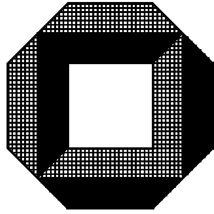


Universität Karlsruhe (TH)
Institut für
Programmstrukturen
und Datenorganisation
Lehrstuhl Professor Goos

RAP — Ein Registerzuteiler für CGGG

Sebastian Hack

Studienarbeit

Betreuer
Dipl. Inform. Boris Boesler

Verantwortlicher
Prof. Dr. Gerhard Goos

Januar 2003

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

Inhaltsverzeichnis

1 Einleitung **5**
1.1 Rahmen dieser Arbeit 5
1.2 Verfahren der Registerzuteilung 6
1.2.1 Registerzuteilung durch Graphfärbung 6
1.2.2 Weitere Verfahren 8

2 Vorarbeiten **9**
2.1 Terminologie und Notation 9
2.2 Schnittstelle zu CGGG 9
2.3 Lebensdaueranalyse 10
2.4 Der PDG 11
2.4.1 Definitionen 12
2.4.2 Der PDG in RAP 13
2.4.3 Anpassung an FIRM und CGGG 14
2.4.4 Beispiel 14

3 RAP **17**
3.1 Überblick 17
3.2 Die einzelnen Phasen im Detail 18
3.2.1 Aufbau des IFG — build_ifg 19
3.2.2 Das Hinzufügen der Kind-IFG — add_subregion_conflicts 19
3.2.3 Die Eliminierung nutzloser Kopien — coalesce 22
3.2.4 Das Berechnen der Spill-Kosten — calc_spill_costs 22
3.2.5 Das Abbauen des Graphen — simplify 23
3.2.6 Das Färben des Graphen — color 25
3.2.7 Das Hinzufügen von Spill-Code — insert_spill_code 25

4 Unterschiede, Erweiterungen und Ausblick **31**
4.1 Prinzipielle Unterschiede 31
4.2 Erweiterungen 32
4.2.1 Grundblöcke als kleinstes Allokationsgebiet 32
4.2.2 Verschiedene Registerklassen 32
4.3 Ausblick 34

A Messungen **37**
A.1 Erzeugter Spill Code 37
A.2 Anteil von RAP an der Übersetzerlaufzeit 38

B Notation **41**

Kapitel 1

Einleitung

Speicher ist in Rechnern nicht einheitlich organisiert. Er ist in Hierarchien eingeteilt, die sich vor allem durch Zugriffsgeschwindigkeit und Speicherkapazität unterscheiden. Je mehr Daten ein Speicher fassen kann, desto langsamer ist der Zugriff darauf. Am einen Ende der Hierarchie stehen Massenspeicher wie Festplatten oder CD Laufwerke, am anderen Ende stehen die Register des Mikroprozessors.

Die Entwicklung der letzen Jahre hat gezeigt, dass der Abstand zwischen Prozessortakt, der auch die Zugriffsgeschwindigkeit auf die Register bestimmt, und Hauptspeichertakt immer größer wird. Man versucht, die langen Latenzen durch immer mehr Cache-Hierarchien wett zu machen. Deshalb ist es von zentraler Bedeutung, so viele Daten wie möglich in den Prozessorregistern zu halten[1].

Während des Befehlsauswahlprozesses wird normalerweise immer von unendlich vielen vorhandenen Registern (auch virtuelle Register, Pseudo-Register, oder temporäre Variable genannt) ausgegangen. Ein anderes Vorgehen würde zu viel Komplexität erzeugen und sich nur schwer in die bekannten Verfahren einbinden lassen. Man überlässt die Abbildung der virtuellen Register auf die wirklich in der Zeilarchitektur vorhandenen, dem Registerzuteiler. Nachdem der Befehlsauswahlprozess beendet ist, analysiert der Registerzuteiler die Beziehungen zwischen den virtuellen Registern und versucht eine möglichst effiziente Zuteilung der Register zu finden.

1.1 Rahmen dieser Arbeit

Es wurde versucht, ein modernes Registerzuteilungsverfahren in den am IPD Goos entwickelten Codegeneratorgenerator CGGG [1] zu integrieren. CGGG basiert auf der ebenfalls am IPD entwickelten Zwischenrepräsentation FIRM [10], die Datenabhängigkeiten und Kontrollfluss des Programmes in SSA-Form in Graphen repräsentiert. CGGG überdeckt die Graphen mit Hilfe von Termersetzungssystemen (BURS) und wählt mit Hilfe einer A^*-Suche die entsprechende Befehlssequenz aus. Die Registerzuteilung wird nach der Befehlsauswahl auf die von CGGG gefundene Befehlssequenz angewendet.

[1] Insofern man an einer geschwindigkeitsoptimierten Übersetzung interessiert ist

Als Verfahren wurde RAP (Register Allocation over the PDG) [8] gewählt. Es ist eine Erweiterung der gängigen Graphfärbungsverfahren, die hierarchisch arbeitet. Die grundlegenden Merkmale wurden alle implementiert, an einigen Stellen modifiziert und für den praktischen Gebrauch erweitert.

1.2 Verfahren der Registerzuteilung

Das momentan in Übersetzern am meisten verbreitete Verfahren ist die *Registerzuteilung durch Graphfärbung*. Es geht zurück auf die Forschungen von Gregory Chaitin [4], welche 1992 von Preston Briggs [3] massgeblich erweitert wurden. Ich beschreibe hier nur schematisch die Funktionsweise der graphfärbenden Registerzuteilung (GRZ) und werde auf die Details bei der Beschreibung des implementierten Algorithmus näher eingehen.

1.2.1 Registerzuteilung durch Graphfärbung

Die Registerzuteilung durch Graphfärbung beruht auf einer Reduktion einer maßgeblichen Eigenschaft von berechneten Werten eines Programms auf das Problem des Färbens eines Graphen mit k Farben (wobei k die Anzahl der vorhandenen Register ist): Die *Interferenz*. Man sagt, zwei Werte interferieren, wenn sie nicht im selben Register gehalten werden können. Der Registerzuteiler berechnet infolgedessen für alle temporären Variablen die Interferenz zu allen anderen temporären Variablen. Gehalten werden diese Informationen in einem Graphen, der die “interferiert mit”-Beziehung ausdrückt. Für jede temporäre Variable gibt es eine Ecke im *Interferenzgraphen*. Interferieren zwei temporäre Variablen, dann sind die zugehörigen Ecken durch eine Kante verbunden.

Wie kann man bestimmen, ob zwei Werte im selben Register gehalten werden können? Dazu bedarf es der *Lebensdaueranalyse*[2], die feststellt, welcher Wert wann lebendig ist. Was bedeutet jedoch *lebendig*?

Zu jedem Wert gibt es mindestens zwei Stellen in der erzeugten Befehlssequenz, an denen der Wert entweder ausgewertet (er ist das Resultat einer Berechnung), oder genutzt (aus ihm wird ein anderer Wert berechnet) wird. Zwischen einer solchen Auswertung und den darauffolgenden Nutzungen ist der Wert lebendig. Würde man einem anderen Wert, der an diesen Stellen auch lebendig ist, dasselbe Register zuteilen, so würde man einen der beiden Werte zerstören. Also muss der Registerzuteiler zu jeder Stelle des Codes wissen, welche Werte gerade lebendig sind. Die ensprechenden Ecken dieser Werte muss er dann im Interferenzgraphen durch eine Kante verbinden. In Abbildung 1.1 ist ein Programmfragment zu sehen, bei dem neben jeder Zeile die Menge der an dieser Stelle lebendigen Werte angegeben ist. Dangeben ist der Interferenzgraph für dieses Programmfragment zu sehen.

[2]engl.: Liveness Analysis

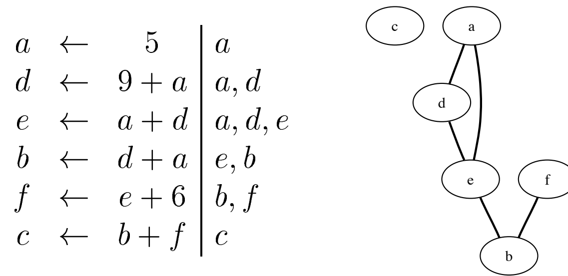


Abbildung 1.1: Lebensdauer

Nach dem Aufbauen des Interferenzgraphen muss dieser noch gefärbt werden. Jede Farbe entspricht einem physisch vorhandenen Register. Zwei Ecken, die durch eine Kante verbunden sind, müssen unterschiedliche Farben haben, so wird gewährleistet, dass zwei temopräre Variablen, die gleichzeitig lebendig sind, nicht dasselbe Register zugeteilt bekommen.

Da das Färben eines Graphen mit k Farben NP-vollständig ist, hängt die Effizienz hinsichtlich einer guten Registerzuteilung von einer geeigneten Heuristik für das Färben ab. Prinzipiell funktioniert das Färben so, dass dem Graphen nach und nach Ecken entnommen, auf einen Stapel gelegt und dann wieder eingefügt werden. Beim Einfügen wird darauf geachtet, dass die eingefügte Ecke eine andere Farbe erhält als ihre Nachbar-Ecken. Dies ist die erweiterte Heuristik von Briggs, die weitaus weniger pessimistisch ist, als die Heuristik von Chaitin, der annimmt, dass die Farben aller Nachbarn einer Ecke alle unterschiedlich sind, was, wie die Praxis zeigt, nicht immer der Fall ist.

Es kann natürlich sein, dass eine Ecke nicht eingefärbt werden kann. Dies ist der Fall, wenn die Ecke einen Grad hat, der größer als k ist und die Nachbarn schon entsprechend "bunt" eingefärbt sind, so dass für jene Ecke keine Farbe übrig ist. Um trotzdem ein funktionierendes Programm zu erhalten, muss der Registerzuteiler versuchen Interferenzen zu eliminieren, um so die Grade der betroffenen Ecken zu senken. Dies geschieht durch das Auslagern von Werten[3]. Der Registerzuteiler entscheidet, ob ein Wert nicht mehr in einem Register gehalten wird, sondern in den Speicher ausgelagert wird (z. B. ein Platz in der Funktionsaufrufschachtel). Je nach Prozessorarchitektur interferiert der Wert dann nur noch mit den Werten, die zum Zeitpunkt des Ladens oder Rückschreibens in den Speicher lebendig sind. Insbesondere interferieren zwei ausgelagerte Werte niemals, da sie beide nur direkt an ihrer Benutzung und Auswertung lebendig sind. Dieses Verfahren wird solange wiederholt, bis der Graph korrekt gefärbt ist.

