DIPLOMARBEIT

Vergleich von Registerallokationsalgorithmen

ausgeführt am Institut für Computersprachen
der Technischen Universität Wien

unter Anleitung von
o. Prof. Dipl.-Ing. Dr. Manfred Brockhaus
und
Univ.-Ass. Dipl.-Ing. Martin Anton Ertl

durch

Wolfgang Ambrosch

Jägerhausgasse 16
2500 Baden

Wien, am 3. November 1995

Kurzfassung

Die Registerallokation ist in einem modernen Compiler eine der wichtigsten Optimierungen. Die meisten heute verwendeten Registerallokationsverfahren basieren auf der 1981 von Gregory Chaitin erstmals implementierten Methode des Graphenfärbens. Mit diesen Verfahren beschäftigt sich die vorliegende Arbeit.

Nach einer Erklärung der grundlegenden Ideen und Begriffe stellen wir anhand einer konkreten Implentierung ein Phasenmodell für die Registerallokation vor. Wir analysieren bekannte Registerallokationsverfahren und zeigen, wie sich ihre Bestandteile in unser Phasenmodell integrieren und dort miteinander kombinieren lassen. Wichtige Details wie etwa die Anpassung eines Registerallokators an verschiedene Zielmaschinen, die Handhabung verschiedener Registerklassen oder die Behandlung von Unterprogramm-Aufrufkonventionen werden in einem eigenen Kapitel mit Hilfe von Beispielen dargestellt.

Wir vergleichen die Verfahren anhand einer größeren Testsuite nach den Kritierien „Resourcenbedarf bei der Allokation“, „Ausnutzung der vorhandenen Register“ und „Verhalten beim Auslagern von Pseudoregistern“.

Eine genaue Beschreibung der von uns als Grundlage verwendeten Verfahren rundet die Arbeit ab.

Abstract

Register allocation is one of the most important optimizations in modern compilers. Most of the algorithms used today are based on the work of Gregory Chaitin who was the first to implement a graph coloring register allocator in 1981. This thesis concentrates on register allocation via graph coloring.

After explaining the fundamental ideas we present a basic framework for register allocation. We analyze existing algorithms and show how their parts can be integrated in and combined with the help of our framework. Important but often neglected details as the retargeting of register allocators, the handling of different classes of registers, or the effects of different subprogram linkage conventions are covered in a separate chapter.

Experimental results on the effectiveness of the described methods for a large suite of test programs are given. Compile-time characteristics, register usage and the behavior in the case of spilling were some of the issues we investigated.

An exact description of the papers our work was based on completes the thesis.

Danksagungen

Zunächst möchte ich mich den Mitarbeitern des Instituts für praktische Informatik bedanken: bei Prof. Manfred Brockhaus, dessen Art, das Institut zu leiten, die Arbeit dort zum Vergnügen macht, bei meinem Betreuer Martin Anton Ertl, auf den viele wesentliche Ideen in dieser Arbeit zurückgehen und bei Andreas Krall, der nicht nur auf die ausgefallensten Fragen zu den eigenartigsten Mikroprozessoren sofort eine Antwort weiß. Es muß sich wohl um ein Versehen der Natur handeln, wenn dieses Institut nicht von Diplomanden gestürmt wird.

Ohne meinen langjährigen Freund und Diplomandenkollegen Felix Beer gäbe es keinen Compiler zu meinem Registerallokator. Viel Geduld, literweise Tee und die vielen tausend Zeilen, die er zu diesem Projekt beigesteuert hat, haben es erst möglich gemacht. Andreas Stiasny, der sich rechtzeitig bereit gefunden hat, diese Arbeit durchzulesen, hat dabei geholfen, den Text in vielen Details zu verbessern.

Was andere für mich getan haben, läßt sich nicht so leicht in Worte fassen. Nur mit Hilfe meiner Angehörigen und Freunde war es mir möglich, diese Arbeit erfolgreich zu beenden.

Allen diesen Menschen gilt mein aufrichtiger Dank. Das kollektive „wir“, das in diesem Text verwendet wird, schließt sie alle mit ein.

Baden, am 3. November 1995

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

1.1 Ziel

Das Problem der effizienten Registerallokation hat in den letzten Jahren zunehmend an Bedeutung gewonnen. Einerseits erleichtern moderne Prozessorarchitekturen mit ihrer vergleichsweise großen Anzahl von Maschinenregistern diese Aufgabe gegenüber älteren Architekturen spürbar, andererseits wollen die vielen Register auch genützt sein — moderne RISC-Architekturen kommen ja derzeit mit mindestens 32 Ganzzahl- und 32 Fließkommaregistern.

Mit dem Angebot an Registern steigt allerdings auch der Bedarf; vor allem im technisch-wissenschaftlichen Bereich sind „registerhungrige" Programme nicht unüblich, durch die immer häufiger werdende Verwendung von Programmgeneratoren, die Quelltexte erzeugen und deren Optimierung dann den Compilern überlassen, und nicht zuletzt durch superskalare RISC-Prozessoren, bei denen das erforderliche „Code Scheduling" die Anzahl der benötigten Register wieder in die Höhe treibt, wächst die Bedeutung von Compilern mit effizienten Registerallokatoren zusehends.

Ausgehend von einem verfügbaren Compiler-Frontend für die Sprache C, dem `lcc`, wurde daher ein experimenteller Codegenerator entwickelt, der mit einer Reihe von modernen Optimierungsverfahren arbeitet.

Daher wurden mit dieser Diplomarbeit zwei Ziele verfolgt:

1. Einerseits sollte für den Compiler ein funktionierender und möglichst leicht erweiterbarer Registerallokator entstehen. Der Registerallokator sollte sowohl portabel sein (sich also leicht auf einer anderen Maschine übersetzen lassen) als sich auch leicht an unterschiedliche Eigenschaften verschiedener Maschinen, für die Codegeneratoren geplant oder in Arbeit waren[1], anpassen lassen. Die Laufzeitoptimierung des Compilers selbst war ausdrücklich nicht Ziel der Arbeit.

[1] Hier sind vor allem die RISC-Architekturen MIPS, SPARC, DEC-Alpha, POWER-RISC oder HP-PA zu erwähnen

2. Andererseits bot es sich bei der Gelegenheit an, einen Vergleich von verschiedenen Registerallokatoren, die alle mit der Methode des Graphenfärbens arbeiten, durchzuführen, da es in der publizierten Literatur keinen einigermaßen vollständigen Vergleich gibt. Dabei sollten nicht nur die ausgewählten Methoden implementiert werden, sondern soweit wie möglich auch die Unterschiede zwischen den einzelnen Verfahren in möglichst frei kombinierbare Unterprogramme verpackt werden.

Diese Arbeit widmet sich also dem Problem der Registerallokation. Der Rest des Codegenerators ist in [Bee93] beschrieben.

1.2 Registerallokation und globale Optimierung

Das „Backend“ eines Compilers, dem der Registerallokator zuzurechnen ist, besteht in der Regel aus mehreren Phasen. Jede Phase führt eine oder mehrere Transformationen oder Optimierungen auf dem Zwischencode aus, wobei in vielen Fällen die bestmögliche Anordnung der Phasen nicht eindeutig definiert werden kann („phase ordering problem“).

Auch der Registerallokator ist eine dieser Optimierungsphasen, und zwar, wie Hennesy und Patterson in [HP90] anmerken, „eine der wichtigsten — wenn nicht die wichtigste überhaupt“[2]. Dieser Fall liegt auch beim Registerallokator vor, wo im Normalfall zwei Alternativen in Betracht gezogen werden müssen:

1. Der Registerallokator kann vor der Maschinencodeerzeugung („Code selection“) laufen

2. Der Registerallokator kann nach der Maschinencodeerzeugung laufen.

Führt man die Registerallokation vor der Maschinencodeerzeugung, also noch am Zwischencode, durch, wird das „Retargeting“, also das Anpassen des Compilers an eine andere Zielmaschine, etwas erleichtert, da der Registerallokator in diesem Fall auf etwaige Besonderheiten, die von der Codeerzeugung eingeführt werden, keine Rücksicht mehr nehmen muß[3]. Diese Besonderheiten ziehen in fast allen Fällen zwar nur kleine Änderungen im Registerallokator nach sich, allerdings führen diese „kleinen Änderungen“ oft zu schwer entdeckbaren Fehlern.

[2]Im Original: „ ...register allocation ...is one of the most important — if not the most important — optimizations“

[3]Natürlich hängt das auch vom verwendeten Zwischencode ab — ein bestimmter Zwischencode ist im Regelfall für manche Klassen von Maschinen besser geeignet als für andere. Allerdings existieren in dieser Hintersicht kaum mehr Unterschiede, wenn man sich auf moderne RISC-Prozessoren beschränkt. Trotzdem sollte man immer einen Merksatz im Hinterkopf behalten: „There is no such thing as a machine-independent optimization!“, wie Wulf in [Deb87] zitiert wird.

Auf der anderen Seite muß man unter Umständen etwas Genauigkeit opfern — nicht immer läßt sich garantieren, daß der Registerbedarf des Zwischencodes sich exakt mit dem Registerbedarf des späteren Maschinencodes deckt[4]; vor allem bei Maschinen mit komplexen Adresierungsarten können sich hier Unterschiede ergeben.

Führt man die Registerallokation nach der Maschinencodeerzeugung aus, ist natürlich der Registerbedarf exakt feststellbar; allerdings werden auch hier komplexere Adressierungsarten zum Problem: Bei der Erzeugung des Maschinencodes sollte schon bekannt sein, welche Pseudoregister in Maschinenregistern gehalten werden können und welche nicht, weil das ja auch die Auswahl der Adressierungsart bestimmt. Da aber erst der Registerallokator für die Zuweisung von Maschinenregistern zuständig ist, sind auch die Adressierungsarten während der Maschinencodeerzeugung in vielen Fällen noch nicht bestimmbar; daher müssen Teile der Codeerzeugung vom Registerallokator oder einer weiteren, nachfolgenden Phase erledigt werden.

Da das leichte Anpassen an neue Maschinen eine wesentliche Vorgabe für unsere Arbeit war, wird die Registerallokation bei unserem Compiler vor der eigentlichen Maschinencodeerzeugung durchgeführt. Allerdings wird in einer eigenen, vor dem Registerallokator laufenden „Vorselektion“ der Zwischencode so umgeschrieben, daß der Registerbedarf so weit wie möglich dem des Maschinencodes entspricht.

1.3 Graphenfärben – ein modernes Verfahren

Graphenfärben („Graph Coloring“) ist ein relativ neues Verfahren zur Registerallokation. Das graphentheoretische Problem, das dem Verfahren seinen Namen gegeben hat, besteht darin, die Knoten eines ungerichteten Graphen mit einer vorgegebenen Anzahl von Farben so zu färben, daß je zwei durch eine Kante verbundene Knoten in keinem Fall die gleiche Farbe bekommen[5]. In einem Registerallokator stehen die Knoten für Variablen, die ein Maschinenregister zugewiesen bekommen sollen, die Farben für die vorhandenen Maschinenregister und eine Kante bedeutet, daß die verbundenen Knoten nicht das gleiche Register zugewiesen bekommen dürfen. Auch Algorithmen, die nur feststellen, ob ein gegebener Graph mit einer gegebenen Anzahl an Farben eingefärbt werden kann, sind bereits NP-vollständig. Alle Lösungen für diese Probleme können daher nur auf heuristischem Weg gefunden werden.

[4] Bei den Prozessoren der MIPS-Familie ist ein Maschinenregister für den Assembler reserviert; der Assembler stellt mehr Instruktionen zur Verfügung, als die Maschine tatsächlich hat. Einige Instruktionen der Assemblersprache werden daher vom Assembler in eine Sequenz von Maschineninstruktionen übersetzt, wobei unter Umständen dieses reservierte Register benötigt wird.

[5] Kurze Einführungen in die notwendigen Grundbegriffe der Graphentheorie finden sich in [KZM85] und [Bö81].

Obwohl das Verfahren schon in den frühen siebziger Jahren vorgeschlagen wurde, erfolgte die erste Implementierung erst Anfang der achtziger Jahre durch Gregory Chaitin [CAC+81], [Cha82] im IBM-Forschungslaboratorium in Yorktown Heights. Ausschlaggebend dafür war wahrscheinlich, daß die relativ massiven Datenstrukturen, die ein solcher Registerallokator benötigt, erst zu diesem Zeitpunkt problemlos in den Hauptspeichern unterzubringen waren.

Vor allem in den Jahren ab 1985 erschienen eine ganze Reihe von Arbeiten, welche aufbauend auf den Ergebnissen von Chaitin einige Verbesserungen des grundlegenden Verfahrens brachten. Heute beinhalten sehr viele kommerzielle und nichtkommerzielle Compiler einen Registerallokator, der nach dem Prinzip des Graphenfärbens arbeitet. Diese weite Verbreitung hat mehrere Gründe:

- Die Methodes des Graphenfärbens ist relativ einfach; wenn man einige der vorgeschlagenen Verbesserungen außer acht läßt, kann ein derartiger Registerallokator schnell implementiert werden[6].

- Aus dem oben angeführten Grund sind derartige Allokatoren im allgemeinen auch kurz und übersichtlich.

- Für Registerallokatoren gibt es im Prinzip drei verschiedene "Komplexitätsstufen":

  Lokal. Maschinenregister werden auf der Ebene der Instruktionsblöcke vergeben.

  Global. Bei der Vergabe von Maschinenregistern wird immer ein ganzes Unterprogramm auf einmal betrachtet.

  Interprozedural. Der Inhalt mehrerer Unterprogramme wird bei der Registervergabe in Betracht gezogen.

  Graphenfärben alloziert die Register auf einer „globalen“ Basis. Gegenüber der „lokalen“ Sicht anderer Allokatoren bringt dies einige Verbesserungen mit sich ohne daß man die vergrößerte Komplexität der interprozeduralen Allokation in Kauf nehmen muß.

- Die Registerallokation arbeitet auf eigenen Datenstrukturen, die in einem eigenen Durchlauf aus dem Zwischencode heraus aufgebaut werden. Dadurch kann das in der Vorbereitungsphase notwendige, maschinenabhängige Lesen des Zwischencodes sauber von der eigentlichen Registerallokation getrennt werden.

[6]Briggs gibt zum Beispiel die Zeit von etwa zwei Mannwochen für die Integration eines lauffähigen Registerallokators in einen bereits fertigen Compiler an [BCKT89]; wir haben etwas länger gebraucht, weil der Rest des Codegenerators zur Zeit der Entstehung des Registerallokators ebenfalls erst geschrieben wurde; trotzdem haben auch wir deutlich weniger als vier Mannwochen gebraucht.

1.4 `lcc` – ein portabler Compiler

Basis unserer Implementierung war das `lcc`-Frontend [FH90][FH91a][FH91b], das die lexikalische, syntaktische und semantische Analyse für die Quellsprache `ANSI C` erledigt und dem Codgenerator einen annotierten Zwischencode überläßt.

Das `lcc`-Frontend hat einige Eigenschaften, die auch das Design unseres Codegenerators nachhaltig beeinflußt haben. Im Gegensatz zu anderen Compilern, welche die Codeerzeugung eindeutig vom Rest des Compilers abschotten, indem sie in einem separat aufgerufenen Programm untergebracht wird, kommunizieren die einzelnen Phasen des Compilers beim `lcc` durch simple Unterprogrammaufrufe; die einzelnen Phasen sind also eng miteinander gekoppelt, wodurch der Compiler insgesamt kompakter und schneller als bei anderen Implementationen ist.

Die Schnittstelle zwischen dem Codegenerator und dem Rest des Compilers besteht aus 19 Funktionen, einigen gemeinsamen Datenstrukturen sowie einem DAG-Zwischencode mit 36 Operatoren[7]. Sie ist dadurch kompakt und überschaubar; das wurde vor allem dadurch erreicht, daß ausschließlich die Anforderungen der Quellsprache `C` und die von modernen (32 Bit) RISC-Prozessoren als Zielmaschinen im Design berücksichtigt wurden. So wird zum Beispiel angenommen, daß die Datentypen `int`, `long` und `unsigned` auf die selbe Art repräsentiert werden und daß es nur einen Typ von Zeigern gibt. Außerdem wurden einige DAG-Operatoren, die mit Hilfe von anderen dargestellt werden können, weggelassen [FH90, S. 34].

Während der DAG-Zwischencode bei einfachen Codegeneratoren durchaus Vorteile bringt, hat er bei unserem optimierenden Compiler vieles nicht unbedingt erleichtert. Der Grund dafür liegt vor allem darin, daß die in den DAGs gehaltene Information nicht immer in dieser Form brauchbar ist und daß Codetransformationen auf einem einfachen, linearen Zwischencode oft einfacher durchzuführen sind, weil beim Umschreiben weniger Details zu beachten sind. Mehr zu diesem Thema finden Sie in [Bee93].

Auf der anderen Seite war die einfache Schnittstelle durchaus ein Gewinn für das Projekt; viele Details, die zum Beispiel bei Änderungen im populären GNU-Compiler eher verwirrend wirken, sind beim `lcc` gar nicht erst zu beachten. Für jedes weitere, auf den `lcc` und unsere Arbeit aufbauende Projekt muß aber klar sein, daß ...

- ... die enge Kopplung zwischen den Compilerphasen bei der eventuellen späteren Verwendung von neuen `lcc`-Versionen Änderungen auch im Codegenerator notwendig machen könnte.
- ... und daß die Einschränkungen bezüglich der verfügbaren Operatoren und Datentypen nur schwer zu umgehen sind.

[7] Einige zusätzliche Operatoren wurden von uns hinzugefügt.

1.5 Notation

An vielen Stellen werden in dieser Arbeit kleine Maschinencodestücke als Beispiele gebracht. Dabei gelten folgende Konventionen:

- Alle Assemblerinstruktionen sind in der Form

  Opcode Zieloperand, Quelloperanden

  notiert.

- Als Assemblerinstruktionen werden nur `mov` (Verschiebebefehl), `add` und `mul` verwendet.

- Operandennamen, die mit dem Buchstaben `R` beginnen, bezeichnen echte Maschinenregister, solche, die mit dem Buchstaben `P` beginnen, bezeichnen Pseudoregister und Kleinbuchstaben stehen für Adressen im Hauptspeicher.

- Ein Pluszeichen (`+`) bei einem Quelloperand bezeichnet die letzte Verwendung dieses Operanden.

Kapitel 2

Grundlagen

Bevor wir unseren Registerallokator im Detail beschreiben, wollen wir zunächst einige von uns verwendete Grundbegriffe vorstellen und die prinzipielle Arbeitsweise des Registerallokators anhand von Beispielen erklären. Dieses Kapitel ist einerseits für alle jene bestimmt, die mit der Registerallokation durch Graphenfärben noch nicht vertraut sind. Da wir auch versucht haben, die in unseren Quellen verwendeten englischen Vokabeln und Begriffe auf deutsch wiederzugeben, sollte dieses Kapitel andererseits auch helfen, etwaige Mißverständnisse mit unseren Übersetzungen auszuräumen.

2.1 Vorbereitung

Die Aufgabe des Registerallokators ist es, die Zwischendarstellung eines Programms, die unter der Annahme der Verfügbarkeit einer unbegrenzten Anzahl an *Pseudoregistern* aller möglichen Registerklassen generiert wurde, auf einem konkreten Zielprozessor mit einer begrenzten Anzahl an Maschinenregistern abzubilden.

Bevor mit der eigentlichen Registerallokation begonnen werden kann, muß das Programm zunächst so umgeformt werden, daß alle Werte, denen später Maschinenregister zugeordnet werden sollten, in Pseudoregistern gehalten werden. Dazu gehören zum Beispiel globale und lokale Variablen, vom Compiler generierte Zwischenergebnisse, einzelne Mitglieder von Strukturen und einzelne Feldelemente. Welche Werte das im Einzelfall tatsächlich sind, hängt von der Vorarbeit ab, die von den vorangegangenen Phasen des Codegenerators geleistet wurde. Auf jeden Fall ist die Umwandlung in die Pseudoregister-Darstellung von der Zielmaschine abhängig; schon hier muß darauf Rücksicht genommen werden, wie sich die Datentypen der Quellsprache mit Hilfe von Maschinenregistern darstellen lassen[1].

[1] In vielen Fällen wird diese Abhängigkeit aber in der Praxis kaum auffallen, weil sich moderne Prozessoren bezüglich der zur Verfügung gestellten Datentypen sehr ähnlich sind.

Als zweiter Schritt muß die Datenflußanalyse wesentliche Informationen sammeln und diese Informationen auf die einzelnen Instruktionsblöcke und Instruktionen verteilen. Um die Effizienz des Allokators zu erhöhen, sollten die Definitionsbereiche der einzelnen Variablen in zusammenhängende *aktive Bereiche* („live ranges") aufgeteilt werden. Ein solcher aktiver Bereich reicht von der Definition einer Variablen bis zur letzten Benützung des vorher definierten Wertes. Jeder aktive Bereich entspricht dabei einem Pseudoregister; einer Variablen werden auf diese Art im allgemeinen mehrere Pseudoregister zugeordnet. Ein dafür geeigneter Algorithmus ist in [CFR+89] beschrieben. Er führt zur Darstellung des Programms in SSA-Form („*Static Single Assignment Form*"); in dieser Darstellungsform gehört zu jedem Pseudoregister genau eine Definition und eine oder mehrere Verwendungen.

Außerdem ist eine Analyse der aktiven Bereiche der einzelnen Pseudoregister notwendig. Jedem Instruktionsblock sollte eine Liste der Pseudoregister zugeordnet werden, die beim Eintritt in den entsprechenden Block aktiv sind. Außerdem kann man hier auch gleich bei jeder Instruktion jene Operanden kennzeichnen, deren aktive Bereiche bei dieser Instruktion enden[2]. Außerdem müssen noch Daten über die Ausführungshäufigkeit der einzelnen Instruktionsblöcke berechnet und diese den einzelnen Blöcken zugeordnet werden.

2.2 Registerallokation

Nachdem die beschriebenen Vorarbeiten inklusive der Datenflußanalyse abgeschlossen sind, wird das Programm sequentiell Block für Block[3] und Instruktion für Instruktion gelesen; dabei wird der Interferenzgraph (Konfliktgraph) aufgebaut. Der Interferenzgraph ist ein ungerichteter Graph, dessen Knoten die Pseudoregister sind. Eine Kante im Interferenzgraph bedeutet, daß sich die aktiven Bereiche der durch die Kante verbundenen Pseudoregister überlappen. Ein Beispiel für ein einfaches Programm, die aktiven Bereiche der Variablen und den daraus generierten Interferenzgraphen finden Sie in Abbildung 2.1.

Die einzelnen Knoten des Interferenzgraphen lassen sich in freie („unconstrained") und gebundene („constrained") Knoten einteilen. Ein Knoten ist frei, wenn sich für ihn auf jeden Fall ein Maschinenregister (eine Farbe) finden läßt. Das ist dann der Fall, wenn die Anzahl der verfügbaren Maschinenregister größer als die Anzahl der Nachbarn des Pseudoregisters im Interferenzgraphen (d.h. größer als der Grad des Pseudoregisters) ist. Ein Knoten ist gebunden, wenn er nicht frei ist. Wenn für den Graphen in Abbildung 2.1 zwei Farben zur Verfügung stehen, dann ist nur `b` frei, `a`, `c` und `d` sind gebunden.

[2] Die Annotierung der Instruktionsblöcke mit der Liste der aktiven Pseudoregister muß von der Datenflußanalyse gemacht werden, die Annotierung der Instruktionen vereinfacht nur die Arbeit des Registerallokators

[3] Die Reihenfolge der einzelnen Instruktionsblöcke ist dabei egal

```
int a,b,c,d;

...
b=a
c=a+b
d=d
a=c+d
...
```

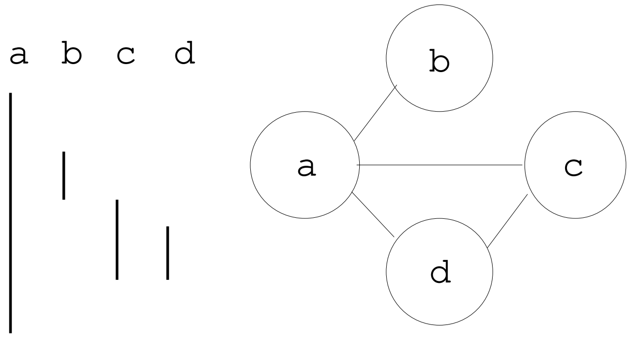


Abbildung 2.1: *Ein kleines Programm und der zugehörige Interferenzgraph*

Die Vorgangsweise beim Färben soll an einem kleinen Beispiel verdeutlicht werden. Dabei nehmen wir an, daß unsere Zielmaschine drei Register hat. In der Abbildung 2.2 ist der Vorgang graphisch dargestellt.

Links oben sieht man ein Programmstück mit fünf Pseudoregistern, `P1` – `P5`. Die aktiven Variablen am Beginn des Programmstücks sind unter `LiveIn` zu finden, die Variablen, die am Ende noch aktiv sind, unter `LiveOut`. Der Interferenzgraph in Form von Interferenzlisten ist rechts davon abgebildet, noch weiter rechts sieht man den Interferenzgraphen graphisch dargestellt.

Der Graph wird jetzt schrittweise vereinfacht; ein freier Knoten nach dem anderen wird entfernt, wobei durch das Entfernen eines Knotens der Grad anderer Knoten kleiner wird, so daß sich im Idealfall der gesamte Graph auf diese Art abbauen läßt.

Nachdem der letzte Knoten entfernt ist, werden die vorher entfernten Knoten in umgekehrter Reihenfolge wieder in den Graphen eingefügt. Dabei werden dann die Maschinenregister vergeben[4]. Am Schluß ist wieder der gesamte Interferenzgraph vorhanden; wenn das Färben erfolgreich war, sind allen Pseudoregistern Maschinenregister zugeordnet.

Diese Beschreibung gilt im wesentlichen für alle Färbealgorithmen; nur das „Prioritätsgesteuerte Färben“ nach Chow/Hennessy führt hier einige wesentliche Änderungen ein, die im entsprechenden Abschnitt beschrieben sind.

[4] Das Entfernen und Einfügen wird normalerweise nicht ganz so gemacht, wie es hier beschrieben ist: beim Abbauen des Graphen müssen im Normalfall Variablen angepaßt werden, die die Anzahl der Nachbarn eines Knoten angeben, während beim Einfügen und Färben hauptsächlich Buch über die bereits vergebenen Register geführt werden muß

2.3 Zusammenlegen von Pseudoregistern

Durch die dem Registerallokator vorangehenden Phasen des Codegenerators werden an vielen Stellen Kopierbefehle eingefügt, die unter Umständen nicht notwendig sind. Diese Kopierbefehle entstehen zum Beispiel:

- beim Verschieben von Pseudoregistern am Unterprogrammanfang sowie vor und nach dem Aufruf eines Unterprogramms, um die korrekte Registerzuordnung für den Unterprogrammaufruf herzustellen.
- Beim Auflösen der ϕ-Funktionen, die von der SSA-Analyse erzeugt wurden.

Da manche dieser Kopierinstruktionen erst unmittelbar vor dem Aufruf des Registerallokators in den Zwischencode eingefügt werden, ist hier der richtige Platz, sie wieder zu entfernen [Bee93].

