Diplomarbeit

Speicherpartitionierung in DSP-Compilern

Daniel Kotte

21. April 2000

Betreuer:
Dr. Rainer Leupers
Prof. Dr. Peter Marwedel

Inhaltsverzeichnis

1 Einleitung und Motivation

Wenn in der Öffentlichkeit von Computern und Prozessoren die Rede ist, ist meist der Bereich der Personal Computer (PC) mit seinen x86-kompatiblen Prozessoren Intel Pentium und AMD Athlon oder auch der der Apple-Computer gemeint. Weitgehend unbeachtet bleibt, daß die Mehrheit der weltweit produzierten Prozessoren in sogenannten *Eingebetteten Systemen* arbeitet, die in vielen verschiedenen Anwendungsgebieten wie Mobiltelefonen, Waschmaschinen, Autos oder auch Satelliten zum Einsatz kommen.

Diese Systeme basieren auf unterschiedlichsten Hard- und Softwarekomponenten, die jeweils für das spezielle Einsatzgebiet zusammengestellt und optimiert werden. Im Gegensatz zu den Allzweck-Systemen wie Mainframes, Workstations oder PCs besitzen sie in der Regel keine allgemeinen Benutzerschnittstellen wie Monitor, Tastatur oder Maus, sondern sind in ihre Anwendungsumgebung *eingebettet*, d.h. Art und Umfang ihrer Ein- und Ausgaben richten sich nach dieser Umgebung. In einem Personenwagen nimmt z.B. die Einspritzsteuerung des Motors verschiedene Betriebsdaten des Motors (Temperatur, Drehzahl etc.) auf, verarbeitet sie und sendet als Ausgabe ein entsprechendes Signal an die Einspritzdüsen.

An diesem Beispiel wird deutlich, daß Eingebettete Systeme informationsverarbeitende Systeme sind. Aus der Umgebung werden physikalische Größen aufgenommen, verarbeitet und beeinflußt.

In realen Anwendungen werden noch besondere, durch die Anwendung bestimmte Eigenschaften des Systems gefordert ([Leu00]):

- Vor allem für Steuerungen sind Echtzeit-Bedingungen eine wichtige Voraussetzung, um eine maximale Antwortzeit des Systems vorhersagen zu können.
- Gerade in sicherheitsrelevanten Bereichen wie der Flugzeugelektronik ist eine hohe Zuverlässigkeit der Systeme unerläßlich.
- In mobilen Geräten sind die Größe und der Stromverbrauch des Systems wichtige Eigenschaften, die direkten Einfluß auf deren Effizienz haben.

Um diesen Forderungen entsprechen zu können, werden in Eingebetteten Systemen in der Regel keine der sogenannten *General Purpose Processors (GPP)* eingesetzt. Sie sind zwar in der

Anwendung sehr flexibel und könnten daher für verschiedene Applikationen verwendet werden, ihre niedrige Rechenleistung bei Algorithmen der digitalen Signalverarbeitung verbunden mit einem hohen Stromverbrauch machen ihren Einsatz aber nicht sinnvoll. Auch können Echtzeitbedingungen mit vielen GPPs durch deren superskalare Struktur nicht realisiert werden.

Stattdessen werden Prozessoren bevorzugt, die durch eine spezielle, an die Applikation angepaßte Architektur für diesen begrenzten Anwendungsbereich besonders geeignet sind.

Dies sind auf der einen Seite selbst entwickelte *Application Specific Integrated Circuits (ASICs)*, die eine sehr starke Anpassung an eine Anwendung ermöglichen und daher auch sehr effizient arbeiten. Ihre wesentlichen Nachteile sind die niedrige Flexibilität und die hohen Entwicklungs-, Fertigungs- und Testkosten. Zudem ist die Entwicklungszeit des Prozessors zusammen mit der benötigten Software länger als bei der Verwendung einer Standard-CPU, für die oft schon C-Compiler verfügbar sind. Dies kann bei den immer kürzer werdenden Marktzyklen schon ein Ausschlußgrund für den Einsatz eines ASICs sein.

Auf der anderen Seite stehen die Prozessoren, die im Gegensatz zu ASICs, die für nur eine Anwendung entwickelt werden, und GPPs, die möglichst viele Anwendungen unterstützen sollen, gerade den für Eingebettete Systeme wichtigen Bereich der Signalverarbeitung gut beherrschen. Dies sind die *Digitalen Signalprozessoren (DSPs)*. Sie besitzen bestimmte Architektureigenschaften wie Multiply-Accumulate-Befehle (MAC), heterogene Registersets, Saturierungsarithmetik und besondere Adressierungsarten z.B. zur Unterstützung von Ringbuffern, wodurch sie sich gut für die Realisierung von z.B. digitalen Filtern oder Fouriertransformationen eignen. Echtzeitbedingungen werden von ihnen in der Regel auch unterstützt.

Ein Nachteil der auf die digitale Signalverarbeitung spezialisierten DSP-Architekturen ist, daß die effiziente Programmierung derzeit fast nur in Assembler möglich ist. Die im Vergleich zur C-Programmierung längeren Softwareentwicklungszeiten und die schlechtere Wartbarkeit des Assemblercodes sind eine gravierende Schwäche der Anwendungsentwicklung für DSPs. Der von den für einige DSPs erhältlichen C-Übersetzer generierte Assemblercode ist nicht in der Lage, die Möglichkeiten auszunutzen, die die spezialisierten Architekturen bieten. Dies liegt zum einen daran, daß die existierenden Compiler im kommerziellen Bereich meist auf der Technologie der traditionellen Übersetzer beruhen, die für die Prozessorklasse der GPPs mit ihren homogenen Registersets geschaffen wurden. Zum anderen unterscheiden sich selbst im Bereich der DSPs die eingesetzten Architekturen und Befehlssätze der verschiedenen Hersteller so stark, daß sich die Investition eines Herstellers in einen Übersetzer, der nur eine Prozessorklasse unterstützt, nicht immer lohnt.

Es besteht daher schon seit einiger Zeit das Bestreben, retargierbare Compiler zu entwickeln. Der Unterschied zu traditionellen Übersetzern besteht darin, daß sie neben dem Quellcode auch noch die Beschreibung der Zielarchitektur als Eingabe erhalten. Aus diesem Modell der Zielarchitektur können sie dann den Befehlssatz des Prozessors ableiten und mit diesem den gewünschten Maschinencode erzeugen (Abbildung 1.1 auf der nächsten Seite).

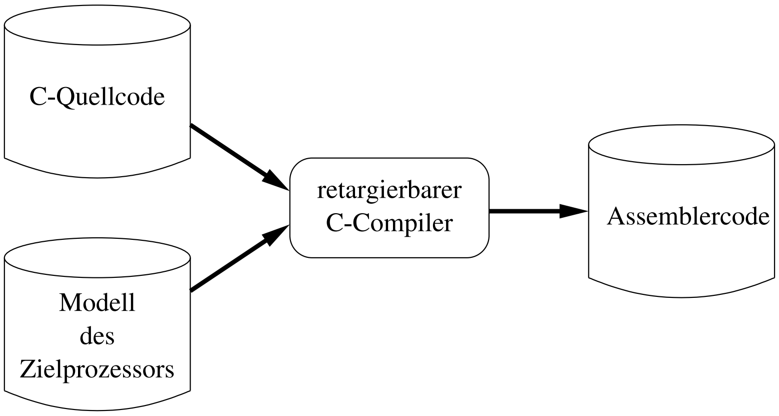


Abbildung 1.1: Ein- und Ausgaben eines retargierbaren C-Compilers

Ein großes Anwendungsfeld der retargierbaren Compiler ist bei der Entwicklung von neuen Prozessoren. Während der Konzeption kann so das Architekturmodell des Prototypen ständig erweitert und verfeinert werden, ohne am Übersetzer selbst Quellcodeänderungen vornehmen zu müssen. Alle Änderungen am Prozessormodell fließen direkt als Eingabe in den Compiler mit ein. Die direkten Vorteile sind eine kürzere Entwicklungszeit neuer Prozessoren und damit ein möglicherweise entscheidender Marktvorteil.

Allerdings können auch retargierbare Compiler noch nicht alle Möglichkeiten der Zielarchitekturen ausschöpfen. So stellen viele DSPs zur Erhöhung der Speicherbandbreite mehrere parallele Speicherbänke zur Verfügung. Die Befehlssätze dieser Prozessoren unterstützen dann das gleichzeitige Laden oder Speichern mehrerer Datenwörter in einem Taktzyklus und schaffen damit die Voraussetzung für die effiziente Implementierung typischer DSP-Algorithmen wie z.B. eines FIR-Filters [Lee88].

Meistens wird in den heute verfügbaren Übersetzern auf eine Strategie zur Aufteilung der Daten eines Programms verzichtet. So kann z.B. der von Motorola für den DSP56300 entwickelte ANSI-C-Compiler alle Daten nur in einer der vorhandenen Speicherbänke allokieren [Mot]. Es kann zwar zur Compilezeit eingestellt werden, in welcher Bank allokiert werden soll, ein Aufteilen der Daten auf die Bänke ist aber nicht vorgesehen.

Motiviert durch diese Defizite des Umgangs bestehender DSP-Compiler mit mehreren Speicherbänken soll in dieser Arbeit die Möglichkeit einer Partitionierung der Programmdaten durch den Compiler untersucht werden.

Folgende Ziele sollen dabei möglichst verwirklicht werden:

- Die Aufteilung der Daten eines Programms soll optimal erfolgen. Dies bedeutet, daß durch die Partitionierung die maximale Anzahl von parallelen Speicherzugriffen realisiert und damit die Ausführungszeit der übersetzten Programme minimiert wird.

- Das Verfahren soll unabhängig vom Zielprozessor sein. Dies kommt dem Bestreben nach Retargierbarkeit entgegen und macht es anwendbar für verschiedene Architekturen.

Es ist schwierig, beide Ziele gleichzeitig zu verwirklichen, da sich die beiden Punkte offensichtlich widersprechen. Denn je weniger Annahmen bezüglich der Zielarchitektur getroffen werden, desto weniger kann auch auf ihre speziellen Besonderheiten des Speicherinterfaces eingegangen werden, so daß eine Optimalität des generierten Codes nicht mehr garantiert werden kann.

In dieser Arbeit wird nun ein Verfahren entwickelt, das versucht, beiden Zielen gerecht zu werden. Dazu werden in Kapitel 2 zunächst bestehende Ansätze zur Speicherpartitionierung vorgestellt. Trotz des Ziels der Architekturunabhängigkeit mußte das entwickelte Verfahren zur Evaluierung natürlich für einen bestimmten Prozessor implementiert werden. Auf diesen und die weitere Entwicklungsumgebung wird in Kapitel 3 eingegangen. In Kapitel 4 wird dann ausführlich auf die realisierte Speicherpartitionierung eingegangen. Die erreichten Resultate für die hier gewählte Zielarchitektur werden in Kapitel 5 vorgestellt und Kapitel 6 faßt die vorliegende Arbeit noch einmal kurz zusammen. Dabei wird auf einige Fragen eingegangen, die sich bei der Anwendung des Verfahrens ergeben haben, und ein Ausblick auf mögliche Verbesserungen und Erweiterungen gegeben.

2 Andere Arbeiten

Üblicherweise wird die Verantwortung für die Nutzung mehrerer Speicherbänke dem Programmierer überlassen. Dies kann direkt bei der Programmierung in Assembler oder auch durch die Verwendung von Spezialbefehlen (z.B. Pragmas) in höheren Programmiersprachen geschehen. Kommerzielle Compiler unterstützen bisher keine automatische Speicherbankzuweisung. In der Forschung allerdings existieren verschiedene Ansätze, eine geeignete Partitionierung der Variablen eines Programmes zu finden, um die verfügbaren Speicherbänke des Zielprozessors auszunutzen. Diese Ansätze sollen im folgenden vorgestellt werden.

2.1 Powell et. al.

Powell et. al. generieren optimierten Assemblercode aus Signalflußblockdiagrammen. Die einzelnen Blöcke dieser Diagramme repräsentieren abstrakte Funktionen, die in einer Meta-Assembler-Sprache implementiert sind.

Deren Sprachumfang soll sich am Befehlssatz des Zielprozessors (hier: Motorola 56000 [Mot86]) orientieren. Statt fester Operanden enthält diese Sprache Registerklassen, in denen die jeweils in diesem Befehl möglichen Register zusammengefaßt werden. So kann z.B. die Registerklasse XREG die beiden Register X0 und X1 enthalten.

Aus den Befehlen der Meta-Assembler-Sprache werden Blöcke zusammengesetzt und in einer Bibliothek gespeichert. Jeder Block realisiert eine bestimmte Funktion und enthält zusätzliche Angaben zu Art (Input/Output/...), Zugriff (Read/Write) und Registerklasse der benötigten Parameter.

Um eine Vorstellung von der Abstraktionsebene dieser Funktionen zu bekommen, wird das Beispiel vorgestellt, das Powell et. al. in [PLN92] anführen.

Die zu implementierende Funktion führt eine Suche in einer Tabelle mittels linearer Interpolation durch. Am Beginn des Blocks stehen die Deklarationen aller benötigten Variablen (Tabelle 2.1 auf der nächsten Seite).

Im Anschluß daran muß der eigentliche Meta-Assembler-Code angegeben werden (Abbildung 2.1 auf der nächsten Seite).

	Name	Zugriff	Registerklasse
Input:	in	Lesen	XREG
Output:	out	Lesen/Schreiben	ACC
Vector State:	table	Lesen/Schreiben(reg)	YADDREG
Parameter:	lenMinusOne	Lesen/Schreiben	ACC
Temporär:	tA	Lesen/Schreiben	ACC
Temporär:	tX	Lesen/Schreiben	XREG
Temporär:	tY	Lesen/Schreiben	YREG

Tabelle 2.1: Deklarationen am Anfang eines Blockes

```
MOVE lenMinusOne,tX
MAC  in,tX,out      #<\$80,tA
ASR  out            table.reg,tY
ADD  tY,out         out.low,tA.mid
ASR  tA             out,table.reg
MOVE #<\$40,out
ADDL tA,out         tA,tX           Y:(table.reg)+,tY
MPY  -tX,tY,out     out,tX          Y:(table.reg),tY
MACR tX,tY,out
```

Abbildung 2.1: Meta-Assembler-Code

Desweiteren enthält die Meta-Assembler-Sprache noch Instruktionen zur Angabe von Aliasen und bedingter Codegenerierung. Aliase in diesem Beispiel könnten `lenMinusOne` und `out` sein. Sie würden dann beide demselben Register zugewiesen werden. Auch ist es möglich, dem Codegenerator die Speicherbankauswahl zu überlassen.

Der Codegenerator führt nun der Reihe nach verschiedene Schritte durch, die hier vereinfacht dargestellt werden sollen.

1. Erkennen von Blöcken, die idealerweise zu einem Block zusammengefaßt werden können (Addierer + Multiplizierer → Multiply-Accumulate)
2. Auswertung der Bedingungen jedes Blockes um festzustellen, welche Teile des Meta-Assemblers zur Codegenerierung verwendet werden sollen
3. Scheduling der Blöcke. Dies geschieht mit Hilfe mehrerer Heuristiken.
4. Register werden den Symbolen der Blöcke zugewiesen. Auch hierfür verwenden die Autoren verschiedene Heuristiken.
5. Bei der Speicherallokation werden statischen Variablen, Vektoren und Spilldaten Speicheradressen zugewiesen. Die Zuweisung zu den verschiedenen Speicherbänken X und Y

erfolgt abwechselnd.

6. Einfügen von Code zum Datentransfer zwischen Registern und zwischen Registern und Speicher.
7. Kompaktierung des Assemblercodes

Powell et.al. erreichen mit ihrem Ansatz der Codeerzeugung zum Teil sehr kompakten Assemblercode. Es darf dabei allerdings nicht vergessen werden, daß die einzelnen Blöcke des Signalflußblockdiagramms von Hand in Meta-Assembler programmiert werden müssen. Da diese Sprache auch dem Assembler des Zielprozessors ähnlich sein muß und daher für jeden Prozessor anders ist, sind die Kosten der Retargierbarkeit des Ansatzes sehr groß. Der für diese Diplomarbeit interessante Aspekt der Verteilung von Daten auf die beiden Speicherbänke X und Y läßt eine spezielle Strategie zur Partitionierung vermissen. Die Technik der abwechselnden Zuweisung von Variablen zu den Speicherbänken muß man als Greedy-Strategie bezeichnen.

2.2 Sudarsanam/Malik

Im Gegensatz zur Arbeit, die im vorherigen Abschnitt vorgestellt wurde, befassen sich Sudarsanam und Malik ganz speziell mit dem Problem der Speicherbank- und Registerallokation in Prozessoren mit zwei Speicherbänken [SM95]. Als Zielprozessor verwenden sie ebenfalls den Motorola DSP56000 (s. Abbildung 2.2 auf der nächsten Seite).

Die Autoren fassen Register- und Speicherbankallokation zu einem Schritt zusammen mit der Begründung, daß DSPs mit zwei Speicherbänken wie u.a. auch der Motorola DSP56000 die Eigenschaft besitzen, Daten einer Speicherbank (z.B. Bank *X*) nur in bestimmte Register (Register *X0* und *X1*, aber nicht *Y0* und *Y1*) transferieren zu können. Daraus folgern sie eine Abhängigkeit zwischen den beiden Zuweisungsproblemen, die eine kombinierte Betrachtung der Einzelprobleme erforderlich macht.

Ihr Verfahren benötigt als Eingabe optimierten sequentiellen Code mit symbolischen Register- und Variablenreferenzen. Der darauf aufbauende dreistufige Algorithmus nimmt dann nur noch die Zuweisung zu den physikalischen Registern und Speicherbänken vor. Allerdings werden globale Variablen und alle Funktionsparameter fest in einer Speicherbank allokiert. Eine interprozedurale Datenflußanalyse, die die Benutzung der globalen Variablen und der Funktionsparameter, die als Zeiger übergeben werden, über alle Funktionen verfolgt, ist nicht vorhanden. Außerdem werden die verschiedenen Elemente von Feldern und Strukturen jeweils als einzelne Variablen betrachtet.

Im ersten Schritt des Algorithmus wird der sequentielle Code kompaktiert. Dabei wird versucht, zwei Anweisungen zu einer zusammenzufassen, wenn keine Datenabhängigkeiten verletzt wer-

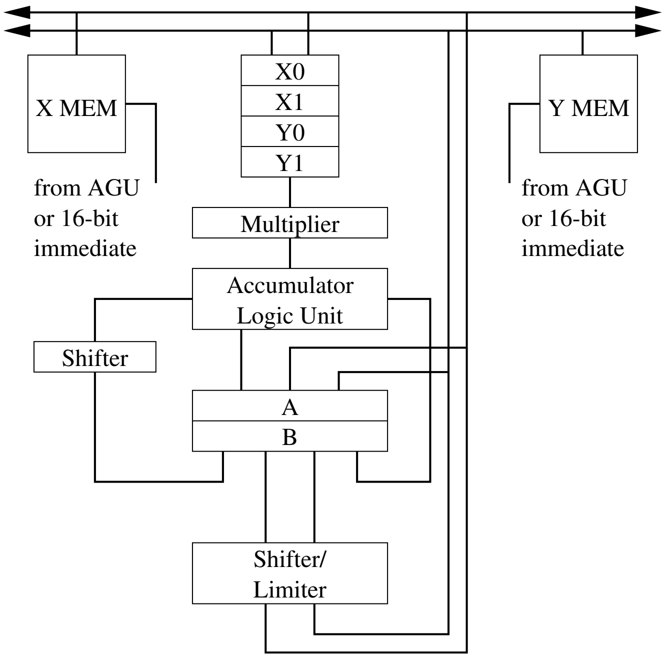


Abbildung 2.2: Vereinfachte Darstellung der ALU und der Speicherbänke des Motorola DSP56000 (vgl. [Mot86])

den und die entstandene Parallel-Move-Instruktion von der Architektur unterstützt wird. Unterschieden werden beim DSP56000 drei Typen von Moves: Laden/Speichern eines Registers aus dem/in den Speicher (Speicher-Register-Move), Transfer eines Registerinhalts in ein anderes Register (Register-Register-Move), Laden einer Konstanten in ein Register (Konstanten-Move). Die Architektur unterstützt die parallele Ausführung in einer Instruktion von einem

- Speicher-Register-Move mit einem anderen Speicher-Register-Move oder einem Register-Register-Move,
- Register-Register-Move zusammen mit einem Speicher-Register-Move oder einem Konstanten-Move,
- Konstanten-Move mit einem Register-Register-Move.

Für die Gültigkeit der verschiedenen Kombinationen existieren noch weitere spezielle Einschränkungen, die hier nur am Beispiel der Kombination von zwei Speicher-Register-Moves exemplarisch angedeutet werden sollen.

Die parallele Ausführung von zwei Speicher-Register-Moves ist beim Motorola DSP56000 nur möglich, wenn folgende Bedingungen eingehalten werden:

1. Die beiden Moves adressieren unterschiedliche Speicherbänke.
2. Ein Move vom bzw. zum X-Speicher verwendet als Quell- bzw. Zielregister entweder X0, X1, A oder B.
3. Ein Move vom bzw. zum Y-Speicher verwendet als Quell- bzw. Zielregister entweder Y0, Y1, A oder B.
4. Beide Moves verwenden Register-Indirekte Adressierung und die dabei benutzten Adreßregister gehören zu unterschiedlichen AGUs (Der Motorola DSP56000 verfügt über zwei Adreßgenerierungseinheiten (AGU), die jeweils über vier Adreßregister verfügen. Die erste AGU besteht aus den Registern R0, R1, R2 und R3, die zweite aus R4, R5, R6 und R7.)

Ein Beispiel einer gültigen Instruktion ist demnach

```
MOVE X:(R0),X1 Y:(R4),Y1.
```

Im zweiten Schritt wird ein Graph erstellt, der die Register- und Speicherverweise als Knoten und die Nebenbedingungen bzw. Einschränkungen der Prozessorarchitektur als Kanten enthält.

Den Kanten werden Kosten zugewiesen, die entstehen, wenn die entsprechende Nebenbedingung verletzt wird. Die Aufgabe besteht nun darin, allen Knoten eine Beschriftung, die den jeweiligen Allokationsort der symbolischen Referenz angibt, zuzuweisen, so daß die Kosten der verletzten Nebenbedingungen minimal werden. Dabei kann den Registerknoten (*reg*) als Beschriftung *X0*, *X1*, *Y0*, *Y1*, *A* oder *B* entsprechend den Registern des Motorola DSP56000 und den Knoten für die Speicherverweise (*var*) eine Beschriftung *X-MEM* bzw. *Y-MEM* für die Speicherbänke zugewiesen werden.

Die Kantenbedingungen lassen sich wie folgt unterteilen:

Rote (Interferenz-)Kanten verbinden zwei Registerknoten reg_a und reg_b miteinander, wenn a und b gleichzeitig lebendig sind. Daraus folgt, daß durch rote Kanten verbundene symbolische Registerreferenzen verschiedene physikalische Register zugewiesen werden müssen. Wenn dies nicht möglich ist, wird Spillcode zur Zwischenspeicherung eines Wertes im Speicher erforderlich. Jede rote Kante wird mit einem konstanten Kostenwert von 10 versehen, der die Kosten für den zusätzlichen Code darstellen soll.

