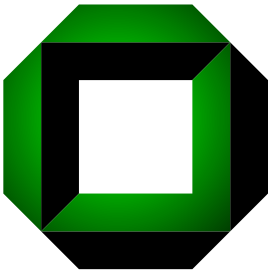

Universität Karlsruhe (TH)

Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation
Lehrstuhl Prof. Goos

Entwurf und Implementierung eines Firm-basierten Backends für die Synchrone Transfer-Architektur

Diplomarbeit von Christian Würdig

31. März 2007

Betreuer:
Dipl.-Inform. Michael Beck
verantwortlicher Betreuer:
Prof. em. Dr. Dr. h.c. Gerhard Goos

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben.

Ort, Datum Unterschrift

Kurzfassung

Aufgrund der Entwicklung immer größerer Speicherkapazitäten und den immer besser werdenden Displays, rückt der Einsatz von Multimediaanwendungen auf mobilen Geräten immer mehr in den Vordergrund. Das erfordert den Einsatz neuer Prozessorarchitekturen, die bei einem möglichst geringen Energiebedarf den notwendigen Datendurchsatz bewältigen können. Diese Lücke soll die Synchrone Transfer-Architektur, STA, schließen.

In dieser Arbeit wird der Entwurf und die Implementierung eines Backends für einen STA basierten Prozessor beschrieben. Es wird untersucht, wo die Probleme bei der Codeerzeugung liegen und mit welchen Änderungen am Prozessor sich der Aufwand für den Übersetzer reduzieren und die Codequalität verbessern lässt. Des Weiteren wird ein optimales Verfahren zur Befehlsanordnung vorgestellt, um die Güte der bereits implementierten Heuristiken abschätzen zu können. Zudem kann so analysiert werden, wie stark sich die Anordnung auf die Registerzuteilung und die Laufzeit des übersetzten Programms auswirkt.

Das Backend wurde in einen Firm-Übersetzer integriert und die Leistungsfähigkeit der erarbeiteten Lösungsansätze mit Messungen belegt.

Danksagung

Ich möchte mich an dieser Stelle bei allen Mitarbeitern und Studenten des Lehrstuhl Goos für die offene und entspannte Atmosphäre bedanken. Ein besonderer Dank geht an Micheal Beck für seine fachkundige und freundliche Betreuung und an Sebastian Hack und Daniel Grund für die Grundsteinlegung des Backends, in das diese Arbeit integriert wurde. Boris Boesler möchte ich für die fachkundige Hilfe seitens der Firma Dresden Silicon danken. Weiterer Dank geht an Michael Beck, Götz Lindenmaier, Sebastian Hack, Matthias Braun, Christoph Mallon und alle anderen, die an der Entwicklung von Firm beteiligt waren. Für die vielen anregenden Diskussionen gilt mein Dank Matthias Braun und Christoph Mallon. Dank auch an Adam, Andreas, Christoph, Matthias, Michael, Moritz und Sebastian für die sehr unterhaltsamen Entspannungsübungen während der Pausen.

Ganz besonders bedanke ich mich bei meiner Lebensgefährtin Wencke Weit für ihre Unterstützung in den arbeitsreichen Wochen. Ebenso möchte ich mich bei meinen Eltern bedanken, die mich immer während meines Studiums voll unterstützt und mich meinen Weg haben finden lassen.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Aufgrund des zunehmenden Einsatzes von Multimedianwendungen steigen die Anforderungen an die Leistungsfähigkeit von mobilen Geräten immer weiter. Allerdings fällt die Zunahme der Energiekapazität von Batterien und Akkus im Vergleich zu den Leistungssteigerungen bei den Prozessoren, eher gering aus, da die physikalischen Grenzen hier wesentlich enger gesetzt sind. Damit sind die Ziele für Prozessoren im mobilen Bereich hoch gesteckt: Die Leistung muss mit den Anwendungsanforderungen Schritt halten, während sich die Energieaufnahme nicht stärker entwickeln sollte, als die Leistungsfähigkeit der Batterien. Zudem soll das Chipdesign gut skalieren und flexibel konfigurierbar sein, um möglichst schnell auf geänderte Anwendungensbedingungen reagieren zu können.

Mit der Synchronen Transfer-Architektur wird versucht, diesen Anforderungen gerecht zu werden. Sie stellt eine explizit programmierbare Datenflussarchitektur für digitale Signalprozessoren dar. Obwohl das Konzept schon vor einigen Jahren entwickelt wurde, ist die Codequalität, die von den existierenden Übersetzern geliefert wird, unbefriedigend. Es hat sich als schwierig herausgestellt, die speziellen Eigenschaften dieses Designs mittels der etablierten Verfahren und Darstellungen gut zu modellieren.

Die am IPD entwickelten Zwischendarstellung Firm hingegen bildet, als graphbasierte SSA-Darstellung, die STA sehr gut ab. Sie stellt im Prinzip einen STA-Prozessor mit unbegrenzten Ressourcen dar. Es liegt somit die Vermutung nahe, dass ein Übersetzer auf dieser Basis eine wesentlich bessere Codequalität bei geringerem Entwicklungsaufwand erreichen kann.

1.2 Aufgabenstellung

Die Aufgabe dieser Arbeit ist der Entwurf und die Implementierung eines Backends für einen Prozessor basierend auf der Synchronen Transfer-Architektur. Es soll zudem erörtert werden, wie gut sich Firm zur Codegenerierung für diese Architektur eignet. Zu diesem Zweck sind Gütekriterien zu entwickeln, um eine entsprechende Bewertung vornehmen zu können. Es werden die auftretenden Probleme analysiert und Vorschläge gemacht, wie sie sich durch Änderungen im Entwurf des Prozessors vereinfachen oder gar vermeiden lassen.

Eine weitere Aufgabe ist die Entwicklung eines ILP Modells zur Befehlsanordnung, um festzustellen, wie stark sich die Registerzuteilung und die Laufzeit eines Programms durch eine optimale Anordnung in der Praxis verbessern lassen.

Der verwendete Prozessor besitzt einen Skalar- und einen Vektorbefehlssatz. Da das Problem der automatischen Vektorisierung sehr komplex und aufwändig ist, wird auf die Unterstützung der Vektoreinheiten verzichtet und nur Code für die skalaren Einheiten generiert.

Das Backend soll zudem in die bestehende Infrastruktur des Firm-Übersetzers integriert werden und sich möglichst nahtlos in das Projekt einfügen.

1.3 Übersicht

In Kapitel 2 werden zunächst die Synchrone Transfer-Architektur und der darauf basierende Prozessor vorgestellt. Des Weiteren werden die notwendigen Grundlagen des Übersetzerbaus und der mathematischen Optimierung gegeben. Anschließend werden in Kapitel 3 verwandte Arbeiten zur Befehlsauswahl und -anordnung diskutiert. Außerdem wird das Übersetzerbauwerkzeug CoSy, auf dem der Referenzübersetzer basiert, präsentiert. Kapitel 4 stellt die erarbeiteten Lösungsansätze für die einzelnen Phasen der Codegenerierung vor, deren Implementierung in Kapitel 5 beschrieben wird. Die aus der Implementierung gewonnenen Messergebnisse sind das Thema des Kapitels 6. Abgeschlossen wird die Arbeit mit der Zusammenfassung und einem Ausblick in Kapitel 7.

2 Grundlagen

2.1 Die Synchrone Transfer Architektur – STA

Die *Synchrone Transfer-Architektur*, kurz STA, wurde erstmals 2004 in [CRS+04] vorgestellt. Beim Entwurf lag der Fokus auf zwei Eigenschaften. Zum Einen soll die Architektur für Prozessoren im Bereich der Multimedia-Anwendungen auf batteriebetriebenen Geräten, wie z.B. Mobiltelefonen oder PDAs, geeignet sein. Das heißt, es muss ein möglichst großer Durchsatz an Daten bei geringem Energieverbrauch gewährleistet werden.

Zum Anderen ist es das Ziel, den Prozess einer konkreten Implementierung in Hardware so weit wie möglich zu automatisieren. Dies verlangt eine sehr gute Skalierbarkeit, was die Anzahl der Funktionseinheiten sowie deren Platzierung und Verdrahtung betrifft. Eine weitere Bedingung war, dass die Architektur im DSP-Bereich auch Echtzeitanforderungen, wie vorhersagbarer maximaler Laufzeit, genügen muss.

Als Ergebnis, das den Anforderungen gerecht wird, ist die STA als eine vereinfachte Form der *Transport Triggered Architecture* [Cor98] entstanden. Während bei herkömmlichen Register-basierten Architekturen die Daten in der Regel zwischen einer Funktionseinheit und einem Registersatz transportiert werden, fließen sie bei der TTA zwischen den Funktionseinheiten. Dazu besitzen die Funktionseinheiten Eingänge, in die Daten geschrieben werden und Ausgänge, in denen das Ergebnis der Operation gespeichert wird.

Die Registersätze sind ebenfalls als separate Funktionseinheiten angelegt, um Werte, die länger benötigt werden, zwischenspeichern zu können. Ein weiterer Unterschied ist, dass bei TTA der Datenfluss im Programm definiert wird und nicht die Folge der Operationen. Auf einem RISC Prozessor wird eine Addition von zwei Werten z.B. folgendermaßen spezifiziert:

```
add r3, r1, r2
```

In einem Programm für TTA erfolgt die Spezifikation über den Datenfluss:

```
mov r1, add.i1
mov r2, add.i2
mov add.o1, r3
```

Mit der zweiten `mov` Anweisung wird der den Inhalt von `r2` in den zweiten Eingang geschrieben, was die eigentliche Operation auslöst. Anschließend wird das Ergebnis aus dem ersten Ausgang der Funktionseinheit in `r3` geschrieben. Dieses Verhalten wird über lokale Warteschlangen in den Funktionseinheiten und einem Kontroller realisiert. Die Operanden werden in die Warteschlangen

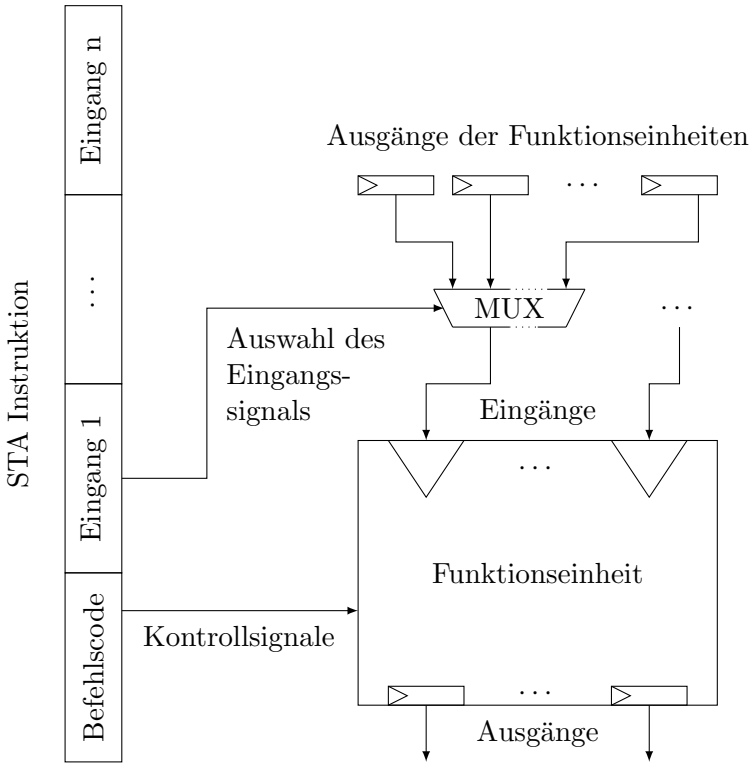

Abbildung 2.1: Der STA Kontrollflusspfad

eingereiht und der Kontroller entscheidet, wann genau die Operation auszuführen ist.

Bei der ST-Architektur wird dies dahingehend verändert, dass die Operationen explizit über die Befehlskodierung gesteuert und ausgelöst werden. In Abbildung 2.1 ist der STA-Kontrollflusspfad dargestellt.

Eine Funktionseinheit besitzt eine beliebige Anzahl an Ein- und Ausgängen, wobei die Ausgänge gepuffert sind. Das heißt das Ergebnis einer Berechnung bleibt solange erhalten, bis es durch eine neue Operation überschrieben wird. Vor jeden Eingang ist ein Multiplexer geschalten, der wiederum mit beliebig vielen Funktionseinheitsausgängen verbunden ist.

Das Verhalten dieser Multiplexer wird durch Signale gesteuert, die aus der Instruktionsdekodierung erzeugt werden. Hier wird für jeden Eingang der Funktionseinheit die entsprechende Kodierung abgelegt, die den gewünschten Ausgang durchschaltet. Alternativ kann auch eine Konstante in der Anweisung gespeichert sein, die der Funktionseinheit vom Instruktionsdekodierer zur Verfügung gestellt wird. Im ersten Teil des Instruktionswortes ist die zu aktivierende Funktionseinheit und die auszuführende Operation kodiert.

Es werden also in jedem Takt die Funktionseinheiten und die Multiplexer durch die aktuelle Anweisung konfiguriert. Somit liegt ein synchrones Netzwerk vor, in dem in jedem Takt Daten produziert und konsumiert werden. Der entsprechende Datenflusspfad ist in Abbildung 2.2 dargestellt.

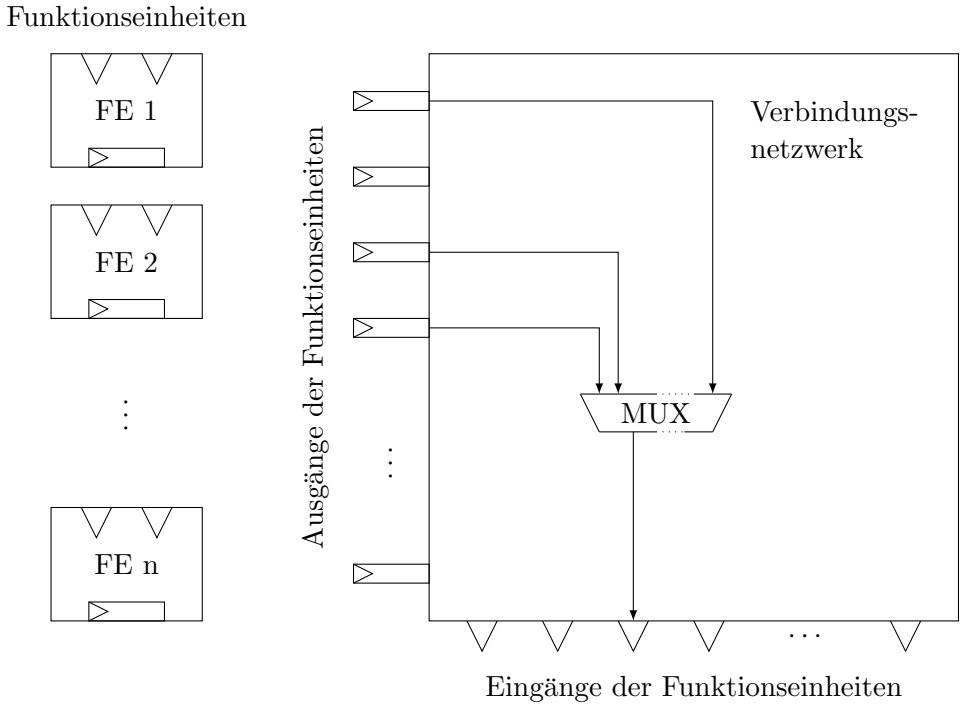

Abbildung 2.2: Der STA Datenflusspfad

Die Programmierung eines STA Systems erfolgt, im Gegensatz zu TTA, ähnlich der einer normalen Registersatz basierten Architektur. Als Beispiel die Addition zweier Werte:

```
alu1.add r1.x r2.x
```

2.2 Der VDSPFXT16

Der VDSPFXT16 ist ein STA basierter 16 Bit DSP, entwickelt am Vodafon Lehrstuhl der TU Dresden. Die Entwicklung des Prozessors erfolgte mit dem Ziel, ihn zur Unterstützung von Multimedia Anwendungen auf mobilen Endgeräten, wie Handys oder PDAs, einzusetzen. Dies impliziert die Notwendigkeit, einen hohen Datendurchsatz bei möglichst geringer Energieaufnahme zu gewährleisten. Bei der ST-Architektur sind die Werte mehr über die Funktionseinheiten verstreut und müssen nicht ständig durch den Registersatz laufen. Dies reduziert den Energieverbrauch und entlastet die allgemeinen Register. Zudem ermöglicht es die Einfachheit des STA Konzeptes, eine Spezifikation des Prozessors zu erstellen, aus der das Modell auf Registertransferebene sowie Simulator und Debugger automatisch generiert werden können. Das ermöglicht das Testen von Anwendungen mit verschiedenen Ausprägungen der Hardware[1],

[1] z.B. Anzahl und Art der Funktionseinheiten, Größe des Registersatzes, Speichergröße, etc.

wodurch die Festlegung eines endgültigen Designs sehr erleichtert wird. Aus diesen Gründen wurde die STA dem Chipdesign zu Grunde gelegt.

In der aktuellen Ausprägung besitzt der Prozessor zwei disjunkte Datenflusspfade – einen Skalar- und einen Vektordatenpfad. Die jeweiligen Netzwerke sind vollverbunden, d.h. jeder Ausgang einer beliebigen Funktionseinheit kann auf jeden Eingang einer beliebigen Funktionseinheit durchgeschalten werden. Die Menge der skalaren Funktionseinheiten setzt sich aus vier ALUs[2] für Addition, Subtraktion und Bitoperationen sowie je einer Einheit für Multiplikation, Schiebeoperationen, Vorzeichenerweiterung, Dekoder für Konstanten, Speicherzugriff und Sprungbefehle zusammen. Divisionen sind nicht in Hardware realisiert und werden zu einem Aufruf an die entsprechende Bibliotheksroutine umgesetzt.

Zur Ausnutzung der Parallelität wird die VLIW[3] Technik unterstützt (siehe auch [Fis83]). Das bedeutet, dass mehrere Befehle, die in einem Takt parallel ausführbar sind, zu einer großen Instruktion zusammengefasst werden. Momentan kann ein VLIW aus bis zu fünf Befehlen bestehen.