Dabei kann man sich den Umstand zunutze machen, daß Ziel und Quelle einer Kopieroperation dann im gleichen Register gehalten werden können, wenn sie nicht miteinander interferieren. Das läßt sich mit Hilfe des Interferenzgraphen leicht feststellen. Dafür müssen nur alle Kopierinstruktionen im Programm nach diesen Kriterium untersucht werden. Interferieren Ziel und Quelle nicht, dann können diese beiden Pseudoregister zusammengelegt („coalesced“) werden und der nunmehr unnötige Kopierbefehl kann entfernt werden. Da durch das Zusammenlegen von Pseudoregistern eventuell neue Gelegenheiten zum Zusammenlegen geschaffen werden, wird der Vorgang „Interferenzgraph aufbauen – Zusammenlegen“ solange wiederholt, bis sich keine Pseudoregister mehr zusammenlegen lassen.

Das folgende Beispielprogramm soll den Vorgang illustrieren:

```
                  s t l
mv s,l
mv t,s            |
mv l,s            | |
add x,l,t           | |
```

Anfänglich interferieren also die Pseudoregister `t` und `l` miteinander, und zwar aufgrund der Regel, die besagt:

> *Zwei Pseudoregister interferieren dann, wenn eines am mindestens einem Definitionspunkt des anderen aktiv ist.*

Beachten Sie, daß `t` und `s` durch eine spezielle Behandlung des Registerkopierbefehls beim Aufbau des Interferenzgraphen nicht miteinander interferieren. Die folgende zweite Regel sorgt dafür:

> *Ziel und Quelle eines Registerkopierbefehls interferieren an der Stelle des Kopierbefehls auch dann nicht miteinander, wenn der Quelloperand dort aktiv ist.*

Dadurch ist es möglich, die Pseudoregister `t` und `s` zum Pseudoregister `ts` zusammenzulegen:

```
                 ts  l
mv ts,l          |
*                |
mv l,ts          |
add x,l,ts       |   |
```

Auch die Interferenzen werden zusammengelegt, so daß also jetzt das neue Pseudoregister `ts` mit `l` interferiert. Dadurch wird die Elimination des zweiten Registerkopierbefehls verhindert. Nachdem der Interferenzgraph in nächsten Durchlauf neu aufgebaut worden ist, interferieren `ts` und `l` nicht mehr, sie können also zu einem neuen Pseudoregister `lts` zusammengelegt werden:

```
                 lts
mv lts,l         |
*                |
*                |
add x,lts,lts    |
```

2.4 Auslagern

Nicht jeder Interferenzgraph läßt sich färben; wenn für das Beispiel aus Abbildung 2.2 nur zwei Register zur Verfügung stehen, kann das Färben auf die oben beschriebene Art nach dem Entfernen des zweiten Pseudoregisters aus dem Graphen nicht fortfahren, weil alle verbleibenden Pseudoregister mindestens zwei Nachbarn haben; für diesen Graphen wären also drei Maschinenregister notwendig.

Daher kann einem der Pseudoregister kein Maschinenregister zugeordnet werden; es muß ausgelagert („spilled") werden. Es wird also nach jeder Definition in einer Speicherzelle abgelegt und vor jeder Verwendung aus der Speicherzelle in ein Register geladen, wobei es aber jeweils nur kurz aktiv ist.

Abbildung 2.3 zeigt, was passiert, wenn Pseudoregister `P3` ausgelagert wird. `P3`, das hier durch eine Multiplikation entsteht, ist über das Ende des Programmstücks hinaus aktiv; wenn es ausgelagert wird, muß es nach der Definition abgespeichert werden; dadurch endet der Aktivierungsbereich sofort nach der Definition, und der resultierende Interferenzgraph sieht entsprechend einfacher aus. Dieser Graph kann jetzt erfolgreich mit zwei Registern gefärbt werden.

Die Auswahl des Pseudoregisters, das ausgelagert werden soll, ist kritisch für den weiteren Verlauf der Registerallokation. Daher werden für diesen Fall alle Pseudoregister mit einer Kostenfunktion bewertet. Der Wert dieser Kostenfunktion sollte den relativen zusätzlichen Aufwand, der entsteht, wenn das Pseudoregister nicht in einem Maschinenregister gehalten werden kann, möglichst genau wiedergeben. In der Praxis kann dieser Wert nur eine Annäherung sein, weil zum Beispiel Effekte, die durch die Instruktionspipelines entstehen, an dieser Stelle des Allokators nur schwer zu bewerten sind [HP90, S. 121].

Zum Auslagern wird jenes Pseudoregister ausgewählt, das die geringsten Auslagerungskosten hat. Dabei werden üblicherweise nur jene Pseudoregister berücksichtigt, die noch nicht gefärbt wurden — wenn also während des Färbens die Notwendigkeit entsteht, ein Pseudoregister auszulagern, wird ein Auslagerungskandidat nur unter den noch nicht gefärbten gesucht. Johnson und Miller erwähnen allerdings in [JM86, S. 103], daß in ihrem Compiler das Einbeziehen der bereits gefärbten Pseudoregister bessere Ergebnisse gebracht hat.

Übrigens existieren zwei verschiedene Sichtweisen des Problems: die eine Sichtweise — jene von Chaitin — geht davon aus, daß alle Pseudoregister prinzipiell Maschinenregistern zugewiesen werden; in diesem Fall wird der Wert dieser Funktion, wie gerade beschrieben, als Kosten, die beim Auslagern entstehen, interpretiert. Die zweite Sichtweise — jene von Chow — geht davon aus, daß die Pseudoregister zunächst nicht in Registern gehalten werden und interpretiert den Wert dieser Funktion dann konsequenterweise als den Nutzen, der entsteht, wenn das Pseudoregister in einem Maschinenregister gehalten werden kann [CH90, S. 516][5].

2.5 Termination

Der vorgestellte Algorithmus wird solange wiederholt, bis es gelingt, allen vorhandenen Pseudoregistern Maschinenregister zuzuordnen. Wenn nicht alle Pseudoregister gefärbt werden können, werden einige ausgelagert, was im Normalfall, wie im obigen Beispiel, den Grad einiger Pseudoregister soweit erniedrigt, daß das Färben im nächsten Durchgang erfolgreicher ist.

Ohne diesen Umstand exakt zu beweisen, möchte ich dennoch auf die Beobachtung hinweisen, daß im Extremfall jedes Pseudoregister speicherresident gemacht wird. Vor jedem „verarbeitenden“ Maschinenbefehl, der im allgemeinen maximal zwei Operanden und ein Zielregister hat, stehen dann exakt zwei Ladebefehle. Diese zwei geladenen Operanden sind zum Zeitpunkt der Ausführung des Befehls

[5] Chow und Hennessy führen an, daß sie auch dann ein ausführbares Programm erhalten, wenn die Registerallokationsphase weggelassen wird, was bei den von Chaitin abstammenden Verfahren definitiv nicht möglich ist

die einzigen Werte[6], die in Registern vorhanden sein müssen; mithin genügen in diesem Fall zwei Register, um die Operation durchzuführen, da im allgemeinen als Zielregister bereits wieder ein Operandenregister verwendet werden kann.

[6] Abgesehen von eventuell notwendigen Operandenadressen

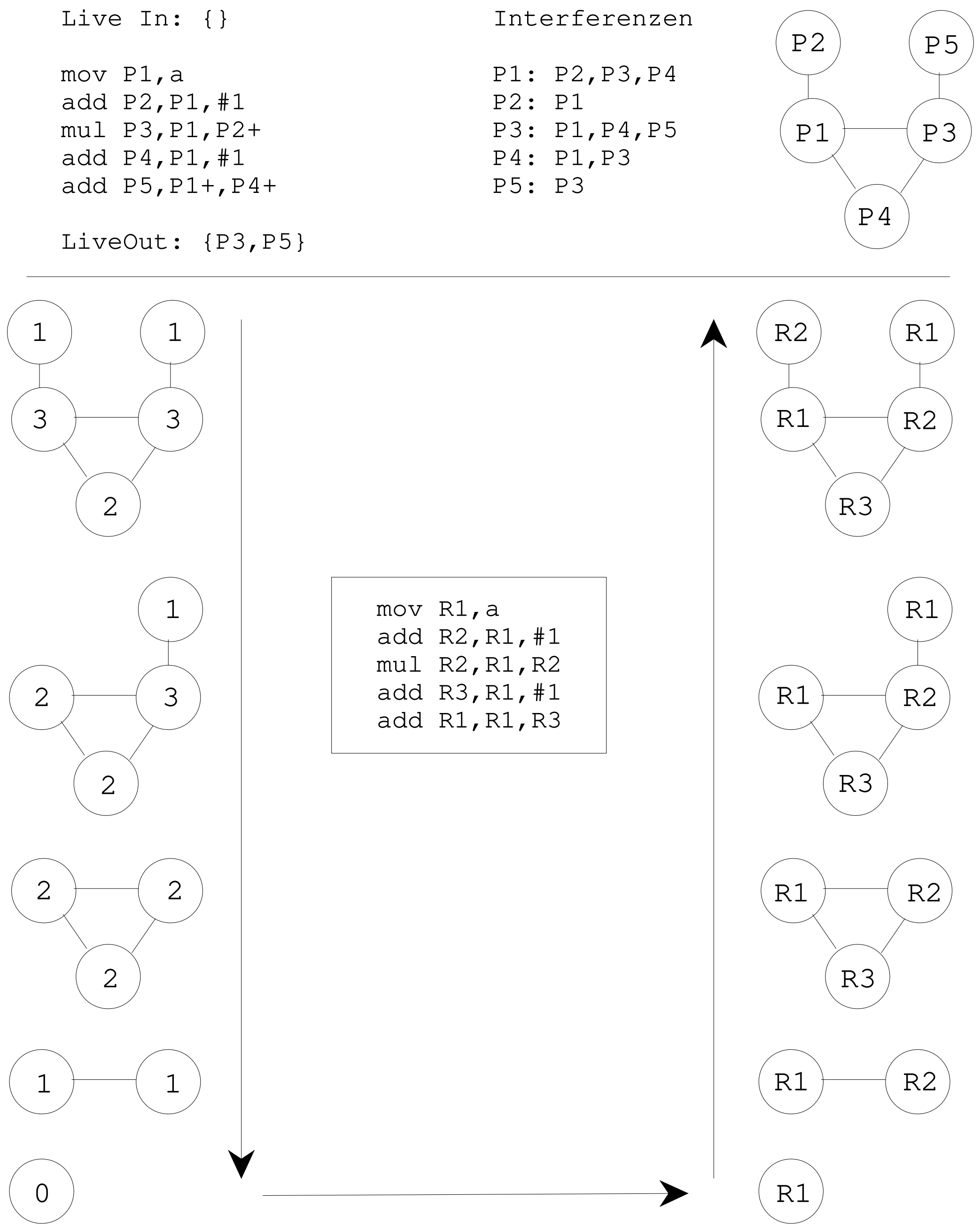
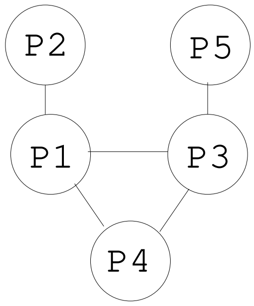


Abbildung 2.2: *Färben eines einfachen Interferenzgraphen*

```
Live In: {}

mov P1,a
add P2,P1,#1
mul P3,P1,P2+
add P4,P1,#1
add P5,P1+,P4+

LiveOut: {P3,P5}
```

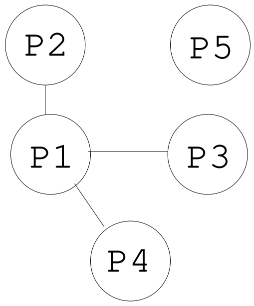
```
Interferenzen

P1: P2,P3,P4
P2: P1
P3: P1,P4,P5
P4: P1,P3
P5: P3
```



Spill P3:

```
Live In: {}

mov P1,a
add P2,P1,#1
mul P3,P1,P2+
mov b,P3+
add P4,P1,#1
add P5,P1+,P4+

LiveOut: {P5}
```

```
Interferenzen

P1: P2,P3,P4
P2: P1
P3: P1

P4: P1
P5:
```




```
mov R1,a
add R2,R1,#1
mul R2,R1,R2+
mov b,R2+
add R2,R1,#1
add R1,R1+,R2+
```

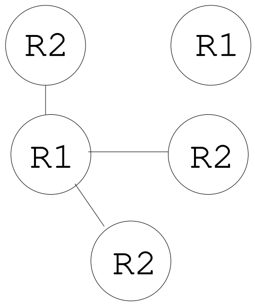


Abbildung 2.3: *Auslagern*

Kapitel 3

Die Implementierung

Während der letzte Abschnitt eine Einführung in die Grundlagen des Graphenfärbens geboten hat, geht es hier um die Implementierung. Es wird ein Modell für den Registerallokator vorgestellt, in das die einzelnen Bestandteile der vorgestellten Verfahren eingepaßt werden und das die Kombination von verschiedenen Methoden ermöglicht.

3.1 Die Hauptschleife

Auf der obersten Ebene besteht der Registerallokator aus einer Vorbereitungsphase und einer Schleife, in der zuerst die Aktivierungsbereiche der Variablen ermittelt werde, bevor der eigentliche Registerallokator aufgerufen wird. Diese Schleife wird solange durchlaufen, bis der Registerallokator erfolgreich war. Beim prioritätsgesteuerten Färben wird die Schleife immer nur einmal durchlaufen:

`prog` Registerallokator
  bereite die Registerallokation vor
  `do`
    führe die Aktivitätsanalyse durch
  `until` (Allokatordurchgang() $\equiv$ `true`)
`endprog`

3.1.1 Die Vorbereitungsphase

Bevor das Programm vom Registerallokator übernommen wird, hat es bereits einige Transformationen durchlaufen[1]. Eine genauere Beschreibung dieser Transformationen finden Sie in [Bee93].

[1] Diese Transformationen werden nicht direkt vom Registerallokator durchgeführt, weil deren Ergebnisse auch schon von anderen Optimierungsphasen benötigt werden

3.1.1.1 Programmdarstellung mit Pseudoregistern

Die wichtigste dieser Transformationen hat als Ergebnis die Darstellung des Programms mit Hilfe von Pseudoregistern. Dabei wird jedem Aktivierungsbereich einer Programmvariablen ein eigenes Pseudoregister zugeordnet. Das bringt bei der Registerallokation folgende Vorteile:

- Jedem Aktivierungsbereich kann ein anderes Maschinenregister zugeteilt werden. Dadurch hat der Registerallokator weniger Einschränkungen zu berücksichtigen.

- Wenn ausgelagert werden muß, kann der Registerallokator hier selektiver vorgehen: anstatt einer ganzen Programmvariablen müssen oft nur einzelne Aktivierungsbereiche ausgelagert werden.

Die Trennung der Aktivierungsbereiche wird unter Zuhilfename der SSA-Form ("Static-Single-Assignment-Form", [CFR+91]) bewerkstelligt. Dabei werden alle Programmvariablen als eine Menge von Werten („values") dargestellt, wobei jeder Wert von genau einer Zuweisung erzeugt wird. Die Werte entsprechen dann den Pseudoregistern. Wenn durch den Kontrollfluß mehrere Werte einer Variablen an einer Stelle zusammentreffen, wird eine Pseudozuweisung generiert, die wieder einen neuen Wert erzeugt. Diese Pseudozuweisungen werden ϕ-Knoten genannt.

Vor der Registerallokation müssen die ϕ-Knoten allerdings wieder in maschinennähere Instruktionen umgewandelt werden, weil ihre Semantik zwar klar definiert ist, von realen Prozessoren aber nicht unterstützt wird. Sie werden daher in `COPY`-Instruktionen umgewandelt, welche die einzelnen Werte aufeinander kopieren. Die meisten dieser `COPY`-Instruktionen werden später vom Registerallokator in der Phase „Zusammenlegen" wieder entfernt.

Abbildung 3.1 zeigt ein Programm vor, während und nach der Transformation in die SSA-Form. Wie Sie in der Abbildung erkennen können, liegt das Programm nach der Umwandlung der ϕ-Knoten nicht mehr in strikter SSA-Form vor, da es zu einem Pseudoregister dann unter Umständen wieder mehrere Zuweisungen geben kann.

3.1.1.2 Sonstige Vorbereitungsschritte

Zusätzlich muß noch die Ausführungshäufigkeit jedes Instruktionsblocks abgeschätzt werden, damit bei der späteren Berechnung von Kostenfunktionen häufiger verwendete Pseudoregister identifiziert werden können. Diese Abschätzung wird zum Beispiel durch Intervallanlyse [Sha80] oder durch Profiling gewonnen.

```
    s = 0
    i = 0
    .
    .
    goto l2
l1:
    s = s + i
    i = i + 1
    .
    .
l2:
    .
    .
    if (i < 5) goto l1
```

```
    s0 = 0
    i0 = 1
    .
    .
    goto l2
l1:
    s2 = s1 + i1
    i2 = i1 + 1
    .
    .
l2:
    i1 = φ(i0, i2)
    s1 = φ(s0, s2)
    if (i1 < 5) goto l1
```

```
    s0 = 0
    i0 = 1
    s1 = s0
    i1 = i0
    goto l2
l1:
    s2 = s1 + i1
    i2 = i1 + 1
    s1 = s2
    i1 = i2
l2:
    .
    .
    if (i1 < 5) goto l1
```

Abbildung 3.1: *Vorbereitung des Programms für den Registerallokator. Links das Quellprogramm, in der Mitte unmittelbar nach der Transformation in die SSA-Form und rechts nach dem Entfernen der ϕ-Knoten.*

3.1.2 Aktivitätsanalyse

Die Aktivitätsanalyse wird vor jedem Durchlauf des Registerallokators aufgerufen. Sie muß folgende Informationen zur Verfügung stellen:

1. für jeden Instruktionsblock die Menge der Pseudoregister, welche beim Eintritt in den Block aktiv sind.

2. für jeden Operanden jeder Instruktion eine Statusvariable, welche angibt, ob der Aktivierungsbereich des verwendeten Pseudoregisters an dieser Stelle

endet.

Algorithmen, die zur Aktivitätsanalyse verwendet werden können finden Sie in [ASU86], [FL91] und in [Bee93].

3.2 Der Allokatordurchgang

Ein Durchlauf des Registerallokators geschieht nach folgendem Schema:

```
prog Allokatordurchgang
   do
      baue Konfliktmatrix auf
      baue Konfliktlisten auf
   until ( Zusammenlegen ( ) ≡ false )
   berechne Auslagerungskosten
   scheide vorgefärbte Pseudoregister aus
   erfolg ← Strategie ( )
   return ( erfolg )
endprog
```

3.2.1 Konfliktmatrix aufbauen

In der ersten Phase muß zunächst das gesamte Programm Instruktion für Instruktion gelesen werden. Dabei werden die Instruktionsblöcke in einer grundsätzlich beliebigen Reihenfolge durchlaufen; in jedem Block werden alle Instruktionen von vorne nach hinten gelesen[2]. Alle für den Registerallokator notwendigen Informationen werden in dieser Phase aus dem Zwischencode „extrahiert". Diese Informationen sind:

- Informationen zur Aktivität der Pseudoregister. An jeder Stelle im Programm muß bekannt sein, welche Pseudoregister aktiv sind. Die aktiven Pseudoregister werden, beginnend von der Menge der zu Beginn des Blocks aktiven Pseudoregister, in einer Liste beziehungsweise in einem Bitfeld mitgeführt. Bei jeder Instruktion werden die Pseudoregister entfernt, deren Aktivierungsbereich endet und jene hinzugefügt, die von der Instruktion definiert werden.

- Für das prioritätsgesteuerte Färben ist es zusätzlich notwendig, für jedes Pseudoregister, das in einem Instruktionsblock aktiv ist, folgende Informationen mitzuführen:

[2] Bei vielen anderen Implementierungen ist die Reihenfolge umgekehrt, weil gleichzeitig die Aktivierungsinformation berechnet wird. Da diese Information bereits vor dem Aufruf des Registerallokators zur Verfügung gestellt wird, können wir von vorne nach hinten lesen

  - Einen Verweis auf den Instruktionsblock.
  - Einen Verweis auf das Pseudoregister.
  - Die Anzahl der Definitionen und Verwendungen im betreffenden Instruktionsblock.
  - Eine Statusvariable, die angibt, ob das erste Auftreten eine Verwendung oder eine Definition ist.

  Diese Informationen werden in einem eigenen Datensatz gesammelt; diese Datensätze werden in je zwei Listen eingefügt: die eine Liste ist über den Instruktionsblock erreichbar und enthält alle in diesem Block aktiven Pseudoregister und die andere ist über jedes Pseudoregister zu erreichen und enthält alle Blöcke, in denen das entsprechende Pseudoregister aktiv ist.

- Für das prioritätsgesteuerte Färben wird außerdem für jedes Pseudoregister ein Bitfeld angelegt, in dem jedes Bit für einen Instruktionsblock steht. Die Bits aller Blöcke, in denen das Pseudoregister aktiv ist, werden gesetzt. Damit ist es später möglich, schnell die Menge der Instruktionsblöcke festzustellen, in denen jedes von zwei Pseudoregistern aktiv ist.

- Informationen zur Kostenabschätzung. Für jedes Pseudoregister wird mitgezählt, wie oft es definiert und verwendet wird und über wieviele Unterprogrammaufrufe hinweg es aktiv ist. Außerdem wird noch der aktuelle Wert der „Fläche“ mitgeführt (siehe dazu Gleichung 4.2 auf Seite 54).

- Datentypen der Pseudoregister. Der Datentyp eines Pseudoregisters bestimmt, welche Maschinenregister zu seiner Aufnahme in Frage kommen. Für jedes Pseudoregister wird hier also die Menge der zulässigen Maschinenregister initialisiert, die unter Umständen beim Auftauchen von Instruktionen, die diese Menge weiter einschränken, noch verändert werden kann. Die Menge der insgesamt für ein Pseudoregister zulässigen Maschinenregister ist also die Schnittmenge dieser Menge über alle Instruktionen, in denen das entsprechende Pseudoregister vorkommt. Wenn r_p die Menge aller zulässigen Maschinenregister für das Pseudoregister p und I_p die Menge aller Instruktionen ist, in denen p vorkommt, dann gilt (auch als Invariante während des Durchlaufens der Instruktionen):

$$r_p = \bigcap_{i \in I_p} r_i$$

- Zusätzliche statistische Information, wie die Anzahl der durchlaufenen Instruktionsblöcke und Instruktionen[3].

[3] Diese Werte stehen zwar im Prinzip bereits zur Verfügung, allerdings kann sich unter Umständen die Anzahl der vom Registerallokator durchlaufenen Blöcke und Instruktionen von der tatsächlichen Anzahl unterscheiden

Die wichtigste Aufgabe dieser Phase bleibt allerdings, wie der Name schon sagt, der Aufbau der Konfliktmatrix. Zwei Pseudoregister werden als „adjazent“ (oder: benachbart, oder: miteinander interferierend) angesehen, wenn sie nicht im gleichen Maschinenregister gehalten werden können. Zur Berechnung der Adjazenz verwenden wir die bereits von Chaitin verwendete Methode [CAC+81, S. 96]: zwei Pseudoregister sind dann einander benachbart, wenn das eine an mindestens einem Definitionspunkt des anderen aktiv ist. In der Praxis kann das leicht an jedem Definitionspunkt mit Hilfe der Menge der aktiven Pseudoregister festgestellt werden.

Dabei muß allerdings beachtet werden, daß Ziel und Quelle einer `COPY`-Operation, bei der ein Pseudoregister auf ein anderes kopiert wird, an diesem Punkt *nicht* miteinander interferieren. Nur dann ist es möglich, Ziel- und Quelle zu einem einzigen Pseudoregister zusammenzulegen (siehe Seite 14).

3.2.2 Zusammenlegen

Die wesentlichen Vorbereitungen für das Zusammenlegen von Pseudoregistern und das Entfernen von unnötigen `COPY`-Instruktionen werden bereits während des Aufbaus der Konfliktmatrix erledigt. Das Zusammenlegen geschieht, indem alle Instruktionsblöcke und alle Instruktionen im Programm durchlaufen werden. Dabei können die Instruktionsblöcke im Prinzip wieder in einer beliebigen Reihenfolge durchlaufen werden. Allerdings weist Briggs in [Bri92, S. 104] darauf hin, daß es besser ist, zunächst die `COPY`-Operationen in den häufiger ausgeführten Blöcken zu entfernen, da das Entfernen eines `COPY`-Befehls eventuell das Entfernen von anderen verhindern kann. Daher sollte eventuell eine Vorsortierung nach der Ausführungshäufigkeit erfolgen.

```
prog Zusammenlegen
    for ( b ∈ instruktionsbloecke )
        for ( i ∈ instruktionen )
            if ( i ≡ Kopierinstruktion )
                if ( Ziel und Quelle von i interferieren nicht )
                    entferne Kopierbefehl
                    lege Quell- und Zieloperand zusammen
                    passe den Konfliktgraph an
                endif
            endif
        endfor
    endfor
endprog
```

Jede `COPY`-Operation, für die eine der folgenden Bedingungen zutrifft, kann entfernt werden:

1. Quelle und Ziel sind identisch.

2. Quelle und Ziel sind nicht identisch, sie interferieren aber auch nicht miteinander und sind nicht mit verschiedenen Registern vorbelegt worden.

Beim Entfernen werden Quell- und Zieloperand zu einem einzigen Pseudoregister zusammengelegt.

Nach dem Zusammenlegen müssen noch alle Vorkommnisse des entfernten Pseudoregisters umbenannt sowie die bei jedem Block gespeicherten Mengen der beim Eintritt aktiven Variablen auf Stand gebracht werden.

Für einige Sonderfälle ist es günstig, wenn man das Zusammenlegen etwas einschränkt und einige Kopierbefehle, die eigentlich entfernt werden könnten, nicht entfernt. Ein Beispiel dafür ist ein Kopierbefehl, der einen registerresidenten Unterprogrammparameter kopiert. Der Quelloperand dieses Kopierbefehls ist mit dem Argumentregister vorbelegt. Für Argumentregister gilt bei den meisten Prozessoren die Konvention, daß sie bei Unterprogrammaufrufen wenn notwendig vom aufrufenden Programm gesichert werden müssen. Wenn dieser Parameter über mehrere Unterprogrammaufrufe hinweg aktiv ist, die „sein" Register aber nicht für ihre Parameter benötigen, ist es günstiger, den Kopierbefehl nicht zu entfernen, weil dann die Möglichkeit besteht, den Zieloperand in ein Register zu legen, das von den aufgerufenen Programmen gesichert werden muß. Damit spart man sich möglicherweise einige Speicher- und Ladebefehle.

3.2.3 Auslagerungskosten berechnen

Wie bereits beschrieben wird die Auswahl von auszulagernden Pseudoregistern über eine Kostenfunktion getroffen. Damit diese Funktion die tatsächlichen Kosten so genau wie möglich wiedergibt, sind folgende Parameter zu beachten:

1. Die Kosten für einen Ladebefehl c_l (Transfer Hauptspeicher – Register)
2. Die Kosten für einen Speicherbefehl c_s (Transfer Register – Hauptspeicher)
3. Die Kosten für einen Registertransfer c_t (Transfer Register – Register)

Im Normalfall — schließlich sind nur relative Werte von Interesse — läßt sich c_t mit 1 annehmen[4] und c_l und c_s als ein Vielfaches von c_t. Mögliche Bewertungen sind:

- $c_t \leftarrow 1$ und $c_l, c_s \leftarrow 2c_t$ [CH90]
- $c_t \leftarrow 1$, $c_s \leftarrow c_t$ und $c_l \leftarrow 1.35c_t$

[4] Die Bewertung der Registertransferbefehle erfolgt nach der Phase „Zusammenlegen".