Die roten Kanten werden zuerst eingefügt. Anschließend werden dem Graphen für alle kompaktierten Instruktionen, die zwei Move-Befehle enthalten, bestimmte nicht-rote Kanten hinzugefügt.

Grüne Kanten stehen für einen parallelen Speicherzugriff. Sie verbinden zwei Registerknoten reg_a und reg_b miteinander, wenn reg_a an der ersten Move-Operation beteiligt ist und reg_b an der zweiten. Der andere Operand muß jeweils eine Speicherreferenz (var_a/ var_b) sein. Grüne Kanten enthalten auch Zeiger auf diese Speicherreferenzen. Eine korrekte Beschriftung der beteiligten Knoten muß nun mehrere Anforderungen erfüllen:

1. Die Knoten var_a und var_b müssen unterschiedliche Beschriftungen erhalten.
2. Falls Knoten var_a die Beschriftung *X-MEM* (*Y-MEM*) erhält, muß der Knoten reg_a mit *X0*, *X1*, *A* oder *B* (*Y0*, *Y1*, *A* oder *B*) beschriftet sein.
3. Falls Knoten var_b die Beschriftung *X-MEM* (*Y-MEM*) erhält, muß der Knoten reg_b mit *X0*, *X1*, *A* oder *B* (*Y0*, *Y1*, *A* oder *B*) beschriftet sein.

Wenn die Bedingungen nicht eingehalten werden können, wird für eine grüne Kante ein Kostenwert von 1 angenommen, da aus einer kompaktierten Instruktion durch das Aufspalten effektiv eine neue Instruktion hinzukommt. Falls die kompaktierte Instruktion mehrfach aufgerufen wird (kann durch Profiling festgestellt werden), ist der Kostenwert gleich der Anzahl der Aufrufe.

Ähnlich den grünen Kanten werden noch blaue, braune und gelbe Kanten eingeführt.

Blaue Kanten stehen für einen Memory-Load parallel zu einem Registertransfer.

Braune Kanten stehen für einen Immediate-Load parallel zu einem Registertransfer.

Gelbe Kanten stehen für einen Memory-Store parallel zu einem Registertransfer.

Schwarze Kanten dienen der Überprüfung der Korrektheit von ALU Anweisungen. Sie stellen sicher, daß die Quellregister und das Zielregister den Restriktionen des jeweiligen Befehls entsprechen.

Die Restriktionen des Befehlssatzes sind über den so entstandenen Graphen fast vollständig beschrieben. Es fehlt an dieser Stelle noch die Einschränkung des DSP56000, daß eine Anweisung, die zwei Speicherzugriffe enthält, unbedingt die register-indirekte Adressierung verwenden muß. Die absolute Adressierung steht hier also nicht zur Verfügung. Da auch dort, wo absolute Adressierung erlaubt ist, dies einen zusätzlichen Codebedarf von einem Wort bedeutet, diese Adressierung also in Bezug auf die Codegröße teuerer ist als indirekte Adressierung, verzichten die Autoren insgesamt darauf. Die implementierte Adreßregisterallokation geht generell von indirekter Adressierung aus und verwendet General-Offset-Assignment (GOA) [LDK$^+$95].

Im dritten Schritt des Verfahrens wird eine Beschriftung des Graphen gesucht, die minimale Gesamtkosten verursacht. Die Gesamtkosten entsprechen der Summe der Kosten der verletzten Kantenbedingungen und der Kosten des GOA. Eine Minimierung der Kosten entspricht der Minimierung der Abweichung des resultierenden Codes vom kompaktierten Code. Da der Suchraum für das Problem einer kostenminimalen Lösung sehr groß ist, bzw. es sich um ein NP-vollständiges Problem handelt, wenden die Autoren Simulated-Annealing an.

Um die Leistungsfähigkeit ihrer Strategie überprüfen zu können, implementierten die beiden Autoren zusätzlich zum oben beschriebenen Verfahren eine Greedy-Strategie, die ähnlich dem Verfahren von Powell et.al. eine abwechselnde Zuweisung der Variablen auf die beiden Speicherbänke vornimmt. Ein Vergleich der beiden Strategien wurde mit sieben Programmen aus dem DSPstone-Benchmark [Ziv94] und zwei weiteren DSP-Algorithmen von Motorola durchgeführt. Dabei wurde der initiale Assemblercode für die DSPstone-Programme mit dem 56000-Compiler von Motorola erstellt. Für die beiden weiteren Programme existierte handgeschriebener Code.

In jedem Anwendungsfall konnte die Codegröße verringert werden. Die erreichten Verbesserungen bewegten sich im Bereich von 20% bis 57%. Bei diesen starken Verbesserungen zur Anfangscodegröße ist allerdings zu beachten, daß der 56000-Compiler sehr schlechten Code erzeugt. So werden von ihm alle Variablen im Y-Speicher allokiert und alle Speicherzugriffe verwenden absolute Adressierung. Eine starke Verbesserung gegenüber diesem Anfangscode ist also nicht verwunderlich. Auch der handgeschriebene Code macht sehr oft Gebrauch von der teueren absoluten Adressierung.

Durch die unterschiedliche Komplexität der Benchmarks ergaben sich auch stark schwankende Laufzeiten für das Simulated-Annealing. Die Spanne reichte von 1.4 Sekunden bis zu 2.8

Stunden. Dagegen verwundert das gute Abschneiden der Greedy-Strategie, die in sechs der zehn Anwendungen zur gleichen Codegröße führte wie die Strategie von Malik/Sudarsanam. Auch in den anderen vier Fällen war die Verbesserung zum Anfangscode nur um maximal 6% geringer.

Auf den ersten Blick scheint sich also nach diesen Ergebnissen eine aufwendige Strategie zur Speicherbankallokation nicht zu lohnen, da die Greedy-Strategie bei weniger Rechenzeit fast ebenso gute Resultate liefert. Ein wichtiger Punkt, der für die aufwendige Strategie der Autoren spricht, ist aber die bessere Retargierbarkeit ihrer Lösung, da nur die Restriktionskanten des Graphen für einen anderen Prozessor angepaßt werden müssen. In wie weit eine Greedy-Strategie bei einem anderen Zielprozessor zu ähnlich guten Ergebnissen führt ist ungewiß. Weitere Verbesserungen sind sicherlich dann möglich, wenn es gelingt, die Einschränkungen bei der Partitionierung von globalen Variablen, Funktionsparametern und Arrays zu umgehen.

2.3 Saghir et. al.

Ein dritter Ansatz zur Nutzung von zwei Speicherbänken in DSPs kommt von Saghir et. al. [SCL96]. Als Teil der Speicherallokationsphase ihres optimierenden Compilers stellen sie zwei sich ergänzende Verfahren vor. Beide Verfahren wurden für eine VLIW(Very Long Instruction Word)-Modell-Architektur entwickelt. Der Aufbau dieses Zielprozessors ist in Abbildung 2.3 auf der nächsten Seite zu sehen. Von besonderem Interesse sind im Zusammenhang mit den entwickelten Verfahren die beiden Speicherzugriffseinheiten MU0 und MU1 sowie die beiden Speicherbänke X und Y. Jede Speicherbank kann dabei nur über die ihr zugeordnete Speicherzugriffseinheit adressiert werden und besitzt genau einen Port. Alle funktionalen Einheiten haben eine Latenzzeit von einem Takt.

Der verwendete Compiler besteht aus einem GNU-C Compiler Frontend und einem optimierenden Backend, das vom Frontend die ungepackten Maschinenoperationen erhält und darauf architekturspezifische Optimierungen ausführt. Abschließend generiert das Backend VLIW-Befehle.

Die beiden hier vorgestellten Verfahren werden während der Datenallokation im Backend angewendet. In dieser Phase wird versucht, die vorhandenen Variablen den beiden Speicherbänken zuzuweisen, so daß möglichst viele Load- und Store-Befehle parallel ausgeführt werden können. In der späteren Kompaktierungsphase wird diese Parallelität dann ausgenutzt, um die einzelnen Befehle in möglichst wenige VLIW-Befehle unterzubringen.

Das erste Verfahren, *Compaction-Based-Data-Partitioning*, nimmt die Zuweisung der Variablen zu den Speicherbänken vor. Dabei verwendet es einen Interferenzgraphen, dessen Knoten die verschiedenen Variablen und dessen Kanten den möglichen parallelen Zugriff auf ein Variablenpaar repräsentieren. Idealerweise sollten die beiden durch eine Kante verbundenen Variablen in verschiedenen Speicherbänken allokiert werden. Der Interferenzgraph wird für jeden Basisblock

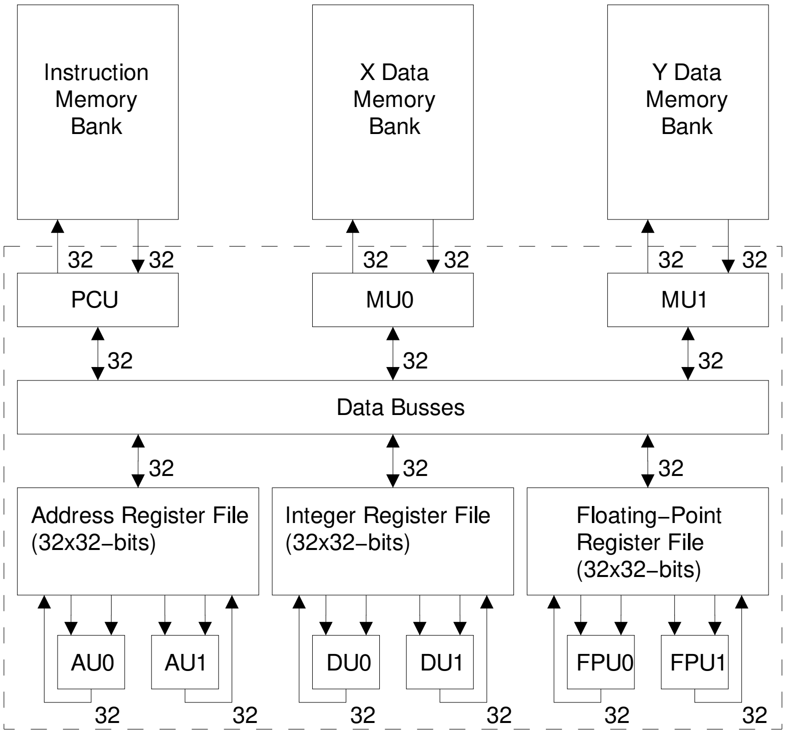


Abbildung 2.3: VLIW-Modell-Architektur

```
D[i] = A[j] + B[k]
B[i] = B[j] - D[k]
C[i] = B[j] - C[k]
C[i] = A[j] + C[k]
for (i=0; i<5; i++) {
   ...
   C[i] = A[i] + D[i]
   ...
}
A[i] = C[j] + D[k]
```

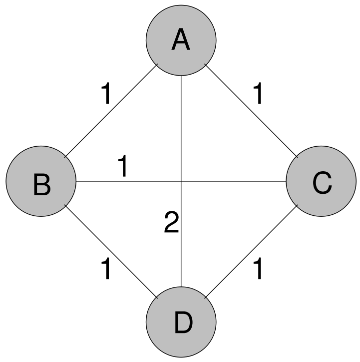


Abbildung 2.4: Beispielprogramm mit zugehörigem Interferenzgraph

erstellt.

Ausgehend vom Datenabhängigkeitsgraphen des entsprechenden Basisblocks wird ein List-Scheduling-Algorithmus angewandt, der vom Ablauf her dem der späteren Kompaktierung entspricht. Jeder Schritt, vom Erstellen des Datenabhängigkeitsgraphen, über die Zuweisung von Prioritäten zu den einzelnen Operationen und dem Generieren eines Data-Ready-Set auf Basis der Prioritäten, bis zum Füllen der VLIW-Instruktionen, ist identisch mit der Ausnahme, daß noch kein kompaktierter Code ausgegeben wird. Statt dessen wird immer dann eine Kante zwischen zwei Knoten des Interferenzgraphen eingefügt, wenn zwei Load-/Store-Operationen auf die von den Knoten repräsentierten Variablen nur dann in eine VLIW-Anweisung kompaktiert werden könnten, falls die beteiligten Variablen in unterschiedlichen Speicherbänken allokiert werden. Das Einfügen einer Kante geschieht allerdings nur dann, wenn die beteiligten Variablen nicht gleich sind bzw. das gleiche Feld referenzieren. In dem Fall, also wenn der Load-/Store-Zugriff jeweils auf die gleiche Variable oder das gleiche Feld stattfindet, wird diese Variable (oder Feld) markiert. Die Markierung ist für das später beschriebene zweite Verfahren notwendig.

Die Kanten des Interferenzgraphen sind mit Kosten gewichtet, die den Effizienzverlust bei einem nicht realisierten parallelen Zugriff darstellen. Die Autoren wählten als Kostenmaß die Schleifen-Schachtelungstiefe des Befehls, der den Speicherzugriff auslöst. Bezweckt wird damit das vorrangige Ausnutzen von Parallelität in inneren Schleifen. Auch andere, aufwendigere Kostenmaße wurden ausprobiert, doch führten diese nach ihrer Aussage selten zu besseren und oft sogar zu schlechteren Ergebnissen.

Die Knoten des Interferenzgraphen werden nun in zwei Mengen partitioniert, deren Variablen dann in unterschiedlichen Speicherbänken allokiert werden sollen. Gesucht wird eine Partitionierung mit minimalen Kosten. Die Kosten berechnen sich aus Addition der Summe der Kosten der Kanten, die komplett in der ersten Menge liegen, und der entsprechenden Kantenkostensumme der zweiten Menge. Saghir et.al. verwenden zur Lösung dieses NP-vollständigen Problems eine Greedy-Strategie.

Dieses erste Verfahren erreichte bei der Mehrzahl der von den Autoren getesteten Benchmarks optimale Ergebnisse, d.h., es erreichte den maximal möglichen Performanzgewinn in Bezug auf die Ausführungsgeschwindigkeit des Assemblercodes. Verglichen wurde dabei mit den Resultaten, die auf einem Prozessormodell mit Dual-Port-Speicher möglich sind. Entsprachen sich die Ergebnisse, konnte man von Optimalität ausgehen. Bei der genaueren Betrachtung der anderen, nicht optimalen Benchmarks fiel auf, daß ein Effizienzgewinn durch das Data-Partitioning allein nicht möglich war, da dazu ein simultaner Zugriff auf mehrere Elemente desselben Arrays erfolgen mußte. Arrays werden aber von Saghir et.al. als Einheiten betrachtet und daher jeweils nur komplett einer Speicherbank zugewiesen.

Als Ausweg wird das zweite Verfahren, *Partial-Data-Duplication*, vorgeschlagen, das diese kritischen Arrays in die jeweils andere Speicherbank kopiert und so immer den simultanen Zugriff auf ihre Elemente ermöglicht. Die kritischen Felder sind gerade diejenigen, die beim ersten Verfahren markiert wurden. Die Duplikation von Feldern erfordert natürlich zusätzlichen Code, um die Konsistenz der dann doppelt vorhanden Werte z.B. auch bei Interrupts zu sichern. Dies führt im allgemeinen zu Laufzeitverschlechterung und höherem Speicherbedarf. Allerdings zeigte sich bei den Benchmarks, daß der nun mögliche parallele Zugriff diese negativen Begleiterscheinungen meist ausgleicht und sogar zu einem Performanzgewinn führen kann. Anzumerken ist aber, daß nur ein Application-Benchmark massiv von diesem zweiten Verfahren profitierte.

Der Performanzgewinn des Compaction-Based-Data-Partitioning schwankt bei den DSP-Kernel-Benchmarks zwischen 13% und 49% und beträgt im Durchschnitt 29%. Diese sehr hohen Werte resultieren daher, daß diese Benchmarks vornehmlich aus wenigen Schleifenoperationen bestehen, die massiv von parallelen Zugriffen auf die in den Schleifen verwendeten Variablen profitieren. Nur bei einem der zwölf Kernel wurde nicht der gleiche Geschwindigkeitszuwachs erzielt wie im idealen Fall (Prozessormodell mit Dual-Port-Speicher).

Der Gewinn bei den DSP-Anwendungs-Benchmarks ist dagegen geringer. Vier Benchmarks konnten erst gar nicht von parallelen Speicherzugriffen profitieren, was dadurch ersichtlich wurde, daß selbst mit Dual-Port-Speicher keine kürzere Programmlaufzeit erreicht werden konnte. Ein einfacher Grund dafür ist, daß Anwendungen im Gegensatz zu KernelBenchmarks in der Regel mehr Kontrollanweisungen als spezielle Schleifenoperationen enthalten. Dadurch ist eine so massive Parallelisierung nicht mehr möglich.

Bei drei der restlichen sieben Testfälle konnte mit Compaction-Based-Data-Partitioning der Idealgewinn von 3% bis 13% erreicht werden, bei den anderen vier jedoch im Durchschnitt nur etwas mehr als die Hälfte des Gewinns im Idealfall. Bei diesen Anwendungs-Benchmarks wurde versucht, mit Partial-Data-Duplication das Ergebnis zu verbessern. Der Erfolg dieser

Maßnahme ist uneinheitlich. In einem Fall, wo der Unterschied zwischen idealem Code und dem nach Compaction-Based-Data-Partitioning sehr groß war, konnte das Ergebnis massiv verbessert werden. Bei den anderen Fällen gab es einmal eine nur leichte Verbesserung und einmal sogar eine Verschlechterung der Codequalität. Dies ist mit dem Implementierungsoverhead zu erklären, der durch die Sicherung der Konsistenz der duplizierten Arrays entsteht.

Als Fazit läßt sich zusammenfassen, daß sich die Anwendung des Compaction-Based-Data-Partitioning im allgemeinen Fall lohnt und zu schnellerem Assemblercode führt. Da sich auch der Speicherbedarf durch die Partitionierung verringert, ist der Nutzen dieser Optimierung noch größer.

Die partielle Duplikation von Arrays sollte dagegen als Spezialoptimierung angesehen werden, da das Verhältnis von Kosten zu Nutzen keine eindeutige Empfehlung zuläßt. Erst wenn es gelingt, eine Aufwands- und Erfolgsabschätzung dieser Optimierung während des Übersetzungsprozesses vom Compiler vornehmen zu lassen, es also zu einem bestimmten Zeitpunkt der Übersetzung möglich ist, die Qualität des zu erzeugenden Codes vorhersagen zu können, kann die partielle Duplikation von Feldern empfohlen werden. Dafür ist es dann notwendig, dem Compiler weitere Informationen zur Entscheidung für und gegen den Einsatz dieser Optimierung zu geben. Als Beispiele seien hier der maximal verfügbare Speicher, da sich durch die Duplikation der Speicherbedarf erhöht, oder auch eine ausreichende Ausführungszeit des Codes genannt.

Ein entscheidender Nachteil des Versuchs von Saghir et.al. ist die Entwicklung der Optimierungen auf einem allgemeinen und homogenen VLIW-Testsystem, so daß keine exakten Rückschlüsse auf die Performanz der beiden Verfahren bei realen DSP-Prozessoren mit normalerweise unregelmäßigen Architekturen möglich sind. Eine Realisierung des Testprozessors als eingebettetes System wäre nach Aussagen der Autoren aus Kostengründen auch nicht sinnvoll. Erst die Anpassung der Verfahren an andere, reale Prozessoren kann daher deren Tauglichkeit zeigen. In diesem Zusammenhang tritt auch die Frage nach der Retargierbarkeit auf. Da Saghir et.al. die Optimierungen im Backend ansiedeln, ist eine Anpassung an den Zielprozessor auf jeden Fall notwendig. Da sich aber die Beschreibung der beiden Verfahren vorwiegend auf Hochsprachenelemente wie Felder und Variablen stützt und die Architektur des Testprozessors sehr allgemein und homogen gehalten ist, sollte eine Implementierung zumindest des Compaction-Based-Data-Partitioning in einer anderen Prozessorumgebung keine Probleme aufwerfen. Beim Partial-Data-Duplication ist die Programmierung des Assemblercodes zur ständigen Sicherstellung der Konsistenz der duplizierten Arrays notwendig. Dieser Aufwand muß daher bei der Bewertung des Verfahrens zusätzlich in Betracht gezogen werden.

3 Die Entwicklungsumgebung

Diese Kapitel soll einen Überblick über die zur Verfügung stehende Entwicklungsumgebung geben.

Als Zielprozessor wurde der Gepard DSP Kern der Austria Mikro Systeme International AG (AMS) ausgewählt. Für ihn sprach auf der einen Seite natürlich die für diese Arbeit notwendige Möglichkeit, Speicherzugriffe auf zwei Speicherbänke parallel in einer Instruktion ausführen zu können. Andererseits befand sich zu Beginn dieser Arbeit ein Compilerbackend für diesen Prozessor an der BTU Cottbus in der Entwicklung. Dieses wurde als Sourcecode zur Verfügung gestellt und konnte daher den jeweiligen Anforderungen gemäß modifiziert werden. Die Gepard Prozessorfamilie wird im folgenden Kapitel 3.1 vorgestellt, das Compilerbackend GEPCC im Kapitel 3.3.

Da das Backend selbst auf dem vom ANSI-C Compilerfrontend LANCE generierten Zwischencode aufbaut und auch dessen Möglichkeiten für den Zugriff auf Zwischencode und Symboltabelle nutzt, wird auch das Frontend kurz in Kapitel 3.2 beschrieben.

Ohne Kompaktierung des Assemblercodes kann die vom Backend evtl. schon ausgenutzte Parallelität auf Intruktionsebene nicht bewertet werden. Auch die Veränderungen der Codequalität als Folge einer Speicherpartitionierung können ohne Kompaktierung nur schwer festgestellt werden. Da das Backend GEPCC selbst keine Kompaktierung des Assemblercodes vornimmt, mußte diese extra implementiert werden. Die Implementierung des Kompaktierers GEPPAC wird in Kapitel 3.4 vorgestellt.

3.1 Der Gepard DSP Kern

3.1.1 Architektur des Prozessors

Bei den Gepard DSP Kernen der Austria Mikro Systeme International AG (AMS) handelt es sich um eine Familie von programmierbaren und parametrisierbaren Festkomma-DSP-Kernen [OFG97][Aus98b]. Durch verschiedene Ausbaustufen kann die Architektur an die gewünschte Funktionalität angepaßt werden. Damit ist es möglich, einen Gepard Kern in unterschiedlichsten Anwendungsgebieten einzusetzen. Die Anpassung geschieht über eine Reihe von Optionen, die in Tabelle 3.1 auf der nächsten Seite aufgelistet sind.