Moderne Architekturen (insbesondere Load/Store-Architekturen) akzeptieren in ihren Maschinenbefehlen aber meistens nur Register als Operanden. Das heisst, dass das Auslagern von Werten in den Speicher neue Registerbelegungen induziert, was dazu führt, dass nach jedem misslungenen Färben der Interferenzgraph neu aufgebaut werden muss.

[3] engl.: spilling

1.2.2 Weitere Verfahren

Registerzuteilung durch Graphfärbung ist ein vergleichsweise altes Verfahren. Die erste Arbeit von Chaitin stammt aus dem Jahre 1981. In der folgenden Zeit wurden mehrere Verbesserungen und Varianten dieses Verfahrens veröffentlicht. Im Kern beruhen diese aber alle auf dem Färben von Interferenzgraphen.

Von den Verfahren, die keine Graphfärbung einsetzen, ist vor allem das *Linear-Scan*-Verfahren [9] zu erwähnen. Es bietet ähnlich gute Ergebnisse hinsichtlich der Güte der Registerzuteilung bei weitaus geringerer Laufzeit. Es wird vollständig auf das Aufbauen und Färben von Graphen verzichtet. Die Zuteiling wird mit der Lebendigkeitsanalyse kombiniert und in einem Durchgang erledigt. Dies macht das Verfahren ideal für *Just-In-Time*-Compiler, wie sie mit der starken Verbreitung von Java aufgekommen sind. Sie übersetzen das Programm während dessen Laufzeit ganz oder teilweise von Java-Bytecode in nativen Maschinencode, um die langsame Interpretation der Java-VM zu umgehen. Ein aufwendiges Verfahren wie z. B. Rap wäre hier viel zu zeitaufwendig.

Trivialerweise gibt es noch die sog. Registerzuteilung *On-The-Fly*, die einfach solange Register vergibt, bis keine mehr vorhanden sind, und dann mit dem Spilling beginnt.

Kapitel 2

Vorarbeiten

In diesem Kapitel werden die Vorarbeiten, die der Registerzuteiler vor der eigentlichen Zuteilung erledigen muss, betrachtet. Sie machen einen nicht unerheblichen Teil der Implementierung aus. Insbesondere sei hier der Aufbau des Programmabhängigkeitsgraphen[1] (im weiteren PDG genannt) und die Lebensdaueranalyse erwähnt.

2.1 Terminologie und Notation

Im folgenden verwende ich die Bezeichnungen „temporäre Variable“, „Wert“ und „Pseudo-Register“ synonym. Gemeint ist ein Pseudo-Register, wie es von CGGG nach der Selektionsphase erzeugt wird. CGGG nummeriert diese Pseudo-Register fortlaufend mit der PSR (Pseudo-Register Nummer).

2.2 Schnittstelle zu CGGG

Die Registerzuteilung geschieht pro Prozedur. Dazu übergibt CGGG dem Registerzuteiler einen Zeiger auf den letzten Suchknoten, der von ihm ermittelten Ersetzungssequenz. Anhand dieses Zeigers kann der Registerzuteiler alle selektierten Instruktionen ablaufen. Entscheidend sind die Benutzungen und Definitionen von Werten (siehe Abbildung 2.1) und deren Reihenfolge, da sie die Lebendigkeit der Werte determinieren. Der Registerzuteiler merkt sich jedes Auftreten eines Wertes und markiert es als *use* oder *def*. Folgende Informationen sind für den Registerzuteiler von Interesse:

- Die PSR (Pseudo-Register-Nummer, auch temporäre Variable gennannt) mit ihrem Nutzungsmodus, sprich: Benutzung *(use)* oder Definition *(def)*. Siehe auch Abbildung 2.1 für ein Beispiel der *use-def-chains*.

- Der Suchknoten, der zur Ersetzungsregel gehört. Er wird beim Einfügen von Spill-/Reload-Instruktionen vor oder nach der Regel wichtig.

[1] engl.: Program Dependence Graph

- Der Graphknoten, da in ihm letztlich das zugeteilte Register abgelegt wird
- Der Graphknoten der Operation, zu der der Operand gehört. Dies ist wichtig, wenn einer der Operanden ausgelagert wird. Dann muss das Einlagern an die entsprechende Stelle gesetzt werden.
- Anweisungen an den Registerzuteiler, die Informationen über eine Registerbelegung für bestimmte Ersetzungen darstellen. Diese können zum Beispiel daher rühren, dass ein bestimmter Maschinenbefehl, der einer Ersetzung entspricht, Register überschreibt oder Operanden in bestimmten Registern erwartet. *Diese Schnitstelle von* CGGG *zum Registerzuteiler ist im Moment noch unimplementiert.*

Letztlich merkt sich der Registerzuteiler die obigen Daten in einer eigenen Datenstruktur, da sich gerade die PSR durch Einfügen von Spill-Code ändern kann.

$$\begin{array}{rcc} a & \leftarrow & 5 \\ d & \leftarrow & 9+a \\ e & \leftarrow & a+d \\ b & \leftarrow & d+a \\ f & \leftarrow & e+6 \\ c & \leftarrow & b+f \end{array} \quad \longrightarrow \quad \begin{array}{c} \text{def}\, a \\ \text{use}\, a \\ \text{def}\, d \\ \text{use}\, a \\ \text{use}\, d \\ \cdots \end{array}$$

Abbildung 2.1: use-def-chains

2.3 Lebensdaueranalyse

Die Lebensdaueranalyse ist essentiell für den Registerzuteiler. Sie bestimmt die Lebensdauer der berechneten Werte, anhand derer der Registerzuteiler die Interferenzrelation bestimmt und die Registerzuteilung vornimmt.

Wie schon erwähnt, wird bei der Befehlsauswahl vereinfachend davon ausgegangen, dass die Zielmaschine unendlich viele Register hat. Die berechneten Werte werden einfach fortlaufend nummeriert. Gerade bei SSA-basierten Zwischensprachen werden sehr viele verschiedene „Pseudo-Register“ vorhanden sein, da ja jeder Wert nur eine Definition hat, und somit sehr viele neue Werte enstehen.

Jeder Wert x hat in einem Programm mindestens eine Definition, bei der er berechnet wird, und Benutzungen, an denen er ausgewertet wird. Die Lebendigkeit der Werte wird durch die Zugehörigkeit zu folgenden vier Mengen charakterisiert, die pro Grundblock B existieren:

1. $in(B)$: Werte, die beim Eintritt in B lebendig sind.
2. $out(B)$: Werte, die beim Verlassen von B lebendig sind.

3. $def(B)$: Werte, die in B vor ihrer Benutzung definiert werden.
4. $use(B)$: Werte, die in B zuerst benutzt, bevor sie evtl. definiert werden.

Es gelten folgende Eigenschaften, die für die Lebensdaueranalyse wichtig sind:

$$in(B) = use(B) \cup (out(B) \setminus def(B)) \tag{2.1}$$

$$out(B) = \bigcup_{S \in succ(B)} in(S) \tag{2.2}$$

Die Mengen $def(B)$ und $use(B)$ lassen sich direkt mit Algorithmus 1 aus den Instruktionen im Block B berechnen. $target_I$ bezeichnet die Menge der temporären Variablen, die von einer Instruktion I beschrieben werden, $source_I$ die Menge der temporären Variablen, die von I gelesen werden.

```
calculate_use_def (B)
1  def(B) := ∅
2  use(B) := ∅
3  for i := I_n ... I_1 do        // I sind die Instruktionen in B
4     def(B) := def(B) ∪ target_i
5     use(B) := (use(B) ∪ source_i) ∩ target_i
6  end
7  globals := globals ∪ use(B)
```

Algorithmus 1: Berechnung der Mengen def_B und use_B

Die Berechnung von $in(B)$ und $out(B)$ ist ein wenig aufwendiger, da sie nach Gleichung 2.1 und Gleichung 2.2 voneinander abhängen. Ein Algorithmus zur Lösung der Gleichungen 2.1 und 2.2 findet sich in dem Buch von Morgan [7] (Seite 134). Dieser wurde auch in unserer Implementierung von RAP verwendet.

2.4 Der PDG

Der RAP-Algorithmus erledigt die Registerzuteilung hierarchisch. Da die Autoren des Algorithmus den PDG als geeignete Struktur, besonders im Hinblick auf das Zusammenarbeiten des Registerzuteilers mit anderen Optimierungsstufen eines Übersetzers erachten, wird die Registerzuteilung auf der vom PDG induzierten Relation im Kontrollflussgraphen (CFG) vorgenommen.

Die traditionellen Registerzuteiler, die Graphfärbung einsetzen, bilden den Interferenzgraphen (IFG) auf der ganzen Routine[2]. RAP hingegen bildet Unter-IFGs, die dann zu dem die Routine repräsentierenden IFG zusammengefügt werden. Zu den Vor- und Nachteilen dieser Herangehensweise, siehe Kapitel 3.

Zunächst möchte ich die grundlegenden Definitionen für den PDG geben und dann näher auf die Bedeutung des PDG für RAP eingehen.

[2] Ich verwende hier Routine für den Bereich Code, für den die Registerzuteilung erfolgen soll

2.4.1 Definitionen

Eine gute Beschreibung des PDG findet sich in [5]. Eigentlich besteht der PDG aus zwei Teilgraphen, dem CDS (Control Dependece Subgraph) und dem DDS (Data Dependence Subgraph). Wichtig für RAP ist nur der CDS, im folgenden ist auch der CDS gemeint, wenn vom PDG die Rede ist. Der CDS stellt die Kontrollabhängigkeiten zwischen den Grundblöcken dar. Wichtig für das Erzeugen des CDS ist die *Nachdominanzbeziehung* zwischen den Ecken des Kontrollflussgraphen (im folgenden CFG).

