzgründen geschickter, beides erst bei der Codegenerierung durchzuführen und ggf. die weitere Anwendung der Transformation zu stoppen. Eine Prüfung im CONDITIONS Block wäre ungleich komplizierter in der Implementierung gewesen.

6.2 Andere Erweiterungsempfehlungen

6.2.1 Strukturvorgabe im Muster

Eine große Einschränkung der Transformationssprache von CTT ist die relativ starre Vorgabe der Programmstruktur. Es ist schwierig, Muster für ein reales Beispiel zu formulieren, ohne vorher das Eingangsprogramm untersucht zu haben. Oft ist auch ein manuelles „Frisieren” des Codes nötig, um die von CTT erwarteten Eigenschaften herbeizuführen.

So kann keine Folge von Codeobjekten erkannt werden, wenn nicht explizit die Metaelemente in bestimmter Anzahl aufgeführt werden. Dies wirkt sich unmittelbar auf die Anwendbarkeit der Transformation aus Abschnitt 6.1.1 aus, die die Optimierung der Konditionalausdrücke in Schleifen durchführen. Werden mehr if-Anweisungen im Muster gefordert, als im Code vorhanden, schlägt die ganze Erkennung fehl. Die Transformation wird nicht ausgeführt. Sind im Code dagegen mehr passende if-Anweisungen vorhanden, werden sie nicht bei der Transformation berücksichtigt.

Eine denkbare Erweiterung für die Formulierung von Objektsequenzen - z.B. beliebig viele aufeinander folgende if-Anweisungen oder beliebig tief verschachtelte Schleifen - sollte ähnlich wie das STMTLIST Element funktionieren. Ein derartiges Element würde ab der Stelle seines Vorkommens bis zum Blockende die if-Anweisungen bzw. die for-Schleifen finden und unter der Folgenummer speichern.

6.2.2 Kontrolle der Transformationsanwendung

Eine differenzierte und flexible Anwendung der Transformationen würde eine deutlich allgemeinere Formulierungen der Transformationen erlauben. Durch die Einführung von Kontrollelementen wie Wiederholung und Kondition wären komplexe Transformationen formulierbar. Die Eliminierung der modulo Operationen könnte von der bedingten bzw. alternativen Transformationsanwendung profitieren, indem entsprechend der Eigenschaften der Schleife oder des modulo Ausdrucks passende Transformationsteile ausgewählt würden. Weiteres Beispiel, das von der automatischen Wiederholung einer Transformation profitieren würde, ist die CSE Optimierung. CSE ersetzt im Moment nur die kleinsten Unterausdrücke. Komplexere Ausdrücke können durch wiederholte Anwendung der CSE Transformation durch den Benutzer ersetzt werden. Denkbar wäre ein Mechanismus in dem *transformer*, der die CSE

	cavity_dtse.c		cavity_mod.c	
SUN	3,45	s	2,61	s
PowerPC	1,28	s	0,98	s
Pentium	2,59	s	1,16	s
ARM7 (arm)	545.799.050	Zykl.	186.060.924	Zykl.
ARM7 (thumb)	615.700.652	Zykl.	152.220.148	Zykl.
TI C6x	1.816.365.336	Zykl.	1.282.862.334	Zykl.

Tabelle 1: Eliminierung der modulo Ausdrücke

Transformationen jeweils auf die neu eingeführten CSE-Variablen anwendet.

Für Erweiterungen dieser Art wäre allerdings ein tiefer Eingriff in den Mechanismus der Zusammenarbeit zwischen dem *matcher, conditioner* und dem *transformer* vonnöten. Die Verarbeitungen der Musterdatei und des Eingangsquellcodes bei der eigentlichen Musterzuordnung zu den Codeobjekten müssten entkoppelt werden.

7 Ergebnisse und Bewertung

Wir wollen in diesem Kapitel ein Resüme der Arbeit geben. Darüber hinaus wollen wir praktische Resultate in Form von Meßergebnissen für reale Programmbeispiele zeigen, auf die die vorgestellten Quellcodeoptimierungen angewendet wurden. Danach soll eine Bewertung des vorgestellten Ansatzes der Mustererkennung gefolgt von einer Zusammenfassung das Kapitel abschließen.