Eigenschaft	Wertebereich	Standard	Erläuterung
Datenwortbreite	8. . . 64 bit, in 8 bit Schritten	16	Datenregister und Busse
Datenadreßbreite	8. . . 23 bit	16	$<=$ Datenwortbreite
Programmadreßbreite	8. . . 19 bit	16	$<=$ Datenwortbreite
Programmwortbreite	32 bit	32	ein Programmwort ist immer 32 bit breit
Eingabewortbreite des Multiplizierers	8. . . 64 bit	Datenwortbreite	Ausgabe $= 2*$ Datenwortbreite $+$ Guard Bits
Guard Bits des Multipliziererausgangs	0. . . 16	8	$<=$ Datenwortbreite -2
Anzahl der Indexregister	8, 16	8	
Anzahl der Akkumulatoren	2. . . 4	4	
C/D Register verfügbar	0, 1	0	C- und D-Register können als ALU Operand, Sprungbedingung und Schleifenzähler verwendet werden
Nur Modifier	0, 1	0	Indexpaarregister können nur als Modifyregister verwendet werden
Schleifenhardware	0. . . 8	0	Stufen der Schleifenhardware
Adressierungsarten	0. . . 6	0	0=$\pm m$, 1=Modulo-Adressierung, 2=Bit-Reversed, . . .

Tabelle 3.1: Verfügbare Optionen der Gepard-Architektur

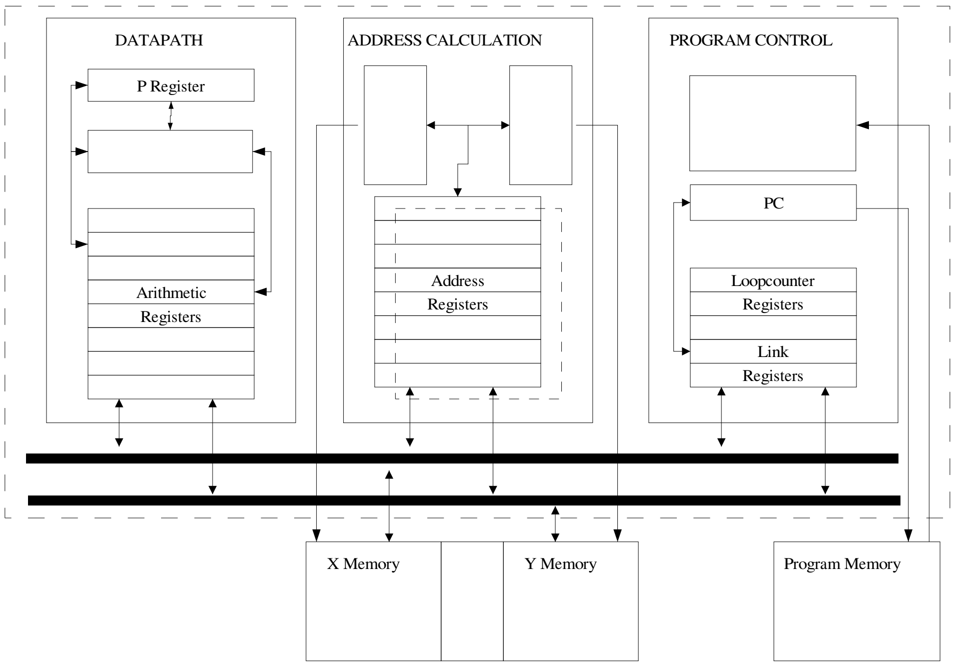


Abbildung 3.1: Architektur des Gepard

Die Architektur des Gepard (Abbildung 3.1) kann als RISC-Architektur bezeichnet werden, da der Befehlssatz nur wenige und dabei einfache Instruktionen enthält (s.a. Abschnitt 3.1.2 auf der nächsten Seite). Jede Instruktion wird in einem Takt ausgeführt. Bedingt durch die für den Programmierer transparente dreistufige Pipeline mit den Phasen Fetch – Decode – Execute besitzen alle Sprungbefehle einen Branch-Delay-Slot.

Als funktionale Einheiten lassen sich im Prozessorkern die Steuereinheit (CNTR), die Adreßgenerierungseinheit (AGU) und der Datenpfad unterscheiden.

Die Steuereinheit beherbergt den Befehlszähler (PC), die Linkregister (LR0 und LR1) zum Speichern der Rücksprungadressen bei Interrupts oder Unterprogrammen und soweit vorhanden die Schleifenregister (LC - Schleifenzähler, LS - Schleifenanfang, LE - Schleifenende). Der PC ist nur über die Linkregister zugänglich. Ein spezielles Statusregister ist nicht vorhanden. Die einzig möglichen Sprungbedingungen sind die Vorzeichenbits der Akkumulatoren und der Adreßregister vorgesehen. Die gesamte Ablaufkontrolle des ausgeführten Programms erfolgt in dieser Einheit.

Die AGU enthält die Indexregister (I0. . . I7/I15), über die jeder Zugriff auf den Speicher erfolgen muß. Unterstützt werden zwei Speicherbänke, X und Y, auf die parallel in einem Takt über zwei verschiedene Indexregister zugegriffen werden kann. Verschiedene Arten der Postfixmodifikation sind je nach Ausbaustufe des Kerns möglich (Tabelle 3.1 auf Seite 22).

Der Datenpfad des Gepard ist in Abbildung 3.2 auf der nächsten Seite detaillierter dargestellt. Er beinhaltet den Multiplizierer, die Arithmetisch-Logische-Einheit und mehrere Register. Der Multiplizierer, der auch den Multiply-Accumulate-Befehl ausführt, speichert sein Ergebnis im P-Register mit mehr als der doppelten Wortbreite. Dessen Inhalt kann über einen Shifter und die ALU in die zwei bis vier Akkumulatorregister (A0. . . A3) kopiert werden. Bis auf das P-Register können alle Register mit den Datenspeichern über zwei Datenbusse kommunizieren. Parallel zum Multiplizieren kann die ALU einen weiteren arithmetischen Befehl ausführen.

3.1.2 Der Instruktionssatz

Der Instruktionssatz des Gepard besteht aus 24 Kernbefehlen, die im folgenden kurz mit ihrer Bedeutung aufgelistet werden [Aus98d].

`ABS Op2,An` - Def: $|Op2| \rightarrow An$
Absolutwert

`ADD Op1,Op2,An` - Def: $Op1 + Op2 \rightarrow An$
Addition

`AND Op1,Op2,An` - Def: $\forall i : Op1[i]\ and\ Op2[i] \rightarrow An[i]$
Bitweises logisches Und

`CMPZ Op2,An` - Def: $(Op2 == 0)\ ?\ (-1 \rightarrow An) : (0 \rightarrow An)$
Setzt/löscht einen ACCU in Abhängigkeit von Op2

`INTDIS` - Abstellen der Interrupts

`INTEN` - Einstellen der Interrupts

`JN Op1,addr` - Def: $(Op1 < 0)\ ?\ (addr \rightarrow PC) : (PC + 1 \rightarrow PC)$
Verzögerter Sprung zu addr, wenn Op1 negativ

`LDC #const,Op1` - Def: $const \rightarrow Op1$
Lade Konstante in Register Op1

`LDX (Op1),Op2` - Def: $X[Op1] \rightarrow Op2$
Lade Register aus dem X-Speicher

`LDY (Op1),Op2` - Def: $Y[Op1] \rightarrow Op2$
Lade Register aus dem Y-Speicher

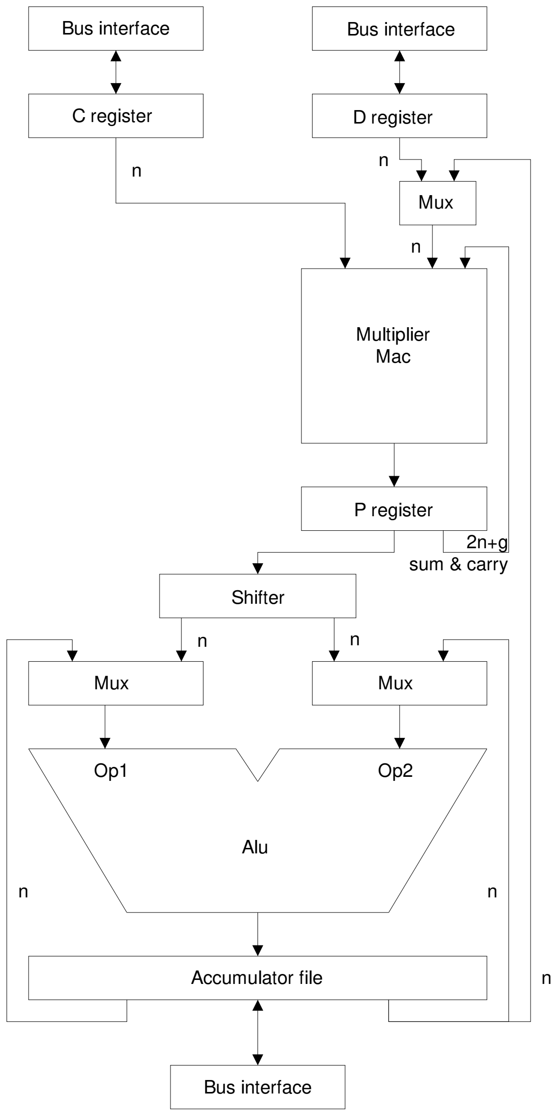


Abbildung 3.2: Datenpfad des Gepard

`LOOP Op1,addr` - Def: $Op1 \rightarrow LC, addr \rightarrow LE$
Starte Hardware-Schleife

`LSR Op2,An` - Def: $\forall i > 0 : Op2[i] \rightarrow An[i+1], 0 \rightarrow An[msb]$
Logisches Schieben nach rechts um eine Stelle

`MAC C,Op2` - Def: $P + C * Op2 \rightarrow P$
Multiply-Accumulate

`MNOP` - NOP im Multiplizierer

`MUL C,Op2` - Def: $C * Op2 \rightarrow P$
Multiplikation

`MV Op1,An` - Def: $Op1 \rightarrow An$
Kopiere Teile des P-Registers in Akkumulator

`NOP` -

`OR Op1,Op2,An` - Def: $\forall i : Op1[i]$ *or* $Op2[i] \rightarrow An$
Bitweises logisches Oder

`RETI` - Def: $LR1 \rightarrow PC$
Verzögerter Rücksprung vom Interrupt

`STX Op1,(Op2)` - Def: $Op1 \rightarrow X[Op2]$
Speichere Register in den X-Speicher

`STY Op1,(Op2)` - Def: $Op1 \rightarrow Y[Op2]$
Speichere Register in den Y-Speicher

`SUB Op1,Op2,An` - Def: $Op1 - Op2 \rightarrow An$
Subtraktion

`XOR Op1,Op2,An` - Def: $\forall i : Op1[i]\ xor\ Op2[i] \rightarrow An$
Bitweises logisches Xor

Dazu kommen noch sieben Makrobefehle, die sich aus den vorgenannten Kernbefehlen zusammensetzen.

`INCR x;` - Def: `ADD x,ONES,x;`
Erhöhe den Inhalt von Register x um 1

`DECR x;` - Def: `SUB x,ONES,x;`
Erniedrige den Inhalt von Register x um 1

`CLR x;` - Def: `ADD NULL,NULL,x;`
Lösche das Register x

`NOT x;` - Def: `XOR x,ONES,x;`
Negation

`LSL x;` - Def: `ADD x,x,x;`
Logisches Schieben nach links um eine Stelle

`J x;` - Def: `JN ONES,x;`
Verzögerter Sprung an Adresse x

`CALL x;` - Def: `J x; LDC #@+1,LR0;`
Funktionsaufruf

3.1.3 Parallelität auf Instruktionsebene

Der Gepard-Prozessor ist in der Lage mehrere Operationen in einem einzigen Takt auszuführen. Operationen werden dabei in vier Klassen eingeteilt:

ACCU alle Befehle, die auf den Akkumulator-Registern arbeiten, z.B. ADD, SUB, AND

MULT alle Befehle, die die Multipliziereinheit ansprechen: z.B. MUL, MAC

LDST die Load/Store-Befehle: LDX, LDY, STX, STY

OTHER dies sind alle anderen Befehle, vornehmlich Sprungbefehle wie JR, die Interruptbefehle INTEN und INTDIS und der Load-Constant-Befehl LDC

Die Befehle der LDST-Klasse führen alle Speicherzugriffe des Gepard-Kerns aus. Eine Kategorisierung der Zugriffe ist für das Verständnis der Parallelisierung verschiedener Operationen erforderlich. Dies führt zunächst auf eine genauere Beschreibung der Load/Store-Befehle am Beispiel des LDX-Befehls.

```
LDX (Op1),Op2
```

Die Adresse, von der ein Wert gelesen werden soll, steht im Indexregister Op1. Der gelesene Wert wird in das Zielregister Op2 geschrieben. Nach dem Speicherzugriff ist eine Indexregistermodifikation möglich. Dies kann ein Inkrement oder Dekrement im Bereich von -7 bis +7 oder die Addition des Inhalts des entsprechenden Modifikationsregisters sein. Im einfachsten Gepard-Core sind die Indexregister in Paaren angeordnet: I0/I1, I2/I3, I4/I5, I6/I7. Das zu einem bestimmten Indexregister gehörende Modifikationsregister ist dann gerade die andere Hälfte des Paares. I4 kann also mit dem Wert aus I5 modifiziert werden und I5 mit dem Inhalt von I4.

Die drei verschiedenen Speicherzugriffe werden nun wie folgt charakterisiert:

Full-Move bezeichnet einen Load/Store-Befehl mit beliebigem Zielregister und beliebiger Indexregistermodifizierung

Parallel-Full-Move entspricht dem Full-Move, wird aber parallel zu einem anderen Befehl ausgeführt

Parallel-Short-Move wird parallel zu einem anderen Befehl ausgeführt und besitzt die Einschränkung des Zielregisters auf ein arithmetisches Register und der fehlenden Möglichkeit der Indexregistermodifikation durch Inkrememnt oder Dekrement

Für die möglichen Kombinationen aller Operationen gelten nun folgende Bedingungen:

1. Wenn keine ACCU- oder MULT-Operation vorliegt, können zwei Full-Moves ausgeführt werden.
2. Ein ACCU-Befehl kann immer mit einem MULT-Befehl kombiniert werden.
3. Zusätzlich zu ACCU- bzw. MULT-Befehl können parallele Speicherzugriffe erfolgen. Dies können ein Parallel-Full-Move oder zwei Parallel-Short-Moves sein.
4. Die Befehle der OTHER-Klasse können mit keinem anderen Befehl kombiniert werden.
5. Falls zwei Moves in einer Instruktion auftreten, müssen sie verschiedene Speicherbänke adressieren.

Es ergeben sich dadurch folgende gültige Kombinationen:

- `jn i2,label;`
- `mul c,a0; add a3,a1,a1; ldy (i0),a3;`
- `sub a2,a3,a1; mac c,a1; stx a1,(i0); ldy (i6),c;`
- `mac c,a1; add a3,a2,a1;`
- `sub a3,a2,a1; sty a1,(i0)*;`
- `mul c,a0; mv ph1,a3; ldx (i6),c;`

- `mv ph1,a1; ldy (i2)+1,NULL;`
- `stx a3,(i4)*; sty a1,(i4)*;`

Ungültige Beispiele sind:

- `jn i2,label; add a3,a1,a1;`
  Kombination eines Befehls der OTHER-Klasse (jn) mit einem ACCU-Befehl.
- `sub a2,a3,a1; mac c,a1; stx a1,(i0)+1; ldy (i6),c;`
  Kombination eines Full-Move (stx) mit einem ACCU-Befehl und einem weiteren Move
- `add a3,a2,a1; mv ph1,a1;`
  Kombination zweier ACCU-Befehle
- `stx a3,(i4); sty a1,(i4)*;`
  Inkonsistente Indexregistermodifikation
- `ldx a3,(i4); stx a1,(i2);`
  Beide Moves adressieren die gleiche Speicherbank X

3.2 Das ANSI-C Compiler Frontend LANCE (V1)

Das LANCE(V1)-System umfaßt ein ANSI-C Compilerfrontend zur Übersetzung des Sourcecodes in maschinenunabhängigen Zwischencode (Intermediate Representation - IR) und mehrere Programme zur Optimierung des erzeugten Zwischencodes [DL99]. Die verschiedenen implementierten Optimierungen sind im folgenden aufgelistet.

- Constant Propagation
- Copy Propagation
- Constant Folding
- Dead Code Elimination
- Jump Optimization
- Global Common Subexpression Elimination
- Loop Invariant Code Motion

- Induction Variable Elimination
- Funtion Inlining
- If Optimization

Um das Ziel von LANCE, nämlich die Unterstützung bei der Erzeugung von effizientem Code für eingebettete Prozessoren, zu erreichen, besteht die Möglichkeit, über eine C++-Bibliothek, deren Aufbau in Abschnitt 3.2.4 auf Seite 36 näher beschrieben wird, auf alle Informationen der IR zuzugreifen. Auch Daten- und Kontrollflußanalysen werden unterstützt.

Im weiteren Verlauf dieses Kapitels wird kurz die Funktionsweise der LANCE-Programme und der Aufbau des Zwischencodes dargestellt. Im Anschluß daran folgt die Erläuterung der wichtigsten Klassen der LANCE-Klassenbibliothek.

3.2.1 Benutzung der LANCE-Programme

Alle Programme können direkt von der Kommandozeile gestartet werden. Für komplexe Analysen des Zwischencodes ist die Nutzung der vorhandenen graphischen LANCE-Benutzeroberfläche zu empfehlen. Da diese Analysen aber nicht Bestandteil dieser Arbeit sind, gehe ich im weiteren nur auf die Benutzung der Basisprogramme ein.

Am Anfang jedes Übersetzungsvorgangs steht der Aufruf des Frontends:

```
analyze <sourcefile.c> [ox-options]
```

Als erste Option muß dabei die ANSI-C Quelldatei angegeben werden. Die weiteren Optionen gelten für den internen Parser `ox` und brauchen nur angegeben werden, wenn das Frontend dies durch eine entsprechend ausführliche Fehlermeldung fordert. Nach Programmende sind im Ausgabeverzeichnis von LANCE zwei neue Dateien zu finden: `sourcefile.c.ir` und `sourcefile.c.st`. Die erste enthält den eigentlichen Zwischencode in Drei-Adreß-Form, die zweite die Symboltabelle (s. Abschnitt 3.2.3 auf der nächsten Seite).

Der Aufruf der Optimierungen folgt immer demselben Prinzip: als erste und in der Regel einzige Option muß die C-Quelldatei angegeben werden.

```
dce <sourcefile.c>
```

Dabei ist es unerheblich, ob diese Datei (`sourcefile.c`) im aktuellen Verzeichnis existiert, da die Optimierungen die Quelldatei nicht mehr lesen müssen. Der Name der Quelldatei wird nur zur Bestimmung der Namen der Symboltabellen- und Zwischencodedateien benötigt. Diese werden dann eingelesen und eventuell verändert (optimiert) unter dem gleichen Namen wieder im Ausgabeverzeichnis von LANCE abgespeichert.

Die Reihenfolge, in der der Benutzer die Optimierungen anwendet, ist ihm überlassen. Ein Shellskript, daß automatisch alle Optimierungen zum Teil auch mehrfach nacheinander anwendet, ist aber vorhanden (`optscript`).

3.2.2 Konfiguration von LANCE

Die Konfiguration des Frontends ist möglich durch Editieren zweier Textdateien, `IR.cfg` und `WIDTH.cfg`.

Drei der fünf Optionen der Datei `IR.cfg` beeinflussen die Art und Weise wie sogenannte High-Level-Anweisungen vom Frontend übersetzt werden. Zu den High-Level-Anweisungen gehören hier die ANSI-C Konstrukte der If-Anweisung, der For-Schleife und der Arrayzugriffe über Indizes. Zu beachten ist, daß einige der Optimierungen, vor allem diejenigen, die auf einer Datenflußanalyse beruhen, nur durchführbar sind, wenn im Zwischencode keine dieser High-Level-Anweisungen mehr vorhanden sind. In Tabelle 3.2 auf der nächsten Seite sind die Optionen zusammen mit dem Typ ihres Wertes und den jeweils eingestellten Werten angegeben.

In der Konfigurationsdatei `WIDTH.cfg` werden die Größen der C-Datentypen hinterlegt. Außerdem muß dem Frontend die Größe der kleinsten adressierbaren Einheit im Speicher mitgeteilt werden. Alle Größen werden in Bits angegeben. In Tabelle 3.3 auf der nächsten Seite sind die Werte eingetragen, die für den Zielprozessor AMS Gepard anzugeben sind.

3.2.3 Aufbau der IR

Als Beispiel soll hier die Übersetzung des *real_update* Kernel-Benchmarks (Abbildung 3.3 auf Seite 33) aus dem DSPstone Projekt [Ziv94] gezeigt werden. Der Zwischencode, der in Abbildung 3.4 auf Seite 34 dargestellt ist, repräsentiert die IR nach Anwendung des Frontends und des Optimierungsskripts `optscript`. Ein Teil der resultierenden Symboltabelle ist in Abbildung 3.5 auf Seite 35 zu finden.

Der Zwischencode ist wie auch der Sourcecode in Funktionen eingeteilt. Diese enthalten verschiedene Arten von Anweisungen. In Abbildung 3.4 auf Seite 34 ist deren Drei-Adreß-Form gut zu erkennen.

Alle Anweisungen beginnen mit der Angabe der Nummer der repräsentierten Sourcecodezeile, eingeschlossen in Klammeraffen (`@`), um die Analyse des Zwischencodes zu erleichtern. Der vorherrschende Anweisungstyp ist hier die Zuweisung. Als einziger zusätzlicher Typ tritt in diesem Beispiel eine ReturnVal-Anweisung am Ende der Main-Funktion auf. Alle weiteren Anweisungstypen, die hier nicht zu sehen sind, werden in Abschnitt 3.2.4 auf Seite 36 vorgestellt.

