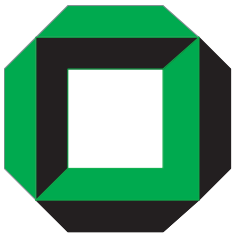

Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

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

Heuristisches Auslagern in einem SSA-basierten Registerzuteiler

Diplomarbeit
Matthias Braun

Oktober 2006

Betreuer:
Dipl.-Inform. Sebastian Hack

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

Ein elementarer Bestandteil eines Übersetzers ist die Registerzuteilung. Für Programme in SSA-Form lässt sich Registerzuteilung in drei unabhängige Phasen aufspalten: Auslagern, Färbung des Graphen und Kopienminimierung.

Auslagern erfolgt hierbei unabhängig vom Interfernzgraph und der Graphfärbung. Dies ermöglicht es Kontext und Struktur des Programms zu berücksichtigen. In dieser Arbeit werden die Vorteile dieses Ansatzes untersucht und effiziente heuristische Algorithmen für lokales und globales Auslagern präsentiert. Die bei der Implementierung gewonnenen Erfahrungen werden zusammengefasst.

Abschliessende Messungen zeigen, dass die Verfahren praktikabel und konkurrenzfähig sind.

Danksagung

Hiermit möchte ich mich bei allen Mitarbeitern des IPD für die freundliche und produktive Athmosphäre bedanken. Ein besonderer Dank geht an Sebastian Hack für seine fachkundige und engagierte Betreuung. Mein Dank gilt auch Martin Trapp, Götz Lindenmaier, Michael Beck, Rubino Geiß, Christian Würdig, Adam Szalkowski, Christoph Mallon und allen anderen die an der Entstehung von Firm beteiligt waren. Ein Dank gilt auch all den Autoren der Open-Source Software, die das Arbeiten an dieser Arbeit einfacher gemacht haben.

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Aufgabenstellung . . . 2
- 1.2 Aufbau der Arbeit . . . 2

2 Grundlagen **3**
- 2.1 Zwischensprachen . . . 3
- 2.2 Steuerflussgraph . . . 3
- 2.3 Dominanzrelation . . . 4
- 2.4 Lebendigkeit . . . 4
- 2.5 Schleifenbaum . . . 5
- 2.6 Ausführungshäufigkeiten . . . 6
- 2.7 Registerzuteilung . . . 7
- 2.8 SSA-Form . . . 7
  - 2.8.1 Semantik der Φ-Funktion . . . 8
- 2.9 Registerzuteilung auf SSA-Form . . . 9

3 Verwandte Arbeiten **11**
- 3.1 Registerzuteilung durch Graphfärben . . . 11
- 3.2 Registerzuteilung durch priorisiertes Färben . . . 12
- 3.3 Lokale Registerzuteilung . . . 13
- 3.4 Auslagern auf SSA-Form . . . 13
- 3.5 GCC . . . 14
  - 3.5.1 Lokale Zuteilung . . . 14
  - 3.5.2 Globale Zuteilung . . . 15
  - 3.5.3 Auslagern und Registerbeschränkungen . . . 15

4 Lösungsansatz **17**
- 4.1 Kostenmodell . . . 17
- 4.2 Auslagern von Φ-Funktionen . . . 17
- 4.3 SSA Wiederaufbau . . . 19
- 4.4 Allgemeine Überlegungen . . . 19
- 4.5 Der Algorithmus von Belady . . . 20

4.5.1 Nächste Verwendungen 21
4.5.2 Problemfälle 21
4.6 Der Algorithmus von Morgan 22
4.7 Werte wiederberechnen 23
4.8 Speicherkopienminimierung 24

5 Implementierung **27**
5.1 Die Zwischensprache Firm 27
5.2 Struktur des Backends 29
5.3 Auslagern und SSA-Wiederaufbau 30
5.3.1 Implementierung von Speicherkopien 31
5.4 Der Algorithmus von Belady 31
5.4.1 Nächste Verwendungen 33
5.5 Der Algorithmus von Morgan 33
5.6 Minimierung von Speicherkopien 35

6 Messungen **39**
6.1 Messumgebung 39
6.1.1 Benchmarks 39
6.1.2 Konfigurationen 40
6.2 Übersetzungszeit 40
6.3 Kosten 41
6.4 Auslagerungsplätze Verschmelzen 42
6.5 Rematerialisierung 42
6.6 Speicherzugriffe und Instruktionen 44
6.7 Laufzeiten 45
6.8 Vergleich mit weiteren Übersetzern 45

7 Zusammenfassung und Ausblick **49**
7.1 Zusammenfassung 49
7.2 Ausblick 49
7.2.1 Φ-Kaskaden 49
7.2.2 Geschicktes Graphenclustern 51
7.2.3 Rematerialisierungen 51
7.2.4 Nächste Verwendungen 51

1 Einführung

Die Unterschiede in den Latenzzeiten zwischen Prozessor und Hauptspeicher nehmen ständig zu. Während man exponentielle Leistungssteigerungen bei der Entwicklung von Prozessoren beobachten kann, wächst die Geschwindigkeit des Hauptspeichers nur linear. Auf heute gebräuchlichen Pentium 4 oder Athlon Prozessoren benötigt ein Zugriff auf den Speicher etwa 5 Taktzyklen mehr als ein direkter Registerzugriff.[1] Die Vermeidung von Speicherzugriffen ist daher eines des wichtigsten Ziele bei der Optimierung von Programmen.

Die meisten Programme benötigen mehr Speicherstellen für Zwischenergebnisse als Register vorhanden sind. Die Registerzuteilung eines Übersetzers muss daher viele dieser Werte in den Hauptspeicher auslagern, was zusätzliche Speicherzugriffe zur Folge hat. Eine geschickte Wahl der auszulagernden Werte und eine Platzierung der Ein- und Auslagerungsbefehle an selten ausgeführten Programmstellen vermeidet viele dieser Zugriffe. Gutes Auslagern wirkt sich deshalb direkt auf die Geschwindigkeit der übersetzten Programme aus und bestimmt die Qualität einer Registerzuteilung.

Das verbreitete Verfahren von Chaitin/Briggs baut einen Interferenzgraph der Werte eines Programms auf und erzeugt eine Registerzuteilung durch Färbung dieses Graphen. Erst wenn die Färbung nicht möglich ist werden Werte ausgelagert und eine erneute Färbung versucht. Auslagern ist hierbei eng mit der Färbung des Graphen verwoben und erfolgt nur als Reaktion auf ein fehlgeschlagenes Zuteilen. Wie Hack, Grund und Goos [HGG05] zeigen, haben Interferenzgraphen von Programmen in SSA-Form spezielle Eigenschaften, die Registerzuteilung in drei Schritten erlauben: Auslagern, Färbung des Graphen und Kopienminimierung. Auslagern unabhängig von der Färbung des Graphen ermöglicht effizientere Auslagerungsalgorithmen und bessere Ergebnisse durch eine genauere Analyse des Programmkontexts.

[1] Dies gilt für Zugriffe auf den Pufferspeicher der ersten Ebene. Zugriffe auf Bereiche die nicht in Pufferspeichern liegen benötigen bis zu 100 zusätzliche Taktzyklen.

1.1 Aufgabenstellung

Ziel dieser Arbeit ist es existierende Auslagerungsverfahren zu evaluieren und zu analysieren, so dass aus aus den gewonnenen Erkenntnissen heuristische Verfahren zum Auslagern auf SSA-Form entwickelt werden können. Dabei wird speziell auf die Probleme und Möglichkeiten eingegangen, die beim Auslagern durch die SSA-Form entstehen.

1.2 Aufbau der Arbeit

Im nächsten Kapitel werden Grundlagen aus dem Übersetzerbau und der Graphentheorie erklärt, die für diese Arbeit relevant sind. Ausserdem wird das Verfahren der Registerzuteilung für Programme in SSA-Form vorgestellt.

In Kapitel 3 wird ein Überblick über verwandte Arbeiten gegeben und das Auslagerungsverfahren eines Übersetzers betrachtet.

Kapitel 4 beschreibt zunächst wie die SSA-Form beim Auslagern von Programmen erhalten werden kann. Die darauf folgenden allgemeinen Beobachtungen dienen als Grundlage für die anschliessend vorgestellten Auslagerungsalgorithmen. Zuletzt wird die Entstehung von Speicherkopien erläutert und ein gieriger Algorithmus zur Minimierung dieser Kopien vorgestellt.

In Kapitel 5 wird von der praktischen Umsetzung dieser Ansätze berichtet. Die groben Strukturen der Auslagerungsphase werden erklärt und die Algorithmen im Detail beschrieben.

Kapitel 6 präsentiert die Ergebnisse von Vergleichsmessungen der Heuristiken untereinander und mit anderen Übersetzern.

Im letzten Kapitel werden die Ergebnisse der Arbeit kurz zusammengefasst, bewertet und ein Ausblick auf zukünftige Erweiterungen gegeben.

2 Grundlagen

2.1 Zwischensprachen

Übersetzer bestehen aus einem Frontend und einem Backend. Das Frontend führt die syntaktische und semantische Analyse des Quellprogramms durch und transformiert es in eine Zwischensprache. Das Backend, dass die Befehle der Zwischensprache in Befehle der Zielarchitektur übersetzt und architekturspezifische Optimierungen durchführt.

In dieser Arbeit betrachten wir Zwischensprachen, in denen einzelne Funktionen durch einen Steuerfluss- und einen Datenabhängigkeitsgraph gegeben sind.

2.2 Steuerflussgraph

Definition 1 *Ein* Grundblock *ist eine Sequenz von Befehlen mit der Eigenschaft, dass stets alle Befehle des Grundblocks ausgeführt werden, wenn der Erste ausgeführt wird.*

Das bedeutet, dass es nicht möglich ist, mitten in einen Grundblock zu springen. Nur der letzte Befehl eines Grundblocks kann ein Sprung sein. Die Ziele von Sprungbefehlen sind weitere Grundblöcke.

Wir partitionieren Zwischensprachprogramme in eine Menge von Grundblöcken, die zu einem Steuerflussgraph verbunden sind:

Definition 2 *Der* Steuerflussgraph *ist aus den Grundblöcke eines Programms gebildet. Sie werden als Knoten im Graph dargestellt. Zwei Knoten a und b sind mit einer gerichteten Kante verbunden, falls a einen Sprung zu b enthält. Der Steuerflussgraph besitzt einen ausgewiesenen Startblock an dem die Programmausführung beginnt.*

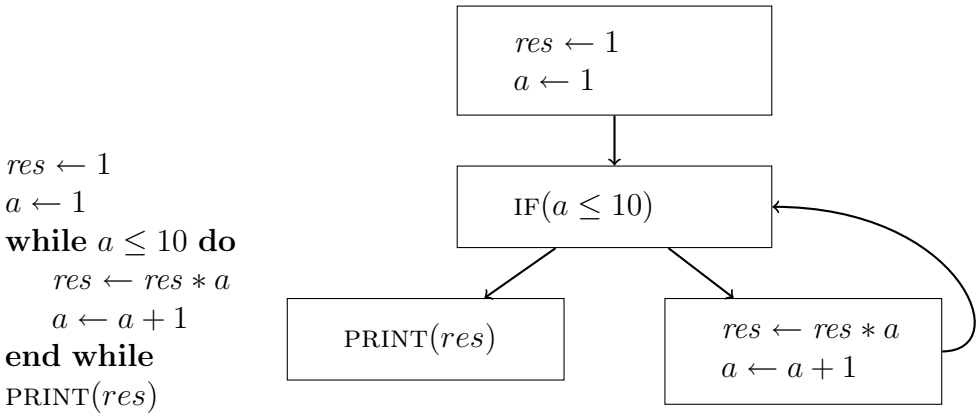

Abbildung 2.1: Beispiel für einen Steuerflussgraph

2.3 Dominanzrelation

Definition 3 *Wenn alle Pfade im Steuerflussgraph eines Programms, die vom Startblock zu Grundblock b führen, den Grundblock a enthalten, dann sagt ma a* dominiert *b. Formal: a* dom *b*

Einen Algorithmus zur Berechnung der Dominanzrelation ist in [LT79] angegeben. Dort wird auch bewiesen, dass die Dominanzrelation transitiv, reflexiv und antisymmetrisch ist. Sie induziert damit eine Halbordnung auf den Grundblöcken.

Definition 4 *Die Dominanzgrenze $DF(X)$ von einem Knoten X im Steuerflussgraph ist die Menge aller Knoten Y die nicht von X dominiert werden mit mindestens einem Vorgänger der von X dominiert wird.*

Cytron et al. [CFR+91] präsentieren einen effizienten Algorithmus zur Berechnung der Dominanzgrenzen.

2.4 Lebendigkeit

Definition 5 *Eine Variable oder ein Wert x ist an einer Stelle p im Programm* lebendig*, wenn es Pfade von p zu einer Verwendung von x gibt, die keine Definition von x enthalten.*

Definition 6 *Lebendige Werte müssen in einem Register gespeichert sein. Die Zahl der an einem Programmpunkt x gleichzeitig lebendigen Werte bezeichnet man deshalb als* Registerdruck*. Der Registerdruck eines Grundblocks ist das Maximum des Registerdrucks an allen Stellen innerhalb des Grundblocks.*

Die Lebendigkeiten am Anfang und Ende von Grundblöcken lässt sich mit einer rückwärtsgerichteten Datenflussanalyse bestimmen oder im Falle von SSA-Programmen mit Hilfe der Dominanzrelation und der Definiert-Benutzt-Beziehung (siehe zum Beispiel Nielson und Nielson [NNH05]). Sind die lebendigen Werte am Ende eines Grundblocks bekannt, so lassen sich die lebendigen Werte an einem anderen Punkt im Grundblock durch ein Rückwärtslaufen in der Befehlsfolge herausfinden: Ein Wert, der von einem Befehl verwendet wird ist für alle folgenden Befehl lebendig, wird er definiert, so ist er für alle folgenden Befehle nicht lebendig.

Definition 7 *Werte oder Variablen, die zu Beginn eines Grundblocks lebendig sind, bezeichnet man als* hineinlebend. *Werte oder Variablen die am Ende des Grundblocks lebendig sind, also noch in nachfolgenden Grundblöcken verwendet werden, als* herauslebend. *Argumente von Φ-Funktionen stellen Verwendungen am Ende der jeweiligen Vorgängerblöcken dar. Sie sind nicht notwendigerweise im Grundblock der Φ-Funktion lebendig. Man bezeichnet diese Werte als* am Ende lebend.