Definition 1 (Nachdominanz) *Sei $G = (E, K)$ ein CFG.* **start** *und* **stop** *seien die Ecken in E, für die gilt:*

$$|pred(\mathbf{start})| = |succ(\mathbf{stop})| = 0$$

Eine Ecke $e \in E$ wird von einer anderen Ecke $f \in E$ nachdominiert*, wenn jeder Weg von e zu* **stop** *(ohne e) f enthält.*

Bemerkung 1 *Insbesondere gilt für alle Ecken e: e wird nicht durch e nachdominiert.*

Die Nachdominanzrelation eines CFG ist äquivalent zur Dominanzrelation des dualen CFGs, dem RCFG[3]. Zur Berechnung der Dominanzrelation existiert ein schneller Algorithmus von Lengauer und Tarjan [6], der in $O(m \log n)$ läuft[4], wobei m die Anzahl der Kanten und n die Anzahl der Ecken im CFG ist. Wichtig bei der Berechnung der Dominanz (und natürlich auch der Nachdominanz) ist nicht, die Dominanzrelation für jeden Knoten zu berechnen, sondern nur den *Baum* der unmittelbaren Dominatoren aufzubauen. Durch Laufen über diesen Baum kann man alle Dominanzrelationen erhalten. Im folgenden bezeichne ich den unmittelbaren Nachdominator einer Ecke e mit $ipdom(e)$[5].

Mit Hilfe der Nachdominanz lässt sich nun die Kontrollabhängigkeitsbeziehung von Ecken in einem CFG definieren:

Definition 2 (Kontrollabhängigkeit) *Sei $G = (E, K)$ ein CFG und $e, f \in E$. f ist von e* kontrollabhängig*, wenn und nur wenn:*

1. *es einen Weg w von e zu f gibt, wobei jede Ecke $z \in w \wedge z \neq e \wedge z \neq f$ von f nachdominiert wird,* und
2. *e von f nicht nachdominiert wird.*

Der RAP Algorithmus verwendet eine leicht abgeänderte Variante des PDG. In Schleifen kommt es vor, dass eine Ecke e von sich selbst kontrollabhängig ist. Anstatt eine Kante von e zu sich selbst zu ziehen, wird die Ecke nur markiert. Um die Kontrollabhängigkeiten zu berechnen wird der CFG zuerst (noch vor der Berechnung der Nachdominanz) mit

[3]engl.: Reverse Control Flow Graph

[4]Es existiert noch eine andere Version, die in $O(m\alpha(m, n))$ läuft, wobei $\alpha(m, n)$ das inverse der Ackermann-Funktion ist.

[5]*ipdon*: immediate post dominator

einer weiteren Ecke **entry** erweitert und die Kanten (**entry**, **start**), (**entry**, **stop**) hinzugefügt (siehe hierzu auch [5]). Algorithmus 2 berechnet die Menge $cd(B)$ der von B kontrollabhängigenen Ecken in einem CFG.

```
compute_control_dependence (g : (E, K))
1  for B ∈ E do
2    for P ∈ succ(B) do
3      X := P
4      while X ≠ ipdom(X) do
5        if X = B then
6          Markiere X als Schleifenkopf
7        else
8          cd(X) := cd(X) ∪ B
9        end
10       X := ipdom(X)
11     end
12   end
13 end
```

Algorithmus 2: Berechnung der Menge $cd(B)$ jede Ecke B eines CFG

2.4.2 Der PDG in RAP

Wie bereits erwähnt, benutzt RAP den PDG um die Registerzuteilung zuerst auf Teilbereichen des zu bearbeitenden Programmfragments auszuführen, um dann Schritt für Schritt die Färbung für die gesamte Routine vorzunehmen.

In der im Papier beschriebenen Version von RAP existieren noch keine Grundblöcke. Jede Anweisung des Quellprogramms wird als Ecke im CFG verstanden. Deswegen ist es nötig, die Ecken, die dieselben Kontrollabhängigkeiten (gemäß Definition 2) haben, in *Regionen* zusammenzufassen. Wir werden später sehen, dass der FIRM-Aufbau, der ja entsprechende Anweisungen schon in Grundblöcken zusammenfasst, die Bestimmung von Regionen überflüssig macht.

In dem Aufsatz von FERRANTE findet sich keine exakte Defnition einer *Region*, es wird lediglich beschrieben, dass eine Region aus Ecken besteht, die unter gleichen *Kontrollbedingungen* stehen.

Das RAP-Papier führt einige Bezeichnungen ein, die im folgenden noch benötigt werden. Insbesondere verstehen die Autoren des RAP-Papiers unter einer *Region* etwas anderes als FERRANTE.

1. Eine Region im Sinne des RAP-Papiers ist eine Regionsecke im Sinne von FERRANTE mit all seinen Unterregionen. Rückwärtskanten werden ignoriert.

2. Die unmittelbare Region[6] einer Region ist die Regionsecke im Sinne von FERRANTE.

[6] engl.: Immediate Region

3. Eine Unter-Region einer Region R ist eine der FERRANTE-Regionsecke untergeordnete Ecke.

Bemerkung 2 *Mit obigen Erläuterungen ist die von* RAP *definierte Untermenge des PDG zyklenfrei.*

Die obigen Erläuterungen sind, wenn man sie genau betrachtet, nicht ganz korrekt bzw. ungeeignet für RAP. Das Problem ist, dass der CDS auch Zyklen enthalten kann. Demnach würde die Unterregion einer Ecke B, wenn sie eine Rückwärtskante enthält, die unmittelbare Region R wieder enthalten, was im RAP-Algorithmus eine Endlosrekursion zur Folge hätte. Eine Anfrage an die Autoren ergab:

Bemerkung 3 *Rückwärtskanten im PDG sind für* RAP *zu ignorieren.*

2.4.3 Anpassung an FIRM und CGGG

Der wohl größte Unterschied unserer Implementation zur im Papier beschriebenen, betrifft die Grundblöcke. Im Papier zählt jede Anweisung des Quellprogrammes (die Autoren sprechen nur von einem C-Frontend) als Ecke im PDG und somit auch im CFG. Da FIRM und CGGG im CFG schon Grundblöcke gebildet haben, schien es uns vernünftig, Grundblöcke als Regionen zu benutzen und nicht einzelne Anweisungen. Dies hat zwei konkrete Vorteile:

1. Der Aufbau von Regionsecken wird entscheidend vereinfacht, denn Grundblöcke sind Regionsecken. Alle Anweisungen in einem Grundblock haben per definitionem gleiche Kontrollbedingungen. Somit müssen mit Algorithmus 2 nur noch die Kontrollbedingungen zwischen den Grundblöcken ermittelt werden.

2. Messungen der Autoren des RAP-Papiers ergaben, dass einzelne Anweisungen des Quellprogramms wohl zu klein sind um effizient Registerallokation zu betreiben. Dadurch, dass die Registerzuteilung immer zuerst auf Anweisungsebene durchgeführt wird, scheint wohl sehr viel Spill-Code erzeugt worden zu sein. Sie weisen explizit daraufhin, dass größere Einheiten (wie z .B. Grundblöcke) wahrscheinlich geeigneter sind. Siehe auch Abschnitt 4.2.1 für eine nähere Beschreibung.

2.4.4 Beispiel

Abbildung 2.3(a) zeigt den Kontrollflussgraphen der Java-Methode in Abbildung 2.2, wie er von FIRM an CGGG übergeben worden ist. Die Ecken des Graphen sind Grundblöcke, die fortlaufend nummeriert sind. Ecke 12 ist in einer Rautenform dargestellt, da sie als Schleifenkopf identifiziert worden ist (Schleifenköpfe sind von sich selbst kontrollabhängig). Dies wird deutlich, wenn man sich den Programmtext der Methode ansieht. Die Bedingung der **while**-Schleife erzeugt die Befehle, die in den Blöcken 6–12 enthalten sind. Die Entscheidung, ob der Schleifenrumpf ausgeführt wird oder nicht, wird in Block 12 getroffen (Das Verfolgen der Kante zu Block 13 verlässt die Schleife.) Deswegen ist Block 12 als Schleifenkopf anzusehen und nicht Block 6, der das Ziel der Rückwärts-Kante $(5, 6)$ ist.

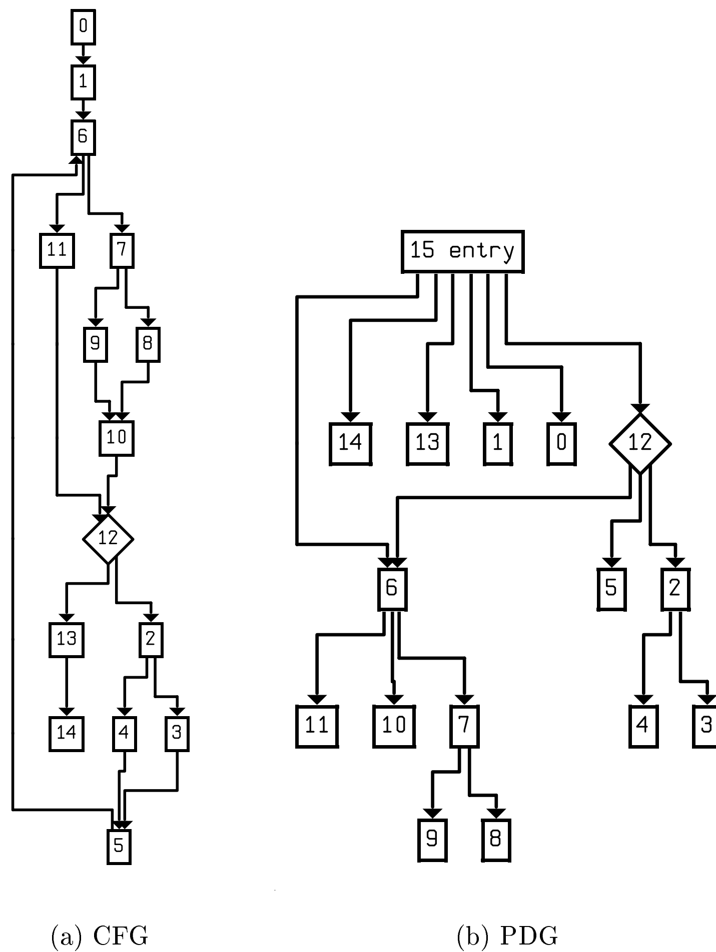
```
public static int gcd (int a, int b) {
  int i = 0;

  while((a != b) && (i >= 0)){
    if(a > b) {
      a = a - b;
    }
    else {
      b = b - a;
    }
    ++i;
  }
  return(a);
}
```

Abbildung 2.2: ggT in Java

Abbildung 2.3(b) zeigt entsprechenden PDG, wie er in RAP verwendet wird. Interessant ist, dass Block 6 (der erste Block der Schleifenbedingung) von *zwei* anderen Blöcken kontrollabhängig ist: Einerseits von **entry**, andererseits aber auch von Block 12. Dies zeigt, daß der PDG im Sinne von RAP kein Baum ist, sondern ein DAG[7], da ja Rückwärtskanten im PDG ignoriert werden und somit keine Zyklen entstehen können (siehe Bemerkung 3.)