Die Einheiten arbeiten, bis auf drei Ausnahmen, mit 16 Bit Werten: Die Multiplikations-, Schiebe- und Vorzeichenerweiterungseinheit produzieren je ein 32 Bit Ergebnis. Es besteht aber die Möglichkeit, das oberere und untere Datenwort der Berechnung separat auszulesen. Die Schiebeeinheit erwartet zusätzlich den zu verändernden Operanden ebenfalls als 32 Bit Wert. Der Dekoder für Konstanten ist notwendig, da die direkte Kodierung in den Befehl[4], aufgrund der benötigten Bitbreite in VLIW Bündeln, nicht möglich ist. Zum Zwischenspeichern von Werten gibt es auch einen Registersatz von 32 16 Bit Registern. Auf diesen Registersatz kann nicht direkt zugegriffen werden, sondern es existieren zu diesem Zweck zwei Schreibe- und vier Leseeinheiten.

Die Vorzeichenerweiterungseinheit und der Dekoder haben eine Verzögerung von 0 Takten, d.h. sie stellen ihr Ergebnis unmittelbar bereit. Sprünge dauern vier Takte – ein Takt für den Sprungbefehl und drei Zusatzzyklen, die sogenannten *Branch-Delay-Slots*, in denen Befehle noch ausgeführt werden bevor tatsächlich gesprungen wird. Speicherzugriffe benötigen zwei Takte und alle anderen Instruktionen einen Takt.

Die Einheiten des Vektordatenpfades arbeiten mit SIMD Vektoren, bestehend aus vier 16 Bit Worten. Es werden die gleichen Funktionen unterstützt, wie im Skalarpfad, außer Sprünge und das Laden von Konstanten, was in den skalaren Einheiten erfolgen muss. Für den Datenaustausch zwischen den beiden Datenpfaden gibt es zwei Verbindungseinheiten, die ICUs[5]. Auch im Vektorpfad gibt es einen Registersatz zur Zwischenspeicherung von Werten. Er besteht aus 16 SIMD Registern.

[2] *engl.* Arithmetic Logic Unit
[3] *engl.* Very Long Instruction Word
[4] Die Kodierung von Konstanten in der Instruktion wird auch *Immediate* genannt
[5] *engl.* Inter Connection Unit

2.3 Übersetzerbau

2.3.1 Grundlegende Begriffe

Ein Übersetzer kann grob in ein *Frontend*, einen Zwischensprachteil und ein *Backend* unterteilt werden (siehe [GW84]). Das Frontend ist für die syntaktische und semantische Analyse des Quellprogramms zuständig und überführt es in eine *Zwischensprache*. Die Zwischensprache abstrahiert idealerweise von den Eigenschaften der Quell- und Zielsprache, so dass auf ihr sprachunabhängige Optimierungen, wie Konstantenfaltung, Eliminierung gemeinsamer Teilausdrücke (CSE[6]), usw., einfach durchzuführen sind. Nachdem alle Zwischensprachoptimierungen durchgeführt wurden, ist es die Aufgabe des Backends, weitere zielmaschinenspezifische Optimierungen durchzuführen und das Programm von der Zwischen- in die Zielsprache zu transformieren. In dieser Arbeit wird eine Zwischensprache betrachtet, in der einzelne Funktionen durch einen Steuerfluss- und einen Datenabhängigkeitsgraph gegeben sind. Zur besseren Veranschaulichung der Definitionen wird im Folgenden das Programm 1 als Beispiel zu Grunde gelegt.

Programm 1 Berechnung von 2^n

```
1: function POT2(n)
2:     cnt ← 1
3:     res ← 1
4:     while cnt ≤ n do
5:         cnt ← cnt + 1
6:         res ← res · 2
7:     end while
8:     return res
9: end function
```

Definition 2.1 (Datenabhängigkeitsgraph)**.** *Der* Datenabhängigkeitsgraph *wird durch die Instruktionen eines Programms P gebildet. Dabei sind zwei Instruktionen $a, b \in P$ durch eine gerichtete Kanten (a, b) verbunden, gdw. die Instruktion a abhängig ist von den Daten, die von Instruktion b erzeugt oder verändert werden.*

Definition 2.2 (Grundblock)**.** *Ein* Grundblock *ist eine maximale Menge von Instruktionen für die gilt: Wenn eine Instruktion ausgeführt wird, dann werden alle Instruktionen des Grundblocks ausgeführt.*

Ein Grundblock beginnt also mit einer Sprungmarke, die den einzigen Eingang darstellt, und endet mit einem (bedingten) Sprung zu einem anderen Grundblock, der den einzigen Ausgang darstellt. Es gibt bei Blöcken, die mit bedingten Sprüngen enden, potentiell mehrere Steuerflussnachfolger, die abhängig von der Bedingsauswertung angesprungen werden. Die Partitionierung des Programms in Grundblöcke bildet die Grundlage für den Steuerflussgraphen.

[6] *engl.* Common Subexpresseion Elimination

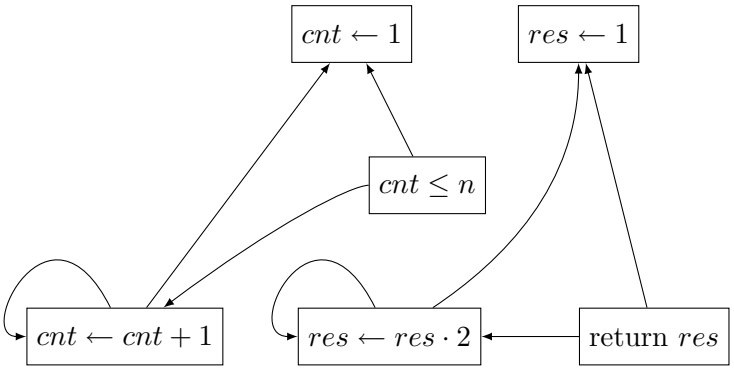

Abbildung 2.3: Datenabhängigkeitsgraph des Programms 1

Definition 2.3 (Steuerflussgraph)**.** *Der* Steuerflussgraph *wird aus den* Grundblöcken *eines Programms gebildet, die die Knoten des Graphen darstellen. Zwei Blöcke a und b sind durch eine gerichtete Kante (a, b) verbunden, gdw. a einen Sprung zu b enthält. Der Steuerflussgraph besitzt einen ausgezeichneten Block, den* Startblock*, an dem die Programmausführung beginnt.*

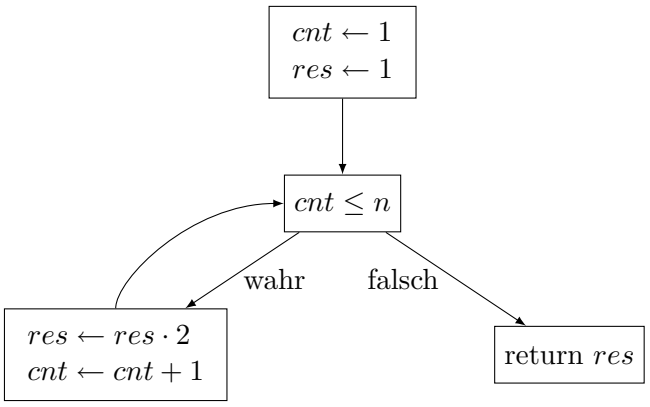

Abbildung 2.4: Steuerflussgraph des Programms 1

Zwei wichtige Eigenschaften von Knoten eines Datenabhängigkeitsgraphs sind die Lebendigkeit und die Interferenz. Die Bestimmung der Lebendigkeit von Variablen an Programmpunkten spielt insbesondere für den Registerzuteiler eine große Rolle, um festzustellen, welche Werte im selben Register gespeichert werden können.

Definition 2.4 (angeordneter Datenabhängigkeitsgraph)**.** *In einem* angeordneten Datenabhängigkeitsgraph *sind die Instruktionen eines Grundblocks durch Sequentialisierungskanten linearisiert. Eine gerichtete Kante (a, b) bedeutet: a wird vor b ausgeführt.*

Definition 2.5 (Programmstelle)**.** *Eine Programmstelle entspricht einer Instruktion im angeordneten Datenabhängigkeitsgraphen.*

Jeder Knoten eines Grundblocks besitzt genau eine ein- und eine ausgehende Sequentialisierungskante. Ausgenommen davon sind der erste Knoten, der nur eine ausgehende Kante besitzt, sowie der letzte Knoten, mit nur einer eingehenden Kante. Eine Anordnung heißt *gültig*, gdw. keine Abhängigkeiten im Graph verletzt werden. Das Verfahren zur Anordnung der Anweisungen eines Programms wird *Befehlsanordnung* oder *Scheduling* genannt.

Definition 2.6 (Lebendigkeit)**.** *Eine Variable a heißt* lebendig *an einer Programmstelle p, wenn es einen Pfad über die Anordnungs- und Steuerflusskanten von p zu einer Benutzung von a gibt, der keine Definition von a enthält.*

Definition 2.7 (Interferenz)**.** *Zwei Variable a und b* interferieren*, gdw. es eine Programmstelle p gibt, an der beide Werte gleichzeitig lebendig sind.*

2.3.2 SSA-Form

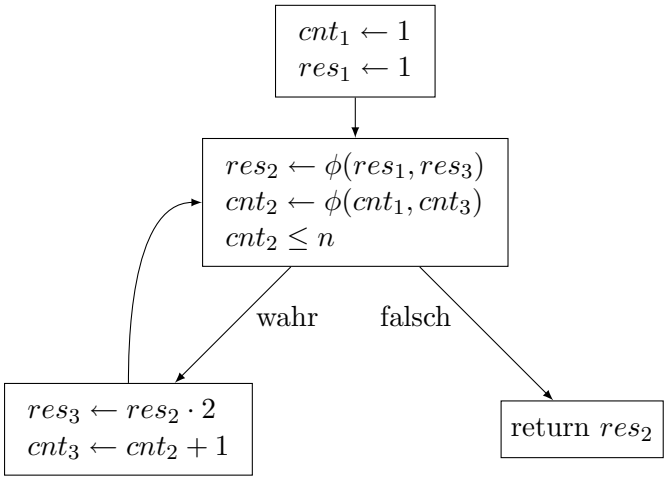

Abbildung 2.5: Steuerflussgraph des Programms 1 in SSA-Form

Ein Programm ist in SSA-Form gdw. jede seiner Variablen an genau einer Stelle im Programm definiert wird. Diese Eigenschaft bewirkt, dass die Definiert-Benutzt-Beziehungen explizit im Programm dargestellt werden, d.h. bei jeder Verwendung einer Variablen die Definition unmittelbar bekannt ist und nicht erst durch eine Datenflussanalyse berechnet werden muss. In [CFR+89] wird gezeigt, dass sich jedes Programm in SSA-Form überführen lässt. Dies geschieht im Prinzip dadurch, dass die Definitionen einer Variable durch Werte ersetzt werden, die eine eindeutige Wertnummer erhalten. Verschiedene Zuweisungen an eine Variable v im Quellprogramm werden also durch Werte v_1, v_2, ... ersetzt. Ein Problem entsteht bei der Zusammenführung des Steuerflusses, wenn die gültige Definition einer Variablen bei einer Benutzung vom Ablaufpfad abhängt. An dieser Stelle wird die sogenannte ϕ-Funktion eingesetzt, die abhängig vom Ablaufpfad die jeweils gültige Definition auswählt.

Zum Aufbau der SSA-Form gibt es mehrere Verfahren. Cytron et al. [CFR+89] zeigen, dass die ϕ-Funktionen in den Blöcken der iterierten Dominanzgrenze der

Definitionen benötigt werden. Das Verfahren transformiert ein, in einer Zwischenprache vorliegendes, Programm in die SSA-Form. Ein zweites Verfahren wurde von Click [Cli95] entwickelt. Es führt während eines Durchlaufs durch den attributierten Strukturbaum eine erweiterte Wertnummerierung durch und erzeugt die SSA-Form direkt. Der Aufwand für beide Verfahren beträgt $O(n^2)$, mit n als Anzahl der lokalen Variablen. In der Praxis ist der Aufwand zur SSA Konstruktion jedoch meist linear.

Definition 2.8 (Dominanz)**.** *Ein Block X dominiert einen Block Y, gdw. auf allen Pfaden des Steuerflussgraphen vom Startblock zu Y, X immer vor Y ausgeführt wird. Man schreibt dann kurz: $X \operatorname{dom} Y$*

Definition 2.9 (Dominanzgrenze)**.** *Die* Dominanzgrenze *eines Blocks X, $DF(X)$, ist die Menge aller Blöcke, die nicht von X dominiert werden, aber mindestens einen Vorgänger besitzen, der von X dominiert wird.*
$DF(X) = \{Y \mid \exists P \in Pred(Y) : X \operatorname{dom} P \wedge \neg(X \operatorname{dom} Y)\}$
Die Dominanzgrenze einer Menge von Blöcken, ist die Vereinigung der Dominanzgrenzen aller Blöcke dieser Menge.
$DF(M) = \bigcup_{X \in M} DG(X)$

Definition 2.10 (iterierte Dominanzgrenze)**.** *Die* iterierte Dominanzgrenze $DF^+(M)$ *ist der minimale Fixpunkt von:*

$$\begin{aligned} DF_0 &= DF(M) \\ DF_{i+1} &= DF(M \cup DF_i) \end{aligned}$$

Definition 2.11 (ϕ-Klasse)**.** *Eine* ϕ-Klasse θ *ist eine maximale Menge von ϕ-Funktionen, für die gilt:*
Für alle $\phi_a, \phi_b \in \theta$ existiert ein Pfad im Datenabhängigkeitsgraphen von ϕ_a nach ϕ_b oder umgekehrt und alle Knoten auf diesem Pfad sind ϕ-Funktionen. Die Menge der Operanden $Op(\theta)$ entspricht der Vereinigung aller nicht-ϕ Operanden aller Elemente von θ.
$Op(\theta) = \bigcup_{\phi \in \theta} \{pred(\phi) \setminus \theta\}$

2.3.3 Das Backend eines Übersetzers

Die Aufgaben des Backends lassen sich in vier Phasen unterteilen.

Befehlsauswahl

Die Befehlsauswahl, auch Codegenerierung genannt, bestimmt für alle Operationen der Zwischensprache die konkreten Maschinenbefehle und wandelt das Quellprogramm in das Maschinenprogramm um. Zwischensprachen besitzen in der Regel nur einfache Operationen, während für viele Prozessoren Befehle existieren, die mehrere dieser Operationen gleichzeitig erledigen können. Eine der Hauptaufgaben bei Auswahl besteht also darin, die entsprechenden komplexen Muster zu finden und mit den Maschinenbefehlen zu überdecken.

Im Laufe der Zeit wurden viele Verfahren entwickelt, um diesen Prozess möglichst gut zu automatisieren. Die Grundidee bei allen automatischen Methoden ist die Trennung der Algorithmen zur Befehlsauswahl und von der Beschreibung der Zielmaschine. Damit soll vermieden werden, die Befehlsauswahl für jede Architektur neu implementieren zu müssen. Statt dessen wird für jede Zielmaschine eine Spezifikation angegeben, aus der der Codegenerator generiert wird. In Kapitel 3.1 wird ein Überblick über die bestehenden Verfahren, sowie ihre Vor- und Nachteile gegeben.

Befehlsanordnung

An die Auswahlphase schließt sich die Anordnung der ausgewählten Maschinenbefehle an, d.h. sie werden in eine gültige, sequentielle Ausführungsreihenfolge gebracht. Dies muss vor der Registerzuteilung geschehen, da sonst die Lebendigkeit der Werte nicht bestimmt werden kann. Es wäre somit nicht möglich, zu entscheiden, welche Werte sich ein Register teilen können und welche nicht. Die Befehlsanordnung hat demzufolge Einfluss auf die Auslagerung von Registern. Eine ungünstige Anordnung kann dazu führen, dass mehr Werte als nötig gleichzeitig lebendig sind und es somit zu mehr Auslagerungen kommt. Als optimal wird eine Anordnung betrachtet, wenn es keine andere Anordnung gibt, die zu einer kürzeren Laufzeit des Programms führt.

In dieser Arbeit wird das Problem der Befehlsanordnung auf ein ganzzahliges lineares Programm abgebildet, um herauszufinden wie stark sich die Anordnung der Befehle in der Praxis auf die Auslagerungen und die Laufzeit auswirkt. Da das Finden einer optimalen Anordnung NP-vollständig ist, wurden verschiedene heuristische Verfahren entwickelt, von denen die am häufigsten verwendeten in Kapitel 3.2 vorgestellt werden.

Registerzuteilung

Die Registerzuteilung weist den angeordneten Maschinenbefehlen Register zu, in denen das Ergebnis der Berechnung gehalten werden soll. Da zu einem Zeitpunkt mehr lebendige Werte existieren können, als die Zielmaschine Register besitzt, müssen einige Werte in den Speicher ausgelagert werden. Die Anzahl lebendiger Werte an einer Programmstelle p wird auch *Registerdruck* genannt. Der Registerdruck eines Grundblocks entspricht dem maximalen Registerdruck über alle Programmstellen innerhalb des Grundblocks. Analog entspricht der Registerdruck des Programms dem maximalen Registerdruck über alle Grundblöcke.

Chaitin stellte 1982 ein Verfahren vor, das die Registerzuteilung auf das Graphfärbeproblem reduziert [Cha82]. Dazu wird ein Interferenzgraph aus den Werten aufgebaut, in dem zwei Werte genau dann durch eine Kante verbunden sind, wenn sie interferieren. Anschließend wird versucht, den Graph mit den k vorhandenen Registern zu färben. Weil das Graphfärbeproblem bekanntermaßen NP-vollständig ist, kommen an dieser Stelle heuristische Verfahren zum Einsatz. Es reicht daher nicht, den Registerdruck des Programms auf k zu senken, denn unter Umständen findet die Heuristik trotzdem keine Lösung. Aus

diesem Grund ist das Verfahren iterativ, d.h. es wird solange der Registerdruck gesenkt, bis das Färben gelingt.

Hack, Grund und Goos [HGG06] konnten zeigen, dass sich dieser Aufwand bei Programmen in SSA-Form reduzieren lässt. Die Erkenntnis ist, dass der Interferenzgraph bei dieser Darstellung chordal ist und damit in polynomieller Zeit gefärbt werden kann. Zudem entspricht die Anzahl der benötigten Farben der Anzahl der Knoten in der größten Clique des Graphen. Es reicht also, den Registerdruck des Programms auf k zu senken um sicherzustellen, dass k Register ausreichend sind. Damit ist der Registerzuteilungsprozess nicht länger iterativ, sondern kann in den zwei Phasen Auslagern und Färben durchgeführt werden.

Befehlsausgabe

Die Ausgabe der Assemblerbefehle für die Zielarchitektur ist die einfachste Phase. Sie entspricht einem Lauf über den Graphen entlang der Anordnungskanten, wobei für jeden besuchten Knoten der entsprechende Maschinenbefehl ausgegeben wird.

