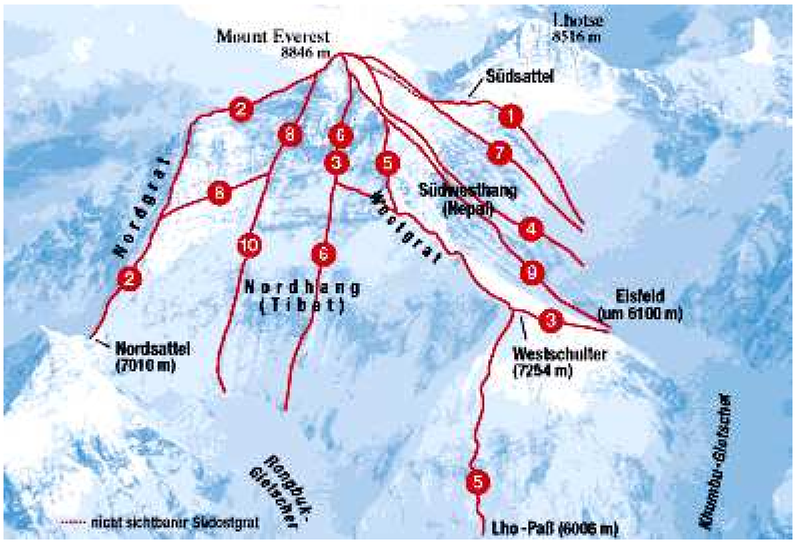
Über die Verwendung ganzzahliger linearer Programmierung zur Suche nach längsten Programmpfaden



Diplomarbeit

Fachbereich 14, Informatik
der Technischen Fakultät
der Universität des Saarlandes

von

Henrik Theiling

theiling@cs.uni-sb.de

Saarbrücken,
September 1998

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit eigenständig und nur mit Hilfe der angegebenen Quellen und Hilfsmittel angefertigt habe.

Diese Diplomarbeit wurde noch keiner anderen Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Saarbrücken, den 29. September 1998

(Henrik Theiling)

Inhaltsverzeichnis

Danksagung

Ich möchte an dieser Stelle Prof. Dr. Reinhard Wilhelm für die Möglichkeit danken, an seinem Lehrstuhl diese Diplomarbeit über ein interessantes Thema anzufertigen. Bei Prof. Dr. Gerhard Weikum bedanke ich mich für seine Bereitschaft, diese Arbeit zu begutachten. Der gesamten Arbeitsgruppe des Lehrstuhls für Programmiersprachen und Übersetzerbau danke ich für die angenehme Arbeitsatmosphäre. Besonderer Dank gebührt Dr. Christian Ferdinand für seine Betreuung und seine Geduld bei vielen Gesprächen.

Weiterhin danke ich Dr. Christian Ferdinand für die Implementierung der Cache-Analyse sowie Jörn Schneider für Diskussionen über seine Pipeline-Analyse, für Hilfe in Bezug auf SPARC-Assembler und andere hilfreiche Gespräche. Matthias Fischmann, Björn Huke und Markus Löckelt danke ich für das Aufspüren un- oder mißverständlicher Gedankengänge und für Beratung bei formalen oder methodischen Problemen. Den letzteren dreien und Noëmi Preissner danke ich außerdem für ihr Korrekturlesen und für Layoutratschläge.

Zum Entstehen und zur Verwirklichung beigetragen haben außerdem Yau-Tsun Steven Li, Sharad Malik und Andrew Wolfe durch das Projekt cinderella, das die Inspirierung brachte, lineares Programmieren zur Pfadanalyse zu benutzen, James Larus und seine Arbeitsgruppe durch die Implementierung einer Bibliothek zum Zugriff auf SPARC-Programme auf Maschinenbefehlsebene im Rahmen des Wisconsin Architectural Research Tool Sets (WARTS) sowie Michel Berkelaar mit dem frei erhältlichen LP-Löser lp_solve, der sämtliche meiner linearen Probleme lösen mußte.

Zu guter Letzt möchte ich meiner Schwester Kirsten und meinen Eltern Ursula und Manfred danken, die mich nicht nur während meines Studiums in vielerlei Hinsicht unterstützt und beraten haben.

Erstes Kapitel

Einleitung

Die vorliegende Arbeit beschäftigt sich mit der Analyse von Programmen zum Aufspüren von schlechtesten Programmpfaden mittels ganzzahligen linearen Programmierens (*engl.: Integer Linear Programming* = *ILP*). In diesem Kapitel wird geklärt, warum man überhaupt an schlechtesten Programmpfaden interessiert ist, und warum dazu als Methode das lineare Programmieren gewählt wurde.

1.1 Echtzeitsysteme

Echtzeitsysteme zeichnen sich durch Zeitschranken aus, innerhalb derer das Echtzeitsystem reagiert haben muß. Bei Echtzeitprogrammen ist die Korrektheit nur dann gegeben, wenn zusätzlich zur Funktionskorrektheit garantiert ist, daß die Zeitvorgaben eingehalten werden.

Echtzeitsysteme findet man in vielen Bereichen heutiger Technik, z. B. in Verkehrsleitsystemen (für Autobahnen und Flughäfen), Atomkraftwerken, Echtzeitsimulationen und Spielen und in Steuerungssystemen aller Art. Steuerungssysteme unterschiedlichster Komplexität reichen von zivilen Anwendungen wie medizinischen Geräten (etwa Überwachung von Gesundheitszuständen), Telekommunikationsanlagen oder Bildverarbeitung (Entschlüsselung für Bezahlfernsehen, Dekompression von Videos usw.) bis hin zu militärischen Anwendungen wie Raketensteuerungen oder automatischen Zielsuchsystemen.

Man unterscheidet die *schwachen Echtzeitsysteme*, in denen die vorhandenen Zeitschranken einen Vorschlag darstellen, also eine Überschreitung zwar unerwünscht ist, aber keine Gefahr darstellt (etwa Videofilmdekompression, bei der fehlende Einzelbilder zwar störend sind, aber nicht gefährlich), und die *harten Echtzeitsysteme*, in denen eine Überschreitung der Zeitschranken fatale Folgen haben kann (etwa Überwachungsanlagen für Atomkraftwerke). In harten Echtzeitsystemen muß also unbedingt die Einhaltung der Zeitschranken gewährleistet sein.

Die Aufgabe eines Echtzeitsystems ist es, eine Menge von Programmen auszuführen, für die Zeitschranken gegeben sind, und dabei sicherzustellen, daß diese Zeitschranken eingehalten werden. Die Verfahren zum Überprüfen dieser Bedingungen nennen sich Planbarkeitsanalyse (*engl.: schedulabitity analysis*). Die Voraussetzung für die Anwendbarkeit dieser Verfahrens ist, daß die Maximallaufzeiten (*engl.: worst case execution time = WCET*) der Programme vor ihrer Ausführung bekannt sind. Vor allem bei harten Echtzeitsystemen müssen die vorhergesagten Maximallaufzeiten der Programme unbedingt *korrekt* sein. Die Laufzeitanalyse, die diese Maximallaufzeit vor Benutzung des Programmes in Echtzeitsystemen berechnen soll, muß daher auf beweisbar korrekten Analysemethoden beruhen.

Das allgemeine Problem, die Maximallaufzeit von beliebigen Programmen korrekt und vollständig zu berechnen, ist unlösbar. Aus diesem Grund muß man entweder die Korrektheit oder die Vollständigkeit aufgeben. Da die Korrektheit unabdingbar ist, muß die Vollständigkeit aufgegeben werden, und man versucht eine möglichst genaue Annäherung an die WCET zu berechnen.

Es gibt zwei Gesichtspunkte, die bei der Suche nach der Maximallaufzeit betrachten werden:

(i) *Mikroarchitekturmodellierung.* Diese modelliert das Verhalten der zugrundeliegenden Hardware, um herauszufinden, wie lange die Ausführung eines Programmpfades auf einer bestimmten Maschine benötigt.

(ii) *Programmpfadanalyse.* Diese bestimmt den längsten Pfad das Programmes. „Lang“ bezieht sich auf die Ausführzeit des Pfades, also hängt diese Analyse von den Ergebnissen der Mikroarchitekturmodellierung ab.

Die Analysen zur Berechnung der Maximallaufzeit dürfen diese auf keinen Fall unterschätzen, d. h. die vorhergesagten WCETs müssen *sicher* sein. Die Vorhersage sollte dabei möglichst genau sein.

1.2 Moderne Hardware

Heutige Rechenanlagen verwenden ausgeklügelte Hardware-Komponenten zur Beschleunigung der Ausführungszeit von Programmen, u. a. Caches, also schnelle Zwischenspeicher zwischen Prozessor und Hauptspeicher, die dazu dienen, wiederholte Zugriffe auf Speicherzellen zu beschleunigen. Eine Programmanalyse, die Caches nicht betrachtet, muß sehr pessimistisch jeden Hauptspeicherzugriff als *Cache-Miss* einstufen, d. h. annehmen, das Datum, auf das zugegriffen wird, befinde sich nicht im Cache und müsse aus dem Hauptspeicher geladen werden.

Der Geschwindigkeitsunterschied zwischen einem Zugriff auf den Hauptspeicher und auf den Cache kann erheblich sein. Bei einem handelsüblichen PC dauert ein Cache-Zugriff auf den First-Level-Cache typischerweise einen Takt, also bei 233 MHz etwa 4 ns, ein Hauptspeicherzugriff jedoch zwischen 12 und 70 ns, also bis zu 17mal so lange. Dies macht deutlich, daß es notwendig ist, das Cache-Verhalten zu analysieren, um genaue Vorhersagen der Maximallaufzeit zu finden.

Weitere Beschleunigung erreichen moderne Prozessoren durch Pipelines, die es ermöglichen, mehrere Befehle überlappt auszuführen. Jeder Maschinenbefehl wird dabei in mehreren Schritten ausgeführt, und falls keine Konflikte auftreten, kann der Prozessor verschiedene Schritte mehrerer Maschinenbefehle gleichzeitig bearbeiten. Eine solche Prozessor-Pipeline muß ebenfalls analysiert werden, da sonst angenommen werden muß, daß keine Überlappung der Maschinenbefehle möglich ist. Diese pessimistischen Annahmen über Cache und Pipelines führen meist zu stark überschätzten Maximallaufzeiten, wenn korrekte Methoden verwendet werden, wodurch das Echtzeitsystem Systemressourcen verschwenden muß. Daher soll die Überschätzung der Laufzeiten durch eine möglichst genaue Modellierung der Hardware minimiert werden.

Aufgrund der schlechten Ergebnisse der bisherigen Approximation von WCETs gab und gibt es die Vorgehensweise, die Ausführzeit von Echtzeitprogrammen mehrfach mit verschiedenen Eingaben zu messen, um dadurch die WCET abzuschätzen. Die Möglichkeit, die Ausführzeit für alle Eingaben zu messen, ist meist nicht gegeben, da dies extrem viele Messungen erfordern würde. Daher muß die WCET i. a. aus wenigen Messungen hergeleitet werden. Diese Vorgehensweise ist für harte Echtzeitsysteme extrem gefährlich, da die so geschätzte Laufzeit keineswegs eingehalten werden *muß*. Daher ist es dringend nötig, korrekte Analysetools zu entwickeln, die trotzdem enge Schranken berechnen.

1.3 Bekannte Analysemethoden

Cache-Analysen wurden in [Fer97b, FMW97, FMW98, Fer97a, AFMW96] ausführlich beschrieben. Die Ergebnisse, die diese Cache-Analysen liefern, werden von der in dieser Arbeit beschriebenen Laufzeitanalyse benutzt, um die Suche nach schlechtesten Programmpfaden zu implementieren. Eine Erweiterung um die Benutzung der Ergebnisse einer Pipeline-Analyse ließe sich ebenfalls leicht in das hier beschriebene Analysetool einbauen.

Wie sich in früheren Werken gezeigt hat (*siehe* [LMW95b, LMW95a, LMW96]), kann lineares Programmieren dazu benutzt werden, elegant das Problem der Programmpfadsuche zu formulieren. Aus sehr lokalen Analysen des Kontrollflusses können Nebenbedingungen erzeugt werden, die zusammengenommen den Kontrollfluß des Programms beschreiben. Die Lösung des linearen Problems liefert dann bei Minimierung die beste Laufzeit, bei Maximierung die schlechteste Laufzeit. Letztere ist für Echtzeitsysteme besonders interessant. Wie sich ebenfalls gezeigt hat (*siehe* [LMW96, OS97]), eignet sich diese Methode schlecht zur Modellierung moderner Mikroarchitekturen mit Caches und Pipelines, da die Menge der Nebenbedingungen sehr schnell ein Maß überschreitet, das aufgrund sehr hoher Lösezeiten die Analyse praktisch unmöglich macht.

1.4 Ziel dieser Arbeit

Das Ziel der Arbeit war es, zwei Ansätze bei der Analyse zusammenzuführen. Wie sich gezeigt hat, eignet sich die abstrakte Interpretation (*siehe* [CC77a]) sehr gut zur Modellierung von Mikroarchitekturen (*siehe* [Fer97b]). Andererseits liefert eine Pfadanalyse mittels linearen Programmierens sehr genaue Ergebnisse (*siehe* [LMW96]). Daher sollten beide in aufeinanderfolgenden Analyseschritten verknüpft werden, um eine effiziente, genaue und korrekte Pfadanalyse unter Berücksichtigung des Verhaltens der Mikroarchitektur zu erhalten.

Da die Pfadanalyse dazu dient, einem Echtzeitsystem Vorhersagen zu geben, die möglichst knapp oberhalb der Maximallaufzeit des Programmes liegen, sollten nicht die engen Grenzen, die die Analyse liefert, durch Verwendung eines speziell angepaßten Compilers, der möglicherweise keine Codeoptimierung bietet, wieder zunichte gemacht werden. Das Analysetool sollte Ausgaben von Standardcompilern (etwa `gcc`) akzeptieren, um den optimierten, schnellen Code zu analysieren, und gute Approximationen der Maximallaufzeiten für diesen Code berechnen statt für nichtoptimierten, langsamen Code.

