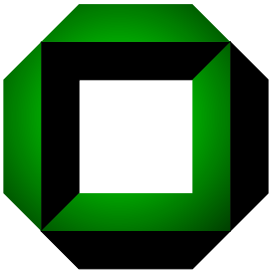


Universität Karlsruhe (TH)

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

Kopienminimierung in einem SSA-basierten Registerzuteiler

Diplomarbeit von Daniel Grund

August 2005

Betreuer:
Dipl.-Inform. Sebastian Hack

Verantwortlicher Betreuer:
Prof. em. Dr. Dr. h.c. Gerhard Goos

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

Ort, Datum Unterschrift

Kurzfassung

Die Registerzuteilung ist eine der wichtigsten Optimierungsphasen eines Übersetzers. Sie weist möglichst vielen Variablen eines Programms Prozessorregister zu und lagert die verbleibenden in den Hauptspeicher aus. Somit hat sie großen Einfluss auf das Laufzeitverhalten des übersetzten Programms.

Neben dem eigentlichen Zuteilen von Registern ist die Minimierung von unnötigen Kopien eine der Hauptaufgaben eines Registerzuteilers. Durch die Minimierung dieser Kopien wird im Allgemeinen jedoch der Registerbedarf erhöht, was zu zusätzlichen Auslagerungen führen kann und somit kontraproduktiv ist. Die in dieser Arbeit betrachtete Registerzuteilung für so genannte SSA-Programme ermöglicht es, die größte Menge an Kopien zu bestimmen, deren Eliminierung keine Auslagerung hervorruft.

In dieser Arbeit stellen wir, neben einer genauen komplexitätstheoretischen Einordnung, optimale Lösungs- und Näherungsverfahren zur Kopienminimierung vor. Abschließende Messungen belegen die Leistungsfähigkeit der von uns vorgeschlagenen Heuristik.

Inhaltsverzeichnis

1 Einleitung

Kopienminimierung ist die Aufgabe, alle in einem Programm enthaltenen Kopieroperationen zu finden und möglichst viele unnötige davon zu entfernen. Diese können zum Beispiel beim Abbau der so genannten SSA-Form und bei der Behandlung von Registerbedingungen entstehen. Die Kopienminimierung findet in den meisten bisherigen Ansätzen während der Registerzuteilung statt, wird aber auch teilweise schon beim SSA-Abbau berücksichtigt. Das Optimieren, also Eliminieren, einer Kopie kann negative Auswirkungen haben. Zum Beispiel kann sich dadurch der Registerbedarf erhöhen und deshalb, aufgrund des eingefügten Auslagerungscodes, die Ausführungsgeschwindigkeit des Programms abnehmen. In einem klassischen Registerzuteiler können diese Auswirkungen mit akzeptablem Zeitaufwand nicht immer sicher vorhergesagt werden. Will man den Registerbedarf nicht unnötig erhöhen, ist man deswegen auf pessimistische Verfahren festgelegt.

1.1 Anforderungen

In dieser Arbeit soll die Kopienminimierung in einem neuartigen Registerzuteiler untersucht werden. Dieser teilt die Register auf Programmen in SSA-Form zu. Eine Eigenschaft (Chordalität) der zugehörigen Interferenzgraphen erlaubt es uns, den Registerbedarf an jeder Stelle im Programm exakt zu bestimmen und die oben genannten negativen Auswirkungen zu vermeiden. Durch diese neue Vorgehensweise ergibt sich auch eine leicht veränderte Aufgabenstellung für die Kopienminimierung: Ändere eine gegebene zulässige Registerzuteilung so ab, dass die Anzahl der Kopien minimiert wird, ohne den Registerbedarf über die zur Verfügung stehenden Register zu erhöhen. Es sei bemerkt, dass die Lösung nicht minimal viele Register benutzen muss.

Ziele der vorliegenden Arbeit sind die komlexitätstheoretische Einordnung, sowie die Entwicklung und Implementierung von Verfahren zur exakten bzw. näherungsweisen Lösung des Problems.

1.2 Lösungsansatz

Der geführte Komplexitätsbeweis ordnet das Problem in die bekannte Klasse NP-vollständiger Probleme ein. Aufgrund der Schwierigkeit solche Probleme effizient zu lösen, verfolgt die Arbeit zwei Lösungsmethoden: Zum einen eine Tauschheuristik, die eine initiale Registerzuteilung direkt manipuliert, um

Kopien zu eliminieren. Zum anderen ein Verfahren, das eine optimale Lösung mittels Transformation auf ein mathematisches Optimierungsproblem (mixed integer linear programming, MILP) bestimmt. Mit Hilfe der optimalen Lösungen wird auch die Güte der Heuristik bewertet. Die Implementierung erfolgt in einem Compiler, der die SSA-Zwischendarstellung Firm [1] [2] verwendet. Nach Befehlsauswahl und -anordnung, die die SSA-Eigenschaft erhalten, schließt sich die Registerzuteilung an. In deren letzter Phase findet die Kopienminimierung gleichzeitig mit dem SSA-Abbau statt.

1.3 Überblick

Zunächst werden in Kapitel 2 die für diese Arbeit nötigen Grundlagen der Graphentheorie und der mathematischen Optimierung gegeben. In Kapitel 3 stellen wir verwandte Arbeiten zur Kopienminimierung vor und vergleichen diese miteinander. Kapitel 4 motiviert zunächst die neue Vorgehensweise für Kopienminimierung und führt auf die neue Problemstellung hin. Mit einem Überblick über die Phasen der Registerzuteilung, wird aufgezeigt, wie sich unser Ansatz der Kopienminimierung in die Registerzuteilung einfügt. Nach einer Komplexitätsbetrachtung werden zwei Lösungsmethoden für das Problem vorgestellt. Kapitel 5 beschreibt die praktische Umsetzung der Verfahren aus Kapitel 4 in einem Übersetzer. Die mit diesem Übersetzer durchgeführten Messungen der Laufzeit und Güte der Lösungsverfahren werden in Kapitel 6 präsentiert. Kapitel 7 schließt mit einer Zusammenfassung und einem Ausblick.

2 Grundlagen

2.1 Graphentheorie

Definition 2.1 (Graph). Ein ungerichteter Graph $G = (V, E)$ ist ein Tupel bestehend aus einer Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$. G heißt *schlicht*, falls $\forall v \in V : (v, v) \notin E$.

Definition 2.2 (Teilgraph). $H = (\bar{V}, \bar{E})$ heißt *Teilgraph* von $G = (V, E)$, falls $\bar{V} \subseteq V, \bar{E} \subseteq E$ und $(v_1, v_2) \in \bar{E} \Rightarrow v_1, v_2 \in \bar{V}$.
Für eine Teilmenge $\bar{V}$ von V ist $G[\bar{V}]$ ein *induzierter Teilgraph* mit Knotenmenge $\bar{V}$ und maximal großer Kantenmenge: $(v_1, v_2) \in \bar{E} \Leftrightarrow v_1, v_2 \in \bar{V} \wedge (v_1, v_2) \in E$.

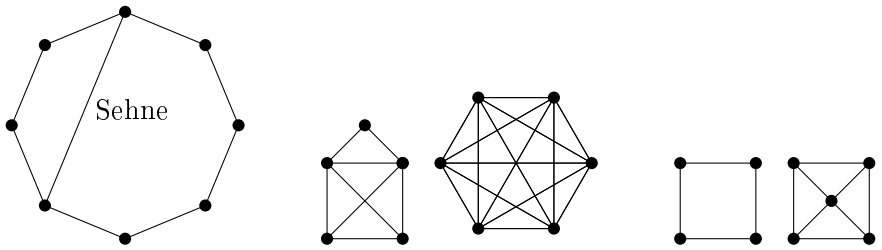
Definition 2.3 (Clique). Eine *Clique* ist eine vollständig verbundene Teilmenge der Knoten eines schlichten Graphen. $C \subseteq V$ heißt Clique in $G = (V, E) \Leftrightarrow \forall v_1, v_2 \in C : v_1 = v_2 \vee (v_1, v_2) \in E$.
Ist der Graph nicht schlicht, muss eine Clique zusätzlich alle möglichen Schlingen enthalten: $\forall v \in S : (v, v) \in E$.
Eine *maximale Clique* enthält maximal viele Knoten aus V. Eine *abgeschlossene Clique* muss nicht maximal sein, kann aber durch Hinzunahme eines weiteren Knoten aus V nicht zu einer größeren Clique ergänzt werden.

Definition 2.4 (stabile Menge). Eine *stabile Menge* ist eine Teilmenge der Knoten eines schlichten Graphen, innerhalb derer keine zwei Knoten adjazent sind. $S \subseteq V$ heißt stabile Menge in $G = (V, E) \Leftrightarrow \forall v_1, v_2 \in S : v_1 = v_2 \vee (v_1, v_2) \notin E$.
In nichtschlichten Graphen sind alle Knoten einer stabilen Menge schlingenfrei.
Die Definitionen für *maximale* und *abgeschlossene stabile Mengen* ergeben sich analog zu denen für Cliquen.

Definition 2.5 (chordal). Ein Graph heißt *chordal* (oder trianguliert), falls alle induzierten Kreise aus genau drei Knoten bestehen. Äquivalent dazu ist die Aussage, dass alle Kreise mit mindestens vier Knoten eine Sehne besitzen. Eine Sehne ist eine Kante zwischen zwei nicht aufeinander folgenden Knoten eines Kreises. Siehe dazu auch Abbildung 2.1.

Definition 2.6 (simplizial). $N(v) = \{w \in V | (v, w) \in E\}$ bezeichne die Nachbarschaft von v; alle zu v adjazenten Knoten. Ein Knoten $v \in V_G$ heißt simplizial, falls der induzierte Teilgraph $G[\{v\} \cup N(v)]$ aus v und seiner Nachbarschaft eine Clique ist.

Definition 2.7 (perfekte Eliminationsordnung, PEO). Eine Folge $v_1, \ldots, v_n$ aller Knoten eines Graphen G heißt perfekte Eliminationsordnung, falls jedes v_i in $G[v_i, \ldots, v_n]$ ein simplizialer Knoten ist.



(a) Kreis mit Sehne (b) chordale Graphen (c) nicht chordale Graphen

Abbildung 2.1: Chordalität an Beispielen

Satz 2.8. Ein Graph G ist genau dann chordal, falls G eine perfekte Eliminationsordnung besitzt. Für einen Beweis siehe zum Beispiel [3].

Unter Graphfärben versteht man das Problem, die Knotenmenge V eines ungerichteten Graphen in disjunkte Teilmengen $V_1 \dots V_k$ so zu partitionieren, dass keine Kante zwei Knoten innerhalb einer Partition verbindet. Anders ausgedrückt teilt man jedem Knoten eine Farbe zu, sodass keine zwei benachbarten Knoten dieselbe Farbe besitzen. Der Graph heißt dann, je nach Anzahl der benutzten Teilmengen, k-färbbar. Von einer minimalen Färbung spricht man dann, falls k minimal unter allen zulässigen Färbungen ist.

Nach Golumbic [3] kann man einen chordalen Graphen effizient minimal färben. Für eine perfekte Eliminationsordnung $v_1, \dots, v_n$, die in $O(|V| + |E|)$ berechenbar ist, färbt man die Knoten in der Reihenfolge $v_n, \dots, v_1$, wobei man nur beachten muss, dass die für einen Knoten gewählte Farbe nicht schon in seiner Nachbarschaft verwendet wird. Diese Vorgehen resultiert in einer minimalen Färbung.

2.2 Mathematische Optimierung

Definition 2.9 (Lineares Programm). Ein mathematisches Optimierungsproblem

$$\begin{aligned} & min\ z = f(x) \\ \text{u.d.N.:}\quad & g_i(x) \leq b_i \\ & g_j(x) = b_j \\ & x \in \mathbb{R}^n \end{aligned}$$

bei dem f und alle g_i, g_j linear sind, heißt lineares Optimierungsproblem oder *Linear Program (LP)*. Diese lassen sich immer[1] in folgender Standardform darstellen:

$$\begin{aligned} \text{(LP)}\quad & min\ z = c^T x \\ \text{u.d.N.:}\quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

[1] Nicht beschränkte $x \in \mathbb{R}$ kann man durch $a - b$ mit $a, b \geq 0$ substituieren. Detaillierte Informationen zu Transformationen finden sich zum Beispiel in [4].

Sind für alle bzw. einige der x_i nicht beliebige reelle, sondern nur ganze Zahlen zulässig, so nennt man das Problem *Integer Linear Program (ILP)* bzw. *Mixed Integer Linear Program (MILP)*:

$$\begin{array}{lll} \text{(ILP)} & min\ z = c^T x & \\ & \text{u.d.N.:} & Ax \leq b \\ & & x \in \mathbb{N}^n \end{array}$$

Definition 2.10 (zulässig, optimal). Die Menge $M = \{x : Ax \leq b, x \geq 0\}$ heißt *zulässiger Bereich* des LPs und ein $x \in M$ *zulässige Lösung*. x^* ist eine *optimale Lösung*, falls $\forall y \in M : c^T y \geq c^T x^*$.

Für LPs existieren Algorithmen, die eine optimale Lösung in polynomialer Zeit finden können, sofern das Problem nicht unbeschränkt oder unlösbar ist. Für (M)ILPs gilt im Allgemeinen jedoch, dass die Bestimmung einer optimalen Lösung NP-vollständig ist. Fast alle Algorithmen benutzen daher die Simplexmethode in Kombination mit Branch & Cut- oder Branch & Bound-Verfahren um MILPs zu lösen. Dabei wird das Problem (P) zunächst *relaxiert*, indem die Ganzzahligkeitsbedingungen ignoriert werden. Man löst also zunächst die so genannte *LP-Relaxation* (Q) von (P). Ist die dadurch erhaltene Lösung für (P) nicht zulässig, fügt man zusätzliche Ungleichungen (Restriktionen) zu (Q) hinzu, die diese Lösung auch in (Q) unzulässig macht, ohne dabei jedoch zulässige Lösungen von (P) auszuschließen. Die erste Lösung von (Q), die auf diese Art und Weise bestimmt wird, und auch für (P) zulässig ist, ist eine optimale Lösung von (P).

Für ein zweidimensionales Problem lässt sich dies recht gut veranschaulichen. Die Nebenbedingungen definieren Geraden, die den Raum in eine zulässige und eine unzulässige Hälfte unterteilen. Der Schnitt all dieser (konvexen) Halbräume bildet den konvexen zulässigen Bereich M eines LPs. Liegt ein (ILP) vor, so verkleinert sich der zulässige Bereich auf die Gitterpunktmenge im ursprünglichen Bereich. Abbildung 2.2 zeigt die Zusammenhänge zwischen dem zulässigen Bereich eines ILPs, dessen konvexer Hülle und der zugehörigen LP-Relaxation.

Definition 2.11 (gültig, stark, facettendefinierend). Gegeben seien ein ILP, eine lineare Ungleichung $a^T x \leq b$, der zulässige Bereich M, sowie dessen konvexe Hülle $\Pi = \text{conv}(M)$ des Problems. Die Ungleichung heißt *gültig*, falls $\forall x \in M : a^T x \leq b$. Eine gültige Ungleichung ist *stark gültig*, falls $\exists x \in M : a^T x = b$. Gilt sogar noch $\dim(M \cap \{x : a^T x = b\}) = \dim(\Pi) - 1$, dann heißt die Ungleichung *facettendefinierend*.

Eine starke gültige Ungleichung definiert also eine Hyperebene, die mit dem zulässigen Bereich mindestens einen Punkt gemeinsam hat. Bei facettendefinierenden Ungleichungen hat der Schnitt aus zulässigem Bereich und Hyperebene maximale Dimension, und definiert somit genau eine Seite der konvexen Hülle $\Pi = \text{conv}(M)$. Hat ein LP eine optimale Lösung, dann ist auch mindestens eine Ecke des zulässigen Bereichs optimale Lösung. Könnte man alle Seiten von Π durch Ungleichungen beschreiben, wäre eine optimale Lösung der LP-Relaxation automatisch auch eine optimale Lösung für das ILP selbst, da alle Ecken von Π durch Gitterpunkte definiert sind. Findet man problemspezifische Klassen von

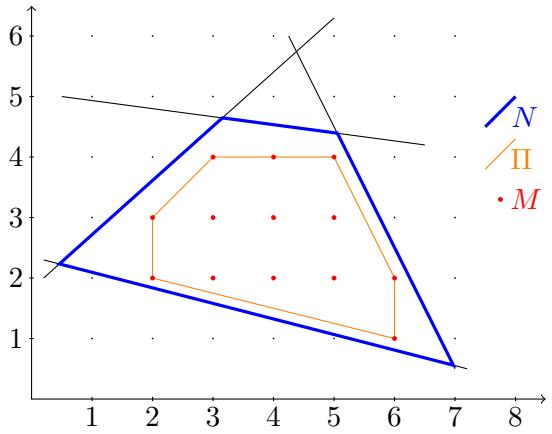
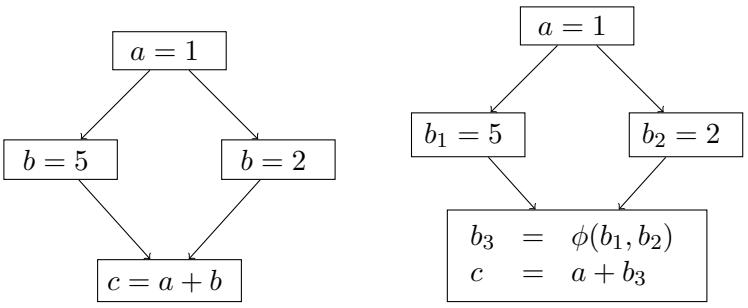


Abbildung 2.2: Zulässiger Bereich M, dessen konvexe Hülle Π und zulässiger Bereich N der LP-Relaxation.

starken gültigen oder gar facettendefinierenden Ungleichungen, kann man damit den Suchraum effektiv beschneiden.

2.3 SSA-Form

Ein Programm ist in SSA-Form, falls jede Variable nur einmal definiert wird. An Stelle von n Definitionen einer Variable treten n Variablen, so genannte abstrakte Werte, die jeweils nur einmal definiert werden. Befinden sich auf den Pfaden zu einer Benutzung einer Variable mehrere Definitionen, so ist zunächst nicht klar, welcher abstrakte Wert benutzt werden soll. Hierfür verwendet die SSA-Form einen notationellen Trick, die ϕ-Funktion. Sie wird an den iterierten Dominanzgrenzen der entsprechenden Definitionen platziert (siehe [5]).



(a) Originalprogramm (b) SSA-Programm

Abbildung 2.3: Einfaches Beispiel für die Transformation in die SSA-Form

2.4 Entstehung von Kopien

Im Folgenden soll kurz aufgezeigt werden, wie durch einen Übersetzer Kopien in einem Programm entstehen können. Die folgenden zwei Beispiele sind die wichtigsten und werden später noch anhand unseres Modells detaillierter erklärt.

2.4.1 SSA-Abbau

Da für ϕ-Funktionen kein äquivalenter Hardwarebefehl existiert, ist es Aufgabe des SSA-Abbaus, die ϕ-Funktionen zu entfernen, und durch andere Befehle zu simulieren. Im Wesentlichen werden dabei am Ende von Grundblöcken, die direkte Steuerflussvorgänger des ϕ-Grundblocks sind, Kopien der ϕ-Argumente erzeugt und eingefügt. Damit wird sichergestellt, dass beim Eintritt in den ϕ-Grundblock alle Argumente der ϕ-Funktion in derselben physischen Ressource, sprich Register, gehalten werden. Falls es möglich ist, manchen Argumenten dasselbe Register wie dem Ergebnis der ϕ-Funktion zuzuweisen, werden die entsprechenden Kopien natürlich überflüssig.

2.4.2 Registerbedingungen

Auch bei der Behandlung von Registerbedingungen können Kopien entstehen, falls die Werte vor einem Befehl in die richtigen Register umkopiert werden, um die Bedingungen zu erfüllen. Es gibt sogar Kopien, auf die man nicht verzichten kann. Gegeben sei zum Beispiel das folgende lineare Codefragment, in dem zwei Befehle vorhanden sind, die ihr Ergebnis immer in einem bestimmten Register ablegen.

$$\begin{aligned} x &= op(a, b) \\ &\vdots \\ y &= op(c, d) \\ z &= x + y \end{aligned}$$

Interferieren diese Ergebnisse wie im Beispiel, so ist es aus Korrektheitsgründen notwendig, spätestens vor dem zweiten Befehl eine Kopie des Ergebnisses des ersten zu erstellen, damit der Wert nicht verloren geht. In diesem konkreten Fall ist die Kopie unverzichtbar und nicht zu optimieren. Platziert man jedoch pessimistisch Kopien für alle Registerbedingungen, um die Korrektheit zu gewährleisten, ist Optimierungspotenzial vorhanden.

3 Verwandte Arbeiten

In diesem Kapitel werden die bisherigen Ansätze zum SSA-Abbau und der Kopienminimierung vorgestellt. Es werden, soweit vergleichbar, Vor- und Nachteile der Arbeiten und ihr Zusammenhang aufgezeigt.

Ron Cytron et al. beschreiben in „Efficiently computing static single assignment form and the control dependence graph“ [5] hauptsächlich, wie man Programme in SSA-Form überführen kann. Sie verwenden dazu das Konzept der Dominanzgrenzen, um eine so genannte *Minimal SSA Form* zu konstruieren. Ein sehr kleiner Teil der Arbeit befasst sich mit dem SSA-Abbau. In Kapitel 7 wird ein naiver Algorithmus erwähnt, der jede k-stellige ϕ-Funktion durch k Kopieroperationen in den entsprechenden Steuerflussvorgänger ersetzt. Die Autoren räumen ein, dass dadurch sehr viele unnütze Kopien entstehen. Die Entfernung von unnötigen Kopien überlassen sie der Entfernung toten Codes (dead code elimination) und der Verschmelzungsphase (coalescing) während der Registerzuteilung, die im nächsten Abschnitt behandelt werden.

1998 zeigen Briggs, Cooper et al. [6] anhand zweier Beispielklassen, dass der Abbaualgorithmus aus [5] in manchen Situationen fehlerhaften Code erzeugt und ergänzen ihn entsprechend. Durch die in dieser Arbeit vorgestellten *semi-pruned*- und *pruned*-SSA-Formen werden zwar die Anzahl der ϕ-Funktionen reduziert, und somit potenziell auch die Anzahl der Kopien, doch der Abbaualgorithmus wurde nur in Punkto Korrektheit abgeändert. Es werden weiterhin alle Kopien platziert.