2.4 Ganzzahlige lineare Optimierung

Definition 2.12 (ganzzahliges lineares Programm)**.** *Ein* ganzzahliges lineares Programm *P wird charakterisiert durch ein Ungleichungssystem*

$$Ax \leq b$$

mit einer reellen Kostenmatrix $A \in \mathbb{R}^{m \times n}$, einem Variablenvektor $x \in \mathbb{N}_0^n$ und einem reellen Bedingungsvektor $b \in \mathbb{R}^m$. Die Menge der gültigen Lösungen für x sei M.

Definition 2.13 (ganzzahlige lineare Optimierung)**.** *Gegeben sei ein ganzzahliges lineares Programm P und ein reeller Kostenvektor $c \in \mathbb{R}^n$. Unter* ganzzahliger linearer Optimierung *versteht man die Minimierung der* Zielfunktion

$$c^T x = S$$

d.h. gesucht wird eine Lösung $x^* \in M$, so dass gilt:

$$\forall y \in M : c^T x^* \leq c^T y$$

Die Lösung x^* heißt optimal.

Im Falle der vorliegenden Arbeit werden binäre Entscheidungsvariablen zur Modellierung des Problems verwendet. Das heißt, dass der Definitionsbereich des Variablenvektors $\vec{x}$ auf $\{0,1\}^n$ beschränkt wird. Man spricht dann auch von ganzzahliger linearer 0-1-Optimierung. Das Finden der optimalen Lösung x^* ist im allgemeinen NP-vollständig.

3 Verwandte Arbeiten

In diesem Kapitel werden zwei Arten von verwandten Arbeiten vorgestellt. Zum Einen Arbeiten zu Verfahren der Befehlsauswahl und Befehlsanordnung und zum Anderen ein Werkzeug zur Generierung von Übersetzern.

3.1 Befehlsauswahl

Die folgenden Abschnitte zur Beschreibung der einzelnen Verfahren zur Befehlsauswahl dienen lediglich der Erläuterung der grundlegenden Prinzipien und sollen keine formalen und vollständigen Abhandlungen darstellen, da dies den Rahmen der Arbeit sprengen würde. Weitergehende Information und ausführlichere Referenzen finden sich in der Diplomarbeit von Hannes Jakschitsch [Jak04].

3.1.1 Klassische Befehlsauswahl

Die Verfahren der klassischen Befehlsauswahl, in der Literatur auch als interpretierende Befehlsauswahl bekannt, sind die am einfachsten zu implementierenden. Das Eingabeprogramm, das in der Zwischendarstellung vorliegt, wird dabei von einer abstrakten Maschine ausgeführt, die alle nicht statisch berechenbaren Teile, z.B. Operationen die von erst zur Laufzeit bekannten Werten abhängen, als konkreten Maschinencode ausgibt. Das bekannteste Beispiel für statische Berechenbarkeit ist die Konstantenfaltung. Übersetzer, die mit nur einem Durchlauf über die Zwischendarstellung arbeiten, betreiben in dieser Maschinensimulation auch noch Registerzuteilung sowie alle Zielmaschinen spezifischen Optimierungen (algebraische Vereinfachungen, Kurzauswertungen boolscher Ausdrücke etc.). Dies erhöht natürlich die Komplexität des Verfahrens unter Umständen beträchtlich. Moderne Übersetzer führen diese Optimierungen jedoch in der Regel schon vorher auf der Zwischensprache aus, so dass sich nur noch um die Auswahl der Befehle gekümmert werden muss.

Die zwei Hauptverfahren, die nach diesem Schema arbeiten, sind die Makroexpansion und die Entscheidungstabellen. Bei der Makroexpansion wird für jede Operation der Zwischensprache eine Funktion (oder auch Makro) aufgerufen, die, unter Auswertung der Argumente und Nebenbedingungen, den entsprechenden Maschinencode ausgibt. Bei der Variante der Entscheidungstabelle existiert für jeden Befehl auf Zwischensprachebene eine Tabelle, die für alle Varianten des Befehls, den jeweils auszugebenden Maschinencode enthält.

3.1.2 Befehlsauswahl mit Termersetzung

Der klassische Ansatz der Termersetzungsverfahren arbeitet auf Zwischensprachen und Zielmaschinenbeschreibung als Baumdarstellung. Die Zwischenspra-

che und die Befehle der Zielmaschine werden als Grammatik in Präfixform spezifiziert und anschließend wird der, durch ein konkretes Programm gegebene, Ableitungsbaum mittels der Maschinengrammatik zerteilt. Dies ist ein (unvollständiges) Beispiel einer Grammatik einer Zwischensprache:

```
Expr ::= Op Expr Expr | const | Var
Op   ::= + | - | *
```

Und dies ein Beispiel einer Maschinengrammatik:

```
reg -> var,         30, {print 'LD  $R, $1';}
reg -> +(reg reg),   5, {print 'ADD $R, $1, $2';}
reg -> +(reg var),  35, {print 'ADD $R, $1, $2';}
reg -> *(reg reg),  10, {print 'MUL $R, $1, $2';}
```

Bei den Produktionen der Maschinengrammatik definiert das Nichtterminal der linken Seite den Ressourcentyp des gelieferten Ergebnisses (Register, Speicher) und die rechte Seite beschreibt das Befehlsmuster. An die Regeln werden zusätzlich noch ein Kostenmaß, das bei mehrdeutigen Anwendungen als Entscheidungshilfe dient (normalerweise wird die kostengünstigste Anwendung bevorzugt) und eine Aktion, die bei der endgültigen Anwendung der Regel ausgeführt wird, annotiert. Die Ausführung dieser Aktionen in Postfixreihenfolge ergeben das zugehörige Maschinenprogramm. Diese Art der Termersetzung, das Finden und Überdecken, ist das einfachste Verfahren und wurde z.B. in CGSS [JL80] als *LR*-Zerteiler implementiert.

Da einfache Überdeckung aber meist unbefriedigende Codequalität liefert, wurde der Ansatz erweitert zu Termersetzungssystemen, im folgenden TES genannt, die auch Baumtransformationen erlauben. Damit läßt sich z.B. das Distributivgesetz als Termersetzungsregel wie folgt definieren:

```
*(A +(B C)) -> +(*(A B) *(A C))
```

Diese Art der Regelspezifizierung erlaubt eine relativ kompakte und elegante Spezifikation des Ersetzungssystems, hat aber den Nachteil, dass die Termersetzung auf einem kompletten Baum, aufgrund der in den Regeln enthaltenen Variablen (in diesem Beispiel `A`, `B` und `C`), nicht mehr effizient durchführbar ist. Die Lösung ist die Überführung des TES in ein Grundtermersetzungssystem GTES, in dem die Ersetzungsregeln keine Variablen mehr enthalten. Dies geschieht, indem alle Regeln $l \rightarrow r$, die Variablen enthalten, durch eine Menge von Regeln ersetzt werden, die durch alle *benötigten* Substitution der Variablen mittels Grundtermen (Terme ohne Variablen) entstanden ist. Allerdings ist die Konstruktion eines vollständigen GTES aus einem TES nur semi-entscheidbar, d.h. die entsprechenden Algorithmen terminieren nur, falls das GTES existiert. Das Problem hierbei ist, dass die eigentlich unendliche Menge von Substitutionen so eingeschränkt werden muss, dass die entstehende endliche Menge das gleiche leistet, wie die unendliche Menge.

Dieser Weg, die Spezifikation als Termersetzungssystem zu erlauben und daraus automatisch ein Grundtermersetzungssystem zu generieren, ist in BEG (Backendgenerator) implementiert, der auf der Arbeit von Emmelmann [Emm94]

beruht. Für das resultierende Grundtermersetzungssystem kann dann z.B. das Bottom Up Pattern Matching, auch BUPM genannt, eingesetzt werden, das eine effektive Befehlsauswahl ermöglicht. Dabei werden, im Baum von unten nach oben gehend, alle passenden Muster gefunden und anschließend wird von oben herab die kostengünstigste Abdeckung selektiert.

Da, wie bereits erwähnt, die automatische Erzeugung der Grundtermersetzungssysteme nicht immer funktioniert, wurden die Bottom Up Rewrite Systeme, kurz BURS, entwickelt, die direkt mit einer Spezifikation als TES arbeiten. Diese Systeme finden *alle* möglichen Überdeckungen, wobei es exponentiell viele „sinnvolle" gibt[1] und anschließend erfolgt die Suche nach dem globalen Optimum bezüglich der Kosten. Dieses Problem ist im Allgemeinen NP-hart. Zur Reduzierung der Suchkosten wurde u.a. auf die A*-Suche zurückgegriffen, trotzdem sind die resultierenden Codegeneratoren relativ langsam und deutlich komplexer, als auf Grundtermersetzungssystemen basierende.

3.1.3 Befehlsauswahl mit Graphersetzung

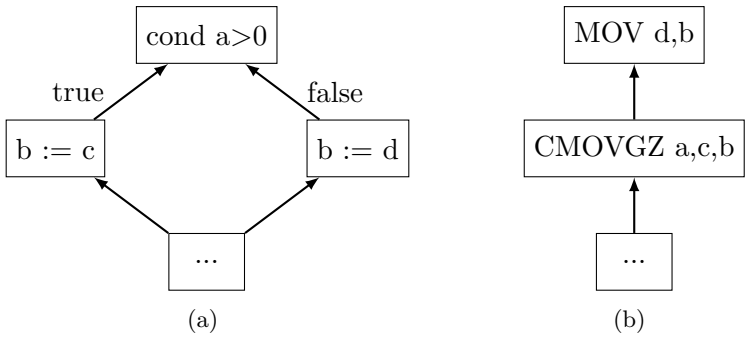

Abbildung 3.1: IF-THEN-ELSE Sequenz (a) und die mögliche Implementierung mittels Conditional Moves (b)

Codeerzeugung mittels Termersetzung hat den Nachteil, dass sie Zwischensprachen in Baumdarstellungen erfordert. Die Zwischensprachen moderner Übersetzer sind heute jedoch in Graphform, da sich nur so die Verwendung gemeinsam genutzter Teilausdrücke richtig modellieren läßt. Der erste Ansatz war, die bekannten Termersetzungsverfahren auf Graphen zu erweitern, indem man den Graph in Bäume aufbricht. Dies führt aber dazu, dass sich z.B. gemeinsam verwendete Teilausdrücke nicht mehr erkennen lassen. Unter Umständen gehen also Informationen verloren, die für Optimierungen fehlen oder erst neu berechnet werden müssen.

Es gibt auch eine Erweiterung des BURS Ansatzes auf DAGs (Directed Acyclic Graphs, gerichtete nicht zyklische Graphen) durch Boesler [Boe98], die allerdings auch nur Baummuster finden kann. Dieses Verfahren wurde im CGGG von Boesler [Boe05] implementiert, es zeigen sich dabei jedoch die Grenzen dieser Variante. So ist es z.B. nicht möglich, eine Regel anzugeben, die das

[1] eine genauere Analyse findet sich in [Boe03]

IF-THEN-ELSE Muster aus Abbildung 3.1 findet und mittels einer Kopie und eines Conditional Move Befehls überdeckt.

Ein Graphersetzungsverfahren, das auf DAGs operiert, stellt die von Eckstein, König und Scholz vorgeschlagene Abbildung der Codeerzeugung aus SSA-Graphen auf PBQP dar [EKS03]. Dieses Verfahren wurde in Behind von Hannes Jakschitsch [Jak04] implementiert und erlaubt die Spezifikation von Graphmustern.

3.2 Befehlsanordnung

3.2.1 List Scheduling

Eines der gebräuchlichsten Anordnungverfahren ist das *List Scheduling*, bei dem die Befehle grundblockweise angeordnet werden. Das Prinzip ist sehr einfach. Zunächst wird eine Menge von Knoten bestimmt, die bereit zur Anordnung sind. Ein Knoten ist genau dann bereit, wenn alle seine Vorgänger bereits angeordnet wurden. Dabei gilt ein Vorgänger auch dann als angeordnet, wenn er einem anderen Grundblock entstammt. ϕ-Funktionen können ebenfalls sofort angeordnet werden. Aus dieser Menge bereiter Knoten wird anschließend ein Knoten zur Anordnung ausgewählt. Diese Auswahl erfolgt durch eine Heuristik, für die verschiedene Kriterien in Frage kommen. Üblicherweise wird der Knoten mit den meisten Nachfolgern ausgewählt, da seine Verzögerung automatisch auch die Verzögerung einer großen Gruppe anderer Befehle nach sich zieht. Dieses Verfahren wird auch in dem Firm-Übersetzer des Institutes verwendet.

3.2.2 Software Pipelining

Software Pipelining ist eigentlich nur eine Abwandlung des List Scheduling. Im wesentlichen unterscheiden sich die beiden Verfahren bei der Auswahlheuristik. Beim Software Pipelining erfolgt eine Simulation der Zielmaschine und es wird abgeschätzt, welche Konflikte und Verzögerungen entstehen würden, wenn ein bestimmter Befehl in einem bestimmten Simulationszustand abgesetzt wird. Auf dieser Basis wird eine Priorität für die bereiten Knoten berechnet und der Knoten mit der höchsten Priorität wird ausgewählt. Thomas Müller stellt in seiner Doktorarbeit einen Ansatz vor, bei dem zur Maschinensimulation endliche Automaten aus einer Spezifikation heraus generiert werden [Mül95]. Implementiert ist dies z.B. in der Gnu Compiler Collection.

3.2.3 Trace Scheduling

Trace Scheduling ist ein grundblockübergreifendes Verfahren und wurde 1981 von Fisher [Fis81] vorgestellt. Es basiert auf dem Ansatz, dass häufig ausgeführte Teile des Codes, die sogenannten *Traces*, zusammen angeordnet werden sollten, um die Laufzeit des Programms zu verbessern. Dazu werden zunächst die am häufigsten ausgeführten Pfade des Programms festgestellt. Dies geschieht entweder durch eine Heuristik zur Abschätzung der Ausführungshäufigkeiten

oder durch die Auswertung eines dynamischen Ausführungsprofils. Anschließend werden die Pfade in die Traces unterteilt. Eine Unterbrechung erfolgt dabei immer an Schleifengrenzen, so dass Schleifen einen eigenen Trace darstellen. Die Traces werden nach ihrer Ausführungshäufigkeit sortiert und mittels eines grundblocklokalen Verfahrens angeordnet. Danach müssen noch Befehle an den Ein- und Ausgängen der Traces eingefügt werden, um die Codeverschiebungen über Grundblockgrenzen hinweg zu kompensieren.

3.2.4 ILP-basierte Befehlsanordnung

Daniel Kästner und Sebastian Winkel präsentieren in [KW01] eine ILP Modellierung für die IA-64 Architektur. Neben der Korrektheit der Anordnung wurde der Fokus auf die möglichst gute Ausnutzung der VLIW Bündel und die Minimierung der Ausführungszeit gelegt. Dieses Verfahren wird in der vorliegenden Arbeit aufgegriffen und erweitert (siehe Kapitel 4.2). Daneben wurde eine ganze Reihe weiterer Ansätze entwickelt, die zum Teil auch Befehlauswahl und Registerzuteilung umfassen. Umfassendere Referenzen dazu finden sich in [KW01].

3.3 CoSy – ein Übersetzerbauwerkzeug

Im folgenden Abschnitt wird lediglich die grundlegende Funktionsweise und das prinzipielle Design des Systems erläutert. Ausführlichere Darstellungen finden sich in [AAvS94] und [COS03].

CoSy [cos] ist ein Werkzeug zur automatischen Generierung von Übersetzern aus einer Spezifikation heraus. Ein solcher Übersetzer besteht aus einer Menge von sogenannten Engines, deren Zusammenarbeit für die Transformation der Quell- in die Zielsprache sorgt. Diese Engines greifen auf einen gemeinsamen Datenbereich, den CDP[2] zu, über den sie kommunizieren und Daten austauschen können.

Der CDP stellt die eigentliche Zwischenrepräsentation von CoSy dar und wird *Common CoSy Medium Intermediate Representation*, kurz CCMIR genannt. Hier werden alle Informationen und Analysedaten der einzelnen Engines gespeichert, so dass es einen zentralen Punkt gibt, an dem alle Informationen zu finden sind. CCMIR selbst ist wiederum in der *full-Structure Definition Language*, fSDL, spezifiziert, einer Sprache zur Beschreibung von Zwischendarstellungen. Mittels fSDL werden die Datenstrukturen und Objekte der Zwischensprache und ihre Attribute festgelegt. Außerdem wird für jede Engine per fSDL deren logische Sicht auf die CCMIR spezifiziert, sowie die Art und Weise des Zugriffs der Engine auf die Objekte und Attribute.

Die Definition von CCMIR in fSDL wird von ACE als Teil von CoSy mitgeliefert und muss nicht jedes Mal neu erstellt werden. Die Darstellung ist hierarchisch gegliedert, mit dem der Quelldatei entsprechenden Modul als Wurzel. Darin enthalten sind die Objekte für die einzelnen Funktionen, die wiederum in Grundblöcke eingeteilt sind. Die Anweisungen des Programms werden als

[2] *engl.* Common Data Pool

Liste von Ausdrucksbäumen in den Grundblöcken dargestellt. Informationen über Datentypen, lokale und globale Variablen, Funktionsparamter, etc. sind als Attribute in den entsprechenden Objekten gespeichert. Es besteht die Möglichkeit, CCMIR über ein *Target Description File* anzupassen. So können z.B. Größe und Ausrichtung oder auch das das Fehlen bestimmter Datentypen festgelegt werden.

Die Interaktion der Engines, sowie deren Schnittstellen, werden in der *Engine Description Language*, EDL, beschrieben. Engines können sequentiell oder parallel arbeiten, sowie als Pipeline, bei der die Ausgabe einer Engine die Eingabe der nächsten darstellt. Zudem können Kontrollstrukturen wie Schleifen oder die bedingte Ausführung von Engines definiert werden. Es ist auch möglich, in EDL neue Engines zu erstellen, die wiederum andere Engines aufrufen können.

Aus der fSDL Beschreibung der Engines und der Zwischensprache wird die endgültige Zwischendarstellung generiert. Dies, zusammen mit der EDL Spezifikation, dient als Grundlage zur Generierung des *Data Manipulation and Control Package*, DMCP. Das Paket ist die Schnittstelle der Engines zur Zwischsprache und sämtliche Zugriffe auf die CCMIR laufen darüber. Es werden Funktionen, wie das Erstellen, Löschen und Kopieren von Knoten oder des Setzen bestimmter Attribute bereitgestellt.