1.5 Vorgehensweise

Die Eingabe für die Programmpfadanalyse besteht aus dem Kompilat des Programmes mit Debug-Informationen zum Auffinden von Quellcodezeilen und aus durch die Mikroarchitekturanalyse gewonnenen Ausführzeiten von Befehlen auf einer idealisierten Maschine. Da noch keine vollständige Mikroarchitektursimulation zur Verfügung stand, mußte eine idealisierte Hardware zugrundegelegt werden. Sobald eine solche Modellierung Ausführzeiten auf realen Prozessoren liefern kann, können diese Ergebnisse direkt als Eingabe für die hier beschriebene Programmpfadsuche dienen.

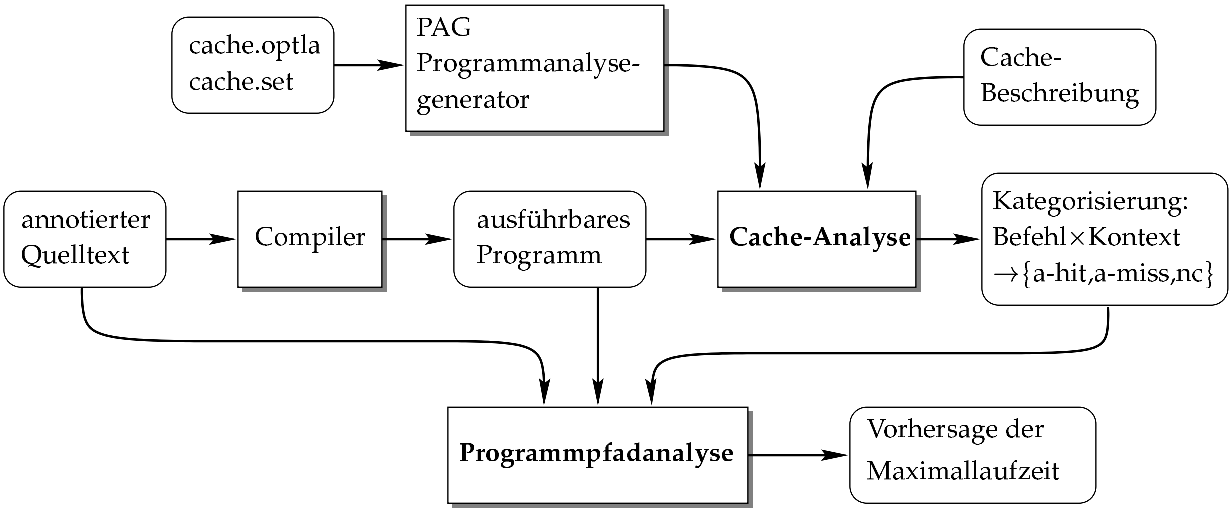


Abbildung 1.1: Fluß der Daten bei der Programmpfadanalyse.

Eine anschauliche Darstellung, wie die Analyse ablaufen wird, sieht man in Abbildung 1.1. Die mit dem Programmanalysegenerator PAG (*siehe* [TMLA98]) erzeugte Cache-Analyse liefert eine Kategorisierung des Cache-Verhaltens für jeden Maschinenbefehl des Programmes in jedem Ausführkontext. Diese Kategorisierung wird die Grundlage für die Pfadanalyse sein, den schlechtesten Pfad zu berechnen und damit die Maximallaufzeit des Programmes vorauszusagen.

1.6 Gliederung

Die vorliegende Arbeit gliedert sich wie folgt:

Zweites Kapitel: Grundlagen. Hier werden die formalen, mathematischen Grundlagen vorgestellt, die später benötigt werden. Dazu gehören eine Einführung ins lineare Programmieren und die Beschreibung der zugrundeliegenden Mikroarchitekturanalyse. Diese beschränkt sich im Moment auf eine Cache-Analyse.

Drittes Kapitel: Erzeugung des linearen Problems. Ab Seite 33 wird beschrieben, wie das lineare Problem aus dem Kontrollflußgraphen des untersuchten Programmes hergeleitet wird, und wie die Ergebnisse der Mikroarchitekturanalyse in die Analyse einfließen.

Viertes Kapitel: Annotationssprache. In diesem Kapitel wird eine Sprache beschrieben, mit dem der Benutzer Eigenschaften des analysierten Programmes beschreiben kann. Es wird gezeigt, wie der Sprachaufbau die aufgetretenen Probleme löst.

Fünftes Kapitel: Implementierung. Dieses Kapitel beschreibt ab Seite 69, wie praktisch vorgegangen wurde, um die theoretischen Ansätze in ein Analysetool einzubauen, und wie dieses Analysetool benutzt werden kann, um die Laufzeit von Programmen möglichst einfach zu berechnen.

Sechstes Kapitel: Ergebnisse. Anschließend folgt ab Seite 95 eine Diskussion der Ergebnisse und ein Ausblick auf mögliche Erweiterungen des Verfahrens und der Implementierung.

Siebtes Kapitel: Vergleich und Ausblick. Den Abschluß der Arbeit bildet eine kurze Zusammenfassung mit Vergleich mit anderen Analysemethoden.

Zweites Kapitel
Grundlagen

In diesem Kapitel sollen die Grundlagenbegriffe formal definiert werden, auf denen die folgenden Kapitel aufbauen. Die verwendeten Notationen lassen sich im Symbolverzeichnis ab Seite 109 nachschlagen.

2.1 Programme

In diesem Abschnitt werden wichtige Begriffe eingeführt, die beim Umgang mit dem untersuchten Programm nötig sind. Es werden die Strukturen erklärt, in die sich ein Programm gliedert, und der Leser erfährt, unter welchen Bezeichnungen und Namen auf diese in dieser Arbeit Bezug genommen wird.

2.1.1 Programm und Maschinenbefehle

Die Analyse bearbeitet ein Programm $\mathcal{P}$, das als Folge von Maschinenbefehlen vorliegt. Diese Folge wird dargestellt als Menge von Tupeln aus Adressen $\subseteq \mathbb{N}_0$ und Maschinenbefehlen. Die Menge I dieser Tupel wird abhängig vom Kontrollfluß des Programmes in Basisblöcke eingeteilt.

2.1.2 Basisblöcke

Der Kontrollfluß des untersuchten Programmes wird durch Sprungbefehle und Prozeduraufrufe definiert. Diese Sprünge teilen das Programm in Basisblöcke ein. Ein Basisblock ist dabei eine Folge von Befehlen, bei der der Kontrollfluß am Anfang eintritt und am Ende wieder herausfließt, ohne die Möglichkeit des Anhaltens oder der Verzweigung außer am Ende der Folge.

Die Menge der Basisblöcke heiße B und bestehe aus disjunkten Teilmengen von I:

$$B \subseteq \mathfrak{P}(I) \quad \text{wobei für } b_1, b_2 \in B \text{ gilt:} \quad b_1 \neq b_2 \iff b_1 \cap b_2 = \emptyset. \tag{2.1}$$

Des weiteren sei gefordert, daß jeder Maschinenbefehl einem Basisblock angehöre:

$$\bigcup_{b \in B} b = I \tag{2.2}$$

2.1.3 Prozeduren

Die Einteilung eines Programmes in kleine Programmstücke dient der Strukturierung und der Möglichkeit der Wiederverwendung mehrfach benutzter ähnlicher Abläufe. In dieser Arbeit heißen diese Programmstücke *Prozeduren*, um sie von mathematischen Funktionen zu unterscheiden. In anderer Literatur finden sich z. B. auch die Begriffe *Funktion* und *Routine*.

Sei $\mathcal{F}$ die Menge der Prozeduren von $\mathcal{P}$. Jeder Basisblock gehört zu einer dieser Prozeduren. Die Funktion Φ ordne nun jedem Basisblock b die Prozedur zu, zu der b gehört:

$$\Phi : B \to \mathcal{F} \tag{2.3}$$

Die Prozedur, die als erstes im Programm aufgerufen wird (das *Hauptprogramm*) werde mit φ_{start} bezeichnet.

Zusätzlich sind diejenigen Basisblöcke interessant, die einen Aufruf einer Prozedur enthalten. Sei dazu folgende Funktion definiert:

$$\begin{aligned} f_{call} \quad : \quad B \quad &\to \quad B \cup \{nocall\} \\ b \quad &\mapsto \quad \begin{cases} nocall & \text{falls } b \text{ keinen Prozeduraufruf enthält,} \\ b' \in B & \text{falls } b' \text{ von } b \text{ aufgerufen wird} \end{cases} \end{aligned} \tag{2.4}$$

Zusätzlich sei die Menge der Basisblöcke, die Prozeduraufrufe enthalten, $\mathcal{C}$ von $\mathcal{P}$, wie folgt definiert:

$$\mathcal{C} := \{b \in B \mid f_{call}(b) \neq nocall\} \tag{2.5}$$

Definition 2.1 (Call-Graph)
Der *Call-Graph* von $\mathcal{P}$ ist ein Tripel $\hat{G} = (\mathcal{F}, \hat{E}, \varphi_{start})$ mit

$$\hat{E} = \{(\varphi, \varphi') \in \mathcal{F} \times \mathcal{F} \mid b \in \mathcal{C} \text{ und } \varphi = \Phi(b) \text{ und } \varphi' = \Phi(f_{call}(b))\} \text{ und} \tag{2.6}$$
$$\varphi_{start} \in \mathcal{F} \text{ und} \tag{2.7}$$
$$\varphi_{start} \notin \Phi(f_{call}(\mathcal{C})). \tag{2.8}$$

Der Call-Graph ist der gerichtete Graph über Funktionen, dessen Kanten durch Prozeduraufrufe definiert werden. φ_{start} werde als einzige Prozedur nicht von anderen Funktionen aufgerufen, d. h. es muß sichergestellt sein, daß bei einem Programmlauf das Hauptprogramm genau einmal ausgeführt wird.[1]

Um die Lesbarkeit zu erhöhen, sollen Kanten von v_1 nach v_2 statt als (v_1, v_2) als $[v_1, v_2\rangle$ bezeichnet werden, wodurch die Richtung der Kante deutlich wird.

Die Notation $f(A)$ mit $f : A \to B$ soll die Menge ausgeben, die die durch Abbildung aller Elemente von A entsteht. Also ist z. B.

$$f_{call}(\mathcal{C}) := \{b' \mid b \in \mathcal{C}, b' = f_{call}(b)\}.$$

Eine ähnliche Notation wird im folgenden auch für Operatoren verwendet. Für $\diamond : A \times B \to C$ sei

$$a \diamond B := \{c \mid b \in B, c = a \diamond b\}.$$

2.1.4 Kontrollflußgraph

Die Basisblöcke werden im Kontrollflußgraphen als Knoten angesehen. Jede Prozedur hat genau einen Kontrollflußgraphen.

Definition 2.2 (Kontrollflußgraph)
Ein Kontrollflußgraph ist ein Tripel $G = (V, E, v_0)$, wobei (V, E) ein gerichteter Graph, V die Menge der Knoten, $E \subseteq V \times V$ die Menge der Kanten und $v_0 \in V$ ist, und bei dem gilt:

$$V \subseteq B \text{ und} \tag{2.9}$$
$$\forall v_1, v_2 \in V : \Phi(v_1) = \Phi(v_2) \text{ und} \tag{2.10}$$
$$\forall v \in V,\ v \neq v_0 :\ v \notin f_{call}(\mathcal{C}). \tag{2.11}$$

[1] Ist dies nicht der Fall, so läßt sich der Mangel durch Einführung einer Hilfsprozedur, die als neues Hauptprogramm fungiert und das alte Hauptprogramm aufruft, leicht beheben.

Der Basisblock v_0 heiße *Anfangsblock*. Der Anfangsblock von φ_{start} werde mit b_{start} bezeichnet. Falls keine Verwechslungen auftreten können, wird ein Kontrollflußgraph auch kurz mit (V, E) geschrieben werden.

Für alle Kontrollflußgraphen außer dem Hauptprogramm gilt: $v_0 \in f_{call}(\mathcal{C})$, d. h. jede Funktion wird aufgerufen. Das Hauptprogramm werde nicht von Prozeduren aufgerufen, da gefordert wurde, daß es genau einmal pro Programmlauf ausgeführt wird.

Sei $\mathcal{G}$ die Menge aller Kontrollflußgraphen des Programmes $\mathcal{P}$ und $\mathcal{E}$ die Menge aller Kantenmengen aller Graphen aus $\mathcal{G}$, also

$$\mathcal{G} := \bigcup_{G} G \tag{2.12}$$

und

$$\mathcal{E} := \bigcup_{G=(V,E)\in\mathcal{G}} E. \tag{2.13}$$

Nun seien zwei Funktionen definiert, die die eingehenden und ausgehenden Kanten eines Basisblocks b angeben:

$$\begin{array}{rcccl} in & : & B & \to & \mathfrak{P}(\mathcal{E}) \\ & & b & \mapsto & \{[b', b\rangle \in \mathcal{E}\} \end{array} \tag{2.14}$$

$$\begin{array}{rcccl} out & : & B & \to & \mathfrak{P}(\mathcal{E}) \\ & & b & \mapsto & \{[b, b'\rangle \in \mathcal{E}\} \end{array} \tag{2.15}$$

Es wird gefordert, daß keine bedingten Prozeduraufrufe auftreten, d. h. daß jeder Basisblock $b \in \mathcal{C}$ garantiert genausooft ausgeführt wird wie die Kante $[b, f_{call}(b)\rangle$ durchlaufen. Des weiteren wird vorausgesetzt, daß die Prozedurrücksprünge sich im Kontrollflußgraphen des Aufrufers durch entsprechende Kanten (*engl.: fall through edges*) offenbaren. Abbildung 2.1 zeigt die Kontrollflußgraphen und Aufrufkanten für einen bedingten Prozeduraufruf.