2.5 Schleifenbaum

Definition 8 *Eine* starke Zusammenhangskomponente *in einem Graph G, kurz* SZK *ist ein induzierter Teilgraph $G'(V', E')$ von G in dem es von jedem Knoten $x \in V'$ zu jedem Knoten $y \in V'$ einen Pfad gibt und falls es in G keine SZK gibt die G' enthält.*

SZKs können weitere SZKs enthalten. Dies führt zur Definition des Schleifenbaums:

Definition 9 *Der* Schleifenbaum *besteht aus Knoten, die jeweils ein Menge von Knoten des Steuerflussgraph G enthalten. Damit induziert jeder Knoten einen Teilgraph G' von G. Die Wurzel des Baums enthält alle Knoten des Steuerflussgraph. Jeder Knoten besitzt genau ein Kind für jede starke Zusammenhangskomponente in dem von ihm induzierten Teilgraph G'.*

Um den Speicherverbrauch zu reduzieren werden die in den Kindern enthaltenen Grundblöcke häufig nicht in den Elternknoten aufgeführt. Ein Beispiel für einen Steuerflussgraph mit zugehörigem Schleifenbaum ist in Abbildung 2.2 gezeigt.

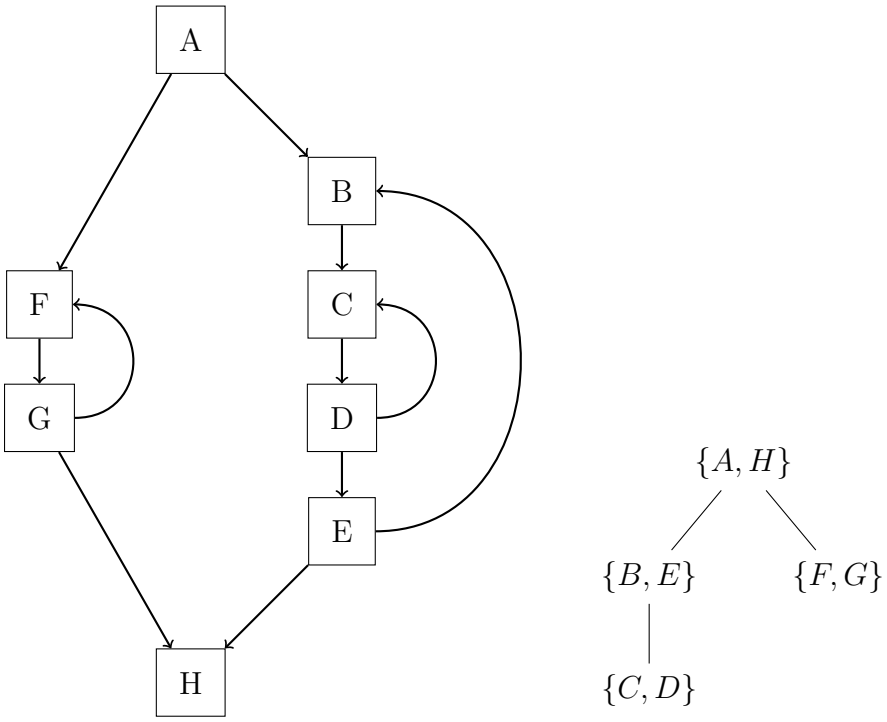

Abbildung 2.2: Steuerflussgraph mit Schleifenbaum

2.6 Ausführungshäufigkeiten

Nicht alle Grundblöcke eines Programms werden gleich oft ausgeführt. Code in Schleifen wird oft signifikant häufiger ausgeführt, als anderer Code. Fügt man zusätzliche Befehle in einen Grundblock ein, so hat dies um so größere Auswirkungen auf die Laufzeit, je häufiger der Grundblock ausgeführt wird.

Definition 10 *Die* Ausführungshäufigkeit $execfreq_b$ *eines Grundblocks b gibt an, wie oft ein Grundblock im Durchschnitt über alle möglichen Programmläufe ausgeführt wird.*

Auslagerungscode ist am effizientesten, wenn er in Grundblöcken mit niedriger Ausführungshäufigkeit platziert ist, daher sollten die Ausführungshäufigkeiten bereits zur Übersetzungszeit bekannt sein. Die Abfolge und damit auch die Ausführungshäufigkeiten von Grundblöcken hängen jedoch von den Eingabedaten des Programms ab und müssen geschätzt werden. Wu und Larus [WL94] beschreiben solche Verfahren. Eine weitere Möglichkeit ist, das

Programm in einer speziellen Form zu übersetzen, in der alle Grundblöcke Instrumentierungscode enthalten. Dieser Code zählt wie oft ein Grundblock ausgeführt wurde. Damit lassen sich für die verwendeten Eingabedaten die exakten Ausführungshäufigkeiten ermitteln.

2.7 Registerzuteilung

In Zwischensprachen werden beliebig viele gleichartige Speicherplätze für Berechnungsergebnisse verwendet. Die meisten Architekturen besitzen jedoch eine Speicherhierarchie: Es existiert eine begrenzte Zahl von Registern, auf die Instruktionen direkt und ohne Verzögerung zugreifen können. Danach kommt der Hauptspeicher, der durch eine Hierarchie von Pufferspeichern beschleunigt wird. Am Ende stehen langsame aber große Massenspeicher. Die Anzahl der Hardware-Register genügt meistens nicht, um alle Werte eines Programms zu speichern. Die Aufgabe der Registerzuteilung ist es, Werte möglichst sinnvoll auf Register und Hauptspeicher zu verteilen, so dass die Anzahl der Zugriffe auf den Hauptspeicher minimiert wird. Außerdem müssen Beschränkungen der Architektur beachtet werden. So sind Register typischerweise in Klassen unterteilt. Instruktionen können auf bestimmte Klassen oder Kombinationen von Registern beschränkt sein.

2.8 SSA-Form

Datenflussanalyse beschäftigt sich mit der Frage, woher die Argumente einer Berechnung kommen, und an welchen Stellen Ergebnisse verwendet werden. Eine kompakte Art Datenflussabhängigkeiten darzustellen ist die SSA-Form.

Statische Einmalzuweisung [RWZ88] (engl. static single Assignment Form - *SSA*) ist die Darstellung von Programmen in einer Form, in der es für Variable statisch genau eine Zuweisung gibt.[1] Auf diese Weise ist bei einer Verwendung einer Variablen direkt ersichtlich, wo deren Wert definiert worden ist.

Zur Transformation in SSA-Form wird bei aufeinanderfolgende Zuweisungen für jede Zuweisung eine neue Variable erzeugt und statt der ursprünglichen verwendet. Gibt es auf verschiedenen Pfaden im Programm unterschiedliche Zuweisungen für eine Variable, so wird an dem Punkt im Programm, an dem diese Pfade zusammenfließen eine weitere Variable erzeugt, die ihren Wert von einer Φ-Funktion erhält: Der Wert einer Φ-Funktion hängt davon ab, von welchem Steuerflussvorgänger die Funktion erreicht wurde.

[1] Eine Zuweisung kann zur Laufzeit des Programms natürlich mehrfach ausgeführt werden

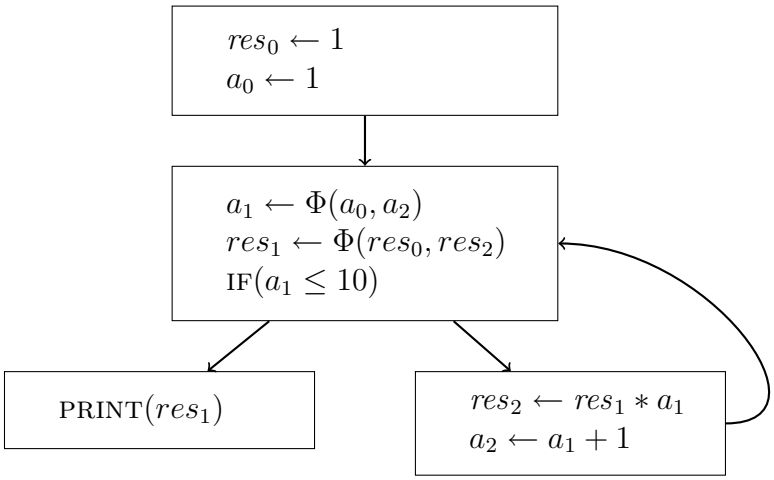

Abbildung 2.3: Programm aus Abbildung 2.1 in SSA-Form

Cytron et al. [CFR+91] zeigen, dass Φ-Funktionen genau in den Grundblöcken der Dominanzgrenze eines Wertes benötigt werden. In der Arbeit wird ausserdem ein effizienter Algorithmus zur Berechnung der Dominanzgrenzen und der SSA-Form präsentiert.

Ein wichtige Eigenschaft von Programmen in SSA-Form ist, dass die Definition eines Wertes stets ihre Verwendungen dominiert. Hierbei ist zu beachten dass Φ-Funktionen keine echten Verwendungen eines Wertes darstellen wie im folgenden Abschnitt erläutert.

2.8.1 Semantik der Φ-Funktion

Φ-Funktionen sind keine echten Instruktionen sondern ein notationeller Trick, um Werte abhängig vom Steuerfluss auszuwählen. Φ-Funktionen werden vor der Codeausgabe durch Befehle in den Steuerflussvorgängern ersetzt, die die entsprechenden Werte auf den Speicherplatz der von der Funktion definierten Variable kopieren. Das bedeutet, dass die Argumente der Φ-Funktionen keine echten Verwendungen im selben Grundblock sind, sondern diese Werte am Ende der jeweiligen Steuerflussvorgänger verwendet werden. Die Definitionen dieser Werte dominieren nicht notwendigerweise die Φ-Funktion, sondern nur die entsprechenden Steuerflussvorgänger.

Semantisch gesehen werden alle Φ-Funktionen gleichzeitig ausgewertet, wenn ihr Grundblock erreicht wird. Daraus folgt, dass Φ-Funktionen immer die ersten Befehle in einem Grundblock sein müssen. Wenn sich Φ-Funktionen ge-

genseitig referenzieren, muss darauf geachtet werden, dass die Werte vor der Auswertung als Argumente verwendet werden.

2.9 Registerzuteilung auf SSA-Form

Wie Hack, Grund und Goos [HGG05] zeigen, ist Registerzuteilung auch für Programmen in SSA-Form möglich und sinnvoll. Φ-Funktionen lassen sich nach der Registerzuteilung abbauen, ohne dass die Zuteilung ungültig wird. Registerzuteilung auf SSA-Form hat entscheidende Vorteile gegenüber anderen Ansätzen:

- Der Interferenzgraph von Programmen in SSA-Form ist chordal. Ein chordaler Graph lässt sich in polynomieller Zeit färben. Bei gegebener Dominanzrelation und Lebendigkeiten liegt der Aufwand in $O(r \cdot n)$, wobei r die Anzahl der Register und n die Anzahl der Instruktionen ist.

- Zum Färben des Interferenzgraph sind so viele Farben nötig wie die Anzahl der Knoten in der größten Clique im Graph. Da der Graph perfekt ist gilt dies auch für alle induzierten Teilgraphen. Damit genügt es den Registerdruck an jeder Stelle des Programms auf $\leq k$ zu senken um sicherzustellen, dass der Interferenzgraph k-färbbar ist. Es kann also ohne expliziten Interferenzgraph ausgelagert werden.

- Kopien treten nur durch Abbau von Φ-Funktionen und bei der Behandlung von Registerbeschränkungen auf. Sie können also in einem eigenen Schritt nach der Färbung des Graphen durch Umfärben der Knoten minimiert werden.

3 Verwandte Arbeiten

3.1 Registerzuteilung durch Graphfärben

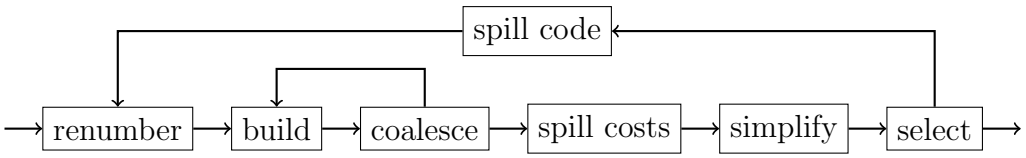

Abbildung 3.1: Schema des Chaitin/Briggs Registerzuteilers

Chaitin [Cha82] formuliert das Problem der Registerzuteilung und des Auslagerns als Graphfärbungsproblem. Dazu wird ein Interferenzgraph aufgebaut: Jede Variable des Programms wird durch einen Knoten repräsentiert, zwei Knoten sind verbunden falls es einen Punkt im Programm gibt an denen beide Variablen lebendig sind. Eine k-Färbung des Graphen stellt eine gültige Registerzuteilung mit k Registern dar. Das Färbungsproblem ist NP-Vollständig, weshalb eine Heuristik zum Einsatz kommt: Sukzessive werden alle Knoten mit Grad $< k$ aus dem Graphen entfernt und auf einen Keller gelegt. Falls der Graph nur noch Knoten vom Grad $\geq k$ enthält, so wählt der Algorithmus einen der verbleibenden Knoten aus, fügt Auslagerungscode an alle Definitionen und Einlagerungscode an alle Verwendungen der Variable ein und entfernt den Knoten aus dem Graphen. Sind keine Knoten mehr übrig, so wird falls Variablen ausgelagert wurden der Interferenzgraph neu aufgebaut und das Verfahren erneut durchgeführt. Ansonsten werden alle Knoten nacheinander vom Keller genommen und wieder in den Graphen eingefügt, dabei wird der Knoten jeweils mit einer zu allen seinen Nachbarn unterschiedlichen Farbe gefärbt.