In CoSy sind bereits viele Engines implementiert. So gibt es mehrere C und auch eine C++ Frontend-Engine, die das Quellprogramm in die Zwischensprache transformieren. Des Weiteren sind diverse Standardoptimierungen, wie Konstantenfaltung, Eliminierung gemeinsamer Teilausdrücke, Daten- und Kontrollflussanalyse, etc. als Engines enthalten. Auch das Backend besteht aus mehreren Engines (Musterüberdecker, Registerzuteiler, etc.). Die Codeerzeugung basiert dabei auf BEG (siehe Kapitel 3.1.2). Die dazu notwendigen Engines werden aus der *Code Generator Description* automatisch generiert.

4 Lösungsansatz

4.1 Befehlsauswahl

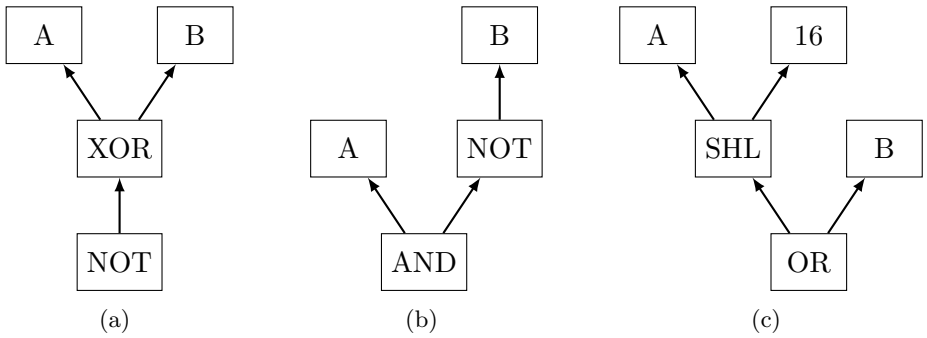

Abbildung 4.1: komplexe STA Befehle XNOR (a), BIC (b) und MRG (c)

Jedes der in Kapitel 3.1 vorgestellten automatischen Verfahren hat spezifische Nachteile. Entweder wird eine Zwischensprache in Baumdarstellung vorausgesetzt, wie z.B. bei BUPM, oder das Finden einer guten Überdeckung dauert, wie z.B. bei BURS, in manchen Fällen extrem lange. Zudem ist der Anschluß eines existierenden Generator-Generators und das Erstellen der entsprechenden Spezifikation auch sehr aufwändig und lohnt sich nur für Architekturen, die viele verschiedene komplexe Operationen besitzen.

In der vorliegenden Arbeit wird die Befehlsauswahl daher als einfaches Makrosubstitutionsverfahren realisiert, in dem die generischen Firm-Knoten durch maschinenspezifische STA-Knoten ersetzt werden. Abgesehen von Vektorbefehlen, besitzt die zugrunde liegende STA Implementierung lediglich drei komplexe Operationen, `XNOR`, `BIC` und `MRG`, die den Mustern in Abbildung 4.1 entsprechen. Das Finden dieser Muster wird folgendermaßen durchgeführt: Für jeden potentiellen Wurzelknoten überprüft man die Vorgänger und fasst die Firm-Knoten zu einem einzigen STA-Knoten zusammen.

Um festzustellen, wie gut Firm zur Codegenerierung für den vorliegenden Prozessor geeignet ist, wird eine Statistik darüber geführt, welche Ersetzungen in welchem Umfang auftreten. Dabei werden vier Ersetzungstypen Firm → STA unterschieden: $1 \rightarrow 1$, $1 \rightarrow n$, $n \rightarrow 1$ sowie $n \rightarrow m$. Pauschal lässt sich sagen, dass viele Ersetzungen der ersten und zweiten Klasse für eine gute Eignung sprechen und viele Ersetzungen der dritten und vierten Klasse eher dagegen. Bei Ersetzungen der dritten und vierten Klasse überlappen sich die Muster der Maschinenbefehle und es ist somit das Überdeckungsproblem zu lösen, was im Allgemeinen NP-vollständig ist (siehe auch Kapitel 3.1).

4.2 Befehlsanordnung

Das Problem der Befehlsanordnung wird in dieser Arbeit mittels einer ILP Formulierung gelöst. Die Ausgangsbasis bildet eine Arbeit von Daniel Kästner und Sebastian Winkel [KW01], welche eine ILP basierte Befehlsanordnung für den Itanium vorstellen. Das dort entwickelte Modell muss jedoch noch erweitert und angepasst werden. So können zum Einen einige Teile entfallen, da es keine besonderen Einschränkungen an die Zusammensetzung der VLIW Bündel gibt, abgesehen davon, dass jede Funktionseinheit nur einmal als ausführende Einheit aufgenommen werden darf. Weiterhin wird der Registerdruck nicht in die Kostenfunktion mit einbezogen. Dieser spielt auf der ST-Architektur eine große Rolle, da so wenig Werte wie möglich im Registersatz oder gar im Speicher zwischengelagert werden sollen. Die Anordnung wird für jeden Grundblock einzeln durchgeführt, so dass ein gerichteter azyklischer Graph $G = (V, E)$ als Grundlage gegeben ist. Zunächst muss dafür gesorgt werden dass die Anordnung korrekt ist, d.h. es müssen folgende Bedingungen erfüllt sein:

1. Jeder Knoten, der einen Wert produziert, muss angeordnet werden und zwar genau einnmal in genau einem Takt.

2. In jedem Takt darf auf jeder Funktionseinheit höchstens ein Knoten ausgeführt werden.

3. Es dürfen insgesamt maximal b Befehle pro Takt ausgeführt werden, wobei b der VLIW Bündelgröße entspricht. Im vorliegenden Fall ist $b = 5$.

4. Es müssen alle Abhängigkeiten beachtet werden. Kein Befehl darf ausgeführt werden, bevor nicht alle Befehle, von denen er abhängt auch ausgeführt wurden und deren Ergebnis vorliegt.

Der Registerdruck wird über zwei Bedingungen im Modell abgebildet:

1. Es wird bestimmt, in welchen Takten ein Knoten lebendig ist. Ein Knoten gilt als lebendig, wenn er bereits angeordnet wurde, aber es noch mindestens einen von ihm abhängigen Knoten gibt, der noch nicht angeordnet wurde.

2. Der Registerdruck für die Funktionseinheiten ergibt sich aus der Anzahl der Knoten, die auf ihnen ausgeführt wurden und noch lebendig sind.

Im Folgenden wird beschrieben, wie mit Hilfe von Entscheidungsvariablen und Ungleichungen das ILP zur Anordnung gebildet wird.

4.2.1 Notationen

Zunächst erfolgt die Einführung einiger Notationen, die bei der Beschreibung des Modells verwendet werden. Ausgangsbasis ist ein gerichteter azyklischer Graph $G = (V, E)$. Die Menge der Kanten E läßt sich in zwei Teilmengen unterscheiden – in die Menge der starken Abhängigkeiten D_S und die Menge der schwachen Abhängigkeiten D_W. Zu den starken Abhängigkeiten gehören alle

echten Datenabhängigkeiten zwischen Knoten. Schwache Abhängigkeiten stellen alle Speicherabhängigkeiten sowie allgemeine Abhängigkeitskanten (siehe Abschnitt 5.1) dar. Diese Unterscheidung wird bei der Modellierung des Registerdrucks genutzt, um festzustellen, ob ein Wert noch lebendig ist. Dazu müssen nur Nachfolger mit starken Abhängigkeiten beachtet werden.

Des Weiteren sei U eine statisch berechnete obere Zeitschranke für die maximale Anzahl an Zyklen, die zur Ausführung des aktuellen Grundblocks benötigt wird. Dann werden für alle $n \in V$ folgende Funktionen bzw. Mengen definiert:

- Es sind $asap(n)$ der früheste Zeitpunkt und $alap(n)$ der späteste Zeitpunkt, zu dem ein Knoten angeordnet werden kann, ohne Datenabhängigkeiten bzw. die Zeitschranke U zu verletzen.
- Das gültige Anordnungsintervall $I(n)$ ist als $I(n) = \{asap(n), ..., alap(n)\}$ definiert.
- $K(n)$ sei die Menge der Funktionseinheitstypen auf denen n ausgeführt werden kann.
- Für alle $k \in K(n)$ entspricht k_n der Anzahl der insgesamt verfügbaren Funktionseinheiten vom Typ k.
- Mit $C(n) = \{m \mid (m, n) \in D_S\}$ wird die Menge aller Konsumenten von n bezeichnet, sowie deren Anzahl als $c_n = |C(n)|$
- Die Menge der Knoten m, die von einem Knoten n abhängig sind, wird definiert als: $D(n) = \{m \mid (n, m) \in E\}$
- Für alle $m \in D(n)$ liefert $w_m(n)$ die Anzahl der Zyklen, die zwischen dem Ausführungszeitpunkt von n und dem von m liegen müssen, um die Abhängigkeit einzuhalten.

Des Weiteren bezeichnet R die Menge aller verfügbaren Funktionseinheitstypen.

4.2.2 Die Modellierung der Befehlsanordnung

Die Befehlsanordung wird über binäre Entscheidungsvariablen modelliert. Dazu werden für alle Knoten $n \in V$ binäre Variablen x_{nt}^k eingeführt, mit $k \in K(n)$ und $t \in I(n)$, die folgende Bedeutung haben:

$$x_{nt}^k = \begin{cases} 1 & \text{Knoten } n \text{ wird zu Zeitpunkt } t \text{ auf einer Funktionseinheit} \\ & \text{vom Typ } k \text{ ausgeführt} \\ 0 & \text{sonst} \end{cases}$$

Zunächst ist dafür zu sorgen, dass die Befehlsanordnung gültig ist, d.h. es müssen Abhängigkeiten und Ressourcenkonflikte beachtet werden. Außerdem muss die Anordnung vollständig sein.

1.) Zuweisungsbedingungen

Es muss sichergestellt sein, dass jeder Knoten genau einmal zu genau einem Zeitpunkt auf genau einer Funktionseinheit ausgeführt wird:

$$\forall n \in V : \sum_{k \in K(n)} \sum_{t \in I(n)} x_{nt}^{k} = 1$$

2.) Abhängigkeitsbedingungen

Ein Knoten ist unter Umständen von anderen Knoten abhängig, d.h. er darf nicht angeordnet werden, bevor alle anderen Knoten, von denen er abhängt, auch angeordnet sind, unter Berücksichtigung deren Latenz. Für jede der Abhängigkeiten $m \in D(n)$ wird ein Überlappungsintervall $I_O(m,n) = \{t' + w_n(m) - 1 \mid t' \in I(m)\} \cap I(n)$ berechnet, das den kritischen Zeitraum abdeckt, innerhalb dessen Abhängigkeitsverletzungen auftreten können. Zu allen Zeitpunkten $t \in I_O(m,n)$ muss folgendes gelten: Wenn n zu einem Punkt $t_n \in \{asap(n), ..., t\}$ angeordnet wird, dann darf m zu keinem Zeitpunkt $t_m \in \{t - w_n(m) + 1\}$ angeordnet werden, oder umgekehrt. Die folgende Ungleichungsmenge erzwingt diese Bedingung:

$$\forall n \in V, \forall m \in D(n), \forall t \in I_O(m,n) :$$
$$\sum_{k \in K(n)} \sum_{\substack{t_n \le t, \\ t_n \in I(n)}} x_{nt_n}^{k} + \sum_{k \in K(m)} \sum_{\substack{t_m \ge t - w_m(n) + 1, \\ t_m \in I(m)}} x_{mt_m}^{k} \le 1$$

3.) Ressourcenbedingungen

In jedem Zeitschritt stehen nur eine beschränkte Anzahl R_k an Funktionseinheiten für jeden Typ $k \in R$ zur Verfügung - es dürfen also in einem Schritt nicht mehr Knoten ausgeführt werden, als Funktionseinheiten vorhanden sind. Dies wird durch die folgenden Bedingungen gewährleistet:

$$\forall k \in R, \forall 1 \le t \le U : \sum_{n \in V : k \in K(n)} x_{nt}^{k} \le R_k$$

4.) Bündelungsbedingungen

Die STA ist eine VLIW Architektur, d.h. es können in einem Takt mehrere Anweisungen, die zu einem Bündel zusammengefasst werden, ausgeführt werden. Da die Bündelgröße beschränkt ist, müssen auch hierfür Bedingen in das ILP eingefügt werden:

$$\forall 1 \le t \le U : \sum_{k \in K(n)} \sum_{n \in V} x_{nt}^{k} \le b$$

4.2.3 Die Zielfunktion

Jede Lösung des Gleichungssystems, die die vier Bedingungen erfüllt, entspricht einer gültigen Befehlsanordnung (siehe [KW01]). Der ILP Löser wird eine Lösung auswählen, die optimal bezüglich einer Zielfunktion ist. In diesem Fall wird die Anordnung mit der minimalen Ausführungszeit T gesucht, was durch folgende Zielfunktion ausgedrückt wird:

$$\sum_{n \in V} \sum_{k \in K(n)} \sum_{t \in I(n)} t \cdot x_{nt}^k$$

4.2.4 Modellierung des Registerdrucks

Die bisherige ILP Formulierung berechnet die Ausführungsreihenfolge mit der minimalen Ausführungszeit, was unter Anderem durch die Anordnung von parallel ausführbaren Instruktionen in einem VLIW erreicht wird. Dadurch kann sich aber der Druck auf die Ausgänge der Funktionseinheiten so erhöhen, dass Werte in die Registereinheit ausgelagert werden müssen. Dies soll nach Möglichkeit vermieden werden. Zudem kann es bei zu vielen Auslagerungen passieren, dass die Größe der Registereinheit nicht ausreicht und dann in den Speicher ausgelagert werden muss – eine Operation die sehr teuer ist.

Das ILP muss also um ein Kostenmaß erweitert werden, das mit der Anzahl der Auslagerungen korrespondiert, die bei einer bestimmten Anordnung entstehen würden. Dazu werden für jeden Knoten $n \in V$ weitere binäre Entscheidungsvariablen a_{nt}^k und y_{nt}^k eingeführt, die die folgende Bedeutung haben:

$$a_{nt}^k = \begin{cases} 1 & \text{Knoten } n \text{ ist zu Zeitpunkt } t \text{ auf einer Funktionseinheit} \\ & \text{vom Typ } k \text{ lebendig} \\ 0 & \text{sonst} \end{cases}$$

mit $k \in K(n)$ und $asap(n) \leq t \leq U$

$$y_{nt}^k = \begin{cases} 1 & \text{Knoten } n \text{ wird zu Zeitpunkt } t \text{ auf einer Funktionseinheit vom} \\ & \text{Typ } k \text{ ausgeführt, obwohl alle Einheiten dieses Typs belegt sind} \\ 0 & \text{sonst} \end{cases}$$

mit $k \in K(n)$ und $t \in I(n)$
Die Variablen a_{nt}^k werden mit folgenden Bedingungen beschrieben:

$$\forall n \in V, \forall k \in k(n), \forall t : asap(n) \leq t \leq U :$$

$$\overbrace{\sum_{\substack{t_n \leq t, \\ t_n \in I(n)}} c_n \cdot x_{nt_n}^k}^{P} - \overbrace{\sum_{m \in C(n)} \sum_{k \in K(m)} x_{mt}^k}^{B} - c_n \cdot a_{nt}^k \leq 0$$

Es können drei Fälle eintreten:

1. Knoten n ist noch nicht angeordnet, dann ist $P = 0$ und auch $B = 0$, da aufgrund der Abhängigkeitsbedingungen auch seine Konsumenten noch

nicht angeordnet sein können. Der ILP Löser wird $a_{nt}^k = 0$ setzen, weil $P - U \leq 0$ die Ungleichung erfüllt und a_{nt}^k als positiver Kostenfaktor in die Zielfunktion eingeht.

2. Knoten n ist angeordnet, aber es sind noch nicht alle Konsumenten von n angeordnet. In diesem Fall ist $P = c_n$ und $B = c$ mit $0 \leq c < c_n$ und somit ist $P - U > 0$, die Ungleichung wäre also nicht erfüllt. Daher muss die Variable $a_{nt}^k = 1$ gesetzt werden, um eine gültige Lösung zu erhalten. Es gilt dann $c_n - c - c_n = -c \leq 0$ für alle möglichen positiven Anzahlen von Konsumenten (negative gibt es nicht).

3. Knoten n und all seine Konsumenten $C(n)$ sind angeordnet. Damit ist $P = B = c_n$ und $P - B \leq 0$ und somit die Ungleichung erfüllt. Der ILP Löser wird $a_{nt}^k = 0$ setzen, aus dem gleichen Grund wie im ersten Fall.

Die so erzeugte Menge von Ungleichungen erzwingt also das gewünschte Verhalten der Variablen a_{nt}^k.

Zuletzt müssen noch die Zusatzkosten modelliert werden, die entstehen sollen, wenn zu einem Zeitpunkt mehr Werte lebendig sind, als Funktionseinheiten verfügbar. Dazu werden Variablen y_{nt}^k mit der oben genannten Bedeutung eingeführt, die mit folgender Bedingung gesteuert werden:

$$\forall n \in V, \forall k \in K(n), \forall t : asap(n) \leq t \leq U :$$

$$\overbrace{\sum_{m \in V} a_{mt}^k}^{L} - |V| \cdot y_{nt}^k \leq k_n$$

Die Anzahl der Knoten, die zu einem Zeitpunkt t auf einem Funktionseinheitentyp k lebendig sind, wird durch L beschrieben. Wenn $L > k_n$ ist, dann muss $y_{nt}^k = 1$ gesetzt werden, damit die Ungleichung erfüllt wird. Wenn $L \leq k_n$ ist, dann wird der ILP Löser $y_{nt}^k = 0$ setzen, da diese Variablen mit positivem Gewicht in die Zielfunktion eingehen. Somit werden also genau dann zusätzliche Kosten zur Zielfunktion addiert, wenn der Druck auf die Funktionseinheiten so groß wird, dass ausgelagert werden muss.

4.2.5 Die erweiterte Zielfunktion

Die in Abschnitt 4.2.3 beschriebene Zielfunktion muss jetzt nur noch um die neuen Variablen erweitert werden:

$$\sum_{n \in V} \sum_{k \in K(n)} \sum_{t \in I(n)} \left(t \cdot x_{nt}^k + a_{nt}^k + |V| \cdot U \cdot y_{nt}^k \right)$$

Das Hinzufügen der Variablen a_{nt}^k mit positiver Gewichtung bewirkt, dass Knoten nicht länger als notwendig lebendig gehalten werden. Die Variablen y_{nt}^k gehen am stärksten gewichtet in die Zielfunktion ein, da das Auslagern sehr teuer ist.