3.1 Verschmelzung während der Registerzuteilung

Dieser Abschnitt beschreibt diverse Verschmelzungsverfahren, die im Laufe der Jahre entwickelt wurden. Sie sind nicht auf Kopien aus dem SSA-Abbau beschränkt, da sie alle in graphfärbenden Registerzuteilern eingebettet sind, die 1981 erstmals von Chaitin [7] vorgeschlagen wurden. Bei dieser Art der Registerzuteilung wird ein Interferenzgraph gefärbt, indem jeder Knoten einer Lebenszeit und jede Kante einer Interferenz zwischen den Lebenszeiten entspricht.

Zur Kopienminimierung schlägt Chaitin vor, die zwei Knoten einer Kopieroperation im Interferenzgraph zu verschmelzen, sofern die Lebenszeiten nicht interferieren. Somit erhalten die beiden Knoten gleiche Farben, Quell- und Zielregister sind also identisch, und die Kopie kann eliminiert werden. Solch eine Verschmelzung kann für die Färbbarkeit eines Graphen sowohl positive als auch negative Auswirkungen haben. Mit dem Verschmelzen der Knoten geht auch die Vereinigung der Kantenmengen beider Knoten einher. Dies kann einen k-

färbbaren Graphen nicht k-färbbar machen, kann aber auch den Grad der Nachbarknoten reduzieren, was weitere Verschmelzungen ermöglichen kann. Abbildung 3.1 verdeutlicht beide Fälle.

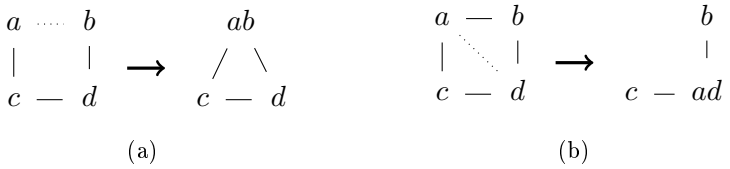


Abbildung 3.1: Negativer (a) und positiver (b) Einfluss der Verschmelzung

3.1.1 Aggressives Verschmelzen

Chaitins Originalverfahren [7] verschmilzt alle nicht interferierenden Knotenpaare von Kopien, was ihm den Namen *aggressive coalescing* einbringt. Unter dem Gesichtspunkt der Kopienminimierung führt dies zu sehr guten Resultaten. Die negativen Auswirkungen auf die Färbbarkeit werden aber komplett ignoriert, was in der Praxis dazu führt, dass der Registerbedarf über das Limit wächst und somit mehr Auslagerungen nötig werden.

3.1.2 Konservatives Verschmelzen

In „Improvements to Graph Coloring Register Allocation“ [8] stellen Briggs, Cooper und Torczon ein neues Färbeverfahren (optimistic coloring) und ein verbessertes Auslagerungsverfahren vor, das Rematerialisierung verwendet. Die Transformation für die Rematerialisierung fügt Kopien in den Code ein, die eine Lebenszeit in mehrere kürzere unterteilt. Da diese Kopien für die Optimierung essenziell sind, aber von einem aggressiven Verschmelzungsalgorithmus entfernt würden, schlagen die Autoren *conservative coalescing* vor. Dieses verschmilzt zwei Lebenszeiten nur dann miteinander, falls die sich daraus ergebende Lebenszeit nicht ausgelagert werden muss. Dies hat den positiven Effekt, dass die Färbbarkeit der Graphen nicht verschlechtert wird und die speziellen Kopien erhalten bleiben, welche die Rematerialisierung ermöglichen. Da die Entscheidung, ob der neue Knoten danach ausgelagert werden muss, aufgrund einer Abschätzung getroffen wird, entfernt dieses Verfahren alleine nur relativ wenig Kopien. Deshalb wird es mit *biased coloring* kombiniert, das während dem Färben probiert, Quelle und Ziel einer Kopie gleich einzufärben. Diese Kombination wird nur auf die speziellen Kopien angewandt. Alle anderen Kopien werden aggressiv verschmolzen. Somit soll konservatives Verschmelzen primär das Verschmelzen von bestimmten Kopien verhindern und eignet sich deshalb nur bedingt zum Leistungsvergleich.

3.1.3 Iteriertes Verschmelzen

George und Appel [9] beschrieben *iterated coalescing* und verzichteten komplett auf aggressives Verschmelzen, weil dies mit ihren teilweise vorgefärbten

Graphen zu unfärbbaren Graphen führen kann. Iteratives Verschmelzen entfernt mehr Kopien durch iteriertes Ausführen von konservativem Verschmelzen. Dabei werden die Knoten des Graphen unterteilt in kopienbezogene und nicht-kopienbezogene. Das Verfahren wird abwechselnd mit der Vereinfachungsphase ausgeführt, die bei Chaitin erst nach dem Verschmelzen ausgeführt wurde. Diese entfernt jetzt aber nur noch nicht-kopienbezogene Knoten aus dem Graphen. Somit wird der Grad von kopienbezogenen Knoten reduziert, was dem konservativen Verschmelzen größeren Spielraum eröffnet. Verschmolzene Knoten, die keine weiteren kopienbezogenen Nachbarn mehr haben, wechseln in die Gruppe der nicht-kopienbezogenen Knoten. Ebenso alle Knoten, bei denen ein Verschmelzen nicht mehr möglich ist. Danach kann eine wiederholte Ausführung der Vereinfachungsphase weitere Knoten entfernen. Somit eliminiert dieses Verfahren mindestens so viele Kopien wie konservatives Verschmelzen und erhöht den Registerbedarf ebenfalls nicht.

3.1.4 Wiederholtes Verschmelzen

Repeated coalescing, ein weiterer Vertreter der konservativen Verschmelzungsalgorithmen, stellen Dinechin et al. in „Code Generator Optimizations for the ST120 DSP-MCU Core“ [10] vor. Sie verbessern das iterierte Verschmelzen von George und Appel dadurch, dass nach jeder Verschmelzung der Interferenzgraph exakt aktualisiert wird. Dies kann in manchen Fällen dazu führen, dass eine weitere Verschmelzung erst ermöglicht wird, was bei einem pessimistischen Übernehmen aller Interferenzkanten eventuell nicht möglich gewesen wäre. Ihren Messungen an zwei größeren Programmen zufolge führt dies zu ca. 14% weniger Kopien, verglichen mit iteriertem Verschmelzen.

3.1.5 Optimistisches Verschmelzen

Das letzte hier vorgestellte Verfahren, *optimistic coalescing*, ist von Park und Moon [11] veröffentlicht worden. Es basiert auf dem aggressiven Verschmelzen nach Chaitin. Dies wird zunächst unverändert auf den Graphen angewandt. Wird während des Färbens festgestellt, dass ein verschmolzener Knoten ausgelagert werden müsste, wird die Verschmelzung rückgängig gemacht. Sind beide Knoten auch dann nicht färbbar, müssen beide ausgelagert werden. Ist genau einer färbbar, wird ihm eine Farbe zugeteilt und der andere ausgelagert. Sind beide färbbar, wird vorerst nur einem eine Farbe zugeteilt, da die Sortierung der restlichen Knoten in der Färbereihenfolge von einem Knoten ausging. Dieses Verfahren nutzt wieder massiv die Vorteile der Verschmelzung aus und verhindert die negativen erst dann, wenn sie wirklich auftreten würden. Dies ist in etwa vergleichbar mit Chaitins pessimistischer und Briggs optimistischer Färbung des Graphen.

3.2 Minimierung während des SSA-Abbaus

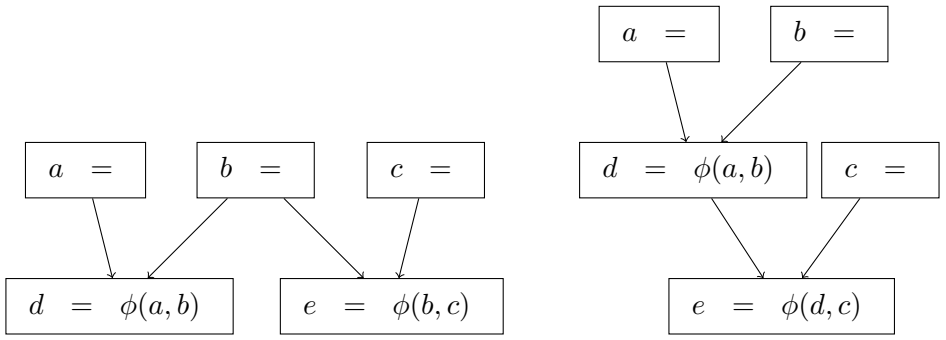
Einen anderen Weg gehen Sreedhar et al. [12] mit „Translating out of SSA Form“. Sie befassen sich nur mit Kopien aus dem SSA-Abbau und minimieren deren

Anzahl schon während des Abbaus. Ein zentraler Begriff in dieser Arbeit ist die *ϕ-Kongruenzklasse*. Dazu zunächst zwei Definitionen:

Definition 3.1 (ϕ-verbunden). Abstrakte Werte x und y heißen ϕ-verbunden, falls x und y in derselben ϕ-Funktion referenziert (benutzt oder definiert) werden. Die entsprechende Relation heiße ϕ-verbunden(a, b).

Definition 3.2 (ϕ-Kongruenzklasse). Ein abstrakter Wert x gehört zur ϕ-Kongruenzklasse $\phi_{\simeq}(x)$, die als reflexive und transitive Hülle von ϕ-verbunden(x, x) definiert ist.

Eine ϕ-Kongruenzklasse für einen Wert x_0 kann man auf folgende Art konstruieren: Ist x_0 weder Ergebnis noch Argument einer ϕ-Funktion so gilt $\phi_{\simeq}(x_0) = \emptyset$. Andernfalls wiederhole folgende Schritte für die Menge $K = \{x_0\}$, bis sich diese nicht mehr ändert. Ist ein $x \in K$ Ergebnis einer ϕ-Funktion, so füge alle Argumente dieser zu K hinzu. Falls $x \in K$ Argument einer ϕ-Funktion ist, erweitere die Menge um deren Ergebniswert. Danach gilt $\phi_{\simeq}(x_0) = K$. Abbildung 3.2 zeigt verschiedene ϕ-Kongruenzklassen.



(a) Kopplung über gemeinsames Argument b (b) Kopplung über eine ϕ-Funktion

Abbildung 3.2: Die Menge $\{a, b, c, d, e\}$ bildet jeweils eine ϕ-Kongruenzklasse.

Weiterhin unterscheiden die Autoren zwei SSA-Formen: Konventionelles und transformiertes SSA (CSSA bzw. TSSA). CSSA erhält man durch Cytrons Aufbaualgorithmus [5]. In CSSA interferieren zwei Werte aus einer ϕ-Kongruenzklasse niemals. Anders in TSSA, das aus CSSA durch Optimierungen wie Codeverschiebung (code motion) entsteht. Diese Transformationen können dazu führen, dass innerhalb einer ϕ-Kongruenzklasse Werte interferieren. Der vorgestellte Algorithmus transformiert zunächst TSSA nach CSSA, indem die Kopien platziert werden. Diese Kopien sorgen dafür, dass die Interferenzen aus den ϕ-Klasse nach außen, auf Quelle und Ziel der jeweiligen Kopie, verlagert werden. Es sei noch erwähnt, dass alle Kopien, die dieses Verfahren platziert, nicht von einem der obigen Verschmelzungsverfahren entfernt werden können, da Quelle und Ziel immer interferieren. Danach schließt sich eine Phase an, die einige spezielle dieser Kopien wieder entfernen kann. Da nun CSSA vorliegt, können alle Werte einer ϕ-Kongruenzklasse einer einzigen Ressource zugeordnet werden.

3.3 Einbeziehung von Registerbedingungen

In diesem letzten Abschnitt sollen noch zwei Veröffentlichungen Erwähnung finden, die SSA-Abbau in Zusammenhang mit Registerbedingungen betrachten.

Leung und George [13] transformieren Maschinencode für Optimierungen in SSA-Form und wieder zurück. Dabei erhalten sie die Zuordnung von Werten zu bestimmten Registern. Ihr Abbaualgorithmus sucht zunächst alle Registerbedingungen und fixiert die Werte auf die entsprechenden Register. Danach werden alle Stellen identifiziert, bei denen ein Register zu früh überschrieben würde, falls man der ϕ-Klasse mit internen Interferenzen eine Ressource zuweisen würde. In der letzten Phase werden diese Konflikte mit Kopien repariert und die ϕ-Funktion abgebaut. Dabei werden nur für die Argumente einer ϕ-Funktion Kopien platziert, die nicht auf dasselbe Register fixiert sind, wie das Ergebnis der ϕ-Funktion. Durch die Fixierung stehen die Register für die restlichen Werte ohnehin schon fest.

Rastello et al. [14] greifen diese Ideen in „Optimizing the translation out-of-SSA with renaming constraints“ auf und verbessern sie, indem die Benutzung von Werten durch ϕ-Funktionen klarer definiert werden. Des Weiteren korrigieren sie den Algorithmus, der in seiner Originalversion in manchen Situationen Werte fehlerhaft fixierte. Der wirkliche Fortschritt der Arbeit findet aber beim SSA-Abbau statt. Sie definieren das Problem *Global Pinning* wie folgt: Fixiere möglichst viele Variablen, die in einer ϕ-Funktion benutzt werden auf dieselbe Ressource, ohne dabei die Anzahl der Reparaturkopien zu erhöhen. Nach einem Beweis der NP-Vollständigkeit von *Global Pinning* schlagen sie einen Algorithmus für *Local Pinning* (ebenfalls NP-vollständig) vor. Dieser löst das etwas einfachere Problem, indem er sequenziell für alle Grundblöcke die darin enthaltenen ϕ-Funktionen gierig (greedy) optimiert.

3.4 Optimale Registerzuteilung

Einen komplett anderen Ansatz verfolgen Goodwin und Wilken [15]. Sie formulieren Registerzuteilung als ein mathematisches Optimierungsproblem. Ihr Modell umfasst Auslagerung, Rematerialisierung, Aufbrechen von Lebenszeiten (live range splitting) sowie Kopienminimierung und Registerbedingungen.

Ihr Analysemodul erzeugt für jede Entscheidung, die getroffen werden kann, eine binäre Variable im Optimierungsproblem. Zu diesen Entscheidungen gehören, ob an einem bestimmten Punkt im Programm ein virtuelles Register einem physischen zugeteilt wird, ob aus- oder eingelagert wird und so weiter. Für alle Lade-, Speicher-, Kopieneliminierungs- und Rematerialisierungvariablen werden Kosten ermittelt, die anfallen, falls die entsprechende Variable den Wert 1 annimmt. Diese Kosten werden dann durch das Lösungsmodul minimiert. Die Kopienminimierung wird mit negativen Kosten modelliert. Falls durch ein Verschmelzen Auslagerungen impliziert werden, fallen für die Auslagerungen (höhere) Kosten an. Somit können durch geeignete Wahl der Auslagerungs- und Kopieneliminierungskosten die in Abschnitt 3.1 beschriebenen, negativen Auswirkungen vermieden werden.

Fu und Wilken [16] verbessern dieses ILP-basierte Verfahren, indem sie die Formulierung des ILP verbessern. Durch Änderungen im Analysemodul identifizieren sie Entscheidungsvariablen, deren Wert in allen optimalen Lösungen schon a priori bekannt ist. Dadurch reduziert sich die Anzahl der Variablen und Nebenbedingungen erheblich. Verglichen mit den Ergebnissen von 1996 erreichen sie 2002, allein durch den Einsatz schnellerer Hardware und einer neueren Version des Lösungsmoduls, bei der Anzahl der Probleme, die in einer bestimmten Zeit gelöst werden können, eine Steigerung um den Faktor 100. Weiterhin erreichen Sie durch die bessere Modellierung einen Faktor von 150. Statistische Analysen ergaben für das neue Modell eine $O(n^{1.1})$ Beziehung zwischen Funktionsgröße und Anzahl der Nebenbedingungen sowie $O(n^{2.5})$ zwischen Funktionsgröße und Lösungszeit. Um noch absolute Zahlen zu nennen sei erwähnt, dass sie mit dieser Methode 2224 von 2399 Funktionen der SPEC92 Integer-Benchmarks innerhalb von jeweils 8 Sekunden optimal lösen konnten. Die Qualität der erzeugten Lösungen lag deutlich über denen eines graphfärbenden Zuteilers. Somit muss man ILP basierte Verfahren inzwischen als Alternative anerkennen, zumal man sie auch noch hybrid mit einer Heuristik kombinieren kann.

4 Unsere Vorgehensweise

In diesem Kapitel wird auf das Kopienminimierungsproblem in unserem SSA-basierten Codeerzeuger hingeführt, es definiert und Lösungen dafür entwickelt. Unsere Vorgehensweise wird zunächst mit neuen Erkenntnissen über SSA-Interferenzgraphen motiviert und die sich daraus ergebende Architektur kurz vorgestellt. Es folgt eine Beschreibung der einzelnen Phasen der Registerzuteilung. Die letzten Abschnitte präsentieren zwei Lösungsverfahren für das Problem.

4.1 Motivation

4.1.1 SSA-Interferenzgraphen

Interferenzgraphen von Programmen in SSA-Form sind *chordal*. Den vollständigen Beweis führt Hack in [17]. Mit einer Definition der Benutzung eines Wertes, die zwischen der Benutzung in einer ϕ-Funktion und einer normalen Benutzung differenziert, folgt auch ein neuer Lebendigkeitsbegriff für Werte in SSA-Programmen, der für die Interferenzrelation essenziell ist. Der Beweis mach davon Gebrauch und nutzt die Tatsache, dass die Dominanzrelation einen Baum bildet und zeigt, dass in SSA-Interferenzgraphen keine induzierten Kreise mit mehr als vier Knoten existieren.

Chordale Graphen lassen sich, im Gegensatz zu beliebigen Graphen, mit einem Aufwand in $O(|V| + |E|)$ minimal färben. Weitere Vorteile werden in Abschnitt 4.2, während den näheren Ausführungen, genannt. Will man diese Vorteile nutzen, liegt es nahe, die SSA-Eigenschaft von der Zwischendarstellung bis zur Registerzuteilung beizubehalten. Somit ergibt sich folgende skizzierte Backendarchitektur.

4.1.2 Backendarchitektur/Einbettung

Unser Backend schließt sich direkt an eine graphbasierte SSA-Zwischendarstellung an. Es findet kein SSA-Abbau statt. Die Befehlsauswahl erzeugt aus den Programmgraphen der Zwischendarstellung Maschinenbefehle für eine Zielarchitektur. Die resultierende Darstellung ist ebenfalls graphbasiert, in SSA-Form und enthält folglich auch noch ϕ-Funktionen. Die anschließende Befehlsanordnung sorgt für eine totale Ordnung auf den Befehlen in den Grundblöcken, wobei ϕ-Funktionen natürlich als erste Befehle eines Grundblocks gewählt werden. Die Registerzuteilung kann jetzt, auf dem immer noch in SSA-Form vorliegenden Programm, operieren. Wie auch Abbildung 4.1 zeigt, findet der SSA-Abbau

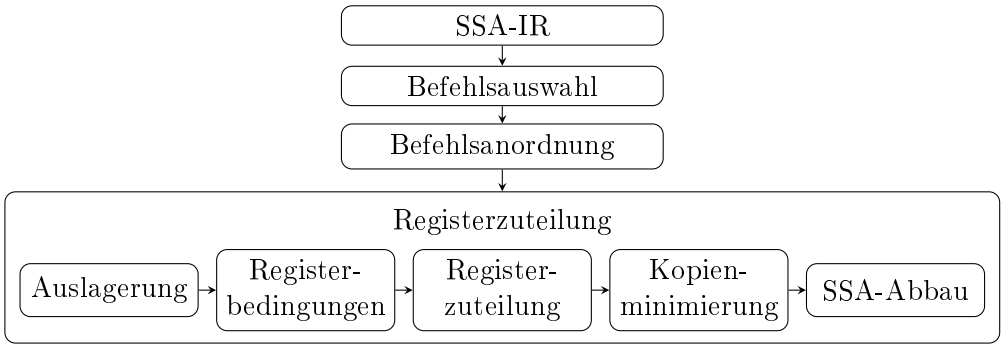


Abbildung 4.1: Backendübersicht mit SSA-Status

während der letzten Phase der Registerzuteilung statt.

4.1.3 Phi-Funktionen

Betrachtet man die übliche Notation von ϕ-Funktionen genauer, fällt einem auf, dass diese leicht irreführend ist.

Schon die Schreibweise als Funktion suggeriert dem Leser, dass die Operanden in verschiedenen Registern gehalten werden müssen. Dies ist jedoch nur für „normale“ Funktionen wie $+, -, *, \ldots$ richtig, aber nicht für ϕ-Funktionen. Da das Ergebnis einer ϕ-Funktion durch den Steuerfluss bestimmt wird, kann alleine durch die Verwendung von Werten in der ϕ-Funktion *keine* Interferenz dieser Werte entstehen[1]. Benutzungen von Werten in ϕ-Funktionen erzeugen nie Interferenzen. Falls die Operanden nicht anderweitig interferieren, will man möglichst alle in dem selben Register halten, weil die ϕ-Funktion dann äquivalent zu einer *no-op* ist.

Sind mehrere ϕ-Funktionen in einem Grundblock enthalten, werden diese gewöhnlicherweise hintereinander notiert. Dies suggeriert eine sequenzielle Ausführung der einzelnen Funktionen. Da aber alle ϕ-Funktionen eines Grundblocks simultan ausgewertet werden müssen, liegt es nahe, n ϕ-Funktionen $v_1 = \phi(a_{11}, \ldots, a_{1m}), \ldots, v_n = \phi(a_{n1}, \ldots, a_{nm})$ als eine einzige vektorwertige Φ-Funktion mit derselben Bedeutung zu notieren. Ist V die Menge der abstrakten Werte des Programms, so sei für ein $A = (a_{ij}) \in V^{n \times m}$ die Funktion Φ wie folgt definiert:

$$\Phi : V^{n \times m} \to V^n$$

$$\Phi(A) = \begin{pmatrix} \phi(a_{11}, \ldots, a_{1m}) \\ \vdots \\ \phi(a_{n1}, \ldots, a_{nm}) \end{pmatrix}$$

Aus $x = \phi(a, b), y = \phi(c, d)$ wird $\begin{pmatrix} x \\ y \end{pmatrix} = \Phi \begin{pmatrix} a & b \\ c & d \end{pmatrix}$.

[1] Die eigentliche Benutzung der Argumentwerte einer ϕ-Funktion geschieht am Ende der Vorgängerblöcke.

4.1.4 Eigenschaften von Kopien

Eine weitere Klarstellung ist für den Begriff *Kopie* nötig. Unter einer Kopie stellt man sich normalerweise die Vervielfältigung eines Wertes vor. Diese Vorstellung ist beim SSA-Abbau aber nicht immer richtig. Dort spricht man zwar meist von Kopien, doch die Eigenschaft der Vervielfältigung ist nicht immer nötig. Beim SSA-Abbau gibt es zwei Klassen von „Kopien“ die sich deutlich unterscheiden: Permutationen und Duplikate.