Briggs, Cooper und Torczon [BCT94] verbessern die Heuristik indem auch Knoten mit Grad $\geq k$ auf den Keller gelegt werden. Auslagerungscode wird erst erzeugt wenn es später nicht möglich ist den Knoten zu färben. In manchen Fällen wie dem Diamant-Graphen (Abbildung 3.2 ist ein Beispiel für eine hypothetische Architektur mit zwei Registern), wird damit eine Färbung möglich, bei der der Algorithmus von Chaitin bereits ausgelagert hätte. Briggs diskutiert ausserdem eine feinere Einteilung der Variablen in Lebensbereiche

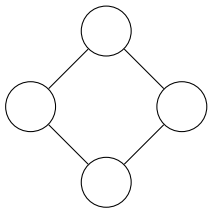

Abbildung 3.2: Diamant Interferenz Graph

und Kriterien zur Auswahl eines Auslagerungskandidaten. Er stellt ausserdem ein Modell für die Neuberechnung von Werten auf.

Die Briggs/Chaitin Heuristik erlaubt es auf elegante Art und Weise Registerbeschränkungen im Graph auszudrücken und zu behandeln. Sie erzielt in der Praxis gute Ergebnisse. Problematisch bei Chaitin's Methode ist, dass ganze Variablen global ausgelagert werden. Wird eine Variable ausgelagert, die in Programmteilen mit hohem und mit niedrigem Registerdruck verwendet wird, so werden unnötige Ein- und Auslagerungsbefehle in die Teile mit niedrigem Registerdruck eingefügt. Um dieses Problem zu entschärfen wird versucht die Lebensbereiche der Variablen vor der Färbung des Graphen geschickt zu splitten (z.B. [BDEO97]). Der iterative Ansatz des Algorithmus kann ausserdem in manchen Fällen einen hohen Aufwand verursachen.

3.2 Registerzuteilung durch priorisiertes Färben

Ein weiterer Ansatz zur globalen Registerzuteilung stammt von Chow und Hennessy [CH84]. In einer ersten Phase werden lokal in Grundblöcken Register auf Lebensbereiche von Variablen zugeteilt, falls die Zuteilung der Register keinen negativen Effekt auf die globale Zuteilung hat. Dies ist dann der Fall, wenn die durch die Zuteilung eingesparten Ein- und Auslagerungsbefehle mehr Kosten verursacht hätten als die eventuell für eine korrekt globale Zuteilung nötigen Ein- und Auslagerungsbefehle am Anfang und Ende des Grundblocks.

Für alle Variablen wird für die zweite Phase ein Interferenzgraph aufgebaut und die für ein Ein- und Auslagern nötigen Kosten berechnet. Die Variablen werden dann nach Kosten sortiert. Es wird nacheinander versucht den Variablen Register zuzuteilen. Ist eine Zuteilung nicht möglich, wird versucht den Lebensbereiche der Variable zu splitten und die entstandenen Lebensbereiche in die Liste einzusortieren. Ist ein Splitten nicht sinnvoll möglich wird für die Variable oder den Lebensbereich Auslagerungscode eingefügt.

Im Gegensatz zu Chaitin's Algorithmus ist priorisiertes Färben in linearer Zeit durchführbar. Chaitin's Färbeheuristik kann eventuell bessere Färbungen finden, da sie nicht auf eine feste Färbereihenfolge wie Chows Algorithmus angewiesen ist. Die Priorisierung der Lebensbereiche stellt allerdings sicher, dass die geschwindigkeits-kritischen Variablen des Programms bevorzugt Register zugeteilt bekommen. Die lokale Zuteilung in einer ersten Phase ermöglicht bessere Registerzuteilung in langen Grundblöcken, da Lebensbereiche erst während der Zuteilung gesplittet werden können, ist eine feinere Zuteilung möglich.

3.3 Lokale Registerzuteilung

Unabhängig vom Problem der globalen Registerzuteilung wurde auch das Teilproblem der Registerzuteilung in einzelnen Grundblöcken betrachtet: Bei einer gegebenen Sequenz von Instruktionen eine korrekte Registerbelegung zu finden, die die Anzahl der Ein- und Auslagerungen minimiert.

Lokale Registerzuteilung wurde von Chow und Hennessy [CH84] als Vorstufe zur globalen Registeruteilung diskutiert. Guo, Garzarán und Padua [GGP03] zeigen die erfolgreiche Anwendung von Belady's Algorithmus [Bel66] zur lokalen Registerzuteilung. Das Problem wird von Farach und Liberatore in [FL98] exakt definiert und ein Kostenmodell aufgestellt. Anschließend wird lokale Registerzuteilung als NP-Vollständig bewiesen. Ein Teile und Herrsche Algorithmus zur optimalen Lösung des Problems vorgeschlagen. Zur heuristischen Lösung wird ein zwei-approximierender Algorithmus angegeben, der die Zuteilung auf ein Flussproblem reduziert. Ausserdem wird eine Variation der Belady-Heuristik vorgestellt.

3.4 Auslagern auf SSA-Form

Bouchez, Darte, Guillon und Rastello [BDGR05] beschäftigen sich mit Auslagern für Programme in SSA-Form. Sie beweisen, dass auch dieses Problem NP-Vollständig ist. Die Vorteile der SSA-Form bei der Darstellung von Werten und Lebenszeiten werden erläutert. Es werden Messungen auf Programmen durchgeführt die einmal ohne und einmal mit SSA-Form entstanden sind. Programme in SSA-Form zeigen einen starken Rückgang bei den Auslagerungskosten bei minimal höheren Kosten durch zusätzliche Kopien.

3.5 GCC

Die Gnu Compiler Collection, kurz GCC ist eine Sammlung von Übersetzern für C, C++, Java, Objective-C, Fortran und Ada. Alle Übersetzer nutzen ein gemeinsames Backend, das Code für eine Vielzahl an Architekturen erzeugen kann. Einen guten Überblick über Registerzuteilung in GCC gibt Makarov [Mak04].

Register werden im GCC Übersetzer in zwei Phasen zugeteilt: Zuerst lokal pro Grundblock und danach in einer weiteren Phase global. Die GCC-Zwischensprache RTL arbeitet mit Pseudoregistern die dann während des Zuteilens auf echte Register ("hard Registers") abgebildet werden.

3.5.1 Lokale Zuteilung

In einer ersten Phase (local.c) wird eine lokale Registerzuteilung durchgeführt. Dabei werden nur Lebensbereiche von Variablen betrachtet, die komplett innerhalb eines Grundblocks liegen, wenn sich also die Definition einer Variable und alle potentiellen Verwendungen dieser Definition im selben Grundblock befinden. Diese Bereiche werden als *Quantity* bezeichnet.

Falls zwei Lebensbereiche nicht interferieren und die Definition des zweiten Lebensbereichs aus der letzten Verwendung des ersten Lebensbereichs entsteht, so werden diese Lebensbereiche verschmolzen. Dies eliminiert Kopierbefehle und bei Architekturen mit Zwei-Adresscode Befehle eventuell nötige zusätzliche Kopierinstruktionen.

Die Lebensbereiche werden dann nach folgender Formel priorisiert:

$$priority = \frac{log_2(references) * frequency * size}{lifetime}$$

references ist die Anzahl der Definitionen und Verwendungen des Lebensbereichs, *frequency* die Anzahl der Definitionen und Verwendungen der Beteiligten Pseudoregister multipliziert mit der Ausführungshäufigkeit der Grundblöcke in denen diese Definitionen und Verwendungen stehen und *size* ist die Grösse der benötigten Register in Bytes[1]. *lifetime* gibt die Zahl der Instruktionen von der Definition des Lebensbereichs bis zur letzten Verwendung an.

Nach obiger Priorisierung wird dann versucht den Lebensbereichen echte Register zuzuweisen.

[1] Registerklassen sind in gcc hierarchisch angeordnet, so dass nicht alle Register einer Klasse zwangsläufig die gleiche Grösse haben.

3.5.2 Globale Zuteilung

Ein ähnliches Verfahren wird in einem weiteren Schritt auf alle noch nicht zugeteilten Werte angewendet. Diesmal werden Pseudoregister über Grundblockgrenzen hinweg betrachtet. Dabei werden die noch nicht zugeteilten Pseudoregistern durchnumeriert und Zuordnungstabellen auf echte Register erstellt. Eine Konfliktmatrix der numerierten Pseudoregister untereinander wird erstellt sowie eine Konfliktmatrix der Pseudoregister mit Hardwareregistern. Falls Pseudoregister in Maschinenbefehlen benutzt werden, die auf ein bestimmtes Register beschränkt sind, so wird dieses Register als bevorzugtes Register für ein Pseudoregister festgelegt.

Die Pseudoregister werden dann nach folgender Gewichtung priorisiert:

$$priority = \frac{log_2(references) * frequency}{livelength}$$

Nacheinander wird dann versucht den Pseudoregistern eines Hardwareregistern zuzuteilen, so dass zwei interferierenden Lebensbereichen nicht das selbe Register zugeteilt wird.

3.5.3 Auslagern und Registerbeschränkungen

Für alle Pseudoregister, die noch kein Register zugeteilt bekommen haben wird in einem letzten Schritt (reload1.c, reload.c) Auslagerungscode erzeugt. Dabei wird versucht Ein- und Auslagerungsbefehle wenn möglich in Grundblöcke mit niedrigerer Ausführungshäufigkeit zu verschieben.

Ausserdem werden in dieser Phase die Registerbeschränkungen betrachtet und eventuell Kopierbefehle eingefügt, damit diese Eingehalten werden. Ausserdem werden architekturspezifische Optimierungen durchgeführt.

4 Lösungsansatz

Im Folgenden werden die Bedingungen zum Auslagern auf Programmen in SSA-Form näher erläutert. Ein Kostenmodell zur Bewertung von Auslagerungen wird aufgestellt und allgemeine Anhaltspunkte für gute Auslagerungen aufgestellt.

Es wird erläutert, wie auf Programmen in SSA-Form ausgelagert werden kann ohne die SSA-Bedingung zu verletzen. Die Entstehung von Speicher-Φ-Funktionen wird erklärt und eine Heuristik zur Vermeidung von Speicherkopien angegeben. Schliesslich werden zwei Heuristiken zum Auslagern vorgestellt.

4.1 Kostenmodell

Auslagern erfolgt durch Einfügen von Auslagerungsbefehlen oder Neuberechnen von Werten. Jeder zusätzlich ausgeführte Befehl verlängert die Laufzeit eines Programms und verursacht deshalb Kosten. Die Kosten pro Befehl sind architekturspezifisch. Manche Grundblöcke eines Programms werden häufiger ausgeführt als andere, deshalb werden die Kosten der Befehle jeweils mit den Ausführungshäufigkeiten (siehe Abschnitt 2.6) der Grundblöcke gewichtet, in denen sie stehen.

$$costs = \sum_{b \in blocks} (execfreq_b \cdot \sum_{i \in b} costs_i)$$

Ziel der entwickelten Heuristiken ist es, bei möglichst geringen Kosten den Registerdruck auf das nötige Maß zu senken.

4.2 Auslagern von Φ-Funktionen

Φ-Funktionen sind keine echten Operationen sondern werden später durch Kopierbefehle in den entsprechenden Steuerflussvorgängern ersetzt (siehe Abschnitt 2.8.1). Dies bedeutet, dass Φ-Funktionen nicht an den Stellen im Programm ausgewertet werden an denen sie stehen. Im Gegensatz zu normalen

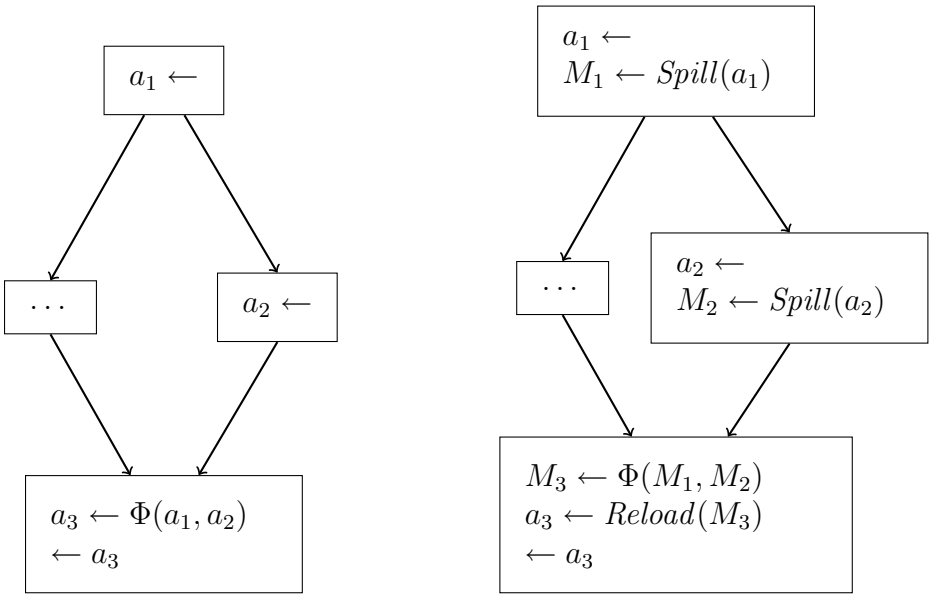

Abbildung 4.1: Auslagern einer Φ-Funktion

Befehlen ist es daher nicht möglich in eine Sequenz von Φ-Funktionen Auslagerungscode zu platzieren. Da man die Werte aber nicht auslagern kann, können maximal soviele Φ-Funktionen in einem Grundblock vorhanden sein, wie Register zur Verfügung stehen.

Sind mehr Φ-Funktionen vorhanden, so müssen dies ganzen Funktionen ausgelagert werden: Dazu werden alle Argumente ausgelagert und die Funktion durch eine spezielles Speicher-Φ-Funktion ersetzt. Die Semantik einer Speicher-Φ-Funktion ist analog zu der einer normalen Φ-Funktion, anstatt Werten in Register werden aber Werte an Speicherstellen ausgewählt. Ein Beispiel für das Auslagern einer Φ-Funktion zeig Abbildung 4.1.

Speicher-Φ-Funktionen können beim Abbau der SSA-Form durch Kopien der Speicherstellen in den Vorgängerblöcken ersetzt werden. Analog zu normalen Φ-Funktionen lassen sich auch hier die Kopien durch geschicktes Zusammenfassen von Speicherstellen vermeiden. Abschnitt 4.8 geht genauer auf diese Möglichkeit ein.

4.3 SSA Wiederaufbau

Wird ein Wert aus- und wieder eingelagert, so hat er mit dem Einlagerungsbefehl eine zweite Definition. Die SSA-Bedingung, dass jeder Wert nur eine Definition hat ist damit nicht mehr erfüllt. Der SSA-Aufbau muss für diesen Wert also erneut durchgeführt werden. Dies geschieht nach dem bekannt dem Algorithmus von Cytron et al. [CFR+91].