Option	Typ	Wert	Erklärung
IR_split_conditionals	boolean	1	gibt an, ob bedingte Anweisungen in einfache IR Anweisungen (bedingte Sprünge) zerlegt werden sollen
IR_split_forloop	boolean	1	gibt an, ob Schleifen in einfache IR Anweisungen zerlegt werden sollen
IR_split_index	boolean	1	gibt an, ob bei Arrayzugriffen die Adresse des Zugriffs explizit mit einfachen IR Anweisungen berechnet werden soll
IR_saturation	boolean	0	gibt an, ob das Frontend die arithmetischen Operatoren ’+’, ’-’ und ’*’ in spezielle Operatoren für die Sättigungsarithmetik übersetzen soll, die einige DSPs unterstützen
IR_constant_folding	boolean	0	gibt an, ob das Frontend bereits ein einfaches Constant Folding durchführen soll

Tabelle 3.2: Optionen in `IR.cfg`

Option	Typ	Wert	Erklärung
WIDTH_addressable	integer	16	kleinste adressierbare Speichereinheit
WIDTH_pointer	integer	16	Größe des Pointer-Typs
WIDTH_short_int	integer	16	Größe eines *short int*
WIDTH_int	integer	16	Größe eines *int*
WIDTH_long_int	integer	16	Größe eines *long int*
WIDTH_float	integer	16	Größe eines *float* (einfache Genauigkeit)
WIDTH_double	integer	16	Größe eines *double* (doppelte Genauigkeit)

Tabelle 3.3: Optionen in `WIDTH.cfg`

```
#define STORAGE_CLASS register
#define TYPE  int

void
pin_down(TYPE *p)
{
  *p = 0 ;
}

TYPE
main()
{
  static TYPE A = 10 ;
  static TYPE B = 2 ;
  static TYPE C = 1 ;
  static TYPE D = 0 ;

  STORAGE_CLASS TYPE *p_a = &A ;
  STORAGE_CLASS TYPE *p_b = &B ;
  STORAGE_CLASS TYPE *p_c = &C ;
  STORAGE_CLASS TYPE *p_d = &D ;

  pin_down(&D) ;

  *p_d  = *p_c + *p_a * *p_b ;

  pin_down(&D) ;

  return(0)  ;
}
```

Abbildung 3.3: DSPstone Kernel Benchmark `real_update.c`

```
function 'pin_down'(13):
@37@ (ptr 'p'(12)) = INT(0) ;
@0@ return ;

function 'main'(22):
@50@ '_t_24'(26) = & 'A'(14) ;
@51@ '_t_28'(30) = & 'B'(15) ;
@52@ '_t_32'(34) = & 'C'(16) ;
@53@ '_t_122'(122) = & 'D'(17) ;
@55@ '_t_44'(46) = call 'pin_down'(13) ('_t_122'(122) ) ;
@58@ '_t_52'(54) = (ptr '_t_28'(30)) ;
@58@ '_t_56'(58) = (ptr '_t_24'(26)) ;
@58@ '_t_50'(52) = '_t_56'(58) * '_t_52'(54) ;
@58@ '_t_60'(62) = (ptr '_t_32'(34)) ;
@58@ '_t_48'(50) = '_t_60'(62) + '_t_50'(52) ;
@58@ (ptr '_t_122'(122)) = '_t_48'(50) ;
@61@ '_t_70'(72) = call 'pin_down'(13) ('_t_122'(122) ) ;
@63@ returnval INT(0) ;
```

Abbildung 3.4: Zwischencode von `real_update.c` (→`real_update.c.ir`)

Anweisungen bestehen aus Ausdrücken. Dabei kann es sich um binäre und unäre Ausdrücke, aber auch um Funktionsaufrufe, Zeiger- und Symbolausdrücke handeln. Letztere verweisen jeweils auf ein Symbol der Symboltabelle (Abbildung 3.5 auf der nächsten Seite).

Zu jedem Symbol werden dessen elf Eigenschaften in der Symboltabelle gespeichert. Das sind (vgl. mit der Symboltabelle in Abbildung 3.5 auf der nächsten Seite):

N ein eindeutiger Index

ID der Bezeichner der entsprechenden Variable im Sourcecode oder aber `_t_n` für ein vom Frontend erzeugtes Symbol (`n` eine natürliche Zahl)

CLS die Klasse des Symbols

SRC Name der Sourcecode-Datei und Zeilennummer der entsprechenden Variablendeklaration

I ein Informationstext, der einige Symboleigenschaften zusammenfaßt

TP der Typ der repräsentierten Variablen

IT der Wert, mit dem die repräsentierte Variable im Sourcecode explizit initialisiert wird

P falls das Symbol ein Funktionsargument darstellt, ist hier die Position des Arguments in der Argumentliste angegeben

```
N:   16
ID: 'C'
CLS: obj
SRC: 'real_update.c':47
I: lvar of 'main'(22)
TP:
        sta int
IT: --- ;
P: 1
MEM_LOC: 0
MEM_CNT: 0
POINTS_TO: 0

N:   17
ID: 'D'
CLS: obj
SRC: 'real_update.c':48
I: lvar of 'main'(22)
TP:
        sta int
IT: --- ;
P: 1
MEM_LOC: 0
MEM_CNT: 0
POINTS_TO: 0

N:   18
ID: 'p_a'
CLS: obj
SRC: 'real_update.c':50
I: lvar of 'main'(22)
TP:
        reg ptr to int
IT: --- ;
P: 1
MEM_LOC: 0
MEM_CNT: 0
POINTS_TO: 0
```

Abbildung 3.5: Ausschnitt aus der Symboltabelle `real_update.c.st`

MEM_LOC gibt die Speicherklasse oder die zugewiesene Speicherbank des Symbols an

MEM_CNT Anzahl der Speicherzugriffe, die auf den Speicherort dieses Symbols erfolgen

POINTS_TO enthält den Index des Symbols, auf das das aktuelle Symbol verweist

Die letzten drei Eigenschaften werden bei der Vorstellung der Speicherpartitionierung in Kapitel 4 auf Seite 53 noch näher beschrieben.

Der Zugriff auf alle Informationen des Zwischencodes und der darin enthaltenen Symbole wird über eine C++-Klassenbibliothek abgewickelt. Diese soll im folgenden näher dargestellt werden.

3.2.4 Klassenbibliothek von LANCE

Der Aufbau der Klassenbibliothek folgt dem hierarchischen Aufbau des Zwischencodes. Die oberste Zugriffsebene ist die Funktionsliste (`class FunctionList`), über die auf alle Funktionen der Reihe nach zugegriffen werden kann. Funktionen selbst (`class Function`) sind als Statement-Liste realisiert. Diese kann auch linear traversiert werden. Die einzelnen Statements (`class IRStm`) repräsentieren die verschiedenen Arten von Anweisungen, die abhängig von ihrem Typ in unterschiedlicher Weise Ausdrücke (`class IRExp`) enthalten:

IRS_ASS Zuweisung, besteht aus einem Ziel- und einem Quellausdruck

IRS_GOTO Sprunganweisung, verweist auf eine Label-Anweisung

IRS_RET Rücksprunganweisung, kann auch einen Rücksprungausdruck enthalten

IRS_LABEL Label-Anweisung

IRS_GUARDED Bedingte Anweisung, die Bedingung (Guard) besteht aus einem Ausdruck, die enthaltene Anweisung kann einen beliebigen Typ besitzen

IRS_CONT Continue-Anweisung in For-Schleifen

IRS_BRK Break-Anweisung in For-Schleifen

IRS_IT If-Then-Anweisung

IRS_ITE If-Then-Else-Anweisung

IRS_FOR For-Schleife

Die letzten fünf Anweisungstypen werden als High-Level-Typen bezeichnet. Ihr Auftreten im Zwischencode ist abhängig von der Konfiguration des Frontends. Sollen Datenflußanalysen auf dem Zwischencode ausgeführt werden, sind High-Level-Anweisungen verboten. Vorteilhaft ist ihre Verwendung zur Generierung von Zwischencode einer höheren Abstraktionsstufe. Im weiteren wird auf ihre Verwendung verzichtet.

Die Ausdrücke, die in den Anweisungen enthalten sind, lassen sich wiederum in verschiedene Klassen einteilen:

IRE_BIN Binärer Ausdruck, enthält wieder zwei Ausdrücke

IRE_UNA Unärer Ausdruck, enthält einen Ausdruck

IRE_CALL Funktionsaufruf

IRE_CAST Ausdruck zur Typumwandlung

IRE_CONST Konstante

IRE_INT Ganzzahlausdruck

IRE_SYM Symbol

IRE_PTR Zeigerausdruck

IRE_IDX Indexausdruck

Das Vorkommen von Indexausdrücken ist abhängig von der Frontendkonfiguration (Option `IR_split_index`). Symbolausdrücke verweisen auf Symboleinträge (`class Symbol`) in der Symboltabelle (`class SymbolTable`). Sie besitzen einen eindeutigen Schlüssel, über den der Eintrag gefunden werden kann.

Symbole können entweder Variablen oder Funktionen repräsentieren oder vom Frontend generiert sein. Im ersten Fall werden sie auch mit dem Variablen- oder Funktionsnamen bezeichnet, im zweiten mit einem eindeutigen Namen der Form `_t_n` mit `n` als ganzer Zahl. Im entsprechenden Eintrag der Symboltabelle werden neben dem Index und dem Namen noch Informationen über Klasse, Geltungsbereich, Typ etc. festgehalten.

Für weitere Informationen zum Aufbau der Klassenbibliothek und dem C++-Interface sei auf die LANCE-Dokumentation [DL99] verwiesen.

3.3 Das Gepard Compiler Backend GEPCC

Der GEPCC wurde am Lehrstuhl für Programmiersprachen und Compilerbau der BTU Cottbus entwickelt. Es handelt sich um ein Compiler-Backend, das auf dem Zwischencodeformat des LANCE(V1)-Systems aufsetzt und Assemblercode für den AMS Gepard Prozessor Kern erzeugt.

Dieses Kapitel soll einen Überblick über den internen Aufbau des GEPCC und die Struktur des generierten Assemblercodes geben. Es liefert damit das notwendige Wissen zum Verständnis der nachfolgenden Kapitel zur Speicherpartitionierung. Grundlage dieser Beschreibung ist die Dokumentation von Schölzel in [Sch99].

3.3.1 Bedienung des Frontends

Das Backend GEPCC startet implizit das Frontend `analyze` des LANCE(V1)-Systems zur Erzeugung der benötigten Dateien mit dem Zwischencode und der Symboltabelle. Daher ist ein vorheriger expliziter Aufruf des Frontends nicht unbedingt nötig. Wenn allerdings noch Zwischencodeoptimierungen durchgeführt werden sollen, ist dies unumgänglich. Für diesen Fall unterstützt der GEPCC einen Kommandozeilenparameter, der die Ausführung des Frontends unterbindet, und damit die Nutzung vorhandener Zwischencodedateien erlaubt.

Der Aufruf des GEPCC erfolgt folgendermaßen:

```
gepcc [-o Zielname][-s Stackbeginn][-m Membeginn Memende ...]
                [-f Nachkomma][-l]Quelldatei ...
```

Dabei haben die verschiedenen Optionen folgende Bedeutung:

-o Durch Angabe dieser Option wird der generierte Assemblercode in der Datei `Zieldatei` gespeichert. Standardmäßig erfolgt die Ausgabe in einer Datei `out.asm` im aktuellen Verzeichnis.

-s Hier kann der Beginn des Stackbereichs im Speicher als Dezimalzahl angegeben werden. Dies sollte die letzte verfügbare Speicheradresse sein. Standardmäßig wird hier 65535 für den 64kB großen Speicher des Gepardtyps 03verwendet.

-m Mit dieser Option ist es möglich Speicherbereiche für globale Variablen anzugeben. Ein Speicherbereich ist durch Anfangsadresse und Endadresse bestimmt. Beide Adressen gehören mit zum jeweiligen Bereich. Es können bis zu fünf Speicherbereiche angegeben werden, so daß maximal zehn Adressen spezifiziert werden müssen.

-f Hier kann die Anzahl der Nachkommastellen der Festkommazahlen des übersetzten C-Programms angegeben werden. Sie darf sich im Bereich 1 bis 14 befinden.

-l Diese Option verhindert den impliziten Aufruf des Frontends. Bei ihrer Angabe müssen daher die Dateien für Zwischencode und Symboltabelle im LANCE-Output-Verzeichnis existieren.

Quelldatei Hier muß mindestens eine zu übersetzende C-Datei angegeben werden. Falls die Option `-l` fehlt, wird zuerst für jede der Quelldateien das Frontend `analyze` aufgerufen. Dann erfolgt die Codegenerierung und die Ausgabe des Assemblercodes in einer Datei.

Bei der Konfiguration von LANCE für die Codegenerierung mit GEPCC ist zu beachten, daß Highlevel-Anweisungen im Zwischencode nicht erlaubt sind. Demzufolge müssen die Optionen `IR_split_conditionals`, `IR_split_forloop` und `IR_split_index` aktiviert sein. Zusätzlich wird empfohlen auch `IR_constant_folding` zu aktivieren und `IR_saturation` auszuschalten.

In der Konfigurationsdatei `WIDTH.cfg` muß für alle Typen eine Bitbreite von 16 eingetragen werden.

3.3.2 Interne Struktur des GEPCC

Die Wertebereiche für die verschiedenen C-Datentypen sind in Tabelle 3.4 auf der nächsten Seite aufgeführt. Zu beachten ist, daß die Gleitkommatypen `float` und `double` als Festkommazahlen interpretiert werden müssen, da der Gepard ein DSP mit Festkommaarithmetik ist. Die Genauigkeit dieser Festkommazahlen beträgt vier Nachkommastellen. Über eine Kommandozeilenoption des Backends kann dies aber an eigene Bedürfnisse angepaßt werden.

Alle Variablen, globale und lokale, werden ausschließlich im X-Speicher allokiert. Der Y-Speicher wird in der ursprünglichen Version des Backends, d.h. ohne die Anpassungen, die im Verlauf dieser Arbeit gemacht wurden, nicht genutzt.

Auch der Stack wird in der X-Speicherbank verwaltet. Sein Adreßbereich beginnt bei 65535 (s.a. Kommandozeilenoption `-s`) und wächst hin zu den kleineren Adressen. In ihm werden bis auf die globalen Variablen alle anderen zu speichernden Werte abgelegt. Als Stackpointer wird das Indexregister I7 verwendet. Es enthält immer die Adresse des obersten Stackelementes, d.h. des Elementes mit der niedrigsten Speicheradresse. Die beiden Operationen *Push* und *Pop* werden dann wie folgt realisiert:

C-Datentyp	Datenworte	Wertebereich	Anmerkung
char	1	-32768 bis 32767	
int	1	-32768 bis 32767	
long	1	-32768 bis 32767	
unsigned char	1	0 bis 65535	
unsigned int	1	0 bis 65535	
unsigned long	1	0 bis 65535	
float	1	abh. von der Kommaposition	Implementierung als Festkommazahl
double	1	abh. von der Kommaposition	Implementierung als Festkommazahl
pointer	1	0 bis 65535	
bitfield			
array			
struct			
union			
enum	1	-32768 bis 32767	

Tabelle 3.4: Umsetzung der C-Datentypen im Backend GEPCC

Push: `ldx (i7)--,null`
`stx c,(i7)`

Pop : `ldx (i7)++,c`

Vor Funktionsaufrufen werden alle Parameter, die an die Funktion übergeben werden sollen, vom Aufrufer der Reihe nach ,angefangen mit dem rechtesten, auf dem Stack gespeichert, so daß im Anschluß daran der erste Parameter die niedrigste Speicheradresse besitzt. Durch das Assemblermakro `call` wird dann beim Funktionsaufruf die Rücksprungadresse im Linkregister LR0 gesichert. Gibt die aufgerufene Funktion einen primitiven Datentyp als Wert zurück, wird dieser im Akkuregister A3 erwartet. Falls es sich um einen komplexen Rückgabewert handelt, wird von der aufgerufenen Funktion zusätzlicher Speicher für diesen Wert angelegt und dessen Adresse im Indexregister I5 übergeben.

Die aufgerufene Funktion speichert zuerst den aktuellen Wert des Stackpointers auf dem Stack und schreibt diesen Wert gleichzeitig auch im Indexregister I6. Es dient im weiteren Verlauf als Framepointer, über den der Zugriff auf die Funktionsparameter und die lokalen Variablen abgewickelt wird.

Danach wird ein *Framebereich* auf dem Stack eingerichtet, der Platz für alle lokalen Variablen dieser Funktion bietet. Erst dann werden neben dem Register LR0 mit der Rücksprungadresse

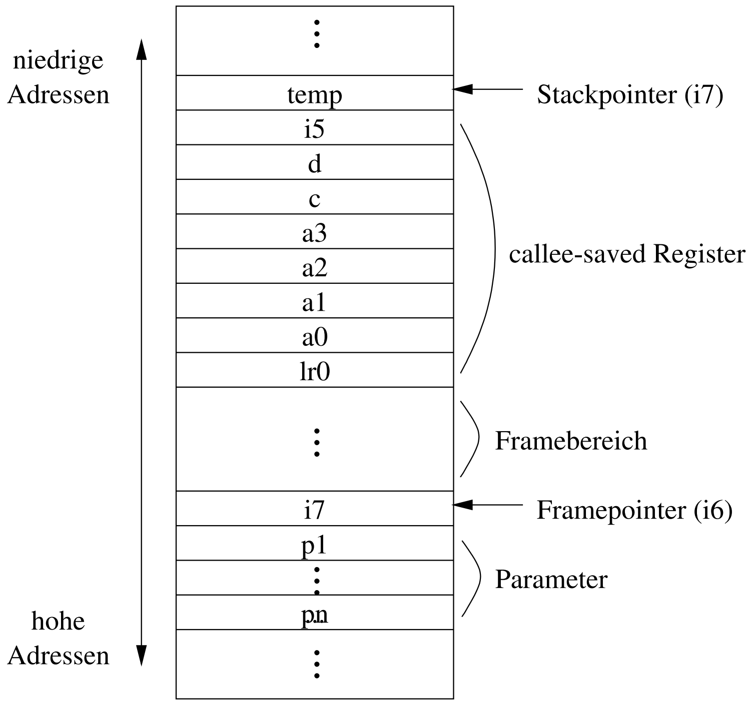


Abbildung 3.6: Speicherorganisation einer Funktion

auch die Akkumulatoren A0, A1, A2, A3 sowie die Register C und D auf dem Stack gesichert (*callee-saved*). Hat die Funktion einen komplexen Rückgabewert wird dort außerdem noch der Inhalt des Indexregisters I5 gespeichert.

Durch diese Speicherorganisation sind die Adressen aller Parameter und lokalen Variablen durch ihre Position relativ zum Framepointer bestimmt (Abbildung 3.6). Die Funktionsparameter liegen dabei im Vergleich zum Framepointer bei höheren Adressen, d.h. ihre Speicheradresse berechnet sich durch Addition eines Offsets zum Framepointer. Dagegen sind die lokalen Variablen bei niedrigeren Adressen zu finden. Ihre Speicheradresse erhält man daher durch Subtraktion eines Offsets vom Wert des Framepointers.

Um so den Wert eines Parameters einer Funktion in das Akkuregister A0 zu laden, muß folgende Befehlssequenz verwendet werden:

`ldx (i6),i0`	Laden des Framepointerwertes als Adreßbasis
`ldc #offset,i1`	Laden des Offsets des Parameters zur Adreßbasis
`ldx (i0)*,null`	Berechnung des Speicheradresse durch Indexregistermodifikation
`ldx (i0),a0`	Laden des Parameterwertes in den Akku

Der Offset des ersten Parameters ist immer 1. Bei weiteren Parametern ist für die Größe des Offsets auch noch der Datentyp der vorherigen Parameter entscheidend. So muß für die Berechnung der Adresse des zweiten Parameters die folgende Logik verwendet werden:

```
ldc #(1 + sizeof(erster Parameter)),i1
```

Der Unterschied beim Zugriff auf lokale Variablen beschränkt sich auf das Laden eines negativen Offsets im zweiten Schritt der obigen Befehlssequenz.

Laufzeitfehler werden vom Backend GEPCC aufgrund der negativen Auswirkungen auf die Laufzeit eines Programms nicht behandelt. Der Programmierer hat daher dafür Sorge zu tragen, daß Fehler wie ein Stacküberlauf oder auch eine Division durch Null nicht auftreten.

3.3.3 Realisierung der Codegenerierung

Die Codegenerierung läßt sich allgemein in die drei Phasen *Instruktionsauswahl*, *Registerallokation* und *Instruktionsanordnung* einteilen.

Bei der Instruktionsauswahl wird die Liste der Zwischencodeanweisungen auf eine semantisch äquivalente Reihenfolge von Assembleroperationen abgebildet. Da diese Abbildung nicht eindeutig ist, besteht das Problem, eine optimale Abbildung bezüglich der resultierenden Codequalität zu finden. Die Codequalität läßt sich nach verschiedenen Kriterien beurteilen. Dies kann die Codegröße, also der Programmspeicherbedarf, die benötigte Größe des Datenspeichers oder auch die Laufzeit des Programms sein. Meist werden die Anforderungen durch die Anwendung selbst bestimmt.

Die Registerallokation weist jedem Wert des Zwischencodes ein Register des Zielprozessors zu, so daß die Zahl der Speicherzugriffe nach Möglichkeit minimiert wird.

Instruktionsauswahl beschreibt den Prozeß des Ordnens und Kompaktierens der Assembleroperationen, so daß die Möglichkeiten der Parallelität auf Instruktionsebene des Zielprozessors voll ausgeschöpft werden.

Das Backend GEPCC soll nun daraufhin untersucht werden, in wie weit diese drei Phasen implementiert sind und welche Strategien dabei zum Einsatz kommen.

Instruktionsauswahl

Beim GEPCC ist keine Strategie zur optimalen Instruktionsauswahl implementiert. Die zu übersetzenden Zwischencodeanweisungen werden in der gleichen Reihenfolge, in der sie im Zwischencode erscheinen, auch bearbeitet. Für jede Anweisung gibt es eine semantisch entsprechende Sequenz von Assemblerbefehlen (s. Kapitel 3.4 in [Sch99]). Diese wird in das Assemblerprogramm eingefügt, sobald die entsprechende Anweisung im Zwischencode erscheint.

Registerallokation

Der Registerallokator des GEPCC unterscheidet vier Klassen von Variablen, die bei der Allokation unterschiedlich behandelt werden:

Globale bzw. statische Variablen Sie werden im globalen Adreßraum gespeichert. Jede Variable erhält dabei eine absolute Adresse. Die Lage des Adreßraums kann dem Backend als Kommandozeilenoption mitgeteilt werden. Befindet sich eine derartige Variable in einem Register und wird sie dort verändert, wird die Änderung auch sofort in den Speicher übertragen.

Lokale Variablen Dies sind die innerhalb einer Funktion deklarierten Variablen. Sie werden auf dem Stack gespeichert und sind über den Framepointer adressierbar. Auch ihre Änderungen werden sofort in den Speicher übertragen.

Funktionsparameter Sie entsprechen im wesentlichen den lokalen Variablen. Der einzige Unterschied ist, daß ihre Lage im Speicher schon am Anfang einer Funktion feststeht. Die Allokation der lokalen Variablen und damit die Festlegung der Speicheradresse wird dagegen erst bei ihrer ersten Benutzung durchgeführt.

Temporäre Variablen Das Frontend generiert diese Variablen. Sie besitzen in der Regel keinen Speicherort wie die Variablen der beiden anderen Klassen, sondern werden nur in den Registern gehalten. Reichen die Register allerdings nicht zum Zwischenspeichern aller benötigten Werte aus, können temporäre Variablen auch in den Speicher geschrieben werden (*Spilling*). Dabei werden sie aber nur dann dort auf dem Stack gesichert, falls sie im weiteren Programmverlauf noch benötigt werden.

Da alle arithmetischen und logischen Befehle des Gepard Register als Quelle bzw. Ziel verwenden, muß der Registerallokator die Werte der oben beschriebenen Variablen auch dort bereitstellen. Dabei werden die verschiedenen Register des Gepard wie folgt verwendet:

A0 ... **A3** Die Akkuregister können als Quell- und Zielregister verwendet werden.

C,D Diese beiden Register können nur als Quellregister benutzt werden.

P Das P-Register entält das Ergebnis einer Multiplikation. Dies wird jedoch immer sofort anschließend in ein Akkuregister transferiert, so daß es nur als temporärer Zwischenspeicher dient.