Um den gesamten Kontrollfluß einschließlich der Prozeduraufrufe eines Programmes in einem Graphen zu vereinen, sei dieser Graph wie folgt definiert:

Definition 2.3 (Gesamtkontrollflußgraph)
Der Gesamtkontrollflußgraph $\mathfrak{G}$ sei definiert als

$$\mathfrak{G} := (B, \mathfrak{E}) \text{ mit}$$
$$\mathfrak{E} := \mathcal{E} \cup \{[b, f_{call}(b)\rangle \mid b \in \mathcal{C}\}$$

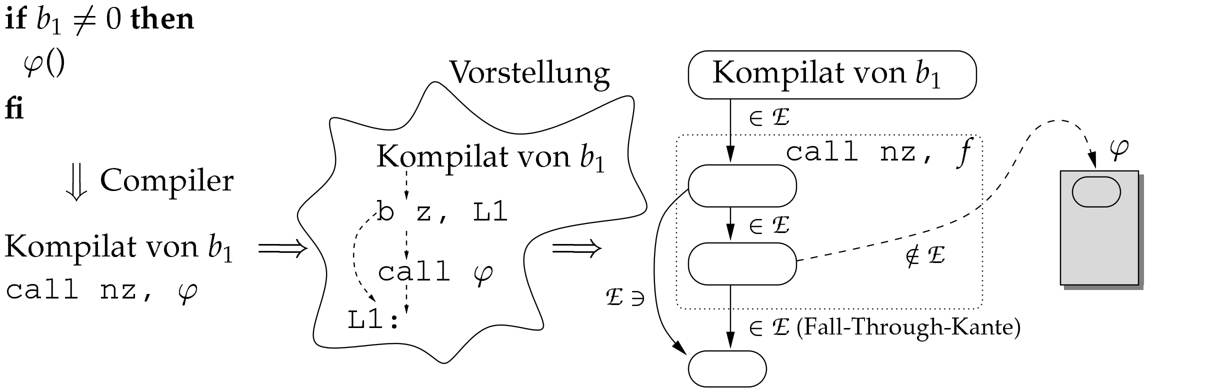


Abbildung 2.1: Kontrollflußgraph für bedingten Prozeduraufruf. Die zugrundeliegende Maschine kenne dabei den Befehl `call nz` (Prozeduraufruf, falls das letzte Ergebnis nicht Null war, etwa nach einem Vergleich auf (Un-)Gleichheit, `b z` (Sprung, falls letztes Ergebnis Null) und `call` (unbedingter Prozeduraufruf).

2.1.5 Schleifen

Eine Schleife soll im Sinne der Definition einer *natürlichen Schleife* aus [ASU86] gesehen werden. Eine natürliche Schleife hat zwei grundlegende Eigenschaften:

(i) Eine natürliche Schleife besitzt genau einen Eintrittspunkt, der *Kopf* heißt.

(ii) Die Schleife muß wiederholbar sein, d. h. es muß mindestens ein Pfad zurück zum Kopf existieren.

Die verwendete Bibliothek EEL benutzt diese Definition und einen Algorithmus aus [LT79] zum Auffinden von Schleifen.

Außer Schleifen dürfen keine anderen Zyklen in den Kontrollflußgraphen nicht vorkommen. D. h. natürlich, daß nicht alle Programme analysierbar sein werden. Zyklen im Kontrollflußgraphen sind immer problematisch bei der Suche nach dem längsten Pfad. Da nur für die wohlgeformten Zyklen Maximalanzahlen der Schleifenwiederholungen angegeben werden können, werden nur die wohlgeformten Zyklen der natürlichen Schleifen betrachtet.

Um Probleme zu vermeiden, wird vorausgesetzt, daß vor jeder Schleife ein leerer Basisblock im Kontrollflußgraphen auftaucht, der nicht zur Schleife gehört. Dieser Basisblock wird als *Pre-Header* der Schleife bezeichnet. Aller Kontrollfluß *muß* die Schleife über diesen Block betreten.

Nützlich sind diese leeren Basisblöcke zur Vermeidung von Sonderfällen an folgenden Stellen:

Prozeduranfang: Beginnt eine Prozedur mit einer Schleife, so könnte man ohne die Existenz von Pre-Headern nicht sagen, die Prozedur werde so oft ausgeführt wie der

```
repeat
  b1
until b2
repeat
  b3
until b4
```

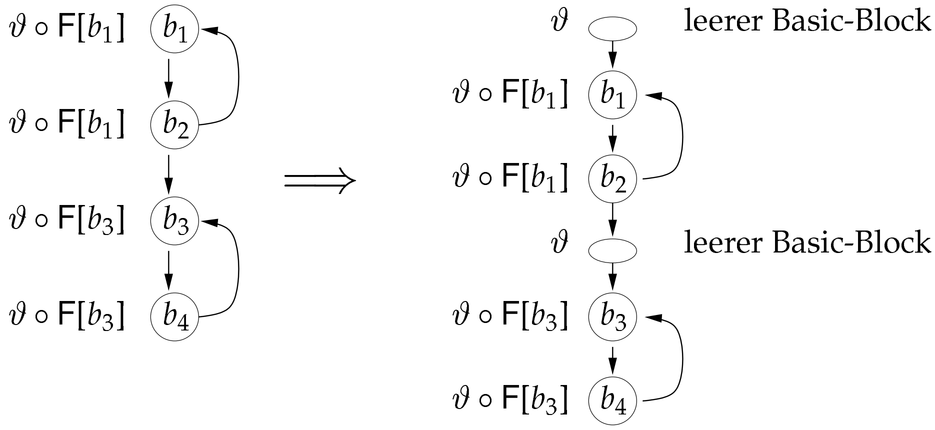


Abbildung 2.2: Einfügen der leeren Basisblöcke am Anfang von Schleifen. Als Beispielkontexte sind die ersten Schleifendurchläufe angegeben. Der Kontext, der außerhalb der Schleifen gilt, sei ϑ.

erste Basisblock der Prozedur. Der leere Basisblock vor einer prozedureinleitenden Schleife macht diese Aussage aber möglich.

Aufeinanderfolgende Schleifen: Es treten Probleme auf, wenn Kanten, die eine Schleife verlassen, direkt eine andere Schleife betreten.

Geschachtelte Schleifen: Ähnlich verhält es sich hier, wenn z. B. eine Schleife mit Abbruchbedingung am Ende direkt mit einer eingeschachtelten Schleife beginnt. Dann zeigen die Eintrittskanten der Schleife nicht auf Blöcke, die zur äußeren Schleife gehören, sondern sie überspringen zwei Ebenen und treten direkt in die innere Schleife ein.

Auf diese Probleme wird nochmal hingewiesen, wenn Sonderfälle ohne die leeren Blöcke aufträten.

Sei $\mathcal{L}$ die Menge der Schleifen des Programmes $\mathcal{P}$. Es gebe nun eine Funktion

$$head : \mathcal{L} \to B, \tag{2.16}$$

die jeder Schleife im Programm ihren Kopf, also den Basisblock, über den aller Kontrollfluß die Schleife betritt, zuordnet.

Die Funktion *pre* liefert den Pre-Header einer Schleife. Dies ist der eindeutig definierte Vorgänger von $head(l)$ außerhalb der Schleife l.

$$pre : \mathcal{L} \to B \tag{2.17}$$

Durch die Einführung des leeren Basisblocks vor jeder Schleife gibt es nun für jede Schleife genau eine Kante, die die Schleife betritt, nämlich die Kante zwischen dem leeren Block und dem Kopf der Schleife. Diese sei durch folgende Funktion gegeben:

$$entry : \mathcal{L} \to \mathcal{E} \tag{2.18}$$

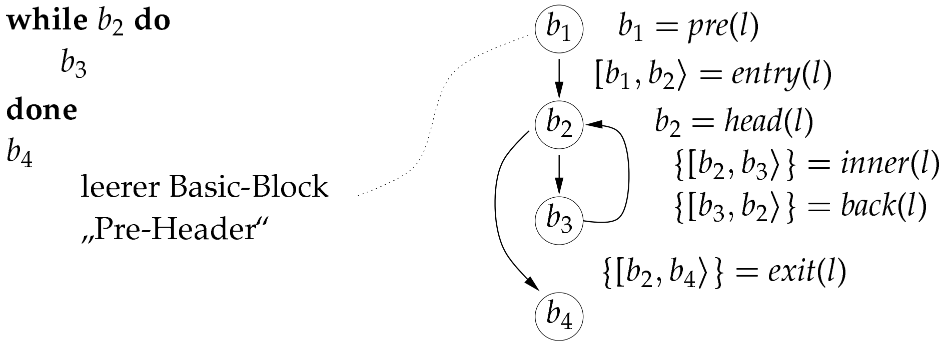


Abbildung 2.3: Eine einfache Schleife mit Abbruchbedingung am Anfang.

Weiterhin gebe es eine Funktion, die die Rückkanten einer Schleife ausgibt, d. h. diejenigen Kanten, die eingehende Kanten des Kopfes sind, die nicht vom Pre-Header kommen. Diese Kanten müssen von Basisblöcken kommen, die zur Schleife gehören.

$$back : \mathcal{L} \rightarrow \mathfrak{P}(\mathcal{E}) \tag{2.19}$$

Als letztes wird eine Funktion benötigt, die die Kanten einer Schleife ausgibt, die die Schleife verlassen.

$$exit : \mathcal{L} \rightarrow \mathfrak{P}(\mathcal{E}) \tag{2.20}$$

Im Moment sei eine Funktion als gegeben angenommen, die einen inneren Basisblock ausgibt, der so oft ausgeführt wird, die die Schleife durchlaufen. Diese Annahme ist problematisch, denn es ist nicht einfach, diesen Block zu bestimmen. Im fünften Kapitel wird genau darauf eingegangen, wie das Problem der Auffindung eines solchen Blocks gelöst wird.

$$inner : \mathcal{L} \rightarrow B \tag{2.21}$$

2.1.6 Kontexte und VIVU-Analyse

Schleifen und rekursive Prozeduraufrufe sind von besonderem Interesse, da Programme hier die meiste Laufzeit verbringen.

Schleifen werden zumeist mehrfach ausgeführt. Da der erste Durchlauf normalerweise den Cache-Inhalt stärker ändert, als weitere Durchläufe, ist es sinnvoll, den ersten von allen weiteren Durchläufen zu unterscheiden.

Bei der verwendeten Analyse werden Schleifen als Prozeduren analysiert, um existierende Verfahren zur Prozeduranalyse verwenden zu können. Abbildung 2.4 zeigt die dazu nötige Transformation der Schleifen.

⋮
while b **do**
    Rumpf
done
⋮
	$\Longrightarrow$	**procedure** $\mathrm{Loop}_1()$
		begin
		    if b **then**
		        Rumpf
		        $\mathrm{Loop}_1()$;
		    fi
		end
		⋮
		$\mathrm{Loop}_1()$;
		⋮
			
			(ii)
			
			(i)

Abbildung 2.4: Überführung einer Schleife in eine Prozedur. (i) ist der Aufruf, der die Schleife im Programm durch einen Prozeduraufruf ersetzt, und (ii) der rekursive Aufruf, der dem Rücksprung zum Anfang der Schleife entspricht.

Kommen im Programm (rekursive) Prozeduraufrufe vor, so kann jeder Befehl in verschiedenen *Kontexten* ausgeführt werden. Ein Kontext entspricht dabei dem Pfad zu dem Befehl im Call-Graphen des Programmes.

Die Prozeduranalysen unterscheiden sich in der Art, wie die Ausführkontexte eines Befehls unterschieden werden. Ein verbreitetes Verfahren ist der *Call-String*-Ansatz, dessen Anwendbarkeit auf Cache-Analysen begrenzt ist (*siehe* [Fer97b]).

Um genauere Ergebnisse zu erzielen, wurde die *VIVU-Analyse* entwickelt, die in PAG implementiert wurde (*siehe* [AM95]). Pfade im Call-Graphen, die sich nur in der Anzahl von wiederholten Durchläufen eines Zyklus unterscheiden, werden nicht gesondert analysiert. Die Ergebnisse des VIVU-Ansatzes können auf natürliche Weise mit denen einer Pfadanalyse verknüpft werden, um die Maximallaufzeit eines Programmes vorherzusagen.

Es folgt die Definition der Menge aller Kontexte T des Programmes $\mathcal{P}$. Die konzeptuell in Prozeduren verwandelten Schleifen sollen im Kontext wegen unterschiedlicher Behandlung beim späteren Erzeugen der Nebenbedingungen des linearen Problems als Schleifen deutlich werden.

Definition 2.4 (VIVU-Kontexte)
Die Menge T aller Kontexte eines Programmes $\mathcal{P}$ ist definiert als

$$T := \bar{T}^* \tag{2.22}$$

mit

$$\bar{T} := \{\mathsf{C}[b],\ \mathsf{R}[b],\ \mathsf{F}[b'],\ \mathsf{O}[b'] \mid b \in \mathcal{C},\ b' \in \mathit{head}(\mathcal{L})\}. \tag{2.23}$$

$$
\begin{array}{ll}
\textbf{procedure}\ a(i : \mathit{integer}); & \\
\textbf{begin} & \\
\quad b_1; & \{\mathsf{C}[b_4], \mathsf{C}[b_5]\} \\
\quad \textbf{while}\ b_2\ \textbf{do} & \\
\qquad b_3; & \{\mathsf{C}[b_4] \circ \mathsf{F}[b_2],\ \mathsf{C}[b_5] \circ \mathsf{F}[b_2], \\
 & \ \mathsf{C}[b_4] \circ \mathsf{O}[b_2],\ \mathsf{C}[b_5] \circ \mathsf{O}[b_2]\} \\
\quad \textbf{done} & \\
\textbf{end}; & \\
 & \\
\textbf{begin} & \\
\quad b_4 : a(3); & \{\varepsilon\} \\
\quad b_5 : a(5); & \\
\textbf{end}. &
\end{array}
$$

Abbildung 2.5: Unterscheidung nach Kontexten in einem einfachen Programm. Im Hauptprogramm sind die Basisblöcke der Aufrufe von a zur Verdeutlichung mit b_4 und b_5 benannt.