Die eigentliche Aufgabe ist es, das Argument einer ϕ-Funktion in einem bestimmten Register bereitzustellen. Dazu wird nicht immer eine Kopie benötigt. Eine Verschiebefunktion, oder allgemeiner, eine *Permutation* ist dazu ausreichend.

Nur in speziellen Situationen ist es nötig, dass der Wert *gleichzeitig mehrmals* vorhanden ist, also *Duplikate* erstellt werden müssen.

Werte, für welche beim SSA-Abbau ein Duplikat erstellt werden muss, besitzen ein einheitliches Identifizierungsmerkmal, wenn man sich das SSA-Programm leicht transformiert vorstellt. Man fügt zunächst in jeden Grundblock für alle Werte, die am Blockanfang lebendig sind, *primitive ϕ-Funktionen* der Form $v^* = \phi(v, \ldots, v)$ ein. Somit sind alle Werte, die in einem Grundblock benutzt werden, durch ϕ-Funktionen definiert. Notiert man dieses Programm mit Φ-Funktionen anstelle von ϕ-Funktionen, lassen sich nötige Duplikate leicht erkennen.

Für jeden Wert, der in einem Spaltenvektor $A_i = (a_{1i}, \ldots, a_{ni})^T$ einer Φ-Funktion mehrmals vorkommt, entstehen Duplikate. Da nämlich alle Ergebnisse der Φ-Funktion paarweise interferieren, müssen allen Ergebnissen unterschiedliche Register zugewiesen werden. Falls ein Wert x in einem A_i k mal vorkommt, impliziert dies aber auch, dass ihm k verschiedene Register zugewiesen werden müssen. Da dies nicht möglich ist, müssen $k-1$ Duplikate von x im i-ten Steuerflussvorgänger eingefügt werden.

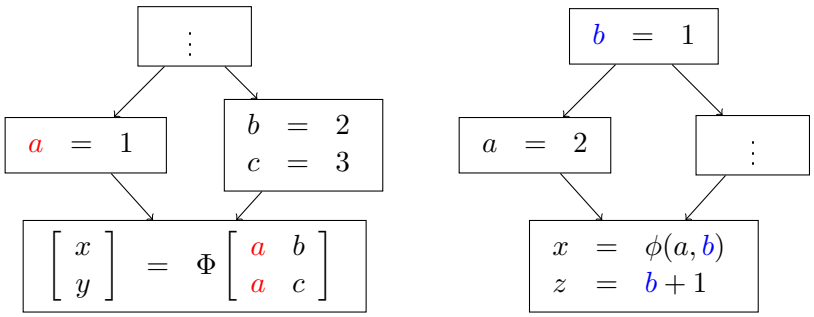
Bei konventioneller Notation, muss man anstatt dessen überprüfen, ob ein Wert in mehreren ϕ-Funktionen eines Grundblocks an gleichen Positionen der Argumentlisten benutzt wird. Darüber hinaus ist zu überprüfen, ob ein Argument einer ϕ-Funktion an deren Programmstelle lebendig ist, denn durch Einfügen der primitiven ϕ-Funktionen wird dieser Fall in der $\Phi-$Notation automatisch mit dem ersten Fall der Mehrfachbenutzung abgedeckt. Abbildung 4.2 verdeutlicht diese Zusammenhänge nochmals.

Es sei ausdrücklich darauf hingewiesen, dass im zweiten Fall die Lebendigkeit eines Arguments an der Stelle der ϕ-Funktion ausschlaggebend ist. Die Interferenz zwischen ϕ-Argument und ϕ-Funktion ist zwar notwendig aber nicht hinreichend, wie Abbildung 4.3 zeigt[2].

4.2 Phasen der Registerzuteilung

In diesem Abschnitt werden alle wichtigen Phasen unseres Registerzuteilers vorgestellt. Dabei liegt der Schwerpunkt auf der Kopienminimierung und dem SSA-

[2] Da die Rückwärtskante kritisch ist, findet die Registerpermutation in dem Grundblock statt, der diese kritische Kante entfernt.



(a) Mehrfachverwendung von a (b) Lebendigkeit von b bei x

Abbildung 4.2: Duplikate durch Mehrfachverwendung und Lebendigkeit eines Arguments an der ϕ-Funktion.

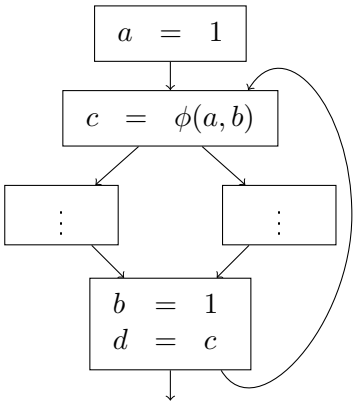


Abbildung 4.3: b und c interferieren, aber ein Duplikat ist nicht nötig.

Abbau. Die anderen Phasen (Auslagerung, Registerbedingungen und Registerzuteilung) werden nur skizziert, aber ihr Zusammenhang mit der Kopienminimierung klar herausgearbeitet.

4.2.1 Voraussetzungen

Unsere Algorithmen verwenden Ergebnisse von diversen Standardanalysen. Dazu gehören zum Beispiel der Aufbau des Dominatorbaums, der für die Färbung wichtig ist und auch für Interferenzaussagen herangezogen werden kann. Die Konstruktion des Interferenzgraphen setzt eine Lebendigkeitsanalyse voraus. Informationen über die Schleifenschachtelung ermöglichen eine sinnvolle Gewichtung der Kopien. Der SSA-Abbau wird durch die Entfernung von kritischen Kanten[3] vereinfacht.

4.2.2 Auslagerung

Die Chordalität der Interferenzgraphen erlaubt es, die Auslagerungsphase vor dem eigentlichen Färben des Graphen durchzuführen. Somit entfällt der für frühere, graphfärbende Registerzuteiler typische iterative Ablauf aus Färbeversuch und Auslagerung komplett. Alle Phasen unseres Zuteilers werden genau einmal sequenziell durchlaufen.

Eine direkte Verschränkung von Kopienminimierungsphase und Auslagerungsphase besteht nicht. Jedoch kann die Auslagerungsphase die Problemstellung für die spätere Kopienminimierung verändern. So können während der Auslagerung neue ϕ-Funktionen entstehen, falls ein Wert auf mehreren Ausführungspfaden lebendig ist und auf einem dieser ausgelagert wird. Wird er dann wieder zurückgeladen, entsteht ein neuer abstrakter Wert, denn in SSA-Form darf jeder Wert nur einmal definiert werden. Da nun mehrere Werte für dieselbe Variable vorhanden sind, müssen an manchen Punkten, wo sich Steuerflusspfade vereinigen, ϕ-Funktionen platziert werden.

Aber auch eine Reduzierung von ϕ-Funktionen ist möglich. Falls das Ergebnis einer ϕ-Funktion ausgelagert wird, werden die Argumente nicht in einem Register sondern im Hauptspeicher zusammengeführt. An die Stelle von Register-zu-Register-Kopien treten dann Speicherbefehle.

4.2.3 Registerbedingungen

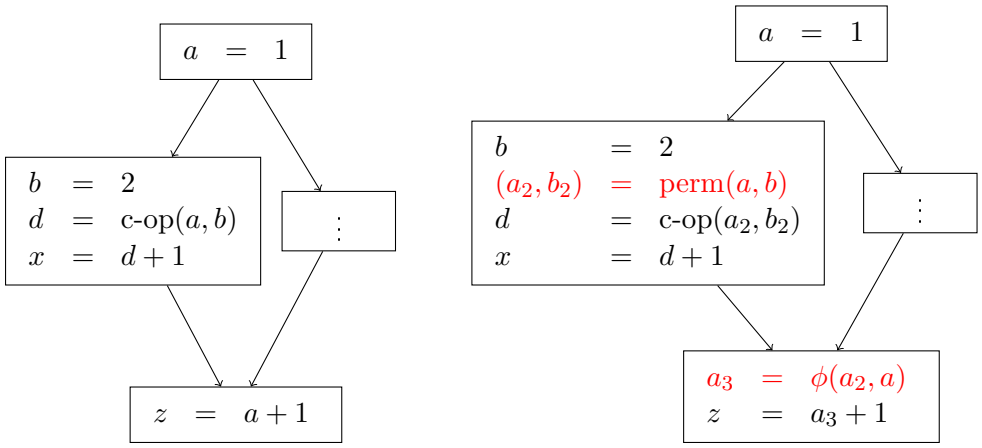
Die Behandlung von Registerbedingungen kann durch eine *Vorfärbung* des Interferenzgraphen modelliert werden, was aber zwei Nachteile mit sich bringt:

1. Um die Korrektheit der Registerzuteilung zu gewährleisten, ist trotzdem noch ein weiterer Mechanismus nötig: Für interferierende Werte, die durch Registerbedingungen in dasselbe Register gezwungen werden, müssen Kopien eingefügt werden, um ein Überschreiben des einen Wertes zu verhindern.

[3] Eine kritische Kante verbinden zwei Grundblöcke B_1, B_2, wobei B_1 mehrere Nachfolger und B_2 mehrere Vorgänger hat.

2. Selbst für chordalen Graphen ist das minimale Färben von vorgefärbten Graphen NP-vollständig.

Es existiert aber noch eine weitere, einheitliche Möglichkeit, die Registerbedingungen zu erfüllen. Könnte man garantieren, dass während des normalen Färbeprozesses an allen Stellen mit Registerbedingungen jeweils alle Farben zur Verfügung stehen, könnte man die Farben frei wählen. Genau dies erreichen wir durch spezielle *Perm-Funktionen*. Diese werden direkt vor jedem Befehl mit Registerbedingungen platziert und führen eine beliebige Permutation der Registerbelegung durch. Die Argumente einer solchen Perm-Funktion sind alle an dieser Stelle lebendigen Werte. Als Ergebnis liefert sie genauso viele neue Werte, denen aber andere Register zugewiesen werden können. Alle Benutzungen der Originalwerte werden durch Benutzungen der Ergebniswerte ersetzt.



(a) Vorher: c-op ist mit Registerbedingungen behaftet.

(b) Nachher: a_2, b_2 und/oder d können jetzt immer die richtigen Register zugewiesen werden.

Abbildung 4.4: Einfügen einer Perm-Funktion wegen Registerbedingungen.

Abbildung 4.4 zeigt die Veränderungen am Programm, die das Einfügen einer Perm-Funktion mit sich bringt. Für den Interferenzgraph des Programms bedeutet das Einfügen einer Perm-Funktion das Einfügen neuer Knoten und ein Durchtrennen aller Interferenzkanten an dieser Stelle, da alle Benutzungen der Originalwerte ersetzt wurden. Genau dieses Durchtrennen sorgt dafür, dass im normalen Färbeprozess die Farben für die Ergebnisknoten einer Perm-Funktion frei gewählt werden können. Denn es existiert eine perfekte Eliminationsordnung, welche diese Knoten am Ende enthält und somit zuerst gefärbt werden können. Wenn man sie zuerst färben kann, sind aber noch keine Farben durch interferierende Werte verbraucht und man kann die Farben frei wählen.

Jede Perm-Funktion ist auf Maschinenebene durch Kopien und/oder Vertauschungen von Registern auszudrücken. Zudem können, wie in der Auslagerungsphase auch, durch das Einfügen neuer Werte ϕ-Funktionen entstehen. Dies erhöht die Anzahl der zu optimierenden Kopien.

4.2.4 Registerzuteilung

Die eigentliche Zuteilung der Register kann grundblocklokal geschehen, wenn man die Blöcke in Dominatorreihenfolge abarbeitet. Für einen Grundblock können Register erst zugeteilt werden, wenn dies für seinen unmittelbaren Dominator erledigt ist. So können die Register von Werten, die beim Eintritt in einen Grundblock lebendig sind, als benutzt markiert werden. Die Bestimmung einer perfekten Eliminationsordnung unter Berücksichtigung der Registerbedingungen liefert eine minimale Färbung für jeden Grundblock. Da die Auslagerungsphase zuvor den Registerbedarf auf die zur Verfügung stehenden Register gesenkt hat, ist diese minimale Färbung auch eine gültige Registerzuteilung.

4.2.5 Kopienminimierung

In diesem Abschnitt wird das Problem der Kopienminimierung möglichst exakt beschrieben und abgegrenzt. Ausführliche Betrachtungen der Komplexität und Lösungsstrategien folgen in den späteren Abschnitten 4.3 bis 4.5 ab Seite 28.

Kopien entstehen in unserem Modell nur durch die Behandlung von Registerbedingungen und den SSA-Abbau. Jede Kopie verursacht nicht negative Kosten in einer bestimmten Höhe und kann als ein Tripel (Argumentwert, Ergebniswert, Kosten) $\in V \times V \times \mathbb{N}$ gesehen werden. Bei einer Perm-Funktion mit n Argumenten ergeben sich n Kopien mit n unterschiedlichen Ergebniswerten. Anders bei ϕ-Funktionen: Zwar ist jedes Argument der ϕ-Funktion auch Argument einer Kopie, aber der Ergebniswert jeder Kopie ist identisch mit dem Ergebnis der ϕ-Funktion.

Für die Berechnung der Kopienkosten können beliebige Strategien Verwendung finden. Um die Kosten für eine Kopie zu berechnen, reichen Argument- und Ergebniswert und bei ϕ-Funktionen zusätzlich die Argumentposition als Information aus. Das einfachste Modell weist jeder Kopie die Kosten 1 zu. Ein realistischeres Modell berücksichtigt die Schachtelungstiefe der Grundblöcke in Schleifen. Bei ϕ-Funktionen muss man für die Berechnung der Kosten die Schachtelungstiefe des Vorgängerblocks verwenden, und nicht die des ϕ-Grundblocks selbst. Sind sogar Profilinformationen vorhanden, kann man diese ebenfalls mit einbringen.

Zur Veranschaulichung des Problems kann man den Interferenzgraphen um einen zusätzlichen, gewichteten Kantentyp Q erweitern. Diese Kanten verbinden die Knoten, die den Argument- und Ergebniswerten einer Kopie entsprechen. Im Kontext der Kopienminimierung sind sie als Gleichfärbewünsche zu interpretieren, da bei *Gleichfärbung* die Kopie entfällt. Durch die Kanten aus Q entstehen bei ϕ-Funktionen im Graphen somit v-ähnliche- oder Sternstrukturen und bei Registerbedingungen einzelne Kanten. Durch Kombination dieser Muster, zum Beispiel durch ϕ-Klassen können theoretisch beliebig komplexe Muster (größere Zusammenhangskomponenten bezüglich Kanten aus Q) entstehen. Abbildung 4.5 zeigt ein einfaches Beispiel einer gewichteten Kopienminimierung.
Nun lassen sich die Kopienminimierungsprobleme in unserem Kontext definieren. Eine Verallgemeinerung auf nicht chordale Graphen ist ohne weiteres möglich, für diese Arbeit aber irrelevant.

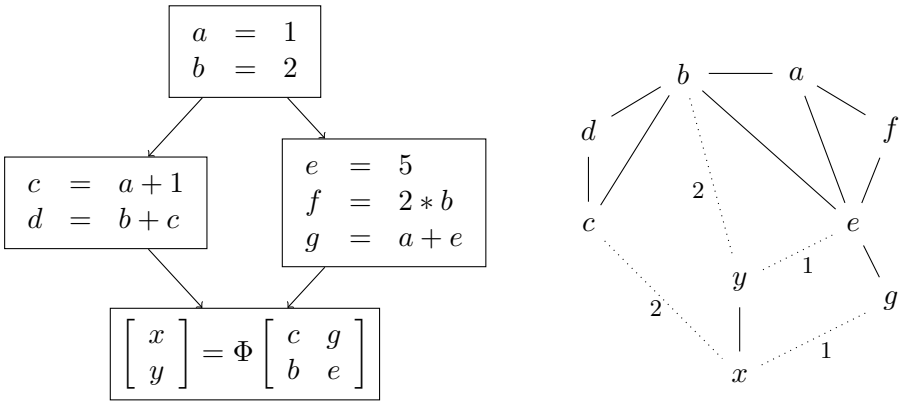


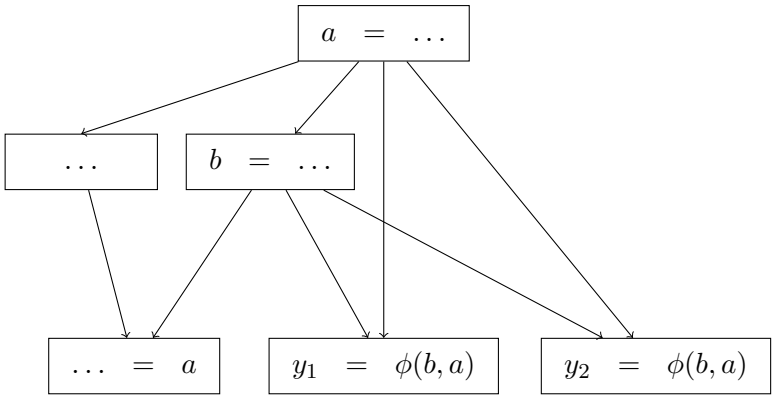
Abbildung 4.5: Beschreibung eines Kopienminimierungsproblems durch einen um gewichtete Gleichfärbungskanten erweiterten Interferenzgraphen.

Definition 4.1 (gewichtete Kopienminimierung, WCM). Gegeben sei ein chordaler, k-färbbarer Graph $G = (V, E, Q)$ mit Knotenmenge V, Interferenzkantenmenge E und Gleichfärbekantenmenge Q. Gesucht ist eine k-Färbung des Graphen mit minimalen Kosten.

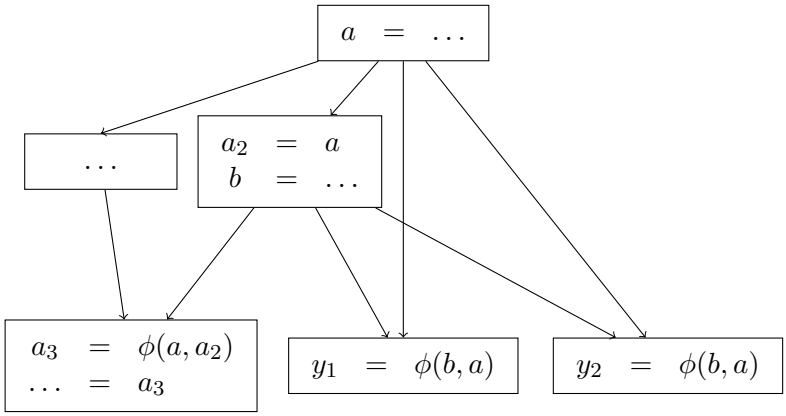
Definition 4.2 (Kardinalitätskopienminimierung, CCM). Sind alle Gewichte bei einer WCM-Instanz identisch 1, nennt man das Problem auch Kardinalitätskopienminimierung.

Es gilt möglichst viele Knotenpaare, die durch Q gegeben sind, gleich einzufärben, um die Kosten zu reduzieren. Es sei ausdrücklich erwähnt, dass keine minimale Färbung des Graphen gesucht ist, sondern nur eine, die maximal k Farben benutzt. Dies resultiert in dem gewünschten Ergebnis: Minimierung der dynamischen Anzahl an Kopien über alle Ausführungspfade eines Programms. Was damit aber nicht minimiert wird bzw. werden soll, ist die statische Anzahl der im Programm vorhandenen Kopierinstruktionen; selbst bei CCM-Modellierung. Abbildung 4.6 zeigt ein Programm, bei dem die statische Anzahl der Kopierbefehle minimiert wurde. Wechselt a vor der Definition von b das Register ($a_2 = a$), so interferieren a und b. Somit kann ihnen dasselbe Register zugewiesen werden und die Kopien für die ϕ-Funktionen entfallen. Durch diese Kopie wird aber eine weitere für $a_3 = \phi(a, a_2)$ nötig, womit insgesamt 2 Kopien eingefügt wurden. Die Optimierung dynamischer Kopien hätte ebenfalls 2 Kopien eingefügt (jeweils eine auf einer Kante zu den ϕ-Blöcken). Durch Hinzufügen von weiteren Blöcken mit $y_i = \phi(b, a)$ steigt die Anzahl der eingefügten Kopien bei Optimierung auf die dynamische Anzahl beliebig, wogegen bei Optimierung auf statische Anzahl immer genau 2 Kopien benötigt werden. Allerdings ist im letzteren Fall die Anzahl ausgeführter Kopien über alle Ausführungspfade um eins erhöht.

Der größte Vorteil der chordalen Kopienminimierung ist, dass die Auswirkungen einer Kopienelimination auf die Färbbarkeit des Graphen exakt bestimmbar sind. Wir können deshalb die k-Färbung gegenüber der Kopienminimierung priorisieren. Falls noch Register frei sind, ziehen wir diese zur Minimierung der Kopien heran, unterlassen dies aber, falls der Graph dann nicht mehr k-färbbar ist. Abbildung 4.7 zeigt optimale Lösungen von drei Probleminstanzen.

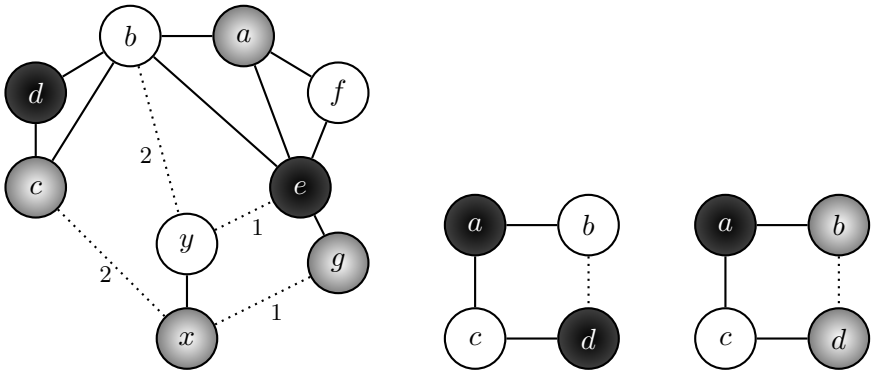


(a) Originalprogramm: Da a und b interferieren sind bei Optimierung der dynamischen Anzahl an Kopien genau 2 nötig.



(b) Optimierung der statischen Anzahl der Kopierbefehle: a und b kann dasselbe Register zugewiesen werden, da sie nicht interferieren. Hier sind insgesamt auch 2 Kopien nötig.

Abbildung 4.6: Unterschied zwischen der Optimierung von statischen und dynamischen Kopien. Der Unterschied der benötigten Kopierbefehle wird durch Hinzufügen von weiteren Blöcken $y_i = \phi(b, a)$ beliebig groß.



(a) WCM-Lösung bei $R = 3$ mit Kosten 1 (y, e) (b) CCM, $R = 2$, Kosten 1 (c) CCM, $R = 3$, Kosten 0

Abbildung 4.7: Optimale Lösungen für WCM- und CCM-Probleme bei gegebener Registeranzahl R.