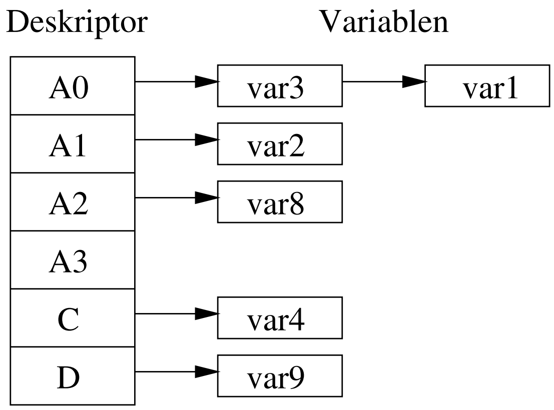


Abbildung 3.7: Datenstruktur für den Registerzugriff

LR0,LR1 Die beiden Linkregister werden nur für die Speicherung von Rücksprungadressen verwendet und bei der normalen Funktionsausführung nicht weiter benutzt.

I0, ... **I7** Für jeden Zugriff auf den Speicher werden die Indexregister benötigt. Zusätzlich fungiert I6 als Framepointer und I7 als Stackpointer.

Der Registerallokator kann über einen Deskriptor die Register A0 ... A3, C und D direkt ansprechen. Damit ist auch der Zugriff auf die in den Registern aktuell enthaltenen Variablen möglich. Eine Besonderheit dieser Datenstruktur ist, daß ein Register unter bestimmten Umständen mehrere Variablen enthalten kann (Abbildung 3.7). Dieser Fall tritt dann auf, wenn im Zwischencode eine direkte Wertzuweisung zwischen zwei Symbolen stattfindet. Statt für die Zielvariable ein neues Register zu allokieren, wird in dem Deskriptor, der den Verweis auf die Quellvariable enthält, nur ein Verweis auf diese neue Variable hinzugefügt.

Die Strategien, die der Registerallokator zur Bestimmung von zu benutzenden Indexregistern und Akkuregistern zusammen mit C- und D-Register verwendet, werden im folgenden kurz dargestellt:

- **Indexregisterwahl**

  Der Zugriff auf den Speicher geschieht über die Indexregister. Falls die Speicheradressen durch Offsetsubtraktion oder -addition berechnet werden müssen, wird ein Indexregisterpaar benötigt. Die Wahl des Indexregisterpaares folgt keiner besonderen Strategie. Das erste freie Paar wird verwendet; wenn es kein freies Indexregisterpaar mehr gibt, wird eins, beginnend mit dem ersten (I0/I1), überschrieben. Wird nur ein Indexregister für den Speicherzugriff benötigt, wird analog verfahren.

- **Wahl eines allgemeinen Registers (A0 ... A3,C,D)**

  Wenn ein Register mit einer Quellvariablen VarQ geladen werden soll, werden die zulässigen Register danach untersucht, ob

  1. diese Variable schon in einem Register gespeichert ist. Dann ist dieses das Quellregister.
  2. es ein freies Register gibt. Dann wird VarQ in diese Register geladen.

  Falls keiner der beiden Fälle zutrifft, muß ein Register geleert werden. Dazu wird untersucht, in welchem der Register die wenigsten zu sichernden Variablen gehalten werden. Dieses Register wird dann verwendet.

  Die Registersuche für eine Zielvariable VarZ unterscheidet sich etwas von dem gerade beschriebenen Verfahren. Hier wird zuerst überprüft, ob

  1. diese Variable als einzige Variable in einem Register gespeichert ist. Dann ist dieses das gesuchte Zielregister.
  2. diese Variable mit anderen Variablen in einem Register gespeichert ist. Dann lösche diese Variablenreferenz aus dem Registerdeskriptor und fahre mit dem nächsten Punkt fort.
  3. es ein freies Register gibt. Dann wird VarZ in dieses Register geschrieben.

  Falls nun auch hier keiner der Fälle zutrifft, wird die gleiche Strategie wie bei den Quellvariablen oben angewandt.

Instruktionsanordnung

Diese Phase der Codegenerierung ist im GEPCC nicht implementiert. Alle Assemblerbefehle werden sequentiell in der Reihenfolge der entsprechenden Zwischencodeanweisungen ausgeführt. Eine Kompaktierung findet nicht statt.

3.3.4 Bewertung

Die generierte Codequalität des Backends GEPCC ist schlecht. In keiner der drei Phasen der Codegenerierung wurde eine Strategie implementiert, die eine auch nur fast optimale Lösung des Teilproblems erwarten läßt. Außerdem werden die Möglichkeiten, die die Architektur des Gepard bietet, nicht ausgenutzt. Dies ist auf der einen Seite die Existenz einer zweiten Speicherbank und damit auch der Möglichkeit, Operanden parallel in einer Instruktion laden zu können. Auf der anderen Seite können unter bestimmten Umständen auch andere Befehle des Gepard parallelisiert werden. Auch dies wird im GPECC durch die fehlende Phase der Instruktionsanordnung und Kompaktierung nicht unterstützt.

Die Einschränkung des Backends auf die ausschließliche Nutzung des X-Speichers soll nun in dieser Arbeit beseitigt werden. Um die daraus erzielten Ergebnisse bewerten zu können, ist aber auf jeden Fall noch die Realisierung einer Kompaktierung des Gepard-Codes notwendig, mit der die Möglichkeiten der Parallelität auf Instruktionsebene ausgenutzt werden können. Eine weitere Verbesserung der Codequalität wäre sicherlich durch eine Neuimplementierung des Backends mit einer optimalen Instruktionsauswahl durch *Tree-Parsing* und speziellen Techniken der Registerallokation möglich. Dies würde allerdings den Rahmen dieser Arbeit sprengen und wird hier nicht weiter verfolgt.

Die Implementierung des Kompaktierers soll nun im nächsten Kapitel vorgestellt werden.

3.4 Der Gepard Kompaktierer GEPPAC

3.4.1 Problembeschreibung

Der Gepard C-Compiler GEPCC generiert vorwiegend sequentiellen Assemblercode. Nur einige wenige interne Codebausteine werden als paralleler Code erzeugt. Diese Bausteine werden in jedes Programm eingefügt und können daher im voraus optimiert werden. Ziel der hier beschriebenen Kompaktierung soll es sein, alle im Assemblercode auftretenden Operationen derart in Instruktionen zu kompaktieren, daß die Möglichkeiten des Gepard-Kerns zur Parallelität auf Instruktionsebene voll ausgenutzt werden und damit die Geschwindigkeit des Programms maximiert und der Platzbedarf minimiert wird. Die Aufgabe besteht also darin, alle Gepard-Befehle eines bestehenden Assemblerprogramms in möglichst wenige Instruktionen einzufügen, wobei auf der einen Seite die Datenabhängigkeiten zwischen den einzelnen Befehlen und auf der anderen Seite die Restriktionen des Gepard bezüglich der Kombination verschiedener Befehle in einer Instruktion berücksichtigt werden müssen. Es wurde gezeigt, daß das Finden einer optimalen Lösung dieses Kompaktierungsproblems NP-vollständig ist (Resource Constrained Scheduling, Problem SS10 in [GJ79]).

Zur zeiteffizienten Lösung des Problems muß also auf heuristische Verfahren zurückgegriffen werden, die jedoch keine Optimalität des Ergebnisses garantieren können, oftmals aber einen guten Kompromiß zwischen Laufzeit des Verfahrens und Güte der erreichten Lösung darstellen. Für GEPPAC wurde daher ein List-Scheduling-Verfahren implementiert, das die Optimierung lokal, d.h. für jeden Basisblock einzeln, vornimmt.

List-Scheduling startet im allgemeinen mit einer leeren Instruktionsliste. Schrittweise werden nun Befehle in die letzte Instruktion der Liste eingefügt, wenn

1. sie *Data Ready* sind (, d.h. alle benötigten Datenwerte zur Verfügung stehen),
2. sie die höchste Priorität unter allen Befehlen besitzen, die *Data Ready* sind,
3. das Einfügen in diese Instruktion für den speziellen Befehl möglich ist.

Die Priorität eines Befehls wird mit Heuristiken berechnet. Gegen das Einfügen eines Befehls können Architekturrestriktionen sprechen, so daß z.B. nicht zwei Sprungbefehle gleichzeitig in einer Instruktion stehen dürfen.

Mit der Kenntnis des allgemeinen Verfahrens wird der nun folgende spezielle Algorithmus für die Kompaktierung des Gepard-Assemblercodes leichter verständlich.

3.4.2 Implementierung des Verfahrens

Der Ablauf der Kompaktierung ist wie folgt:

1. Einlesen des Assemblercodes. Dabei werden alle Befehle in einer sequentiellen Liste gespeichert, also auch alle durch das Backend schon parallelisierten Operationen.
2. Diese Befehlsliste wird in Basisblöcke aufgeteilt. Ein Basisblock besteht aus einem Präfix (einem Label am Beginn des Blocks), einem Postfix (einem Sprung am Ende des Blocks zusammen mit dem Befehl im Branch-Delay-Slot), und den restlichen Befehlen, die dann später parallelisiert werden können. Präfix und Postfix können leer sein.
3. Für jeden Basisblock wird nun ausgeführt:
   a) Erstelle den Datenabhängigkeitsgraphen (DDG) der Befehle des Basisblocks.
   b) Erstelle eine Data Ready List (DRL). Enthalten sind alle Befehle, die im DDG keinen Vorgänger haben, und diejenigen, die nur antiabhängig von diesen Elementen sind. Jeder Befehl besitzt eine Priorität, die der Anzahl seiner Nachfolger im DDG entspricht. Die DRL wird nach der Priorität ihrer Elemenete sortiert.
   c) Erstelle neue leere Instruktion.
   d) Durchlaufe DRL. Für jeden Befehl B:
      - Falls B keinen Vorgänger im DDG hat und er beim Einfügen in die Instruktion keine Architekturrestriktionen des Gepard verletzt, füge ihn ein, lösche ihn dann aus DDG und DRL. Starte den Durchlauf durch die DRL wieder beim Element höchster Priorität.
      - Sonst, fahre mit dem Element nächstniedriger Priorität fort.

e) Nach vollständigem Durchlauf der DRL ist die Instruktion mit mindestens einem Befehl ausgefüllt.

f) Erstelle mit den verbliebenen Befehlen wieder eine DRL und wiederhole ab Punkt (c). Falls keine Befehle mehr vorhanden sind, ist die Kompaktierung dieses Basisblocks abgeschlossen.

3.4.3 Ergebnisse und Bewertung

Um einen Eindruck von der Güte der Kompaktierung zu erhalten, soll nun der sequentielle Code dem resultierenden, kompaktierten Assemblercode nach der Anwendung von GEPPAC gegenübergestellt werden.

Beim Eingangscode handelt es sich um einen kompletten Basisblock des Gepard-Assemblercodes, wie ihn das Gepard-Backend GEPCC für den DSPStone-Benchmark `lms` generiert hat (s. Abbildung 3.8 auf der nächsten Seite). Zusätzlich wurde eine beispielhafte Partitionierung vorgenommen. Dies ist am Auftreten von Zugriffen auf den Y-Speicher (`ldy`/`sty`) zu sehen.

Die Kompaktierung darf nun auf keinen Fall Datenabhängigkeiten zwischen den einzelnen Befehlen verletzen. Um diese Bedingung einzuhalten, werden nur die Befehle in die DRL eingefügt, die selbst Data Ready sind bzw. nur antiabhängig sind von einem Befehl, der sich bereits in der DRL befindet (s. 3(b) in Abschnitt 3.4.2 auf der vorherigen Seite). Letztere dürfen allerdings trotzdem erst dann in eine Instruktion eingefügt werden, wenn diese Abhängigkeit weggefallen ist (s. 3(d) in Abschnitt 3.4.2 auf der vorherigen Seite), der Befehl, von dem sie antiabhängig sind, also schon kompaktiert wurde. Dadurch ist es möglich, daß einer von zwei Befehlen, die in einem Takt abgearbeitet werden, ein bestimmtes Register liest und der andere, davon antiabhängige Befehl dieses Register schreibt. Die Architektur des Gepard unterstützt dies durch die dreistufige Pipeline mit den Stufen Fetch-Đecode-Execute.

Eine zweite, wesentliche Bedingung für die Korrektheit des erzeugten Assemblercodes ist natürlich auch die Übereinstimmung mit den Restriktionen, die die Architektur und damit auch der Befehlssatz selbst enthalten. Dabei handelt es sich um die Regeln, die festlegen, welche Befehle in einer Instruktion kombiniert werden dürfen und welche Anforderungen an deren Operanden zu stellen sind. Ein Beispiel ist, daß keine zwei Zugriffe auf den X-Speicher in einem Takt stattfinden können und daher alle Kombinationen von `ldx` (`ldy`) und `stx` (`sty`) in einer Instruktion verboten sind. Die Sicherstellung dieser Regeln übernimmt eine Funktion in Punkt 3(d).

Eine optimale Kompaktierung kann mit List-Scheduling durch die Abhängigkeit von der Heuristik zur Prioritätsberechnung nicht erwartet werden. Daß aber trotzdem sehr gute Ergebnisse erzielt werden, zeigt der von GEPPAC kompaktierte Assemblercode in Abbildung 3.9.

```
ldx     (i6),i4
ldc     #-7,i5
ldx     (i4)*,null
ldy     (i4),c
stx     c,(i7)
ldx     (i7),i1
ldx     (i1),a1
ldx     (i6),i0
ldc     #-14,i1
ldx     (i0)*,null
ldx     (i0),a2
stx     c,(i7)-1
stx     a2,(i7)
ldx     (i7)+1,c
mul     c,a1
mv      p1,a3;          ldx     (i7),c
ldx     (i6),i0
ldc     #-6,i1
ldx     (i0)*,null
ldy     (i0),d
stx     d,(i7)
ldx     (i7),i1
ldx     (i1),d
add     d,a3,a0
ldy     (i0),d
stx     d,(i7)
ldx     (i7),i1
stx     a0,(i1)
ldc     #1,c
add     d,c,a0
sty     a0,(i0)
ldy     (i4),d
add     d,c,a0
sty     a0,(i4)
ldx     (i6),i0
ldc     #-10,i1
ldx     (i0)*,null
ldx     (i0),d
add     d,c,a0
stx     a0,(i0)
j       _L_160
nop
```

Abbildung 3.8: Assemblercode vor der Kompaktierung

```
ldc     #-7,i5
ldx     (i6),i4
ldx     (i4)*,null
ldy     (i4),c
stx     c,(i7)
ldx     (i7),i1
ldx     (i1),a1
ldc     #-14,i1
ldx     (i6),i0
ldx     (i0)*,null
ldx     (i0),a2
stx     c,(i7)-1
stx     a2,(i7)
ldx     (i7)+1,c
ldx     (i6),i0;        mul     c,a1
ldc     #-6,i1
ldx     (i0)*,null;     mv      pl,a3
ldx     (i7),c;         ldy     (i0),d
stx     d,(i7)
ldx     (i7),i1
ldx     (i1),d
ldy     (i0),d;         add     d,a3,a0
stx     d,(i7)
ldx     (i7),i1
stx     a0,(i1)
ldc     #1,c
add     d,c,a0
ldx     (i6),i0;        sty     a0,(i0)
ldy     (i4),d
ldc     #-10,i1
ldx     (i0)*,null;     add     d,c,a0
ldx     (i0),d;         sty     a0,(i4);     add     d,c,a0
stx     a0,(i0)
j       _L_160
nop
```

Abbildung 3.9: Assemblercode nach der Kompaktierung

Benchmark	Instruktionsanzahl der Eingabedatei	Laufzeit in Sekunden
biquad_N_sections	473	0.79
convolution	256	0.07
dot_product	225	0.07
fir	332	0.12
fir2dim	950	0.27
lms	497	0.21
mat1x3	224	0.08
matrix1	572	0.14
matrix2	631	0.16
n_complex_updates	509	1.73
n_real_updates	346	0.15
real_update	156	0.06
receiver	6055	3.90

Tabelle 3.5: Laufzeiten des Kompaktierers GEPPAC

Ein Problem des Backends GEPCC, Adreßberechnungen immer in drei Schritten durch Laden eines Indexregisters, Laden des Offsets und Berechnen der Adresse durchzuführen, kann durch den Kompaktierer nicht aufgelöst werden. Dies ist in den ersten Zeilen des Codes zu sehen, der sich bis auf eine etwas andere Reihenfolge der Befehle nicht vom sequentiellen Code unterscheidet. Auch die fehlende Möglichkeit des Gepard, Werte aus den Akkumulatorregistern direkt in ein Indexregister zu laden, zeigt sich deutlich. Aufeinander folgende Anweisungen wie `stx d,(i7)` und `ldx (i7),i1` können aufgrund ihrer Datenabhängigkeit nicht zusammengefaßt werden und tauchen mehrfach auf.

Trotzdem konnte in diesem Codeabschnitt eine Reduktion der Codegröße von 42 auf 35 Instruktionen mit jeweils 32bit-Breite erreicht werden. Das entspricht einer Einsparung von 16.67% in diesem relativ berechnungsintensiven Abschnitt. In kleineren Basisblöcken und eher kontrollorientierten Codebereichen fällt die Einsparung geringer aus. Dies wird später im Abschnitt 5 auf Seite 73 deutlich, in dem dann auch die Resultate für die Kompaktierung vollständiger Assemblerdateien zu finden sind.

Die Laufzeiten des Kompaktierers sind in Tabelle 3.5 aufgelistet. Die Zeiten wurden auf einem Linux-System (Kernel 2.2.14, PentiumII-Prozessor mit 333 MHz, 64MB RAM) gemessen. Auch bei größeren Eingabedateien wie z.B. `receiver` dauerte die Kompaktierung nur 3.9 Sekunden und liegt damit auf jeden Fall im akzeptablen Bereich.

Die Komplexität des implementierten List-Schedulings bestimmt sich im wesentlichen aus den Operationen, die bei der Iteration über alle Basisblöcke in jedem Iterationsschritt ausgeführt werden müssen. Das Einlesen des Assemblercodes und dessen Aufteilung in Basisblöcke kann in linearer Zeit durchgeführt werden ($O(n), n =$ Anzahl der unkompaktierten Instruktionen).

Auch das Erstellen des DDG in den Iterationsschritten kann in linearer Zeit erfolgen. Die Komplexität der DRL-Generierung dagegen ist abhängig vom verwendeten Sortierverfahren. Im günstigsten Fall (Quicksort) beträgt sie $O(b * \log b)$ ($b =$ Anzahl der unkompaktierten Instruktionen des Basiblocks). Da die DRL maximal b-mal generiert werden muß, ergibt sich eine Komplexität von $O(b^2 * \log b)$. In der Praxis zeigt sich aber, daß die Laufzeit des Listenscheduling vielmehr von den späteren, oben in Punkt 3(c) beschriebenen Traversierungen der DRL abhängt. Es ergibt sich dort eine quadratische Komplexität in Abhängigkeit von der Länge des Basisblockes, die ja die Anzahl der Elemente der DRL am Anfang bestimmt ($O(b^2), b =$ Anzahl der unkompaktierten Instruktionen des Basiblocks).

In dem Fall, in dem das ganze Programm aus nur einem Basisblock besteht, ergibt sich somit eine Komplexität des Algorithmus von $O(n^2)$.

4 Die Gepard Speicherpartitionierung GEPMEM

4.1 Problembeschreibung

Die Aufgabe besteht darin, alle Variablen, die im Speicher gehalten werden sollen, optimal auf die beiden Speicherbänke des Gepard zu verteilen. Optimal bedeutet in diesem Zusammenhang, daß der resultierende Assemblercode optimal kompaktiert werden kann und so zum kürzesten möglichen Code führt. Dabei muß die Korrektheit des Codes natürlich gewahrt bleiben.

Es ergeben sich mehrere Forderungen, die ein System zur Speicherpartitionierung im optimalen Fall erfüllen sollte:

1. Allen im Speicher abgelegten Variablen sollen Speicherbänke derart zugewiesen werden, daß höchste Parallelität auf Instruktionsebene und damit größte Kompaktierung ermöglicht wird. Zusätzlich zur Minimierung der Codegröße soll auch die Ausführungsgeschwindigkeit des Codes verringert werden.

2. Alle Arten von Variablen (Funktionsparameter, lokale Variablen, globale Variablen, temporäre Variablen) sollen auf die Speicherbänke verteilbar sein, um größtmögliche Freiheit bei der Zuweisung zu haben.

3. Die entwickelte Strategie zur Partitionierung soll retargierbar sein und nicht nur für den Gepard Prozessor eine zufriedenstellende Lösung liefern.

Als Systemumgebung zur Lösung des Problems stehen das Compiler-Frontend LANCE (V1) (Kapitel 3.2 auf Seite 29) und das Backend GEPCC (Kapitel 3.3 auf Seite 38) als Sourcecode zur Verfügung und können entsprechend angepasst werden. Um die Qualität des partitionierten Assemblercodes bewerten zu können, wurde der Kompaktierer GEPPAC (Kapitel 3.4 auf Seite 46) implementiert.

Die oben genannten Forderungen müssen als Maximal-Forderungen betrachtet werden. Eine realistische Implementierung sollte natürlich versuchen, so viele wie möglich davon zu erfüllen. Falls dies nicht möglich ist, müssen die einzelnen Punkte daraufhin untersucht werden, welche Einschränkung der Forderungen eine Realisierung ermöglichen und trotzdem zu einem zufriedenstellenden Ergebnis führen. Die verschiedenen Forderungen werden diesbezüglich im folgenden genauer betrachtet:

1. Die erste Forderung spiegelt praktisch die Zielfunktion der Partitionierung wieder. In dem Umfang, in dem diese Forderung umgesetzt werden kann, verbessert sich auch die Zielfunktion. Die Entscheidung, in welcher Speicherbank die Allokation einer Variablen vorzunehmen ist, soll vom Partitionierungs-Algorithmus vorgenommen werden. Die Qualität der Entscheidung kann leider nicht direkt (online) während der Partitionierung ermittelt werden, sondern erst indirekt (offline) nach der Codegenerierung durch anschließende Kompaktierung und Bestimmung der Zielgrößen (Codegröße und Ausführungszeit).

   Diese Offline-Maße können als Zielfunktion daher nur bei iterativen Lösungsverfahren verwendet werden. Dies bedeutet, daß nach dem Bestimmen einer Lösung, d.h. einer Zuordnung der Variablen zu jeweils einer der beiden Speicherbänke, immer erst der Assemblercode generiert werden muß, um dann nach dessen Kompaktierung mit dessen Länge den Zielfunktionswert zu erhalten. Um diesen Wert zu optimieren, müssen im exakten Fall alle Variablen-Speicherbank-Zuordnungen diesem Verfahren unterzogen werden. Der Zeit- und Rechenaufwand wäre nicht vertretbar. Daraus folgt, daß, wenn nicht ein iteratives Lösungsverfahren angewendet werden soll, ein neues Online-Bewertungsmaß gefunden werden muß, das schon während der Partitionierung die Qualität einer Lösung hinreichend genau angeben kann.