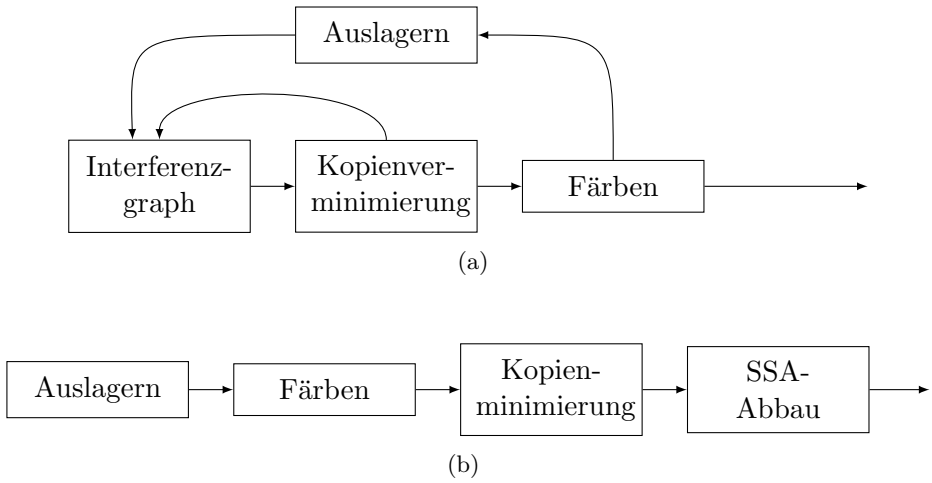

Abbildung 4.2: klassische Registerzuteilung nach Chaintin (a) und Registerzuteilung auf SSA (b)

4.3 Registerzuteilung und -auslagerung

Anschließend an die Befehlsanordnung folgt die Phase der Registerzuteilung und -auslagerung. Dabei wird an jeden Knoten annotiert, in welchem Register der von ihm repräsentierte Wert gehalten werden soll. Falls die Anzahl der Register nicht ausreicht, müssen andere Werte ausgelagert werden. Hack, Grund und Goos [HGG06] konnten zeigen, dass bei Programmen in SSA Form die Registerzuteilung optimal in polynomieller Zeit durchgeführt werden kann (siehe auch Abbildung 4.2). Es reicht dazu aus, den Registerdruck vorher an jeder Stelle im Programm auf die Anzahl der verfügbaren Register zu senken. Damit wird der klassische iterative Prozess nach Chaitin [Cha82] in einen rein sequentiellen Prozess transformiert.

Im vorhandenen generischen Backendrahmenwerk ist bereits die Registerauslagerung nach Belady [Bel66] und die SSA basierte Registerzuteilung nach Hack implementiert. Zur Kommunikation gibt es eine Schnittstelle, über die vom konkreten Backend Informationen erfragt werden können. Allerdings liegt den Implementierungen die Annahme einer Register basierten Architektur zu Grunde, was bei STA nicht gegeben ist. Im Folgenden werden die daraus resultierenden Probleme und die erarbeiteten Lösungsansätze beschrieben.

4.3.1 Modellierung der Registeranforderungen

Wie bereits erwähnt, liegen bei der STA die Werte an den Ausgängen der Funktionseinheiten an, auf die die anderen Einheiten über ein Multiplexer Netzwerk zugreifen können. Ein Ausgang stellt damit einen Zwischenpuffer dar, in dem das Ergebnis einer Operation solange gehalten wird, bis es durch eine neue Operation überschrieben wird. Es sind auch Register vorhanden, welche als eigene Funktionseinheit realisiert sind. Auf diese Einheit kann jedoch nicht direkt zugegriffen werden, sondern es existieren zwei Schreib- und vier Leseeinheiten.

Für die Modellierung der Registeranforderungen gibt es mehrere Möglichkeiten. Zum Einen kann man die Funktionseinheiten als Puffer ignorieren und die Registerzuteilung wie herkömmlich auf dem Registersatz durchführen. Vor jeder Operation müssen dann die benötigten Werte aus den Registern geladen und anschließend das Ergebnis der Operation zurückgeschrieben werden. Dies würde jedoch zu zwei zusätzlichen Zyklen pro Operation führen und außerdem unterbräche es den Fluss der Werte durch die Funktionseinheiten. Mittels einer Gucklochoptimierung, die erkennt, ob ein Wert nur kurzzeitig lebendig ist und dann auf das Schreiben des Wertes in ein Register verzichtet, ließe sich die Anzahl dieser Zusatzzyklen verringern. Allerdings ist aufgrund der starken lokalen Beschränkung das Optimierungspotenzial eher gering. Der Vorteil dieser Variante liegt darin, dass sie die ST-Architektur auf eine herkömmliche Architektur abbildet. Damit können alle Phasen des generischen Backends ohne größere Änderungen angewendet werden.

Die andere Variante ist, die einzelnen Funktionseinheiten direkt als Register zu modellieren und den Registersatz als schnellen Auslagerungsspeicher zu betrachten. Dazu wird für jede Funktionseinheit ein eigenes Register eingeführt. Jetzt kann man entweder all diese Register in einer einzigen Registerklasse zusammenfassen, oder man führt für jeden Funktionseinheitentyp eine eigene Klasse ein.

Das Zusammenfassen in eine einzige Registerklasse hat den Vorteil, dass der Registerzuteilungsalgorithmus nur einmal laufen muss, da er jede Registerklasse einzeln behandelt. Allerdings würden dann auch Kopien entstehen, die sich nicht umsetzen lassen. Der Registerzuteiler könnte z.B. einen Wert von einer `ALU` in die Ladeeinheit kopieren, wenn diese zu dem Zeitpunkt frei ist und die `ALU` benötigt wird. Für eine solche Kopie, von einer Einheit in eine andere, gibt es keine Hardwareunterstützung. Aus diesem Grund wird für jeden Funktionseinheitentyp eine eigene Registerklasse verwendet.

5 Implementierung

Eine Aufgabe dieser Arbeit ist es, das Backend in den am Institut entwickelten Übersetzer zu integrieren. Dies ermöglicht es, die Leistungsfähigkeit und Praktikabilität der entwickelten Ansätze zu überprüfen. Im folgenden Kapitel wird die Implementierung des STA-Backends beschrieben.

5.1 Die Zwischensprache Firm

Der Übersetzer benutzt zur Zwischendarstellung die, in den letzten 10 Jahren am IPD entwickelte, SSA-Darstellung Firm [LBBG05]. Firm ist graphbasiert, d.h. die Funktionen einer Übersetzungseinheit werden als gerichtete Graphen dargestellt, die in Grundblöcke unterteilt sind. Die Operationen einer Funktion sind als Knoten dargestellt, die den Wert repräsentieren, den ihre Ausführung erzeugt. Verbunden sind die Knoten über Datenabhängigkeitskanten. Das bedeutet, dass jeder Knoten auf die Werte zeigt, die er zur Berechnung benötigt. Zudem ist jeder dieser Knoten genau einem Grundblock zugeordnet. Die Grundblöcke sind über Datenflusskanten mit den Sprungbefehlen verbunden, von denen aus sie erreicht werden können.

Der Speicher ist in Firm als symbolischer Speicherwert repräsentiert. Operationen, die auf dem Speicher arbeiten, erhalten diesen Wert als Eingabe und produzieren als Resultat den potentiell veränderten Speicher, den die nächste Speicheroperation wiederum als Eingabe erhält. Auf diese Art und Weise erfolgt eine partielle Anordnung der Speicheroperationen. So wird verhindert, dass z.B. durch eine Optimierung, eine Ladeoperation vor eine Schreibeoperation gezogen wird, die auf den selben Speicherbereich zugreift.

Es gibt noch einen weiteren Kantentyp – die allgemeinen Abhängigkeitskanten. Kanten diesen Typs dienen zur Darstellung von Abhängigkeiten, die nicht durch einen Wertfluss zwischen zwei Knoten repräsentiert wird.

Zur Verdeutlichung ist in Abbildung 5.1 der Algorithmus 1 aus Kapitel 2 als Firm-Graph dargestellt. Schwarze Kanten entsprechen Daten- und blaue Kanten Speicherabhängigkeiten. Rote Kanten stellen den Steuerfluss dar. Rückwärtskanten aufgrund von Schleifen sind gestrichelt.

5.2 Befehlsauswahl

Die Befehlsauswahl ist als einfacher Lauf über den Graphen implementiert. Jeder besuchte Firm-Knoten wird durch einen entsprechenden STA-Knoten ersetzt, der die notwendigen maschinenspezifischen Attribute, wie z.B. Registeranforderungen enthält. Falls der aktuelle Knoten die Wurzel eines komplexen Befehls wie `XNOR` darstellen könnte, wird die Suche nach den entsprechenden

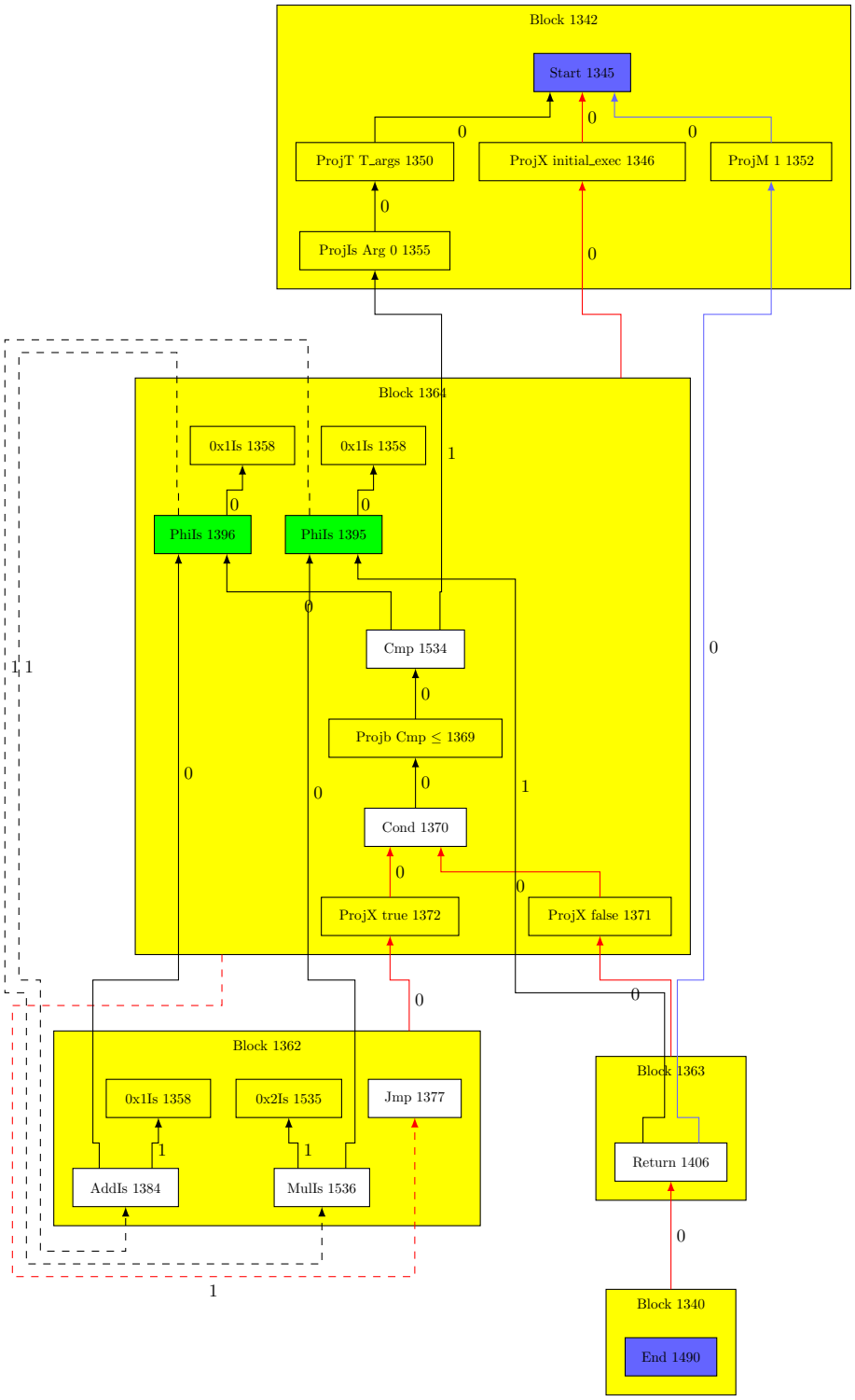

Abbildung 5.1: Firm-Graph des Programms 1 aus Kapitel 2

Vorgängern ausprogrammiert. Da es nur drei Muster gibt, die zudem nur eine geringe Baumtiefe besitzen, ist der Aufwand dafür gering.

5.3 Befehlsanordnung

Zur Kommunikation mit dem ILP Löser gibt es eine Schnittstelle, mit deren Hilfe sich Entscheidungsvariablen und Ungleichungen erzeugen lassen. Nach der Abbildung des Anordnugsproblem beginnt der Problemlöser mit der Berechnung. Anschließend wird das Ergebnis über die Schnittstelle erfragt und die Befehlsanordnung entsprechend der Lösung durchgeführt. Falls innerhalb einer bestimmten Zeitgrenze keine gültige Lösung gefunden wird, dann wird der Block mittels des Listschedulingverfahrens angeordnet. Die implementierte Auswahlheuristik berechnet dabei für jeden Befehl eine Priorität. Bevorzugt werden Instruktionen, von denen viele andere Instruktionen abhängen. Außerdem wird der aktuelle Registerdruck des Blocks berechnet und Instruktionen, die Register freigeben, werden um so stärker bevorzugt, je höher der Registerdruck ist.

5.4 Auslagerungsphase

Die Auslagerungphase wird in zwei Schritten durchgeführt:

1. Auslagerung der Werte in den Registersatz
2. Auslagerung der Werte aus dem Registersatz in den Speicher

Aufgrund der in Kapitel 4.3 beschriebenen Modellierung der Register, muss der Auslagerungsalgorithmus angepasst werden. In der ursprünglichen Implementierung wird die Auslagerung und Registerzuteilung für jede Registerklasse separat durchgeführt. Da bei dem hier vorgestellten Lösungsansatz die Werte bei der Einlagerung die Registerklasse wechseln und diese Einlagerungsklasse für alle Registerklassen die selbe ist, muss die Auslagerung für alle Klassen gleichzeitig erfolgen. Ansonsten können potentiell an einer Stelle im Programm mehr Einlagerungen lebendig sein, als die Einlagerungsklasse Register besitzt.

Wie bereits erwähnt, ist die Auslagerung nach dem Verfahren von Belady [Bel66] realisiert. Der Algorithmus wurde dahingehend geändert, dass die Arbeitsmengen in die einzelnen Registerklassen partitioniert sind und die Werte in die passende Partition eingetragen werden. Mengenoperationen sind als komponentenweise Operationen zu verstehen, die die jeweils passenden Partitionen[1] miteinander verknüpfen. Das angepasste Verfahren ist in Algorithmus 3 als Pseudocode dargestellt. Zum Vergleich ist in Anhang A.1 die ursprüngliche Implementierung aufgeführt.

Es gibt allerdings noch ein weiteres Problem - ϕ-Funktionen mit Argumenten aus verschiedenen Registerklassen. In Abbildung 5.1 tritt es z.B. bei beiden ϕ-Funktionen auf. Einmal wird eine Konstante ausgewählt, die der Registerklasse `Decoder` zugeordnet ist und einmal ein `Add` bzw. `Mul`, die der Registerklasse

[1] d.h. Partitionen, die der gleichen Registerklasse entsprechen

Algorithmus 2 Angepasster Belady Algorithmus für einen Grundblock

```
1:  function DISPLACE(Set Workset, Set Values, Set reg_classes)
2:      X ← ∅
3:      for each c ∈ reg_classes do
4:          T ← Workset_c ∪ Values_c
5:          Sort T ascending by next-use distances
6:          Workset_c ← first AvailableRegister(c) entries from T
7:          X_c ← T \ Workset_c
8:      end for
9:      return X                                  ▷ return all displaced values
10: end function
11:
12: function SPILLBLOCKBELADY(Basic block B, Set reg_classes)
13:     ▷ compute live-in values of all register classes
14:     for each c ∈ reg_classes do
15:         T ← livein_{B,c} ∪ Φ_{B,c}
16:         Sort T ascending by next-use distances
17:         W_c ← first AvailableRegister(c) entries from T
18:     end for
19:     in_B ← W                  ▷ record all values being in registers at begin of B
20:     U ← ∅                                  ▷ U contains all values used within B
21:     insn ← sched_first_B
22:     while insn is not sched_end_B do
23:         if insn is not φ then
24:             for each v ∈ Uses_{reg_classes}(insn) \ W do
25:                 Place Reload(v) before insn
26:             end for
27:             X ← DISPLACE(W, insn, Uses_{reg_classes}(insn), reg_classes)
28:             ▷ Remove all values being diplaced before first usage from in_B
29:             in_B ← in_B \ {X \ U}
30:             U ← U ∪ Uses_{reg_classes}(insn)
31:             DISPLACE(W, insn, Defs_{reg_classes}(insn), reg_classes)
32:         end if
33:         insn ← sched_next_B(insn)
34:     end while
35:     out_B ← W
36: end function
```

Algorithmus 3 Angepasster Belady Auslagerungsalgorithmus

```
1:  function SpillBelady(Function F, Set reg_classes)
2:      ▷ spill all basic blocks
3:      for each basic block B ∈ F do
4:          SpillBlockBelady(B, reg_classes)
5:      end for
6:      ▷ ensure all needed values are in start workset
7:      for each basic block B ∈ F do
8:          for each P ∈ pred_B do
9:              for each v ∈ in_B \ out_P do
10:                 Place Reload(v) on control flow edge P → B
11:             end for
12:         end for
13:     end for
14: end function
```

`Alu` bzw. `Mul` entsprechen. An dieser Stelle wäre es möglich, mit Hilfe eines `And(Const, Const)` bzw. `Mul(Const, 1)` dafür zu sorgen, dass die Konstante in eine ALU bzw. die Multiplikationseinheit verschoben wird. Wenn aber alle Operanden in Funktionseinheiten sind, die keine Operation mit einem neutralen Element besitzen, so ist dies nicht möglich. Im allgemeinen Fall müssen solche ϕ-Funktionen also ausgelagert werden.