Eine exakte Abschätzung ist kaum möglich, weil ja dann auch die teilweise nicht vorhersagbaren Effekte von Prozessorpipelines eingerechnet werden müßten.

Die Auslagerungskosten c_p berechnen sich dann folgendermaßen:

$$c_p = c_l \sum_{i \in L} e_i + c_s \sum_{j \in S} e_j - c_t \sum_{k \in T} e_k \tag{3.1}$$

Dabei sind L die Ladebefehle, die erzeugt werden müssen, wenn ein Pseudoregister nicht einem Maschinenregister zugewiesen werden kann, S die entsprechenden Speicherbefehle T die Register-Register-Transferbefehle, in denen das Pseudoregister als Quelloperand vorkommt, und e die Ausführungshäufigkeit der entsprechenden Instruktion.

Die Ausführungshäufigkeit e_i wird mit

$$e_i = 10^{d_i} \tag{3.2}$$

geschätzt, wobei d die bei der Intervallanalyse ermittelte Schachtelungstiefe des Instruktionsblocks ist, der die Instruktion i enthält. Besser ist die Ermittelung der tatsächlichen Ausführungshäufigkeit mit Hilfe eines Profilers. Auch andere Abschätzungen für e_i sind möglich. Vergleichen Sie dazu [JM86, S. 101].

Zur eigentlichen Reihung der Pseudoregister wird dann die Kostenfunktion noch mit dem Grad g_p des Pseudoregisters „normalisiert“, um beim Auslagern Pseudoregister mit niedrigen Auslagerungskosten *und* möglichst vielen Nachbarn gleichzeitig berücksichtigen zu können:

$$h_{0_p} = \frac{c_p}{g_p} \tag{3.3}$$

Obwohl alle von uns implementierten Verfahren im Prinzip die Auslagerungskosten wie hier beschrieben berechnen, gibt es beim prioritätsgesteuerten Färben noch einen verfahrensbedingten Unterschied. Die Kosten müssen nämlich bei den Verfahren nach Chaitin nur einmal (zum Beispiel beim Aufbau der Konfliktvektoren) berechnet werden; beim prioritätsgesteuerten Färben können sich die Kosten durch das Aufteilen von Pseudoregistern noch während des Färbens ändern und müssen daher dynamisch angepaßt werden.

Beachten Sie, daß nicht die Anzahl der Definitionen und Verwendungen als Basis für die Berechnung herangezogen werden sollte, sondern besser die tatsächliche Anzahl der zusätzlichen Lade– und Speicherbefehle[5]. Verschiedene Auslager-Algorithmen werden hier also auch verschiedene Werte ermitteln.

Wenn zum Beispiel mehrere Verwendungen des gleichen Pseudoregisters „nahe beieinander“ stehen, muß für diese Gruppe von Verwendungen nur ein Ladebefehl generiert werden. Hier sind verschiedene Methoden denkbar, um festzustellen,

[5] Im Programmtext ist es günstig, sowohl die Berechnung der Auslagerungskosten als auch das tatsächliche Auslagern vom gleichen Unterprogramm erledigen zu lassen, welches parametergesteuert einmal die eine und dann die andere Funktion erfüllt.

was eine solche Gruppe von „nahe beeinanderstehenden“ Verwendungen ist. Ein gängiges Verfahren, das z.B. von Chaitin [Cha82] oder Briggs [BCKT89] beschrieben wird und das auch wir verwenden, definiert als „Nachbarschaft“ eine Serie von direkt aufeinanderfolgenden Instruktionen in denen kein Aktivierungsbereich endet. Ein Beispiel dazu finden Sie im Abschnitt 3.4 auf Seite 40.

3.2.4 Konfliktvektoren aufbauen

Die Konfliktinformation wird im Registerallokator durch zwei verschiedene Datenstrukturen repräsentiert: einerseits durch eine Bitmatrix, in der jedes Bit eine Relation darstellt und andererseits durch Listen oder Felder, die für jedes Pseudoregister angeben, welche anderen Pseudoregister seine Nachbarn im Graph sind.

Mit Hilfe der Bitmatrix kann das Eintragen einer neuen Relation und das Beantworten der Frage, ob zwei gegebene Pseudoregister zueinander adjazent sind, rasch und effizient erledigt werden. Diese Art von Zugriffen wird in den ersten Phasen gebraucht, während beim eigentlichen Färben vor allem ein schneller Zugriff auf alle benachbarten Pseudoregister notwendig ist. Deswegen wird in dieser Phase die Bitmatrix-Darstellung in die Konfliktvektoren-Darstellung umgewandelt. Danach wird die Konfliktmatrix nicht mehr benötigt, der von ihr belegte Speicher kann also freigegeben werden.

3.2.5 Vorbelegte Pseudoregister ausscheiden

In manchen Fällen ist es wünschenswert oder notwendig, einem Pseudoregister ein bestimmtes Maschinenregister fest zuzuweisen. Diese Zuweisung nennen wir „vorbelegen“. Der am häufigsten auftretende Fall, wo das Vorbelegen notwendig ist, ist die Verwendung eines Pseudoregisters als Unterprogramm-Parameter.

Das eigentliche Vorbelegen geschieht bereits vor dem Aufruf des Registerallokators während der ersten Transformationen im Optimierer, weil zu diesem Zeitpunkt die Informationen über die übergebenen und zu übergebenden Unterprogrammparameter direkt vorhanden sind[6].

Die Pseudoregister, welche mit Maschinenregistern vorbelegt werden, müssen beim eigentlichen Färben nicht mehr berücksichtigt werden. Daher können sie jetzt aus den Konfliktvektoren entfernt werden — zu einem früheren Zeitpunkt ist das nicht möglich, weil sie beim Zusammenlegen mit berücksichtigt werden müssen.

[6] Diese Entscheidung ist pragmatisch und wurde hauptsächlich von den bereits vorhandenen Möglichkeiten im `lcc` beeinflußt.

3.3 Färben

Die ersten Phasen, die der Vorbereitung des eigentlichen Färbens dienen, können von den verschiedenen Methoden gemeinsam verwendet werden. Die Unterschiede zwischen den Methoden treten erst in dieser Phase zutage. Daher gibt es hier eine Anzahl von Funktionen und Unterfunktionen, die großteils frei miteinander kombiniert werden können; damit lassen sich die verschiedensten Registerallokatoren durch die möglichen Kombinationen der einzelnen Funktionen bilden.

3.3.1 Die Allokationsstrategie

Auf der obersten Ebene entscheidet eine aufgerufene Funktion über die *Strategie*, die vom Registerallokator verfolgt wird. Hier sind momemtan drei mögliche Strategien implementiert (siehe Abbildung 3.2):

Einfaches Färben. Diese Strategie entspricht im großen und ganzen der, die von Chaitin und Briggs in ihren Allokatoren verwendet wird [CAC+81] [BCKT89]:

```
prog StrategieSimpel
    h ← Standardheuristik
    PsrListe ( h )
    erfolg ← Faerbe ( )
    if ( erfolg ≡ false )
       Auslagern ( )
    endif
    return ( erfolg )
endprog
```

Färben mit verschiedenen Heuristikfunktionen. Die Auswahl der Pseudoregister für das Färben und Auslagern wird mehrmals mit verschiedenen Heuristikfunktionen durchgeführt. Das von jeder Heuristikfunktion erzielte Allokationsergebnis wird bewertet und das beste Ergebnis genommen. Diese Methode wurde von Bernstein et al. [BGG+89] erfunden und erstmals implementiert:

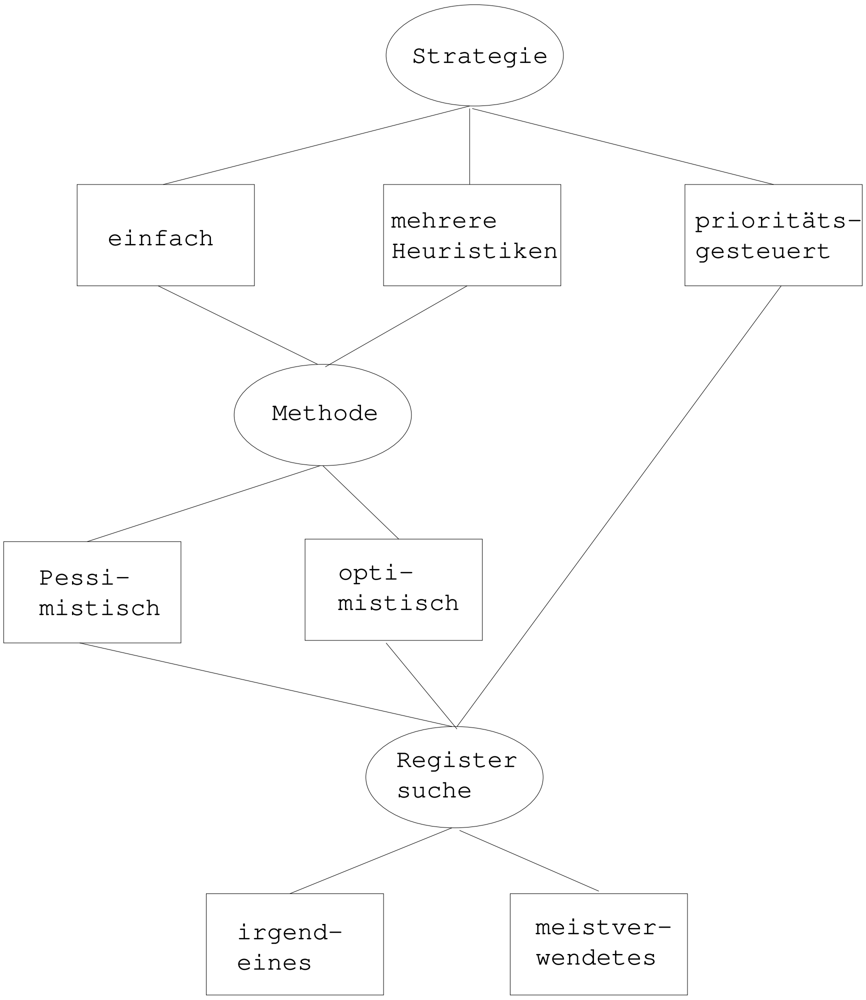


Abbildung 3.2: *Die Allokationsstrategien*

```
prog StrategieMultipel
   minkosten ← ∞
   for ( h ∈ heuristiken )
      psrlisteTmp ← PsrListe ( h )
      if ( Auslagerkosten ( psrlisteTmp ) < minkosten )
         psrlist ← psrlistTmp
         minkosten ← Auslagerkosten ( psrlisteTmp )
      endif
   endfor
   erfolg ← Faerbe ( )
   if ( erfolg ≡ false )
      Auslagern ( )
   endif
   return ( erfolg )
endprog
```

Prioritätsgesteuertes Färben. Diese Strategie unterscheidet sich deutlich von den beiden anderen; sie bildet einen eigenen Block innerhalb der Färbealgorithmen und läßt sich mit den meisten anderen Subfunktionen nicht kombinieren. Entspricht dem „Priority Based Coloring" von Chow und Hennessy [CH90]:

```
prog StrategiePrio
   h ← Standardheuristik
   psrliste ← PsrListe ( h )
   while ( gebundene Pseudoregister ∈ psrliste )
      c ← gebundenes Pseudoregister mit höchster Priorität
      Färbe c , teile c oder lagere c aus
   endwhile
   for ( c ∈ psrliste )
      Faerbe ( c )
   endfor
   erfolg ← true
   return ( erfolg )
endprog
```

Nachdem die Liste der Pseudoregister erstellt ist, werden zunächst nur die gebundenen Pseudoregister, nach Priorität geordnet, gefärbt. Wenn ein gebundenes Pseudoregister nicht sofort gefärbt werden kann, versucht der Allokator den Aktivierungsbereich dieses Pseudoregisters in kürzere Aktivierungsbereiche aufzuteilen, die dann eventuell gefärbt werden können. Erst wenn auch das Aufteilen fehschlägt, muß das Pseudoregister ausgelagert werden.

Dadurch ergeben sich implizit oder explizit noch andere Unterschiede:

1. Bevor der Allokator ein Pseudoregister auslagert, versucht er zunächst, den zugehörigen Aktivierungsbereich in kleinere Bereiche aufzuteilen, wodurch es unter Umständen ermöglicht wird, die neuen, kleineren Aktivierungsbereiche, die in der Regel weniger Konflikte aufweisen, wenigstens zu einem Teil zu färben.

2. Aus dem letzten Punkt ergibt sich in der Praxis ein weiterer bedeutender Unterschied: Jedesmal, wenn ein Aktivierungsbereich aufgeteilt wird, muß auch der Konfliktgraph *exakt* angepaßt werden. Um den Aufwand beim Anpassen vertretbar zu halten, kann Chows Allokator jedes Maschinenregister in jedem Instruktionsblock nur einem Pseudoregister zuteilen — Konflikte sind also nicht auf Instruktionsebene, wie bei den anderen Allokatoren, sondern nur auf Instruktionsblockebene sichtbar. Dadurch entstehen zwar einerseits mehr Konfliktkanten,

andererseits ist es aber sehr einfach möglich, den Konfliktgraphen mitzuführen; vor allem werden die notwendigen Datenstrukturen in der Größe begrenzt[7]. Wesentlich für gute Allokationsergebnisse ist daher auch, daß lange Instruktionsblöcke möglichst vermieden werden; als Grenzfall könnte also jeder Block nur eine einzige Instruktion enthalten, womit die Konflikte wieder auf Instruktionsebene sichtbar sind. Hier muß also über die eingestellte Maximalgröße für Instruktionsblöcke ein Kompromiß zwischen Allokationsaufwand und Allokationsqualität eingegangen werden[8].

3. Wenn es doch notwendig ist, Pseudoregister auszulagern, geschieht dies am Ende des Allokationsvorganges. Der Registerbedarf für die eingefügten Lade- und Speicherbefehle muß bereits vor der Allokation eingerechnet sein, was bedeutet, daß einige Register (im Normalfall 2 – 4, je nach Maschine) von vorneherein reserviert sind und für die „normale“ Allokation nicht zur Verfügung stehen.

4. Ein einmal gefärbtes Pseudoregister behält seine Farbe, unabhängig davon, ob seine Nachbarn ausgelagert werden oder nicht. Dadurch, daß bei den anderen Methoden ein weiterer Durchlauf des Registerallokators erfolgt, wenn das Auslagern von Pseudoregistern notwendig war, könnten diese die vorher getroffenen Entscheidungen „revidieren“ und sich an die durch das Auslagern geänderte Situation besser anpassen.

3.3.2 Liste der Pseudoregister ordnen

Das Färben wird mit dem Aufbau einer geordneten Liste der Pseudoregister begonnen. In jedem Schritt wird ein Pseudoregister ausgewählt, welches dann als erstes in die neue Liste eingefügten wird; je früher ein Pseudoregister also in diese Liste eingefügt wird, desto später wird ihm eine Farbe zugewiesen. Freie Pseudoregister werden nach dem Werte einer „Niveaufunktion“ (siehe unten), gebundene Pseudoregister nach dem Wert einer Heuristikfunktion geordnet:

---

[7] Diese Begrenzung für die Größe der verwendeten Datenstrukturen ist unserer Meinung nach auch der Hauptgrund dafür, warum nach dem Auslagern von Pseudoregistern bei Chaitin ein weiterer Durchlauf des gesamten Allokators notwendig ist. Diesen Durchlauf könnte man einsparen, wenn man, was ja prinzipiell möglich wäre, den Konfliktgraphen und die damit verbundenen Datenstrukturen jedesmal exakt anpaßt.

[8] Der Zeitaufwand steigt laut [LH86] zwar nur um einen Faktor kleiner als 2 an, wenn man kleine Instruktionsblöcke wählt, der Hauptspeicherbedarf allerdings wird dadurch wesentlich größer.

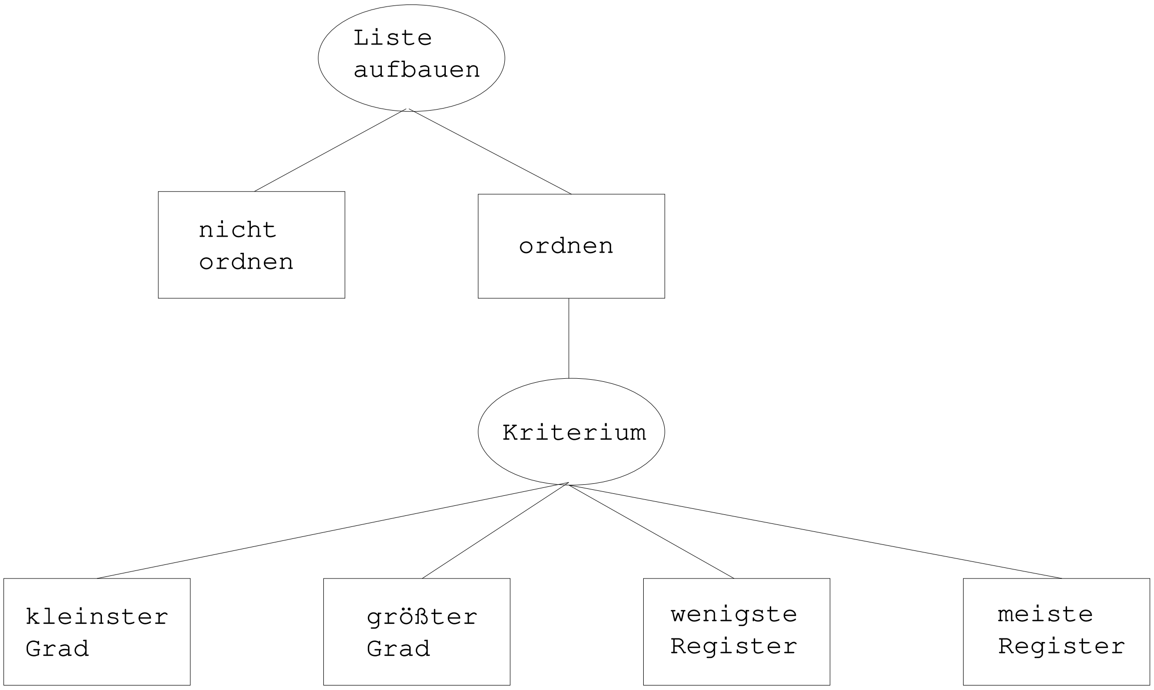


Abbildung 3.3: *Liste der Pseudoregister aufbauen*

```
prog PsrListe( heuristik )
   while ( psrliste ≠ ∅ )
      naechstes ← OrdnePsr ( heuristik, niveau )
      psrliste ← psrliste − naechstes
      faerbeliste ← naechstes + faerbeliste
      passe Konfliktmatrix an
   endwhile
   return ( psrliste )
endprog
```

Aus diesem Grund werden zunächst einmal alle freien Pseudoregister in die Liste eingefügt; da für diese Pseudoregister auf jeden Fall ein Maschinenregister zur Verfügung steht, sollen sie möglichst spät die endgültige Farbe erhalten, um beim Färben der gebundenen Pseudoregister möglichst wenig Einschränkungen behandeln zu müssen.

Die freien Pseudoregister können entweder ohne weitere Sortierung in einer „zufälligen“ Reihenfolge[9] oder auch nach diversen Kriterien sortiert in die Liste

[9] Diese Reihenfolge ist natürlich nicht ganz zufällig — sie ergibt sich vielmehr aus der Reihenfolge des ersten Auftretens jedes Pseudoregisters im Zwischencode und hängt daher unter anderem von der Reihenfolge ab, in der die Instruktionsblöcke gelesen werden

eingefügt werden.

Die Funktion, die das nächste Pseudoregister retourniert, sieht folgendermaßen aus:

prog OrdnePsr(**h**, **n**)
  if (*gebundene Pseudoregister in* **psrliste**)
    p ← *gebundenes Pseudoregister mit kleinster Kostenfunktion* **h**
  else
    p ← *freies Pseudoregister mit kleinstem Niveau* **n**
  endif
  return (**p**)
endprog

Für diese Sortierung wird jedem Pseudoregister ein *Niveau* zugeordnet. Eine konfigurierbare Funktion berechnet dieses Niveau; Das Pseudoregister mit maximalem Niveau wird jeweils als nächstes zum Einordnen in die Liste genommen.

Für die Niveauberechnung stehen mehrere Funktionen zur Verfügung. Bei Briggs werden die freien Pseudoregister beispielsweise so geordnet, daß jene mit dem kleinsten Grad zuerst in die Liste eingefügt werden [BCKT89, S. 278], während bei Bernstein jene mit dem größten Grad zuerst in die Liste eingefügt und damit zuletzt gefärbt werden [BGG$^+$89, S. 260][10].

Hier sind auch andere Bewertungen denkbar, zum Beispiel die Sortierung nicht nach dem Grad, sondern nach der Anzahl der für ein Pseudoregister verfügbaren Maschinenregister. Diese Anzahl hängt davon ab, welchen Registerklassen dieses Pseudoregister zugehört. Die Anzahl der verfügbaren Maschinenregister ist dann die Anzahl der verfügbaren Register in allen Registerklassen, denen das Pseudoregister angehört minus der Anzahl der Nachbarn dieses Pseudoregisters. Auch hier könnten entweder die Pseudoregister mit der kleinsten oder mit der größten Anzahl an verfügbaren Maschinenregister zuerst genommen werden.

Daher sind folgende Funktionen zur Niveauberechnung vorhanden (die Funktion $g(p)$ retourniert den Grad des Pseudoregisters p, die Funktion $f(p)$ die Anzahl der für p verfügbaren Register):

- $p \longleftarrow p_i \mid g(p_i) = \max(g(p_1), \ldots, g(p_n))$ (größte Anzahl Nachbarn)
- $p \longleftarrow p_i \mid d(p_i) = \min(d(p_1), \ldots, d(p_n))$ (kleinste Anzahl Nachbarn)
- $p \longleftarrow p_i \mid d(p_i) = \max(f(p_1), \ldots, f(p_n))$ (meiste freie Register)
- $p \longleftarrow p_i \mid d(p_i) = \min(f(p_1), \ldots, f(p_n))$ (wenigste freie Register)

[10] Nickerson merkt dazu in [Nic90] an, daß bei der Präsentation der Verfahren von Briggs und Bernstein auf der SIGPLAN '89-Konferenz diese beiden Heuristiken diskutiert wurden; die Diskussion ergab, daß Briggs' Heuristik einen schnelleren Compiler ermöglicht und Bernsteins Heuristik die besseren Allokationsergebnisse erwarten läßt.

Wenn keine freien Pseudoregister mehr verfügbar sind, wird ein gebundenes Pseudoregister in die Liste eingefügt. Dabei wird jenes mit den kleinsten Auslagerungskosten (beziehungsweise mit der niedrigsten Priorität) genommen. Die Motivation dahinter ist die, daß beim Färben zunächst die Pseudoregister mit hohen Auslagerungskosten drankommen sollen, weil dann noch eher eine Chance besteht, daß doch ein Maschinenregister frei ist; später, wenn die Register schon knapper sind, macht es dann weniger aus, wenn ein Pseudoregister mit geringen Kosten ausgelagert werden muß.

Jedes gebundene Pseudoregister wird gleichzeitig in eine zweite Liste eingefügt; die „pessimistische“ Allokationsmethode lagert alle diese Pseudoregister sofort aus, ohne zu versuchen, sie zu färben.

Beim prioritätsgesteuerten Färben wird zwar ebenfalls eine Liste der Pseudoregister zusammengestellt; allerdings ist sie dort ausschließlich nach der Prioritätsfunktion geordnet; die Wert der Prioritätsfunktion ist umso größer, je günstiger es ist, das betreffende Pseudoregister in einem Maschinenregister zu halten. Die Liste ist nach der fallenden Prioritätsfunktion geordnet; die gebundenen Pseudoregister mit der höchsten Priorität werden also zuerst gefärbt. Es ist aber zu beachten, daß sich die Prioritäten und die Zuordnung zu den Zuständen „gebunden“ – „frei“ durch das Aufteilen ständig ändert.

3.3.3 Aufteilen

Das prioritätsgesteuerte Färben gibt nicht einfach auf, wenn es einem Pseudoregister kein Maschinenregister zuteilen kann, sondern versucht zunächst, den Aktivierungsbereich in kleinere Aktivierungsbereiche aufzuteilen. Dadurch verringert sich im Normalfall die Anzahl der Interferenzen in den neu generierten, kleineren Aktivierungsbereichen, wodurch ein Färben von einigen dieser Bereiche in den meisten Fällen möglich wird. Das Aufteilen erfolgt wie auch das Färben auf der Ebene von Instruktionsblöcken; ein Aktivierungsbereich, der nur einen Block umfaßt, kann also nicht mehr weiter aufgeteilt werden. Der Algorithmus geht nach dem folgenden Schema vor:

```
prog Aufteilen
    block ← Eintrittspunkt in den Aktivierungsbereich abAlt
    liste ← block
    while ( liste ≠ ∅ and noch Register für abNeu vorhanden )
       block ← erstes Element aus liste
       if ( block ∈ ab )
          abNeu ← abNeu + block
          abAlt ← abAlt − block
          if ( block hat einen Nachfolger )
             liste ← nachfolger(block ) + liste
          else
             for ( i ∈ nachfolger(block ) )
                liste ← liste + i
             endfor
          endif
       endif
    endwhile
    Interferenzen für abAlt und abNeu anpassen
 endprog
```

Zunächst wird also ein Instruktionsblock gesucht, in dem auf jeden Fall noch Register verfügbar sind; ist ein solcher Block nicht zu finden, dann kann das Aufteilen bereits hier wegen Erfolglosigkeit abgebrochen werden. Wenn ein Block mit freien Registern gefunden wird, bildet er den ersten Block eines neuen Aktivierungsbereiches. Es werden in der Folge solange Nachfolgerblöcke des gleichen Aktivierungsbereiches zum neuen Aktivierungsbereich hinzugefügt, bis entweder keine Nachfolgerblöcke mehr existieren oder durch das Hinzufügen eines weiteren Blockes der neue Aktivierungsbereich nicht mehr gefärbt werden kann.

Wenn die Form des neuen Aktivierungsbereiches feststeht, muß am Ende noch der Interferenzgraph angepaßt werden. Für den nun neu generierten Aktivierungsbereich kann nun auf jeden Fall ein Maschinenregister gefunden werden; das Färben kann also an dieser Stelle forgesetzt werden.

Nachdem Aktivierungsbereiche immer gesamte Instruktionsblöcke umfassen müssen, wird das Aufteilen umso erfolgreicher sein, je kürzer die Blöcke gewählt werden. Im Extremfall umfaßt ein Block dann nur mehr eine Instruktion; dann ist der Interferenzgraph gleich exakt wie bei den Optimistischen und Pessimistischen Färbeverfahren.

Eine ausführliche Beschreibung des Aufteilens finden Sie in [LH86].