4.2.6 SSA-Abbau

Die Kopienminimierung hinterlässt eine gültige Registerzuteilung, in der die Kosten der verbliebenen Kopien minimiert oder zumindest reduziert wurden. Dies gilt für die schon explizit vorhandenen Kopien in Form von Perm-Funktionen (Registerbedingungen), sowie für die erst durch den SSA-Abbau entstehenden. Deshalb muss kein optimierender SSA-Abbau verwendet werden. Es werden lediglich Permutationen und Duplikate eingefügt, und den dadurch neu entstehenden Werten die richtigen Register zugeteilt.

Im Folgenden wird immer eine Φ-Funktion $v = \Phi(A)$ im Grundblock B_Φ betrachtet. Dabei sei der Ergebnisvektor $v = (v_1, \ldots, v_n)^T$, die Argumentmatrix A bestehe aus den Spalten $A_1, \ldots, A_m$, ferner sei jedem Ergebnis v_i das Register r_i zugeteilt. Wird ein bestimmtes Argument untersucht, so sei dies $a = a_{ij}$ im Grundblock B_a mit zugeteiltem Register r_a.

Zunächst werden am Ende aller Grundblöcke, die ϕ-Argumente enthalten Perm-Funktionen eingefügt. Das ϕ-Argument a_{ij} ist genau dann Argument einer Perm-Funktion, falls es nicht live-in(B_Φ) ist[4]. Jeder Wert ist höchstens einmal Argument ein und derselben Perm-Funktion. Durch das Einfügen der Perm-Funktionen ändern sich auch die Argumente der Φ-Funktionen: Die Originalwerte werden durch die permutierten Werte (die Ergebnisse der Perm-Funktionen) ersetzt. Jedem dieser Ergebniswerte wird initial das Register zugewiesen, das ihrem entsprechenden Eingabewert zugewiesen ist. Die Perm-Funktion ist zunächst äquivalent zu einer *no-op*.

Nach den Perm-Funktionen werden nun für alle Φ-Funktionen die notwendigen Duplikate im Programm ergänzt. Dabei muss man zwei Fälle unterscheiden: Duplikate, die durch Interferenz von v_i und a_{ij} erzwungen werden und Duplikate, die durch Mehrfachverwendung in einer Φ-Funktion unvermeidbar sind.

[4] Diese Perm-Funktionen haben im Allgemeinen nicht alle an dieser Stelle lebendigen Werte als Argumente; im Gegensatz zu Perm-Funktionen, die aufgrund von Registerbedingungen eingefügt werden.

- Interferiert ein Argument a_{ij} jetzt noch mit v_i, so kann es kein Ergebnis einer Perm-Funktion sein. Das wiederum bedeutet, dass es live-in(B_Φ) ist und mit allen Ergebnissen der Φ-Funktion am Anfang von B_Φ interferiert. Deshalb ist es nötig, dass der Wert a gleichzeitig in mehreren Registern vorhanden ist. Für solche Werte a muss ein Duplikat im Block B_a eingefügt werden. Die Φ-Funktion benutzt dann dieses Duplikat anstatt dem Originalwert.

- Die zweite Quelle für Duplikate ist die mehrfache Verwendung von einem Wert a in einer Spalte A_j einer Φ-Funktion. Weil alle Ergebnisse der Φ-Funktion paarweise interferieren und somit alle Ergebnisregister unterschiedlich sind, müssen auch die Register aller Argumente in einer Spalte A_j verschieden sein. Tritt a in Spalte A_j q-mal auf, muss a in q Registern gleichzeitig vorhanden sein, womit $q-1$ Duplikate nötig sind.

  Die Φ-Funktion verwendet dann in einer Zeile den Originalwert und in den anderen Zeilen die Duplikate, sodass jedes Duplikat genau einmal benutzt wird. Die Zuordnung des Originalwerts zu einer Zeile der Φ-Funktion ist nicht beliebig. Ist nämlich eines der Register $r_1, \ldots, r_n$ identisch mit r_a, zum Beispiel $r_a = r_k$, sollte a Argument der Φ-Funktion in Zeile k sein. Andernfalls würde man ein Ergebnis der Kopienminimierung ignorieren, und später einem Ergebnis der Perm-Funktion unnötigerweise ein anderes Register zuteilen, als dem entsprechenden Argument der Perm-Funktion zugeteilt wurde. Andernfalls kann a einer beliebigen der in Frage kommenden Zeilen als Argument zugewiesen werden.

Zusammengefasst ergibt sich Folgendes:

- a ist Ergebnis einer Perm-Funktion gdw. a und v nicht interferieren.

- Wird a in A_j q-mal verwendet und interferieren a und v, so sind insgesamt q Duplikate nötig und die Zuordnung zu den Zeilen ist beliebig.

- Wird a in A_j q-mal verwendet und interferieren a und v *nicht*, so sind insgesamt $q-1$ Duplikate nötig und die Zuordnung von a ist *nicht* beliebig.

Als letzten Schritt weist man allen Argumenten von Φ-Funktionen noch Register zu. Hierbei kann man relativ einfach vorgehen: Weist man allen Werten in Zeile A_i von A das Register r_i zu, ist der SSA-Abbau beendet und die Registerzuteilung gültig. Dass die Φ-Funktionen nun überflüssig sind, ist leicht einzusehen, da den Argumenten ja gerade das Register des zugehörigen Ergebniswertes v_i zugewiesen wurde. Die Korrektheit der Registerzuteilung ergibt sich durch die Konstruktion der vorherigen Schritte und den Gegebenheiten im Steuerflussgraphen.

Beweis. Die Registerzuteilung wird nur für Duplikate und die Ergebnisse von Perm-Funktionen verändert. Diese Werte werden am Ende von B_a definiert und sind nur innerhalb von B_a lebendig. Die einzigen anderen Werte, die mit diesen Werten interferieren, sind jene, die aus dem Argumentblock B_a herausleben.

Alle diese Werte interferieren paarweise. Deshalb ist noch zu zeigen, dass nach dem SSA-Abbau jedem dieser Werte ein anderes Register zugeordnet ist.

Sei V die Menge der Werte, die am Anfang von Grundblock B_Φ lebendig sind, vereinigt mit den Ergebniswerten v_i der Φ-Funktion. Diese interferieren ebenfalls alle paarweise. Da die Registerzuteilung vor dem SSA-Abbau gültig war, haben auch nach dem SSA-Abbau alle Werte in V unterschiedliche Register. Da alle Duplikate und Ergebnisse von Perm-Funktionen nach Konstruktion immer Argumente unterschiedlicher Zeilen der Φ-Funktion sind, erhalten diese auch immer paarweise unterschiedliche Register. Und da diese Register immer identisch mit einem Ergebnisregister der Φ-Funktion sind, können dies keine Register sein, die Werten aus live-in(B_Φ) zugeteilt sind. Da live-out(B_a) = live-in(B_Φ) gilt, entstehen durch diese Registerzuteilung auch in B_a keine Konflikte. □

Um die Zusammenhänge zu verdeutlichen, zeigt Abbildung 4.8 ein Programm direkt vor und nach dem SSA-Abbau. In diesem Beispiel sind alle oben erwähnten Fälle enthalten:

- Da der Wert a bei der Φ-Funktion lebendig ist, muss ein Duplikat $\bar{a}$ erzeugt werden. Alle anderen Werte sind bei der Φ-Funktion nicht lebendig und deshalb Argumente der eingefügten Perm-Funktionen.

- Die Perm-Funktion links muss die Register anpassen, wohingegen für die Perm-Funktion rechts später keine Befehle erzeugen werden.

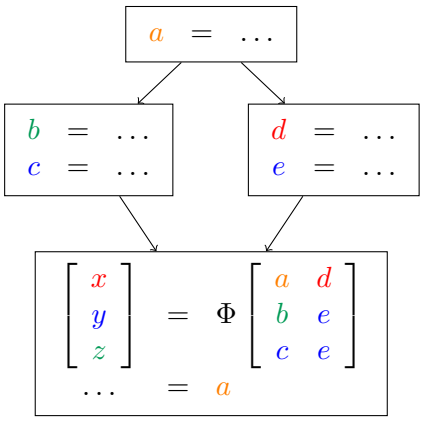
- Der Wert e wird zwei mal in einer Spalte verwendet, somit ist auch ein Duplikat $\bar{e}$ nötig. Bei der Zuordnung von e und $\bar{e}$ zu y und z ist darauf zu achten, dass e y zugeordnet wird, da hier die Register schon übereinstimmen.

Sind alle neuen Werte (Duplikate und Ergebnisse der Perm-Funktionen) platziert, werden allen Argumenten einer Zeile der Φ-Funktion das Ergebnisregister dieser Zeile zugewiesen: Zum Beispiel wird a und d das Register von x zugeteilt.

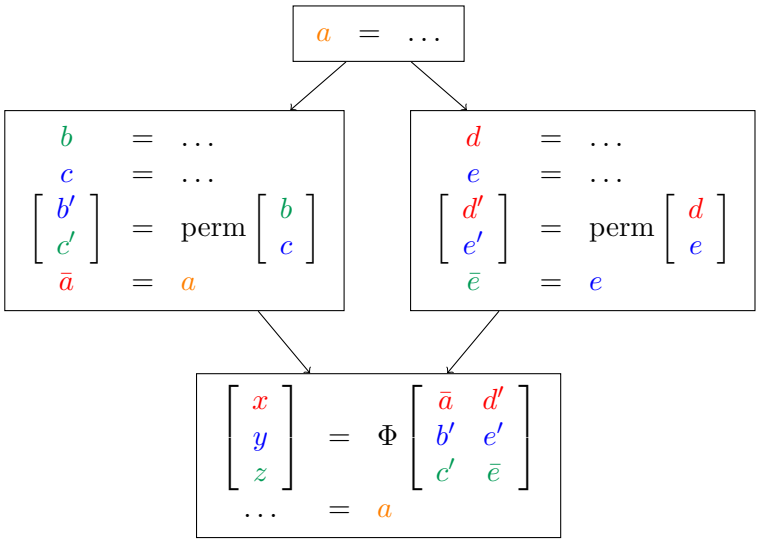
4.2.7 Perm-Abbau

Während dem SSA-Abbau entstehen Perm-Funktionen, die Werte in bestimmte Register tauschen/kopieren. Diese werden nun zusammen mit den Perm-Funktionen der Registerbedingungen durch real existierende Befehle der Zielarchitektur ersetzt.

Aus der Linearen Algebra ist bekannt, dass sich jede Permutation als eine Hintereinanderausführung von Transpositionen darstellen lässt. Diese Transpositionen lassen sich, je nach Architektur, selbst bei maximalem Registerbedarf, durch einen Tauschbefehl (`xchg`) oder drei `xor`-Befehle realisieren. Sind bei einer Perm-Funktion noch Register frei, oder sind die Mengen der Ein- und Ausgaberegister nicht identisch, kann man auch einfache Kopierbefehle (`mov`) verwenden.



(a) SSA-Programm P mit gegebener Registerzuteilung/Färbung



(b) Programm P nach dem SSA-Abbau

Abbildung 4.8: SSA-Abbau durch Einfügen von Perm-Funktionen und Duplikaten, sowie Registerzuteilung für neue Werte.

4.3 Komplexitätsbetrachtung

Das minimale Färben von beliebigen Graphen ist NP-vollständig. Chordale Graphen hingegen lassen sich in $O(|V|+|E|)$ färben. Das Gleichfärben von Knotenpaaren kommt einem Verschmelzen der Knoten gleich, was die Chordalität eines Graphen jedoch zerstören kann. Somit liegt die Vermutung nahe, dass chordale Kopienminimierung NP-vollständig ist.

4.3.1 NP-Vollständigkeit

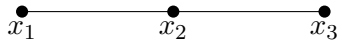
Wir zeigen dies unter Verwendung einer Idee von Rastello et al. aus [18]. Dort geben die Autoren im Anhang einen Beweis für ein ähnliches Problem an. Sie untersuchen das Problem *Global Pinning*, dass sich ohne die Originalnotation aus [18] wie folgt formulieren lässt:

Definition 4.3 (Global Pinning). Finde für eine gegebene Menge von ϕ-Funktionen und ϕ-Argumenten eine k-Färbung, die in einer minimalen Anzahl an Kopien resultiert.

Der Unterschied zu Kopienminimierung (CCM) ist die Menge der betrachteten Knoten: Die bei Global Pinning zu färbende Knotenmenge V_ϕ besteht genau aus den Resultat- und Argumentwerten von ϕ-Funktionen. Die Färbung anderer Knoten im Interferenzgraph $G = (V, E)$ wird nicht betrachtet. Deshalb kann im Allgemeinen eine k-Färbung von V_ϕ nicht zu einer k-Färbung von V ergänzt werden; siehe zum Beispiel Abbildungen 4.7(b) und 4.7(c).

Somit sind alle Global Pinning Probleme Spezialfälle von Kopienminimierung (CCM). Sind alle Werte eines Programms entweder Ergebnisse oder Argumente von ϕ-Funktionen, so sind Global Pinning und CCM äquivalent. Der Beweis aus [18] enthält jedoch leider Fehler. Deshalb wird hier die Idee von Rastello verwendet, der Beweis jedoch auf eine andere Art erbracht.

Beweis. Erfolgt durch Reduktion des Problems k-Färbung: Für einen gegebenen beliebigen Graphen G und ein $k \in \mathbb{N}$ ist das Finden einer k-Färbung NP-vollständig. Wir lösen dies, indem wir eine CCM-Instanz mit k Registern lösen. Sei deshalb $G = (V, E)$ ein ungerichteter Graph mit $V = \{x_1, \ldots, x_n\}$. Als laufendes Beispiel betrachten wir den folgenden Graphen G:



Zunächst konstruieren wir aus G ein SSA-Programm P wie folgt:

- Für alle $x_i \in V$ fügen wir dem Programm einen Grundblock B_i hinzu, der eine Definition des Wertes x_i enthält.
- Für alle $(x_i, x_j) \in E$ kommen drei weitere Grundblöcke hinzu:
  1. B_{ij}^i als Nachfolger von B_i, der eine Definition von a_{ji} enthält.
  2. B_{ij}^j als Nachfolger von B_j, der eine Definition von a_{ij} enthält.

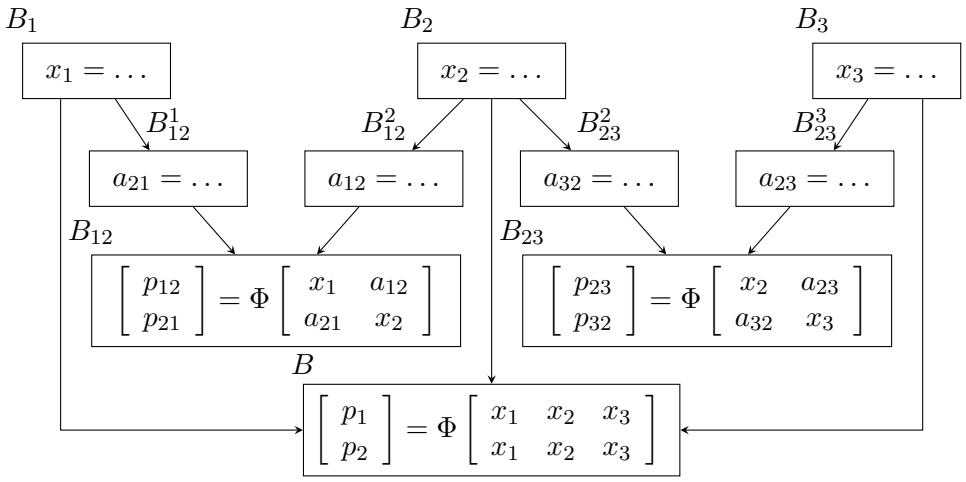
3. B_{ij} mit Vorgängern B_{ij}^i und B_{ij}^j, der eine Φ-Funktion enthält:

$$\begin{bmatrix} p_{ij} \\ p_{ji} \end{bmatrix} = \Phi \begin{bmatrix} x_i & a_{ij} \\ a_{ji} & x_j \end{bmatrix}$$

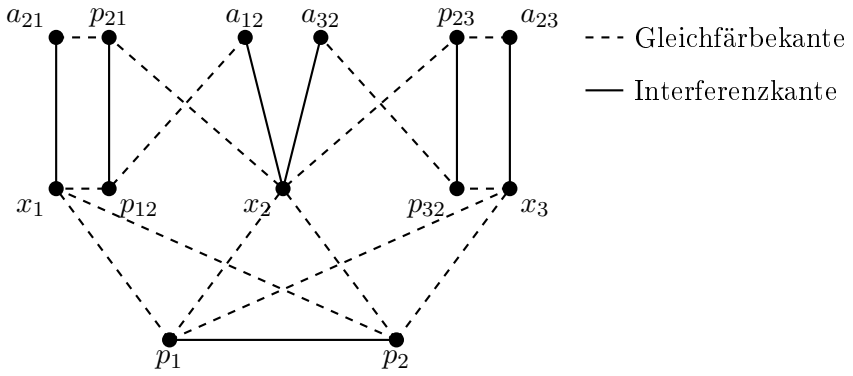
- Einen Grundblock B mit Vorgängern $B_1, \ldots, B_n$ mit Φ-Funktion

$$\begin{bmatrix} p_1 \\ \vdots \\ p_k \end{bmatrix} = \Phi \begin{bmatrix} x_1 & \cdots & x_n \\ \vdots & & \vdots \\ x_1 & \cdots & x_n \end{bmatrix}$$

Somit ergibt sich für das Beispiel folgendes Programmfragment P:



In diesem Programm sind für alle $(i,j) \in E_G$ die Interferenzenkanten (x_i, a_{ji}), (x_j, a_{ij}) und (p_{ij}, p_{ji}) vorhanden. Für $(i,j) \in E_G$ entstehen die Gleichfärbekanten (p_{ij}, x_i), (p_{ij}, a_{ij}), (p_{ji}, x_j) und (p_{ji}, a_{ji}). Dazu kommen unabhängig von E_G die Gleichfärbekanten (p_i, x_j) für alle $i, j \in \{1 \ldots n\}$ und die Interferenzkanten (p_i, p_j) für $i \neq j$. Das Beispiel erzeugt den folgenden Kopienminimierungsgraphen K:



Eine Lösung für dieses Kopienminimierungsproblem existiert genau dann, wenn die Registeranzahl $R \geq k$ ist, denn in den so konstruierten Programmen gibt es nur Interferenzcliquen der Größe 2 und eine Clique der Größe k.

Zunächst betrachten wir eine untere Schranke für die entstehenden Kosten. Da in K jedem x_i nur eine Farbe zugeordnet werden kann gibt es $k-1$ Knoten p_j, die eine andere Farbe als x_i haben. Deshalb verursacht jede optimale Lösung von CCM mindestens $n(k-1)$ Kosten.

Aus jeder k-Färbung f_G von G kann eine optimale Lösung von CCM abgeleitet werden: Man ordnet jedem $x_i \in K$ die Farbe $f_G(x_i)$ zu. Da $f_G(x_i) \neq f_G(x_j)$ für alle $(x_i, x_j) \in E_G$, kann man den Knoten x_i, p_{ij}, a_{ij} dieselbe Farbe zuweisen (Abbildung 4.9(a)). Somit können nur noch Kosten für Gleichfärbekanten (p_i, x_j) entstehen. Da diese Kosten immer mit der unteren Schranke $n(k-1)$ übereinstimmen, muss die Färbung von K optimal sein.

Färbungen f_K von K, die keiner k-Färbung in G entsprechen, haben Kosten echt größer als $n(k-1)$: Da f_K keiner k-Färbung von G entspricht, gibt es eine Kante $(x_i, x_j) \in E_G$ mit $f_K(x_i) = f_K(x_j)$. Deshalb muss $f_K(a_{ij}) \neq f_K(x_i) \neq f_K(a_{ji})$ gelten und zwei der vier Gleichfärbekanten, die adjazent zu p_{ij} und p_{ji} sind, verursachen Kosten (Abbildung 4.9(b)). □

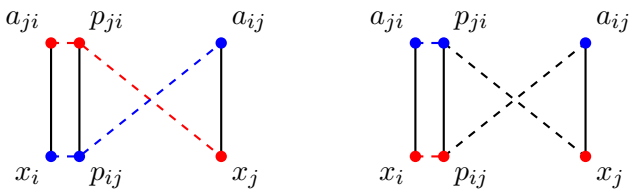


Abbildung 4.9: Färbung bei Ableitung aus einer k-Färbung von G (a) und in anderen Fällen (b).

4.4 Tauschheuristik

Beim Entwurf einer heuristischen Lösungsmethode ist es hilfreich, die auftretenden Probleme auf häufig auftretende Strukturen zu untersuchen. Dadurch kann man den Algorithmus so gestalten, dass er nicht unnötig komplexe Fälle betrachtet, die in der Praxis eher selten auftreten und andererseits durch gute Lösungen der häufigen Fälle eine insgesamt gute Lösung erzielen. Im nächsten Abschnitt präsentieren wir deswegen die Ergebnisse von diversen statistischen Analysen, auf denen unser Tauschalgorithmus beruht, der danach vorgestellt wird.

4.4.1 Statistische Analyse

Die Werte die in diesem Abschnitt präsentiert werden, wurden durch Messungen an den Programmen der SPEC2000-Benchmarks ermittelt. Für die Zielarchitektur wurden 8 Register angenommen.

Zunächst sollen einzelne ϕ-Funktionen betrachtet werden. Abbildung 4.10 zeigt die Verteilung der Argumentanzahl von ϕ-Funktionen und Abbildung 4.11

die Verteilung der Definitionsorte dieser Argumente, relativ zur jeweiligen ϕ-Funktion[5]. Eine typische ϕ-Funktion hat 2.4 Argumente, wobei 54.0% davon im jeweiligen Steuerflussvorgängerblock definiert sind. Da beim SSA-Abbau allen Argumenten einer ϕ-Funktion dasselbe Register zugewiesen werden muss, sind diese Ergebnisse günstig: Man muss im typischen Fall nur wenig Argumente gemeinsam betrachten und es besteht Grund zur Annahme, dass eine Änderung der Registerzuteilung sich nur lokal auswirkt, da die meisten Argumente im Vorgängerblock definiert werden, also wenig Interferenzen haben.