Die Elemente von $\bar{T}$ geben an, ob der Kontrollfluß durch die erste Iteration einer Schleife zum untersuchten Befehl gelangt ist ($\mathsf{F}[b]$), oder durch eine der weiteren Iterationen ($\mathsf{O}[b]$), ob ein erster ($\mathsf{C}[b]$) oder rekursiver Aufruf ($\mathsf{R}[b]$) auf dem Pfad lag.

Die Elemente von T beschreiben Folgen von Basisblöcken, in denen (erste und rekursive) Prozeduraufrufe oder (erste oder weitere) Schleifeneintritte vorkommen.

Der leere Kontext werde mit ε bezeichnet.

Abbildung 2.5 zeigt im Beispiel, wie Basisblöcke nach Kontexten unterschieden werden.

Die Funktion

$$\tau : B \to \mathfrak{P}(T) \tag{2.24}$$

ordne jedem Basisblock die Menge der Kontexte zu, in denen dieser Basisblock vorkommt. Auch den Kanten der Kontrollflußgraphen werden Kontexte zugeordnet. Es wird festgelegt, daß die Kanten in den Kontexten des Quellknotens analysiert werden. Die Funktion τ wird daher wie folgt definiert:

$$\begin{array}{rcccl} \tau & : & E & \to & \mathfrak{P}(T) \\ & & [b, b'\rangle & \mapsto & \tau(b) \end{array} \tag{2.25}$$

2.2 Lineares Programmieren

In diesem Abschnitt wird kurz eingeführt, was lineares Programmieren ist, und wie man lineare Probleme lösen kann. Oft wird darauf verzichtet werden, die Aussagen und Sätze zu beweisen, denn es geht nur um die Einführung der Grundbegriffe – Beweise findet man z. B. in [Chv83, Sch96a, NW88, Sei96].

2.2.1 Lineare Probleme

Zunächst werden Begriffe zum allgemeinen linearen Programmieren eingeführt. Im nächsten Abschnitt wird das ganzzahlige lineare Programmieren besprochen.

Definition 2.5 (Vergleichsoperationen auf Vektoren)
Sei $\Delta \in \{\leqq, =, \geqq\}$ ein Vergleichsoperator und $a, b \in \mathbb{R}^n$. Dann sei

$$a \,\Delta\, b :\Longleftrightarrow a_i \,\Delta\, b_i, \ \forall i = 1, \ldots, n$$

Definition 2.6 (Linearkombination)
Sei $x \in \mathbb{R}^n$ variabel und $a \in \mathbb{R}^n$ konstant. Dann heißt $a^T x$ eine Linearkombination von x.

Definition 2.7 (Lineares Problem)
Seien $t \in \mathbb{R}^d, \ b \in \mathbb{R}^m, \ A \in \mathbb{R}^{m \times d}$ konstant gegeben. Ein *lineares Problem* nennt man die Aufgabe, $t^T x$ so zu maximieren, daß $x \in \mathbb{R}^d_{\geqq 0}$ und $A\,x \leqq b$ gilt. Oder kurz geschrieben:

$$\mathsf{max} : t^T x \mid A\,x \leqq b, \ x \in \mathbb{R}^d_{\geqq 0}.$$

„Lineares Problem" in all seinen Beugungsformen kürzt man oft als *LP* ab.

Definition 2.8
In Definition 2.7 heißt die Funktion $C : \mathbb{R}^d \to \mathbb{R}$ mit $C(x) = t^T x$ *Zielfunktion*. Die Gleichungen, die durch $A\,x \leqq b$ gegeben sind, heißen *Nebenbedingungen*. Ein x heißt *zulässig*, falls $A\,x \leqq b$. Die Menge $P = \{x \in \mathbb{R}^d_{\geqq 0} \mid A\,x \leqq b\}$ heißt *Zulässigkeitsbereich* von x. Ein x^* heißt *optimal*, falls $t^T x^* = \mathsf{max}\{t^T x \mid x \in P\}$.

Die Nebenbedingungen, die in der Definition des linearen Problems erlaubt sind, haben im Moment die Form $a_{k,1}\,x_1 + a_{k,2}\,x_2 + \cdots + a_{k,d}\,x_d \leqq b_k$. Formen der Art

$$a_{k,1}\,x_1 + \cdots + a_{k,d}\,x_d \ \ \Delta \ \ a'_{k,1}\,x'_1 + \cdots + a'_{k,e}\,x'_d \tag{2.26}$$

mit $\Delta \in \{\leqq, =, \geqq\}$ lassen sich auf die Grundform reduzieren, indem man folgende Transformationen vornimmt:

(i) Statt $\sum_1 \, \Delta \sum_2$ schreibe $\sum_1 - \sum_2 \, \Delta \, 0$.

(ii) Statt $\sum \geqq 0$ schreibe $-\sum \leqq 0$.

(iii) Statt $\sum = 0$ schreibe $\sum \leqq 0$ und füge die Nebenbedingung $-\sum \leqq 0$ hinzu.

Es gibt drei Fälle, die bei dem Versuch, das lineare Problem zu lösen, auftreten können:

(i) $P = \emptyset$: Das Problem ist *unlösbar*.

(ii) $P \neq \emptyset$, aber $\nexists \mathsf{sup}\{t^T x \mid x \in P\}$. Dann ist das Problem *unbeschränkt*.

(iii) $P \neq \emptyset$, und $\exists \mathsf{max}\{t^T x \mid x \in P\}$. Dann ist das Problem *lösbar* und hat eine endliche Lösung.

Um Lösungen für ein lineares Problem zu finden, ist es nützlich, obere Schranken für die Zielfunktion angeben zu können. Die kleinste obere Schranke zu finden, ist ebenfalls ein lineares Problem, das wie folgt definiert ist.

Definition 2.9 (Primales und duales Problem)
Sei ein lineares Problem $\mathsf{max} : t^T x \mid A\,x \leqq b,\ x \in \mathbb{R}^d_{\geqq 0}$ gegeben. Dieses Problem heiße *primales Problem*. Das dazu *duale Problem* ist das Problem des Auffindens der kleinsten oberen Schranke von $t^T x$, das wie folgt lautet: bestimme $\mathsf{min} : y^T b \mid y^T A \geqq t,\ y \in \mathbb{R}^d_{\geqq 0}$.

Es gelten die folgenden zwei Sätze.

Satz 2.1 (Schwache Dualität)
Sei $\bar{x}$ eine zulässige Lösung des primalen Problems $\mathsf{max} : t^T x \mid A\,x \leqq b,\ x \in \mathbb{R}^d_{\geqq 0}$ und $\bar{y}$ eine zulässige Lösung des dazu dualen Problems $\mathsf{min} : y^T b \mid y^T A \geqq t,\ y \in \mathbb{R}^d_{\geqq 0}$. Dann gilt:

$$\bar{y}^T b \geqq t^T \bar{x}.$$

Satz 2.2 (Starke Dualität)
Sei x^* eine zulässige Lösung des primalen Problems $\mathsf{max} : t^T x \mid A\,x \leqq b,\ x \in \mathbb{R}^d_{\geqq 0}$ und y^* eine zulässige Lösung des dazu dualen Problems $\mathsf{min} : y^T b \mid y^T A \geqq t,\ y \in \mathbb{R}^d_{\geqq 0}$. Dann gilt:

$$y^{*T} b = t^T x^* \iff x^* \text{ und } y^* \text{ sind optimal.}$$

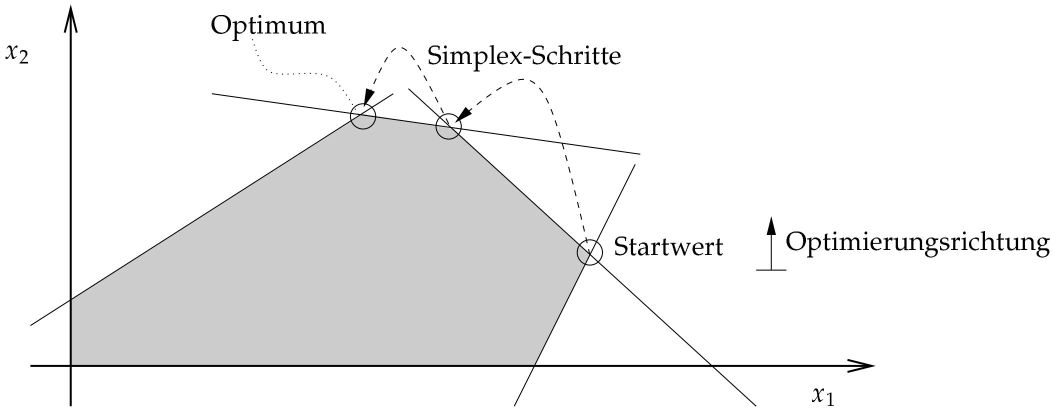


Abbildung 2.6: Veranschaulichter Simplex-Algorithmus im $\mathbb{R}^2_{\geqq 0}$.

Folgerungen

- Ist das primale Problem unbeschränkt, so ist das duale Problem unlösbar.
- Gibt es zulässige Lösungen für primales und duales Problem, so gibt es eine optimale Lösung. Die Werte der Zielfunktionen der beiden Probleme sind bei der optimalen Lösung gleich.

Der folgende *Simplex*-Algorithmus nutzt nun aus, daß sich zu einer zulässigen Basislösung x des *primalen* Problems eine *duale* Lösung y finden läßt. Ist diese Lösung im dualen Problem zulässig, so ist diese Lösung optimal. Falls nicht, kann man die Basislösung verschieben, um den Wert der Zielfunktion zu verbessern (*siehe* [Sei96]).

2.2.2 Simplex-Algorithmus

In diesem Abschnitt soll eine kurze nichtformale Beschreibung des Simplex-Algorithmus gegeben werden. Genaue formale und geometrische Beschreibungen des Algorithmus finden sich z. B. in [Sei96, Chv83].

Die Nebenbedingungen eines linearen Problems grenzen ein konvexes Gebiet im $\mathbb{R}^n_{\geqq 0}$ ein. Die optimale Lösung ist in einem der Eckpunkte dieses Gebietes zu suchen. Ausgehend von einem beliebigen Eckpunkt springt man dann über eine Gerade zu einem Nachbarpunkt, falls dieser einen besseren Wert der Zielfunktion ergibt. Dies wiederholt man so lange, bis keiner der Nachbarpunkte bessere Werte ergibt. Die optimale Lösung ist dann gefunden. Abbildung 2.6 illustriert das Verfahren.

Der Simplex-Algorithmus läßt sich auch für große Probleme einsetzen, da er für die meisten Anwendungen $O(m)$ Zeit benötigt mit m Anzahl der Nebenbedingungen. Man kann jedoch Probleme konstruieren, in denen der Algorithmus $O(2^m)$ Zeit benötigt (z. B.

der Klee-Minty-Würfel (*siehe* [Chv83])). Es gibt komplexitätstheoretisch bessere Algorithmen, z. B. die Ellipsoid-Methode oder den projektiven Skalierungsalgorithmus von Karmarker, die polynomielle Laufzeit haben.

2.2.3 Ganzzahlige lineare Probleme

In dieser Arbeit werden keine beliebigen linearen Probleme erzeugt, sondern nur ganzzahlige. Ganzzahlig bedeutet, daß in Definition 2.7 auf Seite 26 gilt: $x \in \mathbb{Z}^d$. Da schon vorher angenommen wurde, daß $x \geqq 0$, ist sogar $x \in \mathbb{N}_0^d$. Diese Einschränkung war für das zu lösende Problem natürlicherweise gegeben, da die x_i Ausführanzahlen von Basisblöcken angeben werden.

Definition 2.10 (Ganzzahliges lineares Problem)
Seien $t \in \mathbb{R}^d,\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times d}$ konstant gegeben. Ein *ganzzahliges lineares Problem* nennt man die Aufgabe, $t^T x$ so zu maximieren, daß $x \in \mathbb{N}_0^d$ und $A\,x \leqq b$ gilt. Oder kurz geschrieben:

$$\mathsf{max} : t^T x \mid A\,x \leqq b,\ x \in \mathbb{N}_0^d.$$

„Ganzzahliges lineares Problem“ (*engl.: integer linear program*) kürzt man in all seinen Beugungsformen häufig als *ILP* ab.

Eine Lösung für ganzzahlige lineare Probleme zu finden bedarf weiterer Schritte, denn der Simplex-Algorithmus ist nicht ohne weiteres anzuwenden, da die zusätzliche Forderung nach Ganzzahligkeit das Problem erheblich erschwert. ILPs zu lösen ist $\mathcal{NP}$-schwer. Viele große Probleme der Praxis sind jedoch mit moderatem Aufwand lösbar.

Definition 2.11 (LP-relaxiertes Problem)
Sei $\mathsf{max} : t^T x \mid A\,x \leqq b,\ x \in \mathbb{N}_0^d$ ein ganzzahliges lineares Problem. Dann heißt $\mathsf{max} : t^T x \mid A\,x \leqq b,\ x \in \mathbb{R}_{\geqq 0}^d$ das dazu *LP-relaxierte Problem*. Im folgenden wird es auch kurz als *relaxiertes Problem* bezeichnet.