3.3.4 Maschinenregister zuteilen

Nachdem die geordnete Liste der Pseudoregister erstellt ist, kann die eigentliche Zuteilung von Maschinenregistern beginnen. Die vorher erstellte Liste wird von vorne nach hinten, also in umgekehrter Reihenfolge der Erstellung, durchgearbeitet. Hier gibt es wieder zwei mögliche Methoden:

Pessimistisches Färben Bei dieser Methode, die der ursprünglich von Chaitin implementierten entspricht, wird nur dann überhaupt mit der Maschinenregisterzuteilung begonnen, wenn beim Ordnen keine gebundenen Pseudoregister gefunden und zum Auslagern markiert wurden. Andernfalls werden die markierten Pseudoregister sofort ausgelagert.

```
prog FaerbePessimistisch
   if ( auslagerliste ≠ ∅ )
      lagere alle Pseudoregister in auslagerliste aus
      erfolg ← false
   else
      färbe alle Pseudoregister in psrliste
      erfolg ← true
   endif
   return( erfolg )
endprog
```

Optimistisches Färben Bei dieser Methode wird auf jeden Fall mit dem Färben begonnen, egal, ob beim Ordnen gebundene Pseudoregister gefunden wurden oder nicht. Dadurch können eventuell vorher gebundene Pseudoregister doch noch gefärbt werden, weil zum Beispiel einige Nachbarn vorher schon ausgelagert wurden oder einige das gleiche Maschinenregister bekommen haben. Diese Methode wurde ursprünglich von Briggs implementiert:

```
prog FaerbeOptimistisch
   for ( p ∈ psrliste )
      if ( Maschinenregister für p verfügbar )
         Ordne p das Register zu
         psrliste ← psrliste − p
      endif
   endfor
   if ( psrliste ≠ ∅ )
      lagere alle Pseudoregister in psrliste aus
      erfolg ← false
   else
      erfolg ← true
   endif
   return( erfolg )
endprog
```

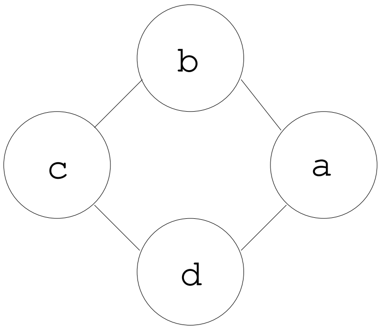


Abbildung 3.4: *Ein Interferenzgraph, der zwei Farben benötigt.*

An einem Beispiel läßt sich leicht erkennen, warum das „Optimistische Färben" in einigen Fällen die besseren Ergebnisse liefert. Betrachten Sie dazu den Interferenzgraphen in Abbildung 3.4. Um diesen Graphen zu färben, benötigt man genau zwei Farben (siehe dazu auch [BCKT89, S. 277]).

Wenn dem Algorithmus von Chaitin nun zwei Farben, zum Beispiel rot und blau, zur Verfügung gestellt werden, findet er zunächst keinen freien Knoten, da in diesem Graph alle Knoten vom Grad 2 sind und ein Knoten dann frei ist, wenn die Anzahl der verfügbaren Farben größer als der Grad des Knotens ist. Er wählt also zufällig einen Knoten, zum Beispiel `a` zum Auslagern aus. Dadurch fällt der Grad von `b` und `d` auf 1, und die restlichen Knoten können als frei in der Reihenfolge `b`, `d` und `c` entfernt werden. Leider hat der Knoten `a` keine Farbe erhalten, obwohl das im Prinzip möglich gewesen wäre.

Dem Algorithmus von Briggs geht es zunächst genauso wie dem von Chaitin. Er entfernt die Knoten dann in der Reihenfolge `a`, `b`, `d` und `c`. Wenn es allerdings ans Färben geht, findet er `a` als ersten in der Liste der zu färbenden Knoten. Da noch keine Farbe an einen Nachbarn vergeben ist, kann `a` die erste freie Farbe erhalten, zum Beispiel rot. Für `b`, den nächsten Knoten in der Liste, steht dann noch blau zur Verfügung, ebenfalls für `d`. Zum Schluß kann dann `c` wieder mit rot gefärbt werden.

3.3.5 Maschinenregister suchen

Jedesmal, wenn einem Pseudoregister ein Maschinenregister zugeteilt werden soll, muß zunächst ein freies Maschinenregister gesucht werden. Dieses Register muß

1. Zum Datentyp des Pseudoregisters passen und …

2. …darf nicht bereits von einem Nachbarn verwendet worden sein.

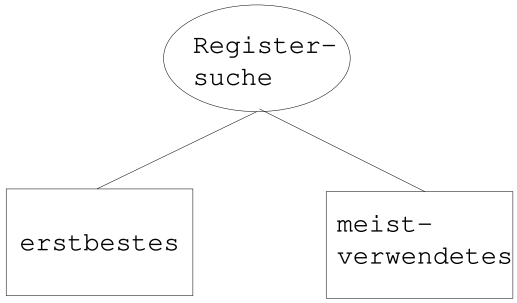


Abbildung 3.5: *Verfahren für die Registersuche*

Um der ersten Bedingung zu genügen, werden die Maschinenregister in Klassen eingeteilt. Jeder Registerklasse ist eine Menge von möglichen Pseudoregister-Datentypen zugeordnet, wobei unter Umständen ein Pseudoregister-Datentyp auch mehreren Registerklassen zugeordnet sein kann[11].

Die Einhaltung der zweiten Bedingung wird dadurch sichergestellt, daß beim Zuordnen eines Maschinenregisters dieses Register für alle Nachbarn im Konfliktgraph gesperrt wird.

Die Registersuche selbst ist ebenfalls konfigurierbar. Es gibt folgende Möglichkeiten (siehe Abbildung 3.5):

Das erstbeste Maschinenregister nehmen Die Maschinenregister werden dabei in einer beliebigen Reihenfolge durchsucht. Das erste freie Register wird genommen.

Das meistbenutzte Register nehmen Durch diese Methode kann der Gesamtregisterbedarf in vielen Fällen etwas reduziert werden. Sie ist allerdings deutlich aufwendiger als die erste Methode[12].

Diese Phase ist beim prioritätsgesteuerten Färben wesentlich komplexer. Die vorher geordneten Pseudoregister werden der Reihe nach gefärbt; wenn ein Pseudoregister sich nicht färben läßt, wird zunächst versucht, den Aktivierungsbereich

[11] Dieser Fall könnte zum Beispiel bei einem Prozessor der Motorola 68K-Familie auftreten, wo es je eine Registerklasse für Adreß- und Datenregister gibt: ein nur temporär gebrauchtes Pseudoregister könnte unter Umständen in beide Registerklassen passen, weil keine speziellen Operationen notwendig sind, die nur in Adreß- oder nur in Datenregistern durchgeführt werden können.

[12] Nachdem aber dieser Teil des Registerallokators nur einen kleinen Teil der Gesamtlaufzeit in Anspruch nimmt, wird diese Aufwandserhöhung in der Praxis kaum spürbar sein. Daher macht sich dieses Verfahren eventuell auch dann bezahlt, wenn dadurch nur kleine Gewinne in der Maschinencodequalität zu erwarten sind.

in kleinere Bereiche aufzuteilen. Unter Umständen läßt sich dann zumindest einer der neuen Aktivierungsbereiche färben. Wenn das Färben trotzdem nicht möglich ist, wird das Pseudoregister unwiderruflich ausgelagert.

3.4 Auslagern

3.4.1 Grundsätzliches

Falls Pseudoregistern kein Maschinenregister zugeteilt werden konnte, müssen diese Pseudoregister ausgelagert werden. Es wird also, üblicherweise auf dem Stack, eine Speicherzelle alloziert, die den Inhalt des Pseudoregisters aufnehmen kann. Dann wird nach jeder Definition des Pseudoregisters dessen Wert in der zugeordneten Speicherzelle gespeichert und vor jeder Verwendung der entsprechende Wert wieder in ein Register geladen. Zwischen den einzelnen Stellen, wo das Pseudoregister definiert oder verwendet wird, ist es nicht aktiv. An den Stellen, wo das Pseudoregister nunmehr inaktiv ist, wird das bisher gehaltene Register frei, wodurch das Färben des Graphen beim nächsten Durchlauf erleichtert wird.

Diese Beschreibung ist insofern noch ungenau, als eigentlich für jeden der kurzen Aktivierungsbereiche, die sich durch das Auslagern eines Pseudoregisters ergeben, jeweils ein neues Pseudoregister angelegt wird. Im folgenden Beispiel wird das deutlich:

```
mov P1,a
add P2,P1,#1
mul P3,P1+,P2
mov P4,b
mul P5,P2+,P4+
add P6,P3+,P5+
```

Wenn das Pseudoregister `P3` ausgelagert werden muß, wird es nach der Definition im Hauptspeicher abgelegt und als neues Pseudoregister `P7` später wieder verwendet:

```
mov P1,a
add P2,P1,#1
mul P3,P1+,P2
mov tmp,P3+
mov P4,b
mul P5,P2+,P4+
mov P7,tmp
add P6,P7+,P5+
```

Dabei ist zu beachten, daß ein Pseudoregister, welches einmal ausgelagert wurde, nicht noch einmal ausgelagert werden sollte. Der Blick auf das Beispiel zeigt auch deutlich, warum: die Aktivierungsbereiche, die vom ausgelagerten `P3` erzeugt wurden, sind jetzt schon so kurz wie möglich; ein nochmaliges Auslagern von `P3` oder `P7` würde also bestenfalls das Programm länger machen. Um dieses nochmalige Auslagern zu verhindern, werden alle beim Auslagern generierten Pseudoregister gekennzeichnet.

Wie schon bei der Beschreibung der Berechnung der Auslagerungskosten erwähnt wurde, wird keineswegs für jede Definition ein Speicher- und für jede Verwendung ein Ladebefehl generiert. Für „benachbarte" Definitionen oder Verwendungen ist dies nicht notwendig. Zwei Definitionen oder Verwendungen werden in unserem Fall als „benachbart" angesehen, wenn zwischen ihnen kein Aktivierungsbereich endet. Diese Definition wird beispielsweise auch von Chaitin [Cha82] oder Briggs [Bri92] verwendet.

Betrachten Sie dazu beispielsweise dieses Programmstück:

```
add P3,P1,P2
add P4,P1,P3
add P5,P4+,P3+
mul P6,P1,P5+
```

Wenn das dreimal verwendete Pseudoregister `P1` ausgelagert werden muß, werden nur zwei Ladebefehle eingefügt, weil die beiden ersten Verwendungen unserer Definition nach benachbart sind:

```
mov P1,tmp
add P3,P1,P2
add P4,P1+,P3
add P5,P4+,P3+
mov P7,tmp
mul P6,P7,P5+
```

Beachten Sie dabei, daß das Einfügen von zusätzlichen Lade- oder Speicherbefehlen in Programmabschnitten, in denen keine Aktivierungsbereiche enden, tatsächlich keinen Sinn machen würde: nachdem alle anderen Pseudoregister in diesem Bereich weiterhin Register belegen und diese nicht frei geben, ändert sich am Registerbedarf auch dann nichts, wenn die Aktivierungsbereiche ausgelagerter Pseudoregister zwischen zwei Instruktionen kurzfristig enden. Trotzdem ist diese Definition von „benachbart" natürlich heuristisch; es können daher durchaus auch andere Definitionen verwendet werden; die hier vorgestellte hat sich in der Praxis bewährt und wird deswegen auch in unserem Compiler angewandt.

3.4.2 Bereinigen

Eine mögliche Verbesserung betrifft die Generierung der Instruktionen, die ausgelagerte Variablen in ein Register holen und wieder in den Speicher zurückschreiben. Von Bernstein et al. [BGG+89] wird eine relativ „aggressive“ Technik vorgeschlagen, bei der zunächst versucht wird, jede ausgelagerte Variable pro Instruktionsblock nur einmal zu laden und wieder abzuspeichern[13].

Diese Technik führt in vielen Fällen zum Erfolg, weil manche Pseudoregister über mehrere Instruktionsblöcke hinweg aktiv sind, aber oft nur in einem davon die Register wirklich knapp sind. Mit Hilfe der vorgeschlagenen Technik erreicht man, daß in allen Instruktionsblöcken, in denen ausreichend Register vorhanden sind, für die entsprechenden ausgelagerten Variablen trotzdem nur maximal eine Lade- und eine Speicherinstruktion erforderlich sind. Nur in den Blöcken, wo tatsächlich zu wenig Register vorhanden sind, ist wie beim konservativen Auslagern für jede Verwendung eine Lade-, und für jede Definition eine Speicherinstruktion notwendig.

Damit das Verfahren für die Instruktionsblöcke, in denen Register knapp sind, nicht blockiert, wird dieses „Bereinigen“ nur für die erste oder die ersten paar Iterationen des Graphenfärbens eingeschaltet. Bei den letzten Iterationen wird dann wieder auf das normale Verfahren umgeschaltet.

[13] Man vergleiche diese Technik mit der Methode von Chow!

Kapitel 4

Datenstrukturen und weitere Algorithmen

In diesem Abschnitt werden Themen behandelt, die entweder vom Umfang oder von der Thematik her schlecht im Kapitel „Implementierung“ unterzubringen waren.

4.1 Der Konfliktgraph

Eine der wesentlichen Objekte in einem Registerallokator, der Graphenfärben verwendet, ist der Konfliktgraph. Der Konfliktgraph ist ein ungerichteter Graph. Jedes Pseudoregister ist ein Knoten in diesem Graph, und je zwei Pseudoregister, die durch eine Kante verbunden sind, dürfen nicht das gleiche Maschinenregister zugewiesen bekommen.

Um den Registerallokator effizient zu implementieren, ist es günstig, den Konfliktgraph in zwei verschiedenen Darstellungen zu speichern, nämlich einerseits in Form einer Matrix, welche in den ersten Phasen gebraucht wird, und andererseits in der Form von Vektoren, die bei der Registerzuteilung günstiger sind.

4.1.1 Darstellung als Matrix

Die Konfliktmatrix ist ein Bitfeld, in dem jedes Bit eine Kante des Konfliktgraphen repräsentiert. Dabei wird aus Effizienzgründen angenommen, daß jedem Pseudoregister eine kleine natürliche Zahl zugewiesen ist[1]. Wenn zwischen zwei Pseudoregistern mit den Nummern n und m eine Kante existiert, dann ist das entsprechende Bit in der Matrix gesetzt.

Die Konfliktmatrix erlaubt es, sehr schnell zu testen, ob zwischen zwei gegebenen Pseudoregistern eine Kante im Konfliktgraphen existiert. Auch das Hin-

[1] Die in unserem Compiler verwendeten Konfliktmatrizen wurden so implementiert, daß es egal ist, ob die Numerierung bei 0 oder bei 1 beginnt.

zufügen von neuen Kanten ist effizient; daher ist diese Darstellung während des Aufbaus des Konfliktgraphen und des Zusammenlegens von Pseudoregistern ideal.

Zwei Eigenschaften des Konfliktgraphen können ausgenützt werden, um den für die Matrix benötigten Speicherplatz zu minimieren:

1. Der Konfliktgraph ist ungerichtet, daher ist die resultierende Matrix symmetrisch, weswegen nur eine Hälfte gespeichert werden muß.

2. Ein Pseudoregister ist trivialerweise Nachbar von sich selbst; daher muß auch die Hauptdiagonale der Matrix nicht gespeichert werden.

Von der vollständigen quadratischen Bitmatrix mit der Seitenläge m benötigen wir also die eine Hälfte exklusive der Hauptdiagonale. Das läßt sich laut Gauß als

$$n = \sum_{i=1}^{m} i - m = \frac{m(m+1)}{2} - m = \frac{m(m-1)}{2} \tag{4.1}$$

schreiben.

4.1.2 Darstellung als Vektoren

Während der Zuteilung von Maschinenregistern ist es vor allem notwendig, alle Nachbarn eines bestimmten Pseudoregisters im schnellen Zugriff zu haben. Dafür ist es günstig, den Konfliktgraphen mit Hilfe von Vektoren darzustellen. Jedem Pseudoregister wird also ein Vektor mitgegeben, der Verweise auf alle in Konfliktgraphen benachbarten Pseudoregister enthält.

Bei allen Verfahren außer beim prioritätsgesteuerten Färben ändert sich ein solcher Konfliktvektor nach dem Aufbau nicht mehr. Beim prioritätsgesteuerten Färben allerdings muß der gesamte Konfliktgraph angepaßt werden, wenn das Aufteilen eines Pseudoregisters notwendig war; dann kann sich auch die Größe eines Konfliktvektors ändern. Dieser Umstand sollte bei der Wahl einer Datenstruktur für die Vektoren beachtet werden.

4.2 Pseudoregister

Das Pseudoregister gehört neben der Konfliktmatrix zu den zentralen Datenstrukturen im Registerallokator. Einige wesentliche Aspekte der von uns gewählten Darstellung sollen hier erwähnt werden.

Für Variablen existiert im `lcc`, den wir als Ausgangsbasis für unseren Compiler verwendet haben, bereits der Datentyp `Symbol`. Dieser Datentyp wurde von uns wesentlich erweitert und wird auch für die Darstellung von Pseudoregistern verwendet.

Informationen, die mit jeden Pseudoregister assoziiert werden, sind:

- Das dem Pseudoregister zugeordnete Maschinenregister.
- Auslagerungskosten, gewichtete Kosten für beim Auslagern notwendige Lade– und Speicherbefehle.
- Eine Liste mit den Nachbarn im Konfliktgraph.
- Bitfelder zur Registersuche, welche Bits für „erlaubte“ und „verbotene“ Register beinhalten. Diese Bitfelder werden während der Registervergabe ständig angepaßt.
- Eine Statusvariable, die anzeigt, ob das entsprechende Pseudoregister ausgelagert werden darf.
- Eine Liste mit „Aktivierungseinheiten“ für das prioritätsgesteuerte Verfahren.
- Ein Bitfeld mit den Instruktionsblöcken, in denen das entsprechende Pseudoregister aktiv ist; dieses wird ebenfalls vom prioritätsgesteuerten Verfahren benötigt.
- Statistische Information: Anzahl der Definitionen und Verwendungen.

4.3 Maschinenregister

4.3.1 Prinzipielles

In einem konfigurierbaren Compiler müssen besondere Anforderungen an die interne Darstellung von Maschinenregistern gestellt werden:

- Einerseits muß der Zugriff auf die Beschreibung einzelner Register einfach und effizient sein, weil diese Informationen häufig gebraucht werden.
- Andererseits muß die Anpassung an eine große Zahl von Zielmaschinen möglichst einfach sein. Daher müssen alle wesentlichen Eigenschaften von Registersätzen zu modellieren sein.

Pro Register existiert daher ein Datensatz, der die wesentlichen Eigenschaften dieses Registers beschreibt. Alle Maschinenregister sind zu einem Feld zusammengefaßt; die Registernummer dient dabei als Index in dieses Feld.

4.3.2 Eigenschaften von Registern

Maschinenregister haben eine Reihe von Eigenschaften, die bei der Allokation beachtet werden müssen. Diese Eigenschaften sind:

Verfügbarkeit : Steht das Maschinenregister dem Allokator zur Verfügung oder handelt es sich dabei um ein fest vergebenes Spezialregister (zum Beispiel um den Instruktionszähler)?

Die prioritätsgesteuerte Registerallokation vergibt die Maschinenregister für ausgelagerte Pseudoregister dann, wenn schon alle anderen Registerzuordnungen endgültig getroffen wurden. Damit zu diesem Zeitpunkt auf jeden Fall noch genug Maschinenregister zur Verfügung stehen — einen zweiten Durchlauf gibt es ja nicht — müssen Register für diesen Zweck reserviert werden. Diese Register sind in unserem Allokator speziell gekennzeichnet, weil nur das prioritätsgesteuerte Färben sie verwendet.

Aufrufkonvention : Wer ist für das Sichern des Registers bei Unterprogrammaufrufen zuständig: das aufrufende Programm („caller saved“) oder das aufrufende Programm („callee saved“)?

Parameterübergabe : Kann das Register zur Parameterübergabe verwendet werden?

Werterückgabe : Können in diesem Register Rückgabewerte einer Funktion stehen?

4.3.3 Spezielle Register

In vielen Fällen existieren für gewisse Operationen spezielle Register; die Prozessoren der MIPS-Familie haben beispielsweise ein eigenes Register, in dem die Ergebnisse von Multiplikations- und Divisionsoperationen abgelegt werden. Bei älteren Architekturen ist auch die Verwendung von „Akkumulatoren“ gängig, welche generell als Ergebnisregister für arithmethische Operationen verwendet werden.

Zur Handhabung von Spezialregistern können mehrere verschiedene Mechanismen herangezogen werden:

1. Spezialregister können einer eigenen Registerklasse zugeordnet werden.

2. Die Notwendigkeit, gewisse Pseudoregister in Spezialregister zu legen, kann erkannt werden und alle anderen Maschinenregister können als „nicht erlaubt“ markiert werden.

3. Spezialregister können ignoriert werden.

Die erste Methode wird normalerweise dann angewandt, wenn ein größerer Satz an Registern mit den gleichen Eigenschaften existiert, etwa Fließkommaregister.

Die zweite Methode ist dann günstig, wenn nur beim Auftreten von gewissen Operationen spezielle Register erforderlich sind. Ein Pseudoregister wird dann zunächst der passenden, allgemeinenen Registerklasse zugeordnet; wenn Operationen auftauchen, die es erforderlich machen, daß das Pseudoregister einem oder einer kleinen Gruppe von speziellen Maschinenregistern zugeordnet wird, können alle anderen Register als "nicht erlaubt" gekennzeichnet werden.

In manchen Fällen ist es hier notwendig, daß auch die Instruktionsselektion darauf Rücksicht nimmt; unter Umständen sollten zum Beispiel Aktivierungsbereiche, die solche Spezialregister benötigen, bereits bei der Instruktionsselektion erkannt und dort möglichst kurz gehalten werden, um ein Blockieren des Registerallokators zu verhindern (diese Situation ist analog zu der, wie sie bei Aufrufkonventionen für Unterprogramme auftritt).

Die dritte Methode kann dann angewandt werden, wenn solche Besonderheiten vom der Maschinencoderzeugung gehandhabt werden können. Ein Beispiel dafür wären die Register `LO` und `HI`, in denen die MIPS-Prozessoren die Ergebnisse von Multiplikations- und Divisonsoperationen ablegen. Hier ist es am besten, dem Registerallokator diese Besonderheit vorzuenthalten: Multiplikation und Division sehen für den Registerallokator dann wie ganz normale Drei-Adress-Operationen aus; zusätzliche Registerverschiebebefehle werden automatisch beim Generieren des Maschinencodes erzeugt.

4.3.4 Registerklassen

Die Registersätze der meisten Prozessoren lassen sich in diverse Klassen verschiedener Funktionalität aufteilen. Beispiele dafür wären etwa die Unterscheidung zwischen Ganzzahl- und Fließkommaregistern oder die Trennung in Adress- und Datenregister.

Alle zu einer Registerklasse gehörenden Register sind in der Registerbeschreibung in unmittelbar aufeinanderfolgenden Feldelementen abgelegt; die Registerklassenbeschreibung unterteilt dieses Maschinenregisterfeld durch die Angabe von Startindex und Anzahl der Register in der Klasse in einzelne Bereiche. Dadurch können sowohl alle vorhandenen Register als auch die Register in einer Klasse bei Bedarf einfach durchlaufen werden.

4.3.5 Pseudoregister und Registerklassen

Die Registerklasse eines Pseudoregisters wird grundsätzlich aus seinem Datentyp bestimmt; die bestimmte Registerklasse wird dann dazu verwendet, das Bitfeld mit den für dieses Pseudoregister erlaubten Maschinenregistern zu initialisieren.

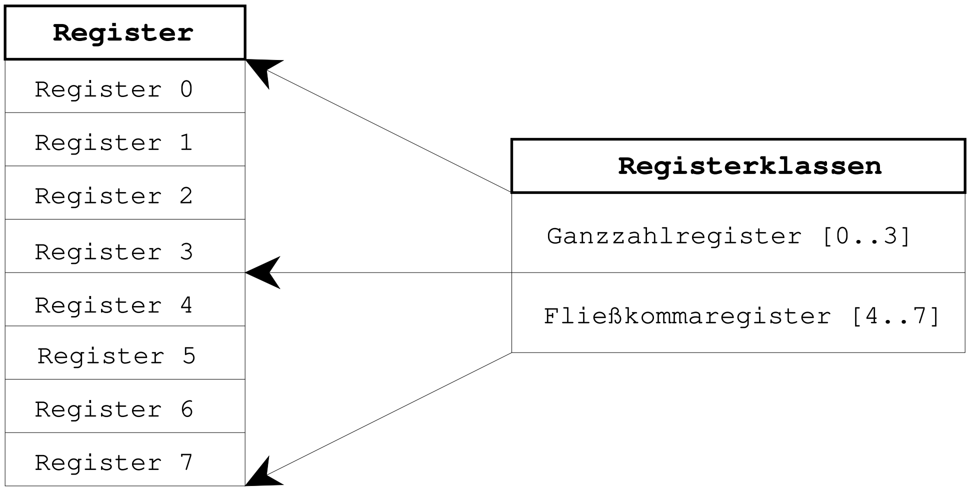


Abbildung 4.1: *Die Datenstrukturen für die Beschreibung der Maschinenregister*

Dabei ist zu beachten, daß ein Pseudoregister nicht immer ganz klar genau einer Registerklasse zuzuordnen ist; unter Umständen kann ein bestimmtes Pseudoregister auch mehreren Registerklassen gleichzeitig angehören, was die Anzahl der verfügbaren Register natürlich erhöht. Aus diesem Grund wird jeder Registerklasse ein Bit zugeordnet. Damit läßt sich auch leicht überprüfen, ob die Registerklassen zweier Pseudoregister sich überschneiden; dieser Test wird sehr häufig benötigt, da zwei Pseudoregister nur dann miteinander interferieren können, wenn die Schnittmenge der Registerklassen nicht leer ist.

4.3.6 Registerpaare

Ein „Registerpaar“ ist eine Paar von zwei (oder mehr) unmittelbar nebeneinanderliegenden Registern; je nach dem verwendeten Prozessor müssen diese Registerpaare gewissen Ausrichtungskonventionen gehorchen; bei den MIPS-Prozessoren muß die kleinere der beiden Registernummern immer geradzahlig sein.

Solche Registerpaare treten in der Praxis sehr häufig auf. So müssen zum Beispiel doppeltgenaue Fließkommazahlen im Normalfall in zwei aufeinanderfolgenden Registern gehalten werden; in vielen Fällen lassen sich auch einfache Strukturen (zum Beispiel komplexe Zahlen) mit einem einzigen Befehl in ein Paar von Registern laden.

Im Gegensatz zum Färben von einzelnen Registern müssen für Registerpaare besondere Vorkehrungen getroffen werden. Unsere Implementation hält sich dabei großteils an die Vorschläge von Preston Briggs [Bri92, S. 33-41]. Ein etwas anderes Verfahren, das etwa zeitgleich, aber unabhängig davon entstanden ist, beschreibt Brian Nickerson in [Nic90].

Der Anzahl an Kanten zwischen zwei Knoten im Interferenzgraphen kommt beim Färben eine besondere Bedeutung zu. Die Anzahl der Kanten wird dazu benutzt, die gebundenen und freien Pseudoregister zu bestimmen; sie tritt auch als Parameter der Kostenfunktionen auf. Daher muß auch bei Registerpaaren die Anzahl an Kanten so angepaßt werden, daß die darauf aufbauenden Funktionen ihre Aufgabe korrekt erfüllen können.

Folgende Regeln, deren Motivation bei Briggs nachzulesen ist, ermöglichen dies:

1. Wenn ein Pseudoregister, welches ein Registerpaar benötigt, mit einem anderen Pseudoregister interferiert, werden zwei Interferenzkanten in den Graph eingefügt. Diese Regel gilt für Maschinen, die ausgerichtete Registerpaare verlangen.