Es ist allerdings nicht ausreichend, jedes ϕ einzeln zu analysieren, sondern es muss immer eine ganze Klasse von ϕ's[2] untersucht werden. Das liegt daran, dass der Registerzuteiler erwartet, dass alle ϕ's einer Klasse die gleiche Registerklasse besitzen. Eine Klasse θ muss also genau dann ausgelagert werden, wenn es zwei Operanden in $Op(\theta)$ gibt, die einer unterschiedliche Registerklasse zugeordnet sind. Die Berechnung der ϕ-Klassen θ für eine Menge von ϕ-Knoten wird in Algorithmus 4 gezeigt. Die Auslagerung der betroffenen ϕ-Funktionen erfolgt als Vorphase vor der Auslagerung der anderen Werte.

Aufgrund der Nachkorrekturen in Beladys Algorithmus[3] können aber erneut solche problematischen ϕ's entstehen. Das geschieht immer dann, wenn mindestens ein Operand auf einem Kontrollflusspfad ausgelagert wird, während mindestens ein weiterer Operand durchgehend in Registern gehalten wird. Diese partiell ausgelagerten ϕ's lassen sich erst nach der Auslagerungphase erkennen.

Für das Problem gibt es mehrere mögliche Lösungen. Die einfachste ist, immer alle ϕ-Funktionen auszulagern. Damit sind keine Sonderbehandlungen notwendig, aber es entstehen mehr Auslagerungen als nötig, die das Programm verlangsamen. Eine andere Lösung ist, zunächst die Auslagerung zu simmulieren und noch keine endgültigen Aus- und Einlagerungsknoten zu platzieren. Anschließend kann man feststellen, welche ϕ-Funktionen partiell ausgelagert werden und sie zur kompletten Auslagerung vormerken. Dieser Simulationsprozess muss allerdings als Fixpunktiteration ausgeführt werden, denn durch die zusätzliche Auslagerung von ϕ-Funktionen können in der anschließenden Aus-

[2] siehe Definition 2.11
[3] Zeilen 7-12 in Algorithmus 6

Algorithmus 4 Berechnung von ϕ-Klassen

```
function CollectPhis(Node n, PhiClass θ)
    if PC(n) is defined then                          ▷ break recursion
        return
    end if
    for each p ∈ pred(n) do
        if p is φ then
            insert p into θ_n
            CollectPhis(p, θ_n)
        end if
    end for
    for each s ∈ succ(n) do
        if s is φ then
            insert s into θ_n
            CollectPhis(s, θ_n)
        end if
    end for
end function

function ComputePhiClasses(Set phi_nodes)
    R ← ∅                                             ▷ record all φ-classes
    for each n ∈ phi_nodes do
        if PC(n) is undefined then
            create new φ-class θ_n
            CollectPhis(n, θ_n)
            insert n into θ_n
            insert θ_n into R
        end if
    end for
    return R
end function
```

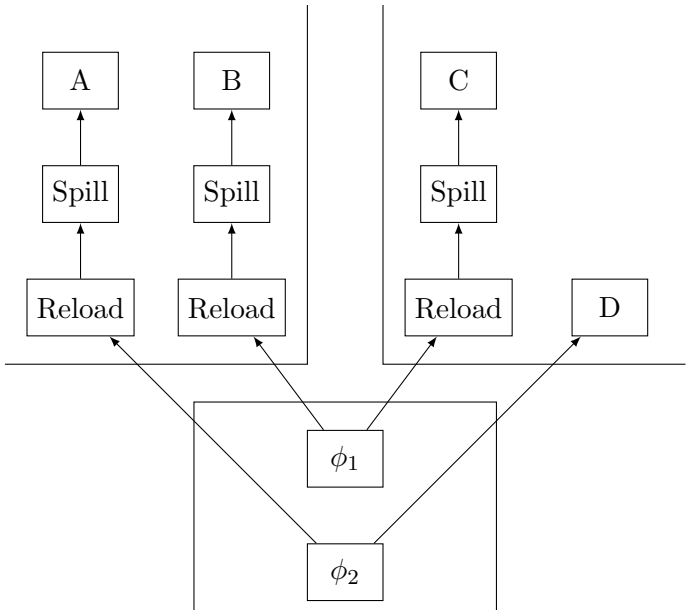

Abbildung 5.2: partiell ausgelagerte ϕ's nach der ersten Simulation

lagerungsphase andere partiell ausgelagerte ϕ's entstehen.

Nehmen wir folgendes Beispiel an: Die Knoten `A`, `B`, `C` und `D` seien Werte, denen die gleiche Registerklasse zugeordnet ist, z.B. `Alu`. Des Weiteren sind am Ende der Simulation Aus- und Einlagerung wie in Abbildung 5.2 platziert. Entsprechend der vorangegangenen Überlegungen wird ϕ_2 vollständig ausgelagert. Dies führt dazu, dass das Register, das von Knoten `D` belegt war, frei ist. Jetzt kann es passieren, dass der Auslagerungsalgorithmus den Wert `C` plötzlich in einem Register halten kann. Nach dem zweiten Auslagerungsschritt wäre ϕ_1 nun partiell ausgelagert, wie in Abbildung 5.3 zu sehen.

Die Simulation und Bestimmung der auszulagernden ϕ-Funktionen muss also so lange durchgeführt werden, bis sich nichts mehr ändert. Schlimmstenfalls sind dazu O(Anzahl der ϕ-Funktionen) Iterationen nötig. In der Praxis sind allerdings nicht mehr als ein bis zwei Durchläufe zu erwarten.

Nach der Behandlung der ϕ-Funktionen erfolgt der erste Durchlauf zur Auslagerung der Werte in den Registersatz. Als Registerklasse für die Einlagerungen wird die Klasse gewählt, die den Leseinheiten des Registersatzes zugeordnet ist. Danach werden die entstandenen Aus- und Einlagerungsknoten in Schreib- und Leseknoten umgewandelt, die auf dem Registersatz arbeiten. Die Werte der Schreibknoten haben die Registerklasse, die dem Registersatz entspricht, als Ergebnis. Anschließend erfolgt im zweiten Schritt die Auslagerung der Werte des Registersatzes. Die Einlagerung der Werte erfolgt in die Register der Ladeeinheit.

Zur Implementierung ist noch zu bemerken, dass die eigentliche Auslagerung als generisches Modul realisiert ist. Der konkrete Auslagerungsalgorithmus benachrichtigt dieses Modul über die vorgenommenen Einlagerungen. Daraufhin werden die notwendigen Aus- und Einlagerungsknoten erzeugt. Dabei kann statt

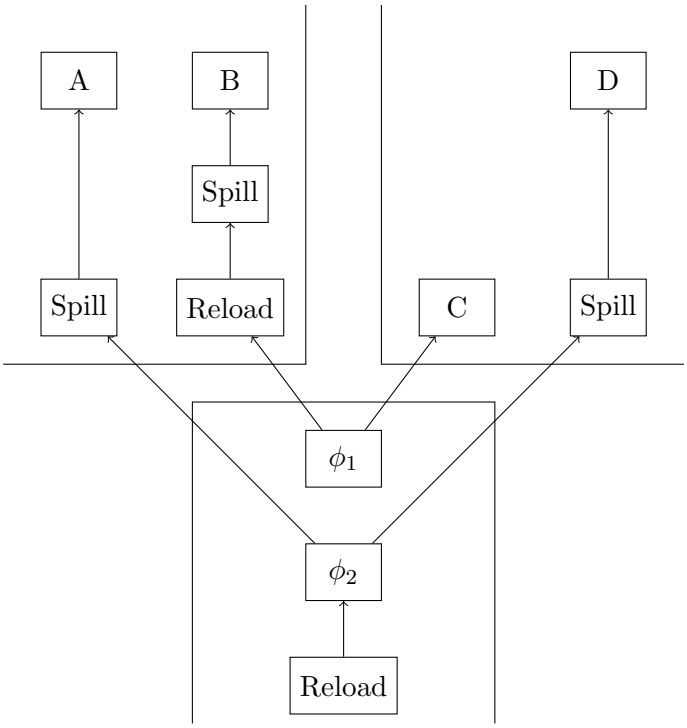

Abbildung 5.3: erneut partiell ausgelagerte ϕ's nach der zweiten Simulation

einer Einlagerung auch eine Rematerialisierung, d.h. eine Neuberechnung des Wertes erfolgen, falls z.B. alle Operanden der ursprünglichen Operation an der Einlagerungsstelle lebendig sind.

5.5 Registerzuteilung

Die Registerzuteilung ist bereits implementiert und muss nicht angepasst werden. Allerdings sind noch die Aus- und Einlagerungsknoten zu transformieren. Die ausgelagerten Werte werden im Funktionskeller gespeichert. Für den Zugriff auf einen solchen Wert ist die Adresse im Speicher explizit zu berechnen als `Kellerzeiger + Offset`. Da die Ladeeinheit keinerlei Adressberechnung unterstützt, ist eine zusätzliche ALU notwendig, die zu dem Zeitpunkt der Aus- oder Einlagerung unter Umständen nicht zur Verfügung steht. Aus diesem Grund werden zwei Register des Registersatzes ständig freigehalten, um gegebenfalls einen Wert aus einer ALU zwischenspeichern zu können. Das zweite Register ist notwendig, da bei der Wiederherstellung der Einheit, der Wert erst aus dem Registersatz gelesen wird und somit auch eine Leseeinheit freigemacht werden muss. In Abbildung 5.4 ist der ungünstigste Fall für eine Auslagerung sowie die notwendigen Transformationen gezeigt. Die Knoten `A` und `R` stellen die belegte ALU bzw. Leseeinheit dar und die gepunkteten Linien repräsentieren die Befehlsanordnungskanten.

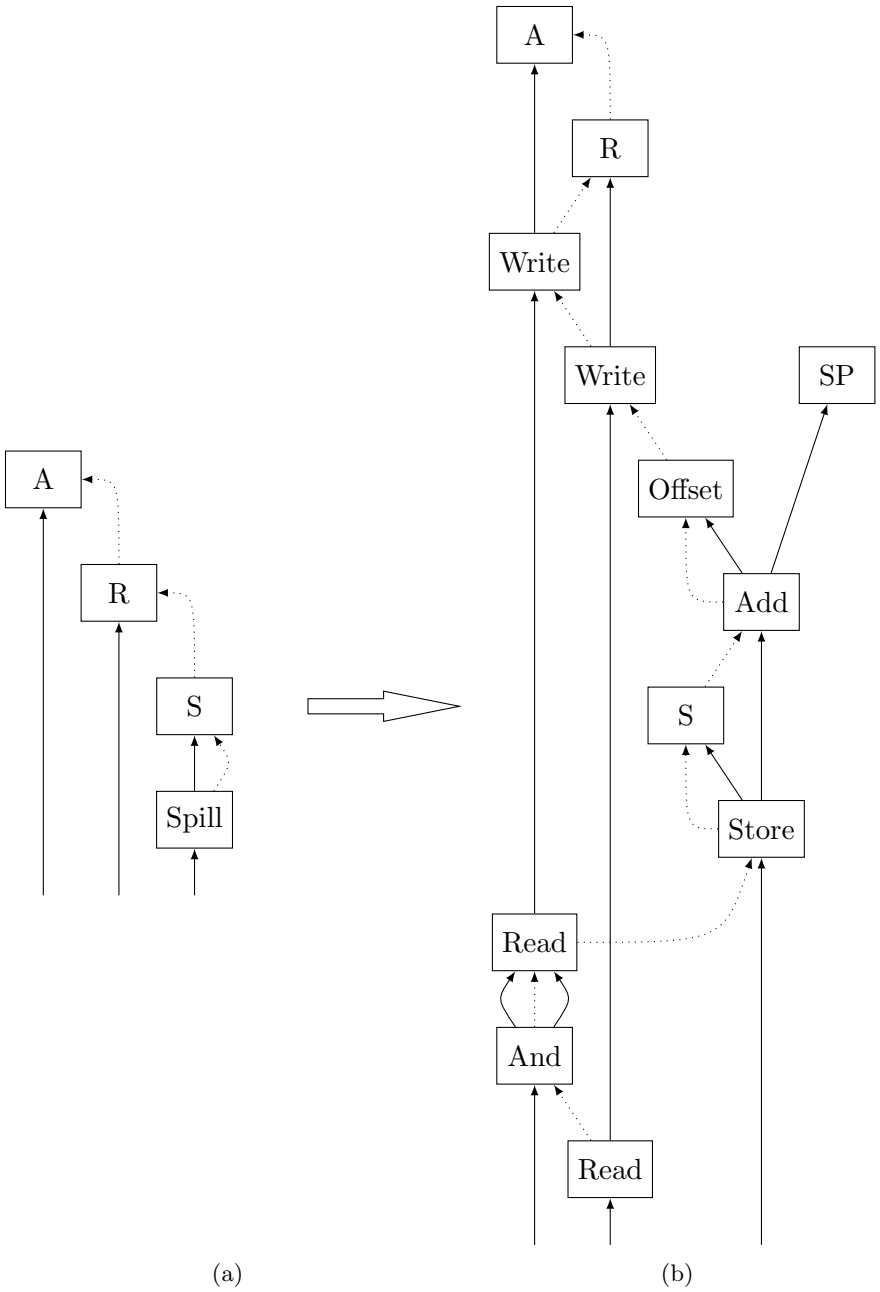

Abbildung 5.4: Ungünstiger Fall für eine Auslagerung (a) und der transformierte Graph (b)

5.6 Befehlsausgabe

Die Ausgabe der Befehle erfolgt als einfacher Lauf über den Graphen entlang der Anordnungskanten, bei dem für jeden besuchten Knoten der entsprechende Assemblerbefehl ausgegeben wird. Zusätzlich müssen noch die VLIW Bündel gebildet werden. Für den Assembler gehören alle Befehle, die direkt untereinander stehen zu einem Bündel. Getrennt werden die einzelnen Bündel durch Leerzeilen oder Sprungmarken.

Zu diesem Zweck wurde ein VLIW-Simulator implementiert, der vor jedem Befehl überprüft, ob dieser noch in das aktuelle Bündel passt. Es gibt drei Fälle, in denen der Simulator in den nächsten Takt weiterschalten, d.h. eine Leerzeile ausgeben muss:

1. Das Bündel hat bereits die maximale Größe von fünf Befehlen erreicht.
2. Die Einheit, auf der der Befehl ausgeführt wird, ist im aktuellen Bündel schon belegt.
3. Ein Operand des aktuellen Befehls ist noch nicht fertig mit der Berechnung.

Falls es einen Takt gibt, in dem kein Befehl ausgeführt werden kann, wird ein `NOP` ausgegeben. In Anhang A.2 ist der Pseudocode für den VLIW-Simulator angegeben.

6 Messungen

Zur Auswertung der verwendeten Verfahren wurden verschiedene Messungen an einer Reihe von Benchmarks durchgeführt. Zunächst erfolgt die Untersuchung der Befehlsanordnung in Bezug auf die Ersetzungsmuster. Anschließend werden die Verbesserungen der ILP-basierten Befehlsanordnung bei den Auslagerungen und der Ausnutzung der VLIW Bündel im Vergleich zur Heuristik dargestellt. Außerdem erfolgt eine Analyse der ausgelagerten ϕ-Funktionen. Dies dient der Evaluierung möglicher Verbesserungen des in Kapitel 5.4 beschriebenen Problems von ϕ-Funktionen mit Operanden aus unterschiedlichen Registerklassen. Abschließend erfolgen Vergleichsmessungen mit den Resultaten des Übersetzers der Firma Dresden Silicon.

6.1 Messumgebung und Benchmarks

Als ILP Löser kam die Software ILOG CPLEX 9.1 [cpl] auf einem PC mit einem 2.40GHz Intel Pentium 4 Prozessor und 1GB RAM zum Einsatz. Für die Vergleichsmessungen wurden die Benchmarks mit dem STAcc, einem CoSy-basierten Übersetzer, der von Dresden Silicon entwickelt wird, übersetzt. Die Laufzeitmessungen der übersetzten Programme erfolgte mit einem VDPFXT16 Simulator, der ebenfalls von Dresden Silicon zur Verfügung gestellt wurde. Folgende Bechmarks kamen zum Einsatz:

- **fft:** Eine ganzzahlige FFT Implementierung. Für die Messungen wurde eine FFT mit einem Fourier Fenster der Größe 128 berechnet.
- **hanoi:** Die Türme von Hanoi. Zur Messung wurden Türme der Größe 10 verschoben.
- **fannkuch:** Eine Fixpunktberechnung von Permutationen der Menge $\{0, \ldots, n\}$ (siehe auch [AR94]). Zur Messung wurde $n = 5$ gewählt.
- **quicksort:** Sortierung einer Zahlenreihe nach dem Quicksort Verfahren. Für die Messungen wurde eine zufällige Zahlenreihe der Länge 50 sortiert.
- **queens:** Ein Programm zur Berechnung des n-Damenproblems. Für die Messungen wurde das 8-Damenproblem berechnet.
- **sieve:** Berechnung von Primzahlen mit Hilfe des Siebs des Eratosthenes. Für die Messungen wurden die Primzahlen bis 100 bestimmt.

Die Implementierung wurde in zwei Optimierungsstufen gemessen:

- **O1:** Es kamen nur Standardoptimierungen, wie Konstantenfaltung, Eliminierung gemeinsamer Teilsausdrücke, Entfernung toten Codes, etc. zum Einsatz. Insbesondere erfolgte kein offener Einbau[1] von Funktionen. Da der STAcc das nicht beherrscht, dient diese Optimierungsstufe zum besseren Vergleich der Messergebnisse.

- **O3:** Es wurden alle Optimierungen, insbesondere auch der offene Einbau von Funktionen, aktiviert.

6.2 Befehlsanordnung

Der am häufigsten vorkommende Ersetzungstyp, mit einem Anteil von 93,5%, ist $1 \rightarrow 1$. Die restlichen 6,5% stellt der Typ $1 \rightarrow n$ dar, wobei in diesem Fall $n = 2$ gilt. Diese Ersetzungen entstehen bei Sprüngen, da hier neben dem Sprung noch ein zusätzlicher Befehl zum Laden des Sprungziels erzeugt werden muss. Der Typ $n \rightarrow 1$ trat nicht auf, weil keines der komplexen Muster, beschrieben in Kapitel 4.1, in einem der Benchmarks gefunden wurde. Für Ersetzungen des Typs $n \rightarrow m$ gibt es bei dem vorliegenden Prozessor keinen Anwendungsfall. Daraus ergibt sich, dass die Befehlsauswahl auf Firm für den VDSPFXT16 sehr einfach und effizient durchführbar ist.