7.1 Laufzeittests

Um die Effektivität der angewendeten Quellcodeoptimierungen zu demonstrieren, wurde eine Reihe von Laufzeittests durchgeführt. Als Testobjekt diente der Quellcode des früher erwähnten *cavity detector* (s. Abs. 3.2). Die Ausgangsbasis für die Anwendung der Transformationen ist eine Version des *cavity detector*, die nach der Anwendung von DTSE und einiger Standardoptimierungen (Loop Unrolling, Constant Propagation, Loop Normaliziation) entstanden ist, wodurch sie mehr Gelegenheiten für die von uns vorgestellten Quellcodeoptimierungen bietet.

Der Testablauf beginnt mit der Formulierung und Durchführung geeigneter Transformationen mit dem CTT. Eine Transformation wird ggf. iterativ

	cavity_cond.c		cavity_cse.c	
SUN	1,69	s	1,55	s
PowerPC	0,68	s	0,69	s
Pentium	1,09	s	1,10	s
ARM7 (arm)	159.460.828	Zykl.	176.216.382	Zykl.
ARM7 (thumb)	132.576.806	Zykl.	141.394.610	Zykl.
TI C6x	1.199.133.412	Zykl.	1.228.311.290	Zykl.

Tabelle 2: Optimierung der if-Ausdrücke und CSE in Schleifen

	cavity_mod.c	cavity_cond.c	cavity_cse.c
SUN	75,7 %	64,8 %	91,7 %
PowerPC	76,6 %	69,4 %	101,5 %
Pentium	44,8 %	94,0 %	101,0 %
ARM7 (arm)	34,1 %	85,7 %	110,5 %
ARM7 (thumb)	24,7 %	87,1 %	106,7 %
TI C6x	70,6 %	93,5 %	102,4 %

Tabelle 3: Relative Laufzeit bzgl. der Vorgängerversion

auf den vorhergehenden Quellcode angewendet, bis eine Optimierung vollständig durchgeführt ist.

Im nächsten Schritt wird der vom CTT erzeugte Quellcode auf mehreren Architekturen übersetzt und die Laufzeit gemessen. Es wurden folgende Plattformen benutzt, wobei - bis auf den PowerPC auf dem der gcc 2.95.4 eingesetzt wurde - jeweils die nativen Compiler der Hersteller mit der höchsten Optimierung verwendet wurden:

- SUN UltraSparc IIi, 333MHz
- Apple PowerPC, 500MHz
- Intel Pentium III, 900MHz
- ARM7TDMI, ARM-mode und THUMB-mode, 32.7MHz
- TI C6x, 133MHz

Nach dem Testlauf wird die nächste Optimierung durchgeführt. Ausgehend von der DTSE Version werden die Optimierungen in folgender Reihenfolge angewendet:

1. Eliminierung der modulo Ausdrücke.

	cav_dtse.c	cav_cse.c	Verbesserung
SUN	3,45	1,55	55,1 %
PowerPC	1,28	0,69	46,1 %
Pentium	2,59	1,10	57,5 %
ARM7 (arm)	545.799.050	176.216.382	67,7 %
ARM7 (thumb)	615.700.652	141.394.610	77,0 %
TI C6x	1.816.365.336	1.228.311.290	32,4 %

Tabelle 4: gesamte relative Verbesserung

2. Optimierung der if-Anweisungen in Schleifen.

3. CSE Optimierung auf Ausdrücken mit Indexvariablen wobei nur die kleinsten Unterausdrücke ersetzt wurden.

Es wurde durchgehend die reale Hardware - ggf. in Form von Evaluationboards - benutzt und keine Emulatoren. Gemessen wurde entweder die Laufzeit in Sekunden oder die Anzahl der Zyklen.

Die Tabellen 1 und 2 zeigen die gemessenen Werte für die Laufzeit bzw. die Zyklenanzahl. Die Tabelle 3 gibt für jede Optimierung jeweils die relative Veränderung zu der vorherigen Version wieder. In der letzten Tabelle (Tab. 4) ist die relative gesamte Verbesserung nach der Durchführung aller Optimierungen aufgeführt.

Die große Effektivität der modulo Optimierung bei dem ARM Prozessor läßt auf eine teure Realisierung der modulo Operation durch eine Bibliotheksfunktion schliessen. Auch der Pentium scheint eine wenig effiziente Implementierung der modulo Operation zu haben. Eine gute Unterstützung der modulo Instruktion in Hardware bei den anderen GP Architekturen - SUN, PowerPC- drückt sich in dem tendenziell kleineren Gewinn im Vergleich zu dem ARM System aus. Die TI Architektur reagiert anscheinend empfindlich auf die neu eingeführten Variablen.