Nach der PDG-Definition von Ferrante sollte noch eine Kante $(6, 12)$ hinzugefügt werden, da die Schleifenbedingung ja auch vom Schleifenkopf kontrollabhängig ist, was mit Definition 2 leicht nachzuprüfen ist. Diese Kante fehlt in der von RAP verwendeten Darstellung gemäß Bemerkung 3.

[7]Directed Acyclic Graph: *Gerichteter azyklischer Graph*



(a) CFG

(b) PDG

Abbildung 2.3: CFG und PDG zu Abbildung 2.2

Kapitel 3

RAP

RAP ist letztlich nichts weiter, als ein graphfärbender Registerzuteiler nach Briggs-Heuristik, der den PDG als Hierarchie verwendet. Neu daran ist die Art und Weise, wie die berechneten Teilresultate verschmolzen werden. Die folgenden Absätze geben einen kurzen Überblick über die Phasen von RAP. Die Einzelheiten werden dann in den folgenden Abschnitten besprochen.

3.1 Überblick

RAP läuft mit einer Tiefensuche durch den PDG ohne Rückwärtskanten (im folgenden RAP-PDG genannt). An einer Ecke des RAP-PDGs wird zunächst der Interferenzgraph (IFG) für diese Ecke errechnet. Im folgenden werden die schon berechneten Interferenzgraphen der RAP-PDG-Nachfolger mit dem gerade aktuellen IFG verschmolzen. Letzlich ist dann in der Ecke **entry** der komplette IFG für die Prozedur zu finden. Er ist massgeblich für die Zuteilung der Register an die temporären Variablen.

Nun wird versucht, unnütze Kopien zu elimieren. Dabei wird die berechnete Information aus dem IFG zu Rate gezogen, denn zwei Werte, von denen der eine als Kopie des anderen entsteht, können vereinigt[1] werden, wenn sie nicht interferieren. Wenn dies der Fall ist, kann die Kopieranweisung gestrichen werden. Einerseits entfallen dadurch viele Kopien, die eigentlich nicht nötig sind und zum Beispiel beim Abbau der SSA-Form entstehen, andererseits werden die Lebenszeiten der temporären Variablen aber auch erhöht.

Da das Vereinigen von temporären Variablen natürlich die Interferenzinformation ändert, muss danach der IFG neu aufgebaut werden. Dies geschieht solange, bis keine Vereinigungen mehr durchgeführt werden können.

Danach werden nun für die Ecken des IFG die Spill-Kosten berechnet. Sie sind ausschlaggebend für die Selektion der möglicherweise auszulagernden temporären Variablen. In die Berechnung gehen Häufigkeit der Benutzung, Lokalität zu einem Grundblock und Scheifenschachtelungstiefe ein.

[1] engl: coalesced

Nun können die Ecken sukzessive aus dem IFG entfernt werden und auf den Färbe-Keller gelegt werden. Dabei werden die im vorigen Schritt berechneten Spill-Kosten und der Grad einzelner Ecken im IFG berücksichtigt.

Der Färbeschritt holt nun Ecke für Ecke vom Färbe-Keller und versucht eine Farbe zu finden, die von keinem Nachbarn der Ecke besetzt ist. Gelingt dies nicht, erhält die Ecke keine Farbe und wird als ausgelagert markiert.

Falls einige Ecken ausgelagert werden sollen, wird Spill-Code eingefügt. Dazu ist die Betrachtung mehrerer Gegebenheiten wichtig, vor allem wenn Spill-Code in andere Blöcke eingefügt werden muss, wenn die temporäre Variable global zum momentan betrachteten Grundblock ist. So kann es vorkommen, dass die temporäre Variable in schon bearbeiteten Ecken des RAP-PDG benutzt wird. Das erzwingt eine Änderung des schon berechneten IFG für diese Ecke. Da das Einfügen von Spill-Code auf den meisten Plattformen[2] neue Registerbenutzungen induziert, muss der IFG neu aufgebaut werden, somit läuft die ganze Prozedur in einer Schleife, bis keine Werte mehr ausgelagert werden müssen.

Algorithmus 3 stellt die verschiedenen Phasen im Pseudocode dar.

```
Rap (B : Block)
1  spilled := true
2  while spilled do
3     repeat
4        ifg := build_ifg(B)
5        add_subregion_conflicts(B, ifg)
6        coalesced := coalesce(B, ifg)
7     until ¬coalesced
8     calculate_spill_costs(B, ifg)
9     simplify(B, ifg, color_stack)
10    spill_set := color(B, ifg, color_stack)
11    spilled := spill_set ≠ ∅
12    if spilled then
13       insert_spill_code(B, ifg, spill_set)
14    end
15 end
```

Algorithmus 3: RAP

3.2 Die einzelnen Phasen im Detail

Im folgenden werden die einzelnen Phasen von RAP im Detail beschrieben. Ich gehe hier speziell auf die implementierte Version ein, die sich durch einige Feinheiten von dem in [8] beschriebenen Original unterscheidet. Die Funktionen, die nicht mit einem Algorithmus angegeben sind, unterscheiden sich kaum von der Version im RAP-Aufsatz und können dort nachgelesen werden.

[2]Speziell auf Load/Store-Plattformen

3.2.1 Aufbau des IFG — build_ifg

Das Aufbauen des IFG für einen Grundblock geschieht analog zur Berechnung der *use*-Menge eines Grundblockes. Zunächst wird eine Menge *live*, welche die an einer Stelle lebendigen Werte enthält, mit der Menge *out* des Blockes initialisiert. Der IFG des Blockes enthält als Ecken alle Elemente aus *out*.

Die Befehlssequenz wird nun von hinten nach vorn bearbeitet; bei Erreichen einer Benutzung wird die PSR-Nummer der temporären Variable in *live* eingefügt und eine entsprechende Ecke in den IFG eingefügt; bei Erreichen einer Definition wird die PSR-Nummer p der Definition der Menge *live* entnommen und Kanten von p zu den Ecken der sich in *live* befindlichen temporären Variable gezogen.

Trifft build_ifg auf eine ausgelagerte temporäre Variable, so wird unterschieden, ob es sich um eine Definition (Spill) oder um eine Benutzung (Reload) handelt. Beim Spill erlischt die Lebenszeit der Variable sofort, da sie ja in den Hauptspeicher ausgelagert wird und keine Interferenzen mehr verursachen kann. Beim Reload darf die Lebenszeit nicht sofort beendet werden, denn die Benutzung (die dem Reload entspricht), kann ja als Operand einer mehrstelligen Operation dienen. Somit muss die Lebenszeit noch solange verlängert werden, bis die Definition des Ergebnisses der Operation erfolgt. Dies ist der Grund für die in Abschnitt 2.2 erwähnte Speicherung des Graphknotens der Operation.

Zuletzt müssen noch alle temporären Variablen, die beim Eintritt in den Grundblock lebendig zueinander sind, in Interferenz gesetzt werden. Abbildung 3.1 zeigt die IFGn aller Grundblöcke des Beispiels aus Abbildung 2.2. Die Grundblöcke enthalten jeweils ihren IFG und sind mit der selben Nummer versehen wie in Abbildung 2.2. Die Kanten zwischen den Grundblöcken sind die PDG-Kanten des Beispiels. Die Ecken des Graphen in einem Grundblock tragen folgende Bezeichnung:

PSR-Nummer Grad Datenflusseigenschaft

Die Datenflusseigenschaft einer Ecke p besteht aus drei Flags:

Flag	Bedeutung
d	$p \in def(B)$
u	$p \in use(B)$
o	$p \in out(B)$

3.2.2 Das Hinzufügen der Kind-IFG — add_subregion_conflicts

Dies ist die für die PDG-Struktur wesentliche Funktion. Sie verschmilzt den IFG eines Grundblocks V (der „unmittelbaren Region“) mit den IFGn der RAP-PDG-Kinder. Es sind folgende Situationen zu beachten:

1. Alle Interferenzen aus den Unter-Regionen müssen in den IFG von V eingefügt werden.

```
build_ifg (B : Block, ifg : (E, K))
1    live := ∅
2    for p | p ∈ out(B) ∧ (p ∈ use(B) ∨ p ∈ def(B)) do
3       live := live ∪ p
4       add_to_ifg(ifg, p)
5    end
6
7    for u := usedef_n(B) ... usedef_1(b) do
8       if def(u) then
9          live := live \ psr(u)
10         draw_edge_to_all(ifg, psr(u))
11      else if use(u) then
12         if spilled(u) then
13            draw_edge_to_all(ifg, psr(u))
14         else
15            live := live ∪ psr(u)
16         end
17      end
18   end
19
20   for i := 1 ... | live | do
21      for j := i + 1 ... | live | do
22         draw_edge(e_i, e_j)
23      end
24   end
```

Algorithmus 4: build_ifg

Abbildung 3.1: Interferenzgraphen im PDG für das ggT Beispiel aus Abbildung 2.2

2. Ein Wert v ist in V lebendig, wird dort nicht referenziert (ist also weder in $use(V)$ noch in $def(V)$), aber dafür in einer Unterrregion von V. Dann interferiert v mit allen Werten in V.