6.3 Auswertung der ILP Modellierung

Zwei charakteristische Messgrößen für ein ILP Modell sind die Anzahlen der erzeugten Variablen und Bedingungen in Abhängigkeit der Problemgröße. In Abbildung 6.1 ist, zur besseren Veranschaulichung, neben diesen Werten auch der Graph der Funktion $f(x) = 2x^2$ dargestellt. Anhand der Daten lässt sich so für beide Kennzahlen ein quadratisches Verhältnis ablesen. Die Transformation des Anordnungsproblems in ein ILP ist mit der vorgestellten Modellierung demnach effizient durchführbar, zumal in der Praxis die maximale Größe der Grundblöcke auf 200-300 Instruktionen beschränkt ist.

Weiterhin wurde die, zur Lösung des ILPs benötigte, Laufzeit gemessen. Zur Bestimmung der Problemlösungsdauer kamen neben den bereits vorgestellten Programmen auch Teile der SPEC2000-Benchmarksammlung zum Einsatz. Für diese ließ sich zwar die Codeerzeugung durchführen, aber es konnten keine ausführbaren Programme erstellt werden, da sie extensiven Gebrauch von Bibliotheksaufrufen machen, welche der verwendete Simulator nicht unterstützt. In Abbildung 6.2 ist die Laufzeit des Problemlösers im Verhältnis zur Blockgröße aufgetragen. Da für die Zeit eine logarithmische Skala gewählt wurde, lässt sich eine exponentielle Abhängigkeit feststellen, wie es bei einem NP-vollständigen Problem zu erwarten war. Gut zu erkennen ist aber auch, dass die meisten Blöcke weniger als 25 Instruktionen besitzen und bei 96% aller Probleme die Lösung in unter einer Sekunde bestimmt werden konnte.

[1] *engl.* Inlining

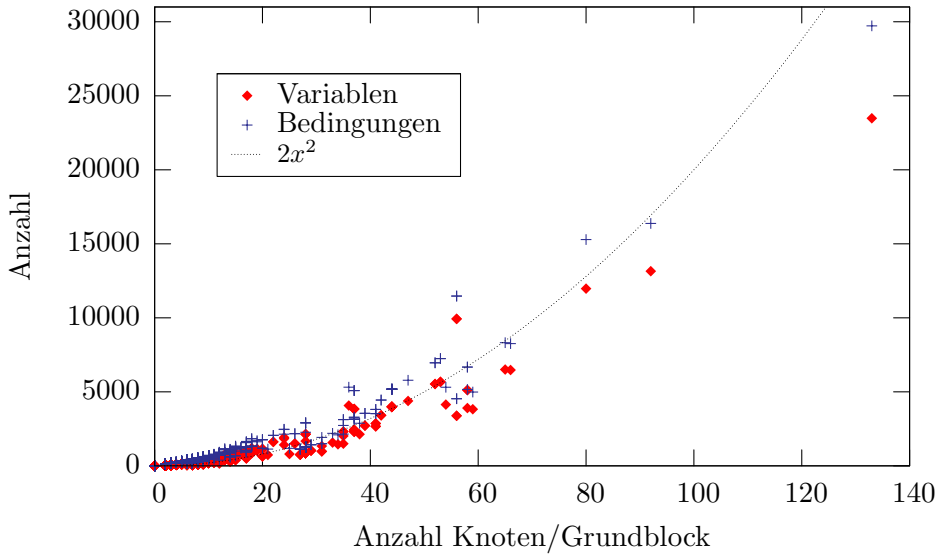

Abbildung 6.1: Anzahl der erzeugten ILP-Variablen und -Bedingungen in Abhängigkeit der Blockgröße mit eingeblendeter Referenzkurve

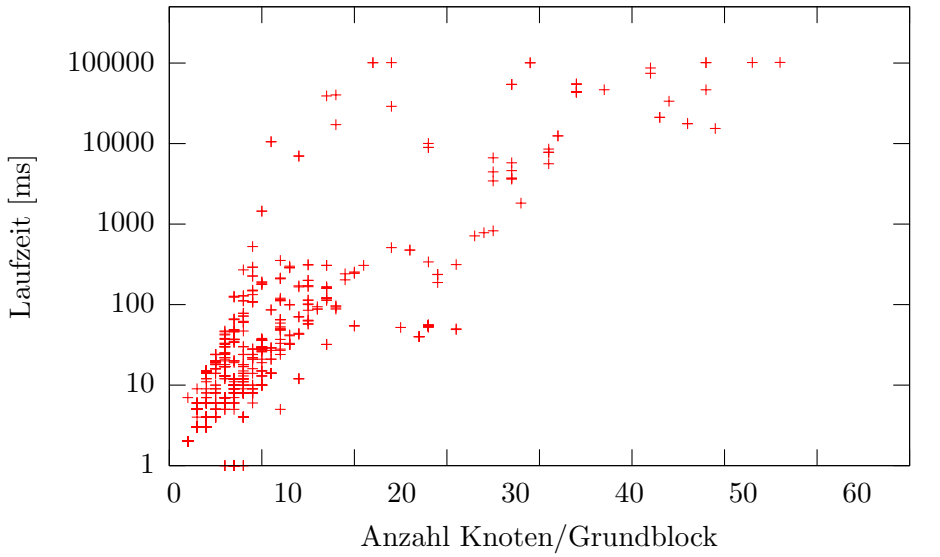

Abbildung 6.2: Laufzeit des ILP Lösers in Abhängigkeit der Blockgröße

Benchmark	Heuristik			ILP		
	Spills	Reloads	Remats	Spill	Reloads	Remats
fannkuch	22 (0)	58 (0)	30 (0)	22 (0)	59 (0)	27 (0)
fft	73 (13)	147 (19)	81 (0)	70 (12)	146 (18)	77 (0)
hanoi	13 (6)	19 (8)	24 (0)	13 (6)	19 (8)	20 (0)
queens	12 (0)	30 (0)	15 (0)	12 (0)	30 (0)	11 (0)
quicksort	27 (6)	59 (8)	35 (0)	27 (6)	59 (8)	24 (0)
sieve	7 (0)	14 (0)	16 (0)	7 (0)	14 (0)	16 (0)
Gesamt	154 (25)	327 (35)	201 (0)	151 (24)	327 (34)	175 (0)

Tabelle 6.1: Entstandene Aus- und Einlagerungen (in Klammern stehen die Werte für Aus- und Einlagerungen im Speicher)

Benchmark	Auslagerungen		
	Gesamt	ϕ-Funktionen	Anteil
fannkuch	22	10	45.5%
fft	70	34	48.6%
hanoi	13	3	23.1%
queens	12	4	33.3%
quicksort	27	9	33.3%
sieve	7	5	71.4%
Gesamt	151	65	43.0%

Tabelle 6.2: Verhältnis aller Auslagerungen zu den Auslagerungen von ϕ-Funktionen entstanden bei der ILP Anordnung

Für die Messungen wurden 100 Sekunden als obere Zeitschranke festgelegt, nach denen der ILP-Löser abbrach. Innerhalb dieser Zeit konnte für 98,8% aller Blöcke eine optimale Lösung bestimmt werden. Bei 0,5% wurde allerdings nicht einmal eine gültige Lösung gefunden, so dass auf das heuristische Anordnungsverfahren zurückgegriffen werden musste. Für die restlichen 0,7% betrug die mittlere Abweichung vom Optimum 10,6%.

Ein weiteres Gütekriterium für die Befehlsanordnung ist die Menge der entstehenden Aus- und Einlagerungen. In Tabelle 6.1 sind die Aus- und Einlagerungen sowie die Rematerialisierungen aufgeführt, die bei Verwendung der Heuristik und der ILP Lösung auftraten. Die Werte in Klammern repräsentieren jeweils den Teil der Operationen, die auf dem Speicher stattfinden. Offensichtlich werden bei der ILP Anordnung im Vergleich zur Heuristik kaum Befehle gespart. Das liegt vermutlich an dem, in Kapitel 5.4 beschriebenen, Problem der notwendigen Auslagerung von ϕ-Funktionen. Es werden so bei der ILP Lösung, trotz besserer Anordnung, Werte ausgelagert, die eigentlich in Registern gehalten werden könnten. Wie die Werte aus Tabelle 6.2 belegen, stellen sie mit 43% einen großen Anteil an den Auslagerungsbefehlen.

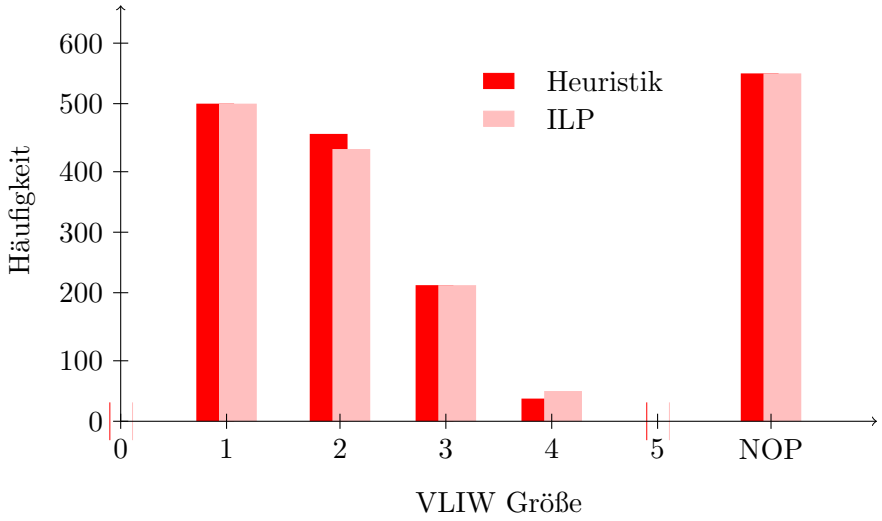

Abbildung 6.3: Verteilung der VLIW Bündelgrößen bei heuristischer und ILP Anordnung

Benchmark	**STAcc**	**Heuristik**		**ILP**	
	Takte	Takte	Verb. um	Takte	Verb. um
fannkuch	40317	33711	16.4%	33614	16.6%
fft	418477	248537	40.6%	246453	41.1%
hanoi	261054	119810	54.1%	115717	55.7%
queens	5655153	3371467	40.4%	3331911	41.1%
quicksort	21125	11814	44.1%	11459	45.8%
sieve	7316	4817	34.2%	4772	34.8%
Durchschnitt	100.0%		38.3%		39.2%

Tabelle 6.3: Laufzeitmessungen bei Optimierungsstufe O1

Benchmark	**STAcc**	**Heuristik**		**ILP**	
	Takte	Takte	Verb. um	Takte	Verb. um
fannkuch	40317	33250	17.5%	33053	18.0%
fft	418477	149921	64.2%	149611	64.2%
hanoi	261054	119794	54.1%	115702	55.7%
queens	5655153	1693428	70.1%	1686736	70.2%
quicksort	21125	11840	44.0%	11485	45.6%
sieve	7316	4795	34.5%	4751	35.1%
Durchschnitt	100.0%		47.4%		48.1%

Tabelle 6.4: Laufzeitmessungen bei Optimierungsstufe O3

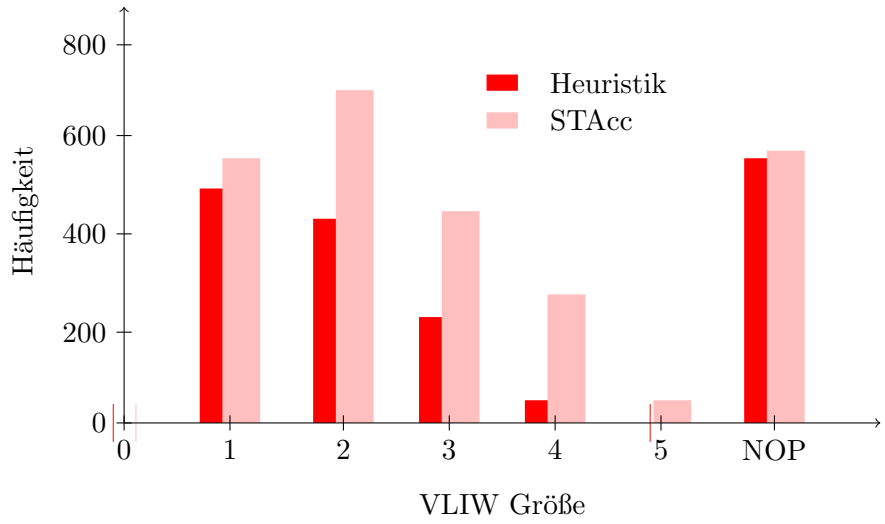

Abbildung 6.4: Verteilung der VLIW Bündelgrößen bei heuristischer Anordnung mit im Vergleich zu STAcc

Neben der Verbesserung des Registerdrucks zur Auslagerungsreduzierung war die Maximierung der VLIW Bündelauslastung eine weiteres Ziel der ILP Formulierung. Abbildung 6.3 zeigt, dass im Vergleich zur Heuristik mehr Bündel der Größe 4 und weniger der Größe 2 entstehen. Die durchschnittliche Bündelgröße steigt bei der ILP Anordnung von 1,8 auf 1,85. Auch hier spielt das Einfügen von Aus- und Einlagerungsbefehlen eine große Rolle, so dass sich mittels einer nachträglichen Befehlsanordnung, unter Beachtung der zugeteilten Register, noch einige Verbesserungen erzielen lassen.

6.4 Vergleichmessungen

Die Tabellen 6.3 und 6.4 zeigen die zur Ausführung der jeweiligen Programme benötigten Taktzyklen. Selbst auf der einfachsten Optimierungsstufe liegen die resultierenden Laufzeiten mehr als 38% unter denen des STAcc. Die Aktivierung aller Optimierungen bringt noch einmal ca. 9% Laufzeitgewinn. Das ist insbesondere auf den offenen Einbau von Funktionen zurückzuführen, denn ein Funktionsaufruf ist eine sehr teure Operation, da alle belegten Funktionseinheiten ausgelagert werden müssen.

Im Vergleich der Programmlaufzeiten bei heuristischer und ILP-basierter Anordnung ist festzustellen, dass die optimale Anordnung im Schnitt eine um ca. 0,9% bessere Laufzeit erzielt, maximal sind 1,7% zu verzeichnen. Die Heuristik ist demzufolge als sehr gut einzuschätzen, insbesondere, da sie die Anordung in wesentlich kürzerer Zeit durchführt als die ILP Modellierung.

Wie in Abbildung 6.4 zu sehen, erzeugt der STAcc mit einem Durchschnitt von 2,27 etwas größere Bündel als der Firm-Übersetzer. Allerdings enthalten die vom STAcc produzierten Programme im Schnitt 52% mehr Befehle. Ein Histogramm über die Verwendung der Funktionseinheiten ist in Abbildung 6.5

gegeben. Eines der häufigsten Befehlsmuster in den STAcc Programmen ist:

- Lade Konstante
- Führe Operation mit Konstante aus
- Lese/Schreibe Wert aus/in Registersatz
- Lese/Schreibe Wert aus/in Registersatz

Dadurch erklärt sich der überproportional hohe Anteil an Bündeln der Größe 4 im Vergleich zum Firm-Übersetzer.

Abbildung 6.6 veranschaulicht die Benutzung der verschiedenen Funktionseinheiten bei den verschiedene Optimierungstufen. Es wurde vermutet, dass durch den offenen Einbau von Funktionen die Codegröße ansteigt, was sich aber nicht bestätigt hat. Durch den Einsatz aller Optimierungen reduziert sich die Anzahl der erzeugten Befehle um etwa 7,2%. Der Anteil der `NOP` Operationen beträgt bei den STAcc Programmen 22,19% und bei den Firm Programmen 32,11%.

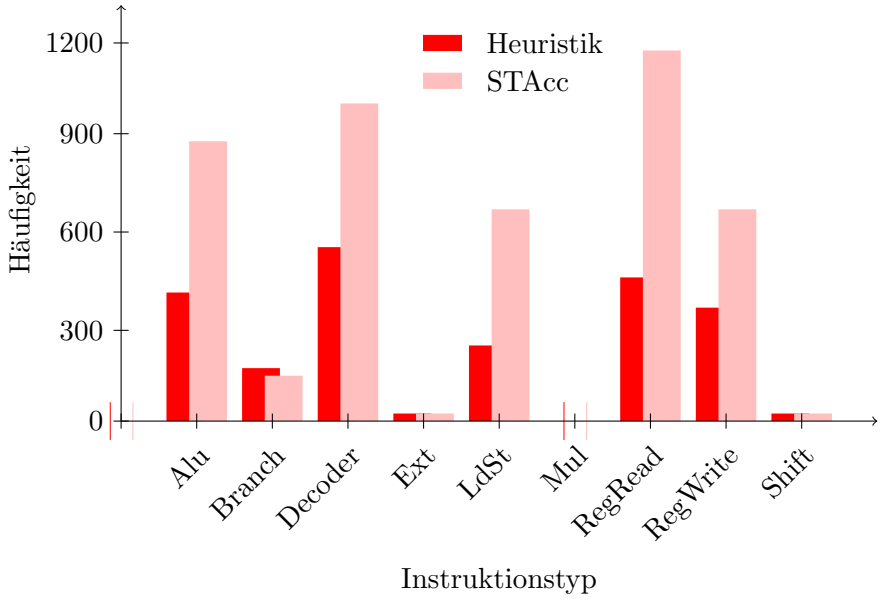

Abbildung 6.5: Verteilung der benutzten Funktionseinheitstypen bei heuristischer Anordnung im Vergleich zu STAcc

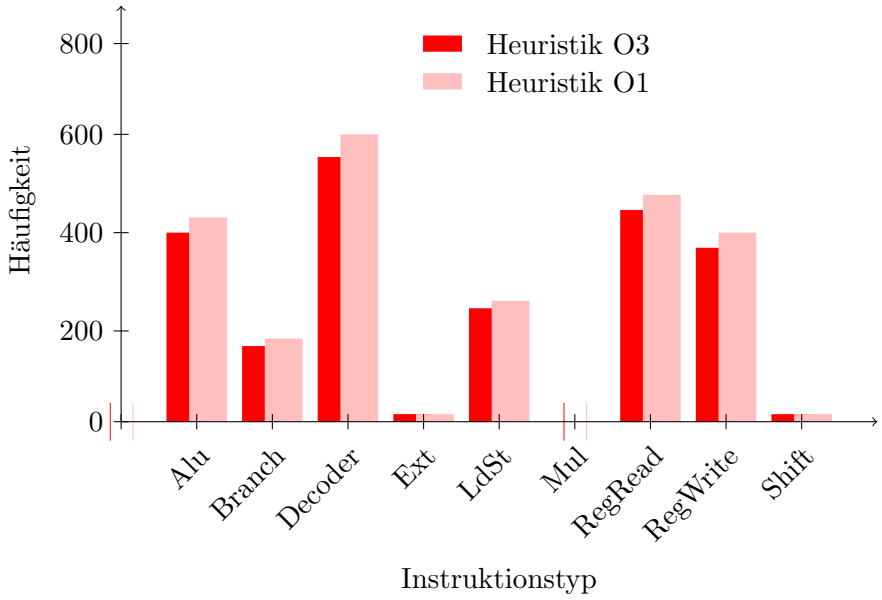

Abbildung 6.6: Verteilung der benutzten Funktionseinheitstypen bei heuristischer Anordnung mit verschiedenen Optimierungsstufen

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde der Entwurf und die Implementierung eines Backend für einen STA basierten Prozessor vorgestellt. Für die einzelnen Phasen des Backends wurde untersucht, welche Probleme sich speziell aufgrund der verwendeten Architektur ergaben und es wurden entsprechende Lösungsansätze vorgestellt und implementiert.