Eine kleinere Streuung der erzielten Gewinne ergibt sich bei der Optimierung der Konditionalausdrücke. Da diese Optimierung nur den Code dupliziert und den Kontrollfluß ändert ohne neue Variablen einzuführen, ist die erzielte Verbesserung stabiler bzgl. wechselnder Architekturen. Es gibt keinen großen Einfluß auf die Cache- bzw. Registernutzung.

Die Auffällige Unwirksamkeit der CSE Optimierungen läßt sich einerseits durch die Anwendung der CSE Optimierung auf relativ kleine Unterausdrücke, die innerhalb von komplexen Ausdrücken vorkommen. Der eingesparte Berechnungsaufwand der ersetzten CSE-Ausdrücke macht nur einen kleinen Teil in den Kosten für die komplexen Ausdrücke aus. Andererseits können nur solche Architekturen von CSE profitieren, die auch mit der stark gestiegenen Anzahl neuer Variablen effizient umgehen können. Nur wenn noch genügend freie Register zur Verfügung stehen, können die teuren Speicherzugriffe vermieden werden. Man sieht ganz klar, daß die eingebetteten Architekturen hier deutlich schlechter abschneiden. Die Anwendung der CSE Optimierung im Quellcode an sich ist zwar architekturunabhängig, die erzielten Verbesserungen hängen allerdings von der Zielplatform ab.

7.2 Bewertung und Schlußbemerkungen

Die grundsäztliche Anwendbarkeit und Praxistauglichkeit der vorgestellten Methode der Quellcodeoptimierung mit Hilfe der Mustererkennung wurde in der Arbeit belegt. Mit der Optimierung auf der Quellcodeebene wird ein großes Optimierungspotential genutzt, das von der DTSE Optimierung geschaffen wurde aber für die Compileroptimierung oft nicht zugänglich ist. Einzelne Optimierungen erzielen eine Verbesserung von bis zu 75% (ARM; cavity_mod.c) bzw. 35% (SUN; cavity_cond.c). Nach der Anwendung aller von uns vorgestellten Optimierungen ergibt sich eine gesamte Verbesserugng zwischen 32% (TI) und 77% (ARM).

Es muß jedoch darauf hingewiesen werden, daß das Modell der Quellcodemuster und deren Erkennung erheblichen Einschränkungen unterliegen. Nach den Erfahrungen des Autors, die bei dem praktischen Teil dieser Arbeit gemacht wurden, ist das CTT mit seiner Transformationssprache nicht in der Lage wirklich allgemeingültige Transformationen zu formulieren, die auf beliebige, reale Quellcodebeispiele anwendbar wären. Dies begründet sich darin, daß die Programmstruktur ein integraler Bestandteil der Musterdefinition ist. Infolge dessen, wird die Vorgabe an die Programmstruktur mit der wachsenden Komplexität des Musters enger. Im allgemeinen fällt diese Einschränkung zu strikt aus. So ist es nicht möglich im Quellcode nach einem Ausdruck zu suchen. Man muß festlegen, wo der Ausdruck zu suchen ist.

Die Formulierung der Muster in CTT muß nicht nur spezifizieren wonach gesucht wird - z.B. nach dem Ausdruck `x%3`, sondern auch in welchem Kontext sich das Objekt befinden soll - ist z.B. `x%3` Teil einer Arrayadressierung. Es ist daher immer eine Analyse eines Programms notwendig, bevor eine Transformation formuliert werden kann. Ein Muster wird für einen konkreten Quellcode erstellt. Der Nutzen von CTT besteht dann in der Möglichkeit der wiederholten Anwendung der Transformation. Wenn eine neue, leicht geänderte Version des Programms vorliegt, kann eine Optimierung durch die schon vorhandene Transformation durchgeführt werden.

Ein anderes Anwendungsgebiet liegt möglicherweise in der Untersuchung einer Optimierung an sich. Hierfür wird ein Beispielcode erstellt, der diese Optimierung ermöglicht und gleichzeitig von dem CTT leicht als Muster beschrieben werden kann.