2. Für Maschinen, bei denen die Registerpaare nicht ausgerichtet sein müssen, werden zwei Interferenzkanten zwischen einem interferierenden Registerpaar und einem Einzelregister und drei Interferenzkanten zwischen zwei interferierenden Registerpaaren eingefügt.

Während der originale Algorithmus von Chaitin auch bei dieser Darstellung in vielen Fällen den Registerbedarf überschätzt, kann der „optimistische" Algorithmus von Briggs damit ausgezeichnet Ergebnisse erzielen.

Im wesentlichen sind folgende Dinge zu implementieren:

- Ein Knoten im Interferenzgraph kann mehrere aufeinanderfolgende Register belegen.

- Die Anzahl der benachbarten Knoten und die Anzahl der Interferenzkanten kann unterschiedlich sein. Dies ist besonders beim Einfügen und Entfernen von Knoten zu beachten.

4.3.7 Registerfenster

Wenn ein Prozessor mit Registerfenstern arbeitet, müssen die Parameterübergabe- und Rückgaberegister explizit unterschiedlich gekennzeichnet sein. Andernfalls ist die Unterscheidung nicht notwendig, weil auf beiden Seiten eines Unterprogrammaufrufs die gleichen Register verwendet werden.

4.3.8 Registersuche

Beim Zuweisen von Maschinenregister an Pseudoregister müssen einige Dinge beachtet werden:

1. Der Datentyp des Pseudoregisters muß zum Datentyp des Maschinenregisters passen.
2. Zwei Nachbarn im Interferenzgraph darf nicht das gleiche Maschinenregister zugewiesen werden.
3. In Unterprogrammen, die keine weiteren Unterprogramme aufrufen, ist es günstiger, zunächst Register zu verwenden, die von aufrufenden Programm gesichert werden müsssen. Umgekehrt sollten beim Vorhandensein von Unterprogrammaufrufen vor allem für jene Pseudoregister, die über Unterprogrammaufrufe hinweg aktiv sind, Register verwendet werden, die vom aufgerufenen Programm gesichert werden müssen.

Um diese Suche schnell durchführen zu können, ist mit jedem Pseudoregister ein Bitfeld assoziiert, in dem für jedes Maschinenregister ein Bit vorhanden ist. Anhand dieses Bitfeldes kann geprüft werden, ob ein bestimmtes Maschinenregister für dieses Pseudoregister noch verfügbar ist.

Zu Beginn der Registerallokation muß zunächst einmal verhindert werden, daß gewisse Register überhaupt zugewiesen werden. Dazu gehören Register, die grundsätzlich nicht für die Allokation zur Verfügung stehen und Register, die nicht zu den Registerklassen des Pseudoregisters kompatibel sind. Diese Register werden gleich zu Beginn gesperrt.

Wenn einem Pseudoregister ein Maschinenregister zugeordnet wird, wird das gerade vergebene Maschinenregister für alle Nachbarn des Pseudoregisters gesperrt. Das kann effizient über die Liste der benachbarten Pseudoregister geschehen. Die eigentliche Registersuche beschränkt sich dann auf die Suche nach freien Bits im Bitvektor. Dabei müssen im Normalfall nur Teile des Bitvektors berücksichtigt werden — nur unter Registern einer passenden Registerklasse darf ein verfügbares Pseudoregister gesucht werden.

4.4 Unterprogramme

Für die Parameterüber- und -rückgabe bei Unterprogrammaufrufen existieren für bestimmte Prozessoren, Betriebssysteme oder gar einzelne Compiler Konventionen. Moderne Aufrufkonventionen sehen vor, daß manche Parameter in Registern und manche auf dem Stack übergeben werden. Um korrekten Code generieren zu können, muß diesem Thema besondere Sorgfalt gewidmet werden.

Hier müssen zwei Sichtweisen gesondert behandelt werden: einerseits die Sicht des aufgerufenen Unterprogrammes, welches Parameter entgegennehmen und

Rückgabewerte liefern muß, andererseits die Sicht des aufrufenden Programmes, welches die Unterprogrammparameter vor dem Aufruf an den richtigen Stellen plazieren muß.

4.4.1 Aus der Sicht des Aufgerufenen

Wenn ein Unterprogramm Parameter übernimmt, liegen beim Eintritt in das Unterprogramm wahrscheinlich einige der Parameter am Stack, während andere in Registern zu finden sind. In den meisten Fällen muß diese Anordnung aus einem der folgenden Gründe rasch geändert werden:

- Das Unterprogramm ruft seinerseits weitere Unterprogramme auf; in diesem Fall müssen die Register für die Parameterübergabe vor dem Aufruf geräumt werden[2].
- Parameter, die auf dem Stack liegen, werden eigentlich in Registern benötigt. Diese Parameter müssen dann in Register geladen werden.
- Parameter, die in Registern übergeben werden, sollten nicht in diesen Maschinenregistern gehalten werden, weil andere Pseudoregister eine höhere Priorität haben.

Um diese verschiedenen Fälle in den Griff zu bekommen, wird für jeden in einem Register übergebenen Parameter schon beim Prozedureintritt ein `COPY`-Befehl generiert, der das Maschinenregister in ein neues Pseudoregister kopiert und damit einen neuen Aktivierungsbereich beginnt. Wenn dieses Parameterregister später nicht mehr benötigt wird, zum Beispiel, weil das Unterprogramm keine weiteren Unterprogramme aufruft, weil bei keinem der weiteren Unterprogrammaufrufe dieses Register benötigt wird oder weil auch der neue Aktivierungsbereich vor dem nächsten Unterprogrammaufruf endet, kann der `COPY`-Befehl später wieder entfernt werden.

Wenn das Register allerdings benötigt wird, dann muß der `COPY`-Befehl erhalten bleiben. Dazu ist es notwendig, daß Ziel und Quelle miteinander interferieren. Das wird einfach dadurch erreicht, daß alle Pseudoregister, die mit dem selben Maschinenregister vorgefärbt wurden, identisch sind. Dann interferieren Ziel und Quelle zwar nicht direkt beim ersten `COPY`-Befehl — das wird ja sogar bewußt vermieden — sondern dann, wenn ein Aktivierungsbereich, der einem der Unterprogrammparameter entspricht, sich über einen anderen Unterprogrammaufruf hinweg erstreckt.

[2] Ausnahmen von dieser Regel liegen dann vor, wenn der Prozessor Registerfenster bietet, wie zum Beispiel die Prozessoren der `SPARC`-Familie, und wenn Parameter direkt an weitere Unterprogramme „durchgereicht“ werden.

4.4.2 Aus der Sicht des Aufrufenden

Aus der Sicht des aufrufenden Programmes ergeben sich ähnliche Notwendigkeiten: hier sind die als Parameter benötigten Pseudoregister mit großer Wahrscheinlichkeit nicht im richtigen Maschinenregister und müssen daher erst in diese kopiert werden. Auch hier werden vor dem Unterprogrammaufruf kurze Aktivierungsbereihe eingerichtet, in welche die entsprechenden Pseudoregister durch `COPY`-Befehle geschoben werden. Diese kurzen Aktivierungsbereiche werden mit den richtigen Maschinenregistern vorbelegt; je nach dem vorhandenen Möglichkeiten werden dann die `COPY`-Befehle wieder entfernt oder müssen erhalten bleiben.

Beachten Sie, daß es diese Lösung ermöglicht, daß Parameterregister zwischen Unterprogrammaufrufen für „normale“ Pseudoregister zur Verfügung stehen.

4.4.3 Ein Beispiel

Ein Beispiel: Das folgende kurze C-Programm

```
int f(int a,int b) { return( b + g(a) ); }
```

erzeugt den folgenden Code:

```
a1 = COPY p0+       ; a1 im Parameterregister P0
b1 = COPY p1+       ; b1 im Parameterregister P1
p0 = COPY a1+       ; a1 im Parameterregister P0
e0 = CALL g         ; e0 im Ergebnisregister E0
t0 = COPY e0+       ; mu\ss{} wieder herauskopiert werden
e0 = ADD b1+,t0+    ; t0 im Ergebnisregister E0
RET
```

Die Parameter `p0` und `p1` werden also zunächst kopiert, um gegebenenfalls die Register freizumachen. Vor dem Aufruf von `g()` wird der Parameter `a1` in das richtige Register geschoben, und der Rückgabewert von `g()` wird dann für die Addition verwendet. Einige der Pseudoregister sind wegen der Parameterübergabekonventionen bereits vorbelegt (wir nehmen für dieses Beispiel an, daß die Parameter in den Register `P0` und `P1` übergeben und Rückgabewerte im Register `E0` retourniert werden; außerdem verzichten wir darauf, das Parameterregister rund um den Unterprogrammaufruf zu sichern — das würde in diesem Fall den Code nur unübersichtlicher machen). Das obige Programm hat den folgenden Interferenzgraph:

```
p0: p1 b1
p1: p0 a1
a1: p1 b1
b1: p0 a1 e0
e0: b1
t0: b1
```

Alle `COPY`-Befehle können in diesem Programmstück entfernt werden, da keine Interferenzen zwischen Ziel und Quelle existieren. Wenn man noch die bereits vorbelegten Register einbezieht, ensteht daraus folgendes Maschinenprogramm:

```
e0 = CALL g
e0 = ADD p1,e0
RET
```

Ein anderer Fall liegt beim Programm

```
int f(int a,int b) { a + g(b) ); }
```

vor. Es erzeugt den folgenden, fast identischen Code

```
a1 = COPY p0+       ; a0 im Parameterregister P0
b1 = COPY p1+       ; b0 im Parameterregister P1
p0 = COPY b1+       ; a2 im Parameterregister P0
e0 = CALL g         ; t0 im Ergebnisregister E0
t0 = COPY e0+       ; mu\ss{} wieder herauskopiert werden
e0 = ADD a1+,t0+    ; t0 im Ergebnisregister E0
RET
```

woraus der Interferenzgraph

```
p0: p1 a1
p1: p0 a1
a1: p0 p1 e0 b1
b1: a1
e0: a1
t0: a1
```

resultiert, der dann zum Maschinencode

```
a1 = COPY p0        ; a0 im Parameterregister P0
p0 = COPY p1        ; a2 im Parameterregister P0
e0 = CALL g         ; t0 im Ergebnisregister E0
e0 = ADD a1,e0      ; t0 im Ergebnisregister E0
RET
```

wird. Hier muß der Parameter `a`, der in einem Parameterregister liegt, das beim Unterprogrammaufruf benötigt wird, zunächst in ein anderes Register geschoben werden, während `b` zunächst im Parameterregister bleiben kann, um dann vor dem Prozeduraufruf nach `p0` geschoben zu werden.

4.5 Heuristikfunktionen

Wenn genug Register vorhanden sind, sodaß das Auslagern von Pseudoregistern nicht notwendig ist, sind kaum Unterschiede zwischen den verschiedenen Registerallokationsverfahren erkennbar. Erst beim Auslagern werden große Unterschiede deutlich. Hier gibt es zwei wesentliche Ansatzpunkte, um den generierten Maschinencode zu verbessern:

Auslagern vermeiden. Am günstigsten ist es natürlich, das Auslagern von Pseudoregistern zu vermeiden. Das „optimistische“ Färbeverfahren von Briggs setzt an diesem Punkt an.

Auslagerungskosten verringern. Wenn ausgelagert werden muß, sollten die Auslagerungskosten möglichst gering gehalten werden. Das heißt, das Pseudoregister ausgelagert werden sollten, die entweder selten verwendet werden oder deren Verwendungen in möglichst selten ausgeführten Teilen des Programms liegen. Das Verfahren von Bernstein et al., bei dem verschiedene Heuristikfunktionen dazu verwendet werden, die Auslagerungskosten zu minimieren, setzt an diesem Punkt an.

Die parallele Anwendung mehrerer Heuristiken ist ganz allgemein bei NP-vollständigen Problemen meist von Vorteil. Die Wahrscheinlichkeit, der optimalen Lösung näher zu kommen, steigt, wenn man mehrere Methoden ausprobiert und das beste gelieferte Ergebnis übernimmt[3]. Ein anderes Beispiel für den Erfolg einer solchen Methode läßt sich bei Keßler für die lokale Registerallokation finden [KPR91].

Um die Heuristikfunktionen berechnen zu können, wird zunächst eine als „Fläche“ bezeichnet Hilfsgröße eingeführt:

$$a_p = \sum_{i \in I_p} 5^{d_i} w_i \tag{4.2}$$

I_p ist dabei die Menge von Instruktionen, bei denen ein gegebenes Pseudoregister p aktiv ist, d_i ist die Ausführungshäufigkeit des Instruktionsblocks, zu dem i gehört und w_i ist die „Breite“ einer Instruktion, die als Anzahl der bei dieser Instruktion aktiven Pseudoregister definiert ist. Der Wert von a_p ist für Pseudoregister, die vorwiegend gleichzeitig mit besonders vielen anderen Pseudoregistern aktiv sind, besonders stark an. Die Auslagerung eines solchen Pseudoregisters nimmt also besonders viel „Druck“ von den betroffenen Bereichen im Programm.

Die „nullte“ Funktion ist jene, die von Chaitin und Briggs verwendet wird:

$$h_{0_p} = \frac{c_p}{g_p} \tag{4.3}$$

[3] Briggs beschreibt in [Bri92] diese Erscheinung als „Glättung des NP-Rauschens“

Dabei ist c die Kostenfunktion aus Gleichung 3.1 und g ist der Grad des Pseudoregisters.

Die erste von Bernstein et al. verwendete Heuristikfunktion ist

$$h_{1_p} = \frac{c_p}{g_p^2} \tag{4.4}$$

Diese Funktion gibt dem Grad eines Knotens im Interferenzgraphen eine höhere Gewichtung; je höher der Grad eines ausgelagerte Pseudoregisters ist, desto größer ist die Wahrscheinlichkeit, daß ein benachbartes Pseudoregister danach vom gebundenen in den freien Zustand wechselt.

Die zweite Funktion ist

$$h_{2_p} = \frac{c_p}{a_p g_p} \tag{4.5}$$

Hier ist c_p wieder die Kostenfunktion, a_p ist die oben definierte Fläche und g_p der Grad des Pseudoregisters p.

Die dritte Funktion ist

$$h_{3_p} = \frac{c_p}{a_p g_p^2} \tag{4.6}$$

Diese Funktion entspricht h_{2_p}, gibt aber wieder dem Grad eines Knotens eine höhere Gewichtung.

4.6 Wie man den Compiler beschleunigt

Anwender von Compiler mögen allzu langsame Compiler auch dann nicht, wenn diese den besten Code erzeugen. Zumindest beim Durchführen von Tests ist auch der Programmierer selbst in den meisten Fällen für einige Zeit in der Rolle des Anwenders, und allzulange Compilerläufe verhindern dann vielleicht, daß das eine oder andere Testprogramm durch den Compiler geschickt wird.

Mit Hilfe eines Profilers und mit dem Merksatz „Mach' den häufigen Fall schnell" im Hinterkopf konnte die Compilerlaufzeit für den „häufigen Fall" — für die Algorithmen, die voreingestellt sind – im Durchschnitt um den Faktor 20 verringert werden. Dazu waren nur wenige Stunden Programmieraufwand notwendig.

Zunächst konnten auch wir die Beobachtung machen [Bri92], daß bei fast allen Compilerläufen der Registerallokator den weitaus größten Teil seiner Laufzeit mit dem mehrfachen Aufbauen des Interferenzgraphen verbraucht. Einsparungen sind also im wesentlichen hier zu erwarten.

Folgende Beobachtungen und Änderungen haben uns geholfen, die Compilerlaufzeit deutlich zu verkleinern:

- Ein Durchlauf der Schleife „Interferenzgraph aufbauen – Zusammenlegen" dauert lange genug. Je mehr `COPY`-Befehle der Zwischencode enthält, desto mehr Durchläufe dieser Schleife sind in der Regel notwendig. Wenn beim Rücktransformieren der SSA-Form keine unnötigen Instruktionen eingefügt werden, verringert das die Laufzeit des Allokators beträchtlich.

- Gemeinsam mit dem Aufbau des Interferenzgraphen wurden bei uns auch andere Dinge erledigt wie zum Beispiel das Berechnen der Auslagerungskosten. Nachdem diese Berechnung nur einmal nach dem letzen Durchlauf der Schleife „Interferenzgraph aufbauen – Zusammenlegen" notwendig ist, wurden die entsprechenden Programmteile aus dieser Schleife herausgenommen. Obwohl dann ein zweiter Durchlauf über das Programm notwendig war, konnten wir auch dadurch die Laufzeit wieder fast halbieren.

- Zum Aufbau der Konfliktmatrix müssen bei jeder Instruktion festgestellt werden, welche Pseudoregister gerade aktiv sind. In der ersten Compilerversion wurden dies mit Hilfe eines Bitfeldes gemacht, in dem jedes Bit für ein Pseudoregister stand und die Bits der gerade aktiven Pseudoregister gesetzt waren. Daher mußten in einer Schleife immer alle Bits (also alle Pseudoregister) getestet werden. Hier hat es sich als günstiger erwiesen, die aktiven Pseudoregister in einer Liste zu halten, da bei den meisten Programmen das Verhältnis zwischen der Gesamtzahl der Pseudoregister und den aktiven Pseudoregistern deutlich größer als 10 ist.

- Die notwendigen Datenstrukturen in einem optimierenden Compiler können sehr schnell wachsen, so daß auch bei den heute üblichen Speichergrößen von mehreren zehn Megabytes oft der Hauptspeicher ausgeht (vor allem auf Rechnern im Mehrbenutzerbetrieb). Daher sollte man das Programm nach „Speicherlecks" durchsuchen, belegten Speicher so früh wie möglich wieder freigeben und versuchen, den Speicherbedarf zu optimieren.

Kapitel 5

Verwandte Arbeiten

In diesem Kapitel werden die aus der Literatur bekannten Verfahren, die als Grundlage für den von uns implementierten Registerallokator gedient haben, vorgestellt. Dieses Kapitel dient als Ergänzung zu den Kapiteln „Grundlagen“ und „Implementierung“; hier finden Sie Erklärungen und Anmerkungen zu den von uns verwendeten Verfahren; vor allem haben wir uns auch bemüht, brauchbaren Pseudocode zu den bekannten Verfahren zusammenzustellen, da dieser in fast allen von uns verwendeten Quellen fehlt.

5.1 Chaitin

5.1.1 Vorbemerkungen

Chaitin war nicht der erste, der sich mit der Registerbelegung mittels Graphenfärben beschäftigt hat; er war allerdings der erste, der dieses Verfahren nicht nur beschrieben, sondern auch auf einer existierenden Maschine implementiert und in einen funktionsfähigen Compiler integriert hat. Alle anderen beschriebenen Algorithmen bauen auf der Arbeit von Chaitin auf.

Grundlage für die folgende Beschreibung waren [CAC+81] und [Cha82].

5.1.2 Der Algorithmus

Den Pseudocode finden Sie in Abbildung 5.1.

Aus dem Interferenzgraphen wird eine Liste aller Pseudoregister, psrliste, gebildet, die dann „abgearbeitet“ wird. Freie Pseudoregister können sofort entfernt und in die Liste `faerbeliste` eingehängt werden, weil für sie auf jeden Fall eine Farbe gefunden werden kann. Wenn keine freien Pseudoregister mehr zu finden sind, muß der Registerallokator eine geeignete Auswahl davon auslagern. Zur Auswahl dieser Pseudoregister dient eine geeignete Kostenfunktion, deren Wert eine Abschätzung dafür ist, wie teuer es wäre, einen Pseudoregister im Speicher anstatt in einem Maschinenregister zu halten. Die ausgelagerten Pseudoregister

```
prog chaitin
  do
    wiederhole ← false
    führe die Datenflußanalyse durch
    psrliste ← Liste aller Pseudoregister
    while ( psrliste ≠ ∅ )
      while ( freier knoten existiert )
        psrliste ← psrliste − knoten
        faerbeliste ← faerbeliste + knoten
      endwhile
      if ( gebundener knoten existiert )
        wähle knoten mit kleinstmöglichen Kostenfunktion
        psrliste ← psrliste − knoten
        auslagerliste ← auslagerliste + knoten
      endif
    endwhile
    if ( auslagerliste ≠ ∅ )
      lagere jeden knoten in auslagerliste aus
      wiederhole ← true
    else
      färbe jeden knoten in faerbeliste
    endif
  until ( wiederhole ≠ true )
endprog
```

Abbildung 5.1: *Registerbelegung nach Chaitin*

werden in die Liste `auslagerliste` eingefügt. Diese Schritte werden solange wiederholt, bis die Liste `psrliste` leer ist.

Wenn jetzt auch die Liste `auslagerliste` leer ist, befinden sich alle Pseudoregister in `faerbeliste` und können gefärbt werden. Andernfalls müssen alle Pseudoregister aus der Liste `auslagerliste` ausgelagert und das Verfahren wiederholt werden.

5.2 Briggs

5.2.1 Vorbemerkungen

An der Rice-University wurde im Rahmen eines Forschungsprojekts ein `FORTRAN`-Compiler entwickelt, dessen Registerallokator ebenfalls auf der Methode von Chaitin beruht. Beim Testen von Programmen, die sehr viele Fließkommaberechnungen enthielten und deren Interferenzgraphen relativ dicht waren, konnten

```
prog briggs
  do
    wiederhole ← false
    führe die Datenflußanalyse durch
    psrliste ← Liste aller Pseudoregister
    while ( psrliste ≠ ∅ )
      while ( freier knoten existiert )
        suche knoten mit kleinstem Grad
        psrliste ← psrliste − knoten
        faerbeliste ← faerbeliste + knoten
      endwhile
      if ( gebundener knoten existiert )
        suche knoten mit kleinstmöglichen Auslagerungskosten
        psrliste ← psrliste − knoten
        faerbeliste ← faerbeliste + knoten
      endif
    endwhile
    do
      faerbe ← false
      for ( knoten ∈ faerbeliste )
        if ( knoten kann gefärbt werden )
          färbe knoten
          faerbeliste ← faerbeliste − knoten
          faerbe ← true
        endif
      endfor
    until ( faerbe ≡ false )
    if ( faerbeliste ≠ ∅ )
      wiederhole ← true
      lagere jeden knoten in faerbeliste aus
    endif
  until ( wiederhole ≡ false )
endprog
```

Abbildung 5.2: *Registerbelegung nach Briggs*

einige Probleme mit der Heuristik von Chaitin entdeckt werden. In [BCKT89] werden diese Probleme untersucht und einige Verbesserungen präsentiert.

5.2.2 Der Algorithmus

Den Pseudocode dazu finden Sie in Abbildung 5.2.

Die Grundstruktur des verwendeten Algorithmus ist die gleiche wie bei Chaitin (vergleiche Abbildung 5.1). Allerdings sind im Detail einige Unterschiede zu erkennen.

Ein Unterschied besteht darin, daß die freien Pseudoregister nicht in beliebiger Reihenfolge aus dem Graphen entfernt werden. Vielmehr wird immer jenes freie Pseudoregister mit dem kleinsten Grad entfernt[1].

Der zweite Unterschied, der zu den größten Verbesserungen führt, zeigt sich beim eigentlichen Färben. Während Chaitin alle Pseudoregister, die beim Aufbauen der Liste `faerbeliste` zum Auslagern markiert wurden, auf keinen Fall mehr färbt, versucht Briggs trotzdem, ihnen eine Farbe zuzuweisen. Das gelingt in vielen Fällen deswegen, weil unter Umständen auch andere Nachbarn eines Pseudoregisters, die seinen Grad erhöhen und deswegen dazu geführt haben, daß es ursprünglich zum Auslagern markiert wurde, nicht oder mit einer bereits von anderen Nachbarn verwendeten Farbe gefärbt worden sind und daher noch immer eine Farbe frei sein kann. Dadurch kann mit dem Algorithmus von Briggs in vielen Fällen ein Pseudoregister doch noch gefärbt werden, das bei Chaitin auf jeden Fall ausgelagert wird.

5.3 Bernstein

5.3.1 Vorbemerkungen

In [BGG+89] finden sich weitere Verbesserungen, die direkt auf der Arbeit von Chaitin aufbauen. Diese Arbeit widmet sich vor allem Techniken, die der Minimierung der bei der Auslagerung erzeugten Instruktionen dienen. Dazu werden einerseits statt einer drei verschiedene, komplementäre Heuristikfunktionen zur Auswahl von Auslagerungskandidaten genommen und andererseits wird versucht, ausgelagerte Variablen pro Instruktionsblock nur einmal zu laden und wieder abzuspeichern.

5.3.2 Der Algorithmus

Den Pseudocode finden Sie in Abbildung 5.3.

Der Algorithmus baut wieder direkt auf dem von Chaitin auf, führt aber drei neue „Heuristikfunktionen“ ein. Diese Heuristikfunktionen werden nacheinander dazu verwendet, um Pseudoregister zu ermitteln, die ausgelagert werden müssen. Am Schluß werden die summierten Auslagerungskosten der einzelnen Listen von Pseudoregistern verglichen; die Auslagerungsliste mit den kleinsten summierten Kosten wird dann zum Färben verwendet. Die eigentliche Kostenfunktion ist die,

[1] Briggs verwendet dazu in seiner Implementierung eine eigene Datenstruktur, die den Vorgang dieser Suche optimiert. Diese wurde von uns aus Gründen der Einheitlichkeit nicht verwendet, wodurch sich das Laufzeitverhalten etwas verschlechtert

```
prog bernstein
   do
      wiederhole ← false
      führe die Datenflußanalyse durch
      psrliste ← Liste aller Pseudoregister
      auslagerkosten ← 0
      for ( h ∈ Heuristiken )
         sichere den Interferenzgraphen
         while ( psrliste ≠ ∅ )
            while ( freier knoten existiert )
               knoten ← freier knoten mit größtem Grad
               psrliste ← psrliste − knoten
               faerbeliste ← knoten + faerbeliste
            endwhile
            if ( gebundener knoten existiert )
               suche knoten für den h minimal ist
               psrliste ← psrliste − knoten
               auslagerliste ← knoten + auslagerliste
               auslagerkosten ← auslagerkosten + kosten (knoten )
            endif
         endwhile
         sichere faerbeliste und auslagerliste
         stelle den Interferenzgraphen wieder her
      endfor
      wähle Heuristik h faerbeliste und auslagerliste mit kleinsten Auslager-
ungskosten
      if ( auslagerliste ≠ ∅ )
         lagere jeden knoten in auslagerliste aus
         bereinige die Instruktionsblöcke von unnötigen Speicherreferenzen
         wiederhole ← true
      else
         färbe jeden knoten in faerbeliste
      endif
   until ( wiederhole ≡ false )
endprog
```

Abbildung 5.3: *Graphenfärben nach Bernstein.*

welche auch von Chaitin verwendet wird. Sie ist in Gleichung 3.1 auf Seite 27 wiedergegeben.

Zusätzlich wird die Technik des „Bereinigens“ zur Minimierung der Anzahl der beim Auslagern notwendigen Speicher- und Ladebefehle verwendet. Dabei wird versucht, beim Auslagern mit nur einem Lade- respektive Speicherbefehl

```
prog chow;
  führe die Datenflußanalyse durch
  psrliste ← Liste aller Pseudoregister
  gebunden ← true
  while ( gebunden )
    gebunden ← false
    for ( t ∈ psrliste )
      if ( t ist gebunden )
        versuche t zu splitten
      endif
    endfor
    c ← Pseudoregister mit höchster Priorität
    if ( c in einem Register gehalten werden soll )
      if ( c kann gefärbt werden )
        färbe c
      else
        lagere c aus
      endif
    else
      lagere c aus
    endif
  endwhile
endprog
```

Abbildung 5.4: *Graphenfärben nach Chow/Hennessy.*

pro Instruktionsblock auszukommen. Gelingt dies nicht, dann ist ein weiterer Durchlauf des Allokators notwendig.