Die Befehlsauswahl wurde als einfaches Makrosubstitutionsverfahren realisiert, da sich im Vergleich zu kostengesteuerten Term- und Graphersetzungsverfahren keine wesentlichen Nachteile ergaben und die Laufzeit deutlich geringer ist. Des Weiteren wurde ein ILP Modell für die Befehlsanordnung vorgestellt, wobei sich herausgestellt hat, dass aufgrund der Verwendung von Rematerialisierungen in der Auslagerungsphase der Vorteil der besseren Anordnung weitaus geringer ausfällt, als zu erwarten. Für die Registerzuteilung wurde eine Abbildung des Hardwaredesigns auf die Registeranforderungen beschrieben sowie die daraus resultierenden Probleme und ihre Lösung.

Zusammenfassend konnte gezeigt werden, dass sich Firm aufgrund seiner Ähnlichkeit zu STA sehr zur Codegenerierung für den verwendeten Prozessor eignet und deutlich bessere Codequalität liefert als die bisherige Implementierung mittels CoSy.

7.1 Ausblick

Im Folgenden wird gezeigt, an welchen Stellen die implementierten Verfahren noch Verbesserungspotential besitzen und wie sie in zukünftigen Arbeiten noch erweitert werden können.

7.1.1 Prozessordesign

Eine wesentliche Verbesserung in der erzeugten Codequalität ließe sich mit Sicherheit dadurch erreichen, dass die Lade- und Speichereinheit einen zusätzliche Eingang bekommt, in dem ein Offset zur Basisadresse angegeben werden kann. Dies spart einerseits eine ALU bei Zugriffen auf Strukturen und Reihungen mit konstantem Index und andererseits erfordern Auslagerungen in den Speicher keine zusätzlichen Einheiten mehr. Damit hat man auch die zwei ständig frei gehaltenen Register als zusätzliche schnelle Auslagerungsspeicher zur Verfügung (siehe auch Kapitel 5.5). Zwar dauert dann ein Speicherzugriff aufgrund der zusätzlichen internen Addition unter Umständen ein Takt länger, aber die Vorteile überwiegen stark.

Ein weiteres Problem ist, dass der Schachtel- und Basiszeiger in Registern des Registersatzes gehalten werden. Daher müssen sie bei einem Zugriff jedes Mal geladen und bei Veränderungen zurückgeschrieben werden. Dies könnte man

durch eine Unterstützung in Form eines Akkumulators, der nur Addieren und Subtrahieren kann, mit zwei Ausgängen verbessern. Schachtel- und Basiszeiger können so ständig für Zugriffe bereitgehalten werden.

Eine dritte Verbesserungsmöglichkeit ist das Hinzufügen von Kopieroperationen zu jeder Einheit. Diese Operation bewirkt, dass der erste Operand einer Einheit, in deren Ausgäng kopiert wird. Damit könnt man dann alle Register in einer Klasse zusammenfassen und muss auch die ϕ-Funktionen, die Werte von unterschiedlichen Funktionseinheitstypen auswählen nicht mehr extra behandeln. Man spart dadurch auch zahlreiche Graphdurchläufe für den Registerzuteiler und die Extrabehandlungen. Die Kopien lassen sich im Idealfall als Operation mit null Takten Latenz realisieren, so dass sie, abgesehen von einem Platz in einem VLIW Bündel und zusätzlichem Hardwareaufwand, nichts kosten würden.

7.1.2 Befehlsanordnung

Erweiterung des ILP Modells

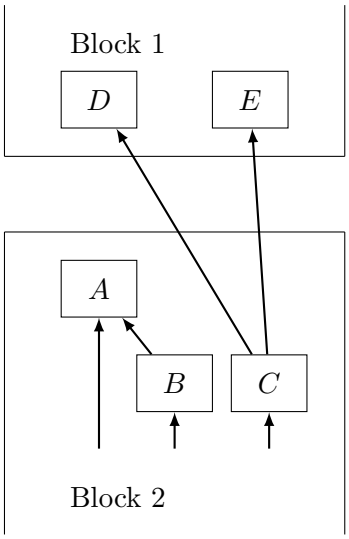

Abbildung 7.1: Problematischer Fall aufgrund der Nichtbeachtung hineinlebender Werte

Die aktuelle ILP Modellierung liefert bereits gute Ergebnisse, wie anhand der Messergebnisse in Kapitel 6 zu sehen ist. Es gibt allerdings noch Punkte, in denen das Modell verbessert werden kann.

Zum Einen wird lediglich in Funktionseinheitstypen unterschieden. Besser wäre eine Differenzierung in die einzelnen Funktionseinheiten. Damit könnte man sich die Registerzuteilung sparen, denn die ILP Lösung würde bereits die Information liefern, welcher Knoten auf welcher Funktionseinheit ausgeführt wird. Außerdem ist nur so die korrekte Modellierung von Befehlseinschränkungen möglich, die eine Anweisung auf eine bestimmte Funktionseinheit festlegen. So wird z.B. auf dem Pentium4$^{\text{TM}}$ eine arithmetische Negation nur auf

der ersten von zwei ALUs ausgeführt. In [KW01] wird vorgeschlagen, dies mit zusätzlichen Bedingungen für die Ressourcenbeschränkung abzubilden, indem Ungleichungen folgender Art für alle betroffenen Knoten $n \in V' \subseteq V$ eingeführt werden:

$$\forall k \in R, \forall 1 \leq t \leq U : \sum_{n \in V' : k \in K(n)} x_{nt}^{k} \leq 1$$

Dies führt aber dazu, dass zu dem Zeitpunkt t, wenn so ein Knoten angeordnet wird, kein weiterer Knoten ausgeführt werden kann, wodurch Optimierungspotential verschenkt wird. In der konkreten STA Implementierung gibt es solche Einschränkungen zwar nicht, aber da das ILP Verfahren generisch umgesetzt wurde, und somit auch von anderen Backends genutzt werden kann, müssen diese Fälle trotzdem betrachtet werden. Der Nachteil der detaillierteren Modellierung ist, dass sich die Anzahl der Variablen erhöht – bei der ST-Architektur würde sie sich z.B. mehr als Verdoppeln.

Ein anderes Problem ist, dass in den Grundblock hineinlebende Werte nicht bei der Kostenmodellierung des Registerdrucks beachtet werden. Ein Wert wird als in einen Grundblock hineinlebend bezeichnet, wenn er zu Beginn des Blocks lebendig ist und innhalb des Blocks verwendet wird. In Abbildung 7.1 ist ein Fall zu sehen, in dem es zu einer ungünstigen Befehlsanordnung kommen kann. Angenommen, die Werte A bis D werden alle auf dem gleichen Funktionseinheitstyp berechnet und es gibt nur zwei dieser Einheiten. Des Weiteren haben D und E keine weiteren Verwender und die kürzeste Ausführungsreihenfolge sei (A, B, C). Der ILP Löser wird diese Reihenfolge als Lösung wählen, da er nicht erkennt, dass D und E Ressourcen belegen. Somit werden zwei Auslagerungen (A und B) benötigt, anstatt nur einer (A oder B). Die beste Lösung in diesem Beispiel, bezogen auf die Gesamtkosten, ist (C, A, B).

Die dritte Erweiterungsmöglichkeit des Modells besteht in der Ausnutzung der Branch-Delay-Slots. Prinzipiell ist dies sehr einfach möglich, indem zugelassen wird, dass Sprünge entsprechend viele Takte vor Ende des Blocks angeordnet werden dürfen. Allerdings ist nicht sichergestellt, dass die in diesen Takten angeordneten Befehle nicht ausgelagert werden. Falls es zu einer Auslagerung kommt, stehen unter Umständen nicht genügend Takte zur Verfügung, um sie korrekt auszuführen.

Sonstige Erweiterungen

Durch das Einfügen von Aus- und Einlagerungsbefehlen wird die ursprüngliche Befehlsanordnung verändert. Das wirkt sich insbesondere beim Auslagern in den Speicher auf die Zusammensetzung der VLIW Bündel aus. Mittels einer nachträglichen Befehlsanordnung ließe sich dieser negative Effekt vermutlich etwas kompensieren. Zudem könnte man an dieser Stelle die Unterstützung für die Branch-Delay-Slots realisieren, da die Register bereits zugeteilt sind und die Gefahr der Auslagerung eines Befehls in einem dieser Takte nicht mehr besteht.

7.1.3 Auslagerung

Ein weiterer Ansatz besteht darin, die Befehlsanordnung und das Auslagern in einem ILP Modell zu verbinden. Szalkowski stellt in seiner Arbeit [Sza06] eine Variante zur Auslagerung mit Rematerialisierungen als ganzzahlige lineare Optimierung vor. Die Integration der Befehlsanordnung in dieses Verfahren würde vermutlich zu deutlich besserem Auslagerungsverhalten führen, da diese beiden Phasen sehr stark voneinander abhängen.

7.1.4 Vektorisierung

Das volle Potential des Prozessors kann nur bei voller Unterstützung der Vektoreinheiten ausgenutzt werden. Aktuell wird im Rahmen der Diplomarbeit von Andreas Schösser [Sch07] ein Verfahren entwickelt, wie sich aus einer Hochsprachenbeschreibung Transformationsregeln für ein Graphersetzungssystem generieren lassen. Dabei wird unter Anderem auch der Anwendungsfall der automatischen Vektorisierung untersucht. Die Integration dieses Verfahrens in das Backend würde sicher zu einer weiteren Verbesserung des erzeugten Codes beitragen.

A Anhang

A.1 Beladys Algorithmus

Algorithmus 5 Beladys Auslagerungsalgorithmus für einen Grundblock

```
1:  function DISPLACE(Set Worklist, Set Values, Registerclass class)
2:      T ← Worklist ∪ Values
3:      Sort T ascending by next-use distances
4:      Worklist ← first AvailableRegister(class) entries from T
5:      return T \ Worklist                          ▷ return all displaced values
6:  end function
7:
8:  function SPILLBLOCKBELADY(Basic block B, Registerclass class)
9:      T ← livein_{B,class} ∪ Φ_{B,class}
10:     Sort T ascending by next-use distances
11:     W ← first AvailableRegister(class) entries from T
12:     in_{B,class} ← W          ▷ record all values being in registers at begin of B
13:     U ← ∅                                  ▷ U contains all values used within B
14:     insn ← sched_first_B
15:     while insn is not sched_end_B do
16:         if insn is not φ then
17:             for each v ∈ Uses_class(insn) \ W do
18:                 Place Reload(v) before insn
19:             end for
20:             X ← DISPLACE(W, insn, Uses_class(insn), class)
21:             ▷ Remove all values being diplaced before first usage from in_B
22:             in_{B,class} ← in_{B,class} \ {X \ U}
23:             U ← U ∪ Uses_class(insn)
24:             DISPLACE(W, insn, Defs_class(insn), class)
25:         end if
26:         insn ← sched_next_B(insn)
27:     end while
28:     out_B ← W
29: end function
```

Algorithmus 6 Beladys Auslagerungsalgorithmus

```
1:  function SpillBelady(Function F, Registerclass class)
2:      ▷ spill all basic blocks
3:      for each basic block B ∈ F do
4:          SpillBlockBelady(B, class)
5:      end for
6:      ▷ ensure all needed values are in start workset
7:      for each basic block B ∈ F do
8:          for each P ∈ pred_B do
9:              for each v ∈ in_{B,class} \ out_{P,class} do
10:                 Place Reload(v) on control flow edge P → B
11:             end for
12:         end for
13:     end for
14: end function
```

A.2 Der VLIW Simulator

Algorithmus 7 VLIW Simulator

```
1: function SIMULATE(Instruction insn, State VLIWState)
2:     R ← Ressource(insn)
3:     nextstep ← false
4:     if not RessourceIsFree(R, VLIWState) then
5:         nextstep ← true
6:     else if InstructionNumber(VLIWState) ≥ maxInstruction then
7:         nextstep ← true
8:     else
9:         for each p ∈ pred(insn) do
10:            if Step(p) + Latency(p) > CurrentStep(VLIWState) then
11:                nextstep ← true
12:                break
13:            end if
14:        end for
15:    end if
16:    if nextstep = true then
17:        ▷ advance to next step
18:        increment CurrentStep(VLIWState)
19:        InstructionNumber(VLIWState) ← 0
20:        FreeAllRessources(VLIWState)
21:        return false
22:    else
23:        ▷ insn will be emitted here, update VLIWState
24:        increment InstructionNumber(VLIWState)
25:        AcquireRessource(VLIWState, R)
26:        Step(insn) ← CurrentStep(VLIWState)
27:        return true
28:    end if
29: end function
30:
31: function EMIT(Instruction insn, State VLIWState)
32:    repeat
33:        res ← SIMULATE(insn, VLIWState)
34:    until res = true
35:    emit code for insn
36: end function
```

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[AAvS94] Martin Alt, Uwe Aßmann, and Hans van Someren. Cosy Compiler Phase Embedding with the CoSy Compiler Model. In *CC '94: Proceedings of the 5th International Conference on Compiler Construction*, pages 278–293, London, UK, 1994. Springer-Verlag.

[AR94] Kenneth R. Anderson and Duane Rettig. Performing Lisp Analysis of the FANNKUCH Benchmark. *SIGPLAN Lisp Pointers*, VII(4):2–12, 1994.

[Bel66] Laszlo A. Belady. A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[Boe98] Boris Boesler. Codeerzeugung aus Abhängigkeitsgraphen. Master's thesis, Universität Karlsruhe (TH), IPD, June 1998. available at http://www.info.uni-karlsruhe.de/~boesler/thesis.ps.gz.

[Boe03] Boris Boesler. A Modification to BURS in Codegeneration. Technical Report 2003-12, Universität Karlsruhe (TH), Fakultät für Informatik, 6 2003. available at http://www.info.uni-karlsruhe.de/~boesler/papers/burs-tech-report.pdf.

[Boe05] Boris Boesler. *Codeerzeugung mit Graphersetzung und Lösungsgraphen*. PhD thesis, Universität Karlsruhe, Aachen, Februar 2005.

[CFR+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, New York, NY, USA, 1989. ACM Press.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101, New York, NY, USA, 1982. ACM Press.

[Cli95] Clifford Click. *Combining analyses, combining optimizations*. PhD thesis, Rice University, 1995. available at http://citeseer.comp.nus.edu.sg/article/click95combining.html.

[Cor98] Henk Corporaal. *Microprocessor architectures from VLIW to TTA*. Wiley, 1998.

[cos] CoSy: A compiler development system. http://www.ace.nl.

[COS03] CoSy Compilers: Overview of Construction and Operation, April 2003. CoSy System Documentation, available at http://cosy.biz/compiler/paper-construct.pdf.

[cpl] ILOG CPLEX: Mathematical Programming Optimizers. http://www.cplex.com.

[CRS+04] G. Cichon, P. Robelly, H. Seidel, E. Matus, M. Bronzel, and G. Fettweis. Synchronous Transfer Architecture (STA). In *Proceedings of the 4th International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'04)*, pages 126–130, July 2004. available at http://wwwmns.ifn.et.tu-dresden.de/publications/2004/Cichon_G_SAMOS_04.pdf.

[EKS03] Erik Eckstein, Oliver König, and Bernhard Scholz. Code Instruction Selection based on SSA-Graphs. In *SCOPES*, pages 49–65, 2003.

[Emm94] Helmut Emmelmann. *Codeselektion mit regulär gesteuerter Termersetzung*. Oldenbourg, 1994. ISBN 3-486-23252-5.

[Fis81] Joseph A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Computers*, 30(7):478–490, 1981.

[Fis83] Joseph A. Fisher. Very Long Instruction Word architectures and the ELI-512. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 140–150, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press.

[GW84] Gerhard Goos and William Waite. *Compiler Construction*. Springer, Jan 1984. available at ftp://i44ftp.info.uni-karlsruhe.de/public_html/papers/ggoos/CompilerConstruction.ps.gz.

[HGG06] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In Alan Mycroft Andreas Zeller, editor, *Compiler Construction 2006*. Springer, March 2006. to appear.

[Jak04] Hannes Jakschitsch. Befehlsauswahl auf SSA-Graphen. Master's thesis, IPD Goos, November 2004. available at http://www.info.uni-karlsruhe.de/papers/da_jakschitsch.pdf.

[JL80] H.-St. Jansohn and R. Landwehr. CGSS: Ein System zur automatischen Erzeugung von Code-Generatoren. Master's thesis, Institut für Informatik II, 1980.

[KW01] Daniel Kästner and Sebastian Winkel. ILP-based Instruction Scheduling for IA-64. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 145–154, New York, NY, USA, 2001. ACM Press.

[LBBG05] Götz Lindenmaier, Michael Beck, Boris Boesler, and Rubino Geiß. Firm, an Intermediate Language for Compiler Research. Technical Report 2005-8, Dept. of Computer Science, University of Karlsruhe (TH), March 2005. available at http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=ira/2005/8.

[Mül95] Thomas Müller. *Effiziente Verfahren zur Befehlsanordnung*. PhD thesis, IPD Goos, 1995.

[Sch07] Andreas Schösser. Graphersetzungsregelgewinnung aus Hochsprachen und ihre Anwendung. Master's thesis, 9 2007. to appear.

[Sza06] Adam M. Szalkowski. Rematerialisierung mittels ganzzahliger linearer Optimierung in einem SSA-basierten Registerzuteiler. Master's thesis, 9 2006. available at http://www.info.uni-karlsruhe.de/papers/da_szalkowski.pdf.

Index