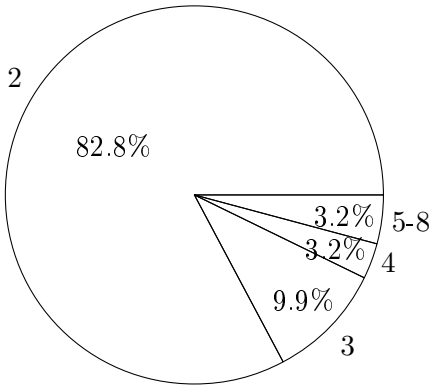


Abbildung 4.10: Verteilung der Argumentanzahl von ϕ-Funktionen

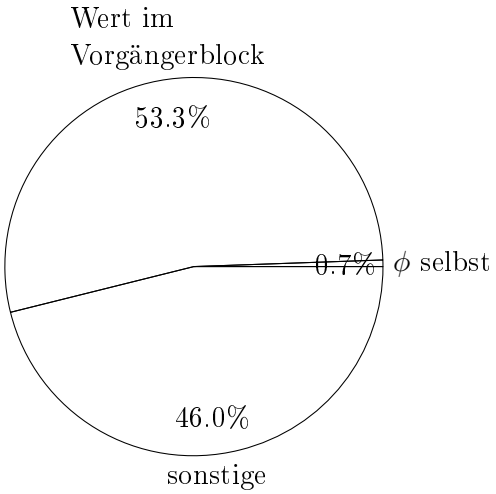
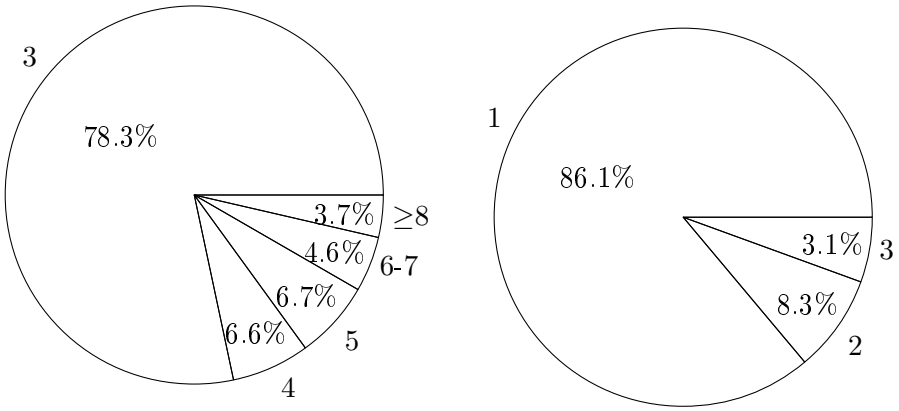


Abbildung 4.11: Verteilung der Argumenttypen von ϕ-Argumenten

Eine weitere zu klärende Frage ist, welchen Einfluss ϕ-Klassen auf die Optimierung haben. Die Verteilung der Größe einer ϕ-Klasse ist in Abbildung 4.12(a) dargestellt, wobei die Anzahl unterschiedlicher Werte als Größe bezeichnet wird. Noch aussagekräftiger ist Abbildung 4.12(b), die nur die Anzahl von ϕ-Funktionen pro ϕ-Klasse betrachtet. Da 86.1% aller ϕ-Klassen aus genau einer ϕ-Funktion bestehen, könnte man auf die Betrachtung von ϕ-Klassen als Ganzes

[5] Die Werte wurden vor der Entfernung kritischer Kanten erhoben.

verzichten und nur die ϕ-Funktionen isoliert betrachten. Dies hat den Vorteil, dass man die ϕ-Klassen nicht bestimmen muss und die Heuristik leichter implementierbar ist.



(a) Verteilung der ϕ-Klassengröße (Werte pro ϕ-Klasse)

(b) Verteilung der in ϕ-Klassen enthaltenen ϕ-Funktionen (ϕ-Funktionen pro ϕ-Klasse)

Abbildung 4.12: Kenngrößen von ϕ-Kongruenzklassen

4.4.2 Der Algorithmus

Der Tauschalgorithmus versucht eine gegebene gültige Färbung durch Tauschen von Farben so abzuändern, dass dadurch eine Gleichfärbung von mehr Knotenpaaren erreicht wird. Aufgrund der aus der statistischen Analyse gewonnenen Erkenntnisse zerlegen wir das Problem in mehrere kleine Teilprobleme, so genannte Optimierungseinheiten (OUs). Diese werden in einer bestimmten Ordnung, die durch die Gewichtung der einzelnen Kopien vorgegeben wird, sequenziell abgearbeitet. Die Collect-Phase bestimmt alle Optimierungseinheiten und legt die Reihenfolge der Abarbeitung fest. Die Optimize-Phase probiert für jede Optimierungseinheit eine Gleichfärbung der enthaltenen Knoten aus. Dadurch auftretende Konflikte werden, falls möglich, rekursiv aufgelöst.

COLLECT

In diesem Teilschritt werden zunächst alle Optimierungseinheiten und deren Parameter bestimmt. Eine Optimierungseinheit besteht immer aus einem Wert E und einem oder mehreren Werten A_i. Dabei sind alle A_i Argumente von Kopien, die alle E als Ergebnis haben und nicht mit E interferieren. Da der Ergebnisknoten E eindeutig ist, können die Kosten für die Kopien mit den Argumenten assoziiert werden. Es gibt genau drei Situationen für die Optimierungseinheiten erzeugt werden:

1. Für jede ϕ-Funktion $r = \phi(a_1, \ldots, a_n)$ wird eine Optimierungseinheit erzeugt, wobei $E = r$ und die *Menge* der Argumente $\{a_1, \ldots, a_n\}$ die Werte A_i sind. Alle A_i sind paarweise verschieden. Tritt ein Wert mehrfach als

Argument auf, werden die Kosten für die verschiedenen Argumentpositionen addiert. Gilt $r = a_i$ für ein i (was bei Schleifen auftreten kann) wird a_i nicht in die Optimierungseinheit aufgenommen, da die Gleichfärbung hier trivialerweise immer erfüllt ist.

2. Für jede Perm-Funktion $(r_1, \ldots, r_n) = \text{perm}(a_1, \ldots, a_n)$ werden n Optimierungseinheiten mit jeweils $E = r_i$ und $A_1 = a_i$ erzeugt.

3. Für alle Zwei-Adress-Befehle $a = op(a, c)$. Diese werden im gesamten Verlauf des Backends wie „normale“ Drei-Adress-Befehle $a = op(b, c)$ mit einem Gleichfärbewunsch von a und b behandelt. Erst wenn Assemblertext generiert wird, muss bei Bedarf, bei unterschiedlichen Registern von a und b, eine Kopie $a = b$ eingefügt werden.

Anhand der gegebenen Kosten für Gleichfärbekanten, wird die Abarbeitungsreihenfolge der Optimierungseinheiten festgelegt. Diese ist für die Heuristik insofern wichtig, als dass einmal optimierte Optimierungseinheiten fixiert werden, später also nicht mehr verändert werden können. Wir haben uns dazu entschieden die Reihenfolge nach den maximalen Einzelkosten in einer Optimierungseinheit zu bestimmen. Möglich wäre auch eine Mittelwertbildung, dann aber kann die Optimierung einer wichtigen Kopie verzögert werden, falls sich mehrere A_i mit kleinen Kosten in derselben Optimierungseinheit befinden.

OPTIMIZE

In der durch die Collect-Phase festgelegten Reihenfolge werden nun die Optimierungseinheiten sequenziell und isoliert betrachtet. Nach einer Initialisierung folgt eine mehr oder weniger große Anzahl an Versuchen die Registerzuteilung abzuändern. Da diese in einer günstigen Reihenfolge durchgeführt werden, kann beim ersten Erfolg abgebrochen werden und die Änderungen an der Zuteilungen werden dauerhaft gemacht und teilweise fixiert.

Init: Zunächst wird für die betrachtete Optimierungseinheit eine sortierte Warteschlange aufgebaut. Deren einzelne Einträge QN_i bestehen aus einem Zielregister r_i, einem Konfliktgraphen K_i und einer maximal gewichteten stabilen Menge S_i dieses Konfliktgraphen[6]. Pro Register r_i, das E zugeteilt werden könnte, wird ein Eintrag QN_i erstellt. Der *Konfliktgraph* K_i ist initial genau der Teilgraph, der durch E und die A_j dieser Optimierungseinheit im Interferenzgraph I induziert wird: $K_i = I[(\bigcup_j A_j) \cup E]$. Da in einer Optimierungseinheit initial keine A_j enthalten sind, die mit E interferieren, ist E in jeder maximal stabilen Menge des Konfliktgraphen K_i enthalten. Bei Änderungen am Konfliktgraphen muss man darauf achten, dass diese Eigenschaft für E erhalten bleibt. Die Sortierung in der Warteschlange wird durch das Gesamtgewicht der maximal stabilen Mengen bestimmt.

Entnimmt man nun der Warteschlange den Eintrag QN_i mit höchstem Gesamtgewicht zuerst, und kann allen Knoten aus S_i dasselbe Register r_i zuweisen, hat man das Bestmögliche für diese Optimierungseinheit erreicht. Diese Belegung wird nun getestet.

[6] Mit Konfliktgraph ist hier nicht der Interferenzgraph gemeint

Test: Aus der Warteschlange wird der beste Kandidat $QN_i = (r_i, K_i, S_i)$ entnommen. Nun werden temporär der Reihe nach, immer beginnend mit E, allen Werten aus S_i das Register r_i zugeteilt. Wird $u \in S_i$ das Register r_i zugeteilt und einem im Interferenzgraphen benachbarten Knoten v ist auch schon Register r_i zugeteilt worden, entsteht ein *Konflikt*. Solche Konflikte werden aufgelöst, indem v das Register zugeteilt wird, das u zuvor hatte; u und v tauschen die Register. Dadurch können mit v und einem weiteren Wert w wieder Konflikte entstehen, die rekursiv aufgelöst werden. Somit kann sich das Registertauschen über den Interferenzgraphen fortpflanzen.

Dieses Tauschen, das bei einem Wert $u \in S_i$ startet, findet immer ein Ende in einem der folgenden vier Fälle:

- Einem Wert v wird ein Register zugeteilt und v hat keine Nachbarn in I mit demselben Register. Erfolgreiches Ende.

- Einem Wert v müsste ein Register zugeteilt werden, welches durch die Registerbedingungen verboten wird. Erfolgloses Ende, da dieser Konflikt nicht aufgelöst werden kann. Verursacht wurde er durch den Wunsch das Register von u auf r_i zu ändern. Deshalb wird K_i um die Schlinge $[u, u]$ erweitert und mit neuer maximal stabiler Menge wieder in die Warteschlange eingefügt. Diese neue stabile Menge enthält u nicht mehr, womit dieser Konflikt beim nächsten Versuch nicht mehr auftreten kann. Gilt $E = u$ wird dieser Eintrag QN_i nicht mehr in die Warteschlange eingefügt, da es keinen Sinn macht, den A_j das Register r_i zuzuteilen, wenn dies für E nicht mehr möglich ist.

- Einem Wert v müsste ein Register zugeteilt werden, v ist aber schon zuvor durch die Optimierung einer anderen Optimierungseinheit fixiert worden[7]. In diesem Fall würde das erneute Ändern des Registers von v zwar bei der aktuellen Optimierungseinheit nützen, aber gleichzeitig eine andere Optimierung wieder zu Nichte machen. Somit wird K_i, wie oben um die Schlinge $[u, u]$ ergänzt und wieder eingefügt, falls $E \neq u$.

- Einem Wert v müsste ein Register zugeteilt werden, v ist aber schon zuvor durch die Optimierung in derselben Optimierungseinheit fixiert worden. Das heißt $u, v \in S_i$ stehen in einer gewissen Wechselwirkung: Es ist nicht möglich gleichzeitig beiden Werten durch diesen Tauschalgorithmus dasselbe Register zuzuweisen. Dieser Konflikt wird durch Einfügen der zusätzliche Kante $[u, v]$ in K_i behoben. Gilt $v = E$ würde eine Kante $[E, A_j]$ in den Konfliktgraphen eingefügt werden. Somit wäre E nicht mehr automatisch in einer maximal stabilen Menge enthalten. Deshalb wird im Fall $v = E$ die Schlinge $[u, u]$ ergänzt.

Bei einem erfolglosen Ende wird ein nicht auflösbarer Konflikt durch das Einfügen einer Kante in K_i modelliert. Diese neue Kante wird dann bei der nächsten Berechnung einer maximal gewichteten stabilen Menge berücksichtigt. QN_i

[7] siehe Teilschritt Fixieren

wird dann entsprechend des neuen Gesamtgewichts wieder in die Warteschlange eingefügt. Danach wird der nächstbeste Eintrag entnommen und getestet.

Apply: Sobald das Registertauschen für einen Eintrag QN_i der Warteschlange ein erfolgreiches Ende nimmt, werden die erreichten Optimierungen fixiert und die temporären Änderungen an der Registerzuteilung dauerhaft gemacht. Die Registerzuteilung von allen Werten in S_i werden nun fixiert, sodass die Optimierungen von nachfolgenden Optimierungseinheiten das Erreichte nicht wieder zerstören können. Danach fährt man mit für die nächste Optimierungseinheit mit Init fort.

4.4.3 Ergänzendes

In diesem Abschnitt sollen die Eigenschaften des Tauschalgorithmus näher betrachtet werden, die ihn zu einer Heuristik machen.

Die Bestimmung einer maximal gewichteten stabilen Menge ist ein NP-vollständiges Problem[8]. Für die durchschnittliche Anzahl an Werten pro Optimierungseinheit stellt dies aber kein Problem dar und kann sogar durch Ausprobieren aller Möglichkeiten in vernünftiger Zeit erledigt werden. Theoretisch kann eine Optimierungseinheit natürlich beliebig viele Werte enthalten. Für diesen Fall muss man ab einer bestimmten Größe der Optimierungseinheit für die Bestimmung der maximal stabilen Menge eine Heuristik benutzen, um lange Laufzeiten zu vermeiden. Die Auswirkungen auf die Lösungsqualität sind als sehr gering einzuschätzen.

Aufgrund der statistischen Untersuchungen haben wir auch auf die Betrachtung von kompletten ϕ-Kongruenzklassen verzichtet. Den vermuteten Gewinn sehen wir im Verhältnis zum Aufwand als zu gering an. Denn die einfache Übertragung der Bestimmung von maximal stabilen Mengen auf ϕ-Klassen ist nicht möglich. Da mehrere ϕ-Funktionen enthalten sein können, muss man berücksichtigen, dass die Argumente mit ihrer zugehörigen ϕ-Funktion in einer stabilen Menge enthalten sein müssen. Auch die Zuteilung von unterschiedlichen Registern an alle enthaltenen ϕ-Funktionen kann für eine optimale Lösung notwendig sein.

Die Heuristik verwendet beim Registertausch immer genau zwei Register. Soll einem Wert v mit Registerzuweisung r_j das Register r_i zugewiesen werden, verwendet die Konfliktauflösung immer nur diese zwei Register. Ein Durchprobieren aller möglichen Register führt zu einer kombinatorischen Explosion und scheidet somit aus. Auf der anderen Seite werden aber pro Optimierungseinheit alle möglichen Register in unterschiedlichen QN_i betrachtet. Somit ist die Frage wie viel Einfluss dies auf die Lösungsqualität hat, nicht leicht zu beantworten.

Ein weiterer Punkt ist sicherlich die sequenzielle Abarbeitung und das Fixieren der Teillösungen. Dies hat von allen genannten Punkten wohl die größte Auswirkung. Der Algorithmus ist aber als eine Heuristik entworfen worden und Backtracking, das nachträgliches Ändern von bereits optimierten Optimierungseinheiten zulässt, würde zu einem langsameren und komplizierteren Algorithmus führen. Außerdem steht für eine exakte Lösung der im nächsten Abschnitt vorgestellte Algorithmus zur Verfügung.

[8] Der oben genannte Konfliktgraph ist nicht notwendigerweise chordal

4.5 Optimale Lösung

Dieser Abschnitt präsentiert einen Algorithmus, der für Kopienminimierungsprobleme optimale Lösungen bestimmt, indem sie auf binäre, lineare Optimierungsprobleme transformiert.

4.5.1 Formalisierung

Zunächst muss das Problem in geeigneter Weise formalisiert werden. Die Kosten werden mit einer Zielfunktion modelliert, die es zu minimieren gilt. Dabei müssen natürlich Nebenbedingungen eingehalten werden, die sich aus den Bedingungen einer gültigen Registerzuteilung herleiten lassen.

Das Kopienminimierungsproblem sei wieder als ein chordaler, k-färbbarer Graph $G = (V, E, Q)$ mit Knotenmenge V, Interferenzkantenmenge E und Gleichfärberelation Q gegeben. Gesucht ist eine optimale Färbung dieses Graphen. Deshalb werden im folgenden die Begriffe Knoten und Wert sowie Register und Farbe äquivalent benutzt. Für jede Kante $e \in Q$ sei w_e das nicht negative Gewicht der Kante bzw. die Kosten der Kopie. Sei weiterhin $C = 1, \ldots, k$ die Menge aller Farben und die Funktion $p : V \to 2^C$ gebe alle zuteilbaren Farben für einen Knoten an. Für eine Färbung des Graphen gibt $c : V \to C$ die Farbe für einen Knoten an.

Zunächst formalisieren wir die Färbung eines Graphen, dann die Einschränkung auf gültige Färbungen und anschließend wird das Optimalitätskriterium festgelegt.

Für jeden Knoten $v_i \in V$ bestimmen die Variablen $x_{i1}, \ldots, x_{ik} \in \{0, 1\}$ die Farbe des Knotens:

$$c(v_i) = n \Leftrightarrow x_{in} = 1$$

Eine gültige Färbung zeichnet sich durch folgende Bedingungen aus:

1. Jeder Knoten hat genau eine Farbe.
2. Diese Farbe ist aus der Menge der zuteilbaren Farben für diesen Knoten.
3. Alle Knoten, die im Interferenzgraph benachbart sind, haben unterschiedliche Farben.

Punkt 1) und 2) lassen sich mit einer Gleichung pro Knoten formalisieren und bilden die erste Klasse (V) von Nebenbedingungen. Man führt nur die Variablen ein, die der Menge der zuteilbaren Farben entsprechen und setzt sie wie folgt in Beziehung zueinander:

$$\forall v_i \in V : \sum_{c \in r(v_i)} x_{ic} = 1$$

Für Punkt 3) müssen die Interferenzkanten aus E betrachtet werden. Für jede Farbe darf höchstens einer dieser Knoten diese Farbe erhalten. So ergibt sich eine weitere Klasse (E) von Nebenbedingungen:

$$\forall [v_i, v_j] \in E \, \forall c \in p(v_i) \cap p(v_j) : x_{ic} + x_{jc} \leq 1$$

Jetzt fehlt nur noch eine Formulierung des Optimalitätskriteriums als Zielfunktion. Die richtige Wahl ist jedoch nicht ganz einfach: Falls man die Zielfunktion direkt durch die Variablen $x_i c$ ausdrücken will, erhält man eine quadratische Zielfunktion[9]. Selbst mit Verwendung der besten uns bekannten Linearisierungstechnik (Chaovalitwongse et al. [19]) für diese Klasse von quadratischen Problemen, liegen die Lösungszeiten deutlich höher als mit dem folgenden von uns gewählten linearen Ansatz.

Für jeden Gleichfärbewunsch $[v_i, v_j] \in Q \backslash E$[10] wird eine zusätzliche binäre Variable y_{ij} eingeführt, und es gelte: $y_{ij} = 0 \Leftrightarrow c(v_i) = c(v_j)$. Jetzt lässt sich die Zielfunktion leicht bestimmen:

$$\min f = \sum_{e:=[v_i,v_j] \in Q} w_{ij} y_{ij}$$

Eine Kopplung der y- an die x-Variablen erreicht man durch eine zusätzliche Klasse (Q) von Nebenbedingungen. Da eine Variable y_{ij} wegen der Minimierung automatisch den Wert 0 annimmt, falls dies möglich ist, müssen die Nebenbedingungen nur dafür sorgen, dass $y_{ij} = 1$, falls $c(v_i) \neq c(v_j)$.

$$\forall [v_i, v_j] \in Q \begin{cases} \forall c \in p(v_i) \cap p(v_j): & y_{ij} \geq x_{ic} - x_{jc} \\ & y_{ij} \geq x_{jc} - x_{ic} \\ \forall c \in p(v_i) \backslash p(v_j): & y_{ij} \geq x_{ic} \\ \forall c \in p(v_j) \backslash p(v_i): & y_{ij} \geq x_{jc} \end{cases}$$

Zusammengefasst ergibt sich folgendes lineare, binäre Optimierungsproblem:

$$\begin{array}{lll} \min f = & \sum_{e:=[v_i,v_j] \in Q} w(e) y_{ij} & \\ u.d.N. & \sum_{c \in r(v_i)} x_{ic} = 1 & v_i \in V \\ & x_{ic} + x_{jc} \leq 1 & [v_i, v_j] \in E, c \in p(v_i) \cap p(v_j) \\ & \left\{ \begin{array}{l} y_{ij} \geq x_{ic} - x_{jc} \\ y_{ij} \geq x_{jc} - x_{ic} \end{array} \right\} & [v_i, v_j] \in Q, c \in p(v_i) \cap p(v_j) \\ & y_{ij} \geq x_{ic} & [v_i, v_j] \in Q, c \in p(v_i) \backslash p(v_j) \\ & y_{ij} \geq x_{jc} & [v_i, v_j] \in Q, c \in p(v_j) \backslash p(v_i) \\ & y_{ij}, x_{ic} \in 0, 1 & \end{array}$$

Diese Formalisierung reicht theoretisch aus, um eine optimale Lösung für Gleichfärbeprobleme zu berechnen. In der Praxis zeigt sich jedoch, dass selbst die Laufzeiten der besten Lösungsmodule unbefriedigend sind. Dem kann man durch Maßnahmen begegnen, die in den nächsten zwei Abschnitten dargelegt sind.

4.5.2 Reduktion der Problemgröße

Cliquenungleichungen

Eine Clique $\{v_1, \ldots, v_n\}$ mit n Knoten hat $\frac{n(n-1)}{2}$ Kanten, für die jeweils eine Ungleichung der Klasse (E) generiert werden müsste. Ist aber bekannt, dass

[9] Die Gleichfärbung von zwei Knoten lässt sich über ein Skalarprodukt der zugehörigen x-Vektoren feststellen: v_i und v_j haben dieselbe Farbe, $\Leftrightarrow x_i x_j = 1$, wobei $x_i = (x_{i1}, \ldots, x_{ik})$.

[10] Für die Kanten in $Q \cap E$ ist eine Gleichfärbung nie möglich.

diese Kantenmenge zu einer Clique gehört, lassen sich all diese Ungleichungen durch eine äquivalente Cliquenungleichung ersetzen:

$$\sum_{i=1}^{n} v_i \leq 1$$

Es ist bekannt, dass für Färbeprobleme von perfekten Graphen Cliquenungleichungen facettendefinierend sind. Die Ungleichungen, die aus einer minimalen Cliquenüberdeckung des Graphen resultieren, reichen aus, um die konvexe Hülle des zulässigen Bereiches exakt zu definieren. Dies zeigt sich auch darin, dass perfekte Graphen in polynomialer Zeit minimal gefärbt werden können. Chordale Graphen sind eine Teilmenge der perfekten Graphen und eine minimale Cliquenüberdeckung lässt sich in $O(|V|^2)$ bestimmen. Somit erhält man eine effizientere, kleinere Formulierung, wenn man alle Ungleichungen der Klasse (E) durch Cliquenungleichungen ersetzt.