3. Ein Wert v, der beim Eintritt in eine Unterregion R lebendig ist, dort aber nicht referenziert wird (sprich, wenn B die unmittelbare Region von R ist, so ist: $v \in in(B)$, $v \notin use(B)$, $v \notin def(B)$), so muss im IFG von V von jeder Ecke eine Kante zu v gezogen werden.

3.2.3 Die Eliminierung nutzloser Kopien — coalesce

Diese Phase hat mit der eigentlichen Registerzuteilung nicht sehr viel zu tun, wird aber hier durchgeführt, da die nötigen Datenstrukturen (der IFG) von der Registerzuteilung berechnet werden, und sie somit sehr leicht implementierbar ist.

Gerade in einer SSA-Darstellung, wie ja Firm eine ist, entstehen sehr viele Kopieranweisungen, da jede temporäre Variable ja nur eine Definition hat. Das bewirkt, dass eine erneute Zuweisung an dieselbe Quelltext-Variable eine neue temporäre Variable erzeugt.

Da die Registerzuteilung hier nach der Befehlsselektion angesiedelt ist, ist die SSA-Form schon abgebaut, was einerseits dazu führt, daß Cggg die durch SSA entstandenen Kopien beibehält und die ϕ-Knoten eliminiert sind, was ein Einfügen von Kopien in Vorgänger-Grundblöcke zur Folge hat. Betrachtet man den emittierten Code, so sieht man, daß ein ganz beträchtlicher Teil der erzeugten Instruktionen Kopieranweisungen sind. Deswegen ist es für die Effizienz des Codes sehr wichtig, unnütze Kopieranweisungen zu eliminieren und die beiden Werte zu verschmelzen.

Das klingt gut und unbedenklich. Man kann sich aber verdeutlichen, dass das Verschmelzen von zwei Werten auch eine Vergrößerung der Lebensdauer zur Folge hat, was den Registerdruck erhöhen *kann*. Damit können im Extremfall andere Werte ausgelagert werden, was mit Sicherheit weniger effizient ist.

3.2.4 Das Berechnen der Spill-Kosten — calc_spill_costs

Die Spill-Kosten berechnen sich hauptsächlich aus der Nutzungshäufigkeit, der Lokalität und der Schleifenschachtelungstiefe einer temporären Variable. Darüberhinaus muss auch berücksichtigt werden, wenn ein Wert schon gespillt worden ist; er erhält dann „unendliche“ (sehr hohe) Spillkosten, um zu verhindern, dass er weiterhin für die Auswahl von auslagerbaren Werten in Betracht gezogen wird.

RAP geht in dieser Phase einen leicht anderen Weg als andere Registerzuteiler ähnlicher Couleur. Es ist gängige Praxis die Schleifenschachtelungstiefe mit in die Spill-Kosten einzubeziehen: Je tiefer ein Wert geschachtelt ist, desto öfter wird er wahrscheinlich referenziert, also sollte er nicht ausgelagert werden. Briggs [3] beschreibt einige Berechnungsformeln in seiner Dissertation.

Die beiden Autoren verzichten auf diese Gewichtung mit dem Hinweis, in einer späteren Phase, einer Art Guckloch-Optimierung, Lade- und Speicherbefehle aus den innersten

Schleifen in die äusseren zu ziehen. Dabei ist anzumerken, dass nicht jeder von RAP eingeführte Spill so bewegt werden kann. Lediglich Spills, die *nicht* auf zu hohen Registerdruck in diesem Grundblock zurückzuführen sind, können bewegt werden. Sie sind nur eingfügt worden, da die temporäre Variable in einem anderen Grundblock gespillt wurde.

3.2.5 Das Abbauen des Graphen — simplify

Die Briggs-Heuristik [3] unterscheidet sich massgeblich von der Chaitinschen [4]. Ich möchte zuerst kurz auf die Chaitinsche Heuristik eingehen, um den Unterschied der beiden Verfahren deutlich zu machen.

Chaitin trifft die Spill-Entscheidung schon in dieser Phase, da er sich darauf verlässt, dass eine Ecke x nur gefärbt werden kann, wenn gilt: $x^\circ < k$ (x° bezeichnet den Grad der Ecke x), wobei k die Anzahl der Farben bzw. Register ist. Der berühmte Diamantgraph in Abbildung 3.2 zeigt die Schwäche dieser Annahme. Er ist mit zwei Farben färbbar, Chaintin's Heuristik veranschlagt aber immer 3.

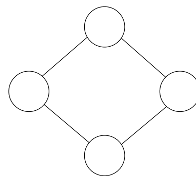


Abbildung 3.2: Der Diamantgraph

Es werden in dieser Phase also Schritt für Schritt Ecken mit Grad $< k$ aus dem Graphen entfernt und auf den Färbe-Keller gelegt. Falls eine Ecke einen Grad $\geq k$ hat, wird sie als *gespillt* markiert und aus dem Graphen entfernt, aber *nicht* auf den Färbe-Keller gelegt. Das erlaubt Chaitin in der nächsten Phase davon auszugehen, dass für jede Ecke eine Farbe vorhanden ist. (Es werden ja nur Ecken mit dem Grad $< k$ auf den Keller gelegt, die dann beim Färben betrachtet werden.)

Briggs handhabt die Situation anders. Er verschiebt die Spill-Entscheidung in die nächste Phase (color), da es ja durchaus sein kann, dass zwei Nachbarn einer Ecke dieselbe Farbe zugewiesen bekommen, was die Ecke theoretisch mit einem Grad von k trotzdem noch färbbar macht. Deswegen werden Ecken mit einem Grad $\geq k$ auch auf den Färbe-Keller gelegt; ist im Graphen keine Ecke mit Grad $\leq k$ vorhanden, wird anhand der zuvor berechneten Spill-Kosten eine Ecke ausgewählt, die gespillt werden soll[3]. Sie wird aus dem Graphen entfernt und auf den Färbe-Keller gelegt. Dies bewirkt, dass die Ecke(n) mit den höchsten Spill-Kosten ganz oben auf dem Keller liegen, was notwendig ist, da sie dann auch früher in den Graphen wieder eingefügt werden, also mit noch nicht so vielen Nachbarn „konfrontiert“ werden, was natürlich die Chancen erhöht, eine Farbe zu finden.

[3] Und zwar die, mit den geringsten Spill-Kosten

Der Vorschlag zur Implementierung stammt aus einem Papier von Briggs [2]. Zu Beginn von simplify werden zwei Mengen *high* und *low* initialisiert:

$$\begin{aligned} high &= \{t \mid t^\circ < k\} \\ low &= \{t \mid t^\circ \geq k\} \end{aligned}$$

Solange beide Mengen nicht leer sind, werden zunächst die Elemente aus *low* aus dem Graphen entfernt und auf den Färbe-Keller gelegt. Wenn *low* leer ist, wird ein Kandidat p aus *high* ermittelt, dessen Spill-Kosten möglichst gering sind. p wird aus *high* entfernt und auf den Keller gelegt. Falls p Nachbarn in *high* hat, wird deren Grad nun natürlich um eins dekrementiert. Falls deren Grad nun kleiner k ist, werden sie aus *high* entfernt und *low* hinzugefügt. Dann beginnt die Prozedur von vorn. Falls das Entfernen von p keine Ecke mit Grad kleiner k produziert hat, so werden weitere Ecken aus *high* entfernt.

```
simplify (B : Block, ifg : (E, K), k : Integer)
1   spilled := {p | p ∈ E ∧ spilled(p)}
2   special := {p | p ∈ E ∧ special(p)}
3   consider := E \ {spilled ∪ special}
4   high := {p | p ∈ consider ∧ p° ≥ k}
5   low := consider \ high
6   loop
7     while low ≠ ∅ do
8       touched := ∅
9       n := pop(low)
10      if n° > 0 then
11        touched := {p | (n, p) ∈ K ∧ p ∈ consider}
12        mark_removed(ifg, p)
13      end
14      touched := touched ∩ high
15      T := {p | p ∈ touched ∧ p° > 0 ∧ p° < k}
16      high := high \ T
17      low := low ∪ T
18      push(color_stack, n)
19    end
20    if high =  then break end
21    n := arg min_{p∈high} spill_cost(p)/p°
22    high := high \ {n}
23    low := low ∪ {n}
24  end
25  for n ∈ spilled do push(color_stack, n) end
```

Algorithmus 5: Simplify

3.2.6 Das Färben des Graphen — color

Das Färben des Graphen ist vergleichsweise simpel. Stück für Stück werden die Ecken vom Färbe-Keller geholt und in den Graphen eingefügt. Dabei werden alle schon eingefügten Nachbarn betrachtet. Haben die Nachbarn schon alle k Farben belegt, wird die Ecke als *gespillt* markiert und nicht eingefügt. Der Keller wird bis zum Ende abgearbeitet.

```
color (ifg : (E, K), color_stack : stack of nodes, k : Integer)
1  spill_set := ∅
2  while color_stack ≠ ∅ do
3     neighbour_colors := ∅
4     p := pop(color_stack)
5     for n | (p, n) ∈ K do
6        neighbour_colors := neighbour_colors ∪ color(n)
7     end
8     found := false
9     for i = 1 ... k do
10       if i ∉ neighbour_colors then
11          found := true
12          color(p) := i
13          break
14       end
15    end
16    if ¬found then spill_set := spill_set ∪ p end
17 end
```

Algorithmus 6: Color

3.2.7 Das Hinzufügen von Spill-Code — insert_spill_code

Das Einfügen von Spill-Code ist in der Arbeit von Norris/Pollock kaum beschrieben. Das hierarische Vorgehen von RAP macht es problematischer als in herkömmlichen graphfärbenden Registerzuteilern. Dafür erlaubt es aber auch einen differnzierteren Spill-Mechanismus. In herkömmlichen graphfäbenden Registerzuteilern wird eine temporäre Variable — einmal als gespillt eingestuft — überall gespillt. Jeder Zugriff darauf erzeugt ein Load bzw. Store. Da die Registerzuteilung auf Grundblockbasis geschieht (auch bei herkömmlichen GRZ), kann es sein, dass in einem Grundblock der Registerdruck zu hoch ist und ein Wert p ausgelagert wird. In anderen Blöcken kann der Registerdruck aber ausreichend niedrig sein, um p in einem Register zu halten. Es wäre nun sinnlos, jeden Zugriff auf p mit einem Speicherbefehl zu versehen. Besser wäre es, p beim Eintritt in den Block B zu laden (wenn $p \in in_B$) und p beim Verlassen in den Speicher zurückzuschreiben (natürlich auch nur, wenn $p \in out_B$). Das Laden direkt beim Eintritt in den Block kann zusätzlich noch den schönen Nebeneffekt haben, dass die Ladelatenz durch Befehle, die evtl. vor dem ersten Zugriff auf p stehen, verborgen wird.