5.4 Chow

5.4.1 Vorbemerkungen

Ein Verfahren, das etwas von den bisher vorgestellten abweicht, wurde von Chow entwickelt. Diese Verfahren wird „Priority Based Coloring“ genannt und ist in [CH90] und [LH86] beschrieben.

5.4.2 Der Algorithmus

Den Pseudocode finden Sie in Abbildung 5.4.

Das „Prioritätsgesteuerte Färben“ nach Chow und Hennessy weist einige bedeutende Unterschiede zu den anderen hier beschriebenen Verfahren vor. Die wesentlichen Unterschiede sind:

- Das Färben erfolgt in einem einzigen Durchlauf erfolgt, in dem die nach ihrer „Priorität“ sortierten Pseudoregister eines nach dem anderen gefärbt oder sofort ausgelagert werden. Anhand der Abbildung 2.2 auf Seite 18 läßt sich das illustrieren: beim Prioritätsgesteuerten Färben werden die Maschinenregister bereits beim Abbauen des Graphen vergeben; das Wiedereinfügen der Knoten kann daher entfallen. Daher erfolgt das Abbauen auch in umgekehrter Reihenfolge: zuerst werden jene Pseudoregister gefärbt, welche die höchste Priorität haben.

- Bevor ein Pseudoregister ausgelagert wird, versucht der Registerallokator, seinen Aktivierungsbereich in mehrere, kürzere Bereiche aufzuteilen

5.5 Weitere Verfahren

5.5.1 Hierarchisches Färben

Ein entscheidender Schwachpunkt der bisher vorgestellten Registerallokatoren liegt darin, daß sie die Kontrollstruktur eines Programms nur ungenau modellieren. Nur durch die Kostenfunktionen fließt die meistens geschätzte Ausführungshäufigkeit in die Berechnung ein.

Callahan und Koblenz stellen in [CK91] einen Registerallokator vor, der zwar einerseits ebenfalls auf Graphenfärben basiert, der aber andererseits versucht, die Kontrollstruktur genauer abzubilden. Um dieses Ziel zu erreichen, wird das Programm mit der Methode von Sharir [Sha80] in hierarchisch ineinander verschachtelte Intervalle aufgeteilt[2], welche die „natürliche“ Kontrollstruktur eines Programmes recht gut wiedergeben; zum Beispiel stellen der Rumpf einer Schleife oder die einzelnen Zweige einer `if` – `then` – `else` - Konstrukt eigene Intervalle dar — daher kommt auch der Name „hierarchisches Färben“ („Hierarchical Coloring“).

Dann wird die Registerallokation zweimal separat auf jedem Teilintervall durchgeführt: im ersten Durchgang wird der Registerbedarf festgestellt und diese Information von „unten“ nach „oben“ gereicht. Im zweiten Durchgang, der in umgekehrter Richtung durchgeführt wird, werden die Register dann tatsächlich vergeben. Dadurch kann garantiert werden, daß die in einem Intervall wichtigen Pseudoregister tatsächlich in Maschinenregistern gehalten werden; aktive, aber nicht benötigte Pseudoregister, die nur den Registerbedarf erhöhen, können in diesen Intervallen im Speicher abgelegt werden.

[2] Diese durch die Intervallanylse von Sharir gewonnenen Intervalle werden noch leicht verändert und als „Tiles“ bezeichnet.

Da der Färbealgorithmus immer nur auf einem kleinen Teil des Programmes durchgeführt wird, benötigen die notwendigen Datenstrukturen wesentlich weniger Platz; auch der Gesamtaufwand, der ja beim Graphenfärben etwa mit dem Quadrat der Programmgröße steigt, ist in den meisten Fällen geringer.

Ein Nachteil des Verfahrens ist zweifellos, daß eine ziemlich komplexe Maschinerie benötigt wird, um die notwendigen Informationen zwischen den einzelnen, separat gefärbten Intervallen hin- und herzutransportieren. So sind zum Beispiel spezielle Vorkehrungen notwendig, um Pseudoregistern, die in mehreren Intervallen aktiv sind und überall in einem Maschinenregister gehalten werden können, nach Möglichkeit in allen Intervallen das gleiche Maschinenregister zuzuweisen, damit möglichst wenige Register-Register-Verschiebungen notwendig sind.

Aufgrund der fehlenden Vergleiche in [CK91] läßt sich auch keine schlüssige Aussage über die Leistungsfähigkeit des Algorithmus machen. Interessant ist aber, daß ein ähnlicher Ansatz, der von Briggs getestet wurde und in [Bri92] beschrieben ist, in den meisten Fällen enttäuschende Ergebnisse brachte, was aber, wie Briggs selbst anmerkt, auch daran liegen kann, daß der veröffentlichte Algorithmus noch nicht ganz ausgereift war.

5.5.2 Interprozedurale Registerallokation

Während die globale Registerallokation inzwischen von fast allen Compilerbauern favorisiert wird[3] und auch bereits ausgereifte Verfahren dafür existieren, bleibt für die interprozedurale Registerallokation noch viel zu tun.

Einige der Implementationsentscheidungen, die für einen interprozeduralen Registerallokator zu treffen sind, wären:

- Werden globale Variablen vom Algorithmus erfaßt, sodaß sie in allen verwendenden Unterprogrammen im gleichen Register sind?
- Wie wird das Problem der separaten Übersetzung gelöst? Werden nur die Unterprogramme in einem Quellcodemodul mit einbezogen oder gibt es einen Mechanismus, der auch die Allokation über Übersetzungseinheiten hinweg ermöglicht?

Ein Ansatz, bei dem sowohl globale Variablen alloziert als auch die Registerallokation über die Grenze der Übersetzungseinheiten weg gelöst sind, stammt von David Wall [Wal86][Wal92]. Der wesentliche Punkt an seinem Algorithmus ist der, daß der interprozedurale Teil der Registerallokation vom Linker erledigt wird.

[3] Eine Ausnahme von dieser Behauptung muß für Christopher Fraser und David Hanson gemacht werden: in ihren Codegeneratoren für den `lcc` verfolgen sie die Strategie, die meisten Optimierungen so einfach und effizient wie möglich zu machen; auch die Registerallokation ist davon nicht ausgenommen — ihr Verfahren beschreiben sie in [FH92].

Eine Erweiterung des prioritätsgesteuerten Registerallokationsverfahrens, das auch interprozedurale Allokation zuläßt, stellt Chow in [Cho88] vor. Bei diesem Verfahren werden globale Variablen nicht mit einbezogen, und die Allokation über die Grenze von Übersetzungseinheiten hinweg ist nur eingeschränkt möglich.

Steenkiste und Hennessy beschreiben in [SH89] einen interprozeduralen Registerallokator für einen LISP-Compiler. Compilierter LISP-Code bietet sehr gute Verbesserungsmöglichkeiten für die interprozedurale Allokation, weil die Häufigkeit von Unterprogrammaufrufen besonders groß ist.

Bemerkenswert ist, daß alle drei Verfahren sich des Graphenfärbens bedienen und auf dieser Basis ihre Erweiterungen implementiert haben.

5.5.3 Registerallokation und Pipelining

Ein zweites heikles Thema ist die zunehmende Nutzung von Möglichkeiten zur parallelen Verarbeitung in modernen Prozessoren, die hauptsächlich in Form von Befehlspipelines auftritt. Um diese vorhandenenen Möglichkeiten nutzen zu können, muß auch der erzeugte Maschinencode entsprechend „präpariert“ werden. Die wichtigste Vorbereitung ist das Umordnen von Instruktionen, um die Effekte von Datenabhängigkeiten gering zu halten („Instruction Scheduling“) [HP90] [War90] [GM86] [SKAH91] [EK92].

Je nach den Gegebenheiten im jeweiligen Compiler wird die Umordnungsphase normalerweise entweder vor („prepass“), nach („postpass“) oder sowohl vor als auch nach dem Registerallokator durchgeführt.

Integrierte Verfahren, welche die Probleme der Umordnung und das der Registerallokation gleichzeitig beachten, sind noch nicht sehr verbreitet [GH88] [BEH91]. Auch von uns wurde der Versuch gemacht, diese beiden Compilerphasen zu integrieren; der entsprechende Algorithmus, „Dependence-Conscious Register Allocation“, ist allerdings noch im Versuchsstadium [EABK92].

5.5.4 Sonstige Verfahren

Das Aufteilen des Interferenzgraphen in mehrere kleinere Einheiten, die einzeln leichter zu färben sind, wird in [GSS89] vorgeschlagen. Als neue Einheiten werden „Cliquen“ innerhalb des Interferenzgraphen gewählt. Wie beim hierarchischen Färben tritt auch hier das Problem auf, daß die separat gefärbten Subgraphen schließlich wieder zu einem einheitliche Ganzen zusammengefügt werden müssen.

Zwei interessante Algorithmen werden in [HGAM92] beschrieben. Der eine, von den Autoren „fat cover“ genannt, setzt an den „breitesten“ Stellen eines Interferenzgraphen an, also an den Stellen, an denen die meisten Variablen gleichzeitig aktiv sind. Von diesen Stellen ausgehend. die sich mit Hilfe von „zyklischen Intervallgraphen“ finden lassen, wird der Graph wieder schrittweise vereinfacht. Das Verfahren bringt angeblich bessere Ergebnisse als „normale“ Registerallokatoren, allerdings krankt der im Artikel präsentierte Vergleich daran, daß als Heuristik nur die „pessimistische“ von Chaitin genommen wurde.

Der zweite Algorithmus erlaubt es, das Auslagern von Pseudoregistern in den Hauptspeicher zu vermeiden und gegebenfalls durch Kopieroperationen in andere Register zu ersetzen. De facto werden also Aktivierungsbereiche wieder in kleinere, durch Registerkopierbefehle getrennte Einheiten zerlegt.

Kapitel 6

Vergleich

6.1 Vergleichsmethoden

6.1.1 Rechner

Für die Ermittlung der Daten wurde eine NeXTStation Mono, Prozessor MC 68040 mit 25MHz Taktfrequenz, 20Mb Hauptspeicher und 400Mb Plattenspeicher eingesetzt. Der Rechner lief unter NeXTStep 3.0, war nicht in ein Netzwerk eingebunden und arbeitete daher ohne nennenswerte Hintergrundlast. Mit Hilfe von des MIPS-Simulators SPIM [Lar91] wurde der MIPS R2000 simuliert.

6.1.2 Registersätze

Die drei folgenden Registersätze wurden für die Tests verwendet (Beim prioritätsgesteuerten Verfahren müssen zusätzlich noch vier Ganzzahl– und zwei Fließkommaregister für die nachträgliche Generierung von Auslagerungscode reserviert werden):

MI32 Dieser Registersatz entspricht dem normalen Registersatz eines MIPS R2000-Prozessors. Von den insgesamt 64 Registern sind 24 Ganzzahl- und 16 Fließkommaregister für den Registerallokator verfügbar[1]

MI16 Diese Registersatz wurde aus dem `MI32`-Registersatz durch das Weglassen einiger Register gewonnen. Insgesamt sind mit diesem Satz 16 Ganzzahl- und 8 Fließkommaregister benutzbar.

[1] Einige Ganzzahlregister sind beim R2000 reserviert, und zwar der Stackpointer, der Framepointer und der „Global"-Pointer, das Register für die Rücksprungadresse, zwei „Kernel"-Register, ein Register für den Assembler und das Register mit dem festverdrahteten Wert 0. Von den 32 Fließkommaregistern sind nur 16 nutzbar, weil auch 32-Bit-Fließkommazahlen nur in Registern mit gerader Registernummer verarbeitet werden können.

MI08 Dieser Registersatz ist ein weiter verkleinerter `MI16`-Satz. Insgesamt sind in diesem Satz 10 Ganzzahl- und 6 Fließkommaregister benutzbar. Er ist allerdings mit der von uns verwendeten MIPS-Unterprogrammaufrufkonvention bereits etwas zu klein: 6 Ganzzahl- und 4 Fließkommaregister werden für die Parameterübergabe verwendet. Sie stehen zwar zwischen Unterprogrammaufrufen auch für andere Zwecke zur Verfügung, doch der verbleibende Rest der Register ist zu klein, um noch vernünftige Vergleichsdaten liefern zu können. Auch das Ändern der Aufrufkonvention für diesen Fall hätte uns die Vergleichsmöglichkeiten genommen. Außerdem dürfen die für das prioritätsgesteuerte Färben reservierten Register nicht identisch mit Parameter-Übergaberegistern sein, so daß in diesem Fall die 6 notwendigen Register noch zusätzlich zur Verfügung gestellt werden mußten. Dieser Registersatz wurde daher nur in einigen wenigen Fällen zu Vergleichen herangezogen.

6.1.3 Testprogramme

Folgende Testprogramme wurden verwendet:

- Das Frontend unseres Compilers `lcc`. Der Compiler selbst darf in keiner Compiler-Testsuite fehlen.

- `doduc` (Nhuan Doduc), ein vielverwendetes Benchmarkprogramm. Die originalen Fortran-Quelltexte wurden mit Hilfe des Fortran-nach-C-Compilers `f2c` nach C übersetzt. Die Programme des `DODUC` stehen stellvertretend für die wichtige Klasse der technisch-wissenschaftlichen Berechnungsprogramme.

- `dvips 5.491` (Thomas Rockicki), ein Programm, das TeX-`DVI`-Dateien nach Postscript übersetzt. Dieses Programm wurde als typisches rechenintensives Anwendungsprogramm in die Testsuite aufgenommen. Es ist äußerst portabel, besteht aus einer großen Anzahl von Unterprogrammen und gehört mit etwa 12000 Zeilen nicht mehr zu den ganz kleinen Programmen. Außerdem enthält es einige extrem lange Unterprogramme, die hauptsächlich aus einer großen `switch`-Instruktion bestehen. Derartige Konstrukte kommen in der Praxis relativ häufig vor.

- Die Public-Domain-Version des UNIX-Dienstprogrammes `compress`. Dieses Programm steht für die große Zahl von kleinen Dienstprogrammen, wie sie jedermann täglich verwendet.

- Das Public-Domain-Tabellenkalkulationsprogramm `sc`, Version 6.8 (Robert Bond). Auch ein interaktives Anwendungsprogramm darf in einer Testsuite nicht fehlen.

Diese Testsuite umfaßt insgesamt 40.144 Programmzeilen mit 695 Unterprogrammen. Während sie für einen kommerziellen Compiler viel zu klein wäre — in der Industrie sind Testsuiten mit mehreren Millionen Zeilen üblich — ist sie im Vergleich zu ähnlichen Forschungsprojekten relativ groß. Briggs nimmt in [Bri92] beispielsweise etwa 30 Einzelfunktionen zur Ermittlung seiner Testdaten. Dadurch können wir gewollte oder ungewollte statistische Schwankungen, wie sie bei kleineren Testsuiten zwangsweise auftreten, stark reduzieren.

6.1.4 Verglichene Verfahren

Gemessen wurden die folgenden Verfahren (bei einigen Verfahren wurden weitere Daten mit zusätzlichen Optionen — zum Beispiel für die Registersuche — ermittelt):

Pess Pessimistisches Färben nach Chaitin (Compileroption `-Rco_dumb`).

Opt Optimistisches Färben nach Briggs (Compileroption `-Rco_lazy`).

Mult Färben mit mehreren Heuristikfunktionen nach Bernstein kombiniert mit dem pessimistischen Färbeverfahren[2] (Compileroption `-Ralloc_multiple -Rco_dumb`).

Prio A Prioritätsgesteuertes Färben nach Chow (Compileroption `-Ralloc_priority`).

Prio B Prioritätsgesteuertes Färben nach Chow mit verkleinerten Instruktionsblöcken[3] (Compileroption `-Ralloc_multiple -Q1`).

6.2 Anteile der Verfahren am Gesamtprogramm

Unser experimenteller Registerallokator umfaßt etwa 6500 Zeilen Programmcode. Das ist etwas mehr, als eine minimale Implementation benötigen würde, die nur ein ausgewähltes Verfahren verwendet. Gemessen an unserem Projekt wären dafür etwa 3000 Zeilen notwendig, wobei etwa 900 Zeilen Hilfsfunktionen enthalten, die auch von anderen Teilen des Compilers verwendet werden können.

Der Anteil der einzelnen Programmteile am Gesamtprogramm ist in der Tabelle auf Seite 70 zusammengefaßt.

[2] Die Kombination von mehreren Heuristikfunktionen mit dem optimistischen Färben macht wenig Sinn, weil die von den Heuristiken ermittelten „Auslagerungsempfehlungen“ vom optimistischen Färbealgorithmus sowieso ignoriert werden

[3] Die Blöcke werden dabei an DAG-Grenzen aufgebrochen; daher umfaßt ein Block auch nach der Verkleinerung im allgemeinen noch mehrere Maschineninstruktionen.

Zweck	Zeilen	Anteil
Konfliktmatrix aufbauen	340	5.2%
Konfliktvektoren aufbauen	60	0.9%
Zusammenlegen	80	1.2%
Gemeinsamer Code für alle Verfahren	50	0.7%
Verfahren `Pess`	120	1.8%
Verfahren `Opt`	160	2.4%
Verfahren `Mult`	110	1.7%
Verfahren `Prio`	580	9.0%
Auslagern bei `Pess`, `Opt`, `Mult`	330	5.1%
Auslagern bei `Prio`	280	4.3%
Maschinenregister	710	11.0%
Registerbeschreibung MIPS	120	1.8%
Hilfsfunktionen	880	13.6%
Debugging	460	7.1%
Sonstiges (Initialisierung, Deklarationen ...)	880	13.6%
Nicht verwendete Verfahren	1300	20.1%
Gesamt	6440	

Anteil der einzelnen Verfahren am Gesamtprogramm

6.3 Statistisches zu den Testprogrammen

Zunächst wurden einige grundsätzliche statistische Daten zu den Testprogrammen gesammelt. Diese Daten können bei der Entwicklung weiterer Algorithmen als Grundlage zur Aufwandsberechnung genommen werden, zumal in der Literatur sonst wenig zu diesem Thema zu finden ist.

Die hier gesammelten Werte decken sich sehr gut mit denen in [Bri92]. Sie sind in der Tabelle auf der Seite 71 tabellarisch dargestellt. Die Daten wurden für jede übersetzte Funktionen ermittelt; angegeben sind jeweils Minimum, Maximum und Median sowie in der Spalte „Gesamt“ die Summe der Werte über alle Funktionen in einem Programm.

In Abbildung 6.1 auf Seite 72 ist das Verhältnis zwischen der Anzahl der Instruktionsblöcke, Instruktionen und Pseudoregister bei verschiedenen Programmen graphisch dargestellt.

Während die Programme `dvips`, `lcc` und `sc` relativ kurze Instruktionsblöcke und etwa 2–3 mal soviele Instruktionen wie Pseudoregister aufweisen, fällt `doduc` hier durch die besonders langen Instruktionsblöcke aus dem Rahmen. Diese langen Blöcke sind typisch für Programme aus dem technisch-wissenschaftlichen Bereich, weil dort häufig lange arithmetische Ausdrücke vorkommen.

	Minimum	Maximum	Median	Gesamt
`compress` (25 Funktionen, 2369 Zeilen)				
Zeit (s)	0.00	6.23	0.03	11.0
Aufrufe	0.00	101.00	3.00	234.00
Interferenzkanten	0.00	1280.00	12.00	4764.00
Instruktionen	5.00	955.00	34.00	2956.00
Pseudoregister	1.00	190.00	8.00	754.00
Blöcke	3.00	177.00	10.00	609.00
`doduc` (41 Funktionen, 5326 Zeilen)				
Zeit (s)	0.03	369.81	3.92	844.00
Aufrufe	0.00	143.00	1.00	294.00
Interferenzkanten	28.00	107488.00	6636.00	616592.00
Instruktionen	23.00	3778.00	499.00	27525.00
Pseudoregister	10.00	2930.00	293.00	18159.00
Blöcke	3.00	465.00	36.00	2489.00
`dvips` (170 Funktionen, 12206 Zeilen)				
Zeit (s)	0.00	16.64	0.47	81.00
Aufrufe	0.00	141.00	3.00	1743.00
Interferenzkanten	0.00	12990.00	39.00	64630.00
Instruktionen	2.00	1443.00	45.00	19602.00
Pseudoregister	0.00	381.00	13.50	5977.00
Blöcke	3.00	313.00	10.50	4145.00
`lcc` (255 Funktionen, 10288 Zeilen)				
Zeit (s)	0.00	92.02	0.08	193.00
Aufrufe	0.00	245.00	4.00	2467.00
Interferenzkanten	0.00	6618.00	76.00	62116.00
Instruktionen	7.00	3681.00	62.00	34296.00
Pseudoregister	2.00	816.00	19.00	9979.00
Blöcke	3.00	682.00	15.00	7334.00
`sc` (204 Funktionen, 9955 Zeilen)				
Zeit (s)	0.00	41.70	0.06	106.00
Aufrufe	0.00	299.00	3.00	1766.00
Interferenzkanten	0.00	7062.00	45.00	42278.00
Instruktionen	3.00	2722.00	41.00	20657.00
Pseudoregister	0.00	381.00	13.00	4924.00
Blöcke	3.00	397.00	11.00	4090.00

Statistische Daten zu den Testprogrammen (Registersatz MI32)

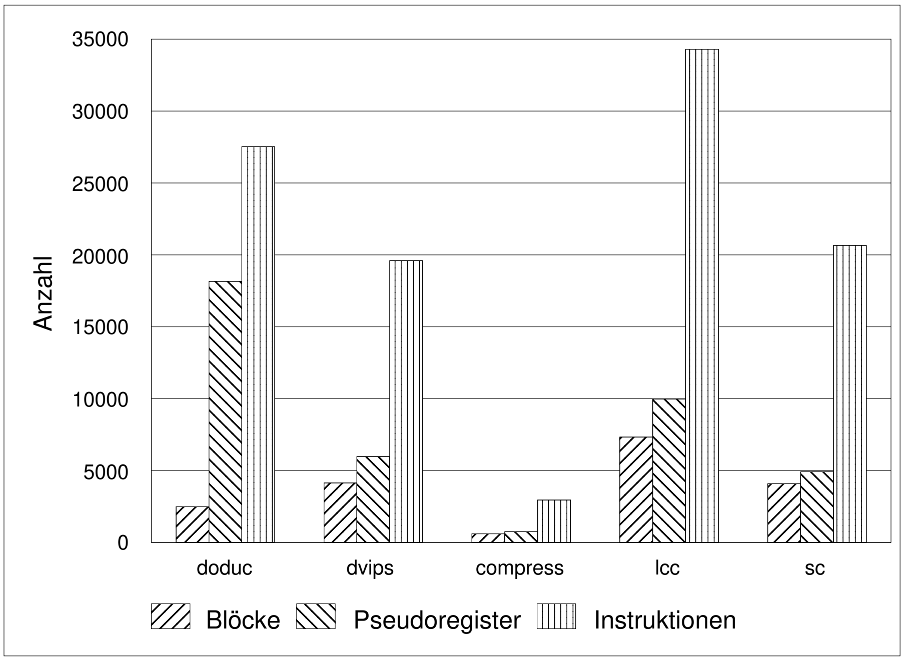


Abbildung 6.1: *Verhältnis zwischen der Anzahl der Instruktionsblöcke, Instruktionen und Pseudoregister bei verschiedenen Programmen*

6.4 Ressourcenbedarf des Registerallokators

6.4.1 Zeit

Einige Daten zu den Laufzeiten des Registerallokators bei verschiedenen Verfahren sind in Abbildung 6.2 auf Seite 73 graphisch dargestellt. Wegen der kurzen Laufzeiten bei `compress` wurden die Daten zu diesem Programm nicht in die Graphik übernommen.

Wie beim Verhältnis zwischen Instruktionsblöcken und Instruktionen fällt auch hier `doduc` wieder aus der Reihe, weil im Gegensatz zu den anderen dargestellten Testprogramme sich hier deutliche Unterschiede in den Laufzeiten zwischen den Verfahren `Opt`, `Pess` und `Mult` zeigen. Der deutlich größere Zeitbedarf für `Pess` und `Mult` gegenüber `Opt` erklärt sich aus der Tatsache, daß beim Verfahren `Opt` relativ wenige Pseudoregister ausgelagert werden, die Zeit für das Auslagern und die neuen Durchläufe des Registerallokators bei `Pess` und `Mul` dominieren hier die Laufzeit. Bei den anderen Testprogrammen kommt Auslagern kaum vor, daher unterscheiden sich auch die Laufzeiten kaum.

Außer bei `doduc` mit `PRIO A`, wo die langen Instruktionsblöcke zur Senkung der Laufzeit beitragen, benötigt das prioritätsgesteuerten Färben die meiste Zeit.

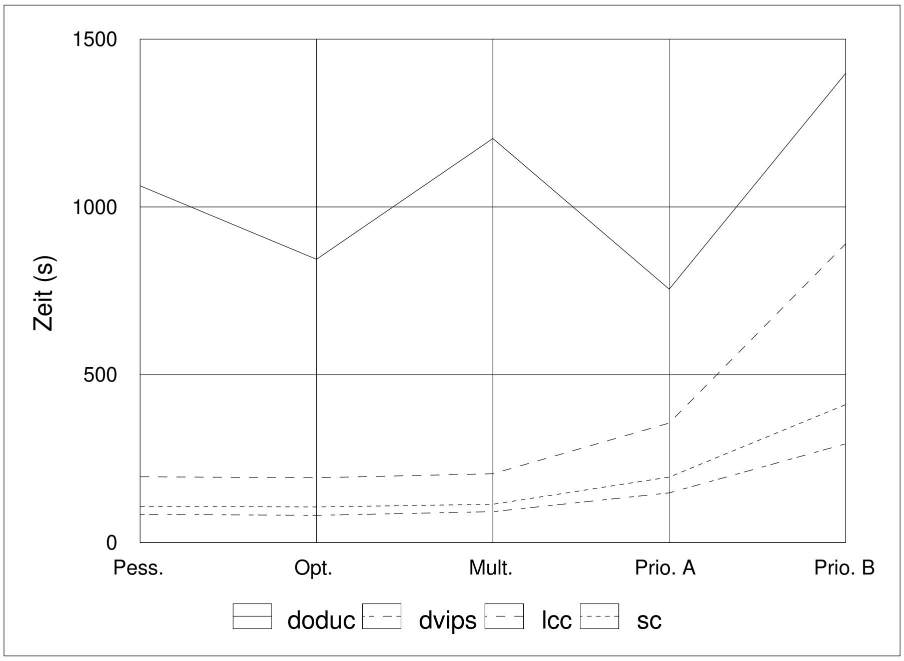


Abbildung 6.2: *Laufzeiten des Registerallokators für verschiedene Programme (Registersatz MI32)*

Vor allem bei den verkleinerten Instruktionsblöcken unter `PRIO B` fällt diese Tendenz auf.