Simpliziale Knoten

Eine weitere Idee die Anzahl der Variablen und Nebenbedingungen zu verringern ist das Ignorieren von Knoten, die nichts mit den Gleichfärbebedingungen zu tun haben. Gegeben sei das Gleichfärbeproblem $G = (V, E, Q)$, eine perfekte Eliminationsordnung $v_1, \ldots, v_n, \ldots v_m$ von G und eine optimale Lösung für das Teilproblem $G' = G[v_n, \ldots, v_m]$.

Sind $v_1, \ldots, v_{n-1}$ nicht zu Kanten aus Q inzident, sind sie nicht Teil von Gleichfärbewünschen. Dann kann man eine optimale Lösung des Teilproblems G' durch einfaches Zu-Ende-Färben der Knoten $v_{n-1}, \ldots, v_1$ zu einer optimalen Lösung des Gesamtproblems vervollständigen.

Dies lässt sich für unsere Zwecke wie folgt ausnutzen: Man entfernt aus G iterativ alle simplizialen Knoten, die nicht zu Kanten aus Q adjazent sind. Somit hat man einen 'maximal' langen Anfang einer perfekten Eliminationsordnung für G bestimmt. Das resultierende 'minimale' Teilproblem, dass durch die lineare Optimierung betrachtet werden muss, ist strukturell zwar gleich, aber kleiner.

4.5.3 Beschneiden des Suchraums

In diesem Abschnitt werden zwei Klassen von Ungleichungen für das Gleichfärbeproblem vorgestellt. Beiden ist die Eigenschaft gemein, dass sie für bestimmte Mengen von Gleichfärbewünschen untere Schranken bestimmen. Fügt man sie dem Problem hinzu, beschneiden sie den Suchraum, der vom Lösungsmodul betrachtet werden muss.

Pfad-Ungleichungen (P)

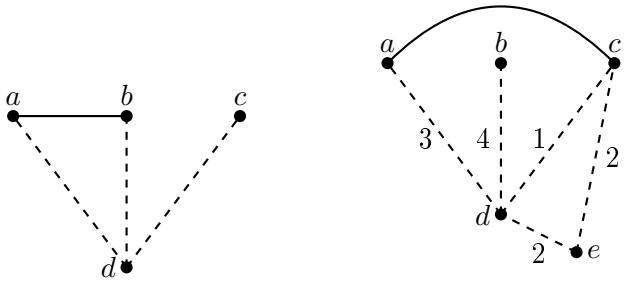
Die Idee für Pfadungleichungen beruht auf der Unvereinbarkeit von Interferenz- und Gleichfärbekanten. Gleichfärbekanten stehen für den Wunsch nach gleichen Farben, Interferenzkanten erzwingen aber unterschiedliche. So kann man bei bestimmten Graphmustern folgern, dass auf keinen Fall alle Gleichfärbewünsche in gegeben Menge erfüllt werden können.

Definition 4.4 (gleichfärbeverbunden). Gegeben sei ein Graph $G = (V, E, Q)$ mit Knotenmenge V, Interferenzkantenmenge E und Gleichfärbekantenmenge Q. Zwei Knoten $a, b \in V$ heißen *gleichfärbeverbunden*, ecc(a, b), falls a und b durch ein Pfad aus Gleichfärbekanten verbunden sind, und keine inneren Knoten des Pfades durch Interferenzkanten verbunden sind; $\exists v_1, \ldots, v_n \in V$:

- $a = v_1, b = v_n$
- $\forall 1 \leq i < n : (v_i, v_{i+1}) \in Q$
- $\forall 1 \leq i < j \leq n : (v_i, v_j) \in E \Rightarrow \{v_i, v_j\} = \{a, b\}$

Gilt $(a, b) \in E$ und ecc(a, b) mit Pfad $v_1, \ldots, v_n$, so ist folgende Ungleichung gültig:

$$\sum_{i=1}^{n-1} y_{i,i+1} \geq 1$$



$y_{ad} + y_{db} \geq 1$

(a) Möglicher Graph für $d = \phi(a, b, c)$

$y_{ad} + y_{cd} \geq 1$

$y_{ad} + y_{de} + y_{ec} \geq 1$

(b) Graph für $d = \phi(a, b, c)$, $e = \phi(d, c)$

Abbildung 4.13: Verschiedene Pfadungleichungen für CCM (a) und WCM (b)

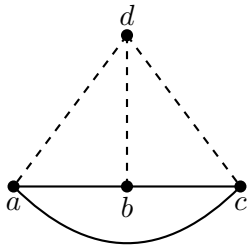
Cliquen-Pfad-Ungleichungen

Gegeben sei eine Clique $C = \{v_1, \ldots, v_n\}$ im Interferenzgraphen und ein Knoten $a \notin C$. Sind alle $(a, v_i) \in Q$, so gilt:

$$\sum_{i=1}^{n} y_{v_i,a} \geq n - 1$$

Dass eine größere Anzahl von Argumenten einer ϕ-Funktion eine Interferenzclique bilden ist eher unwahrscheinlich, allein aufgrund der Tatsache, dass eine ϕ-Funktion unseren Messungen zufolge im Durchschnitt nur 2.4 Argumente hat. Aber falls ein Argument von mehreren ϕ-Funktionen eines Grundblocks verwendet wird, ist die Interferenzclique immer vorhanden, da alle Ergebnisse

einer Φ-Funktion paarweise interferieren. Abbildung 4.14 zeigt dies für $\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \Phi \begin{pmatrix} d & e \\ d & f \\ d & g \end{pmatrix}$.



$$y_{ad} + y_{bd} + y_{cd} \geq 2$$

Abbildung 4.14: Cliquen-Pfad-Ungleichung für CCM. Die Werte e, f und g sind nicht dargestellt.

5 Implementierung

Im Rahmen dieser Arbeit wurden die theoretischen Konzepte, die in Kapitel 4 beschrieben wurden, in die Praxis umgesetzt. Beide Lösungsverfahren für Kopienminimierung wurden in einem Übersetzer implementiert, der hier kurz vorgestellt werden soll.

5.1 Die Zwischendarstellung Firm

Der Übersetzer verwendet das GCC-Frontend, an das die Zwischendarstellung Firm angeschlossen ist. Firm ist eine moderne Zwischendarstellung die in Karlsruhe am IPD Goos entwickelt wird. Sie implementiert statische Einmalzuweisung (SSA) und stellt Programme als Graphen dar, die Steuer- und Datenflussabhängigkeiten modellieren. Von Variablen und Anweisungslisten wird komplett abstrahiert: Die Knoten des Graphen repräsentieren den Ergebniswert eines Befehls (Addieren, Laden, etc.) und die Kanten stehen für den Datenfluss zwischen Befehlen. Die einzelnen Befehle sind nicht angeordnet, sondern nur durch Kanten an Grundblockknoten gebunden, deren Abfolge durch Steuerflusskanten vorgegeben ist. Genauere Erklärungen und Definitionen zu Firm finden sich in [2]. Abbildung 5.1 zeigt den Firm-Graphen für folgende C-Funktion:

```
int main(int x, int y, int *z) {
  int res;

  if (x == y)
    res = 2*y + *z;
  else
    res = 3*x - 4*y;

  return res;
}
```

5.2 Das Backend

An diese Zwischendarstellung schließt sich nahtlos das Backend an. Zunächst erfolgt die Befehlsauswahl auf den SSA-Programmgraphen. Um Registersätze unterschiedlicher Größen berücksichtigen zu können, wurde in dieser Arbeit auf die Befehlsauswahl verzichtet. Die verwendeten Firm-Befehle kommen einer

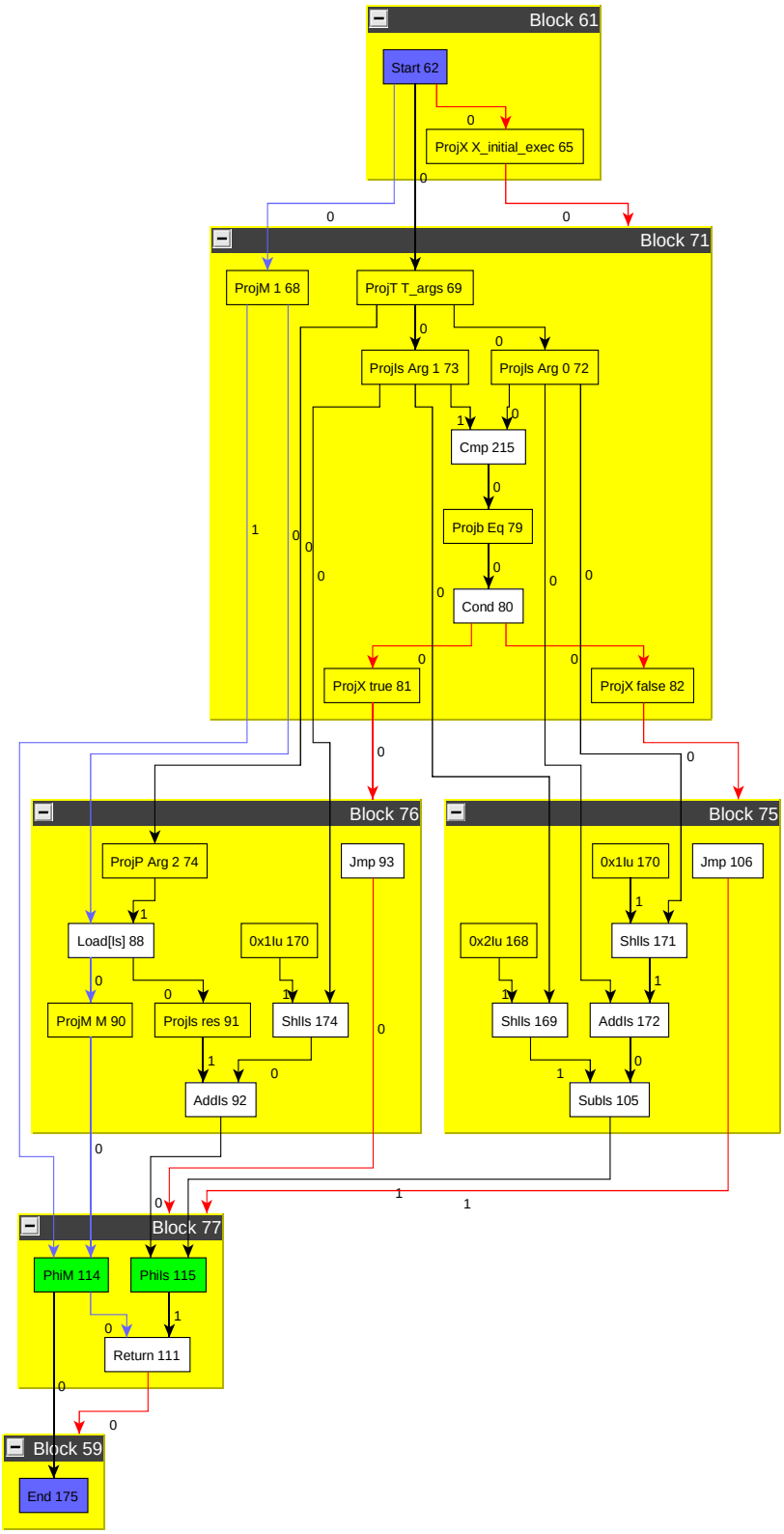


Abbildung 5.1: Die Beispiel-Funktion in FIRM-Darstellung.

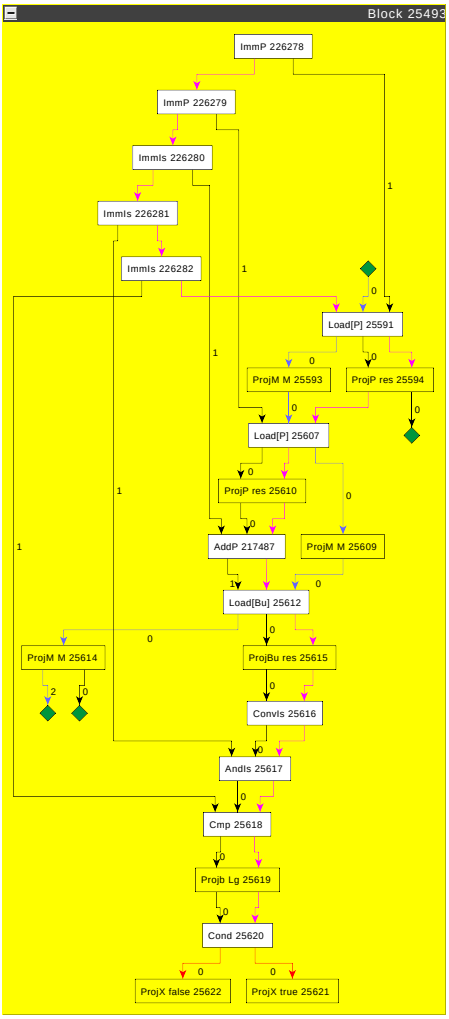


Abbildung 5.2: Ein Grundblock im Assemblergraph nach der Befehlsanordnung. Die Pfeile ohne Nummern (pink/rosa) geben die Befehlsabfolge vor. Grüne Rauten repräsentieren Referenzen in anderen Grundblöcken.

herkömmlichen RISC-Architektur aber sehr nahe. Resultat dieses Verfahrens ist ein SSA-Programmgraph, dessen Knoten Assemblerbefehlen der Zielarchitektur entsprechen. Anschließend wird die Befehlsanordnung mit einem List--Scheduling-Verfahren durchgeführt, das in dem Graphen Scheduling-Kanten ergänzt. Abbildung 5.2 zeigt einen Grundblock nach der Befehlsanordnung und -auswahl für eine prototypisch implementierte Architektur.

Danach findet die Registerzuteilung statt, deren einzelne Phasen in Abschnitt 4.2 skizziert wurden. Im Rahmen dieser Arbeit wurden die beiden beschriebenen Lösungsverfahren für Kopienminimierung sowie der SSA-Abbau implementiert und in die Registerzuteilung integriert. Abbildung 5.3 zeigt ein Programmfragment vor und nach der Kopienminimierung in einer kompakten Darstellung (das Programm selbst liegt immer noch als Graph vor). Das Ergebnis der Optimierung ist an den ϕ-Funktionen erkennbar. Durch die Zuteilung von gleichen Registern für die ϕ-Argumente ändern sich, bedingt durch Interferenzen, auch die Register anderer Werte.

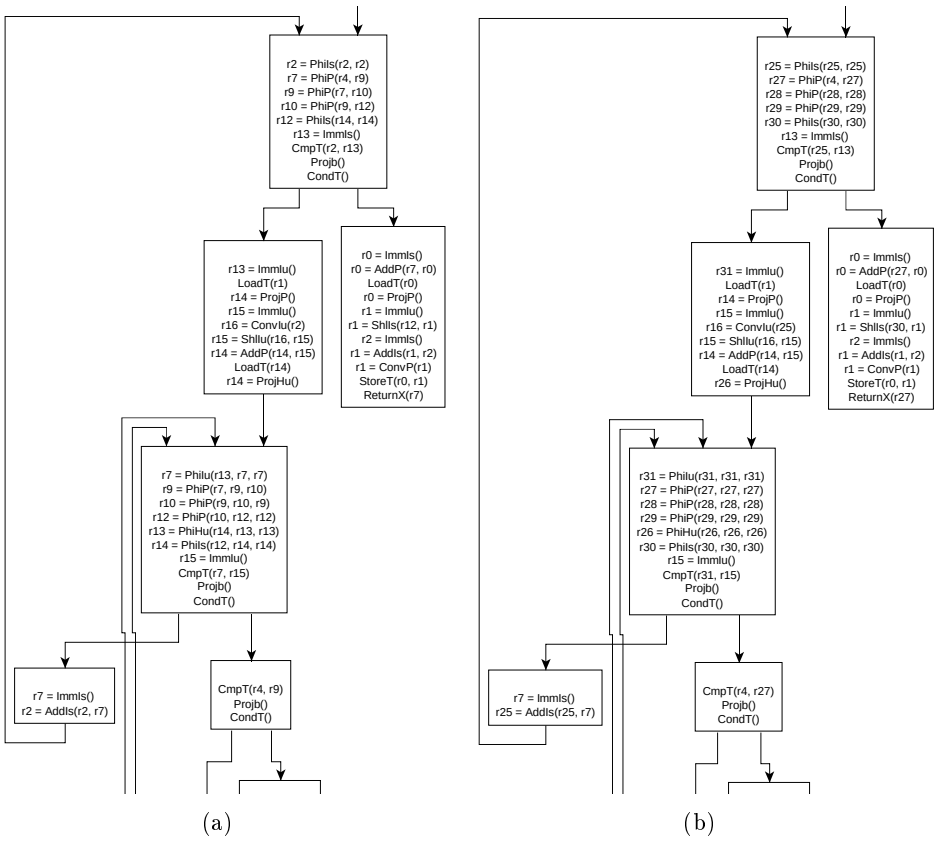


Abbildung 5.3: Teil eines Programms vor (a) und nach (b) der Kopienminimierung. Die Steuerflusskanten sind nicht geordnet; das erste Argument einer ϕ-Funktion entspricht der letzten (rechten) Kontrollflusskante. Die Anweisung `r27 = PhiP(r4, r27)` kann nicht verbessert werden, da `r4` in `CmpT(r4, r27)` benutzt wird, also mit dem Ergebnis `r27` der ϕ-Funktion interferiert.

6 Messungen

In diesem Kapitel wird die Qualität der Lösungen und die Laufzeit der implementierten Algorithmen bewertet. Dies geschieht anhand mehrerer Messreihen, die an den SPEC2000-Benchmarks durchgeführt wurden. Für alle Messungen gelten folgende Voraussetzungen: Die prototypische Zielarchitektur besitzt genau eine Registerklasse der jeweils angegebenen Größe. Ein Register kann jeden Wert eines Programms aufnehmen. Als Kostenfunktion für Kopien wurde $f(x) = d(x)^2 + 1$ gewählt, wobei $d(x)$ die Schleifenschachtelungstiefe der Kopie x angibt. Die Dateien combine.c, cse.c, regexec.c und bmt01.c fanden aufgrund von Hauptspeicherproblemen beim Übergang von Frontend auf die Zwischendarstellung keine Berücksichtigung. Gemessen wurde auf einem Pentium 4 mit 3,2 GHz unter Linux 2.6. Für die Lösung der ILPs wurde CPLEX 9.1 eingesetzt.

6.1 Lösungsqualität

In Bezug auf die Qualität der Lösungen sind die folgenden Fragen von Interesse:

- Um wieviel konnten die Kosten gesenkt werden?
- Wie weit ist die heuristische Lösung vom Optimum entfernt?
- Welchen Einfluss hat die Anzahl der verfügbaren Register?
- Ist die Lösung durch ILP mit einem Zeitlimit eine Alternative?

Tabelle 6.1 enthält die Messwerte unter Annahme von acht verfügbaren Registern. Die Spalte „ILP Optimal“ gibt jeweils die optimale Lösung mit minimalen Kosten an, die durch das ILP-Modul bestimmt wurde. Die Spalten „ILP n Sek.“ enthält die Kosten, die entstehen, falls der ILP-Algorithmus nach maximal n Sekunden (pro Funktion) abgebrochen wird und die bis dahin beste gefundene Lösung verwendet wird. Die vorletzte Spalte gibt die Kosten an, die nach Ausführung der Heuristik verblieben sind. Die letzte Spalte enthält die initialen Kosten[1], die direkt vor der Kopienminimierung vorhanden waren.

Unter Annahme von acht verfügbaren Registern konnte die Heuristik die Kosten insgesamt um 26354 (68.0%) senken. Das sind 96.2% aller optimierbaren Kosten. Damit liegt die Heuristik durchschnittlich 9.0% über der optimalen Lösung. Die optimale Lösung konnte für alle Funktionen innerhalb von 5 Sekunden

[1] In vorherigen Phasen der Registerzuteilung werden weder Lösungen bevorzugt, noch benachteiligt, die Auswirkungen auf die Kopienkosten haben.

pro Funktion bestimmt werden[2](siehe 6.8). Je nach Anspruch an die Übersetzungsdauer, kommen hier sowohl die Heuristik als auch die ILP-Methode in Betracht. Abbildung 6.1 verdeutlicht die prozentualen Verhältnisse mit Ausnahme der initialen Kosten, da diese zu stark abweichen.

Tabelle 6.2 und Abbildung 6.2 zeigen die entsprechenden Ergebnisse für 16 verfügbare Register. Hier fällt zunächst auf, dass die Kosten insgesamt höher sind. Dies hängt mit der Auslagerungsphase zusammen, die jetzt nicht mehr so viele Werte auslagern muss und mehr ϕ-Funktionen im Programm verbleiben. Dies kann auf der einen Seite zu komplexeren Kopienminimierungsproblemen führen, erhöht auf der anderen Seite aber auch die Färbemöglichkeiten, falls eine Funktion weniger als 16 Register benötigt. Interpretiert man die Werte dahingehend, stellt man fest, dass die Probleme eher komplizierter werden. Die Heuristik liefert immer noch gute Werte: 93.5% aller optimierbaren Kosten werden beseitigt, was einen Wert von 11.6% über dem Optimum bedeutet. Die Lösung durch ILP mit Zeitbeschränkung verliert deutlich an Qualität. Ein Zeitlimit von 30 Sekunden erzielt in der Summe ein Ergebnis das der heuristischen Lösung sehr nahe kommt. Somit kommen die zeitlich begrenzten Varianten für die Praxis hier nicht in Frage. Die Heuristik liefert in einem Bruchteil der Zeit Lösungen, die durch ILP erst zwischen 5 und 30 Sekunden Laufzeit gefunden werden. Hier wählt man also entweder das optimale Lösungsverfahren oder die Heuristik.

Die letzte Messreihe mit 32 Registern ist in Tabelle 6.3 und Abbildung 6.3 aufgeführt. Die gesamten Kosten erhöhen sich nicht mehr so stark wie beim Wechsel von 8 auf 16 Register. Die Abweichung der Heuristik liegt bei 11.9% und somit nur geringfügig über dem Ergebnis mit 16 Registern. Die zeitlimitierten ILP-Methoden kommen für die Praxis hier ebenfalls nicht Frage.

[2]Dies widerspricht *nicht* Abbildung 6.8. Erklärung siehe Laufzeitmessungen.