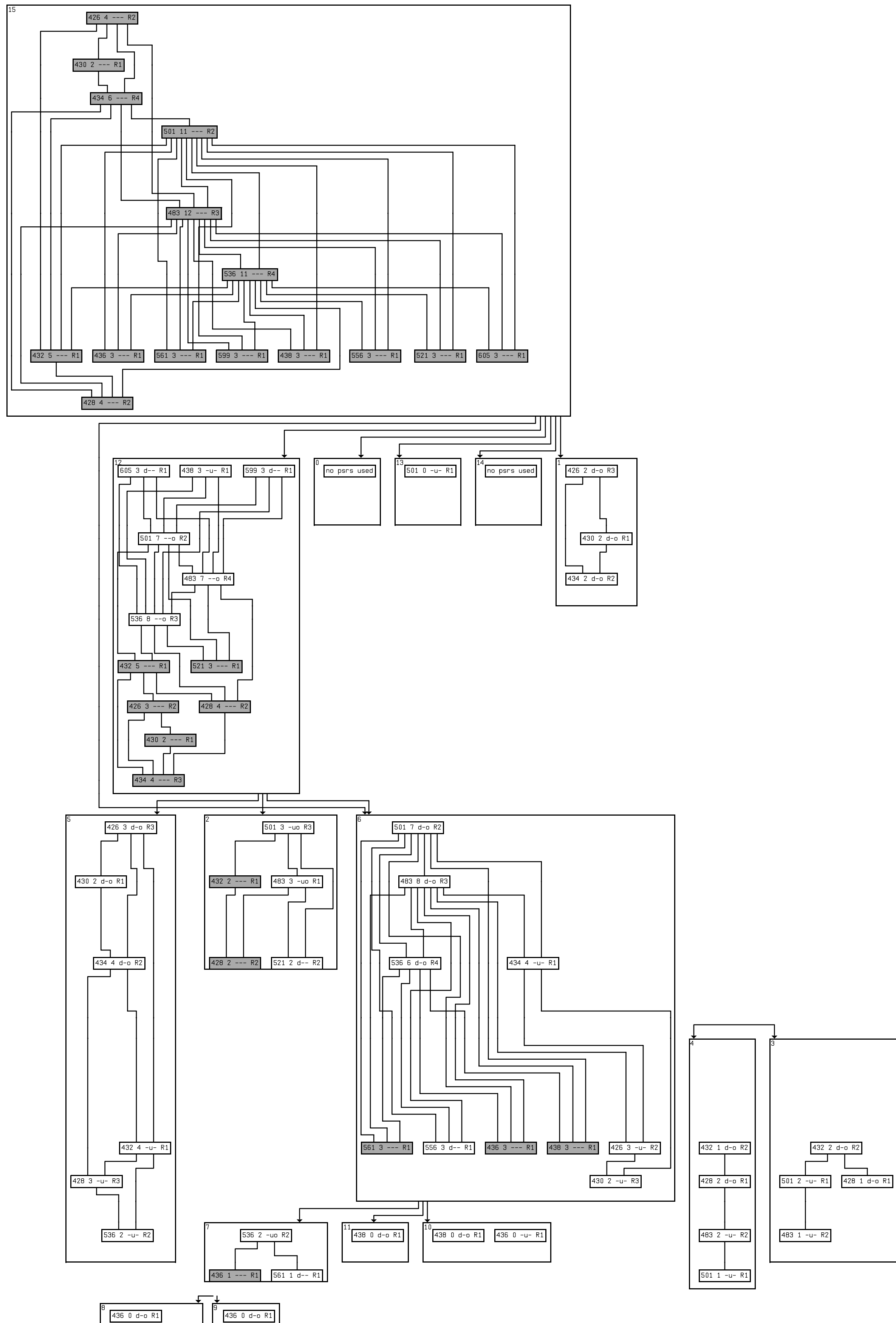


Abbildung 3.3: Gefärbte Interferenzgraphen im PDG für das ggT Beispiel aus Abbildung 2.2

Wird eine temporäre Variable p in einem Block B gespillt, so müssen alle Zugriffe auf p in B als gespillt markiert werden. Die mit der temporären Variablen assoziierte PSR-Nummer wird dabei nicht wieder verwendet, da das Spillen ja eine Aufspaltung der Lebenszeit in kleine unabhängige Lebenszeiten bewirkt. Das hat zur Folge, dass diese unterschiedlichen Lebenszeiten auch in unterschiedlichen Registern gehalten werden können, was durchaus sinnvoll ist. RAP weist jedem Auftreten dieser temporären Variable in B eine neue PSR-Nummer zu, als ob bei den einzelnen Zugriffe unterschiedliche Variablen referenzierten. RAP merkt sich für jede neue PSR-Nummer, die aus einem Spill entsteht, die PSR-Nummer der ursprünglichen Variable, damit den verschiedenen neuen PSR-Nummern, die eine einzige Variable repräsentieren, später derselbe Speicherplatz zugewiesen werden kann.

Nun müssen auch alle anderen Blöcke überprüft werden, ob sie p referenzieren. Es ist zu unterscheiden, ob ein Block C ein PDG-Kind von B ist, oder nicht. Ist C ein PDG-Kind von B, so ist der IFG von C schon fertig berechnet, und vor allem färbbar. p kann in C nur nicht-gespillt vorliegen, denn ein Spill von p in C hätte ja eine Umbenennung von p zur Folge, wie sie im vorigen Absatz beschrieben worden ist. Da die temporäre Variable aber in B gespillt wurde, und somit im Speicher liegt, muss nun in den PDG-Kindern von B dafür gesorgt werden, dass die PSR-Nummer von p nicht mehr verwendet wird. Es wird in jedem PDG-Kind eine neue PSR-Nummer angelegt, die p entspricht, aber nur lokal in diesem Block lebendig ist. Diese neue temporäre Variable wird dann beim Eintritt in den Block geladen und beim Verlassen wieder gespeichert. Das Umbenennen hat aber zur Folge, dass der IFG neu aufgebaut werden muss. p wurde ja aus den PDG-Kindern von B entfernt und ist somit auch nicht mehr in deren IFGn enthalten. Dafür ist aber eine neue PSR-Nummer entstanden, die in den IFG von B eingefügt werden muss. Dazu werden die IFGn aller PDG-Kinder nochmals rekursiv berechnet, und die Interferenzen der neu hinzugekommenen PSR-Nummern in den IFG von B eingefügt.

Desweiteren müssen auch in allen Blöcken, die bisweilen noch nicht von Rap bearbeitet wurden, die gespillten temporären Variablen markiert werden. Falls p in einem solchen Block D lebendig ist, wird p dort zu einer neuen PSR-Nummer umbenannt. Je nachdem, ob p lebendig bei Eintritt oder Verlassen von D war, wird ein Lade- bzw. Speicher-Befehl am Anfang bzw. Ende von D eingefügt.

Mögliche Beispiele für B, C und D wären in Abbildung 2.3 z.B. die Blöcke 7, 9 und 13.

```
delete_from_lt_sets (B : Block, S : set of psr)
1   def(B) := def(B) \ S
2   use(B) := use(B) \ S
3   in(B) := in(B) \ S
4   out(B) := out(B) \ S

insert_spill_code (B : Block, ifg : (E, K), spill_set : set of psr)
1   if spill_set ∩ E ≠ ∅ then
2      for x ∈ usedefs(B) do
3         if psr(x) ∈ spill_set then
4            old_psr(x) := psr(x)
5            psr(x) := new_psr
6            spilled(x) := true
7         end
8      end
9   end
10  delete_from_lt_sets(B, spill_set)
11  for C | C ∈ pdgkid(B) do
12     localize_spill_subregion(C, spill_set)
13  end
14  for C | C ∉ pdg_region(B) do
15     mark_spilled_elsewhere(C, spill_set)
16  end
```

Algorithmus 7: Einfügen von Spill-Code

```
localize_spill_subregion (B : Block, spill_set : set of psr)
1   for C | C ∈ pdgkid(B) do
2     localize_spill_subregion(C, spill_set)
3   end
4   delete_from_lt_sets(B, spill_set)
5   new_psrs := ∅
6   for x ∈ usedefs(B) do
7     if psr(x) ∈ spill_set then
8       if ¬∃y : (psr(x), y) ∈ new_psrs then
9         new_psr := new_psr
10        new_psrs := news_psrs ∪ (psr(x), new_psr)
11      else
12        new_psr := y,   (psr(x), y) ∈ new_psrs
13      end
14      psr(x) := new_psr
15    end
16    if new_psrs ≠ ∅ then
17      G := build_ifg(B)
18      for (old, new) ∈ new_psrs do
19        def(B) := def(B) ∪ new
20        n := add_node(ifg(B), new)
21        for i := 1 ... | neighbours(G, new) |  do add_edge(ifg(B), new, neighbour_i(G, new)) end
22      end
23    end
```

Algorithmus 8: Einfügen von Spill-Code in PDG Unter-Regionen

mark_spilled_elsewhere (B : Block, $spill_set$: set of psr)

```
1   processed := ∅
2   for x ∈ usedefs(B) do
3     if psr(x) ∈ spill_set ∧ XXX then
4       if ¬∃y : (psr(x), y) ∈ processed then
5         new_psr := new_psr
6         processed := processed ∪ (psr(x), new_psr)
7         if psr ∈ in(B) then spilled_extern_in(B) := spilled_extern_in(B) ∪ psr end
8         if psr ∈ out(B) then spilled_extern_out(B) := spilled_extern_out(B) ∪ psr end
9         psr(u) := new_psr
10      else
11        new_psr := y,   (psr(x), y) ∈ processed
12      end
13      if Block was already processed by the rap procedure then
14        set_alias(new_psr, psr)
15      end
16    end
17  end
18
```

Algorithmus 9: Markieren von Werten in Blöcken, die nicht in der momentanen Unter-Region sind

Kapitel 4

Unterschiede, Erweiterungen und Ausblick

Bei der Implementierung von RAP wurden einige Änderungen und Erweiterungen gegenüber der in der Arbeit von Norris/Pollock beschriebenen Version vorgenommen. Einige Dinge wurden aus Zeitmangel nicht vollständig ausgearbeitet, andere wurden hinzugefügt, da die Praxistauglichkeit immer im Vordergrund stand. In diesem Kapitel sollen die Änderungen an RAP, die Erweiterungen und mögliche zukünftige Ergänzungen diskutiert werden.