Das relaxierte Problem (*siehe* Abbildung 2.7) wird benutzt, um Lösungen von ILPs zu bestimmen. Es enthält keine Forderung nach Ganzzahligkeit mehr, und alle zulässigen Lösungen des ILPs sind auch Lösungen des relaxierten Problems. D. h. ausgehend von einer Lösung des relaxierten Problems versucht man, die Forderung nach Ganzzahligkeit zu erfüllen. Der folgende Algorithmus geht so vor.

2.2.4 Branch-and-Bound-Algorithmus

Die Grundidee des Branch-and-Bound-Algorithmus ist es, zunächst das relaxierte Problem zu lösen, dann den Zulässigkeitsbereich unter Berücksichtigung der optimalen

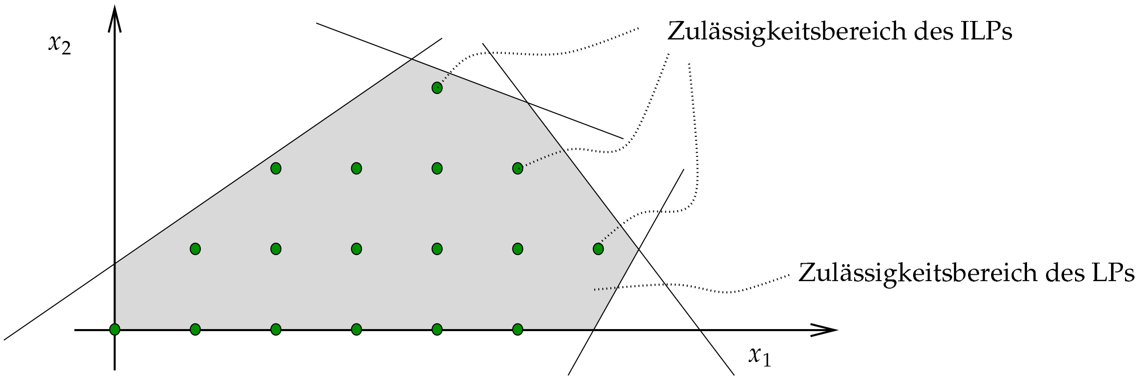


Abbildung 2.7: Zulässigkeitsbereichs eines ILPs im $\mathbb{N}_0^2$ (Gitterpunkte) und zugehöriger Zulässigkeitsbereich des LP-relaxierten Problems (grau unterlegt).

Lösung des relaxierten Problems in zwei eingeschränkte Teilprobleme zu zerlegen, um die Ganzzahligkeit zu fordern und dann die neuen Probleme auf die gleiche Art wiederholt zu lösen, bis alle Variablen ganzzahlig sind.

Sei Ψ ein ILP und Ψ' das dazu relaxierte Problem. Löst man Ψ' mit dem Simplex-Algorithmus, so erhält man, falls das Problem lösbar ist, eine Lösung $\hat{x} \in \mathbb{R}^d_{\geqq 0}$.

Falls $\hat{x} \in \mathbb{Z}^d$, so ist die Lösung von Ψ gefunden. Falls nicht, so wählt man eine Koordinate $i \in \{1, \ldots, n\}$, für die gilt: $\hat{x}_i \notin \mathbb{Z}$. Dann erzeugt man aus Ψ' zwei neue Probleme $\tilde{\Psi}_1$ und $\tilde{\Psi}_2$, indem man jeweils eine der folgende Ungleichungen hinzufügt:

$$x_i \leqq \lfloor \hat{x}_i \rfloor \tag{2.27}$$
$$x_i \geqq \lceil \hat{x}_i \rceil \tag{2.28}$$

Dadurch schließt man $\hat{x}$ in der Lösung von $\tilde{\Psi}_1$ und $\tilde{\Psi}_2$ aus. Man wiederholt nun das Verfahren für diese beiden Teilprobleme, bis alle Variablen ganzzahlig sind.

Diese Beschreibung des Branch-and-Bound-Algorithmus schweigt sich hier über wesentliche Probleme und Methoden aus, etwa über die Wahl der Koordinate oder die Reihenfolge, in der neu entstehende Probleme gelöst werden sollen. Genaueres erfährt man z. B. in [Sei96].

2.3 Mikroarchitekturanalyse

In diesem Abschnitt wird die in [Fer97b] ausführlich beschriebene Cache-Analyse, deren Ergebnisse in den nachfolgenden Kapiteln benötigt werden, eingeführt. Da sich diese Arbeit nur der Ergebnisse der Mikroarchitekturanalyse bedient, soll eine knappe Beschreibung der zugrundeliegenden Analysemethoden genügen.

2.3.1 Abstrakte Interpretation

Abstrakte Interpretation wird zur Programmanalyse benutzt, um Eigenschaften von Programmen zu bestimmen, ohne das Programm eventuell mit allen möglichen Eingaben ausführen zu müssen. Die errechneten Eigenschaften sind z. B. für optimierende Compiler interessant (*siehe* [WM95]) oder für die Planbarkeitsanalyse in Echtzeitsystemen.

Die grundlegende Idee der abstrakten Interpretation wurde von Patrick und Radhia Cousot in [CC77a] vorgestellt. Bei der abstrakten Interpretation werden, um das Programm nicht ausführen zu müssen, die konkreten Werte *beschrieben*, und statt mit den tatsächlichen Werten zu rechnen, wird mit den abstrakten Beschreibungen gerechnet.

Ein Grund für die Benutzung abstrakter Interpretation ist, daß die Ergebnisse der Analyse in endlicher Zeit berechenbar sein sollen. Durch geeignete Beschreibungen läßt sich das sicherstellen. Des weiteren möchte man für *mehrere* Eingaben Ergebnisse erhalten, im Falle der Mikroarchitekturanalyse für *alle*.

Die Beschreibung der Werte kann unterschiedlich genau sein: ist z. B. eine ganzzahlige Variable garantiert 7, so kann man sie als *7* beschreiben. Weiß man, daß die Werte 7, 9 oder 10 möglich sind, könnten mögliche Beschreibungen lauten: *genauer Wert unbekannt*, *Wert zwischen 7 und 10* oder *Wert entweder 7, 9 oder 10*. All diese Beschreibungen sind korrekt, nur unterschiedlich genau.

Für die Mikroarchitekturanalyse müssen die Ergebnisse der Analyse garantiert korrekt sein. Daher muß man die abstrakten Beschreibungen der Werte so wählen, daß dies gewährleistet ist.

Die abstrakten Zustandsbeschreibungen an einem Programmpunkt werden mit der *abstrakten kollektiven Semantik* berechnet. Diese faßt die abstrakten Zustände zusammen, die an diesem Programmpunkt überhaupt entstehen können, wenn der Kontrollfluß über verschiedene Pfade vom Einsprungpunkt zu diesem Programmpunkt fließt.

Wie die abstrakte kollektive Semantik die abstrakten Zustände eines Programmpunktes berechnet, kann in [CC77b, Fer97b] im Detail nachgelesen werden.

2.3.2 Analyseablauf mit PAG

Die PAG-Analyse (*siehe* [TMLA98, AMW95]) mittels abstrakter Interpretation arbeitet entlang des Kontrollflußgraphen des Programmes. Neue abstrakte Werte werden mittels der *Update-Funktion* zur Gesamtinformation hinzugefügt. Wenn der Kontrollfluß von zwei verschiedenen Punkten im Graphen zusammenfließt, berechnet die *Join-Funktion* die abstrakten Werte anhand der einzelnen abstrakten Vorgängerwerte. Das verwendete Verfahren wird in [Mar95] beschrieben und als korrekt bewiesen.

Mit dem Programmanalyse-Generator PAG lassen sich die Menge der abstrakten Werte, die Update- und die Join-Funktion definieren und ein Analysewerkzeug erzeugen, das Programme mittels abstrakter Interpretation analysieren kann.

Die folgende Cache-Analyse bedient sich PAGs zur Modellierung des Caches der Maschine.

2.3.3 Cache-Analyse

Auf modernen Prozessoren ist der Cache dazu da, Zugriffe auf Speicherzellen zu beschleunigen, indem von der CPU benutzte Daten in schnellen Speichern zwischengespeichert werden. Ohne den Cache wartet die CPU oft, während der wesentlich langsamere Hauptspeicher damit beschäftigt ist, ein Datum zu liefern. Steht das Datum schon im Cache, so ist der Zugriff wesentlich schneller, denn die CPU kann aus dem schnellen Zwischenspeicher bedient werden.

Die im Projekt benutzte Cache-Analyse (*siehe* [Fer97b]) benutzt PAG, um Programme mittels abstrakter Interpretation auf ihr Cache-Verhalten hin zu analysieren. Die Informationen, die über das Programm herausgefunden werden sollen, sind für jeden Basisblock in jedem Ausführkontext, ob die Daten, die in diesem Basisblock benutzt werden, im Cache-Speicher der Maschine sind oder nicht. Implementiert ist im Moment nur der Befehls-Cache, d. h. es werden nur Aussagen darüber gemacht, ob sich Maschinenbefehle im Cache befinden oder nicht. Die Analyse liefert folgende Kategorisierung:

$$\begin{aligned} cat \;:\; I \times T &\rightarrow \{ah, am, pers, nc\} \\ (i, \vartheta) &\mapsto \begin{cases} ah & \text{falls } i \text{ im Kontext } \vartheta \text{ sicher im Cache ist} \\ am & \text{falls } i \text{ im Kontext } \vartheta \text{ sicher nicht im Cache ist} \\ pers & \text{falls } i \text{ im Kontext } \vartheta \text{ schlimmstenfalls bei der} \\ & \text{ersten Ausführung nicht im Cache ist} \\ nc & \text{sonst} \end{cases} \end{aligned} \tag{2.29}$$

2.3.4 Einbindung in die Mikroarchitekturanalyse

Zum Zeitpunkt der Implementierung des Projektes war noch keine vollständige Mikroarchitekturanalyse vorhanden. Daher diente die Cache-Analyse als Grundlage, und die Ausführzeiten für die Maschinenbefehle wurden für eine idealisierte Hardware festgelegt. Es wurde angenommen, daß die Maschine für Befehle, die im Cache liegen, einen Takt benötige, für die, die nicht im Cache sind, zehn Takte.

Die im folgenden beschriebene Analyse ist nicht auf diese festen Werte festgelegt. Hätte man eine vollständige Mikroarchitektursimulation, die Maximalwerte für die Ausführzeiten der einzelnen Maschinenbefehle ausgibt, so könnte man diese direkt mit der Analyse verwenden.

Drittes Kapitel
Erzeugung des linearen Problems

In diesem Kapitel wird gezeigt, wie ein lineares Problem erzeugt wird, das eine Suche nach dem längsten Programmpfad ermöglicht. Die Erzeugung wird in mehrere Schritte unterteilt und schrittweise verbessert; teils, um die Problemgröße zu verringern, teils, um engere Maximallaufzeitgrenzen zu berechnen. Wie aus dem zweiten Kapitel bekannt, gliedert sich ein lineares Problem in zwei Teile, die Zielfunktion und die Menge der Nebenbedingungen.

3.1 Ausführzeiten

Die Mikroarchitekturanalyse ordnet jedem Maschinenbefehl in jedem seiner Kontexte eine maximale Ausführzeit zu. Wurde von der Cache-Analyse herausgefunden, daß der Befehl im Cache liegt (Kategorisierung ah), so wird momentan als Ausführzeit auf einer idealisierten Hardware 1 Takt angenommen, ansonsten 10 Takte (Kategorisierungen am, nc und pers). Die Klassifizierung als pers wird später in diesem Kapitel genauer behandelt werden. Also zunächst $\forall i \in I,\ \vartheta \in T$:

$$t'[i,\vartheta] = \begin{cases} 1 & \text{falls } i \text{ im Kontext } \vartheta \text{ als } \mathsf{ah} \text{ klassifiziert wurde} \\ 10 & \text{sonst} \end{cases} \tag{3.1}$$

Für einen Basisblock berechnet sich die maximale Ausführzeit durch Summierung der Ausführzeiten seiner Befehle. Also sei für alle Basisblöcke $b \in B$ und alle Kontexte $\vartheta \in T$:

$$t[b, \vartheta] := \sum_{i \in b} t'[i, \vartheta] \tag{3.2}$$

3.2 Der längste Pfad

Auf der Suche nach der längsten Laufzeit eines Programmes sucht man nach dem Kontrollflußpfad durch das Programm, der am meisten Zeit konsumiert. Als kleinste Einheit werden Basisblöcke betrachtet, d. h. sie sind die Schritte des Pfades. Die Basisblöcke, die durchlaufen werden, benötigen unterschiedlich lange zur Ausführung, also ist der längste Pfad nicht einfach der, der die meisten Schritte enthält, also durch möglichst viele Basisblöcke verläuft, sondern jeder Schritt muß durch seine Ausführzeit gewichtet werden.

Sei $((b_1, \vartheta_1), \ldots, (b_n, \vartheta_n))$ der längste Pfad eines Programmes, wobei die b_i Basisblöcke sind und ϑ_i die Kontexte, in denen die b_i ausgeführt werden. Seien weiter $t[b_1, \vartheta_1], \ldots, t[b_n, \vartheta_n]$ die Ausführzeiten der Basisblöcke auf dem Pfad. Dann dauert die Ausführung dieses Pfades

$$\sum_{i=1}^{n} t[b_i, \vartheta_i]. \tag{3.3}$$

Die Analyse wird den längsten Pfad implizit suchen, d. h. nicht, wie gerade geschehen, eine Folge von durchlaufenen Basisblöcken und Kontexten ausgeben. Statt dessen wird sie die Ausführhäufigkeiten der einzelnen Basisblöcke zählen. Dazu wird obige Darstellung der längsten Laufzeit umformuliert.