Abbildung 6.3 auf Seite 74 zeigt, welchen Anteil die einzelnen Hauptphasen an der Laufzeit des Registerallokators haben. Dargestellt werden die relativen Laufzeiten *im ersten Durchgang* des Registerallokators, in dem auch das Entfernen von Kopierinstruktionen durchgeführt wird. Die Zeiten stammen vom Verfahren `Opt`. In den weiteren Durchläufen geht die Zeit für das Zusammenlegen praktisch gegen 0, während sich die Verhältnisse der anderen Phasen zueinander kaum ändern.

6.4.2 Iterationen

Wenn die verfügbaren Register zum Färben eines gegebenen Konfliktgraphen nicht ausreichen, müssen einzelne Pseudoregister ausgelagert werden; danach wird ein weiterer Lauf des Registerallokators gestartet (ausgenommen beim prioritätsgesteuerten Färben, bei dem nur ein Durchlauf notwendig ist). Diese Durchläufe kosten Zeit. Daher haben wir uns gefragt, wie viele Durchläufe im allgemeinen bei verschiedenen Verfahren beziehungsweise verschiedenen Registersätzen notwendig sind.

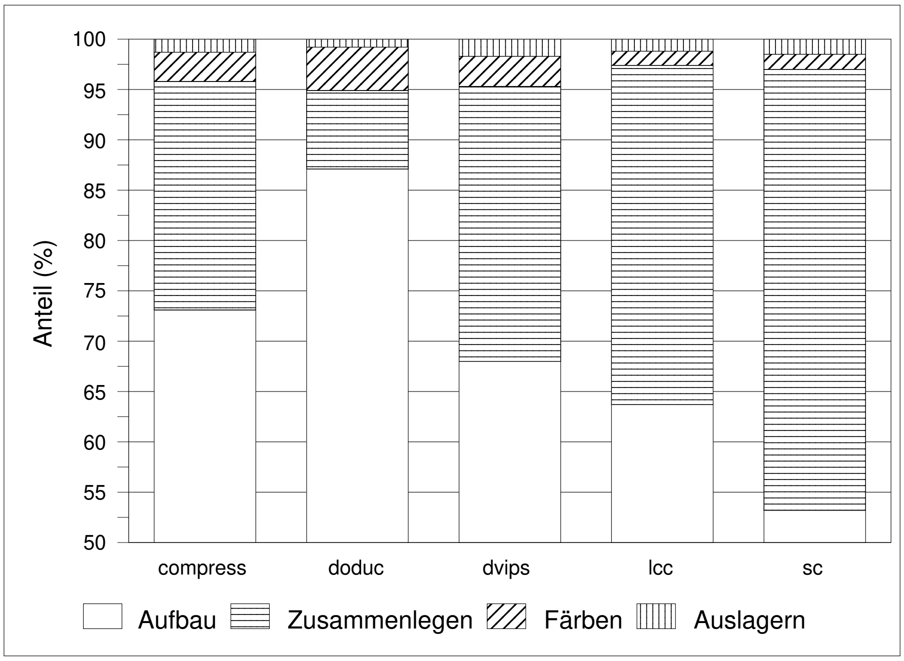


Abbildung 6.3: *Anteile der einzelnen Phasen an der Laufzeit des ersten Durchganges*

Abbildung 6.4 auf Seite 75 zeigt, wie viele Durchläufe des Registerallokators bei verschiedenen Verfahren (`Opt`, `Pess` und `Mul`) für die einzelnen Funktionen aus unserer Testsuite notwendig waren.

In Abbildung 6.5 auf Seite 76 ist zu sehen, wie sich die Anzahl der Durchläufe bei verschieden großen Registersätzen ändert. Für diese Darstellung wurde das Verfahren `Pess` gewählt, weil es Daten liefert, die auch für die anderen Verfahren repräsentativ sind.

6.4.3 Speicherbedarf

Der ungefähre Speicherbedarf des Registerallokators ist in Abbildung 6.6 auf Seite 77 dargestellt. Da sich die einzelnen Verfahren im Speicherbedarf nur geringfügig unterscheiden, wurden nur die gesammelten Daten für das Verfahren `Opt` dargestellt.

Dabei ist zu beachten, daß zur Gewinnung der Daten für jede übersetzte Funktion der Speicherbedarf für den Registerallokator allein geschätzt wurde. Dabei wurden Faktoren wie die Speicherfragmentation nicht eingerechnet; zum angegebenen Bedarf kommt jeweils noch der Bedarf der anderen Teile des Compilers

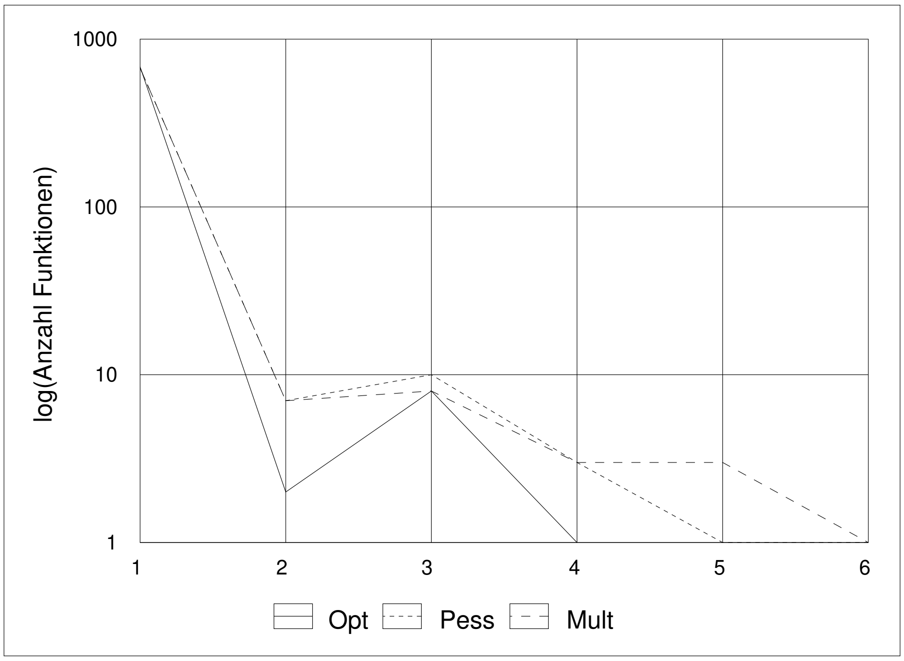


Abbildung 6.4: *Anzahl der Durchläufe des Registerallokators bei verschiedenen Verfahren (Registersatz MI32)*

dazu, so daß der tatsächliche Speicherbedarf des gesamten Compilers in manchen Fällen mehr als doppelt so hoch ist wie angegeben.

6.5 Auslagern

6.5.1 Kosten

Die Auslagerungskosten bei verschiedenen Verfahren sind in den Tabellen 6.5.1, 6.5.1 und 6.5.1 auf den Seiten 77, 78 und 79 zusammengestellt. Aufgelistet werden: die Anzahl der eingefügten Lade- und Speicherbefehle, die Anzahl der insgesamt ausgelagerten Pseudoregister sowie der Logarithmus $\log_{10}$ der Auslagerungskosten.

Auf den ersten Blick erkennt man, daß der MIPS-Registersatz für eine nicht unerhebliche Zahl von Programmen vollständig ausreicht: `compress` und `lcc` kommen bei allen Verfahren ohne das Auslagern von Pseudoregistern aus, bei `dvips` muß auch nur eine unerhebliche Anzahl an Pseudoregistern ausgelagert werden.

Insgesamt zeigt sich hier, daß das optimistische Färben bei den Auslager-

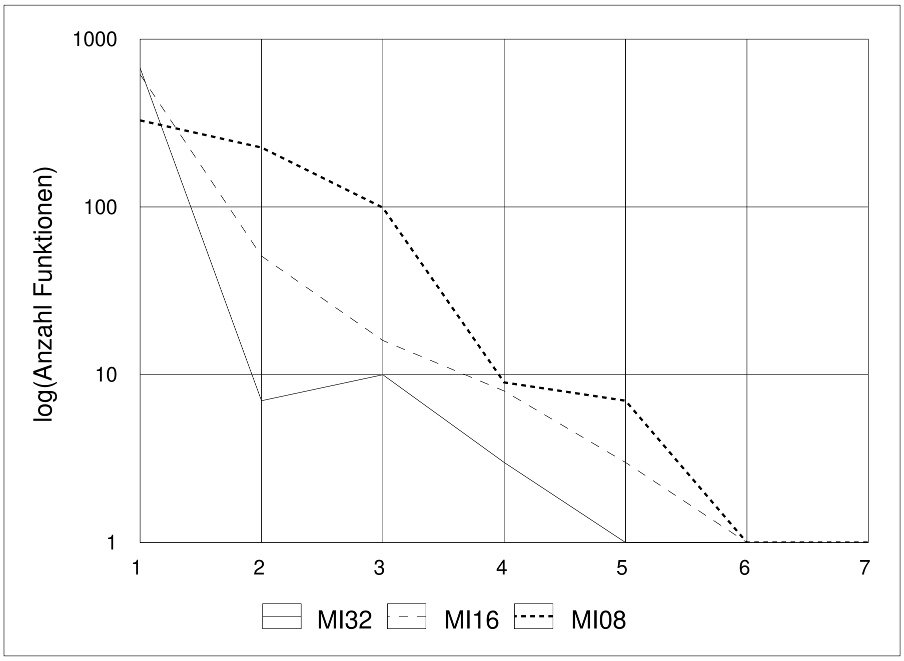


Abbildung 6.5: *Anzahl der Durchläufe des Registerallokators bei verschiedenen Registersätzen*

ungskosten in fast allen Fällen um mindestens eine Größenordnung besser als irgendein anderes Verfahren abschneidet.

Interessant ist, daß die Anwendung von mehreren Heuristikfunktionen in unserem Compiler nicht in allen Fällen Verbesserungen bei den Auslagerungskosten ergibt. Vielmehr ist die originale Heuristik von Chaitin (Gleichung 4.3 auf Seite 54) in einigen hier vorkommenden Fällen sogar besser als die Kombination der drei verwendeten Alternativheuristiken. Auch bei den drei Alternativheuristiken liefert die Heuristik h_{1_p} bei weitem am häufigsten die besten Ergebnisse. Die Daten für 821 Durchläufe des Allokators sind:

Heuristik	Anzahl	Anteil
h_{1_p}	798	97.1%
h_{2_p}	12	1.4%
h_{3_p}	11	1.3%

Das Ergebnis in [BGG+89] — keine der Heuristiken dominiert — kann von uns nicht bestätigt werden.

Zunächst war es naheliegend, von diesem Verhalten auf einen Fehler in unserer Implementierung zu schließen. Nach genauer Untersuchung des Programmes sowie einigen Testläufen, verbunden mit kleinen Änderungen an den Funktionen können wir derartige Fehler mit großer Sicherheit ausschließen. Einige Erklärun-

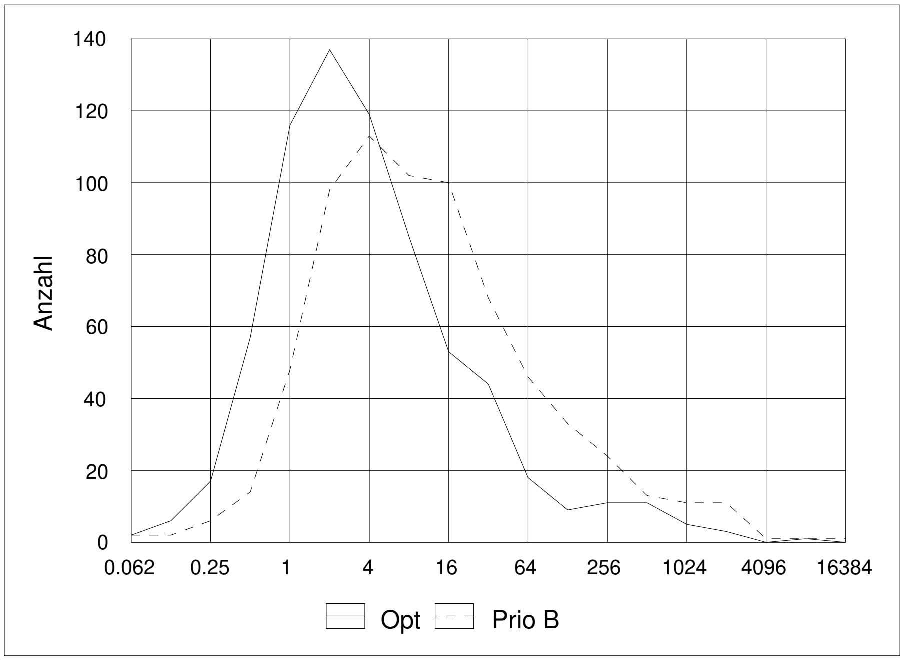


Abbildung 6.6: *Speicherbedarf des Registerallokators*

	Opt	Pess	Mult	Prio A	Prio B
`doduc`					
Ladebefehle	494.00	1495.00	1557.00	1448.00	999.00
Auslagerkosten	7.86	9.76	8.48	9.89	8.36
Auslagerungen	384.00	3975.00	922.00	3842.00	348.00
Speicherbefehle	394.00	779.00	969.00	650.00	394.00
`dvips`					
Ladebefehle	0.00	25.00	13.00	23.00	13.00
Auslagerkosten	0.00	3.83	7.69	3.83	7.69
Auslagerungen	0.00	167.00	2.00	166.00	2.00
Speicherbefehle	0.00	13.00	6.00	11.00	6.00
`sc`					
Ladebefehle	2.00	34.00	30.00	35.00	30.00
Auslagerkosten	9.29	10.22	10.00	10.27	10.00
Auslagerungen	1.00	9.00	5.00	8.00	5.00
Speicherbefehle	2.00	19.00	11.00	19.00	11.00

Auslagern bei verschiedenen Verfahren (Registersatz MI32). Bei `compress` *und* `lcc` *wurden keine Pseudoregister ausgelagert.*

	Opt	Pess	Mult	Prio A	Prio B
`compress`					
Ladebefehle	0.00	9.00	4.00	9.00	4.00
Auslagerkosten	0.00	4.59	3.67	4.59	3.67
Auslagerungen	0.00	2.00	2.00	2.00	2.00
Speicherbefehle	0.00	4.00	4.00	4.00	4.00
`doduc`					
Ladebefehle	1257.00	2705.00	3343.00	2594.00	2504.00
Auslagerkosten	10.25	10.80	10.66	10.78	10.67
Auslagerungen	893.00	7268.00	1578.00	6984.00	687.00
Speicherbefehle	741.00	1354.00	1742.00	1074.00	846.00
`dvips`					
Ladebefehle	47.00	271.00	287.00	299.00	245.00
Auslagerkosten	8.92	10.23	10.46	10.34	10.39
Auslagerungen	21.00	334.00	76.00	329.00	67.00
Speicherbefehle	30.00	161.00	172.00	169.00	155.00
`lcc`					
Ladebefehle	9.00	780.00	71.00	795.00	13.00
Auslagerkosten	5.44	6.20	5.99	6.31	4.99
Auslagerungen	2.00	28.00	14.00	27.00	4.00
Speicherbefehle	2.00	216.00	16.00	215.00	5.00
`sc`					
Ladebefehle	55.00	279.00	204.00	251.00	166.00
Auslagerkosten	10.39	10.94	10.99	10.94	10.93
Auslagerungen	17.00	230.00	63.00	221.00	45.00
Speicherbefehle	29.00	161.00	236.00	144.00	212.00

Auslagern bei verschiedenen Verfahren (Registersatz MI16).

gen für die Diskrepanz zwischen unseren und den in [BGG+89] angeführten Ergebnissen bieten sich an:

- Die Ergebnisse der relativ kleinen Testsuite in der Originalpublikation sind nicht repräsentativ. Dagegen spricht aber der Umstand, daß sich laut den Autoren auch beim Übersetzen des gesamten Compilers durch die angegebenen Heuristikfunktionen bessere Ergebnisse erzielen ließen.

- Einige Besonderheiten in der Struktur unseres Zwischencodes, namentlich die SSA-Form, führen dazu, daß die angegebenen Heuristiken nicht mehr „wirken“. In diesem Fall könnte wahrscheinlich durch andere Heuristikfunktionen eine Verbesserung erzielt werden.

	Opt	Pess	Mult	Prio A	Prio B
`compress`					
Ladebefehle	29.00	155.00	36.00	155.00	37.00
Auslagerkosten	8.85	10.37	8.21	10.37	8.74
Auslagerungen	9.00	37.00	8.00	37.00	8.00
Speicherbefehle	26.00	95.00	25.00	95.00	20.00
`doduc`					
Ladebefehle	1984.00	3565.00	3485.00	3354.00	2641.00
Auslagerkosten	10.60	11.05	10.66	11.03	10.67
Auslagerungen	1369.00	7742.00	1615.00	7592.00	752.00
Speicherbefehle	958.00	1579.00	1777.00	1308.00	914.00
`dvips`					
Ladebefehle	569.00	1650.00	699.00	1594.00	560.00
Auslagerkosten	10.72	11.07	10.83	11.12	10.77
Auslagerungen	155.00	1071.00	195.00	1025.00	157.00
Speicherbefehle	334.00	833.00	375.00	775.00	314.00
`lcc`					
Ladebefehle	666.00	4534.00	335.00	3739.00	82.00
Auslagerkosten	9.78	10.84	9.78	10.84	6.17
Auslagerungen	87.00	743.00	82.00	604.00	25.00
Speicherbefehle	309.00	1252.00	125.00	984.00	35.00
`sc`					
Ladebefehle	296.00	1467.00	715.00	1417.00	555.00
Auslagerkosten	10.91	11.56	11.14	11.50	11.14
Auslagerungen	84.00	517.00	130.00	483.00	83.00
Speicherbefehle	405.00	837.00	364.00	793.00	304.00

Auslagern bei verschiedenen Verfahren (Registersatz MI8).

- Es handelt sich in allen Fällen um Heuristiken; daher ist eine sichere Voraussage über die Ergebnisse nicht möglich; die von uns ermittelten Ergebnisse sind je nach den vorliegenden Verhältnissen genauso wahrscheinlich wie die in [BGG+89] angeführten.

Unsere Testergebnisse scheinen jedenfalls darauf hinzuweisen, daß Chaitins originale Heuristik bei großen Registersätzen generell die besten Ergebnisse liefert, während die Kombination aus den drei Alternativheuristiken ihre Vorteile ausspielen kann, wenn die Maschinenregister knapper werden. Die beste Kombination ergibt sich, wenn man alle vier Heuristiken gleichzeitig ausprobiert.

Die Anzahl der eingefügten Lade- und Speicherbefehle ist in Abbildung 6.7 auf Seite 80 graphisch dargestellt.

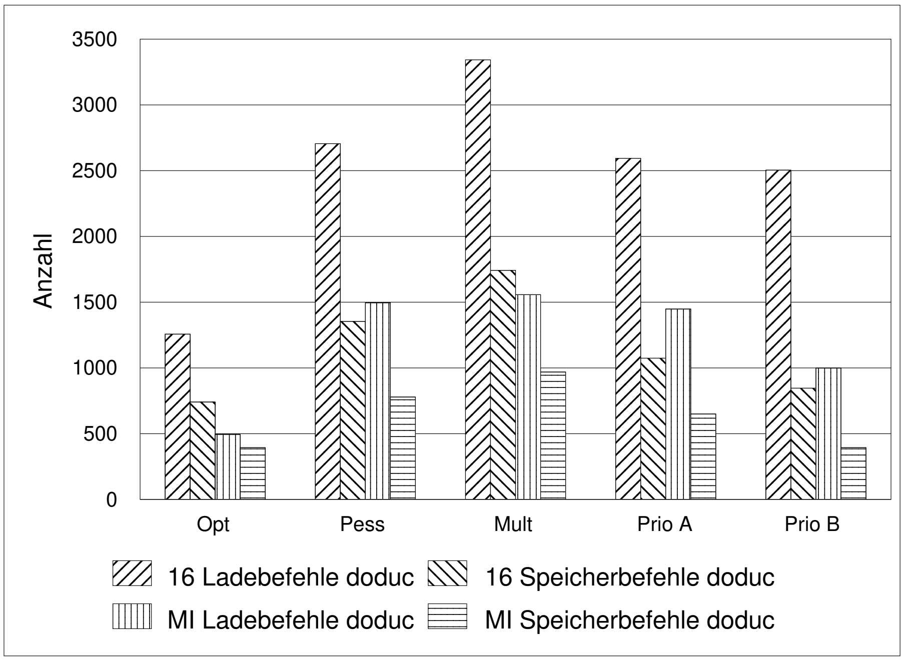


Abbildung 6.7: *Anzahl der eingefügten Lade- und Speicherbefehle bei* `doduc` *(Registersatz MI32)*

Es zeigt sich, daß zwischen dem besten und dem schlechtesten Verfahren große Unterschiede existieren: beispielsweise werden vom besten Verfahren (`Opt`) beim Registersatz `MI16` in Extremfall beim Auslagern nur 1% der Lade- und Speicherbefehle wie vom jeweils schlechtesten Verfahren eingefügt. Die selbe Aussage gilt für die Auslagerkosten — allerdings bedeutet ein großer Unterschied bei den eingefügten Lade– und Speicherbefehlen nicht gleichzeitig einen großen Unterschied bei den Auslagerkosten.

6.5.2 Zusammenlegen

Beim Zusammenlegen von Pseudoregistern werden Kopierbefehle entfernt. Die Gesamtanzahl der Kopierbefehle in unseren Testprogrammen sowie die Anzahl der beim Zusammenlegen entfernten Befehle sind in der folgenden Tabelle zusammengestellt (siehe auch Abbildung 6.10 auf Seite 83):

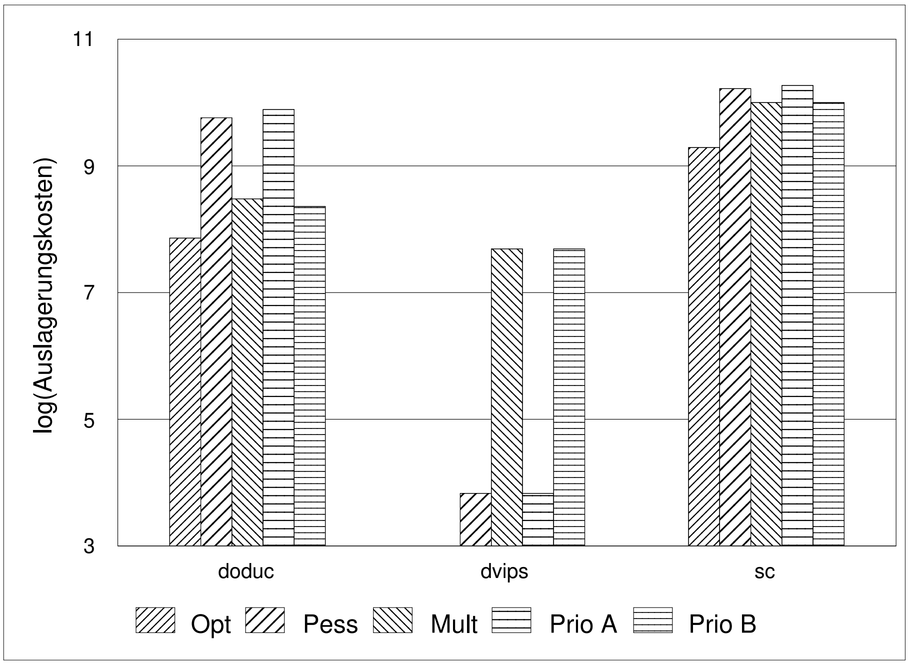


Abbildung 6.8: *Logarithmus der Auslagerungskosten bei* `doduc`, `dvips` *und* `sc` *(Registersatz MI32)*

Programm	Gesamt	Entfernt	Anteil
`compress`	4739	3555	75.0%
`doduc`	2525	2080	82.3%
`dvips`	30694	20811	67.8%
`lcc`	137132	73852	53.9%
`sc`	154635	100075	64.7%

6.5.3 Aufteilen

Wenn beim prioritätsgesteuerten Färben keine Farbe mehr für ein Pseudoregister zur Verfügung steht, versucht der Registerallokator, den Aktivierungsbereich in kleinere Bereiche aufzuteilen. Die Anzahl der notwendigen Aufteilungen bei den Registersätzen `MI32` und `MI16` ist in Abbildung 6.11 auf Seite 84 abgebildet.

6.6 Registerbedarf

Wieviele Register verwenden unsere Testprogramme und wie sparsam gehen die einzelnen Verfahren damit um? Um eine Antwort auf diese Fragen zu bekommen,

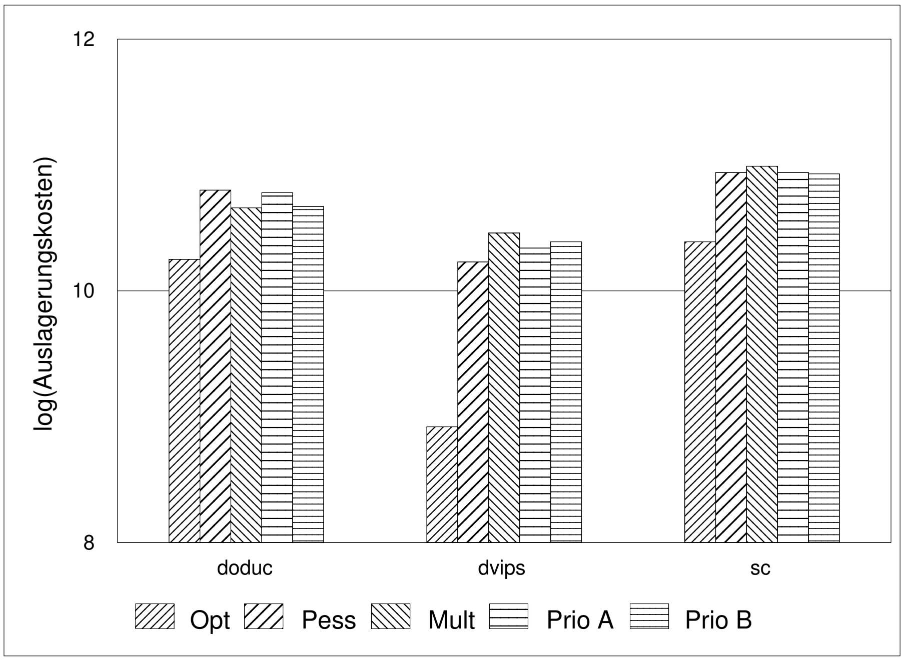


Abbildung 6.9: *Logarithmus der Auslagerungskosten bei* `doduc`, `dvips` *und* `sc` *(Registersatz MI16)*

haben wir die tatsächlich benötigte Anzahl an Maschinenregistern bei Verwendung des `MI32`-Registersatzes gemessen.

Hier zeigte sich ganz allgemein, daß die meisten Programme mit relativ wenigen Registern auskommen — der Mittelwert bei unserer Testsuite liegt bei 10 (oder exakt $\mu = 9.7$, $\sigma = 7.5$), wobei die Kurve bei allen Verfahren ein Maximum bei 6 Registern hat.

Bei den einzelnen Verfahren lassen sich bezüglich des Registerbedarfs zwei Gruppen bilden: das prioritätgesteuerte Färben und alle anderen Verfahren. Nur zwischen diesen beiden Gruppen zeigen sich merkliche Unterschiede im Registerbedarf. Daher ist in Abbildung 6.12 nur je eine Kurve für das prioritätsgesteuerte Verfahren und das optimistische Färben zu sehen. Es fällt auf, daß das Maximum bei 6 Registern beim prioritätsgesteuerten Verfahren noch wesentlich ausgeprägter als bei den anderen Verfahren ist.