7.3 Zusammenfassung

Wir haben zuerst die Codeoptimierung für eingebettete Systeme unter dem Aspekt verschiedener Abstraktionsstufen vorgestellt und haben die daraus hervorgehenden Eigenschaften erläutert. Unter der Betrachtung der verwendeten Modelle haben wir zwischen *low level*, *medium level* und *high level* Optimierungen unterschieden. Wir sind danach detaillierter auf die DTSE Methode als *high level* Optimierung eingegangen und haben festgestellt, daß in dem post DTSE Quellcode immer noch großes Optimierungspotential vorhanden ist. Dabei handelt es sich um Optimierungen, die eher globaler Natur sind, und nicht von einem Compiler angewendet werden können. Wir haben beispielhaft drei Optimierungen auf der Quellcodeebene vorgestellt. Als möglichen Ansatz für die Formalisierung und Automatisierung der Quellcodeoptimierung haben wir die Mustererkennung und die Transformationen des Quellcodes eingeführt. Dabei haben wir ein Werkzeug - das CTT - präsentiert, das mit seiner Transformationssprache das Modell der Quellcodemuster implementiert. Es hat sich gezeigt, daß die Transformationssprache um neue Elemente erweitert werden mußte, damit die von uns betrachteten Quellcodetransformationen sich anwenden ließen. Danach haben wir die Durchführung der Transformationen vorgeführt und weitere Erweiterungen vorgeschlagen. Zum Schluß wurden einige Laufzeittests auf optimierten Programmbeispielen präsentierten, sowie eine Bewertung der vorgestellten Methodik vorgenommen.

verflixt swêr is shcreiben mir tuon schon alle finger wê
Umberto Eco, Baudolino

Abbildungsverzeichnis

Literatur

[1] A. Aho, R. Sethi, and J. Ullman. *Compiler Construktion*. Addison-Wessley, 1984.

[2] G. Araujo and S. Malik. Optimal code generation for embedded memory non-homogeneous register architectures, 1995.

[3] B. Landwehr, P. Marwedel, and R. Dömer. OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming. In *Proc. European Design Automation Conference (EURO-DAC94)*, pages 90–95, Grenoble, France, 1994. IEEE Computer Society Press.

[4] D. H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Software – Practice & Experience*, 22(2):101–110, February 1992.

[5] F. Catthoor, K. Danckaert, C. Kullkarni, and T. Omnes. Data tranfer and storage (dts) architekture issues and exploration in multimedia processors.

[6] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.

[7] Free Software Foundation. *GCC online documentation*, 2001. http://gcc.gnu.org/onlinedocs/gcc-3.0.3/gcc.html.

[8] Cédric Ghez, Miguel Miranda, Arnout Vandecappelle, et al. Systematic high-level address code transformations for piece-wise linear indexing. In *Proc. of SIPS*, Lafayette, October 2000.

[9] S. Group. *The SUIF Library: A set of core routines for manipulating SUIF data structures*. Stanford University, 1994. http://suif.stanford.edu/suif/suif1.

[10] P. Hilfinger, J. Rabaey, D. Genin, C. Scheers, and H. De Man. Dsp specification using the silage language, 1990.

[11] R. Leupers and P. Marwedel. Algorithms for address assignment in DSP code generation. In *iccad*, 1996.

[12] R. Leupers and P. Marwedel. Time-constrained code compaction for dsps. *IEEE Transactions on VLSI Systems*, 5(1), 1997.

[13] Rainer Leupers. Register allocation for common subexpressions in dsp data paths.

[14] Rainer Leupers and Peter Marwedel. Function inlining under code size constraints for embedded processors. pages 253–256.

[15] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steven Tjiang, and Albert Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18(3):235–253, May 1996.

[16] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steven W.K. Tjiang, and Albert Wang. Code optimization techniques for embedded DSP microprocessors, 1995.

[17] Markus Lorez. Gelir, 2001.
http://ls12-www.cs.uni-dortmund.de/research/gelir.

[18] P. Paulin, C. Liem, T. May, and S. Sutarwala. Dsp design tool requirements for embedded systems: A telecommunications industrial perspective, 1995.

[19] J.D. Ullman R. Sethi. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, October 1970.

[20] V. Zivojnovic, J. Velarde, C. Schlager, and M. Meyr. *DSPstone: a DSP-oriented benchmarking methodology*, 1994.