Alle Grundblöcke, die Definitionen des Wertes enthalten, sowie alle Grundblöcke, die in den iterierten Dominanzgrenzen enthalten sind werden markiert.

Danach werden alle Verwendungen des Wertes betrachtet und jeweils die nächsten Definition gesucht. Es genügt in diesem Fall den Dominanzbaum aufwärts nach markierten Grundblöcken zu suchen, da alle Werte in SSA-Form Programmen von ihren Definitionen dominiert werden. Enthält der gefundene Block keine Definition des Wertes, muss er Teil der Dominanzgrenze sein. In diesem Fall wird in diesem Block eine Φ-Funktion erzeugt für deren Argumente auf analoge Weise eine Definition gesucht wird.

4.4 Allgemeine Überlegungen

Die folgenden Überlegungen sind der Ausgangspunkt für die Entwicklung und Beurteilung der vorgestellten Auslagerungsalgorithmen.

- Ist an einer Stelle t der Registerdruck um n größer als die Anzahl der verfügbaren Register, so müssen mindestens n der an t lebendigen Werte ausgelagert werden.

- Muss ein Wert v an einer Stelle t ausgelagert werden, so muss auf allen Pfaden zwischen der Definition von v und t ein Auslagerungsbefehl liegen. Auf allen Pfaden zwischen t und einer Verwendung von v muss ein Einlagerungsbefehl liegen.

- Um beim Auslagern den Registerdruck in einem möglichst großen Bereich des Programms zu senken, sollten Auslagerungsbefehle an einem möglichst frühen und Einlagerungsbefehle an einem möglichst späten Punkt im Programm platziert werden.

- Für jeden Wert ist nur ein Auslagerungsbefehl nötig. Daher ist es oft sinnvoll einen Wert mehrfach statt mehrere Werte einfach aus- und einzulagern.

- Ein- und Auslagerungsbefehle sollten in Grundblöcken mit möglichst niedriger Ausführungshäufigkeit platziert werden.

Die ersten zwei Punkte sind Bedingungen für die Korrektheit der Auslagerung, die Übrigen sind Anhaltspunkte um effizienten Auslagerungscode zu erzeugen.

4.5 Der Algorithmus von Belady

Auslagern von Speicherseiten (*Paging*) ist dem Auslagern bei der Registerzuteilung sehr ähnlich: Betriebsysteme können Programmen einen virtuellen Adressraum zur Verfügung zu stellen, der grösser als der tatsächlich vorhandene Hauptspeicher ist. Um dies zu ermöglichen, werden nicht verwendete Speicherseiten auf die Festplatte ausgelagert und vor ihrer nächsten Verwendung wieder eingelagert. Normalerweise ist es nicht möglich, vorherzusehen welche Speicherseiten wann benötigt werden. Zur theoretischen Untersuchung von Pagig-Algorithmen wird deshalb häufig ein Protokoll von Speicherzugriffen eines Programms verwendet. Belady entwickelte in diesem Zusammenhang den MIN-Algorithmus [Bel66], der bei einem gegebenen Protokoll optimale Einlagerungen für Speicherseiten erzeugt, falls man die Kosten für das Auslagern ignoriert.

Der MIN-Algorithmus arbeitet nach folgendem Prinzip: Das Protokoll wird von Anfang bis Ende durchgegangen, jede Seite auf die Zugegriffen wird und noch nicht im Hauptspeicher liegt wird eingelagert. Ist für eine Einlagerung keine freie Speicherseite mehr vorhanden, so wird diejenige Seite ausgelagert, die nicht mehr verwendet wird, oder deren nächste Verwendung am weitesten in der Zukunft liegt. Wird eine Seite nicht mehr verwendet, so wird sie zuerst ausgelagert.

Wie Guo, Garzarán und Padua [GGP03] zeigen, ist MIN auch zum Auslagern von Registern in langen Grundblöcken gut geeignet. Farach und Liberatore [FL98] zeigen, dass Auslagern in lokalen Grundblöcken NP-vollständig ist, wenn man nicht nur die Zahl der Einlagerungen sondern auch die Zahl der Auslagerungen minimiert. In ihren Messungen sind die Ergebnisse des Belady-Algorithmus durchschnittlich jedoch nur 4% schlechter als die optimale Lösung.

Die Ergebnisse des MIN-Algorithmus lassen sich wie Hack, Grund und Goos [HGG05] zeigen, zu einer global gültigen Auslagerung erweitern. Dazu muss für jeden Grundblock eine Menge von Werten berechnet werden, die zu Beginn des Grundblocks lebendig sind. Bei Grundblöcken mit nur einem Steuerflussvorgänger wird die Menge der am Ende lebendigen Werte des Vorgängers

übernommen. Für Blöcke mit mehrere Steuerflussvorgängern werden alle Werte die in den Block hineinleben nach dem Zeitpunkt ihrer nächsten Verwendung sortiert und so viele Werte in die Menge der lebendigen Werte aufgenommen, wie Register vorhanden sind. Φ-Funktionen die nicht am Anfang der Funktion lebendig sind werden ausgelagert.

Bei Blöcken mit mehreren Vorgängern stimmt die Menge der am Anfang lebendigen Werte nicht unbedingt mit den lebendigen Werten am Ende der Vorgänger überein. In einem letzten Schritt wird deshalb für die die benötigten aber nicht lebendigen Werte Einlagerungscode in den Vorgängerblöcken eingefügt.

4.5.1 Nächste Verwendungen

Die Entscheidung welche Werte ausgelagert werden erfolgt im Belady-Algorithmus in Abhängigkeit von der nächsten Verwendung der Werte. Die *nächsten Verwendung* einer Variable v an einem Punkt p im Grundblock l ist die Anzahl der Befehle von p bis zur nächsten Verwendung von v. Befindet sich keine Verwendung in l, so schlagen Hack, Grund und Goos folgende Formel vor:

$$nextuse(l, v) = \begin{cases} \infty & \text{wenn } v \text{ in } l \text{ nicht lebendig ist} \\ 0 & \text{wenn } v \text{ in } l \text{ verwendet wird} \\ 1 + \min\limits_{l' \in succ_l} nextuse(l', v) & sonst \end{cases}$$

Es wird also jeweils der kürzeste Pfad zur nächsten Verwendung betrachtet oder ∞ zurückgegeben, falls der Wert nicht mehr verwendet wird.

4.5.2 Problemfälle

Der Algorithmus von Belady liefert sehr gute Ergebnisse für lokales Auslagern. Allerdings betrachtet der Algorithmus nur die unmittelbare Umgebung der Auslagerungspunkte. Für Werte deren nächste Verwendung nicht mehr innerhalb des Grundblocks liegt in dem ausgelagert wird, kann der Algorithmus schlechte Entscheidungen treffen. Insbesondere die Struktur des Steuerflussgraph wie Schleifen werden nicht beachtet.

Zwei typische Beispiele sind in Abbildung 4.2 zu sehen: In Beispiel (a) wird der Wert x_0 in Zeile 2 ausgelagert, eine richtige Entscheidung. Die Einlagerung erfolgt allerdings zum spätest möglichen Zeitpunkt, in diesem Fall innerhalb eines Blocks mit hoher Ausführungshäufigkeit innerhalb einer Schleife. Es wäre günstiger gewesen den Wert bereits vor der Schleife einzulagern.

```
1: x_0 ←
2: SPILL(M_0)
3: ... Hoher Registerdruck an diesem
   Punkt. x_0 wurde ausgelagert.
4: loop:
5: x_1 ←RELOAD(M_0)
6: ← x_1
7: if ... then
8:     GOTO(loop)
9: end if
```

(a) Einlagerung an schlechter Stelle

```
1: a ←
2: label:
3: b_0 ← ...
4: SPILL(M_0)
5: ... Hoher Registerdruck an diesem
   Punkt. b_0 wurde ausgelagert.
6: if ... then
7:     GOTO(label)
8: end if
9: ← a
10: b_1 ←RELOAD(M_0)
11: ← b_1
```

(b) Auslagerung an schlechter Stelle

Abbildung 4.2: Problemfälle für den Belady Algorithmus

In Beispiel (b) muss in Zeile 5 entweder Wert a oder der Wert b_0 ausgelagert werden. Der Algorithmus entscheidet sich für b_0, da a vor b verwendet wird. Allerdings muss b_0 in der Schleife ausgelagert werden, während a außerhalb der Schleife hätte ausgelagert werden können.

4.6 Der Algorithmus von Morgan

Die schlechten globalen Auslagerugsentscheidungen von Belady's Algorithmus lassen sich kompensieren, indem einige globale Auslagerungsentscheidungen vorweggenommen werden.

Die folgende Diskussion ist eine theoretische Fundierung des Auslagerungsverfahrens, das von Morgan in [Mor98] vorgeschlagen wird.

Es gibt häufig Werte, die lange Lebenszeiten haben, aber nur in wenigen Grundblöcken wirklich verwendet werden. Typische Beispiele hierfür sind Konstanten, oder Funktionsparameter, die am Anfang der Funktion zugewiesen werden aber erst in späteren Teilen genutzt werden.

Definition 11 *Ein Wert, der in einem Grundblock lebendig ist, aber weder verwendet noch definiert wird, bezeichnet man als* unbenutzt durchlebend. *Der Begriff lässt sich auf beliebige Teilgraphen des Steuerflussgraph erweitern: Ein Wert ist in einem Teilgraph des Steuerflussgraph* unbenutzt durchlebend, *wenn er in jedem Grundblock unbenutzt durchlebend ist.*

Ist ein induzierter Teilgraph gegeben, so lässt sich der Registerdruck in allen seinen Grundblöcken senken, indem unbenutzt durchlebende Werte auf allen Pfaden in den Teilgraph ausgelagert werden und auf allen Pfaden aus dem Teilgraph heraus wieder eingelagert werden. In den Grundblöcken innerhalb des Teilgraph entstehen keine Kosten, da ja keine Ein- und Auslagerungen wegen Definitionen oder Verwendungen auftreten können. Dies ist insbesondere dann günstig, wenn die Grundblöcke des Teilgraph viel höhere Ausführungshäufigkeiten als die Blöcke mit Kanten in und aus dem Teilgraph haben.

Beim Abschätzen der Ausführungshäufigkeiten wurde bereits die Faustregel verwendet, dass eine Schleife im Durchschnitt 10 mal ausgeführt wird. Nimmt man alle Grundblöcke innerhalb einer Schleife so haben diese also ungefähr 10 mal höhere Ausführungshäufigkeiten als die Grundblöcke, von denen aus die Schleife betreten wird.[1] Bei der Suche nach Teilgraphen, aus denen unbenutzt durchlebende Werte ausgelagert werden können, bieten sich also Schleifen an.

Der Algorithmus von Morgan macht sich dies zu nutze: Er führt eine Tiefensuche auf dem Schleifenbaum durch und bestimmt für jede Schleife den maximalen Registerdruck. Solange dieser höher als die tatsächliche Anzahl vorhandener Register ist, werden unbenutzt durchlebende Werte aus den Schleifen ausgelagert. Bei verschachtelten Schleifen werden Werte aus den äußeren Schleifen zuerst ausgelagert. Die Analyse der unbenutzt durchlebenden Werten sowie die Beschreibung einer Implementierung des Verfahrens finden sich in Kapitel 5.5.

4.7 Werte wiederberechnen

Es ist oft möglich Werte nicht in den Speicher auszulagern sondern erneut zu berechnen. Dies ist immer möglich für Werte, die nur von stets verfügbaren Ressourcen abhängen, oder von Werten die ebenfalls neu berechnet werden können. Stets verfügbare Ressourcen sind zum Beispiel Konstanten, der Schachtelzeiger oder Speicherbereiche die konstante Werte enthalten. In diese Klasse fallen typischerweise:

- Laden von Konstanten in Register
- Laden von Programmargumenten aus der Aufrufschachtel
- Laden von globalen Variablen aus dem Speicher

[1] Es wird von einem Steuerflussgraph ausgegangen, in dem alle kritischen Kanten durch einfügen zusätzlicher Grundblöcke entfernt sind. Es ist also insbesondere nicht möglich, dass eine Schleife direkt in eine nachfolgende Schleife springt.

In der Implementierung dieser Diplomarbeit wurde außerdem der Fall, dass ein bereits ausgelagerter Wert später erneut ausgelagert werden soll mit Hilfe von Wiederberechnungen modelliert: In diesem Fall wird dieser Wert nicht erneut ausgelagert, sondern der Wert wird "neu berechnet", indem der Einlagerungsbefehl an der entsprechenden Stelle des Programms dupliziert wird.

Werte, die neu berechnet werden können, werden in den Algorithmen bevorzugt ausgelagert, da hier kein Aufwand zum Aus- und nur geringer Aufwand zum Einlagern erforderlich ist.

Mit entsprechenden Analysen können auch Werte neu berechnet werden, die von anderen lebendigen Werten abhängen. Anstelle einer erneuten Berechnung können auch inverse Operationen genutzt werden um bereits zerstörte Werte wiederherzustellen, oder sofern der Registerdruck nicht über das architekturspezifische Limit steigt, kann die Lebenszeit von anderen Werten verlängert werden um Neuberechnungen zu ermöglichen. Diese erweiterten Analysen werden in den Heuristiken dieser Diplomarbeit nicht beachtet. Mögliche Verfahren werden in [Sza06] beschrieben.

4.8 Speicherkopienminimierung

Auslagerungsbefehle und Speicher-Φ-Funktionen benötigen Speicherplatz in der Aufrufschachtel in den die Werte beim Auslagern kopiert werden. Ein solcher Speicherplatz wird im folgenden *Auslagerungsplatz* genannt. Analog zur Lebendigkeit von Werten lässt sich auch die Lebendigkeit von Auslagerungsplätzen definieren:

Definition 12 *Der Auslagerungsplatz w ist an einer Stelle x im Programm* lebendig*, wenn es Pfade von x zu einem Einlagerungsbefehl der w verwendet gibt, auf denen kein Auslagerungsbefehl auf w liegt.*

Auslagerungsplätze, die an keinem Punkt im Programm gleichzeitig lebendig sind, können verschmolzen werden. Dadurch lässt sich Speicher sparen um die Leistung der Caches zu verbessern. Außerdem lassen sich Speicherkopien für Speicher-Φß vermeiden, wenn der Auslagerungsplatz der Φ-Funktion und des entsprechenden Arguments der selbe ist. Ein Beispiel hierfür ist in Abbildung 4.3 zu sehen.