4.1 Prinzipielle Unterschiede

Folgende Dinge sind in unserer Version anders implementiert bzw. wurden weggelassen:

1. In der Funktion Rap wurden einige Änderungen vorgenommen. Der combine-Schritt aus dem Paper wurde weggelassen. Combine verbessert eigentlich nur den Speicherverbrauch hinsichtlich der Kanten des IFG. Es erhöht aber gleichzeitig den Verwaltungsaufwand, da in einem „kombinierten“ IFG eine Ecke gleich mehrere ursprüngliche Ecken repräsentiert. Dies wird gerade dann problematisch, wenn eine der repräsentierten Ecken gespillt wird und die kombinierte Ecke somit wieder getrennt werden muss. Es schien uns der Mühe nicht Wert, diesen hohen Verwaltungsaufwand einzugehen, nur um den Speicher bei den Kanten zu sparen.

2. Auch simplify und color wurden eher in Anlehnung an Briggs' Doktorarbeit [3] implementiert, da uns seine Implementierung plausibler erschien.

3. Das Verschieben von Spill-Code (in [8] als „Phase 2“ bezeichnet) wurde aus Zeitgründen nicht implementiert.

4.2 Erweiterungen

Rap wurde um die folgenden Eigenschaften erweitert.

4.2.1 Grundblöcke als kleinstes Allokationsgebiet

Die Zwischensprache des C-Übersetzers, für den Norris/Pollock Rap implementierten, ist PDG-basiert. Sie scheint so gemacht zu sein, dass jede C-Anweisung einer Region entspricht. Deswegen führen Norris/Pollock die `Rap`-Prozedur (siehe Algorithmus 3) für jede C-Anweisung aus. Das Problem ist, dass Rap Spillcode an Regionsgrenzen einfügt (siehe Abschnitt 3.2.7). Je weniger Regionen man hat, desto weniger Spill-Code muss eingefügt werden.

Die Zwischenrepräsentation Firm basiert nicht auf dem PDG, sondern benutzt den Kontrollflussgraphen um Kontrollabhängigkeiten darzustellen. Alle Anweisungen (in Firm Datenflussecken) werden einem Grundblock zugeordnet. Dieser ist per definitionem schon die längste Sequenz von Instruktionen unter gleicher Kontrollbedingung. Dies sollte ein günstigeres Verhalten bezüglich Spill-Code bewirken.

4.2.2 Verschiedene Registerklassen

Die meisten Prozessoren unterscheiden zwischen verschiedenen Registerklassen. Zum Beispiel hat der Alpha Prozessor 32 64-Bit Integer-Register (`r0`–`r31`) und 32 64-Bit Gleitkomma-Register (`f0`–`f31`). Beide Registersätze sind disjunkt. Das bedeutet, daß die Registerzuteilung für beide Registersätze getrennt erfolgen kann.

Interessanter sind die Situationen, in denen Registern Sonderbedeutungen zukommen. Zum Beispiel erwartet der Multiplikationsbefehl (`imul`) der ia32-Architektur die beiden Operanden in den Registern `eax` und `edx`. Der Registerzuteiler muss also dafür Rechnung tragen, daß die entsprechenden Werte auch in diesen Registern stehen. Um diesen Sachverhalt zu modellieren, bietet Cggg die Möglichkeit, verschiedene *Registertypen* zu definieren. Abbildung 4.1 zeigt ein Beispiel einer solchen Definition.

```
REGISTERS <r1 .. r16, cc1 .. cc2, f1 .. f8>
    Register            <r1 .. r16>   ->;
    AddrRegister        <r1 .. r8>    ->;
    CondCodeReg         <cc1 .. cc2>  ->;
    DivRes              <r3 .. r4>    ->;
    ModRes              <r3 .. r4>    ->;
    FloatReg            <f1 .. f8>    ->;
```

Abbildung 4.1: Definition verschiedener Registertypen

Registertypen sind hier z. B. `Register`, `AddrRegister` oder auch `CondCodeReg`. Wie man sieht, bildet der Registertyp `AddrRegister` eine echte Untermenge von `Register`. Die

Registersätze bestehen in diesem Beispiel aus `r1`–`r16`, `cc1`–`cc2` und `f1`–`f8`.

Unsere Implementierung von RAP ermöglicht es einerseits, jedem Befehl explizit Quell- und Ziel-Registermengen zuzuweisen, als auch nicht paarweise disjunkte Registertypen zuzulassen, wie das z. B. im Falle der ia32-Architektur von Nöten ist. Praktisch gesehen sind beide Fälle äquivalent, bzw. können mit demselben Trick implementiert werden; Zusätzlich zu allen Ecken des IFG fügt man noch für jedes Register der momentan bearbeiteten Registerklasse R_j eine zusätzliche Ecke r_i mit der „Farbe“ des Registers ein. Will man nun erzwingen, dass einem Wert, repräsentiert durch die Ecke p nur in ein Register einer Teilmenge $R' \subset R_j$ zugewiesen werden kann, so fügt man folgende Kantenmenge dem IFG hinzu:

$$\{(p, r) \mid r \in R_j \setminus R'\} \tag{4.1}$$

Beim Färben des Graphen werden nun automatisch nur Farben aus R' betrachtet, da p nach obigem Schritt Ecken mit den allen Farben, die nicht aus R' sind, zum Nachbar hat.

Definiert man mit CGGG zwei Registertypen R_1 und R_2 desselben Registersatzes R, die nicht paarweise disjunkt sind, so vereinigt der Registerzuteiler diese Typen zu einer *Registerklasse* und fügt für eine Ecke p, welcher nur Register einer Teilmenge $R' \subset R$ zugewiesen werden können, analog zu 4.1 entsprechende Kanten in den IFG ein. Der Registerzuteilungsprozess iteriert über alle vorhandenen Registerklassen.

Im Beispiel aus Abbildung 4.1 würde der Registerzuteiler folgende Registerklassen bilden:

$$\begin{aligned}
\mathcal{R}_1 &= \texttt{Register} \cup \texttt{AddRegister} \cup \texttt{DivRes} \cup \texttt{ModRes} \\
\mathcal{R}_2 &= \texttt{CondCodeReg} \\
\mathcal{R}_3 &= \texttt{FloatReg}
\end{aligned}$$

In diesem Falle sind die Registerklassen mit den Registersätzen identisch. Würde der Registertyp `Register` nur die Register `r9`–`r16` beinhalten, so gäbe es noch eine vierte Registerklasse. Angenommen, ein IFG besteht aus sechs Ecken a–f, wobei

$$\begin{aligned}
a, b &\in \texttt{Register} \\
c, d, e &\in \texttt{AddrRegister} \\
f &\in \texttt{DivRes}
\end{aligned}$$

Der um die Registerecken erweiterte IFG würde dann so aussehen:

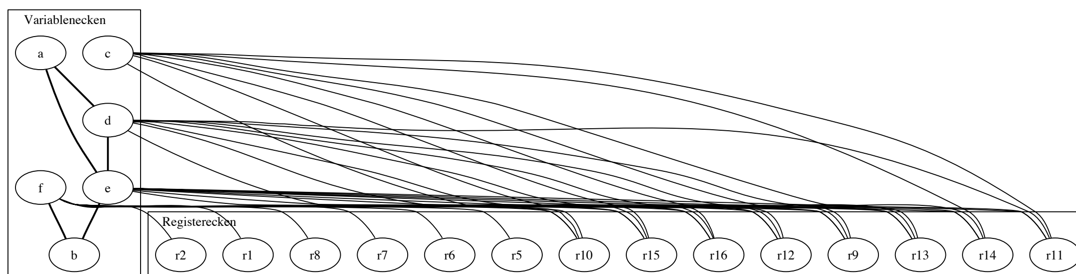


Abbildung 4.2: IFG mit Registerecken

Man sieht, daß diese Erweiterung eine Menge neuer Kanten einfügt, was bei grossen Interferenzgraphen schnell sehr ineffizient wird. Da aber für jeden Registertyp immer dieselben Kanten eingefügt werden (Ecken mit dem Registertyp `AddRegister` haben z. B. immer Kanten zu den Ecken $r9$ bis $r16$), kann man die Zielecken zu einer Ecke zusammenfassen und dieser neuen Ecke einfach mehrere Farben zuweisen. Dies erfordert nur minimale Änderungen in der color-Routine. Der neue IFG sieht dann so aus:

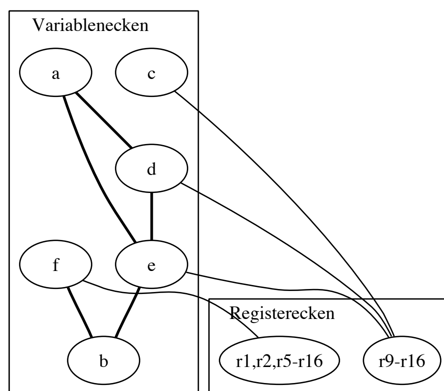


Abbildung 4.3: IFG mit mehrfarbigen Registerecken

4.3 Ausblick

Folgende Erweiterungen und Überarbeitungen wären sinnvoll, um RAP leistungsfähiger zu machen.