Benchmark	ILP Optimal	ILP 5 Sek.	ILP 30 Sek.	Heuristik	Initial
164.gzip	211	211	211	257	592
175.vpr	253	253	253	273	1010
176.gcc	2965	2965	2965	3150	10382
177.mesa	1073	1073	1073	1109	3566
179.art	76	76	76	76	232
181.mcf	61	61	61	66	168
183.equake	66	66	66	72	287
186.crafty	359	359	359	391	1248
188.ammp	550	550	550	586	1378
197.parser	652	652	652	687	1791
253.perlbmk	1412	1412	1412	1604	4215
254.gap	2249	2249	2249	2535	8866
255.vortex	143	143	143	166	1685
256.bzip2	309	309	309	332	700
300.twolf	1007	1007	1007	1111	2649
Summe	11386	11386	11386	12415	38769

Tabelle 6.1: Verbliebene Kosten für Kopien nach Optimierung mit 8 Registern

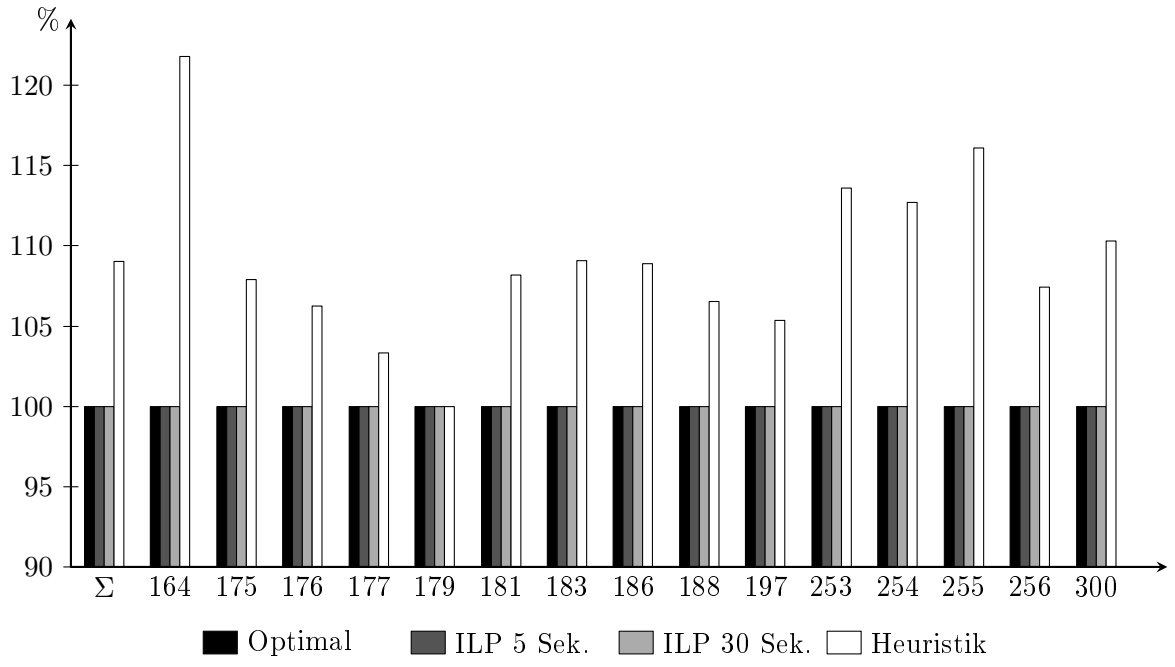


Abbildung 6.1: Prozentuale Abweichungen vom Optimum bei 8 Registern

Benchmark	ILP Optimal	ILP 5 Sek.	ILP 30 Sek.	Heuristik	Initial
164.gzip	354	354	354	432	870
175.vpr	626	626	626	662	1274
176.gcc	6021	8815	6854	6573	17125
177.mesa	2105	3438	2596	2256	6225
179.art	165	165	165	185	382
181.mcf	119	119	119	127	226
183.equake	135	135	135	137	325
186.crafty	698	1099	817	833	1899
188.ammp	870	870	870	928	1951
197.parser	988	988	988	1072	2665
253.perlbmk	2655	4315	3761	3110	7351
254.gap	4370	5735	4521	4963	12658
255.vortex	347	347	347	384	2527
256.bzip2	473	473	473	521	934
300.twolf	1921	2351	1955	2204	4292
Summe	21847	29830	24581	24387	60704

Tabelle 6.2: Verbliebene Kosten für Kopien nach Optimierung mit 16 Registern

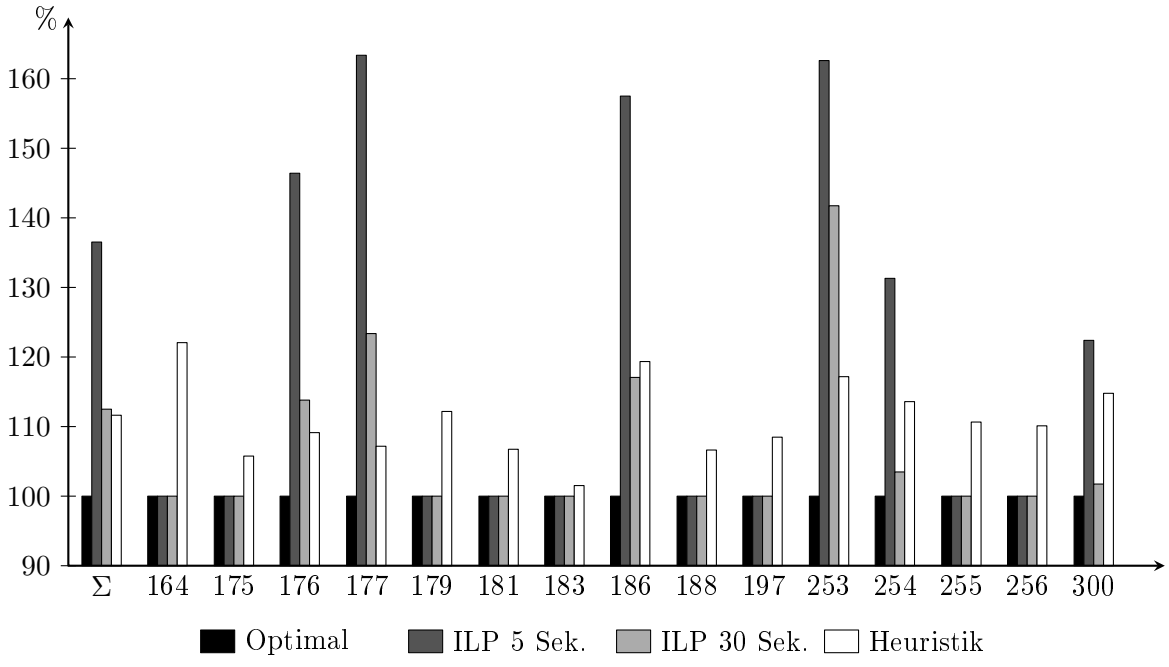


Abbildung 6.2: Prozentuale Abweichungen vom Optimum bei 16 Registern)

Benchmark	ILP Optimal	ILP 5 Sek.	ILP 30 Sek.	Heuristik	Initial
164.gzip	418	423	418	497	1060
175.vpr	668	668	668	717	1352
176.gcc	7020	12052	10361	7716	19080
177.mesa	2802	4988	4462	3072	7658
179.art	165	165	165	185	373
181.mcf	129	129	129	137	243
183.equake	179	238	221	179	455
186.crafty	700	1158	1011	816	1995
188.ammp	979	1051	1039	1042	2047
197.parser	1033	1033	1033	1109	2816
253.perlbmk	2878	5686	4938	3412	8312
254.gap	5824	8168	6976	6553	14634
255.vortex	402	539	418	440	2775
256.bzip2	513	513	513	566	977
300.twolf	2356	4744	4378	2734	5772
Summe	26066	41555	36730	29175	69549

Tabelle 6.3: Verbliebene Kosten für Kopien nach Optimierung mit 32 Registern

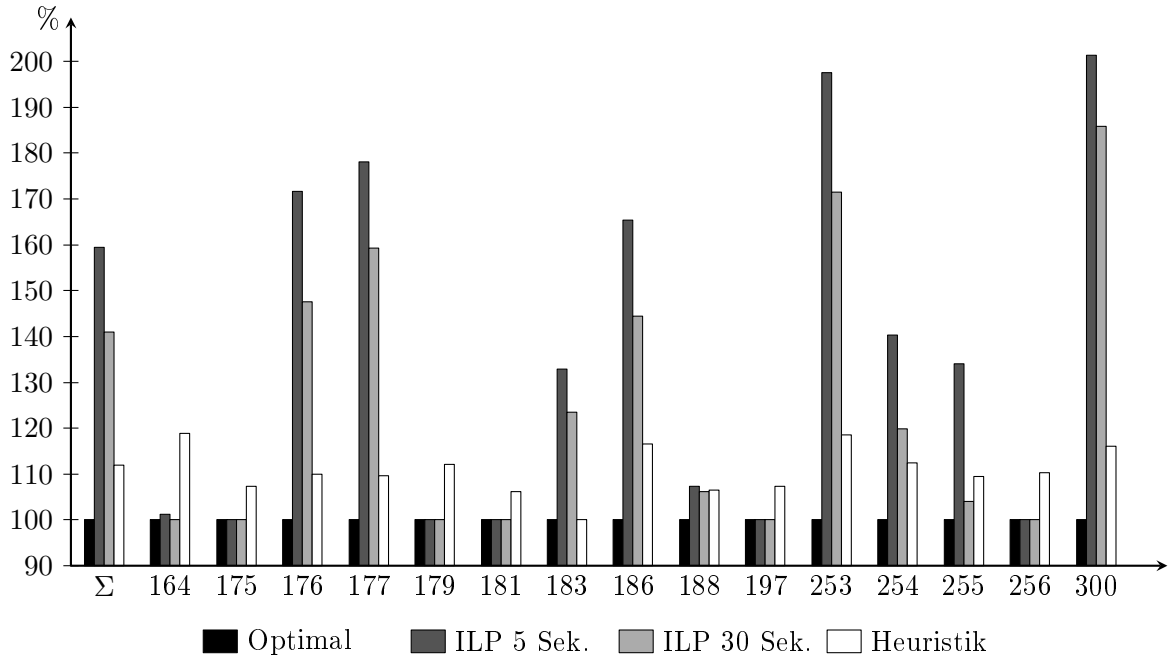


Abbildung 6.3: Prozentuale Abweichungen vom Optimum bei 32 Registern)

Betrachtet man nur die Gesamtkosten (letzte Zeilen in den Tabellen), die nach Optimierung mit einem Verfahren verbleiben, und setzt diese direkt in Beziehung zur Registersatzgröße, ergibt sich Abbildung 6.4. Da auch bei 32 Registern noch ein Anstieg der Gesamtkosten zu verzeichnen ist, wurden weitere Messungen mit 64, 128 und 256 Registern ergänzt. Diese wurden aus Zeitgründen nur mit der Heuristik durchgeführt. Ab 64 Registern stabilisieren sich die Kosten. Man erkennt in allen Fällen eine deutliche Reduzierung der Kosten gegenüber der Ausgangssituation. Die Heuristik liefert immer sehr gute Ergebnisse und scheidet bei 8 Registern am besten ab.

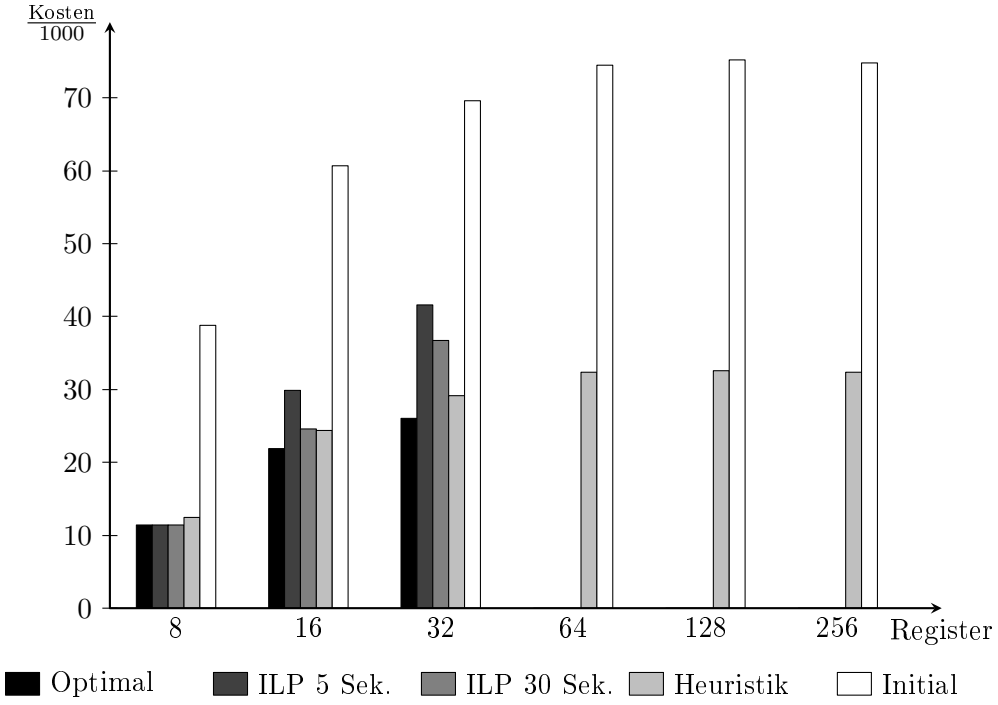


Abbildung 6.4: Auswirkung der Registeranzahl auf die Kosten. Ab 64 Register sind nur Messwerte für die Heuristik und die Ausgangssituation aufgetragen.

6.2 Laufzeit

Gewöhnlich werden Laufzeiten in Bezug zur Programmgröße (Anzahl der Instruktionen) gesetzt. Für die Kopienminimierung ist dies jedoch ungeeignet, da die Heuristik hauptsächlich von der Anzahl der Gleichfärbekanten abhängt. Um aussagekräftigere Darstellungen zu erzielen, wurde deshalb für die x-Achse die Anzahl der ϕ-Argumente gewählt. Die logarithmische Einteilung auf der Zeitachse wurde nicht wegen exponentiell ansteigenden Lösungszeiten gewählt, sondern nur, um die vielfach auftretenden Lösungen mit sehr kleinen Laufzeiten besser darzustellen. Die Messgenauigkeit beträgt 1 ms. Den Abbildungen 6.5, 6.6 und 6.7 kann man entnehmen, dass jeweils mehr als 80% aller Funktionen in weniger

als 10, 20, bzw. 30 ms optimiert wurden. Die maximalen Werte liegen bei 98, 470 und 980 ms. Weiterhin erkennt man, dass mit wachsender Registeranzahl einen Anstieg der Lösungszeiten einhergeht. Deutlich zu sehen sind auch die Auswirkung der Auslagerungsphase auf die Anzahl der ϕ-Funktionen: Je höher die Registeranzahl, desto mehr Gleichfärbekanten sind zu optimieren. Die Abhängigkeit der Zeiten von der Registeranzahl, ist höchstwahrscheinlich auf die Erzeugung der Warteschlangeneinträge der Heuristik pro Register zurückzuführen.

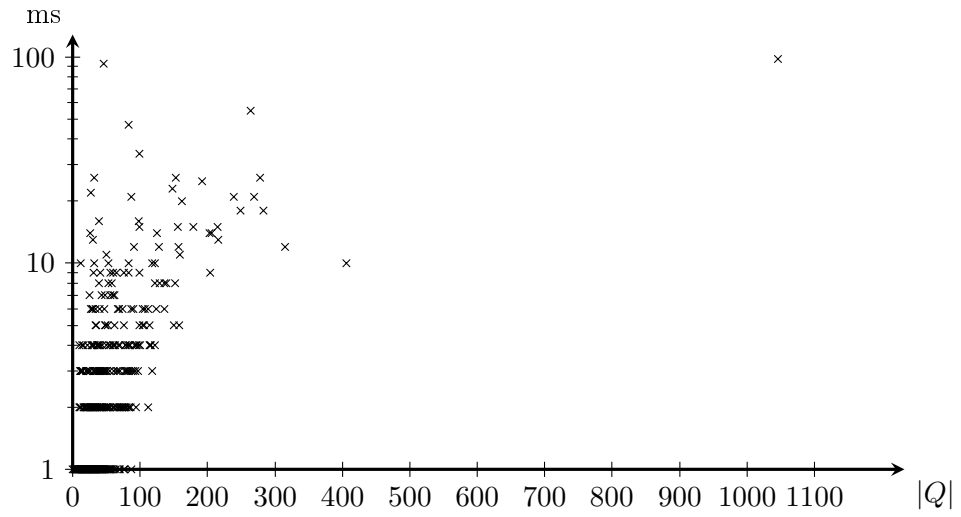


Abbildung 6.5: Laufzeit der Tauschheuristik bei 8 Registern

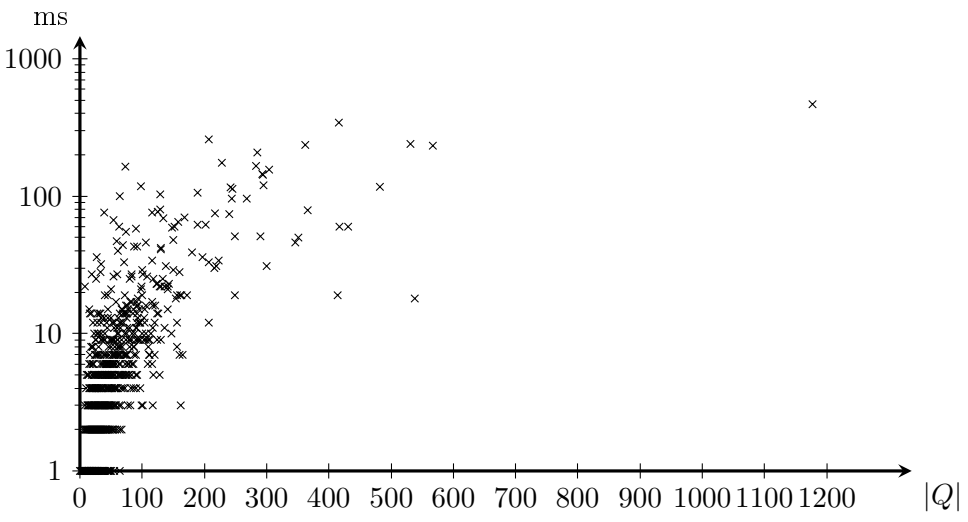


Abbildung 6.6: Laufzeit der Tauschheuristik bei 16 Registern

Die Parameter des ILP-Moduls wurden für die Laufzeitmessungen zur Bestimmung einer optimalen Lösung geringfügig abgeändert. Es wurde weniger Gewicht auf das schnelle Finden von Lösungen gelegt und mehr auf den Nach-

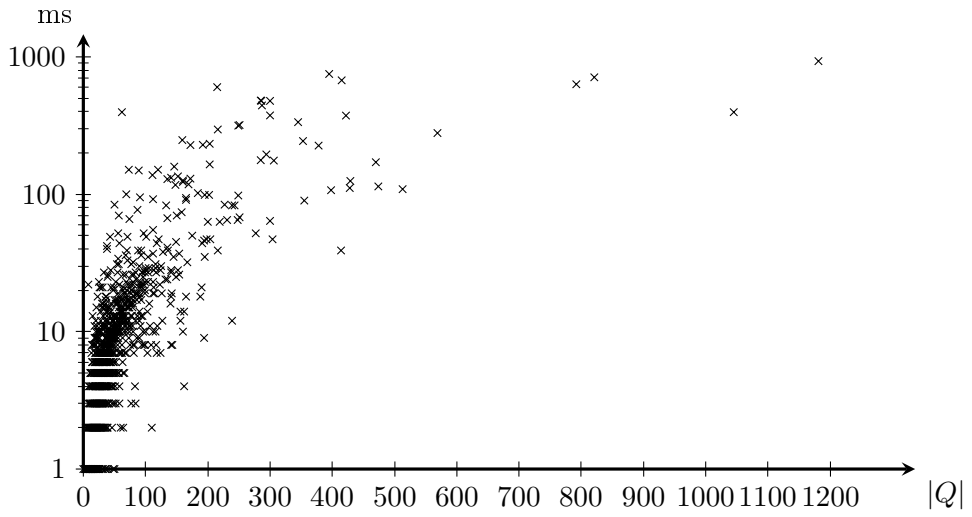


Abbildung 6.7: Laufzeit der Tauschheuristik bei 32 Registern

weis der Optimalität. Da zwischen dem Finden einer optimalen Lösung und dem Beweis der Optimalität ein Unterschied besteht, sind in Abbildung 6.8 auch Lösungszeiten größer als 5 Sekunden enthalten, obwohl nach Tabelle 6.1 alle Funktionen in maximal 5 Sekunden optimal gelöst wurden. Die interaktive Verwendung des ILP-Programms ergab in vielen Fällen tatsächlich, dass die optimale Lösung sehr schnell erreicht wurde, der Beweis der Optimalität jedoch überproportional lange dauerte. Deshalb haben wir beschlossen, für die Bestimmung der optimalen Lösung ein Zeitlimit von 1000 Sekunden zu setzen. Dieses Limit wurde für 16 und 32 Register insgesamt nur 19 mal erreicht, bei 8 Registern nie. Somit stellen die bestimmten optimalen Lösungswerte mit an Sicherheit grenzender Wahrscheinlichkeit mit dem tatsächlichen Optimum überein.