Gesucht ist eine Abbildung $\& : (S \cup P) \rightarrow N$ von der Menge der Auslagerungsbefehle S und Speicher-Φ-Funktionen P auf eine Menge von Auslagerungsplätzen N. Diese soll Speicherkopien vorrangig an Stellen mit hoher Ausführungshäufigkeit vermeiden und die Zahl der Auslagerungsplätze gering halten.

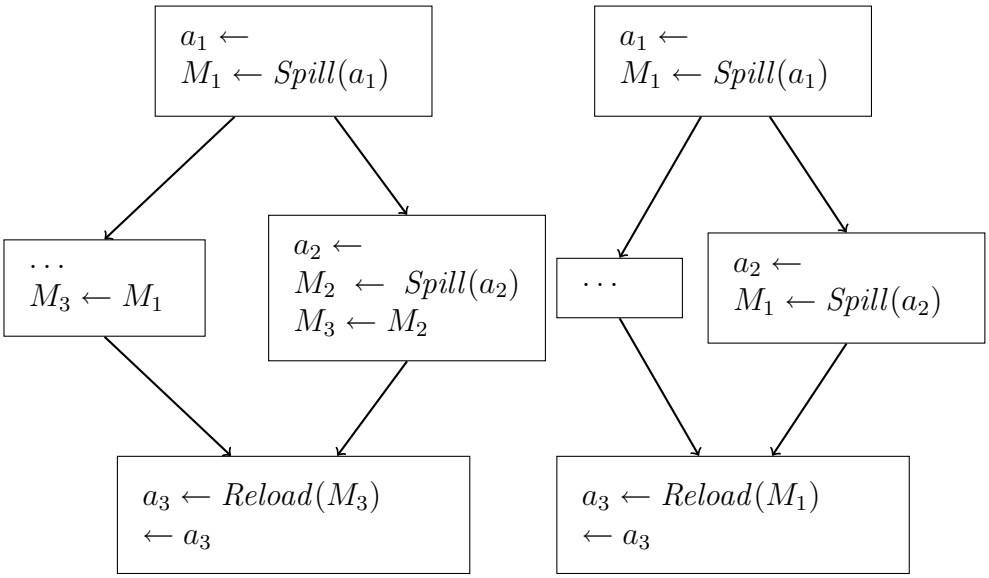

Abbildung 4.3: Verschmelzen von Auslagerungsplätzen vermeidet Speicherkopien

Auslagerungsplätze Zuweisen ist ein ähnlich gelagertes Problem wie Kopienminimierung bei der Registerzuteilung auf SSA-Form: Anstelle von Registern werden hier Auslagerungsplätze zugeteilt, die zuzuteilenden Objekte können in beiden Fällen interferieren. Der einzige Unterschied besteht in der beliebig großen Menge von Auslagerungsplätzen. Das Problem der Kopienminimierung wird in [HGG05] als NP-Vollständig in der Anzahl der Φ-Funktionen bewiesen. Der Beweis bleibt auch unter der Annahme einer beliebig großen Zahl von Registern gültig. Auslagerungsplatzzuweisung ist also ebenfalls NP-Vollständig in der Anzahl der Speicher-Φs.

In dieser Arbeit wurde zur Lösung des Problems ein gieriger Algorithmus verwendet: Es wird zunächst eine Liste aller Auslagerungen und Speicher-Φs erstellt und jeweils ein Auslagerungsplatz zugeteilt. Eine Interferenzmatrix dieser Auslagerungsplätze wird erzeugt. Anschließend werden alle Speicherkopien nach der Ausführungshäufigkeit ihrer Grundblöcke sortiert. In dieser Reihenfolge wird jetzt versucht die Auslagerungsplätze der Φ-Funktionen mit dem des jeweiligen Arguments zu verschmelzen, das die Speicherkopie hervorruft. Zuletzt wird systematisch versucht die verbliebenen Auslagerungsplätze zu verschmelzen.

5 Implementierung

Aussagen über die Leistungsfähigkeit der Algorithmen lassen sich am besten durch Messungen an konkreten Programmen treffen. Deren Implementierung als Teil des Firm-Übersetzers wird im Folgenden erklärt.

5.1 Die Zwischensprache Firm

Grundlage der Implementierung ist ein Übersetzer, der auf der in Karlsruhe entwickelten Zwischensprache Firm basiert.

Firm [LBBG05] ist eine graphbasierte Zwischensprache, das bedeutet, dass Operationen als Knoten in einem Graph dargestellt sind. Von Variablen ist vollständig abstrahiert: Das Programm befindet sich in SSA-Form. Damit repräsentiert jeder Knoten den Wert, den sine Ausführung erzeugt. Knoten sind über Kanten mit den von ihnen verwendeten Werten verbunden. Der Graph ist in Grundblöcke unterteilt. Jeder Knoten ist genau einem Grundblock zugeordnet. Die Grundblöcke sind verbunden mit den Knoten der Sprungbefehle über die sie erreicht werden können. Eine feste Ausführungsreihenfolge ist nicht gegeben. Mögliche Reihenfolgen sind nur implizit durch die Datenabhängigkeiten der Knoten gegeben.

Zur Verdeutlichung ist das Programms aus Abbildung 5.1 in Abbildung 5.2 als Firm-Graph dargestellt. Die Firm-Bibliothek stellt ein Rahmenwerk zur Arbeit mit diesen Graphen und vielfältige Optimierungen zur Verfügung.

```
1: function FIB(v)
2:     if v ≤ 1 then
3:         return 1
4:     end if
5:     result ← FIB(v − 1) + FIB(v − 2)
6:     return result
7: end function
```

Abbildung 5.1: Quelltext des Graphen aus Abbildung 5.2

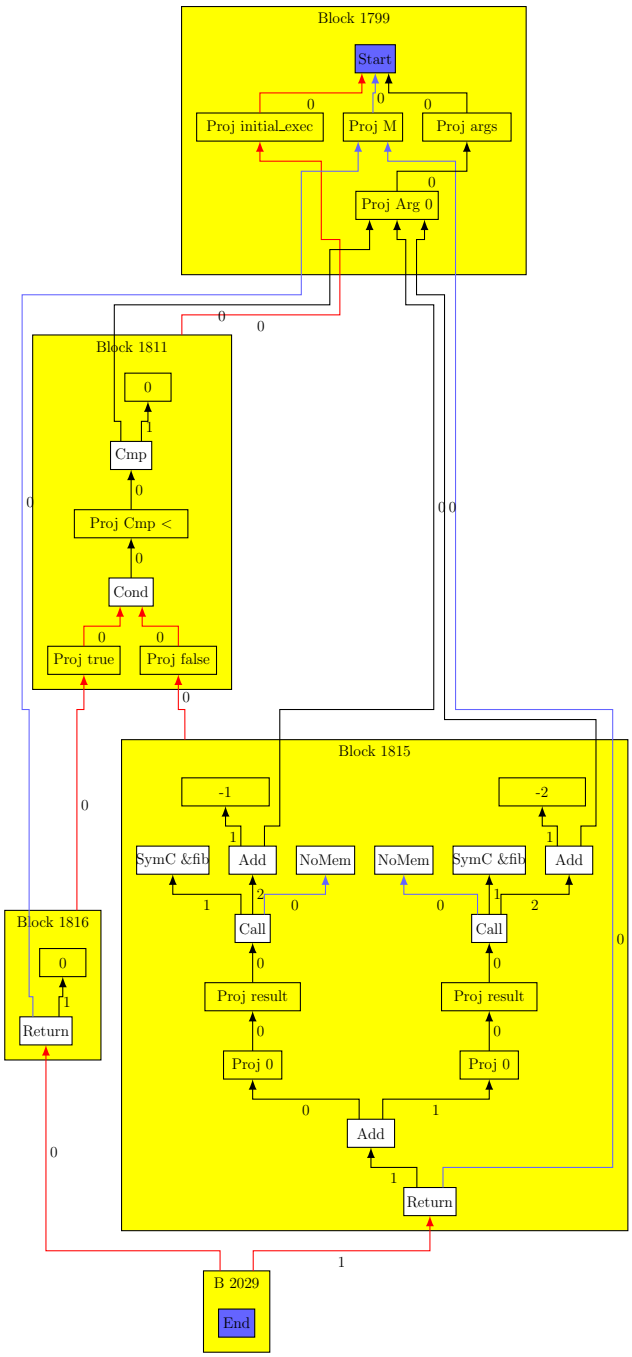

Abbildung 5.2: Firm Graph des Programms aus Abbildung 5.1

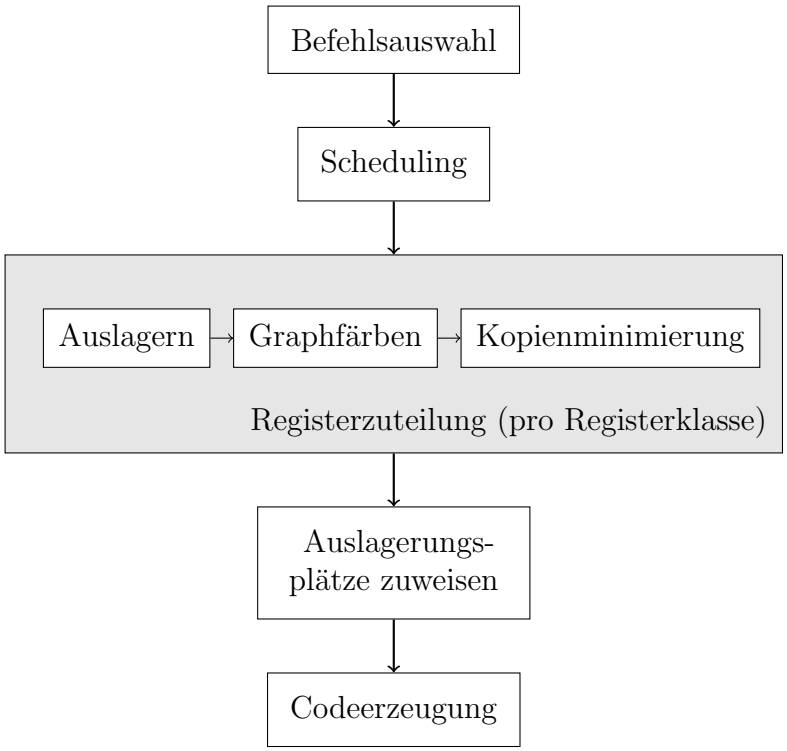

Abbildung 5.3: Phasen des Backend

5.2 Struktur des Backends

Der verwendete Übersetzer kombiniert die Zwischensprache Firm mit einem C-Frontend und einem Backend für die Intel x86 Architektur. Wichtig für die Implementierung der Auslagerungsverfahren ist die Arbeitsweise des Backends (Abbildung 5.3), die im Folgenden kurz beschrieben wird.

Um den Übersetzer konsistent zu halten und um von einigen Optimierungen in der Zwischensprache zu profitieren, arbeitet das Backend durchgängig mit Firm-Graphen zur Darstellung der Programme.

In der Befehlsauswahl Phase werden die generischen Knoten im Firm-Graph durch maschinenspezifische Knoten ersetzt.

Anschliessend wird mit eine Befehlsreihenfolge innerhalb der Grundblöcke festgelegt (Scheduling).

Jeder Maschinenknoten besitzt Attribute die die Anzahl und Art der benötigten Register angeben. Für alle Registerklassen wird jetzt nacheinander Auslagern, Färbung des Graphen und Kopienminimierung durchgeführt. Da-

nach wird den neu eingefügten Ein- und Auslagerungsbefehlen Auslagerungsplätze zugeteilt. Zuletzt wird die Reihenfolge der Grundblöcke festgelegt und der Code ausgegeben.

5.3 Auslagern und SSA-Wiederaufbau

Ausgelagerte Werte werden im Graphen dargestellt, indem Auslagerungsknoten (Spills) eingefügt werden. Diese sind mit dem auszulagernden Wert verbunden und erzeugen einen Speicherwert. Dieser Speicherwert repräsentiert einen Auslagerungsplatz. Die Attribute des Auslagerungsknoten geben an, welcher Auslagerungsplatz verwendet werden soll. Speicherwerte können auch als Argumente von Φ-Knoten verwendet werden, die dann ebenfalls einen Speicherwert produzieren. Werte werden mit Einlagerungsknoten (Reloads) wieder eingelagert. Sie sind mit einem Speicherwert verbunden und erzeugen eine Kopie des vorher ausgelagerten Wertes.

Um die eigentlichen Auslagerungsalgorithmen einfach zu halten, sind Hilfsfunktion zum Auslagern in einem gemeinsamen Modul untergebracht. Um einen wiederholten Wiederaufbau der SSA-Form zu vermeiden werden Aus- und Einlagerungen nicht direkt am Graph vorgenommen sondern gesammelt und später gemeinsam in den Graph eingefügt. Die Schnittstelle des Moduls erlaubt es Punkte im Programm anzugeben, an denen Einlagerungen geschehen sollen, sowie welche Φ-Funktionen ausgelagert werden sollen. Ausserdem lässt sich abfragen wie teuer eine Einlagerung an einem bestimmten Punkt ist. Am Ende der Auslagerungsphase wird die Liste der Einlagerungen abgearbeitet: Ein Wert wird neuberechnet, falls er nur von Konstanten oder anderen Werte, die neuberechnet werden können, abhängt und die Kosten für das Neuberechnen kleiner sind als für eine Einlagerung. Ist keine Neuberechnung möglich, wird ein Einlagerungsknoten eingefügt.

Für alle Werte mit Einlagerungsknoten wird ein Auslagerungsknoten direkt nach der Definition des auszulagernden Wertes eingefügt. Handelt es sich bei dem Wert um eine ausgelagerte Φ-Funktion, so werden stattdessen Auslagerungsknoten für deren Argumente eingefügt und bei weiteren ausgelagerten Φ-Funktionen analog verfahren.