   Wenn das Ergebnis der Optimierung dieses neuen Online-Maßes mit dem der Optimierung der Offline-Maße, der Länge des Assemblercodes nach Kompaktierung und dessen Ausführungsdauer, vergleichbar ist, kann es als Ersatz-Zielfunktion benutzt werden.

2. Aus der zweiten oben gestellten Forderung folgt die Frage, ob bestimmte Variablen eventuell gar nicht oder nur unter bestimmten Bedingungen beliebig in einer der Speicherbänke allokiert werden können. Gesucht ist also eine Klassifikation aller Variablen, die die Einschränkungen bei ihrer Partitionierung beschreibt.

   Folgende Klassifizierungen bieten sich dafür an:

   Zeigertypen oder Nichtzeigertypen Zeigertypen greifen im Gegensatz zu Nichtzeigertypen indirekt auf den Speicher zu. Im Verlauf eines Programms kann sich die referenzierte Speicheradresse ändern. Dies ist bei Nichtzeigertypen nicht möglich.

   Gültigkeitsbereich der Variablen Hier werden globale und lokale Variablen unterschieden. Zu den lokalen Variablen werden auch die an eine Funktion übergebenen Parameter gerechnet, zu den globalen auch die lokalen, statischen Variablen. Die Speicheradresse globaler Variablen muß in allen Funktionen bekannt sein, die lokaler dagegen nicht.

   Aus der Kombination dieser Merkmale ergeben sich vier verschiedene Variablen-Klassen, die jeweils unterschiedliche Anforderungen an die Partitionierung stellen.

   lokale Variablen mit Nichtzeigertyp Diese Klasse ist bezüglich der Verteilbarkeit ihrer Variablen unproblematisch, da die Speicheradresse und die Speicherbank der

```
main()
{
     int a, b;
     int *c;

     a = 5;
     b = 10;
     c = &a;
}
```

Abbildung 4.1: Einfaches Beispiel einer Aliasbeziehung zwischen `a` und `c`

Variablen nach der Allokation bekannt sind und sich im Verlauf des Programms auch nicht mehr ändern können. Der lokale Gültigkeitsbereich der Variablen kommt der Analyse des Zwischencodes entgegen, weil auch diese nur lokal, d.h. funktionsweise arbeitet. Bei der Zwischencodeanalyse sind daher alle Benutzungen einer lokalen Variablen mit Nichtzeigertyp sichtbar. Eine optimale Speicherbankzuweisung zu einer Variablen dieser Klasse innerhalb der betrachteten Funktion ist also auch insgesamt gesehen für diese Variable optimal.

globale Variablen mit Nichtzeigertyp Diese Klasse ist bezüglich der Verteilbarkeit auf die Speicherbänke genauso unproblematisch wie die mit lokalen Variablen, da sich die Speicherorte dieser Variablen während der Ausführung des Programms auch nicht ändern können. Wenn bei der Analyse einer einzelnen Funktion jedoch festgestellt wird, daß die Allokation der Variablen in einer bestimmten Speicherbank für diese Funktion eine optimale Partionierung liefert, kann in einer anderen Funktion die Allokation in einer anderen Speicherbank optimal sein. Es ist daher notwendig, eine Bewertung der sich eventuell widersprechenden Ergebnisse der funktionsweisen Analysen einzuführen, um am Ende die global optimale Speicherbank einer Variabalen zuweisen zu können.

lokale und globale Variablen mit Zeigertyp Ein generelles Problem der Zeigertypen ist die Unsicherheit zu entscheiden, auf welchen Speicherort und damit welche Speicherbank der Zeiger zu einem bestimmten Zeitpunkt des Programmablaufs verweist. Es tritt in besonderem Maße bei Programmiersprachen wie C auf, die eine sehr freie Zeigerarithmetik und damit u.a. auch mehrfach indirekte Adressierungen erlauben.

Durch eine vollständige *Aliasanalyse* könnte das Problem entschärft werden. Ein Alias tritt auf, wenn während des Programms zwei verschiedene Namen den gleichen Speicherort bezeichnen (Abbildung 4.1). Wenn alle Aliase bekannt sind, kann immer festgestellt werden, auf welche Speicherstelle bzw. welche andere Variable ein Zeiger verweist. Damit wäre dann auch bekannt, auf welche Speicherbank der Zugriff bei Dereferenzierung erfolgen muß.

Ohne Einschränkung der erlaubten Zeigerarithmetik ist die Alias-Bestimmung allerdings NP-hart [Lan92].

3. Um eine gute Retargierbarkeit der Partitionierung zu erreichen ist es sinnvoll, diese so früh wie möglich in den Übersetzungsprozess einzubinden. Kann sie im Extremfall schon im Frontend durchgeführt werden, könnten alle auf LANCE beruhenden Backends darauf zurückgreifen. Es wäre eine hohe Allgemeingültigkeit erreicht.

   Im Gegensatz dazu wäre eine Lösung, die erst im Backend die Speicherbankzuweisung vornimmt, erst einmal auf den Gepard-Prozessor beschränkt. Eine Anpassung an andere Zielsysteme ist prinzipiell möglich, aber immer mit höherem Aufwand verbunden. Ganz ins Frontend verlagern lässt sich die Partitionierung jedoch nicht, da zum Zeitpunkt der Zwischencode-Generierung noch keine Aussage darüber getroffen werden kann, ob ein bestimmter Variablenzugriff überhaupt zu einem Speicherzugriff im Assemblercode führt oder nicht. Die genaue Kenntnis dieser Speicherzugriffe ist nach der Codegenerierung auf Basis des Assemblercodes natürlich vorhanden. Die Entscheidung aber, ob zwei Speicherzugriffe auf dieselbe Speicherstelle zugreifen, also die gleiche Variable referenzieren und somit eine Datenabhängigkeit zwischen den Zugriffen besteht, ist dann nur noch sehr schwer möglich ([Bra98][Lan92]). Dies ist aber unbedingt nötig, da ein Verlagern der Variablen in die andere Speicherbank zur Folge hat, daß alle Speicherzugriffe, die diese Variable referenzieren, geändert werden müssen.

   Der günstigste Zeitpunkt zur Partitionierung ist also der, zu dem feststeht, an welcher Stelle im Assemblercode Speicherzugriffe auftreten und eine Zuordnung zwischen Speicherzugriff und Symbol noch möglich ist. Dieser Zeitpunkt liegt in der Codegenerierungsphase.

Als Konsequenz aus dem zuletzt genannten Punkt folgt die Entscheidung, zwei Läufe des Backends durchzuführen. Im ersten Durchlauf sollen alle Speicherzugriffe, die im Assemblercode auftreten, in der Symboltabelle bei den den Zugriff verursachenden Symbolen vermerkt werden. Nachdem dann eine Aufteilung der Symbole auf die Speicherbänke durch die Partitionierung bestimmt und diese Information in der Symboltabelle gespeichert wurde, verwertet das Backend im zweiten Lauf diese Partitionierungsinformation, um die die Allokation der Daten und die Speicherzugriffe entsprechend durchzuführen.

Der Ablauf der Speicherpartitionierung für eine (beispielhafte) Quelldatei `test.c` (Abbildung 4.2 auf der nächsten Seite) ist dann folgendermaßen (s.a. Abbildung 4.3 auf Seite 58):

1. `analyze test.c`

   Erzeugen des Zwischencodes

2. `optscript test.c`

   Optimierung des Zwischencodes

```
int main()
{
   int a, b, c, d, e;

   a = 1;
   b = 2;
   c = 3;
   d = a + b;
   e = d + c;

   return e;
}
```

Abbildung 4.2: `test.c`

3. `gepcc -l -o out1.asm test.c`

   Protokollierung der Speicherzugriffe

4. `gepmem test.c`

   Partitionierung

5. `gepcc -l -o out2.asm test.c`

   Codegenerierung mit Auswertung der Partitionierungsinformation

6. `geppac out2.asm`

   Kompaktierung

Im folgenden werden die einzelnen Schritte genauer erläutert.

4.2 Erzeugen und Optimieren des Zwischencodes

Das LANCE ANSI-C Frontend übersetzt die Quelldatei. Dabei werden die beiden Dateien `test.c.ir` für den generierten Zwischencode und die Datei `test.c.st` für die Symboltabelle erzeugt. Durch Auruf des Shell-Skriptes `optscript` werden die verschiedenen Optimierungen des LANCE-Systems auf den Zwischencode angewendet.

In den folgenden Erläuterungen wird in Beispielen oft der in Abbildung 4.4 auf Seite 59 dargestellte Zwischencode weiter verwendet. Im Gegensatz zum oben beschriebenen Verfahren ist er allerdings nicht durch Anwendung von `optscript` entstanden, sondern es wurden aus Gründen der Anschaulichkeit nur die Optimierungen *Copy Propagation* und *Dead Code Elimination* verwendet.

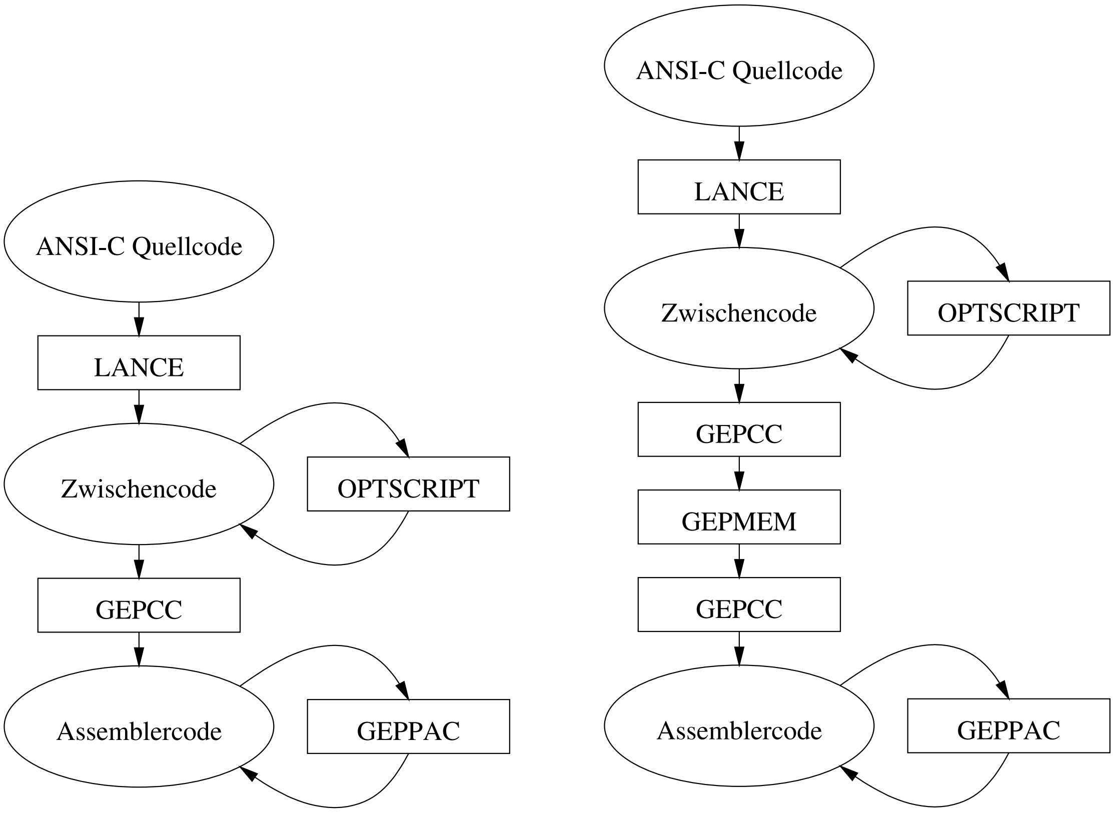


Abbildung 4.3: Der Übersetzungsprozeß ohne und mit Speicherpartitionierung

4.3 Protokollierung der Speicherzugriffe

Das Gepard-Backend generiert aus dem im ersten Schritt erzeugten Zwischencode Assemblercode und schreibt diesen in die Datei `out1.asm`. An dieser Stelle des Prozesses werden noch alle Speichervariablen im X-Speicher allokiert. Der erzeugte Assemblercode entspricht dem des ursprünglichen, unveränderten Backends und wird später nur zu Vergleichszwecken verwendet. Wichtig in diesem ersten Backend-Durchlauf ist allein die Protokollierung aller Speicherzugriffe. Sie werden bei den Symbolen, auf die der Zugriff stattfindet, in der Symboltabelle jeweils mit der Anzahl der Zugriffe vermerkt. Dafür wurden die Symbole von LANCE um die Eigenschaft `MEM_CNT` erweitert (Kapitel 3.2.3).

Außerdem wird der Typ jedes Symbols in der neuen Symboleigenschaft `MEM_LOC` gespeichert. Als Typen werden unterschieden:

- GLOBAL

```
function 'main'(14):

@5@ 'a'(15) = INT(1) ;
@6@ 'b'(16) = INT(2) ;
@7@ 'c'(17) = INT(3) ;
@8@ '_t_19'(21) = 'a'(15) + 'b'(16) ;
@9@ '_t_27'(29) = '_t_19'(21) + 'c'(17) ;
@11@ returnval '_t_27'(29) ;
@0@ return ;
```

Abbildung 4.4: `test.c.ir`

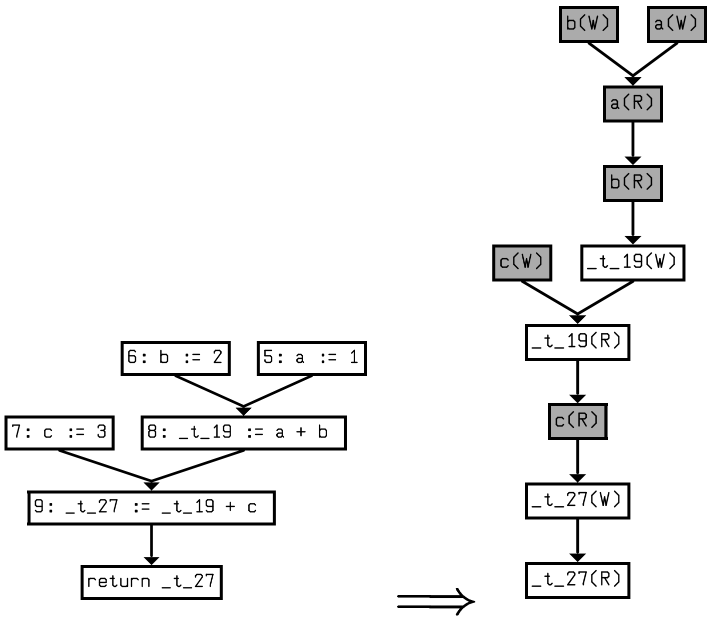
- LOCAL
- SPILL
- PARAM
- NONE

Diese Einteilung richtet sich nach dem Backend, das bei der Speicherorganisation gerade diese ersten vier der fünf Typen unterschiedlich behandelt. So werden die globalen Variablen (GLOBAL) an festen Speicheradressen abgelegt, die lokalen Variablen (LOCAL) und die Funktionsparameter (PARAM) erhalten bestimmte Bereiche innerhalb eines Stackframes und die Zwischenergebnisse von Berechnungen werden auf dem Stack abgelegt (SPILL). Der Typ NONE zeigt an, daß auf dieses Symbol kein Speicherzugriff erfolgt und ist die Standardeinstellung. Anhand dieser Einteilung werden im nächsten Schritt die partitionierbaren Symbole ausgewählt.

4.4 Partitionierung

Der Partitionierungsalgorithmus verwendet als Eingabe den Zwischencode aus der Datei `test.c.ir` und die Symboltabelle aus `test.c.st` mit den im vorigen Schritt hinzugefügten Informationen. Auf Basis der Datenabhängigkeiten im Zwischencode wird funktionsweise jeweils ein Datenabhängigkeitsgraph erstellt (Abbildung 4.5(a) auf der nächsten Seite). Dieser stellt anfangs noch die Abhängigkeiten zwischen den einzelnen Zwischencodeanweisungen dar. In einem zweiten Schritt werden diese Anweisungen zerlegt und in die enthaltenen Symbole aufgeteilt. Die Knoten des Graphen repräsentieren nun die Zugriffe auf die einzelnen Symbole. Unterschieden wird dabei noch zwischen Lese- und Schreibzugriff auf ein Symbol. Symbole, die zu einem Speicherzugriff führen, sind grau markiert (Abbildung 4.5(b) auf der nächsten Seite).

Es ist leicht zu sehen, daß der parallele Zugriff auf zwei verschiedene Symbole unter Beachtung der Datenabhängigkeiten nur dann möglich ist, wenn die Symbole nicht auf einem Pfad liegen.



(a) Datenabhängigkeitsgraph der Zwischencodeanweisungen

(b) Datenabhängigkeitsgraph der Zwischencodesymbole

Abbildung 4.5: Übergang zwischen den Datenabhängigkeitsgraphen

Mit diesem Wissen wird ein Interferenzgraph erzeugt, der die Knoten des Datenabhängigkeitsgraphen enthält. Ungerichtete Kanten werden dann zwischen zwei Knoten eingefügt, wenn die von ihnen repräsentierten Symbole nicht auf einem Pfad im Datenabhängigkeitsgraphen liegen, also parallelisierbar sind. Als Kantengewicht wird die Summe der Anzahl der Speicherzugriffe auf die beiden Symbole gewählt. Diese Anzahlen hatte das Backend in seinem ersten Durchlauf in die Symboltabelle geschrieben. Dadurch ist eine höhere Gewichtung für die Symbole gewährleistet, die zu vielen Speicherzugriffen führen, deren Parallelisierung also einen hohen Gewinn verspricht.

An dieser Stelle tritt folgendes Problem auf. Dadurch daß alle Indexzugriffe auf Arrays von LANCE durch Zeigerzugriffe ersetzt werden, sind bei den Arraysymbolen der Symboltabelle vom Backend keine Speicherzugriffe vermerkt worden. Die eigentlich auf Arrays stattfindenden Zugriffe sind bei den entsprechenden Zeigersymbolen summiert worden. Um die korrekten Zugriffswerte nun als Kantengewichte verwenden zu können, müssen die Speicherzugriffe über Zeiger wieder auf Arrayzugriffe abgebildet werden.

Dazu wurde ein einfaches Verfahren implementiert, daß eine bestimmte Eigenschaft des Zwischencodes ausnutzt. Alle Arrayzugriffe des C-Quellcodes werden von LANCE nämlich nach folgendem Muster übersetzt:

1. Zuweisung der Basisadresse des Arrays an ein temporäres Symbol
2. Veränderung der Basisadresse durch z.B. Addition eines Zahlenwertes oder auch Multiplikation mit einem Zahlenwert. Dieser neue Wert wird in einem neuen temporären Symbol gespeichert.
3. Zugriff auf den Speicher durch Dereferenzierung dieses Symbols

Durch Verfolgung dieser Zuweisungsmuster im Zwischencode können zumindest die Arrayzugriffe rekonstruiert werden, die im Quellcode über Indizes realisiert wurden. Auch der Arrayzugriff über Zeiger kann ermittelt werden, wenn er im Zwischencode das gleiche Zuweisungsmuster erzeugt. Genutzt wurde für diese Analyse die neu im Frontend implementierte Symboleigenschaft `POINTS_TO`. Die Genauigkeit der in der Symboltabelle gespeicherten Speicherzugriffe konnte mit diesem Verfahren stark verbessert werden.

Da das Kantengewicht jeweils den Vorteil ausdrücken soll, der durch den gleichzeitigen Zugriff auf die Symbole entsteht, und dieser Vorteil natürlich abnimmt, je weiter die Speicherzugriffe auf die Symbole im Code auseinanderliegen, wird eine zusätzliche Gewichtung eingeführt. Sie bewirkt, daß das durch die Summe der Speicherzugriffe bestimmte Gewicht mit zunehmendem Abstand der Speicherzugriffe abnimmt. Der Abstand der Speicherzugriffe wird durch den

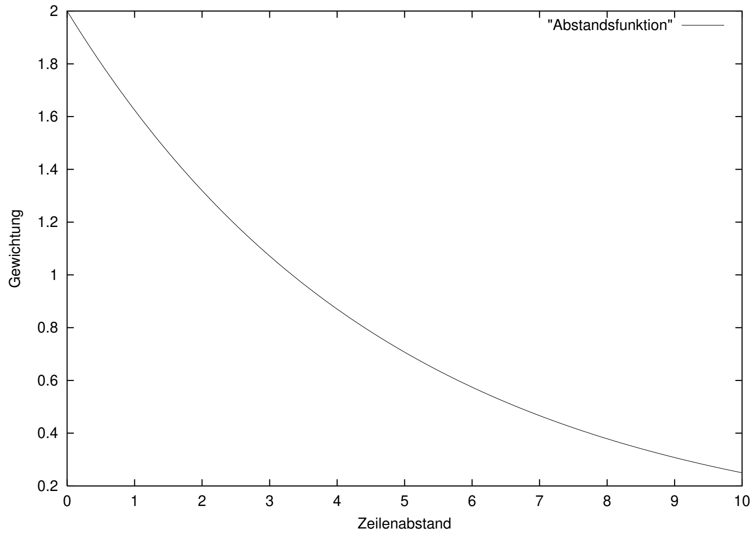


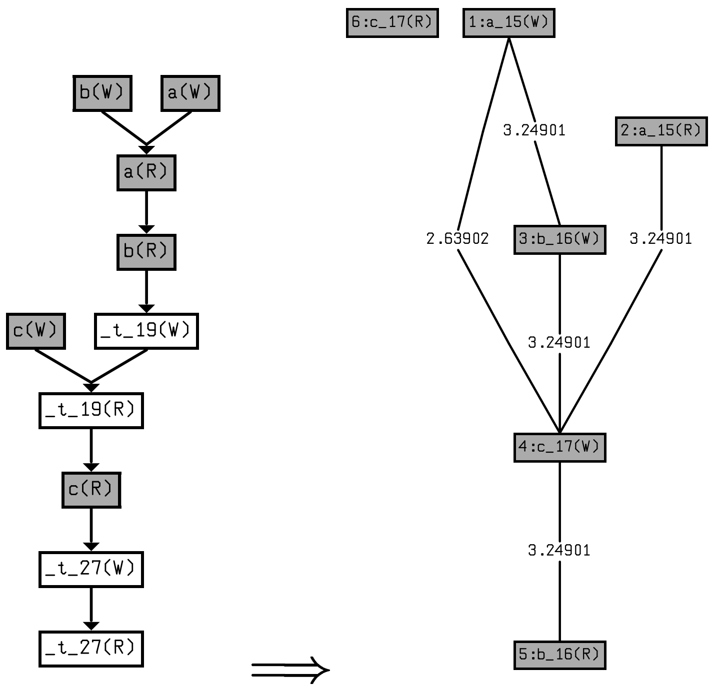
Abbildung 4.6: Funktion zur Gewichtung des Zeilenabstands zwischen Speicherzugriffen ($f(x) = 2^{(1-0.3x)}$)

Zeilenabstand der Symbole im Zwischencode bestimmt. Die Gewichtungsfunktion ist in Abbildung 4.6 zu sehen.