Für jeden Basisblock $b \in B$ und $\vartheta \in T$ sei $c[b, \vartheta]$ definiert als

$$c[b, \vartheta] := \left| \left\{ i \in \{1, \ldots, n\} \mid (b_i, \vartheta_i) = (b, \vartheta) \right\} \right|, \tag{3.4}$$

d. h. $c[b, \vartheta]$ ist die Anzahl der Vorkommen von (b, ϑ) im längsten Programmpfad. Nun läßt sich die Ausführzeit wie folgt darstellen:

$$\sum_{b \in B, \vartheta \in T} t[b, \vartheta] \cdot c[b, \vartheta] \tag{3.5}$$

Ziel wird es im folgenden also sein, die $c[b, \vartheta]$ für den Maximalwert dieser Summe zu bestimmen.

if b_1 **then**
    b_2
else
    b_3
fi
b_4

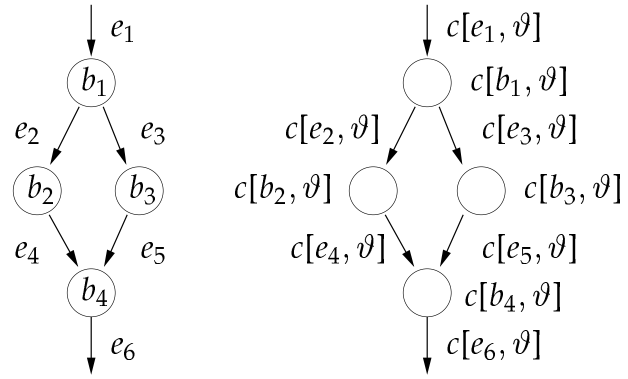


Abbildung 3.1: Ausschnitt aus einem Kontrollflußgraphen beschriftet mit den Durchlaufanzahlen für die Kanten $e_1 \ldots e_6$ und Knoten $b_1 \ldots b_4$ im Kontext ϑ.

3.3 Ausführanzahl

Die $c[b, \vartheta]$ aus dem letzten Abschnitt heißen *Ausführanzahlen* eines Basisblocks. Im folgenden werden diese nach der VIVU-Analyse unterschieden. Die Ausführanzahlen sind $\in \mathbb{N}_0$ (*siehe* Abbildung 3.1):

$$\forall b \in B, \vartheta \in T \text{ sei } c[b, \vartheta] \in \mathbb{N}_0 \tag{3.6}$$

Um die Beziehungen zwischen den Knoten im Kontrollflußgraphen besser aufstellen zu können, werden Variablen definiert, die jedem Paar aus Kante und Kontext ebenfalls eine Zahl aus $\mathbb{N}_0$ zuordnen, die die Durchlaufanzahl dieser Kante angibt:

$$\forall e \in \mathfrak{E}, \vartheta \in T \text{ sei } c[e, \vartheta] \in \mathbb{N}_0 \tag{3.7}$$

3.4 Zielfunktion

Die Zielfunktion wird die Anzahl der Prozessorzyklen angeben, die das untersuchte Programm zur Abarbeitung benötigt. Diese Funktion ist aus Abschnitt 3.2 bereits bekannt und heißt im folgenden Z. Sie wird maximiert werden.

$$\begin{array}{rcll} Z & : & \mathbb{N}_0^{|B \times T|} & \to \mathbb{R}_{\geq 0} \\ & & (c[b, \vartheta])_{b \in B, \vartheta \in T} & \mapsto \displaystyle\sum_{b \in B, \vartheta \in \tau(b)} t[b, \vartheta] \cdot c[b, \vartheta] \end{array} \tag{3.8}$$

$t[b, \vartheta]$ sind dabei konstante Werte, die die Mikroarchitekturanalyse geliefert hat, und $c[b, \vartheta]$ sind die Variablen, die so belegt werden sollten, daß die Zielfunktion maximal wird, also so, daß die maximale Programmlaufzeit berechnet wird.

3.5 Nebenbedingungen

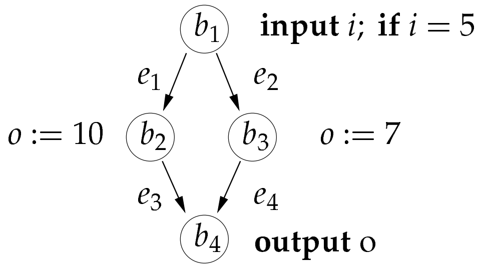
Die Nebenbedingungen im linearen Problem dienen dazu, die Zielfunktion zu so beschränken, daß es möglich ist, ein Maximum zu berechnen. Die grundlegende Idee ist nun, die Ausführhäufigkeiten $c[b, \vartheta]$ der Basisblöcke miteinander zu vergleichen, indem man den Kontrollfluß analysiert. Zusätzlich wird man Bedingungen haben, die einzelne Variablen $c[b, \vartheta]$ beschränken. Die Durchlaufanzahlen $c[e, \vartheta]$ der Kanten werden zusätzlich als Variablen in den Nebenbedingungen benutzt werden.

3.5.1 Beispiel

Die Idee wird klar, wenn man sich das folgende einfache Programm ansieht:

program *Beispielprogramm*;
var i, o : *integer*;
begin
    input i;
    if $i = 5$ **then**
        $o := 10$;
    else
        $o := 7$;
    fi
    output o;
end

Das Hauptprogramm wird wie in der Programmiersprache C als spezielle Prozedur angesehen. In diesem Beispiel gibt es nur diese eine Prozedur, deren Kontrollflußgraph wie folgt aussieht:



Man kann nun Aussagen über die Ausführanzahlen der Basisblöcke unter Verwendung der Durchlaufanzahlen der Kanten machen. Grundlegend ist die Beobachtung, daß ein Basisblock so oft ausgeführt wird, wie die eingehenden Kanten durchlaufen werden. Analog wird er so oft ausgeführt, wie die ausgehenden Kanten durchlaufen werden.

In diesem einfachen Programm werden alle Basisblöcke nur im leeren Kontext ε analysiert. D. h. man erhält aus obigem Kontrollflußgraphen folgende Nebenbedingungen:

	Eingehende Kanten	Ausgehende Kanten
b_1	keine eingehende Kante	$c[b_1, \varepsilon] = c[e_1, \varepsilon] + c[e_2, \varepsilon]$
b_2	$c[b_2, \varepsilon] = c[e_1, \varepsilon]$	$c[b_2, \varepsilon] = c[e_3, \varepsilon]$
b_3	$c[b_3, \varepsilon] = c[e_2, \varepsilon]$	$c[b_3, \varepsilon] = c[e_4, \varepsilon]$
b_4	$c[b_4, \varepsilon] = c[e_3, \varepsilon] + c[e_4, \varepsilon]$	keine ausgehende Kante

Da es sich bei der obigen Prozedur um das Hauptprogramm handelt, weiß man, daß die Prozedur genau einmal beim Programmlauf ausgeführt wird. Daher wird noch folgende Nebenbedingung erzeugt:

$$c[b_1, \varepsilon] = 1 \tag{3.9}$$

Hier sieht man, daß es wichtig ist, daß der erste Basisblock nicht zu einer Schleife gehört, da mit dieser Bedingung sonst die Anzahl der Schleifendurchläufe beschränkt würde.

3.5.2 Grundlegende Nebenbedingungen

Im folgenden werden systematisch die Nebenbedingungen für alle vorkommenden Basisblöcke erzeugt. Dabei geht es meist um die Verknüpfung von Basisblockausführanzahl und Kantendurchlaufanzahl. Grob werden die Kanten in zwei Sorten eingeteilt: die erste Sorte verläuft zwischen Basisblöcken mit gleichem Kontext, die zweite zwischen Basisblöcken mit verschiedenen Kontexten.

3.5.2.1 Grundlegende ein- und ausgehende Kanten

Für jeden Basisblock b, dessen Nachbarknoten (Vorgänger oder Nachfolger) die gleichen Kontexte haben, können die grundlegenden Nebenbedingungen aus Abschnitt 3.5.1 aufgestellt werden. Dazu müssen *alle* Vorgängerknoten bzw. alle Nachfolgerknoten den gleichen Kontext wie b haben.

Zunächst also die Bedingung für den Kontrollfluß, der aus b herausfließt. Es sei für alle $b \in B$ mit $out(b) \neq \emptyset$ und $\forall [b, b'\rangle \in out(b) : \ \tau(b) = \tau(b')$ für alle $\vartheta \in \tau(b)$ folgende Nebenbedingung definiert:

Nebenbedingung (Grundlegende ausgehende Kanten)

$$c[b, \vartheta] = \sum_{e \in out(b)} c[e, \vartheta] \tag{3.10}$$

Analog werden die Bedingungen für die eingehenden Kanten aufgebaut. Hier entsteht aufgrund der VIVU-Analyse ein zusätzlicher Sonderfall bei Rückkanten in Schleifen. Diese Kanten haben als Zielknoten den Kopf der Schleife und die gleichen Kontexte wie dieser. Doch fließt der Kontrollfluß von der ersten Iteration der Schleife über diese

Kanten nicht zurück in die erste Iteration, sondern in die zweite. Damit ändert sich der Kontext. Um diesem Sonderfall Rechnung zu tragen, werden die Rückkanten später bei der Schleifenabhandlung betrachtet werden.

Die Frage liegt nahe, ob bei Schleifen nicht auch ein Sonderfall bei den ausgehenden Kanten auftritt. Doch die Rückkanten werden hier korrekt behandelt (die Rückkanten der ersten wie aller weiteren Iterationen werden tatsächlich so oft ausgeführt, wie der Quellknoten im gleichen Kontext). Die Ein- und Austrittskanten werden wegen des Kontextwechsels nicht in Betracht gezogen, also gibt es auch hier keine Probleme – sie werden später gesondert untersucht.

Bei Prozeduraufrufen entstehen hier keine Sonderfälle, da die Kanten hierfür nicht zu Kontrollflußgraphen gehören, also $\notin \mathcal{E}$.

Für alle $b \in B$ mit $in(b) \neq \emptyset$ und $in(b) \cap back(\mathcal{L}) = \emptyset$ und $\forall [b, b'\rangle \in in(b): \ \tau(b) = \tau(b')$ sei für alle $\vartheta \in \tau(b)$ folgende Nebenbedingung definiert:

Nebenbedingung (Grundlegende eingehende Kanten)

$$c[b, \vartheta] = \sum_{e \in in(b)} c[e, \vartheta] \tag{3.11}$$

Damit sind die grundlegenden Nebenbedingungen nun definiert.

3.5.2.2 Hauptprogramm

Eine weitere, sehr einfache Nebenbedingung fehlt noch, und zwar die, die festlegt, daß das Hauptprogramm einmal ausgeführt wird. Diese wird dadurch formuliert, daß der erste Basisblock des Hauptprogrammes genau einmal ausgeführt wird. Dabei ist wichtig, daß der erste Basisblock nicht zu einer Schleife gehört. Das ist durch die Existenz eines leeren Basisblocks am Anfang jeder Prozedur sichergestellt. Zudem wurde gefordert, daß das Hauptprogramm keinen Rekursionen unterworfen ist. Der erste Basisblock b_{start} kommt nur im leeren Kontext ε vor.

Also wird folgende Nebenbedingung hinzugefügt:

Nebenbedingung (Hauptprogramm)

$$c[b_{start}, \varepsilon] = 1 \tag{3.12}$$

3.5.3 Nebenbedingungen für Kanten verschiedener Kontexte

Basisblöcke, die entweder Vorgängerknoten oder Nachfolgerknoten haben, die einen anderen Kontext haben als sie selbst, oder die gar keine Vorgänger oder Nachfolger haben, kommen an folgenden Stellen vor:

(i) Der erste Block einer Prozedur.

(ii) Rücksprungblöcke einer Prozedur.

(iii) Der Kopf einer Schleife. Hier gibt es Sonderfälle mit Kanten, die die Schleife betreten und mit Rückkanten. Zudem muß das einmalige Ausrollen der VIVU-Analyse beachtet werden.

(iv) Blöcke, deren ausgehende Kanten eine Schleife verlassen.

Keine Sonderfälle treten an Prozeduraufrufsblöcken auf, obwohl man dies zunächst vermuten könnte. Das liegt daran, daß im Kontrollflußgraphen des Aufrufers eine Fall-Through-Kante existiert, die den Kontrollfluß so aussehen läßt, als würde er über den Prozeduraufruf hinwegfließen. Da keine bedingten Prozeduraufrufe erlaubt sind (*siehe* Abschnitt 2.1.4 auf Seite 20), sind die nötigen Nebenbedingungen schon in Abschnitt 3.5.2.1 auf Seite 37 aufgestellt worden.

Die Behandlung der obigen Fälle wird nun beschrieben.

3.5.3.1 Prozeduraufrufe

Die Anzahl der Ausführungen einer Prozedur entspricht der Anzahl der Aufrufe dieser Prozedur im Programm. Die Aufrufanzahlen werden anhand der Kontexte des Aufrufers unterschieden.

Wird eine Prozedur $\varphi \in \mathcal{F}$, deren erster Basisblock b_{callee} heißen soll, von einem Basisblock $b_{\text{caller}} \in \mathcal{C}$ nichtrekursiv aufgerufen, so ist sicher, daß alle Kontexte $\tau(b_{\text{caller}})$ erweitert um $\mathsf{C}[b_{\text{caller}}]$ in $\tau(b_{\text{callee}})$ vorkommen, also, daß $\tau(b_{\text{caller}}) \circ \mathsf{C}[b_{\text{caller}}] \subseteq \tau(b_{\text{callee}})$. Ferner ist es möglich, daß φ rekursiv aufgerufen wird. In dem Fall gilt zusätzlich: $\tau(b_{\text{caller}}) \circ \mathsf{R}[b_{\text{caller}}] \subseteq \tau(b_{\text{callee}})$.