1. Eine verbesserte Methode um Spill-Code zu plazieren. Man könnte einige induzierte Spills vermeiden, wenn man den Kontrollflußgraphen besser analysieren würde. Insbesondere könnte man Schleifen besser berücksichtigen.

2. Eine bessere Spill-Kostenfunktion, die Schleifenschachtelungen berücksichtigt.

3. Eine Schnittstelle von CGGG zum Registerzuteiler, in dem bestimmte Informationen ausgetauscht werden können:

- Erzeugt das Einfügen von Spill-Code zusätzlichen Registerdruck, oder verfügt die Architektur über Speicheroperanden in den Instruktionen (z.B. ia32)?
- Wieviele und welche Werte wurden gespillt. Danach richtet sich die Grösse und das Layout der Spill-Area im Stack-Segment.
- Wieviele Register wurden maximal zugeteilt. Das müsste z.B. ein ia64 Codegenerator wissen, um den Register-Stack-Frame korrekt zu initialisieren.

Anhang A

Messungen

Die Messungen wurden mit folgenden fünf Programmen aus der JIKES-AJACS Testsuite durchgeführt:

Heapsort Sortiert ein Feld mit dem Heapsort-Algorithmus.

Queens Löst das Acht-Damen-Problem.

Quicksort Sortiert ein Feld mit dem Quicksort-Algorithmus.

Sieve Berechnet alle Primzahlen von 2 bis 500 mit dem Sieb des Eratosthenes.

While Berechnet den ggT zweier Zahlen.

Alle Tests wurden für verschieden große Registersätze mit jeweils 4, 5, 6, 7, 8, 10, 12, 16, 32 und 64 Registern durchgeführt. Dabei gab es (bis auf das Condition-Code-Register) keine unterschiedlichen Registerklassen.

A.1 Erzeugter Spill Code

Für jedes der fünf Programme wurden die erzeugten Spills pro Registersatzgröße ermittelt. Es wurden die gespillten Werte des Programms und die daraus resultierenden direkten und induzierten Spills gezählt. Erwartungsgemäß nehmen sie mit zunehmender Registersatzgröße ab. Ein Registersatz von 16–24 Registern scheint ist für diese kleinen Programme ausreichend zu sein.

Es wäre interessant, größere Programme zu testen (insbesondere numerische Algorithmen) um deren Registerverbrauch zu messen. Für gewöhnliche, eher kleinere Prozeduren mit wenig geschachtelten Schleifen scheinen diese 16–24 Register aber ausreichend zu sein.

Hinzu kommt, dass die Befahlsauswahl im Moment noch nicht auf Minimierung des Registerdrucks optimiert ist und eine Rückkopplung zum Codeerzeuger noch nicht vorhanden ist. Es ist möglich (und wahrscheinlich), dass sich durch eine bessere Integration des Registerzuteilers in CGGG eine bessere Registerzuteilung ergibt.

Abbildung A.1 zeigt das Ergebnis der Messung. Folgende Abkürzungen werden benutzt:

R Größe des Registersatzes

GW Anzahl der gespillten Werte im ganzen Programm

DS Direkte Spills. Spills, die daher rühren, dass ein Wert, der in einem Grundblock benutzt wird, gespillt wird.

IS Induzierter Spill. Spills, die durch das Auslagern des Wertes in einem anderen Grundblock enstehen.

Die Zahl in Klammern in der Titelzeile einer Tabelle gibt die Gesamtanzahl der von CGGG erzeugten Werte des Programms an.

Heapsort (662)			
R	GW	DS	IS
4	65	120	97
5	42	64	59
6	28	36	38
7	21	25	25
8	14	14	16
9	12	10	15
10	9	7	12
12	4	3	6
14	1	1	2
16	0	0	0
32	0	0	0
64	0	0	0

Queens (483)			
R	GW	DS	IS
4	49	72	88
5	31	40	61
6	22	26	40
7	17	19	31
8	8	10	10
9	5	8	3
10	3	6	0
12	1	2	0
14	0	0	0
16	0	0	0
32	0	0	0
64	0	0	0

Quicksort (507)			
R	GW	DS	IS
4	69	141	102
5	52	102	77
6	41	67	59
7	35	53	53
8	32	43	49
9	27	34	44
10	23	27	35
12	14	14	20
14	8	10	9
16	4	6	5
32	0	0	0
64	0	0	0

Sieve (389)			
R	GW	DS	IS
4	34	49	52
5	22	27	37
6	14	17	21
7	8	9	12
8	4	6	6
9	2	4	0
10	1	2	0
12	0	0	0
14	0	0	0
16	0	0	0
32	0	0	0
64	0	0	0

While (325)			
R	GW	DS	IS
4	24	32	31
5	11	11	17
6	6	5	10
7	2	1	4
8	0	0	0
9	0	0	0
10	0	0	0
12	0	0	0
14	0	0	0
16	0	0	0
32	0	0	0
64	0	0	0

Abbildung A.1: Erzeugter Spill Code

Für eine nähere Beschreibung des Spillings siehe Abschnitt 3.2.7.

A.2 Anteil von RAP an der Übersetzerlaufzeit

Die fünf Programme wurden mit den verschiedenen Registersätzen jeweils dreißig mal übersetzt. Die Laufzeiten wurden gemittelt und sind in Abbildung A.2 aufgetragen.

Wie zu erwarten, nimmt die Laufzeit mit zunehmender Registersatzgröße ab, da kein Spill-Code mehr erzeugt wird und somit zusätzliches Wiederaufbauen und Neufärben des Graphen entfällt. Warum allerdings die Laufzeit bei 32 Registern über derjenigen mit 16 Registern liegt, ist unklar. Wahrscheinlich ist dieses Verhalten auf Cache-Effekte zurückzuführen.

Register	HeapSort	Queens	QuickSort	Sieve	While
4	3.82	4.08	3.98	4.59	5.05
5	3.53	3.93	3.72	4.46	4.82
6	3.38	3.88	3.55	4.32	5.00
7	3.25	3.78	3.52	4.22	4.60
8	3.13	3.65	3.48	4.11	4.50
9	3.14	3.61	3.50	4.08	4.52
10	3.11	3.58	3.47	4.08	4.50
12	3.13	3.61	3.43	4.07	4.46
14	3.10	3.56	3.38	4.04	4.45
16	3.09	3.55	3.35	4.03	4.48
32	3.27	3.81	3.55	4.31	4.73
64	3.05	3.53	3.25	4.06	4.50

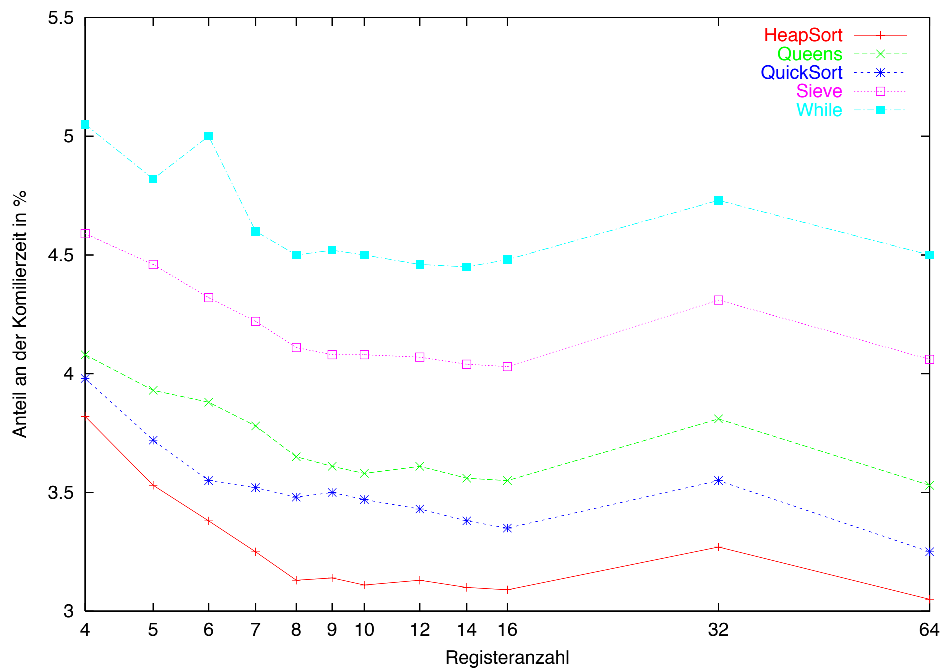


Abbildung A.2: Anteil von RAP an der Kompilierzeit in Prozent

Anhang B

Notation

Folgende typographische Konventionen wurden in den Pseudo-Code-Fragmenten verwendet:

Notation	**Bedeutung**
p°	Grad einer Ecke p eines Graphen
color	Name eines Algorithmus oder einer Funktion
$spill_set$	Objekt (Variable) in einem Programm
$color(p)$	Attribut eines Objektes. Zum Beispiel die Farbe einer Ecke p im IFG

Literaturverzeichnis

[1] Boris Boesler. Codeerzeugung aus Abhängigkeitsgraphen. Master's thesis, Universität Karlsruhe, 1998.

[2] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring Heuristics for Register Allocation. *Proc. ACM SIGPLAN*, pages 275–284, June 1989.

[3] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[4] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P. Markstein. Register Allocation via Coloring. *Computer Languages*, 6:47–57, 1981.

[5] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(2):319–349, July 1987.

[6] Thomas Lengauer and Robert E. Tarjan. A fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[7] Robert Morgan. *Building an Optimizing Compiler*. Butterworth–Heinemann, 1998.

[8] Cindy Norris and Lori L. Pollock. The Design and Implementation of RAP: A PDG-based Register Allocator. *Software, Practice and Experience*, 28(4):401–424, 1998.

[9] Massimiliano Poletto and Vivek Sarkar. Linear scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[10] M. Trapp, G. Lindenmaier, and B. Boesler. Documentation of the Intermediate Representation FIRM. Technical Report 1999-14, Universität Karlsruhe, Fakultät für Informatik, Dec 1999.