Da die Einlagerungsknoten weitere Definitionen eines Wertes sind, ist die SSA-Bedingung nicht mehr erfüllt. Deshalb wird abschliessend die SSA-Form wieder aufgebaut und die Verwendungen der Werte angepasst, wie in Abschnitt 4.3 beschrieben.

5.3.1 Implementierung von Speicherkopien

In allen uns bekannten Architekturen sind Speicherkopien ohne ein freies Register nur weniger effizient oder überhaupt nicht umsetzbar. Allerdings steht erst nach der Auslagerungsphase fest, wo Speicherkopien benötigt werden. Zu diesem Zeitpunkt sind keine zusätzlichen Register mehr zuteilbar. Speicherkopien lassen sich deshalb nur mit einer der folgenden Methoden implementieren:

1. Während des Auslagerns werden für alle ausgelagerten Φ-Funktionen bereits Speicherkopien platziert. Dies stellt sicher, dass ein Register für die Kopien freigehalten wird. Andererseits ist aber auch ein Register verschwendet worden, falls die Kopie später eliminiert wird.

2. Treten Speicherkopien auf, so wird erneut ausgelagert. Da die zusätzlich ausgelagerten Werte eventuell zusätzliche Speicherkopien hervorrufen, muss der Prozess iterativ fortgesetzt werden, bis Register für alle Speicherkopien vorhanden sind.

3. Speicherkopien können innerhalb ihres Grundblocks an einen Ort verschoben werden an dem noch ein Register frei ist. Dabei ist zu beachten, dass dies zusätzliche Interferenzen der Auslagerungsplätze zur Folge haben kann und deren Zuteilung dadurch ungültig wird.

4. Bei der Codeerzeugung, wird vor jeder Speicherkopie ein Wert ausgelagert der nach der Speicherkopie wieder eingelagert wird. Diese ad-hoc Platzierung zusätzlicher Auslagerungen ist natürlich qualitativ nicht so gut wie die Ergebnisse der Heuristiken.

5. Manche Architekturen erlauben es Speicherkopien ohne ein freies Register zu implementieren. (siehe unten)

Wie die Messungen in den Tabellen 6.2 und **??** zeigen, treten Speicherkopien jedoch nur selten auf. Die Messungen zeigen, dass nur eine kleine Zahl von Φ-Funktionen wirklich Speicherkopien hervorrufen. Aus diesem Grund wurde für das zuvor beschriebene x86 Backend eine einfach zu implementierende Methode gewählt: Speicherkopien werden mit Hilfe von push- und pop-Befehlen mit Speicheroperanden ohne zusätzliche Register realisiert.

5.4 Der Algorithmus von Belady

Der Algorithmus von Belady operiert mit Mengen von lebendigen Werten. Diese sind mit Hilfe von Feldern implementiert, da eine Implementierung mit ver-

Algorithmus 1 Belady Auslagerungsalgorithmus

```
 1: procedure PlaceReload(block, value, BeforeInstruction)
 2:     if Spill(value) is undefined then
 3:         Place Spill behind definition of value
 4:     end if
 5:     Insert Reload behind BeforeInstruction
 6: end procedure
 7:
 8: procedure BeladyBlock(block)
 9:     if block already processed then
10:         return
11:     end if
12:     Mark block as processed
13:     calculate LiveIn
14:     for all instr ∈ block do
15:         if instr is a Φ-Instruction then
16:             CONTINUE
17:         end if
18:         for all value ∈ Uses(instr) \ workset do
19:             PlaceReload(block, instr)
20:         end for
21:         workset ← Uses(instr) ∩ Defs(instr)
22:         if |workset| > RegistersAvailable then
23:             Sort worklist by next-use distance
24:             Keep first RegistersAvailable entries in worklist
25:         end if
26:     end for
27: end procedure
```

zeigerten Datenstrukturen wie Bäumen oder verketteten Listen bei der kleinen Anzahl vorhandener Register die Laufzeit nicht verbessert.

In einer Hashtabelle werden für jeden Grundblock die Felder der Werte, die am Anfang und am Ende lebendig sein sollen gespeichert.

Die einzelnen Schritte des Algorithmus sind in Abschnitt 4.5 beschrieben. Pseudocode ist in Algorithmus 1 zu sehen.

5.4.1 Nächste Verwendungen

Um die nächste Verwendung eines Wertes v zu berechnen wird die Befehlsfolge des Grundblocks nach v durchsucht. Findet sich keine Verwendung, so wird die Suche in den Steuerflussnachfolgern fortgesetzt. Dabei werden bereits betrachtete Blöcke mit einem Besucht-Flag markiert um endlose Rekursion zu vermeiden. Die Zwischenergebnisse werden pro Grundblock in einer Hashtabelle gepuffert um weitere Suchen zu beschleunigen. Bei Steuerflussverzweigungen wird jeweils das Minimum der nächsten Verwendungen zurückgeliefert (siehe Abschnitt 4.5.1).

Hierbei ist zu beachten, dass Φ-Funktionen keine Verwendungen sind, die Einlagerungen hervorrufen. Deshalb sollte wenn die nächste Verwendung eine Φ-Funktion ist die Suche nach dem nächsten Verwender der Φ-Funktion fortgesetzt werden. Dies gilt analog für Auslagerungsbefehle die von vorherigen Auslagerungsalgorithmen stammen.

5.5 Der Algorithmus von Morgan

Der Algorithmus von Morgan arbeitet in zwei Phasen: Zuerst wird die Menge der unbenutzt durchlebenden Werte für Schleifen ermittelt, danach ausgelagert.

Die Menge der unbenutzt durchlebenden Werte einer Schleife ist die Schnittmenge aller Mengen unbenutzt durchlebenden Werte der Grundblöcke der Schleife. Um diese zu ermitteln wird eine Tiefensuche durch den Schleifenbaum durchgeführt. Beim Aufsteigen aus einem Ast wird dann jeweils die Schnittmenge aller unbenutzt durchlebenden Werte der inneren Schleifen und Grundblöcke gebildet. Das Ergebnis wird in einer Hashtabelle gepuffert. Alle Werte, die in einer äusseren Schleife unbenutzt durchleben werden aus den Mengen der unbenutzt durchlebenden Werte der inneren Schleifen entfernt. Pseudocode der Analyse ist in Algorithmus 2 angegeben.

In einer weiteren Analyse wird für jede Schleife die Menge der Steuerflusskanten ermittelt, mit denen die Schleife betreten oder verlassen werden kann.

Dabei ist darauf zu achten, dass eine Schleife eventuell auch mit einer Kante aus einer inneren Schleife verlassen werden kann.

Werte sollten bevorzugt aus äusseren Schleifen ausgelagert werden. Gleichzeitig sollten nicht mehr Werte ausgelagert werden als nötig. Im Folgenden werden Formeln für die Anzahl der unbenutzt durchlebenden Werte die aus jeder Schleife ausgelagert werden sollten aufgestellt.

Die Anzahl der zuteilbaren Register ist mit r bezeichnet, der maximale Registerdruck im Grundblock b mit $p(b)$. **root** ist die Wurzel des Schleifenbaumes. Für eine Schleife l gibt $\mathit{blocks}(l)$ die Menge der Grundblöcke der Schleife zurück, $\mathit{innerloops}(l)$ die Menge der inneren Schleifen. $\mathit{parent}(l)$ liefert die nächste äussere Schleife von l. Die Menge der unbenutzt durchlebenden Werte eines Blocks b wird mit $\mathit{ul}(b)$ bezeichnet.

Die Anzahl der unbenutzt durchlebender Werte in einer Schleife l, die für eine Auslagerung in Frage kommen wird mit α bezeichnet:

$$\begin{aligned} \alpha : \text{Schleife} &\rightarrow \mathbb{N}_0 \\ \textbf{root} &\mapsto 0 \\ l &\mapsto |\mathit{ul}(l)| + \alpha(\mathit{parent}(l)) \end{aligned}$$

Die Anzahl der benötigten Auslagerungen um den Registerdruck in der Schleife l auf die Zahl der vorhandenen Register zu senken sei β:

$$\begin{aligned} \beta : \text{Schleife} &\rightarrow \mathbb{N}_0 \\ l &\mapsto \max(\max_{b \in \mathit{blocks}(l)} p(b) - r, \max_{c \in \mathit{innerloops}(l)} \beta(c)) \end{aligned}$$

Damit lässt sich die Zahl der Werte, die aus der Schleife l ausgelagert werden sollten berechnen:

$$\begin{aligned} \theta : \text{Schleife} &\rightarrow \mathbb{N}_0 \\ l &\mapsto \min(\alpha(l), \beta(l)) \end{aligned}$$

θ lässt sich durch eine Tiefensuche im Schleifenbaum bestimmen. Dabei wird bei jedem Abstieg die Anzahl der unbenutzt durchlebenden Werte der äusseren Schleifen übergeben und beim Aufstieg die Zahl der in den inneren Schleifen benötigten Auslagerungen zurückgeliefert.

Die auszulagernden Werte werden dann nach Kosten sortiert[1] und die ersten $\theta(l)$ Werte aus der Schleifen l ausgelagert. Dazu wird in allen Blöcke, die eine

[1] Werte die wiederberechnet werden können sind billiger

direkte Kante in l hinein besitzen Auslagerungscode platziert. In allen Blöcke, die mit einer Kante aus l heraus verbunden sind wird Einlagerungscode platziert. Pseudocode ist in Algorithmus 3 zu sehen.

Algorithmus 2 Auffinden unbenutzt durchlebender Werte

```
 1: procedure ConstructLivethroughUnusedBlock(block)
 2:     LivethroughUnused(block) ← LiveIn(block) ∩ LiveOut(block)
 3:     for all instr ∈ block do
 4:         if instr is a Φ-Instruction then
 5:             CONTINUE
 6:         end if
 7:         LivethroughUnused(block) ← LivethroughUnused(block) \
    Uses(instr)
 8:     end for
 9: end procedure
10:
11: procedure ConstructLiveThroughUnusedLoop(loop)
12:     LivethroughUnused(loop) ← ⊤
13:     for all block ∈ loop do
14:         ConstructLivethroughUnusedBlock(block)
15:         LivethroughUnused(loop) ← LiveThroughUnused(loop) ∩
    LiveThroughUnused(block)
16:     end for
17:     for all childloop ∈ loop do
18:         ConstructLiveThoughUnusedLoop(childloop)
19:         LivethroughUnused(loop) ← LiveThroughUnused(loop) ∩
    LiveThroughUnused(childloop)
20:     end for
21: end procedure
```

5.6 Minimierung von Speicherkopien

Die Minimierung der Speicherkopien sucht den Graph zunächst nach Auslagerungsknoten und Speicher-Φs ab. Diese werden numeriert und ein Interferenzgraph der Speicherwerte aufgebaut. Der Interferenzgraph ist als eine Reihung von Bitvektoren im Speicher repräsentiert. Die Gruppen der verschmolzenen Speicherwerte werden in einer Union-Find-Datenstruktur gespeichert. Zunächst ist jeder Speicherwert eine einelementige Menge. Die Mengen wer-

Algorithmus 3 Algorithmus von Morgan

```
 1: function GetBlockSpillsNeeded(block)
 2:     living ← LiveEnd_block
 3:     maxpressure ← |living|
 4:     for all instruction ∈ ReverseSchedule(block) do
 5:         living ← living \ Defs(instruction)
 6:         living ← living ∪ Uses(instruction)
 7:         maxpressure ←max(maxpressure, |living|)
 8:     end for
 9:     return maxpressure − RegistersAvailable
10: end function
11:
12: procedure ReducePressureInLoop(loop, OuterSpillsPossible)
13:     SpillsNeeded ← 0
14:     SpillsPossible ← |LivethroughUnused_loop| + OuterSpillsPossible
15:     for all block ∈ loop do
16:         BlockSpillsNeeded ← GetBlockSpillsNeeded(block)
17:         SpillsNeeded ←max(SpillsNeeded, BlockSpillsNeeded)
18:         SpillsNeeded ←min(SpillsNeeded, SpillsPossible)
19:     end for
20:     for all childloop ∈ loop do
21:         LoopSpillsNeeded ← ReducePressureInLoop(childloop,
    SpillsPossible)
22:         SpillsNeeded ←max(SpillsNeeded, LoopSpillsNeeded)
23:     end for
24:     iter ←ConstructIterator(LiveThroughUnused_loop)
25:     while SpillsNeeded > 0 and iterisfinished do
26:         for all edge ∈ InEdges(loop) do
27:             Place spill instruction for iterator value on edge
28:         end for
29:         for all edge ∈ OutEdges(loop) do
30:             Place reload instruction for iterator value on edge
31:         end for
32:     end while
33:     return SpillsNeeded
34: end procedure
```

den beim Verschmelzen von Auslagerungsplätzen kombiniert. Der Interferenzgraph wird beim Verschmelzen jeweils für den Repräsentanten der Menge in der Union-Find-Struktur aktualisiert.

Um Speicherkopien zu minimieren werden alle Argumente von Speicher-Φ-Funktionen nach Kosten sortiert. Falls der Auslagerungsplatz der Φ-Funktion und des Arguments nicht verschmolzen werden können entsteht eine Speicherkopie. Die Kosten dieser Kopie müssen mit der Ausführungshäufigkeit des entsprechenden Steuerflussvorgängers des Φs gewichtet werden, in dem die Kopie entstehen würde.

Mit einem gierigen Verfahren wird dann versucht die Auslagerungsplätze der Argumente mit denen der Φ-Funktionen zu verschmelzen.

Zuletzt werden die übrigen Speicherwerte verschmolzen wo dies möglich ist um die Zahl der benötigten Auslagerungsplätze zu minimieren und damit Caches besser auszunutzen.

6 Messungen

Zur Evaluierung der Leistungsfähigkeit der Heuristiken wurden Messungen an verschiedenen Benchmarks durchgeführt. Ziel dabei ist es die Qualität der Auslagerungen im Allgemeinen und die Verbesserungen durch Anwendung von Morgan's Algorithmus zu spezifizieren. Abschliessend werden die Programme mit den Resultaten anderer Übersetzer verglichen.

6.1 Messumgebung

Die Messungen wurden auf einem PC mit Athlon XP 3000+ Prozessor, 1 GB Arbeitsspeicher und SuSE Linux 9.3 mit Kernel Version 2.6.11.4 durchgeführt.