Es ist klar, daß b_{callee} in den Kontexten $\tau(b_{\text{caller}}) \circ \mathsf{C}[b_{\text{caller}}]$ so oft ausgeführt wird wie $[b_{\text{caller}}, b_{\text{callee}}\rangle$ in den entsprechenden Kontexten aus $\tau(b_{\text{caller}})$ durchlaufen. Ebenso wird b_{caller} genausooft ausgeführt wie $[b_{\text{caller}}, b_{\text{callee}}\rangle$, denn es gibt keine Aufrufkanten von bedingten Sprüngen. Somit können die Ausführanzahlen der beiden Blöcke gleichgesetzt werden.

Also wird für alle $b_{\text{caller}} \in \mathcal{C}$ mit $b_{\text{callee}} := f_{call}(b_{\text{caller}})$ für alle $\vartheta \in \tau(b_{\text{caller}})$ mit $\tau(b_{\text{caller}}) \circ \mathsf{C}[b_{\text{caller}}] \subseteq \tau(b_{\text{callee}})$ die folgende Nebenbedingung definiert:

Nebenbedingung (Erster Prozeduraufruf)

$$c[b_{\text{callee}}, \vartheta \circ \mathsf{C}[b_{\text{caller}}]] = c[b_{\text{caller}}, \vartheta] \tag{3.13}$$

Rekursive Aufrufe sind ein wenig unübersichtlich, da sie aus beliebig tiefen Inkarnationen erfolgen können und nach der VIVU-Analyse einmal ausgerollt werden. Daher

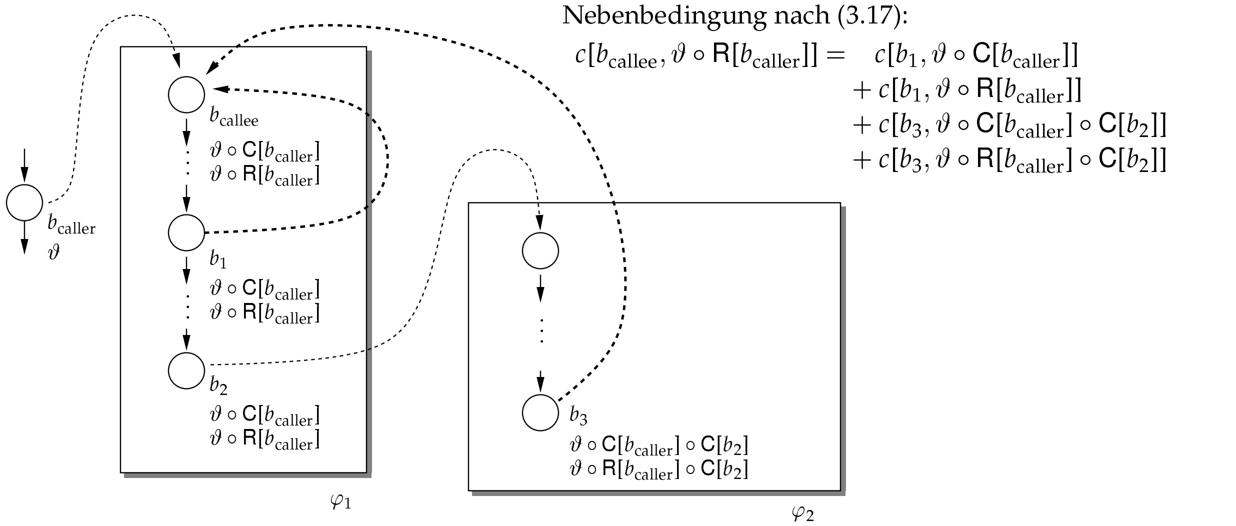


Abbildung 3.2: Ausführanzahl einer Funktion bei rekursivem Aufruf. Im Bild sind zwei rekursive Aufrufe von b_{callee} zu sehen: einer von b_1 aus, ein anderer über b_2 von b_3 aus. Die relevanten Kontexte der Basisblöcke sind mit angegeben.

dient Abbildung 3.2 dazu, die folgenden Gedankengänge leichter nachvollziehbar zu machen.

Im folgenden wird eine Funktion P benötigt, die alle Kontexte ausgibt, die ein bestimmtes Präfix haben:

$$\begin{array}{rcrcl} P & : & T & \to & \mathfrak{P}(T) \\ & & \vartheta & \mapsto & \vartheta \circ T \end{array} \tag{3.14}$$

Diese Funktion soll auch für Mengen von Kontexten definiert sein:

$$\begin{array}{rcrcl} P & : & \mathfrak{P}(T) & \to & \mathfrak{P}(T) \\ & & \Theta & \mapsto & \bigcup\limits_{\vartheta \in \Theta} P(\vartheta) \end{array} \tag{3.15}$$

Für den rekursiven Aufruf der Prozedur gilt nun, daß der erste Basisblock b_{callee} einer Prozedur im rekursiven Kontext so oft ausgeführt wird, wie Rekursionsaufrufskanten aus dem ersten Aufruf oder aus rekursiven Aufrufen ihn betreten. Diese rekursiven Aufrufe zeichnen sich dadurch aus, daß sie Kontexte mit den Präfixen $\vartheta \circ \mathsf{C}[b_{\text{caller}}]$ und $\vartheta \circ \mathsf{R}[b_{\text{caller}}]$ haben.

Definiert sei also eine Funktion P_f, die die Menge dieser Präfixe für alle Kontexte ausgibt:

$$\begin{array}{rcrcl} P_f & : & \mathcal{C} \times T & \to & \mathfrak{P}(T) \\ & & (b, \vartheta) & \mapsto & P(\vartheta \circ \{\mathsf{C}[b], \mathsf{R}[b]\}) \end{array} \tag{3.16}$$

Damit gilt für die Ausführanzahl von b_{callee} in rekursiven Kontexten für alle $\vartheta \in \tau(b_{\text{caller}})$:

Nebenbedingung (Rekursiver Prozeduraufruf)

$$c[b_{\text{callee}}, \vartheta \circ \mathsf{R}[b_{\text{caller}}]] = \sum_{\substack{b \in C,\ f_{call}(b) = b_{\text{callee}}, \\ \vartheta' \in \tau(b) \cap P_f(b_{\text{caller}}, \vartheta)}} c[b, \vartheta'] \tag{3.17}$$

Für rekursive Aufrufe wird weiterhin die Rekursionstiefe eines Prozeduraufrufs benötigt, da sonst das lineare Problem unbeschränkt ist. Die Rekursionstiefe sei für alle $\vartheta \in \tau(b_{\text{caller}})$ definiert als das Verhältnis der Ausführanzahlen des Aufgerufenen zu denen des Aufrufers, also als

$$k_{\text{rec}}[b_{\text{caller}}, \vartheta] := \begin{cases} \frac{c[b_{\text{callee}}, \vartheta \circ \mathsf{C}[b_{\text{caller}}]] + c[b_{\text{callee}}, \vartheta \circ \mathsf{R}[b_{\text{caller}}]]}{c[b_{\text{caller}}, \vartheta]} & \text{falls } c[b_{\text{caller}}, \vartheta] \neq 0 \\ 0 & \text{sonst.} \end{cases} \tag{3.18}$$

Da die maximale Rekursionstiefe nicht automatisch inferiert wird, wird in dieser Arbeit vorausgesetzt, daß sie bekannt ist. Im fünften Kapitel wird beschrieben werden, wie der Benutzer sie definiert. Im Moment sei vorausgesetzt, daß obere Schranken $\tilde{k}_{\text{rec}}[b, \vartheta]$ der tatsächlichen Rekursionstiefe $k_{\text{rec}}[b, \vartheta]$ bekannt sind.

Direkt aus (3.18) lassen sich Nebenbedingungen ableiten, die die Anzahl der Rekursionen begrenzen, indem die oberen Schranken der tatsächlichen Rekursionstiefen benutzt werden. Folgende Nebenbedingung ergibt sich für jedes $\vartheta \in \tau(b_{\text{caller}})$:

Nebenbedingung (Rekursionstiefe)

$$c[b_{\text{callee}}, \vartheta \circ \mathsf{C}[b_{\text{caller}}]] + c[b_{\text{callee}}, \vartheta \circ \mathsf{R}[b_{\text{caller}}]] \leq \tilde{k}_{\text{rec}}[b_{\text{caller}}, \vartheta] \cdot c[b_{\text{caller}}, \vartheta] \tag{3.19}$$

Es müssen keine zusätzlichen Nebenbedingungen für das Verlassen von Prozeduren erzeugt werden, da diese implizit schon über die Beziehungen zwischen den Basisblöcken innerhalb der Prozedur und dem Basisblock, der die Prozedur aufruft, bestehen. Am Aufrufspunkt sorgen bereits die Nebenbedingungen in (3.10) für die Beziehung zum nachfolgenden Block, denn die Kontrollflußgraphen enthalten Fall-Through-Kanten, die angeben, wie der Kontrollfluß nach dem Aufruf weiterläuft (*siehe* Abbildung 3.3).

3.5.3.2 Schleifen

Die Nebenbedingungen für Schleifen sind ähnlich denen der Prozeduraufrufe. Die Unterschiede liegen darin, daß die Kanten, die Prozeduren betreten und verlassen, im

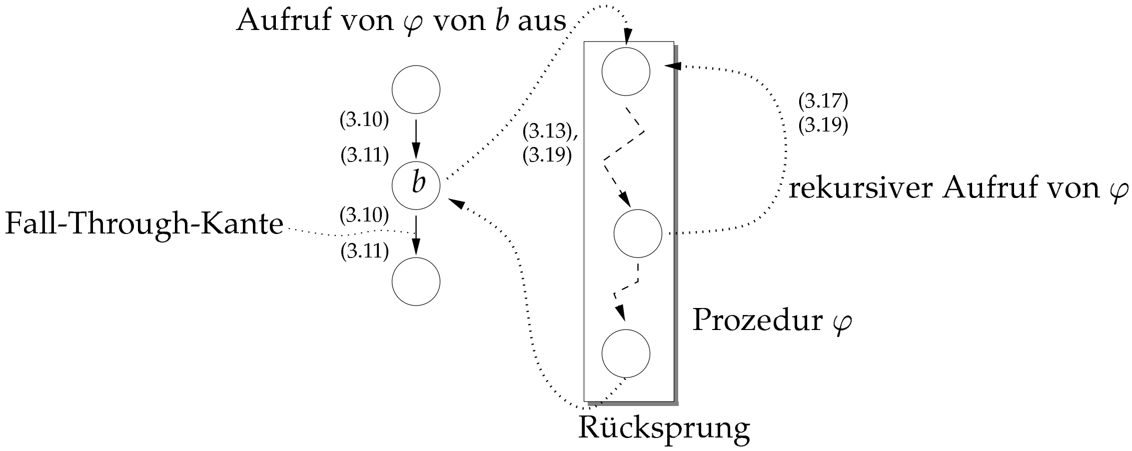


Abbildung 3.3: An den Enden der Kanten ist angegeben, welche Nebenbedingungen die Kanten und Knoten in Verbindung setzen. Der Prozedurrücksprung ist implizit in den anderen Nebenbedingungen enthalten, daher müssen keine Bedingungen erzeugt werden.

Gegensatz zu den entsprechenden Kanten bei Schleifen nicht in den Kontrollflußgraphen vorhanden sind. Des weiteren werden Kontexte bei Schleifen nach dem Kopf unterschieden, bei Prozeduren aber nach dem Aufrufer. Dies ist eigentlich nur eine Vereinfachung – Prozeduren dürfen von verschiedenen Stellen betreten werden, Schleifen hingegen nicht. Die im folgenden aufgestellten Nebenbedingungen weisen eine große Ähnlichkeit mit denen in (3.13), (3.17) und (3.19) auf.

In Abbildung (3.4) sieht man eine einfache Schleife und wie man sie sich nach einmaligem virtuellen Ausrollen vorstellen muß.

Die erste Nebenbedingung wird am Anfang der Schleife l bei der ersten Iteration aufgestellt. Der Kopf der Schleife wird in der ersten Iteration genausooft ausgeführt, wie die Kante vom Pre-Header zum Kopf (die einzige eingehende Kante von l) durchlaufen. Ist ϑ ein Kontext des Pre-Headers, so ist $\vartheta \circ \mathsf{F}[head(l)]$ der entsprechende Kontext des Kopfes der Schleife. Sei $b_h = head(l)$. Dann werden folgende Nebenbedingungen $\forall \vartheta \in \tau(pre(l))$ erzeugt:

Nebenbedingung (Schleifenkopf in erster Iteration)

$$c[b_h, \vartheta \circ \mathsf{F}[b_h]] = c[entry(l), \vartheta] \tag{3.20}$$

Als Nächstes werden für die Rückkanten Nebenbedingungen erzeugt. Die zweite Iteration wird von den Rückkanten der ersten Iteration betreten und alle weiteren Iterationen von den Rückkanten ab der zweiten Iteration. D. h. der Kopf der Schleife wird so oft ausgeführt wie die Summe der Durchläufe der Rückkanten in der ersten und allen weiteren Iterationen. Mit obigen Bezeichnungen erhält man für alle Kontexte $\vartheta \in \tau(entry(e))$:

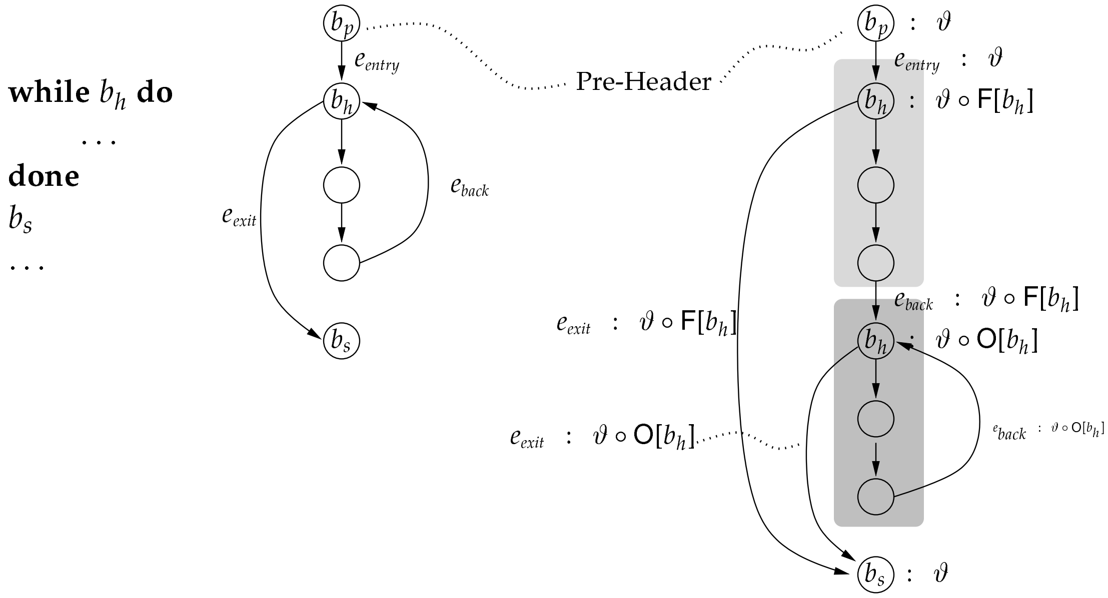


Abbildung 3.4: Eine Schleife und wie sie nach einmaligem Ausrollen aussieht. Die Kontexte sind relativ zum Kontext ϑ außerhalb der Schleife angegeben.