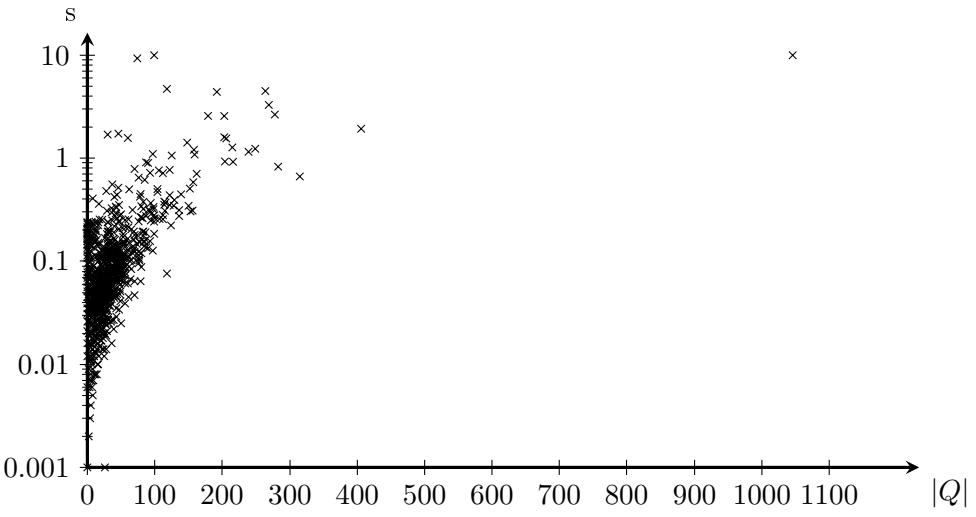


Abbildung 6.8: Laufzeit des ILP-Moduls bei 8 Registern

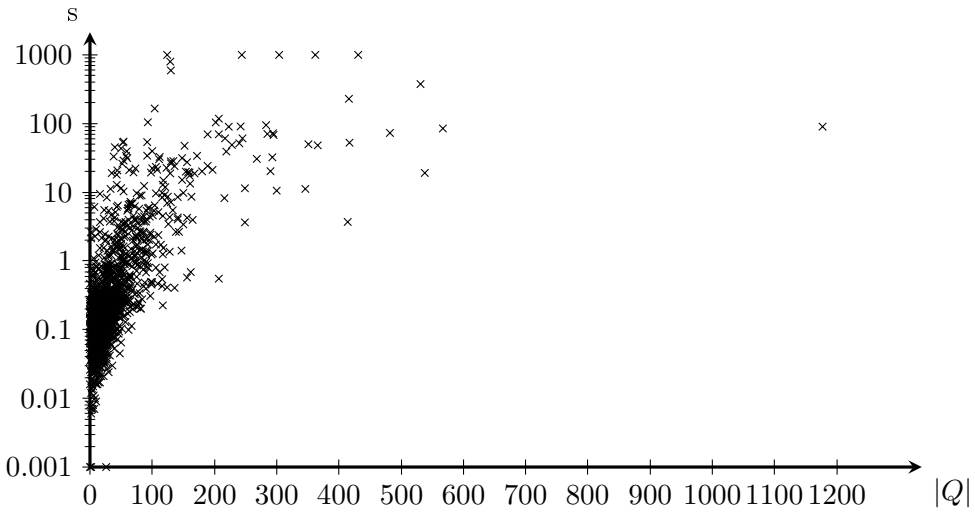


Abbildung 6.9: Laufzeit des ILP-Moduls bei 16 Registern

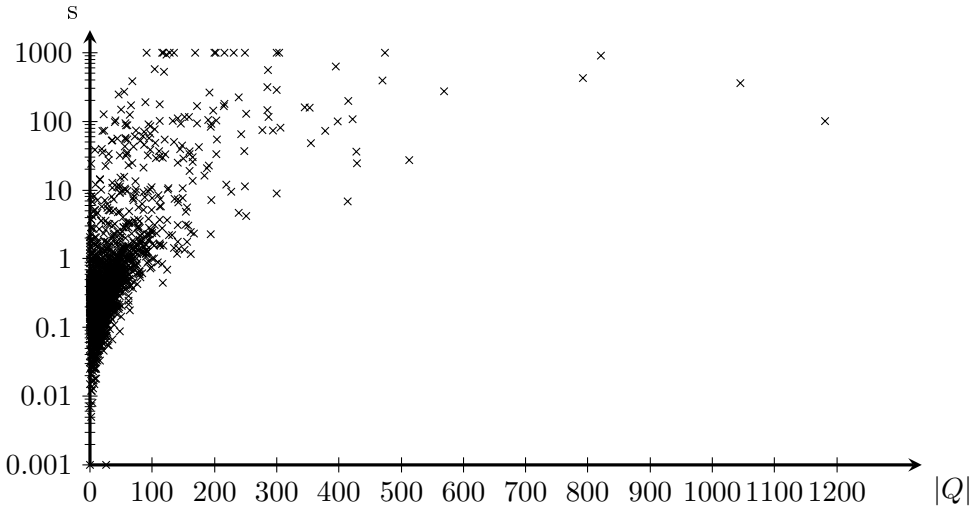


Abbildung 6.10: Laufzeit des ILP-Moduls bei 32 Registern

6.3 Sonstiges

Abschließend soll noch die Modellierung des ILP untersucht werden. Die Abbildungen 6.11 und 6.12 zeigen die Anzahl der Variablen und Nebenbedingungen in Abhängigkeit der Programmgröße ($|V|$ ist die Anzahl der Knoten im Interferenzgraphen). Man erkennt in beiden Fällen einen etwa linearen Zusammenhang mit Faktor 1,7. Da bei 8 Registern mindestens 8 Variablen pro Knoten erzeugt werden und die Gleichfärbebedingungen diese Anzahl noch erhöhen, bedeutet dies, dass vor der Erstellung ca. 80% aller Knoten entfernt werden können (siehe Abschnitt 4.5.2). Dies führt zu weniger Aufwand bei der Erstellung und wesentlich

besseren Lösungszeiten. Die Transformation eines Kopienminimierungsproblems in ein ILP mit der vorgestellten Modellierung, ist also sehr effizient durchführbar. Die Messwerte für größere Registersätze unterscheiden sich nur durch einen höheren Faktor. Für die größte Funktion mit 28248 Knoten wurden bei 32 Registern 318612 Variablen und 304971 Nebenbedingungen generiert. Die optimale Lösung wurde innerhalb von 101 Sekunden gefunden und als optimal bewiesen.

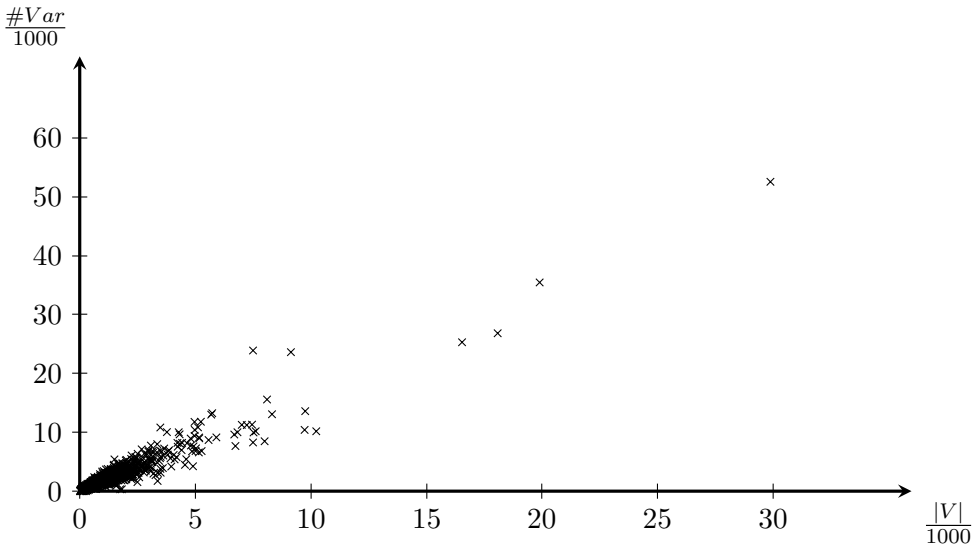


Abbildung 6.11: Variablenanzahl der ILP-Modellierung mit 8 Registern

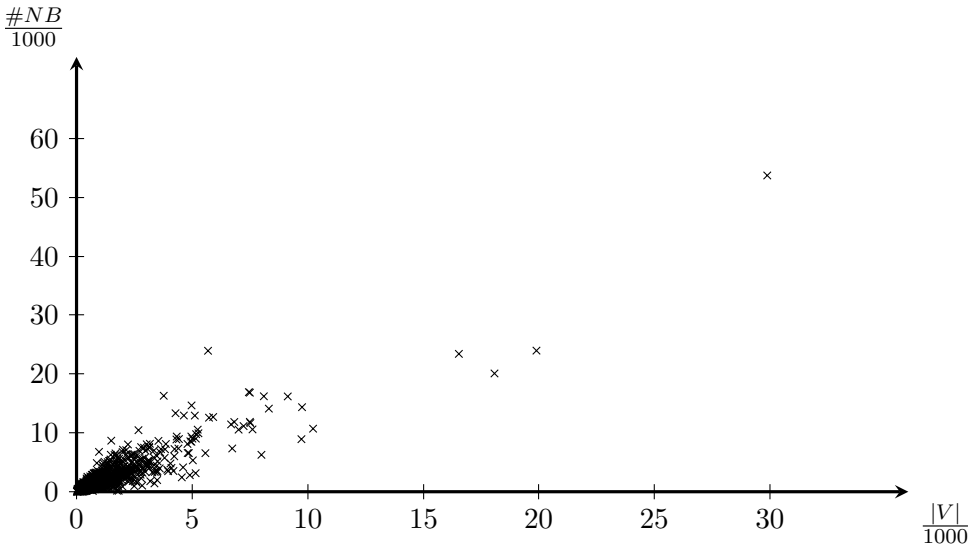


Abbildung 6.12: Nebenbedingungen in der ILP-Modellierung mit 8 Registern

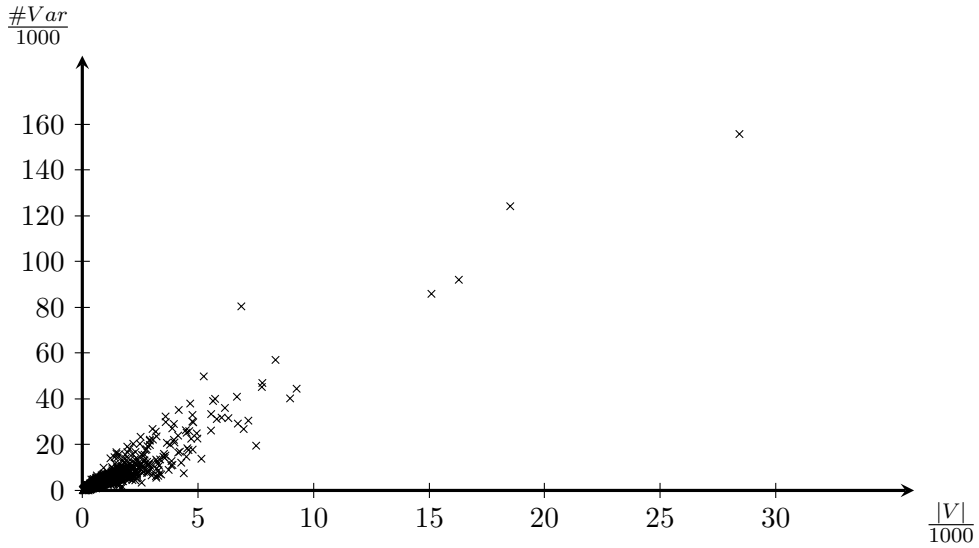


Abbildung 6.13: Variablenanzahl der ILP-Modellierung mit 16 Registern

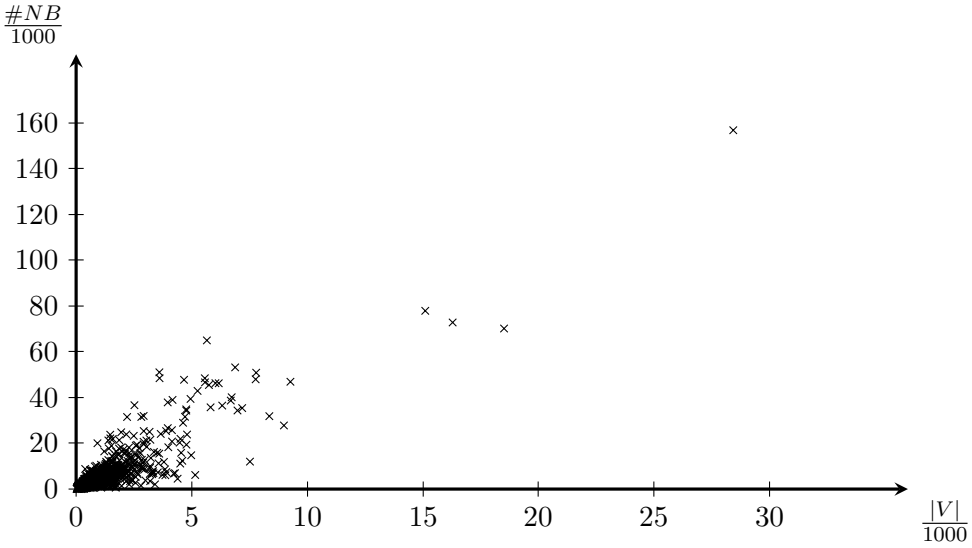


Abbildung 6.14: Anzahl der Nebenbedingungen der ILP-Modellierung mit 16 Registern

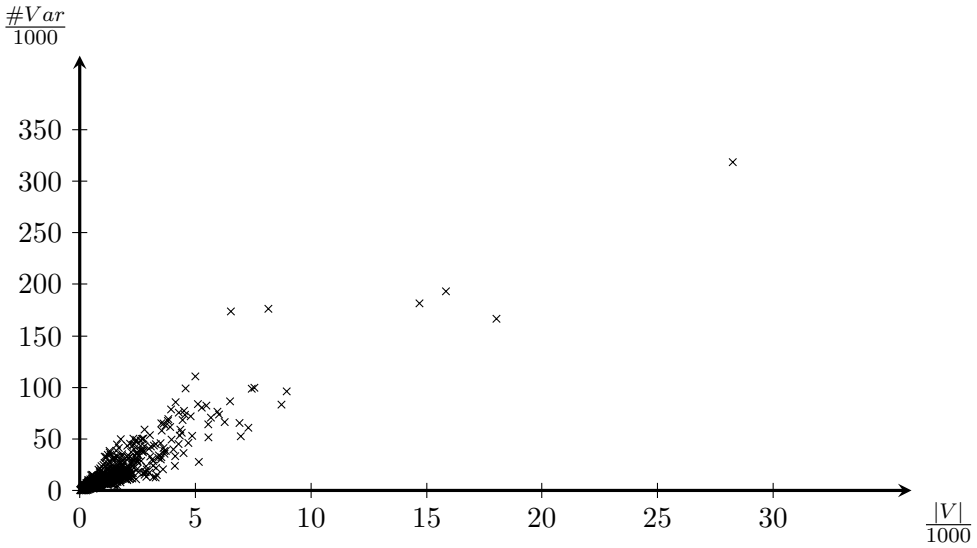


Abbildung 6.15: Variablenanzahl der ILP-Modellierung mit 32 Registern

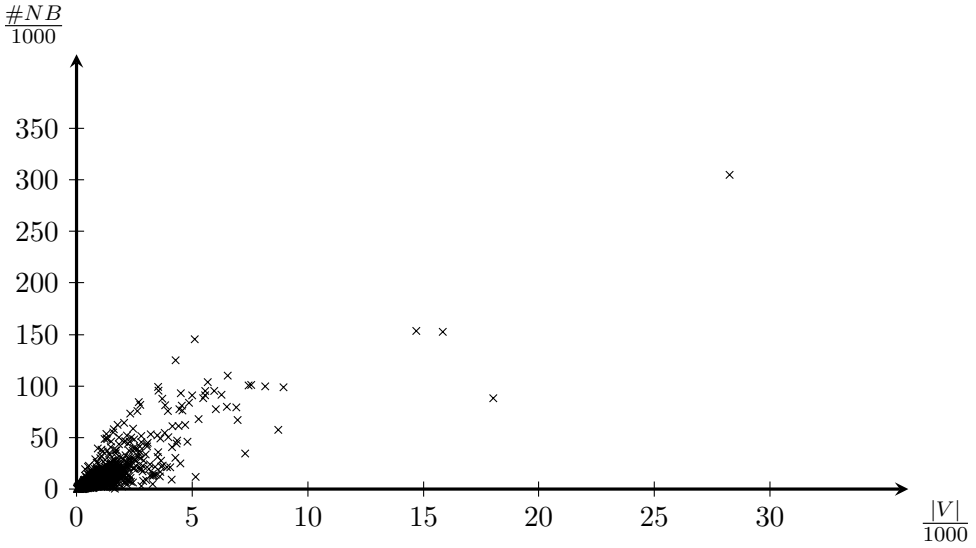


Abbildung 6.16: Anzahl der Nebenbedingungen der ILP-Modellierung mit 32 Registern

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Verfahren für die Kopienminimierung in einem SSA-basierten Registerzuteiler vorgestellt. Durch die neuartige Architektur eines solchen Zuteilers ist es möglich, Kopien nur dann zu eliminieren, falls dies keine negativen Auswirkungen auf den Registerbedarf hat. Verfahren nach aktuellem Stand der Technik können diese Auswirkungen nicht exakt bestimmen, sondern nur abschätzen. Außerdem haben wir gezeigt, wie sich die Behandlung von Registerbedingungen durch die Kopienminimierung ausdrücken lässt.

Sowohl als Ausgangspunkt zum Entwurf von Lösungsverfahren, als auch zur theoretischen Untersuchung, diente eine abstrakte Sichtweise auf das Problem: Die bekannten Interferenzgraphen wurden um einen zusätzlichen Kantentyp erweitert, der Kopien und ihre Kosten modelliert. Die Klassifizierung der Kopienminimierung als NP-vollständiges Problem zog die Entwicklung einer heuristischen Lösungsmethode nach sich. Um diese bewerten zu können, und eine Vergleichbarkeit auch mit zukünftigen Verfahren herzustellen, wurde parallel dazu ein Verfahren zur optimalen Lösung dieser Probleme entwickelt. Hierbei wird die Kopienminimierung in ein mathematisches Optimierungsproblem überführt, das dann von der Standardsoftware CPLEX$^{\text{TM}}$ gelöst wird.

Beide Verfahren wurden implementiert und in einen Übersetzer integriert. Die Ergebnisse von mehreren Messreihen an der SPEC2000-Benchmarksammlung bescheinigen der Heuristik für den praktischen Einsatz gute bis sehr gute Ergebnisse, sowohl die Güte als auch die Laufzeit betreffend.

Eine tiefer gehende Untersuchung der Komplexität der Kopienminimierung könnte zur Definition von Problemklassen führen, deren optimale Lösung effizient berechenbar ist. Eng damit verknüpft ist die noch offene Frage nach der Stärke der vorgestellten Ungleichungen, die in der Formulierung als mathematisches Optimierungsproblem Verwendung finden. Weitere Ungleichungsklassen könnten das Laufzeitverhalten des Lösungsmoduls nochmals verbessern und würden helfen, die Struktur des Problems besser zu verstehen.

A Anhang

A.1 Pseudocode der Heuristik

Die Funktion f liefert die Farbe eines Knotens im Interferenzgraphen und g repräsentiert die temporäre Färbung während des Testens.

```
procedure COALESCE(G)
    pinned = ∅                                              ▷ OPTIMIZE
    for OU = (p, a_1, ..., a_k) do
        create priority queue Q                             ▷ Init
        for colors c assignable to p do
            C_c = G[p, a_1, ..., a_k]
            S_c = maximum weighted independent set of C_c
            Insert (c, C_c, S_c) into Q
        od

        repeat                                              ▷ Test
            candidates = ∅
            g = f
            pop (c, C, S) from Q
            C' = TEST(c, C, S)
            if C' ≠ nil then
                S' = maximum weighted independent set of C'
                Insert (c, C', S') into Q
            fi
        until C' = nil

        if |candidates| > 1 then                            ▷ Apply
            pinned = pinned ∪ candidates
            f = g
        fi
    od
end
```

TEST überprüft, ob allen Knoten in S die Farbe $\mathbf{c}$ zugeteilt werden kann. Ist dies nicht möglich, wird im Konfliktgraph C eine Kante ergänzt.

```
function TEST(c, C, S)
    for u ∈ S do
        (s, v) =TRYCOLOR(u, nil, c)
        if s = ok then
            candidates = candidates ∪ u
        else
            if s = candidate and v ≠ p then
                C' = (V_C, E_C ∪ vu)
            else
                C' = (V_C, E_C ∪ uu)
            fi
            return C'                          ▷ Return enlarged conflict graph
        fi
    od
    return nil                                 ▷ All nodes could be colored
end function
```

TRYCOLOR versucht dem Knoten v die Farbe $\mathbf{c}$ zuzuteilen und entstehende Konflikte rekursiv aufzulösen. Rückgabewert ist der Knoten, der einen nicht auflösbaren Konflikt verursacht hat und der Grund dafür.

```
function TRYCOLOR(v ∈ V_G, u ∈ V_G, c)
    c_v = g(v)
    if c = c_v then
        return (ok, nil)                       ▷ The color of v is already c
    else if v ∈ pinned then
        return (pinned, v)                     ▷ v has been pinned by another OU
    else if v ∈ candidates then
        return (candidate, v)                  ▷ v has already been tested
    else if c ∉ X(v) then
        return (forbidden, v)                  ▷ c is not allowed for v
    fi

    for {n | vn ∈ E_G, n ≠ u, c = g(n)} do      ▷ Look at all conflicting
                                                 neighbours of v
        (s, v') =TRYCOLOR(n,v,c_v)             ▷ Try to give a neighbour v's color
        if s ≠ ok then
            return (s, v')
        fi
    od
    g(v) = c
    return (ok, nil)
end function
```

Literaturverzeichnis

[1] Götz Lindenmaier, Michael Beck, Boris Boesler, and Rubino Geiß. Firm, an intermediate language for compiler research. Technical Report 2005-8, Dept. of Computer Science, University of Karlsruhe (TH), March 2005.

[2] Martin Trapp, Götz Lindenmaier, and Boris Boesler. Documentation of the intermediate representation firm. Technical Report 1999-14, Universität Karlsruhe, Fakultät für Informatik, Dec 1999.

[3] Martin Charles Golumbic. *Algorithmic Graph Theory And Perfect Graphs.* Academic Press, 1980.

[4] Neumann and Morlock. *Operations Research.* Hanser Verlag, 1993.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[6] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, 28(8):859–881, 1998.

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

[8] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[9] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[10] B. Dupont de Dinechin, F. de Ferri, C. Guillon, and A. Stoutchinin. Code generator optimizations for the st120 dsp-mcu core. In *CASES '00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 93–102, New York, NY, USA, 2000. ACM Press.

[11] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004.

[12] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, pages 194–210, London, UK, 1999. Springer-Verlag.

[13] Allen Leung and Lal George. Static single assignment form for machine code. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 204–214, New York, NY, USA, 1999. ACM Press.

[14] F. Rastello, F. de Ferrière, and C. Guillon. Optimizing translation out of ssa using renaming constraints. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 265, Washington, DC, USA, 2004. IEEE Computer Society.

[15] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0–1 integer programming. *Softw. Pract. Exper.*, 26(8):929–965, 1996.

[16] Changqing Fu and Kent Wilken. A faster optimal register allocator. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 245–256, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[17] Sebastian Hack. Interference graphs of programs in ssa-form. Technical report, Dept. of Computer Science, University of Karlsruhe (TH), June 2005.

[18] F. Rastello, F. de Ferrière, and C. Guillon. Report 03-35: Optimizing translation out of ssa using renaming constraints. Technical report, École Normale Supérieure de Lyon, 2003.

[19] Wanpracha Chaovalitwongse, Panos M. Pardalos, and Oleg A. Prokopyev. A new linearization technique for multi-quadratic 0–1 programming problems. *Operations Research Letters*, 32(6):517–522, 2004.

Index