Interessant ist auch, wie unterschiedlich der Registerbedarf bei verschiedenen Programmen verteilt ist. Wie in Abbildung 6.13 auf Seite 86 zu sehen ist, unterscheidet sich auch hier wieder `doduc` entscheidend von anderen Programmen.

Mit Hilfe zweier optionaler Verfahren läßt sich der Registerbedarf noch etwas senken, nämlich …

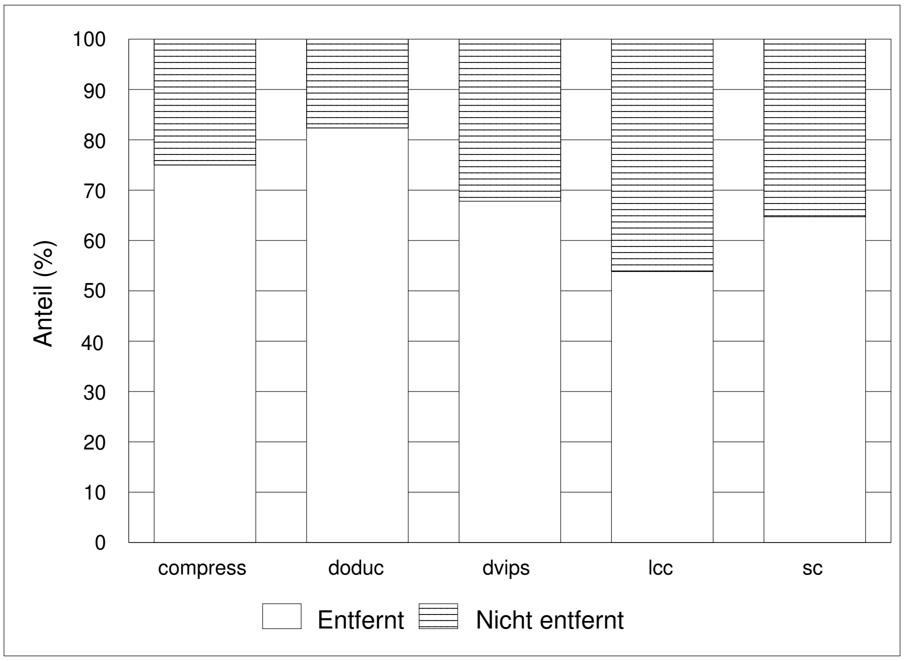


Abbildung 6.10: *Anteil der beim Zusammenlegen entfernten Kopierbefehle bei verschiedenen Programmen*

- ... einerseits durch ein besseres Registersuchverfahren ...
- ... und andererseits durch eine geeignete Vorsortierung der Pseudoregister vor dem Färben.

Die besten Ergebnisse haben wir mit dem Registersuchverfahren „das meistverwendete nehmen" (Option `-Rrs_maxused`) kombiniert mit „Pseudoregister nach der kleinsten Anzahl freier Register ordnen" (Option `-Rtlevel_minfree`) und dem Allokationsverfahren `Opt` erzielt. Durch die Kombination dieser beiden Verfahren lassen sich bei ungefähr 12% der Programme bis zu maximal drei Register einsparen.

Die Anzahl der eingesparten Register relativ zum Standardverfahren `Opt` sind in Abbildung 6.14 auf Seite 87 dargestellt.

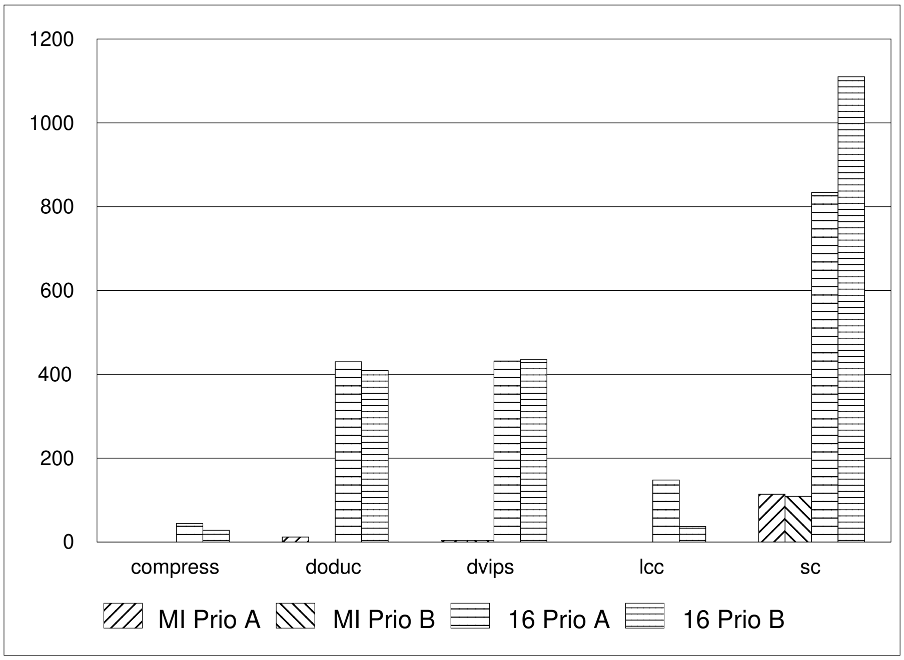


Abbildung 6.11: *Anzahl der aufgeteilten Pseudoregister bei zwei verschiedenen Registersätzen*

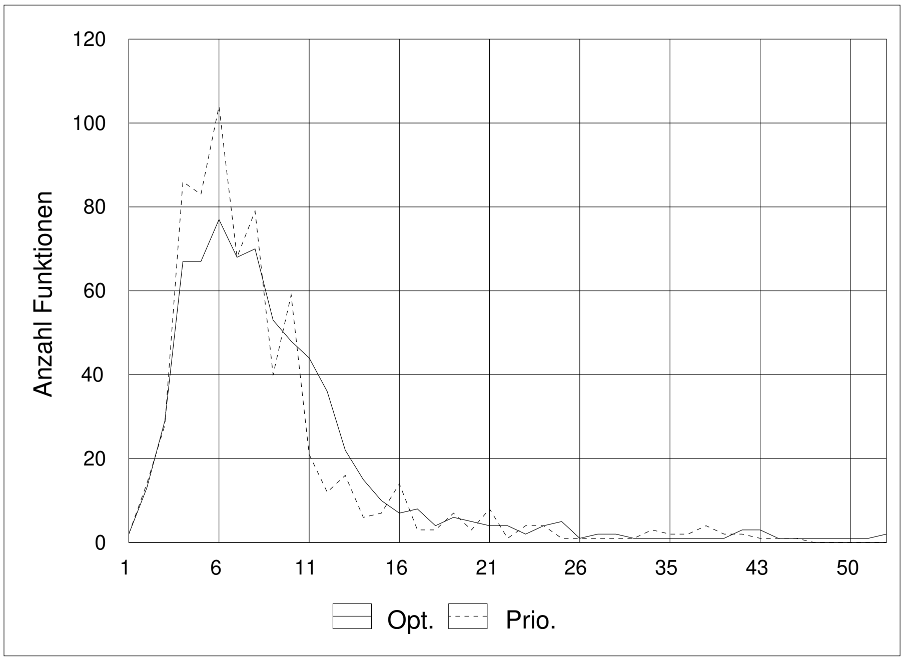


Abbildung 6.12: *Registerbedarf beim optimistischen und beim prioritätsgesteuerten Färben*

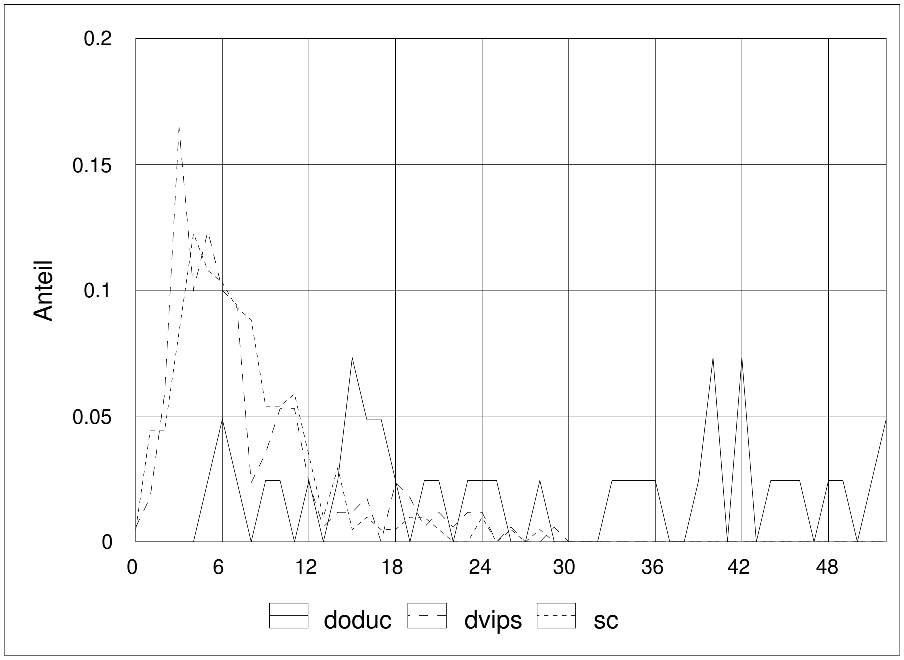


Abbildung 6.13: *Relativer Registerbedarf von verschiedenen Programmen*

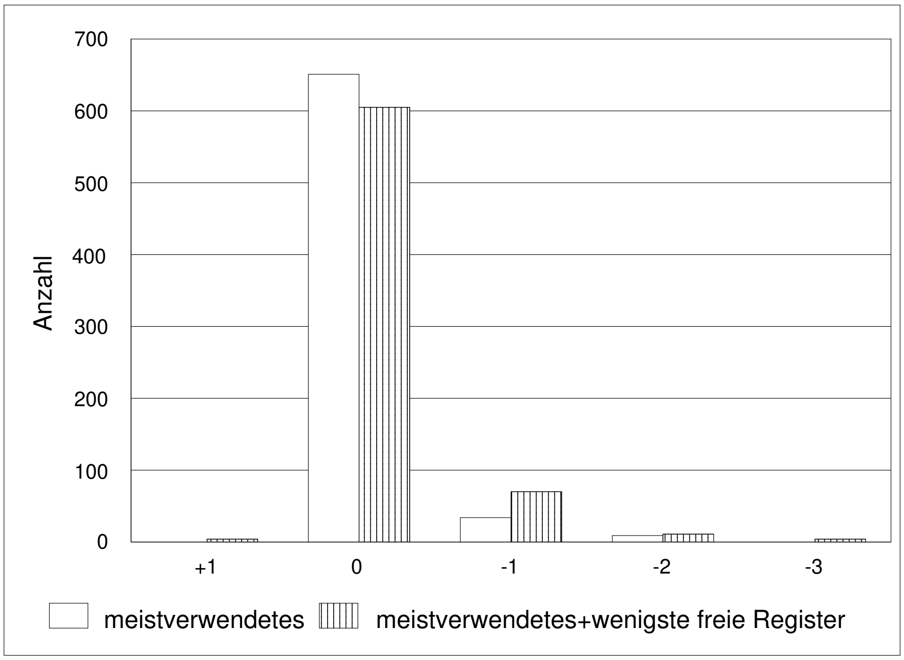


Abbildung 6.14: *Verringerung des Registerbedarfs durch verbesserte Registersuche und Ordnen der Pseudoregister*

Anhang A

Compileraufruf

A.1 Kommandozeilenparameter

Der Registerallokator kann mit einer Reihe von Kommandozeilenparametern gesteuert werden. Alle vom Registerallokator verstandenen Parameter beginnen mit `-R`. Die Konfigurationsmöglichkeiten sind in Abbildung A.1 zusammengefaßt.

Parameter	Default	Bedeutung
`alloc_simple`	⋆	Einfaches Färben
`alloc_multiple`		Färben mit mehreren Heuristikfunktionen
`alloc_priority`		Prioritätsgesteuertes Färben
`torder_none`		Keine Sortierfunktion für Pseudoregister
`torder_dumb`	⋆	Pseudoregister einfach ordnen
`torder_order`		Pseudoregister „intelligent" ordnen
`tlevel_none`	⋆	Keine Pseudoregisterbewertung
`tlevel_minfree`		Kleinste Anzahl an freien Registern
`tlevel_maxfree`		Größte Anzahl an freien Registern
`co_none`		Keine Färbefunktion
`co_dumb`		Färben nach Chaitin
`co_lazy`	⋆	Färben nach Briggs
`rs_any`	⋆	Erstbestes Register nehmen
`rs_maxused`		Register mit den meisten Verwendungen nehmen
`coalesce`	⋆	Zusammenlegen
`no_coalesce`		Nicht zusammenlegen
`test_options`		Optionen testen
`do_nothing`		Registerallokation nicht durchführen

Abbildung A.1: *Die Kommandozeilenparameter für den Registerallokator*

Nicht alle Parameter sind miteinander kombinierbar; die Kombinationsregeln sind folgende:

- `alloc_priority` ist nur mit den Optionen `coalesce`, `no_coalesce`, `rs_any`, `rs_maxused`, `tlevel_none`, `torder_none` und `co_none` kombinierbar.
- `alloc_simple` und `alloc_multiple` sind mit allen Optionen außer `tlevel_none`, `torder_none` und `co_none` kombinierbar.

Anhang B

Glossar

B.1 Englisch – Deutsch

basic block: Instruktionsblock, Block

color: färben

constrained node: gebundener Knoten

dominator: Dominator, Dominatorblock

graph coloring: Graphenfärben

graph: Graph

live range: aktiver Bereich

live: aktiv

register allocation: Registerbelegung, Registerallokation

spill: auslagern, sichern

spilled: speicherresident

temporary: Pseudoregister

unconstrained node: freier Knoten

B.2 Deutsch – Englisch

Dominator, Dominatorblock: dominator

Graph: graph

Graphenfärben: graph coloring

Instruktionsblock, Block: basic block

Pseudoregister: temporary

Registerbelegung: register allocation

aktiv: live

aktiver Bereich: live range

auslagern, sichern: spill

färben: color

freier Knoten: unconstrained node

gebundener Knoten: constrained node

speicherresident: spilled

Literaturverzeichnis

[ASU86] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1986.

Gehört zusammen mit [FL91] zu den besten Einführungen in das Thema „Compilerbau“. Der Schwerpunkt liegt zwar bei der syntaktischen und semantischen Analyse, trotzdem ist dieses Buch auch für jemanden, der mit Codegeneratoren zu tun hat, ein fast unverzichtbares Nachschlagewerk.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy und Linda Torczon. Coloring heuristics for register allocation. In *Programming Language Design and Implementation*, Seiten 275–284, 1989.

Führt eine kleine Änderung in Chaitins Färbeheuristik ein („optimistisches Färben“), die vor allem bei Registerknappheit wesentlich bessere Ergebnisse bringt.

[BCT92] Preston Briggs, Keith D. Cooper und Linda Torczon. Rematerialization. In *Programming Language Design and Implementation*, Seiten 311–321, 1992.

Beim „Rematerialisieren“ werden Werte von einfachen Ausdrücken beim Auslagern nicht abgespeichert und geladen, sondern vor jeder Verwendung neu berechnet. Das ist vor allem bei Konstanten und Adressausdrücken sinnvoll. Hier wird ein Verfahren vorgestellt, das auf der SSA-Analyse beruht.

[Bee93] Felix Beer. Integration globaler Optimierungstechniken in einen portablen ANSI C Compiler. Diplomarbeit, Technische Universität Wien, 1993.

Die Arbeit, die den Rest unseres Compilers genau behandelt. Eine Sammlung von modernen Optimierungsalgorithmen, Pseudocode sowie wertvollen Implementierungshinweisen.

[BEH91] David G. Bradlee, Susan J. Eggers und Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seiten 122–131, 1991.

Vergleich zwischen verschiedenen Strategien für das „Instruction Scheduling“. Ein neues, von den Autoren erfundenes Verfahren namens RASE wird vorgestellt, das Registerallokation und Scheduling integriert.

[BGG+89] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Yishay Mansour, Itai Nahshon und Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. In *Programming Language Design and Implementation*, Seiten 258–263, 1989.

Verbessert Chaitins Algorithmus durch die Verwendung dreier Heuristikfunktionen zur Minimierung der Auslagerungskosten. Außerdem wird ein Verfahren — „Bereinigen“ — vorgestellt, das beim Auslagern weniger Lade- und Speicherbefehle benötigt.

[Bö81] Gert Böhme. *Algebra*. Springer-Verlag, 1981.

Enthält unter anderem eine kurze Einführung in die Graphentheorie.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring*. Dissertation, Rice University, 1992.

Hier wird das Thema „Graphenfärben“ fundiert behandelt. Weder Grundlagen noch Implementationsdetails kommen zu kurz, und auch die Vergleichsdaten sind brauchbar. Weitere Themen: Rematerialisation, Aufteilen von Aktivierungsbereichen und Färben von Registerpaaren.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins und Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):45–57, 1981. Reprinted in [Sta90].

Die grundlegende Veröffentlichung zum Thema „Graphenfärben“. Noch immer eine der wichtigsten Arbeiten, da viele Grundbegriffe, die von anderen Autoren verwendet werden, nur hier erklärt werden.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman und F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Seiten 25–35, Austin, Texas, 1989.

Die grundlegende Arbeit zum Thema „Static Single Assignment“-Form. Siehe auch [CFR+91]

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman und F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oktober 1991.

Ein weiterer Artikel zum Thema „Static Single Assignment"-Form. Bringt einige Verbesserungen gegenüber [CFR+89]

[CH90] Fred C. Chow und John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, Oktober 1990.

Die Register werden in einem einzigen Durchlauf vergeben, eine Prioritätsfunktion gibt die Ordnung vor. Arbeitet auf Ebene der Instruktionsblöcke, kann aber Aktivierungsbereiche aufteilen und muß daher beim Auslagern nur in den Blöcken mit besonders vielen gleichzeitig aktiven Pseudoregistern auslagern. Der Artikel von Larus und Hilfinger [LH86] stellt eine empfehlenswerte Ergänzung zu dieser Arbeit dar.

[Cha82] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, Seiten 98–105, 1982.

Die Ergänzung zu [CAC+81]. Zwischen diesen zwei Veröffentlichungen wurde der Compiler auf einen neuen Prozessor portiert und der Registerallokator an entscheidenden Punkten verändert und verbessert. Vor allem beim Auslagern von Pseudoregistern gibt es große Unterschiede.

[Cho88] Fred C. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Seiten 85–94, 1988.

Ein Verfahren zur interprozeduralen Registerallokation, das auch teilweise mit den Effekten separater Übersetzung zurechtkommt.

[CK91] David Callahan und Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, Seiten 192–203, Toronto, Juni 1991.

Vor dem Färben wird die Kontrollstruktur des Programms mit Hilfe von Sharirs Intervallen erfaßt. Jedes dieser Intervalle („Tiles") wird separat gefärbt, am Schluß werden die Teile zusammengefügt.

[Deb87] Saumya Debray. Machine independent optimizations. comp.compilers / USENET, 1987.

[EABK92] M. Anton Ertl, Wolfgang Ambrosch, Felix Beer und Andreas Krall. Dependence-conscious global register allocation. Institut für Computersprachen, TU Wien, 1992.

Ein neuer Ansatz, der versucht, die gegensätzlichen Anforderungen von Registerallokation und „Code-Scheduling" zu integrieren.

[EK92] M. Anton Ertl und Andreas Krall. Instruction scheduling for complex pipelines. In *Proceedings of the 4th Conference on Compiler Construction*, Seiten 101–130. Springer Verlag, 1992.

Eine saubere und verständliche Darstellung des „Scheduling"-Problems, in dem eine Heuristik zum verbesserten Handling von komplexen Pipelines vorgestellt wird.

[FH90] Christopher W. Fraser und David R. Hanson. A code generation interface for ANSI C. Research Report CS-TR-270-90, AT&T Bell Laboratories, 1990. Revision of September 1991.

Siehe [FH91a].

[FH91a] Christopher W. Fraser und David R. Hanson. A code generation interface for ANSI C. *Software—Practice and Experience*, 21(9):963–988, September 1991.

Beschreibt die Schnittstelle zwischen Analyse– und Synthesephase beim `lcc`. Datenstrukturen und Funktionen werden anhand von Beispielen vorgestellt. Hier findet man die wichtigsten Informationen, wenn Veränderungen am `lcc` oder an unserem Compiler vorgenommen werden sollen.

[FH91b] Christopher W. Fraser und David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, Oktober 1991.

Ein allgemeiner Überblick über den Compiler `lcc`.

[FH92] Christopher W. Fraser und David R. Hanson. Simple register spilling in a retargetable compiler. *Software—Practice and Experience*, 22(1):85–99, Januar 1992.

Ein einfaches und effizientes Verfahren zur Registerallokation, das nichts mit Graphenfärben zu tun hat. Moderne Prozessoren haben nach Meinung der Autoren genug Register, so daß in den meisten Fällen ein sehr einfacher Registerallokator fast gleich gute Ergebnisse liefert wie ein sehr komplizierter.

[FL91] Charles N. Fischer und Richard J. LeBlanc. *Crafting a Compiler with C*. Benjamin Cummings, 1991.

Gehört zusammen mit [ASU86] zu den besten Einführungen in das Thema „Compilerbau". Zwar liegt auch hier der Schwerpunkt auf der syntaktischen und semantischen Analyse, allerdings werden auch Themen wie zum Beispiel „Datenflußanalyse" anhand brauchbarer Beispiel erklärt.

[GH88] James R. Goodman und Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *International Conference on Supercomputing*, Seiten 442–452, 1988.

Zwei Verfahren zur Integration von Registerallokation und Scheduling (die beide nicht mit Graphenfärben arbeiten) werden hier vorgestellt.

[GM86] Phillip B. Gibbons und Steve S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Seiten 11–16, 1986.

Eine heuristischer Algorithmus für das „Instruction Scheduling“ dessen Aufwand in den meisten praktischen Fällen nur linear mit der Problemgröße steigt.

[GSS89] Rajiv Gupta, Mary Lou Soffa und Tim Steele. Register allocation via clique separators. In *Programming Language Design and Implementation*, Seiten 264–274, 1989.

Versucht das Färben von großen Interferenzgraphen durch Aufteilen diese Graphen in kleinere Einheiten und separates Färben zu lösen. Verspricht einen geringeren Gesamtaufwand, aber keinen besseren Maschinencode.

[HGAM92] Laurie J. Hendren, Guang R. Gao, Erik R. Altman und Chandrika Mukerij. A register allocation framework based on hierarchical cyclic interval graphs. In *Proceedings of the 4th Conference on Compiler Construction*, Seiten 176–191. Springer Verlag, 1992.

Ein Verfahren, das zwei neue Techniken vorstellt: Einerseits ein Algorithmus, „fat cover“ genannt, der den maximalen Registerbedarf anhand der „Breite“ eines Interferenzgraphen abschätzt und an den „breitesten“ Stellen zu färben beginnt. Andererseits ein Verfahren, welches das Auslagern in den Hauptspeicher durch Verschieben in eventuell ungenutzte Register zu verhindern versucht.

[HP90] John L. Hennessy und David A. Patterson. *Computer Architecture. A Quantitative Approach.* Morgan Kaufman Publishers, 1990.

„Das“ Standardwerk zum Thema Computerarchitektur. Die Autoren, die selbst an der Entwicklung der RISC-Prozessoren maßgeblich beteiligt waren, behandeln die verschiedensten Architekturen, geben Richtlinien aus der Praxis und zeigen, wie sich die wichtigen Parameter von Computerarchitekturen auch quantitativ vergleichen lassen.

[JM86] Mark Scott Johnson und Terence C. Miller. Effectiveness of a machine-level, global optimizer. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Seiten 99–107, 1986.

Ein Überblick über den Codegenerator der Compiler für die HP-PA-RISC-Prozessoren.

[KPR91] C. W. Keßler, W. J. Paul und T. Rauber. A randomized heuristic approach to register allocation. In *Proceedings of the 1991 Conference on Programming Language Implementation and Logic Programming*, Seiten 195–206, 1991.

Ein Beispiel für einen Registerallokationsalgorithmus, der ohne Graphenfärben auskommt.

[KZM85] Hans Kaiser, Gisela Zeilinger und Rainer Mlitz. *Algebra für Informatiker*. Springer-Verlag, 1985.

Enthält unter anderem eine kurze Einführung in die Graphentheorie.

[Lar91] James R. Larus. SPIM S20: A MIPS R2000 simulator. Technischer Bericht, University of Wisconsin-Madison, 1991. Revision 6 corresponding to SPIM Version 4.2.

Dokumentation zum MIPS-Simulator SPIM, den wir auf den NeXTStations zur Simulation unserer Zielmaschinen einsetzten.

[LH86] James R. Larus und Paul N. Hilfinger. Register allocation in the SPUR lisp compiler. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Seiten 255–263, 1986.

Eine andere Implementierung von Chows prioritätsgesteuertem Färbealgorithmus. Sollte gemeinsam mit dem Artikel von Chow [CH90] gelesen werden.

[Nic90] Brian R. Nickerson. Graph coloring register allocation for processors with multi-register operands. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, Seiten 40–52, 1990.

Ein interessanter Algorithmus, der das Färben von Operanden, die mehrere Register auf einmal benötigen, ermöglichen soll

[SH89] Peter A. Steenkiste und John L. Hennessy. A simple interprocedural register allocation algorithm and its effectiveness for LISP. *ACM Transactions on Programming Languages and Systems*, 11(1):1–32, Januar 1989.

LISP eignet sich besonders für die interprozedurale Registerallokation, weil im compilierten Code sehr viele Unterprogrammaufrufe vorkommen. Dieser Artikel untersucht diese Situation näher und stellt einen Algorithmus zur interprozeduralen Registerallokation vor.

[Sha80] M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5:141–153, 1980.

Ein Verfahren zur Intervallanalyse, mit dem sich die Kontrollstruktur von Programmen sehr gut darstellen läßt.

[SKAH91] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind und David Hunnicutt. Efficient dag construction and heuristic calculation for instruction scheduling. In *MICRO-24, 24^{th} Annual Intl. Symp. on Microarchitecture*, Seiten 93–102, 1991.

Ein Überblick über verschiedene Heuristiken für das „Instruction-Scheduling".

[Sta90] William Stallings, Hrsg. *Reduced Instruction Set Computers.* IEEE Computer Society Press, second edition , 1990.

[Wal86] David W. Wall. Global register allocation at link time. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Seiten 264–275, 1986.

Durch die Einbindung des Linkers in den Registerallokationsvorgang arbeitet der von Wall vorgestellte interprozedurale Registerallokator sehr schnell und erzielt dabei ausgezeichnete Allokationsergebnisse.

[Wal92] David Wall. Experience with a software-defined machine architecture. *ACM Transactions on Programming Languages and Systems*, 14(3):299–338, Juli 1992.

Stellt die experimentelle RISC-Workstation Titan und die Assemblersprache Mahler vor, auf die ein experimenteller, integrierter Compiler/Linker aufbaut. Enhält auch interessante Vergleiche zum Thema „interprozedurale Registerallokation". Siehe auch [Wal86].

[War90] Henry S. Warren, Jr. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):85–92, Januar 1990.

Untersucht das Problem des „Instruction Scheduling" unter den speziellen Voraussetzungen der IBM RS/6000-Prozessoren.