Falls also jeweils zwei parallelisierbare Speicherzugriffe durch zwei Symbole einer Zwischencode-Anweisung hervorgerufen werden, verdoppelt sich das Gewicht der Kante, die zwischen den entsprechenden Knoten des Interferenzgraphen verläuft. Wenn aber die Zwischencodezeilen einen Abstand von z.B. sechs Zeilen besitzen, halbiert sich das ursprüngliche Gewicht der Kante.

Die Transformation eines Datenabhängigkeitsgraphen in einen Interferenzgraphen wird in Abbildung 4.7 auf der nächsten Seite gezeigt.

Eine optimale Partitionierung teilt die Menge der Knoten in zwei Teilmengen, so daß die Summe der Kantengewichte der zwischen diesen Teilmengen verlaufenden Kanten maximal wird. Dabei müssen Knoten, die dasselbe Symbol referenzieren, auch derselben Teilmenge zugewiesen werden. Die Zuweisung eines Knotens zu einer Teilmenge entspricht dann der Zuweisung des entsprechenden Symbols zu einer Speicherbank. Durch Maximierung dieser Summe der Kantengewichte wird erreicht, daß eine mögliche Parallelität zwischen zwei Symbolen, auf die oft im Speicher zugegriffen wird und die im Zwischencode nah zusammen liegen, eher ausgenutzt wird als eine zwischen Symbolen, die zu weniger Speicherzugriffen führen bzw. weiter auseinander liegen (Abbildung 4.8 auf der nächsten Seite). Die Summe der Kantengewichte übernimmt also in diesem Verfahren die Funktion des Offline-Maßes, wie es weiter oben vorgestellt wurde.



(a) Datenabhängig-keitsgraph

(b) Interferenzgraph

Abbildung 4.7: Transformation des Datenabhängigkeitsgraphen in einen Interferenzgraphen.

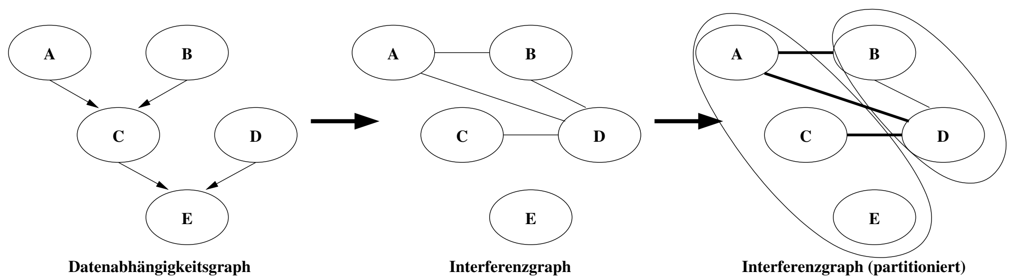


Abbildung 4.8: Schema der Partitionierung

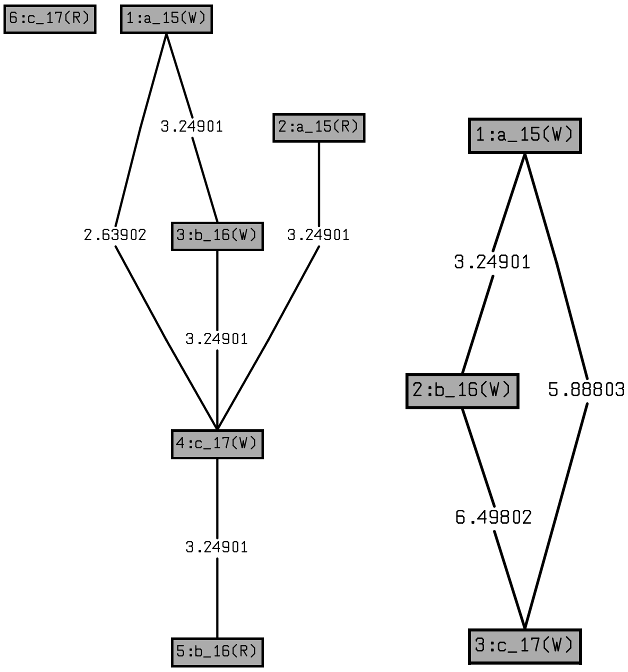
Eine wichtige Nebenbedingung an dieser Stelle ist, daß Knoten, die das gleiche Symbol repräsentieren, nach der Partitionierung auch in der gleichen Knotenmenge enthalten sein müssen. Diese Einschränkung folgt aus der Forderung, daß ein Symbol nur in einer der beiden Speicherbank allokiert werden darf. Wenn auf diese Einschränkung verzichtet wird, könnte es vorkommen, daß die Partitionierung ein Symbol beiden Speicherbänken zuweist. Dies ähnelt grob dem Prinzip des Partial-Data-Duplication von Saghir et.al. (Kapitel 2.3 auf Seite 16). Im folgenden soll aber von dieser Möglichkeit kein Gebrauch gemacht werden, so daß Symbole nur in einer Speicherbank allokiert werden dürfen.

Durch Beibehaltung dieser Einschränkung der Symbolallokation auf eine Speicherbank erhält man die Möglichkeit, den Interferenzgraphen zu vereinfachen. Durch *Zusammenfalten* des Graphen wird die Anzahl der Knoten und vor allem die der Kanten stark vermindert und damit auch die Komplexität des Partitionierungsproblems verringert. Dabei werden alle Knoten, die das gleiche Symbol repräsentieren, auf einen Knoten abgebildet. Alle Kanten, die vorher von diesen Knoten abgingen, gehen nun von diesem einen ab. Falls es im Graphen jetzt vorkommt, daß jeweils zwei Knoten durch mehr als eine Kante verbunden sind, werden auch diese Kanten zu einer zusammengefaßt. Das Gewicht der neuen Kante berechnet sich aus der Addition der Gewichte der Ursprungskanten (Abbildung 4.9 auf der nächsten Seite). Durch diese Faltung des Graphen konnte im Extremfall die Anzahl der Knoten von 259 auf 70 und gleichzeitig die Anzahl der Kanten von 29871 auf 2178 verringert werden (Funktion `decod_fcb` aus `receiver.c`, vgl. Kapitel 5 auf Seite 73).

Diese oben dargestellte Sicht des Partitionierungsproblems als Problem der Maximierung der Kantengewichte zwischen zwei Knotenmengen entspricht dem Problem des maximalen Schnittes (MAX CUT). Dieses ist für nicht planare Graphen NP-vollständig (Problem ND16 in [GJ79]). Um auch größere Probleme zeiteffizient lösen zu können, besteht nun die Möglichkeit, Heuristiken zu entwickeln und auf das Partitionierungsproblem anzuwenden. Man verliert dabei allerdings die Garantie auf eine optimale Lösung.

Ein anderer Weg ist die Formulierung des Partitionierungsproblems als *Lineares Programm (LP)*. Gerade im Bereich der Codegenerierung für DSPs wird die *Lineare Programmierung* vielfach als Hilfsmittel zur optimalen Lösung komplexer, NP-harter Probleme erfolgreich verwendet. Beispiele für die Anwendung sind nicht nur bei Teilproblemen der Codegenerierung wie der optimalen Instruktionsanordnung [Leu97] zu finden, sondern vor allem auch dort, wo durch Kopplung der drei Übersetzungsphasen, Instruktionsauswahl, Registerallokation und Instruktionsanordnung, die Komplexität des zu lösenden Problems enorm ansteigt [WGHB95].

Ein LP ist die Formulierung eines numerischen Optimierungsproblems. Es besteht aus einer zu maximierenden oder minimierenden Zielfunktion und mehreren Gleichungen oder Ungleichungen als Nebenbedingungen, die das Problem näher beschreiben. Zielfunktion und alle Nebenbedingungen müssen linear sein. Abhängig von der Art der Variablen teilt man LPs weiter in ILPs (*Integer Linear Program*) und MILPs (*Mixed Integer Linear Program*) ein. Bei ILPs sind die



(a) Interferenzgraph

(b) Interferenzgraph nach Faltung

Abbildung 4.9: Der Übergang zwischen den Interferenzgraphen

Variablen ganzzahlig, bei MILPs können einige oder auch alle Variablen reellwertig sein. Die Zielfunktion hat in beiden Fällen kontinuierlichen Charakter.

Definition: Als Lineares Programm wird folgendes Problem bezeichnet (nach [GJ79]):

$$\begin{aligned}
\text{Gegeben sind: } & V_i = (v_i[1], v_i[2], \ldots, v_i[n]), 1 \leq i \leq m \\
& D = (d_1, d_2, \ldots, d_m) \\
& C = (c_1, c_2, \ldots, c_n) \\
& \text{mit } v_i[j], d_i, c_j, B \in \mathbb{Z}, 1 \leq i \leq m, 1 \leq j \leq n
\end{aligned}$$

$$\begin{aligned}
\text{Existiert ein } & X = (x_1, x_2, \ldots, x_n), x_j \in \mathbb{Q}, 1 \leq j \leq n \\
\text{so daß: } & V_i \cdot X \leq d_i, 1 \leq i \leq m \\
& C \cdot X \geq B
\end{aligned}$$

Zur Berechnung der Lösung eines LPs kann auf verschiedene Programmpakete (*LP-Solver*), die für MILP- und zum Teil auch ILP-Modelle verfügbar sind, zurückgegriffen werden. Dabei handelt es sich um kommerzielle Varianten wie CPLEX oder das frei verfügbare lp_solve.

Mehrere Argumente machen nun den Einsatz eines LPs zur Lösung von Optimierungsproblemen sinnvoll:

1. Eine gefundene Lösung ist im Sinne des formulierten Modells garantiert optimal.
2. Ein LP ist ein präzises mathematisches Modell des Optimierungsproblems, das weiter verwendet werden kann.
3. Die Erweiterung eines LPs um zusätzliche Bedingungen ist leicht möglich.
4. Es existieren leistungsfähige Programmpakete zur Lösung von LPs.

Diesen Vorteilen muß allerdings gegenübergestellt werden, daß allein die Transformation eines Optimierungsproblems in ein LP das Problem selbst nicht weniger komplex werden läßt. Die Lineare Programmierung ohne Einschränkungen ist von der Komplexität her in NP [GJ79]. Varianten können jedoch in polynomieller Zeit gelöst werden. Auch hat sich gezeigt, daß die Laufzeiten der LP-Solver bei nicht zu großen praktischen Problemen in einem erträglichen Rahmen liegen. Da dies neben der Problemgöße unter anderem auch von der gewählten Formulierung des LP-Modells abhängt, kommt der Transformation des Ursprungsproblems in das zu lösende LP entscheidende Bedeutung zu. Außerdem spiegelt die optimale Lösung des LPs erst dann auch die optimale Lösung des Ursprungsproblems wieder, wenn die Transformation vollständig und korrekt beschrieben ist.

Aufgrund der überschaubaren Problembeschreibung der Partitionierung als Problem des maximalen Schnittes sollte die Transformation in ein LP keine großen Schwierigkeiten bereiten. Auch hat die Komplexität des zu lösenden Problems durch die Faltung des Interferenzgraphen stark abgenommen, so daß die Laufzeiten eines LP-Solvers zur Lösung des LP-Modells nicht zu hoch sein werden. Aus diesen Gründen wird nun ein LP-Modell zur Lösung des Partitionierungsproblems verwendet. Als LP-Solver kommt lp_solve zum Einsatz.

Zur Formulierung des LPs werden alle Knoten des Interferenzgraphen mit 1 beginnend fortlaufend nummeriert. Die Anzahl der Knoten des Interferenzgraphen sei n. Durch die Faltung des Graphen repräsentiert jeder Knoten ein anderes Symbol. Aus der Problemstellung ergibt sich dann folgendes zu lösendes lineares Programm:

- **Definitionen**

$$\forall i \in [1,n] : V_i = \begin{cases} 1 & \text{, falls Knoten } i \text{ zur 1. Knotenmenge gehört} \\ 0 & \text{, falls Knoten } i \text{ zur 2. Knotenmenge gehört} \end{cases}$$

Die Belegung der Variablen V_i soll durch die Lösung des LPs bestimmt werden. Damit ist dann die Aufteilung der Symbole auf die Speicherbänke bestimmt. Da es sich um nicht kontinuierliche Variablen handelt, muß also ein ILP gelöst werden.

$$\forall i \in [1,n], j \in [1,n] : U_{ij} = \begin{cases} 1 & \text{, falls } V_i \neq V_j \\ 0 & \text{, sonst} \end{cases}$$

Die Variablen U_{ij} sind binäre Hilfsvariablen, die anzeigen, ob die Knoten V_i und V_j in der gleichen Knotenmenge sind oder nicht.

$$\begin{aligned} \forall i \in [1,n], j \in [1,n] : G_{ij} = {} & \text{Gewicht der Kante zwischen} \\ & \quad \text{Knoten } i \text{ und Knoten } j \text{ aus} \\ & \quad \text{dem Interferenzgraphen} \\ \forall i \in [1,n], j \in [1,n] : W_{ij} = {} & \text{realisiertes Gewicht der Kante} \\ & \quad \text{zwischen Knoten } i \text{ und Knoten } j \end{aligned}$$

Ein Kantengewicht gilt dann als realisiert, wenn die entsprechende Kante zwei Knoten unterschiedlicher Knotenmengen verbindet, also zwischen den Knotenmengen verläuft.

- **Zielfunktion**

Das Ziel ist die Maximierung der realisierten Kantengewichte, d.h. die Maximierung der Gewichte der Kanten, die zwischen den beiden Knotenmengen verlaufen.

$$\sum_{1 \leq i \leq n} \sum_{1 \leq j \leq n} W_{ij} \rightarrow max$$

- **Nebenbedingungen**

$$\forall i \in [1, n] : V_i \leq 1$$

Da lp_solve für den Datentyp `int` den Zahlenbereich der natürlichen Zahlen vorsieht, ist diese Nebenbedingung notwendig, um die Variablen V_i auf die Werte 0 und 1 beschränken.

$$\forall i \in [1, n], j \in [1, n] : U_{ij} \leq V_i + V_j$$
$$\forall i \in [1, n], j \in [1, n] : U_{ij} \leq 2 - V_i - V_j$$

Durch diese beiden Gleichungen werden die Hilfsvariablen U_{ij} auf die in der Definition genannten Werte beschränkt.

$$\forall i \in [1, n], j \in [1, n] : W_{ij} = G_{ij} * U_{ij}$$

Durch diese Nebenbedingung wird gewährleistet, daß nur die Gewichte der Kanten in die Berechnung der Zielfunktion eingehen, die zwischen den beiden Knotenmengen verlaufen.

Zusammenfassend wird durch die Gleichungen des ILP festgelegt:

1. Jeder Knoten liegt entweder in der ersten oder in der zweiten Knotenmenge.
2. Es werden nur die Gewichte der Kanten gezählt, die zwischen den Knotenmengen verlaufen.
3. Zu maximieren ist die Summe der Kantengewichte dieser realisierten Kanten.

Dies sind exakt die Eigenschaften des Partitionierungsproblems.

Im Gegensatz zu den meisten LP-Solvern, die das MPS-Format für die Eingabedatei fordern, benutzt lp_solve ein eigenes, leichter zu generierendes Format, dessen Einträge den Anweisungen einer Programmiersprache wie C ähneln.

Im Folgenden ist die Eingabe für lp_solve, die aus dem Interferenzgraphen in Abbildung 4.9(b) auf Seite 65 generiert wurde, aufgelistet. Zu beachten ist, daß die Hilfsvariablen U_{ij} im Modell für lp_solve nicht mehr vorkommen. Zur Verringerung der Nebenbedingungen und damit auch zur Steigerung der Berechnungsgeschwindigkeit wurden sie eingespart. Durch Umformung der letzten drei Nebenbedingungen entstehen somit die beiden neuen Bedingungen:

$$\forall i \in [1,n], j \in [1,n] : W_{ij} \leq G_{ij} * (V_i + V_j)$$
$$\forall i \in [1,n], j \in [1,n] : W_{ij} \leq G_{ij} * (2 - V_i - V_j)$$

Diese sind in der Eingabedatei nun auch leicht wiederzufinden.

```
/*
 * v15 <-> a_15 -> (2)
 * v16 <-> b_16 -> (2)
 * v17 <-> c_17 -> (2)
*/

max:  + w15_16 + w15_17 + w16_17;
v15 <= 1;
v16 <= 1;
v17 <= 1;
w15_16 <= 3.24901 * v15 + 3.24901 * v16;
w15_16 <= 6.49802 - 3.24901 * v15 - 3.24901 * v16;
w15_17 <= 5.88803 * v15 + 5.88803 * v17;
w15_17 <= 11.7761 - 5.88803 * v15 - 5.88803 * v17;
w16_17 <= 6.49802 * v16 + 6.49802 * v17;
w16_17 <= 12.996 - 6.49802 * v16 - 6.49802 * v17;
int v15;
int v16;
int v17;
```

Die ersten Zeilen sind Kommentare, die von lp_solve ignoriert werden. Sie dienen allein der Information, welches Symbol von welcher Variablen dargestellt wird. Variablen beginnen wie in den oben beschriebenen Gleichungen mit `v`, wenn es sich um Knotenvariablen handelt, und mit `w`, wenn sie die realisierten Gewichte darstellen. Die anschließende Ziffer bei Knotenvariablen entspricht dem Symbolschlüssel aus der Symboltabelle. Bei den Gewichtsvariablen sind die beiden Symbolschlüssel angegeben, die den durch die entsprechende Kante verbundenen Knoten zugeordnet sind.

Es folgt die Zielfunktion, die die Maximierung der Addition der drei Gewichte `w15_16`, `w15_17` und `w16_17` beschreibt. Daran anschließend sind die Nebenbedingungen aufgeführt. Zuletzt benötigt lp_solve noch die Typangabe der freien Variablen. Da lp_solve nur `int` für die natürlichen Zahlen und `real` für reellwertige Zahlen unterstützt, müssen die eigentlich binären Knotenvariablen als `int` deklariert werden. Die Einschränkung auf die binären Werte geschieht dann durch die Nebenbedingungen.

Für dieses kleine Beispiel berechnet lp_solve folgende Lösung:

```
Value of objective function: 12.3861
v15                     1
v16                     0.999994
v17                     3.39672e-06
w15_16                  2.1036e-05
w15_17                  5.88805
w16_17                  6.498
```

Zwei der normalerweise ganzzahligen Knotenvariablen erscheinen hier als Fließkommawert. Die Ursache liegt darin, daß lp_solve intern alle Variablen mit Fließkommatypen darstellt und so geringe Ungenauigkeiten in der Ausgabe erzeugt.

Das Ergebnis ist trotzdem eindeutig. Die Variablen `v15` und `v16` liegen in der zweiten Knotenmenge, die verbliebene Variable `v17` in der ersten. Dies kann nun auf die repräsentierten Symbole übertragen werden. Die Symbole `a_15` und `b_16` sollten folglich in der zweiten Speicherbank allokiert werden, das Symbol `c_17` dagegen in der ersten.

Diese Zuordnung der Speicherbänke zu den Symbolen wird im Anschluß der Partitionierung in die Symboltabelle eingetragen. In den weiteren Schritten wird diese Information dann ausgewertet.

Bei globalen und statischen Variablen muß die Entscheidung, in welcher Speicherbank allokiert werden soll, solange verzögert werden, bis alle Funktionen ausgewertet sind. Da es vorkommen kann, daß die Variable von verschiedenen Funktionen jeweils in einer anderen Bank gespeichert werden soll, muß eine Bewertung der unterschiedlichen Entscheidungen stattfinden. Ein einfaches Verfahren, das hier Anwendung finden soll, ist die Gewichtung der Speicherbankentscheidung mit der Anzahl der Zugriffe auf die Variable in dieser Funktion. Das folgende kleine Beispiel soll dies verdeutlichen.

Die nach der Partitionierung aller Funktionen vorhandenen Daten sind in Tabelle 4.1 auf der nächsten Seite dargestellt. In Funktion A finden hier fünf Zugriffe auf die globale Variable `i` statt und laut Partitionierung dieser Funktion sollte `i` im X-Speicher allokiert werden. Funktion

	globale Variable i		globale Variable j	
	Zugriffe	Entscheidung	Zugriffe	Entscheidung
Funktion A	5	X-Speicher	2	X-Speicher
Funktion B	2	Y-Speicher	3	X-Speicher

Tabelle 4.1: Beispieldaten zur Aufteilung der globalen Variablen

B sieht dagegen den Y-Speicher als idealen Speicherort an. Da in B allerdings weniger Zugriffe auf `i` stattfinden, wird `i` letztendlich im X-Speicher allokiert. Das Treffen einer Entscheidung für die Variable `j` ist nicht notwendig, da beide Funktionen die gleiche Partitionierungsentscheidung getroffen haben.

4.5 Codegenerierung mit Auswertung der Partitionierungsinformation

Im zweiten Durchlauf des Backends GEPCC werden anders als beim ersten Mal nicht mehr alle Variablen im X-Speicher, sondern den Ergebnissen der Partitionierung folgend auf beide Speicherbänke verteilt. In der resultierenden Assembler-Datei out2.asm werden bei den Zugriffen auf die Werte, die im Y-Speicher allokiert werden sollen, statt den `ldx`/`stx`-Befehlen die entsprechenden `ldy`/`sty`-Befehle verwendet.

4.6 Kompaktierung

Durch die Verwendung von Speicherzugriffen auf die Y-Speicherbank ergeben sich für die Kompaktierung der Assembler-Datei `out2.asm` größere Optimierungsfreiräume als bei der zu Vergleichszwecken durchgeführten Kompaktierung von `out1.asm`. Es lässt sich daher eine geringere Code-Größe und auch eine höhere Ausführungsgeschwindigkeit des kompaktierten Codes erwarten.

Eine genaue Beschreibung des Kompaktierers GEPPAC findet sich in Abschnitt 3.4 auf Seite 46. Die erzielten Ergebnisse werden in Kapitel 5 auf Seite 73 aufgeführt.

5 Experimente und Ergebnisse

Durch Übersetzung und Optimierung verschiedener Testprogramme wurde die gewählte Implementierung getestet. Die erzielten Ergebnisse sollen in diesem Kapitel vorgestellt werden.