Um die Anzahl der ausgeführten Instruktionen und Speicherzugriffe zu messen wurde eine modifizierte Version des Valgrind Speicherdebuggers [val] eingesetzt. Valgrind besitzt einen Modus, der x86 Code instrumentiert und alle Speicherzugriffe in einem Cache-Simulator ausführt. Dabei werden Speicherzugriffe und Instruktionen gezählt. Der Valgrind Sourcecode wurde modifiziert, da er in der ursprünglichen Version Instruktionen die eine Speicherstelle modifizieren, nur als Lesezugriff zählt.[1] Um die Ergebnisse verschiedener Übersetzer vergleichbar zu machen muss der Zugriff jedoch als Lese- und Schreibzugriff gezählt werden. Da durch die Instrumentierung die Geschwindigkeit der Programme um etwa Faktor 100 abnimmt, wurden für die Messungen mit Valgrind kleinere Testdaten gewählt.

6.1.1 Benchmarks

Die verwendeten Benchmarks sind:

- **queens:** Berechnet das n-Damen Problem. Für die Zeitmessungen wurde das 14 Damen Problem berechnet. Für die Valgrind-Messungen das 11 Damen Problem.

[1] Für eine Cache-Simulation ist es unerheblich ob erst ein Lese- und dann ein Schreibzugriff oder nur ein Lesezugriff auf eine bestimmte Speicherstelle erfolgt, da die Anzahl der Cache-Misses gleich bleibt.

- **quicksort:** Sortiert ein Reihung mit Zufallszahlen nach dem Quicksort verfahren. Für die Zeitmessungen wurde eine Reihung mit 10^8 Zahlen sortiert, für die Valgrind-Messungen eine Reihung der Größe 10^6.
- **max:** Berechnet einen Teil des h263 Algorithmus auf einer Reihung mit 2^{20} Elementen. Für die Zeitmessungen wurden 1000 Iterationen berechnet, für die Valgrind-Messungen 30.
- **sieve:** Berechnet alle Primzahlen bis zu einer bestimmten Marke, mittels des Sieb des Eratosthenes. Für die Zeitmessungen wurden die Primzahlen bis 5000 berechnet, für die Valgrind-Messungen bis 200.
- **gzip:** Der Benchmark 164.gzip aus der SPEC 2000 Benchmark Suite. Für die Zeitmessungen wurde ein 64MB Datenblock komprimiert und entpackt, für die Valgrind-Messungen ein 1MB großer Block.
- **bzip2:** Der Benchmark 256.bzip2 aus der SPEC 2000 Benchmark Suite. Für die Zeitmessungen wurde ein 64MB Datenblock komprimiert und wieder entpackt, für die Valgrind-Messungen ein 1MB großer Block.

6.1.2 Konfigurationen

Gemessen wurden die Resultate von folgenden Konfigurationen:

- **Belady:** Auslagern erfolgt nur mit dem Belady-Algorithmus.
- **Morgan:** Der Morgan-Algorithmus wird vor dem Belady-Algorithmus angewendet.
- **ILP:** Eine (fast) optimale Auslagerung die mit Hilfe eines ILP-Lösers berechnet ist. Siehe hierzu auch [Sza06].
- **GCC:** Der GNU C Übersetzer in der Version 3.3.5 mit *-O3 -fomit-frame-pointer -march=athlon-xp* Parametern.
- **ICC:** Der Intel C Übersetzer in der Version 8.0 mit *-O3 -mcpu=pentium4* Parametern.

6.2 Übersetzungszeit

Um das Laufzeitverhalten der Heuristik in Abhängigkeit von der Größe der Funktionen zu bestimmen wurde der Integer Teil der SPEC 2000 Benchmark

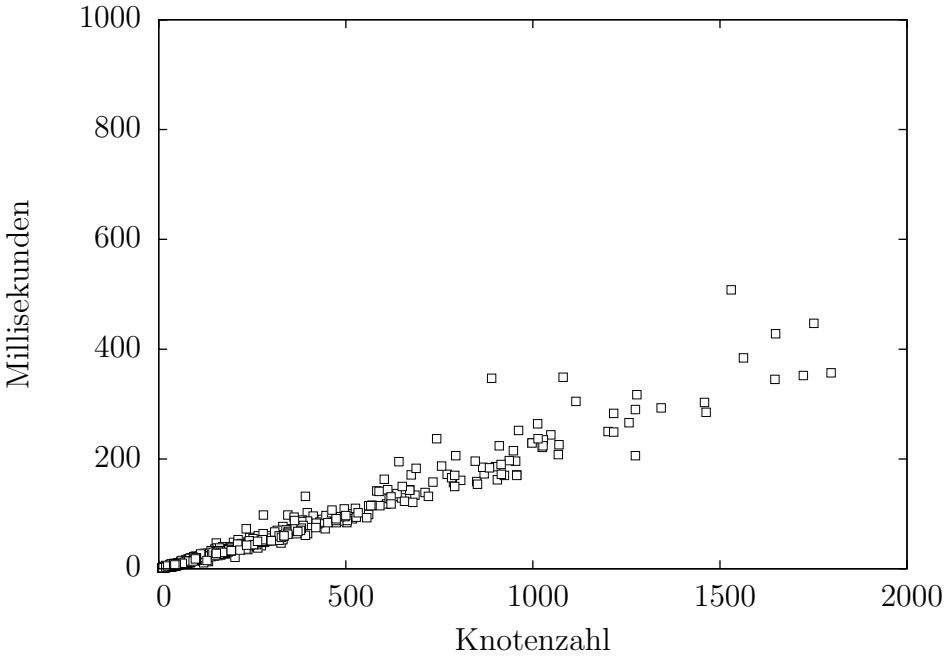

Abbildung 6.1: Verhältnis Funktionsgröße/Zeit zum Auslagern

Tabelle 6.1: Kosten in Tausend nach dem Modell aus Abschnitt 4.1 und Anteil an den Gesamtkosten

Suite übersetzt und dabei pro Funktion die Anzahl der Knoten und die zum Auslagern benötigte Zeit gemessen.

Die Ergebnisse sind in Abbildung 6.1 dargestellt. Auf der X-Achse ist die Größe der Funktionen aufgetragen, auf der Y-Achse die zum Auslagern benötigte Zeit in Millisekunden. Jede Messung wird durch einen Punkt im Diagramm dargestellt.

Die Laufzeit wächst in etwa linear mit der Funktionsgröße.

6.3 Kosten

Die Heuristiken versuchen die Kosten nach dem Modell aus Abschnitt 4.1 zu minimieren. Ein erster Indikator für die Qualität der Auslagerungen sind daher die durch die Auslagerungsbefehle entstandenen Kosten und deren Anteil an

den Gesamtkosten des Programms.

Tabelle 6.1 stellt die Kosten der Morgan-, Belady- und ILP-Konfigurationen dar, die durch die Auslagerungsbefehle entstehen. Ebenso werden die Gesamtkosten des Programms angegeben und der Anteil der Auslagerungsbefehle an den Gesamtkosten. Die Kosten für Speicherkopien sind in der Tabelle noch nicht berücksichtigt.

Die Auslagerungskosten der Morgan-Konfiguration sind verglichen mit den Kosten der Belady-Konfiguration meist deutlich niedriger. In Programmen mit vielfach geschachtelten Schleifen wie queens und quicksort sind die Ergebnisse mit Faktor 3 bzw. 58 erheblich besser. Bei den Programmen aus der SPEC2000 Suite ist die Morgan-Konfiguration etwa um den Faktor 2 besser.

Die Kosten der ILP Lösung sind etwa um den Faktor 2 besser als die Kosten der Morgan-Konfiguration. Zur Berechnung der ILP Lösungen wurde aber etwa die 25 Fache Zeit aufgewendet (nicht in der Tabelle dargestellt).

Beim Sieve Benchmark fällt auf, dass sich die Kosten sogar verbessern: Firm platziert alle Konstanten im Startblock. Durch Wiederberechnung werden manche Konstanten an Stellen mit niedrigerer Ausführungshäufigkeit "verschoben" was die Kosten verbessert.

6.4 Auslagerungsplätze Verschmelzen

Die Tabellen 6.2 und 6.2 stellen die Zahl der Auslagerungsplätze vor und nach der Verschmelzung dar. Zusätzlich ist die Zahl der Speicher-Φs und nicht eliminierten Speicherkopien angegeben.

Die Zahl der Auslagerungsplätze hat sich nach der Verschmelzung etwa halbiert.

Speicher-Φs treten relativ selten auf und die meisten der resultierenden Speicherkopien lassen sich vermeiden. Trotzdem ist die Zahl der Speicherkopien beim gzip Benchmark wohl verantwortlich dafür, dass die Lösung, die mit Belady's Algorithmus erzeugt wurde trotz höherer Kosten in den späteren Laufzeitmessungen besser abschneidet.

6.5 Rematerialisierung

Tabelle 6.3 zeigt die Zahl der platzierten Einlagerungsbefehle und die Zahl der Werte die rematerialisiert wurden und deren Prozentualer Anteil an allen Eingelagerten und rematerialisierten Werten. Es zeigt sich das zwischen 15 und 41 Prozent der wieder einzulagernden Werte neuberechnet werden können.

Mit Algorithmus von Belady:

Benchmark	A.-plätze	Verschmolzen	Speicher-Φs	Speicherkopien
bzip2	429	289	18	2
gzip	488	345	42	4
max	16	15	0	0
queens	10	10	0	0
quicksort	16	16	0	0
sieve	7	6	0	0

Mit Algorithmus von Morgan:

Benchmark	A.-plätze	Verschmolzen	Speicher-Φs	Speicherkopien
bzip2	404	280	45	8
gzip	546	378	194	39
max	16	15	0	0
queens	8	8	0	0
quicksort	13	13	0	0
sieve	5	5	0	0

Tabelle 6.2: Auslagerungsplätze vor und nach Verschmelzung, Speicherkopien

Rematerialisierungen sind also oft möglich und daher eine sinnvolle Ergänzung zum Auslagern in den Speicher.

Benchmark	Einlagerungen	Rematerialisierung	%
bzip2	1025	439	30,0
gzip	1228	698	36,2
max	16	10	38,5
queens	10	7	41,2
quicksort	20	11	35,5
sieve	10	2	16,7

Tabelle 6.3: Zahl der Rematerialisierungen

6.6 Speicherzugriffe und Instruktionen

Die mit Hilfe von Valgrind gemessene Anzahl an ausgeführten Speicherzugriffen und Instruktionen ist in Tabelle 6.4 zu sehen. Dabei wurden zum Vergleich auch die Werte des gcc und icc Übersetzers gemessen. Dabei ist zu beachten, dass der FIRM Übersetzer im Vergleich zu gcc und icc noch keinerlei Alias-Analyse besitzt und deshalb oft mehr Speicherzugriffe stattfinden. Ebenfalls hat das existierende Firm x86-Backend noch schwächen bei der Nutzung von Status Flags was die Zahl der ausgeführten Instruktionen erhöht.

Die Ergebnisse der Kostentabelle lassen sich Qualitativ nachvollziehen: Der Morgan Algorithmus liefert durchgängig bessere Ergebnisse als die Belady Konfiguration, erreicht aber noch nicht ganz das Niveau der ILP-Lösung. Das Ergebnis des Quicksort Benchmarks in dem der Morgan Algorithmus besser Abschneidet als die ILP Lösung zeigt, dass die Kosten nur eine Schätzung sind und in diesem Fall nicht der Realität entsprechen.

Die Anzahl der Speicherzugriffe der ILP und Morgan Lösungen liegt etwa auf dem Niveau der icc und gcc Übersetzer, was angesichts der fehlenden Alias-Analyse beachtlich ist. Die Zahl der Instruktionen ist leider durchgängig höher, was dann auch zu den im nächsten Abschnitt gezeigten im Vergleich längeren Ausführungszeiten beiträgt.

Benchmark	Speicherzugriffe			Instruktionen		
	Belady	Morgan	Verb.	Belady	Morgan	Verb.
bzip2	2120,9	1924,6	10,2%	4803,7	4562,2	5,3%
gzip	1253,2	1452,6	-13,7%	2582,5	2953,1	-12,5%
max	503,4	346,1	45,4%	1384,3	1258,4	10,0%
queens	64,5	25,1	157,1%	186,6	145,7	28,0%
quicksort	172,4	145,3	18,7%	381,8	322,8	18,3%
sieve	209,2	148,7	40,7%	533,2	453,1	17,7%

Tabelle 6.4: Speicherzugriffe und Instruktion in Millionen

6.7 Laufzeiten

Die Laufzeiten der übersetzten Programme sind in Tabelle 6.5 aufgeführt. Die Morgan Konfiguration zeigt um bis zu 15% bessere Ergebnisse als die Belady Variante.

Das schlechtere Abschneiden beim gzip Benchmark ist wohl auf die größere Zahl der Speicherkopien zurückzuführen.

Benchmark	Belady	Morgan	Verbesserung
bzip2	129,3 s	121,1 s	6,8%
gzip	61,6 s	72,4 s	-15,0%
max	17,2 s	16,2 s	6,4%
queens	12,8 s	10,6 s	21,2%
quicksort	27,2 s	23,6 s	15,6%
sieve	3,5 s	3,5 s	0,3%

Tabelle 6.5: Laufzeitverbesserung mit Morgan Algorithmus

6.8 Vergleich mit weiteren Übersetzern

Schliesslich wurden die Übersetzten Programme auch noch mit den Übersetzten Programmen aus den gcc und icc Übersetzern sowie der ILP Lösungen verglichen. Tabelle 6.6 zeigt die Anzahl der Ausgeführten Speicherzugriffe und Instruktionen, Tabelle 6.8 die Laufzeiten auf einem Athlon-XP System, Tabelle 6.7 die Laufzeiten auf einem Pentium 4 System.