Die Testprogramme stammen zum größten Teil aus dem DSPStone-Projekt [Ziv94]. Es handelt sich dabei um Kernelbenchmarks, die jeweils für DSP-Algorithmen typische Codesequenzen realisieren:

- biquad_N_sections — IIR Filter mit vier biquad-Abschnitten
- convolution — Convolution der Länge 16
- dot_product — Multiplikation von zwei zweidimensionalen Vektoren
- fir — 16-stufiger FIR Filter
- fir2dim — zweidimensionaler FIR Filter
- lms — LMS Filter
- mat1x3 — Multiplikation einer 3x3-Matrix mit einem 3x1-Vektor
- matrix1 — Multiplikation von zwei 10x10-Matrizen
- matrix2 — eine weitere Multiplikation von zwei 10x10-Matrizen
- n_complex_updates — 16 Berechnungen der Form $D = C + A * B$ (A, B, C, D sind dabei komplexe Zahlen)
- n_real_updates — 16 Berechnungen der Form $D = C + A * B$ (hier sind A, B, C, D reelle Zahlen)
- real_update — eine Berechnung der Form $D = C + A * B$ mit reellen Zahlen

Zusätzlich wurde die Optimierung der Datei `receiver` durchgeführt. Bei ihr handelt es sich um eine größere Anwendung aus einer *GSM Half-Rate Codec Simulation* [Nac93]. Sie wurde ausgewählt, um die Laufzeiten der Speicherpartitionierung auch bei größeren, realistischen

Benchmark	gepcc	gepcc + geppac		gepcc + gepmem + geppac	
biquad_N_sections	3165	2945	-6.95%	2909	-8.09%
convolution	1764	1648	-6.58%	1565	-11.28%
dot_product	293	283	-3.41%	267	-8.87%
fir	2726	2526	-7.34%	2378	-12.77%
fir2dim	18146	17041	-6.10%	16247	-10.47%
lms	3796	3576	-5.80%	3428	-9.69%
mat1x3	850	806	-5.18%	786	-7.53%
matrix1	106847	90807	-15.01%	88559	-17.12%
matrix2	89206	83467	-6.43%	81219	-8.95%
n_complex_updates	6598	6098	-7.58%	5753	-12.81%
n_real_updates	3542	3409	-3.75%	3182	-10.16%
real_update	146	143	-2.05%	140	-4.11%
receiver					

Tabelle 5.1: Anzahl der benötigten Taktzyklen zur Ausführung des erzeugten Assemblercodes

DSP-Anwendungen beurteilen zu können.

Der Testablauf orientiert sich am in Abbildung 5.1 auf der nächsten Seite skizzierten Schema.

Die Generierung des Zwischencodes und dessen Optimierung mit `optscript` wird in jedem Fall durchgeführt. Anschließend wird die Codegenerierung einmal ohne Speicherpartitionierung (A in Abbildung 5.1 auf der nächsten Seite) und einmal mit duchgeführt (B). In beiden Fällen wird der Assemblercode mit `geppac` kompaktiert, um den Grad der realisierten Parallelisierung bewerten zu können.

Ausführungszeit der Testprogramme

Ein wichtiges Maß für den Erfolg der Partitionierung ist die erreichte Laufzeitverringerung beim übersetzten Programms. Da der AMS Gepard jede Instruktion in einem Taktzyklus abschließt, kann die Laufzeit durch die Anzahl der Instruktionen angegeben werden, die vom Programmstart bis zum Programmende abgearbeitet werden müssen. In Tabelle 5.1 sind als Ausgangsbasis die Werte für den sequentiellen, nicht partitionierten Assemblercode angegeben, die das Backend `gepcc` ohne die Implementierungen dieser Arbeit erreicht. Dem gegenübergestellt sind die Resultate einmal der reinen Kompaktierung und der Kompaktierung mit vorheriger Speicherpartitionierung. Der prozentuale Gewinn ist jeweils in Relation zur Ausgangsbasis angegeben.

Da der Simulator für den AMS Gepard [Aus98a] nicht verfügbar war, mußte die zur Ausführung benötigte Taktzyklenzahl der Benchmarks durch Abzählen bestimmt werden. Aus diesem Grund sind in der Tabelle 5.1 keine Werte für den `receiver` Benchmark enthalten, da durch die

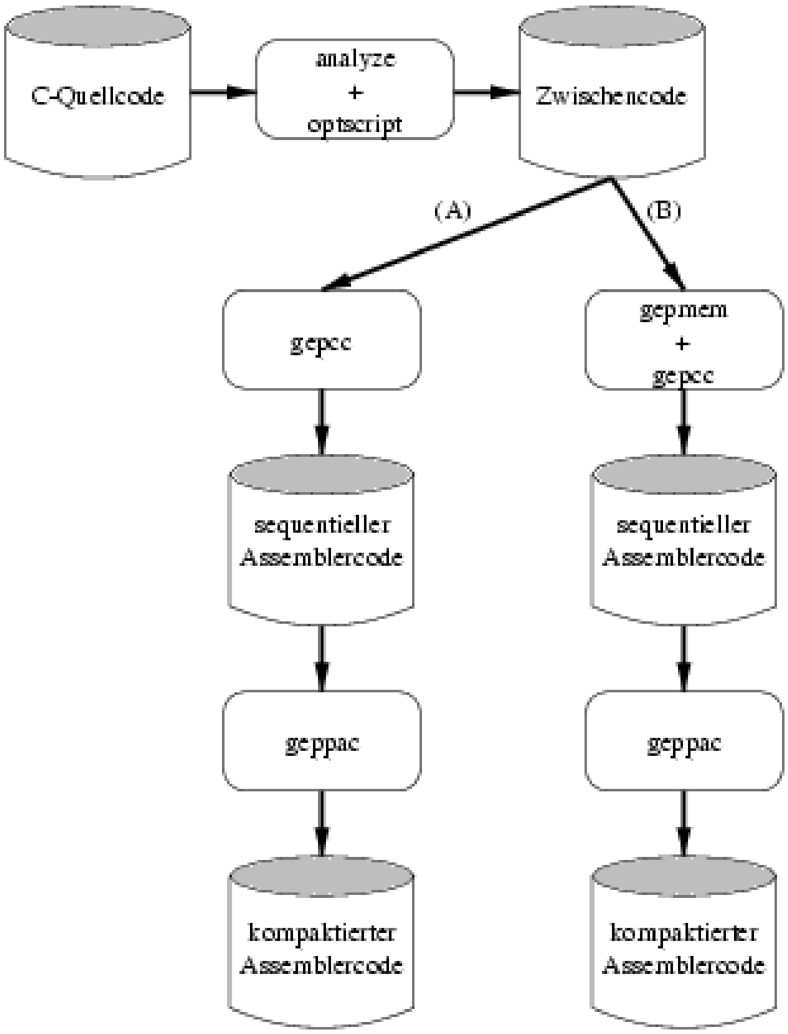


Abbildung 5.1: Schema des Testablauf

Benchmark	gepcc	gepcc + geppac		gepcc + gepmem + geppac	
biquad_N_sections	461	442	-4.12%	430	-6.72%
convolution	247	238	-3.78%	230	-6.88%
dot_product	221	216	-2.26%	207	-6.33%
fir	324	311	-4.01%	301	-7.10%
fir2dim	898	863	-3.90%	834	-7.13%
lms	485	467	-3.71%	454	-6.39%
mat1x3	215	209	-2.80%	205	-4.65%
matrix1	548	530	-3.28%	517	-5.66%
matrix2	607	584	-3.79%	567	-6.59%
n_complex_updates	501	475	-5.19%	454	-9.38%
n_real_updates	338	331	-2.07%	316	-6.51%
real_update	156	153	-1.92%	150	-3.85%
receiver	5926	5655	-4.57%	5523	-6.80%

Tabelle 5.2: Codegröße des erzeugten Assemblers in 32bit-Worten

Größe des Codes ein Abzählen unter Beachtung aller Schleifen und Funktionsaufrufen nicht praktikabel war.

Codegröße der Testprogramme

Die Minimierung des Programmspeicherbedarfs als neben der Laufzeit weiteres wichtiges Maß ist in Tabelle 5.2 aufgelistet. Die Größe des Codes kann über die Anzahl der kompaktierten Instruktionen bestimmt werden, da jede Instruktion die gleiche Größe von 32 Bit besitzt. Da diese Werte mit Programmunterstützung bestimmt werden konnten, sind hier nun auch die Werte für den `receiver` Benchmark enthalten.

Die in den beiden Tabellen 5.1 und 5.2 aufgeführten Resultate sind in den beiden Abbildungen 5.2 auf der nächsten Seite für den Gewinn bei der Laufzeit und in 5.3 auf der nächsten Seite für die Verringerung der Codegöße noch einmal graphisch zusammengefaßt.

Alle Benchmarks konnten den Ergebnissen zufolge von der Speicherpartitionierung profitieren. Dabei wurde im Durchschnitt eine Laufzeitverbesserung von 10.15% gegenüber dem Assemblercode erreicht, den das Backend ohne Erweiterungen generiert. Auch im Vergleich zum kompaktierten, nicht partitionierten Code konnte in jedem Fall ein Geschwindigkeitszuwachs beobachtet werden (um durchschnittlich 3.8%). Ähnlich verhält es sich bei der Betrachtung der Codegrößen. Wieder konnte der allein durch Kompaktierung erzielte Gewinn (im Durchschnitt 3.49%) durch Anwendung der Speicherpartitionierung bei jedem Benchmark gesteigert werden (auf durchschnittlich 5.94%).

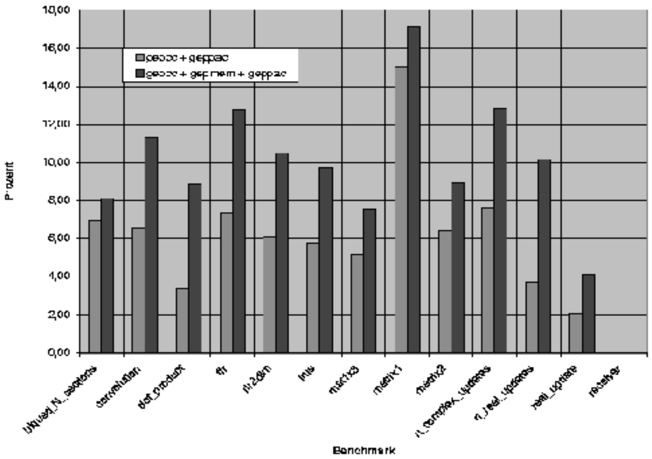


Abbildung 5.2: Prozentuale Verbesserungen der Taktzyklenzahl

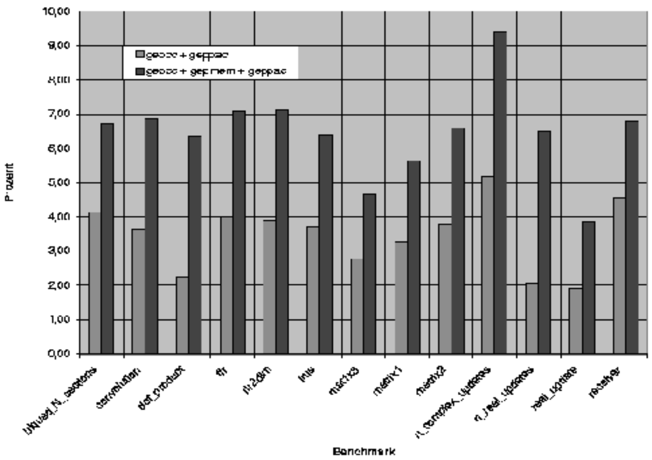


Abbildung 5.3: Prozentuale Verbesserungen der Codegröße

Benchmark	analyze	optscript	gepcc(1)	gepmem	gepcc(2)	geppac
biquad_N_sections	0.04	1.14	0.04	0.18	0.01	0.79
convolution	0.02	0.51	0.02	0.07	0.01	0.07
dot_product	0.04	0.46	0.01	0.07	0.01	0.07
fir	0.05	0.71	0.02	0.14	0.03	0.12
fir2dim	0.09	10.13	0.06	0.99	0.05	0.27
lms	0.05	1.23	0.03	0.26	0.02	0.21
mat1x3	0.02	0.54	0.02	0.15	0.02	0.08
matrix1	0.04	3.20	0.04	0.26	0.03	0.14
matrix2	0.05	5.03	0.03	0.31	0.04	0.16
n_complex_updates	0.05	1.30	0.02	0.32	0.03	1.73
n_real_updates	0.03	0.76	0.02	0.17	0.03	0.15
real_update	0.02	0.29	0.01	0.05	0.01	0.06
receiver	0.56	256.38	0.37	138.40	0.35	3.90

Tabelle 5.3: Laufzeit der Partitionierung in Sekunden

Laufzeiten der Partitionierung

In Bezug auf die praktische Anwendung der Speicherpartitionierung ist die Laufzeit des Verfahrens interessant. Man nimmt bei der Codegenerierung für eingebettete Systeme in der Regel lange Übersetzungszeiten in Kauf, wenn dies zu einer hohen Codequalität führt. Da ein System während seiner Lebensdauer meist nur ein Programm ausführt, lohnt sich der höhere Einsatz bei der Entwicklung und Übersetzung dieses Programms.

Die Laufzeiten der Partitionierung `gepmem` sind sehr niedrig. Sie liegen bei den getesteten Benchmarks immer unter der Laufzeit der Zwischencodeoptimierung `optscript`. Alle Tests wurden dabei – wie schon beim Kompaktierer GEPPAC in Kapitel 3.4.3 auf Seite 48 angegeben – auf einem Linux-PC durchgeführt. Der Rechner war mit einem PentiumII mit 333MHz und 64MB Hauptspeicher ausgestattet. Das System lief mit einem Linux-Kernel der Version 2.2.14 und X-Windows (X11R6).

Entscheidend für die Laufzeit der Speicherpartitionierung ist die Dauer zur Lösung des LPs. Die wiederum wird durch die Größe der gerade betrachteten Funktion und damit der Anzahl der daraus extrahierten Variablen und Nebenbedingungen beeinflußt. Um einen Eindruck von der Komplexität der von `gepmem` erstellten LPs zu bekommen, wird die Anzahl der Variablen und die der Nebenbedingungen für jede Funktion in Tabelle 5.4 auf der nächsten Seite dargestellt.

Benchmark	Funktionen	Variablen	Nebenbedingungen
biquad_N_sections	pin_down	3	22
	main	6	79
convolution	pin_down	3	8
	main	4	21
dot_product	pin_down	0	0
	main	5	30
fir	pin_down	3	9
	main	6	56
fir2dim	pin_down	6	78
	main	8	164
lms	pin_down	6	35
	main	4	60
mat1x3	main	8	70
matrix1	pin_down	4	45
	main	6	88
matrix2	pin_down	4	45
	main	6	94
n_complex_updates	pin_down	5	24
	main	5	100
n_real_updates	pin_down	5	24
	main	5	60
real_update	pin_down	0	0
	main	3	12
receiver	receiver_decod_acb	4	84
	receiver_decod_fcb	4	237
	receiver_decod_gain_vq	11	238
	receiver_decod_norm	4	12
	receiver_level_calc	3	34
	receiver_level_estimator	12	492
	receiver_signal_conceal_sub	13	457
	receiver_SpeechDecoder	6	373

Tabelle 5.4: Größe der generierten Linearen Programme

6 Zusammenfassung

In diesem letzten Kapitel der Diplomarbeit sollen zunächst die erreichten Ergebnisse kurz zusammengefaßt werden. Dazu wird das vorgestellte Verfahren zur Speicherpartitionierung bei DSP-Compilern bezüglich seiner praktischen Anwendbarkeit untersucht. Im weiteren Verlauf soll ein Ausblick auf mögliche Erweiterungen von GEPMEM gegeben werden, die die Leistungsfähigkeit des Verfahrens unter Umständen erhöhen können.

In dieser Diplomarbeit wurde ein Verfahren zur Speicherpartitionierung bei Compilern für Digitale Signalprozessoren entwickelt, das in der vorliegenden Implementierung für den AMS Gepard eine automatische Zuweisung von Programmdaten auf dessen zwei Speicherbänke vornimmt. Anders als in bisherigen Arbeiten wurde dem Aspekt der Retargierbarkeit ein hoher Stellenwert beigemessen. Dies führte zu einer Realisierung, die minimale Annahmen über die Zielarchitektur trifft und konzeptionell näher an einer Zwischencode- als an einer Assemblercodeoptimierung liegt.

Der Kern des Partitionierung ist ein ILP-Modell, das aus Datenabhängigkeitsgraphen und Interferenzgraphen entwickelt und mittels Standardsoftware gelöst wird. Ein großer Vorteil des ILP-Modells ist seine einfache Erweiterbarkeit um speziellere, evtl. prozessorspezifische Bedingungen. Die Größe der ILP-Modelle konnte bei allen Tests so niedrig gehalten werden, daß die Laufzeit der LP-Solver zur ihrer Lösung bei Betrachtung des gesamten Übersetzungsprozesses nicht stark ins Gewicht fällt.

Zusätzlich zu den Anpassungen des Frontends LANCE und des Backends GEPCC wurde ein Kompaktierer für den Gepard-Assemblercode implementiert, der auf das Standardverfahren List-Scheduling aufbaut. Dies wurde notwendig, weil eine Bewertung der Speicherpartitionierung ohne Kompaktierung des erzielten Codes nicht möglich war.

Die erzielten Verbesserungen der Codequalität bestätigen die Implementierung dieses Verfahrens. Zudem konnten alle Testprogramme uneingeschränkt von der Speicherpartitionierung profitieren.

Einen zwiespältigen Eindruck hinterläßt jedoch die zu Beginn der Diplomarbeit getroffene Entscheidung für den AMS Gepard als Zielprozessor. Die Ursache dafür ist nicht der Prozessor selbst. Vielmehr wurden erst im Verlauf der Arbeit durch genaueres Studium des einzigen für diesen DSP Kern verfügbaren Backends GEPCC dessen große Mängel bei der Codegenerierung erkennbar. Hier wäre eventuell die Entwicklung eines eigenen Backends, das zumindest einen Teil der Standardverfahren wie z.B. *Tree-Parsing* implementiert, angebracht gewesen. Allerdings hätte dies auch den Rahmen dieser Arbeit überschritten.

Das vorgestellte Verfahren zur Speicherpartitionierung sollte nun in weiteren, praktisch auch verwendeten Übersetzern eingesetzt werden, um seine Tauglichkeit für unterschiedliche Zielarchitekturen zu überprüfen. Erst dann kann festgestellt werden, ob die Berechnung der Partitionierung auf der Ebene des Zwischencodes auch in Compilern Erfolg hat, die nicht wie der GEPCC eine direkte Abbildung des Zwischencodes auf den Assemblercode vornehmen.

Begründet durch die niedrigen Laufzeiten der Partitionierung bei der funktionsweisen Analyse des Zwischencodes, kann für die Zukunft überlegt werden, ob durch eine Erweiterung des Modells nicht eine globale Programmanalyse in akzeptabler Zeit möglich wird. So könnte eine global optimale Partitionierung erreicht werden. In diesem Rahmen wäre auch eine Erweiterung des im Verfahren verwendeten ILP-Modells um eine geschicktere Behandlung der globalen Variablen realisierbar. Unabhängig davon kann auch eine umfassendere Zeigeranalyse, als im Verfahren mit dem Verfolgen von Zuweisungsmustern im Zwischencode realisiert ist, die Resultate weiter verbessern.

Literaturverzeichnis

[Aus98a] Austria Mikro Systeme International AG. *Embedded DSP Macros - Development Tools, GEPARD DSP Core Family, Assembler & Code Simulator*, 25. März 1998.

[Aus98b] Austria Mikro Systeme International AG. *Embedded DSP Macros - GEPARD Code DSP, Family of Embedded Software Programmable DSP Cores*, 25. März 1998.

[Aus98c] Austria Mikro Systeme International AG. *Embedded DSP Macros - Tentative Information, GEP_03, Embedded Software Programmable DSP Core*, 25. März 1998.

[Aus98d] Austria Mikro Systeme International AG. *Gepard Instruction Set Summary, Release 1.7*, 14. Juli 1998.

[Bas95] Steven Bashford. Code Generation Techniques for Irregular Architectures. Technischer Bericht 596, Lehrstuhl Informatik XII, University of Dortmund, November 1995.

[Bra98] Peter Braun. *Analyse von Assemblercode zur Bestimmung von Datenabhängigkeiten*. Diplomarbeit, Lehrstuhl für Rechnerarchitektur und -kommunikation, Institut für Informatik, Friedrich-Schiller-Universität Jena, 1998.

[CW96] Raul Camposano und Wayne Wolf. Message from the editors-in-chief. *Design Automation for Embedded Systems*, 1996.

[DL99] Steffen Dingel und Rainer Leupers. *LANCE - LS12 ANSI C Compilation Environment, User's Guide*, 12. Mai 1999.

[GJ79] Michael R. Garey und David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[Lan92] William Alexander Landi. *Interprocedural Aliasing in the Presence of Pointers*. Doktorarbeit, Graduate School–New Brunswick, Rutgers, The State University of New Jersey, 1992.

[Lan98] Birger Landwehr. *ILP-basierte Mikroarchitektur-Synthese mit komplexen Bausteinbibliotheken – Wege zu kleineren und schnelleren Schaltungen unter Einsatz algebraischer Optimierungen*. Doktorarbeit, Fachbereich Informatik, Universität Dortmund, 1998.

[LDK+95] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, und A.Wang. Storage Assignment to Decrease Code Size. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, Seiten 186–195. 1995.

[Lee88] Edward A. Lee. Programmable DSP Architectures: Part I. *IEEE ASSP Magazine*, Seiten 4–19, Oktober 1988.

[Lee89] Edward A. Lee. Programmable DSP Architectures: Part II. *IEEE ASSP Magazine*, Seiten 4–14, Januar 1989.

[Leu97] Rainer Leupers. *Retargetable Code Generation For Digital Signal Processors*. Kluwer Academic Publishers, 1997.

[Leu00] Rainer Leupers. *Rechnergestützter Entwurf/Produktion (Mikroelektronik), Begleitmaterial zu Vorlesung im Wintersemester 1999/2000*, 1. Februar 2000.

[MG95] Peter Marwedel und Gert Goossens, Herausgeber. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[Mot] Motorola. *DSP653CCC, Motorola DSP56300 Family, Optimizing C-Compiler User's Manual*.

[Mot86] Motorola. *DSP56000, Digital Signal Processor, User's Manual*, 1986.

[Nac93] ANT Nachrichtentechnik. GSM Half-Rate Codec Simulation, 1993.

[OFG97] E. Ofner, R. Forsyth, und A. Gierlinger. GEPARD, ein parammetrisierbarer DSP Kern für ASICs. DSP Deutschland 1997, 1997.

[PLN92] Douglas B. Powell, Edward A. Lee, und William C. Newman. Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams. In *Proceedings International Conference on Acoustics, Speech, and Signal Processing*, Seiten 5:553–556. 1992.

[Sch99] Mario Schölzel. *Dokumentation zum gepcc V1.0*. Lehrstuhl für Programmiersprachen und Compilerbau, BTU Cottbus, August 1999.

[SCL96] Mazen A.R. Saghir, Paul Chow, und Corinna G. Lee. Exploiting Dual Data-Memory Banks in Digital Signal Processors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seiten 234–243. Oktober 1996.

[SM95] Ashok Sudarsanam und Sharad Malik. Memory Bank and Register Allocation in Software Synthesis for ASIPS. In *Proceedings of the International Conference on Computer Aided Design*, Seiten 388–392. IEEE/ACM, 1995.

[WGHB95] Tom Wilson, Gary Grewal, Shawn Henshall, und Dilip Banerji. An ILP-based Approach to Code Generation. In Peter Marwedel und Gert Goossens, Herausgeber, *Code Generation for Embedded Processors*, Seiten 103–118. Kluwer Academic Publisher, 1995.

[Ziv94] V. Zivojnovic. DSPstone: A DSP-oriented benchmarking methodology. In *International Conference on Signal Processing and Technology*. 1994.