Benchmark	Belady		Morgan		ILP	
	Mem.	Instr.	Mem.	Instr.	Mem.	Instr.
bzip2	2120,9	4803,7	1924,6	4562,2	1737,4	4338,2
gzip	1253,2	2582,5	1452,6	2953,1	611,6	1897,3
max	503,4	1384,3	346,1	1258,4	251,7	1164,0
queens	64,5	186,6	25,1	145,7	10,1	130,7
quicksort	172,4	381,8	145,3	322,8	123,9	316,2
sieve	209,2	533,2	148,7	453,1	148,7	432,8

Tabelle 6.6: Ausgeführte Instruktionen und Speicherzugriffe in Millionen

Bei den Laufzeiten zeigt sich eine Schwäche des Kostenmodells: Prozessorspezifische eigenheiten wie Pipelines und das blockieren von Ausführungseinheiten zeigen, dass die Anzahl der ausgeführten Befehle und Speicherzugriffe nicht immer mit der Laufzeit korrelieren. Eine Erweiterung oder Nachbearbeitung der Ergebnisse um diese Effekte zu berücksichtigen wäre also sinnvoll.

Die Erzielten Laufzeiten sind nur unwesentlich Schlechter als die Laufzeiten der anderen Übersetzer. In einigen Benchmarks werden die Werte der gcc und icc Übersetzer übertroffen. Angesichts der Tatsache, dass der Firm Übersetzer einige Optimierungen wie Alias-Analyse nicht durchführt und das x86 Backend des Firm-Übersetzers erst etwa 2 Jahre in Entwicklung ist, lässt sich vermuten das durch weitere Optimierungen und Detailarbeit dir Ergebnisse von gcc und igcc übertroffen werden können.

Benchmark	Belady	Morgan	ILP	GCC	ICC
bzip2	129,3 s	121,1 s	123,6 s	103,6 s	135,6 s
gzip	61,6 s	72,4 s	51,5 s	67,2 s	49,5 s
max	17,2 s	16,2 s	14,8 s	9,9 s	10,7 s
queens	12,8 s	10,6 s	11,9 s	10,6 s	10,5 s
quicksort	27,2 s	23,6 s	23,7 s	23,8 s	21,6 s
sieve	3,5 s	3,5 s	3,5 s	3,1 s	4,2 s

Tabelle 6.7: Laufzeiten auf Pentium 4 2,4 Ghz

Benchmark	Belady	Morgan	ILP	GCC	ICC
bzip2	129,3 s	121,1 s	123,6 s	103,6 s	135,6 s
gzip	61,6 s	72,4 s	51,5 s	67,2 s	49,5 s
max	17,2 s	16,2 s	14,8 s	9,9 s	10,7 s
queens	12,8 s	10,6 s	11,9 s	10,6 s	10,5 s
quicksort	27,2 s	23,6 s	23,7 s	23,8 s	21,6 s
sieve	3,5 s	3,5 s	3,5 s	3,1 s	4,2 s

Tabelle 6.8: Laufzeiten auf Athlon-XP 3000+

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

In dieser Arbeit wurde das Auslagerungsproblem für Programme in SSA-Form untersucht, ein Kostenmodell aufgestellt und die prinzipielle Vorgehensweise zum Auslagern auf Programme in SSA-Form vorgestellt. Anschliessend wurden konkrete Heuristiken zum Auslagern entwickelt. Ein Auslagern mit dem Algorithmus von Belady erzeugt gute Auslagerungen für lange Grundblöcke und geradlinigen Code, um eine gute Auslagerung im Hinblick auf Schleifen zu erreichen sollte der Algorithmus von Morgan vor dem Auslagern mit Belady's Algorithmus ausgeführt werden.

Die Messungen belegen, dass die entwickelten Verfahren praktikabel und konkurrenzfähig sind.

7.2 Ausblick

Die folgenden Abschnitte zeigen Möglichkeiten auf, wie die vorgestellten Verfahren in zukünftigen Arbeiten erweitert werden können.

7.2.1 Φ-Kaskaden

Ein Problem bei den entwickelten Verfahren ist, dass nur auf SSA-Werten ausgelagert wird. Eine Φ-Funktion ist aber keine Verwendung sondern nur eine Auswahl von Werten. Das Programm in Abbildung 7.1 erzeugt in der bedingten Anweisung in Zeile 5,6 einen um eins erhöhten Wert von x_1. Die Definitionen von x_1 und x_3 sind nur eine Auswahl von Werten mit Φ-Funktionen.

Die Heuristiken betrachten allerdings nur einzelne SSA-Werte. Das Ergebnis des Algorithmus von Belady in Abbildung 7.2 (a) ist nicht optimal: Die Auslagerung geschieht im Hauptteil der Schleife. Allerdings lassen sich der Ein- und Auslagerungsbefehl durch ein Auslagern der Φ-Funktionen in die seltener ausgeführte bedingte Anweisung verschieben (Abbildung 7.2 (b)).

Beim Auslagern von Werten von Φ-Funktionen sollte also festgestellt werden, ob es nicht sinnvoller ist, die ganze Φ-Funktion und weitere Φ-Vorgänger

```
1: x_0 ←
2: loop:
3: x_1 ← Φ(x_0, x_3)
4: ... hoher Registerdruck, x_1 muss
   ausgelagert werden ...
5: if ... then
6:     x_2 ← x_1 + 1
7: end if
8: x_3 ← Φ(x_1, x_2)
9: if ... then
10:    GOTO(loop)
11: end if
```

Abbildung 7.1: Problemfall für die Heuristiken

```
1: x_0 ←
2: loop:
3: x_1 ← Φ(x_0, x_3)
4: M_1 ← SPILL(x_1)
5: ... hoher Registerdruck, x_1 muss
   ausgelagert werden ...
6: if ... then
7:     x_2 ← x_1 + 1
8: end if
9: x_4 ← RELOAD(M_1)
10: x_3 ← Φ(x_4, x_2)
11: if ... then
12:    GOTO(loop)
13: end if
```

(a) Auslagerung mit Heuristik

```
1: x_0 ←
2: M_1 ← SPILL(x_0)
3: loop:
4: M_2 ← Φ(M_1, M_4)
5: ... hoher Registerdruck ...
6: if ... then
7:     x_1 ← RELOAD(M_2)
8:     x_2 ← x_1 + 1
9:     M_3 ← SPILL(x_2)
10: end if
11: M_4 ← Φ(M_2, M_3)
12: if ... then
13:    GOTO(loop)
14: end if
```

(b) Optimale Auslagerung

Abbildung 7.2: Auslagerung des Problemfall mit Morgan+Belady und Optimale Lösung

auszulagern.

7.2.2 Geschicktes Graphenclustern

Der Algorithmus von Morgan ist nicht auf Schleifen beschränkt sondern lässt auf beliebige geschachtelte Regionen des Quellprogramms anwenden. Jede Region mit hoher Ausführungshäufigkeit, die nur mit Grundblöcken niedrigerer Ausführungshäufigkeit verbunden ist, ist ein guter Kandidat für den Algorithmus. Wenn es gelingt den Graph nach weiteren Kriterien als dem Schleifenbaum zu clustern, so gibt es mehr Möglichkeiten zum Auslagern.

7.2.3 Rematerialisierungen

Der Algorithmus von Belady optimiert nur daraufhin, dass Werte möglichst lange ausgelagert bleiben. Hierbei werden Einsparungen durch mögliche Rematerialisierungen oder mehrfaches Auslagern nicht beachtet. Diese Möglichkeiten sollten bei der Auswahl der auszulagernden Werte beachtet werden.

7.2.4 Nächste Verwendungen

Der Abstand für die nächste Verwendung im Belady Algorithmus sind ungenau für Werte, die in den nächsten Grundblöcken liegen. Die Formel aus Abschnitt 4.5.1 ignoriert die Ausführungshäufigkeiten des Programms. So gibt häufig eine nächste Verwendung direkt nach einer Schleife den Ausschlag, das Werte die eigentlich in der Schleife nicht verwendet werden in Registern gehalten werden. Um dieses Problem zu vermeiden wurde der Morgan-Algorithmus eingesetzt.

Gelingt es, die Ausführungshäufigkeiten des Programms in die Werte für die nächste Verwendung einfliessen zu lassen, so sollte der Belady-Algorithmus bereits ähnliche oder bessere Entscheidungen treffen als der Morgan-Algorithmus. Dies würde insbesondere die zweimalige Berechnung der Lebendigkeiten beim kombinierten Belady-Morgan-Algorithmus vermeiden.

Abbildungsverzeichnis

2.1 Beispiel für einen Steuerflussgraph 4
2.2 Steuerflussgraph mit Schleifenbaum 6
2.3 Programm aus Abbildung 2.1 in SSA-Form 8

3.1 Schema des Chaitin/Briggs Registerzuteilers 11
3.2 Diamant Interferenz Graph 12

4.1 Auslagern einer Φ-Funktion 18
4.2 Problemfälle für den Belady Algorithmus 22
4.3 Verschmelzen von Auslagerungsplätzen vermeidet Speicherkopien 25

5.1 Quelltext des Graphen aus Abbildung 5.2 27
5.2 Firm Graph des Programms aus Abbildung 5.1 28
5.3 Phasen des Backend 29

6.1 Verhältnis Funktionsgröße/Zeit zum Auslagern 41

7.1 Problemfall für die Heuristiken 50
7.2 Auslagerung des Problemfall mit Morgan+Belady und Optimale Lösung 50

Tabellenverzeichnis

6.1 Kosten in Tausend nach dem Modell aus Abschnitt 4.1 und Anteil an den Gesamtkosten 41
6.2 Auslagerungsplätze vor und nach Verschmelzung, Speicherkopien 43
6.3 Zahl der Rematerialisierungen 44
6.4 Speicherzugriffe und Instruktion in Millionen 45
6.5 Laufzeitverbesserung mit Morgan Algorithmus 45
6.6 Ausgeführte Instruktionen und Speicherzugriffe in Millionen . . 46
6.7 Laufzeiten auf Pentium 4 2,4 Ghz 46
6.8 Laufzeiten auf Athlon-XP 3000+ 47

Algorithmen

1 Belady Auslagerungsalgorithmus 32
2 Auffinden unbenutzt durchlebender Werte 35
3 Algorithmus von Morgan 36

Literaturverzeichnis

[BCT94] Briggs, Preston ; Cooper, Keith D. ; Torczon, Linda: Improvements to Graph Coloring Register Allocation. In: *ACM Transactions on Programming Languages and Systems* 16 (1994), Mai, Nr. 3, S. 428–455

[BDEO97] Bergner, Peter ; Dahl, Peter ; Engebretsen, David ; O'Keefe, Matthew T.: Spill Code Minimization via Interference Region Spilling. In: *SIGPLAN Conference on Programming Language Design and Implementation*, 1997, S. 287–295

[BDGR05] Bouchez, Florent ; Darte, Alain ; Guillon, Christophe ; Rastello, Fabrice: Register allocation and spill complexity under SSA / LIP. Version: August 2005. `http://www.ens-lyon.fr/LIP/Pub/rr2005.php`. ENS Lyon, France, August 2005 (RR2005-33). – Forschungsbericht

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

[CFR+91] Cytron, Ron ; Ferrante, Jeanne ; Rosen, Barry K. ; Wegman, Mark N. ; Zadeck, F. K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In: *ACM Transactions on Programming Languages and Systems* 13 (1991), Oktober, Nr. 4, S. 451–490

[CH84] Chow, Frederick ; Hennessy, John: Register allocation by priority-based coloring. In: *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*. New York, NY, USA : ACM Press, 1984. – ISBN 0–89791–139–3, S. 222–232

[Cha82] Chaitin, G. J.: Register allocation & spilling via graph coloring. In: *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*. New York, NY, USA : ACM Press, 1982. – ISBN 0–89791–074–5, S. 98–105

[FL98] Farach, Martin ; Liberatore, Vincenzo: On local register allocation. In: *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 1998. – ISBN 0–89871–410–9, S. 564–573

[GGP03] Guo, Jia ; Garzarán, María J. ; Padua, David: The Power of Belady's Algorithm in Register Allocation for Long Basic Blocks. In: *LCPC Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, 2003

[HGG05] Hack, Sebastian ; Grund, Daniel ; Goos, Gerhard: Towards Register Allocation for Programs in SSA-form / IPD Goos. Version: September 2005. `http://www.info.uni-karlsruhe.de/~hack/ra_ssa.pdf`. 2005. – Forschungsbericht

[LBBG05] Lindenmaier, Götz ; Beck, Michael ; Boesler, Boris ; Geiss, Rubino: Firm, an Intermediate Language for Compiler Research. Version: März 2005. `http://www.ubka.uni-karlsruhe.de/cgi-bin/pslist?path=ira/2005`. University of Karlsruhe, März 2005 (2005-8). – Forschungsbericht. – 19 S.

[LT79] Lengauer, Thomas ; Tarjan, Robert E.: A fast algorithm for finding dominators in a flowgraph. In: *ACM Trans. Program. Lang. Syst.* 1 (1979), Nr. 1, S. 121–141. `http://dx.doi.org/http://doi.acm.org/10.1145/357062.357071`. – DOI http://doi.acm.org/10.1145/357062.357071. – ISSN 0164–0925

[Mak04] Makarov, Vladimir N.: Fighting register pressure in GCC. In: Hutton, Andrew J. (Hrsg.) ; Donovan, Stephanie (Hrsg.) ; Ross, C. C. (Hrsg.): *Proceedings of the GCC Developers Summit*, 2004, 85–103

[Mor98] Morgan, Robert: *Building an Optimizing Compiler*. Digital Press, 1998. – ISBN 1–55558–179–X

[NNH05] Nielson, Flemming ; Nielson, Hanne R. ; Hankin, Chris: *Principles of Program Analysis*. second. Springer, 2005. – ISBN 3–450–65410–0

[RWZ88] Rosen, Barry K. ; Wegman, Mark N. ; Zadeck, F. K.: Global Value Numbers and Redundant Computations. In: *Conference*

Record of the 15th Annual ACM Symposium on Principels of Programming Languages (POPL '88). San Diego, CA, USA : ACM-SIGACTm ACM-SIGPLAN, Januar 1988. – ISBN 0–89791–252–7, S. 12–27

[Sza06] Szalkowski, Adam: *Rematerialisierung mittels ganzzahliger linearer Optimierung in einem SSA-basierten Registerzuteiler*, Universität Karlsruhe, Diplomarbeit, Oktober 2006

[val] *Valgrind: A Simulation-based debugging and profiling tool*. `http://www.valgrind.org`,

[WL94] Wu, Youfeng ; Larus, James R.: Static Branch Frequency and Program Profile Analysis. In: *International Symposium on Microarchitecture (MICRO-27)*, 1994, S. 1–11