Nebenbedingung (Schleifenkopf in weiteren Iterationen)

$$c[b_h, \vartheta \circ \mathsf{O}[b_h]] = \sum_{e \in back(l)} \left(c[e, \vartheta \circ \mathsf{F}[b_h]] + c[e, \vartheta \circ \mathsf{O}[b_h]] \right) \tag{3.21}$$

Kanten, die Schleifen verlassen, werden aus zwei Perspektiven betrachtet: von innerhalb und von außerhalb der Schleife. Das erspart die Betrachtung vieler Sonderfälle, da diese Kanten auch mehrere Schleifen gleichzeitig verlassen können. Sei b_s ein Basisblock, der eingehende Kanten $[b', b_s\rangle$ hat, die eine Schleife verlassen. Dann wird eine zusammenfassende Nebenbedingung für alle $\vartheta \in \tau(b_s)$ aufgestellt, die berücksichtigt, daß über eine Kante mehrere Schleifen verlassen werden können. Also $\forall [b', b_s\rangle \in exit(\mathcal{L})$:

Nebenbedingung (Zusammenfassende Kantensumme)

$$c[[b', b_s\rangle, \vartheta] = \sum_{\vartheta' \in \tau(b') \cap P(\vartheta)} c[[b', b_s\rangle, \vartheta'] \tag{3.22}$$

Unter Berücksichtigung dieser Nebenbedingung kann man nun die normalen Nebenbedingungen für ein- bzw. ausgehende Kanten aufstellen. Sei dazu $[b', b_s\rangle \in exit(\mathcal{L})$. Dann definiert man für alle $\vartheta \in \tau(b')$ die einfache Bedingung für ausgehende Kanten (3.10) mit $b = b'$. Ebenso wird für alle $\vartheta \in \tau(b_s)$ die Bedingung (3.11) mit $b = b_s$ hinzugefügt. Bedingung (3.22) verbindet dann die Sichtweisen auf die Kante $[b', b_s\rangle$.

Ähnlich wie bei rekursiven Prozeduraufrufen müssen die Schleifendurchläufe begrenzt werden. Dazu wurde die Funktion *inner*(l) definiert. Die Ausführanzahl dieses inneren

Basisblocks ist gleich der Anzahl der Iterationen (relativ zur Anzahl der Eintritte in die Schleife). Also ist die Anzahl der Iterationen einer Schleife l für alle $\vartheta \in \tau(pre(l))$ wie folgt definiert. Sei $b_i = inner(l)$.

$$k_{\text{rep}}[b_h, \vartheta] := \begin{cases} \frac{c[b_i, \vartheta \circ \mathsf{F}[b_h]] + c[b_i, \vartheta \circ \mathsf{O}[b_h]]}{c[entry(l), \vartheta]} & \text{falls } c[entry(l), \vartheta] \neq 0 \\ 0 & \text{sonst} \end{cases} \tag{3.23}$$

Wie schon bei der Rekursionstiefe wird das Problem der Bestimmung der Schranke der Ausführanzahlen der Schleifen auf das fünfte Kapitel verschoben. Es wird vorausgesetzt, daß obere Schranken $\tilde{k}_{\text{rep}}[b, \vartheta]$ für alle tatsächlichen Ausführanzahlen $k_{\text{rep}}[b, \vartheta]$ bekannt sind. Mit diesen oberen Schranken kann also wie folgt abgeschätzt werden:

Nebenbedingung (Ausführanzahl einer Schleife)

$$c[b_i, \vartheta \circ \mathsf{F}[b_h]] + c[b_i, \vartheta \circ \mathsf{O}[b_h]] \ \leq\ \tilde{k}_{\text{rep}}[b_h, \vartheta] \ \cdot\ c[entry(l), \vartheta] \tag{3.24}$$

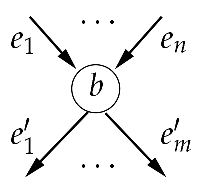
Damit ist die erste Version des linearen Problems vollständig.

3.6 Korrektheit und Lösbarkeit

Der LP-Löser berechnet eine optimale Lösung des ILPs. Daher ist der zugehörige Wert der Zielfunktion garantiert maximal. Zu zeigen bleibt also nur, daß der Maximalwert korrekt ist, also die Maximallaufzeit des Programmes korrekt abschätzt und nicht unter der tatsächlichen WCET liegt.

Die Zielfunktion beschreibt die Laufzeit des Programmes, daher ist die Analyse korrekt, falls die aufgestellten Nebenbedingungen korrekt sind, denn der LP-Löser berechnet dann den Maximalwert der Zielfunktion unter Beachtung korrekter Nebenbedingungen.

Grundlegend gibt es nur drei Sorten von Nebenbedingungen. Die erste Sorte beschreibt durch Gleichungen, wie der eingehende und ausgehende Kontrollfluß mit der Ausführanzahl eines beliebigen Basisblocks zusammenhängt. Die Gleichungen dafür sind denkbar einfach und beruhen immer auf dem Prinzip, daß ein Block so oft ausgeführt wird, wie Kontrollfluß ihn betritt und verläßt. Trotz der Unterscheidung der Kontexte ändert sich nichts an der Korrektheit dieser lokalen Beobachtungen. Nebenbedingungen dieser Art sind (3.10), (3.11), (3.12), (3.13), (3.17), (3.20) und (3.21). Das Prinzip sieht so aus:



$$\sum_{i=1}^{n} c[e_i] = c[b] = \sum_{i=1}^{m} c[e'_i]$$

Die zweite Sorte Nebenbedingungen faßt Ausführanzahlen verschiedener Kontexte zusammen. Auch dies ist eine korrekte Beschreibung. Nur die Nebenbedingung (3.22) fällt in diese Gruppe. Das Prinzip lautet wie folgt: wird ein Basisblock b in den Kontexten $\vartheta_1, \ldots, \vartheta_n$ unterschieden, so ist die Ausführhäufigkeit in einem Kontext ϑ, der Präfix aller ϑ_i ist, gleich der Summe der Ausführhäufigkeiten in den Kontexten ϑ_i. Diese Prinzip ist ebenso für Kanten gültig. Die Nebenbedingung (3.22) beschreibt dies für den Fall, daß eine Kanten sowohl einerhalb einer Schleife, als auch außerhalb betrachtet wird.

Die dritte Gruppe beschränkt Zyklen im Kontrollflußgraphen und im Call-Graphen. Die Nebenbedingungen (3.19) und (3.24) sind nur korrekt, wenn $k_{\text{rep}}[b_h, \vartheta] \leq \tilde{k}_{\text{rep}}[b_h, \vartheta]$ und $k_{\text{rec}}[b_{\text{caller}}, \vartheta] \leq \tilde{k}_{\text{rec}}[b_{\text{caller}}, \vartheta]$. Dies muß vorausgesetzt werden.

Wenn alle Nebenbedingungen korrekt aufgestellt sind und der LP-Löser die Zielfunktion korrekt maximiert, so ist die Analyse korrekt, denn man erhält dann bei Einhaltung der durch die Nebenbedingungen beschriebenen Eigenschaften des Programmes die maximal mögliche Ausführzeit. Fehlende Beschreibungen des Kontrollflusses können dann höchstens zu einer Überschätzung der Laufzeit führen, was zwar unerwünscht ist, aber zulässig.

3.6.1 Lösbarkeit

Nicht jedes beliebige Eingabeprogramm führt zu einem beschränkten Problem. Das untersuchte Programm $\mathcal{P}$ muß folgende Eigenschaften haben:

(i) $\mathcal{P}$ terminiert für jede Eingabe.

(ii) Alle Knoten des Call-Graphen sind von φ_{start} aus erreichbar. Formal: $\forall \hat{G} = (\mathcal{F}, \hat{E}),\ \forall \varphi \in \mathcal{F} :\ \varphi_{start} \vdash^*_{\hat{G}} \varphi$.

(iii) Jeder Knoten eines jeden Kontrollflußgraphen ist vom Anfangsblock des Kontrollflußgraphen erreichbar. Formal: $\forall G = (V, E, b_0),\ \forall b \in V :\ b_0 \vdash^*_G b$.

(iv) Die gefundenen Rekursionen sind die einzigen Zyklen im Call-Graphen.

(v) Die gefundenen Schleifen sind die einzigen Zyklen in den Kontrollflußgraphen.

Um zu zeigen, daß es eine optimale Lösung für das lineare Problem gibt, müssen zwei Behauptungen gezeigt werden:

(i) Das lineare Problem ist beschränkt.

(ii) Es gibt eine zulässige Lösung.

Zunächst zu (ii): Da das Programm für jede Eingabe terminiert, gibt es auch für jede Eingabe einen Programmpfad. Jeder dieser Programmpfade definiert für das lineare Problem eine Belegung der Variablen der Ausführanzahlen der Basisblöcke. Aus der Korrektheit der Nebenbedingungen folgt, daß diese Belegung zulässig sein muß, denn die Nebenbedingungen beschreiben das Programm korrekt.

Die Beschränktheit zu beweisen ist schwieriger. Es genügt nicht zu argumentieren, daß Programm terminiere, also sei das Problem beschränkt, denn es wurden natürlich nicht *alle* Eigenschaften des Programmes in den Nebenbedingungen kodiert. D. h. es wäre möglich, daß das Programm aufgrund von nicht beschriebenen Eigenschaften terminiert, während das ILP trotzdem nicht beschränkt ist. Daß die Nebenbedingungen so beschaffen sind, daß das ILP unter obigen Voraussetzungen beschränkt ist, wird jetzt gezeigt.

Um die Beschränktheit der Zielfunktion zu zeigen, genügt es, obere Schranken für jede darin vorkommende Variable anzugeben, denn die Koeffizienten sind alle positiv.

Definition 3.12 (Zyklusfreier Pfad)
Sei $G = (V, E)$ ein gerichteter Graph und $v = v_1 \vdash \cdots \vdash v_n$ ein Pfad in G. Dann heißt v *zyklusfrei*, falls $|\{v_1, \ldots, v_n\}| = n$, und n heißt die *Länge* des Pfades v.

Behauptung 3.1
Ist die Ausführanzahl des Anfangsblocks b_0 eines Kontrollflußgraphen beschränkt, so sind die Ausführanzahlen aller Blöcke desselben Kontrollflußgraphen beschränkt.

Beweis
Induktion über die Länge n des längsten zyklusfreien Pfades vom Anfangsblock des Kontrollflußgraphen zu einem Knoten b in demselben Graphen.

$n = 1$: Der Pfad besteht nur aus dem Anfangsblock des Kontrollflußgraphen. Dessen Ausführhäufigkeit ist nach Voraussetzung beschränkt.

$n \to n + 1$: Sei b ein Knoten im Kontrollflußgraphen, dessen längster zyklusfreier Pfad von b_0 aus Länge $n + 1$ hat. Es gibt folgende Fälle:

Erster Fall.

(*) b ist nicht Kopf einer Schleife, d. h. $b \notin head(\mathcal{L})$.

(**) Keine Austrittskanten von Schleifen seien unter den eintretenden Kanten von b, d. h. $exit(\mathcal{L}) \cap in(b) = \emptyset$.

Alle eingehenden Kanten kommen wg. (*) also von Knoten b_i, deren Länge des längsten zyklusfreien Pfades $\leqq n$ ist. Die eintretenden Kanten und die Quellknoten dieser Kanten werden dann in den gleichen Kontexten wie b bewertet, denn wg. (*) kann b nicht in einem neuen Schleifenkontext stehen und wg. (**) die eingehenden Kanten nicht im Kontext einer vorangehenden Schleife.

Für alle Ausführanzahlen dieser Knoten gilt also nach Induktionsvoraussetzung:

$$\forall \vartheta \in \tau(b), \forall i: \ \exists M_{i,\vartheta} \therefore c[b_i, \vartheta] \leqq M_{i,\vartheta}.$$

Wg. (3.10) gilt:

$$\sum_{e=[b_i,b'\rangle \in out(b_i)} c[e,\vartheta] = c[b_i,\vartheta] \quad \leqq M_{i,\vartheta}$$

Für jedes b_i sei $e_i = [b_i, b\rangle \in out(b_i)$. Dann folgt (alle Durchlaufanzahlen sind $\geqq 0$):

$$c[e_i, \vartheta] \leqq c[b_i, \vartheta]$$

Damit erhält man:

$$c[e_i, \vartheta] \leqq M_{i,\vartheta}.$$

Sei $M_\vartheta = \sum M_{i,\vartheta}$. Dann folgt: $c[b,\vartheta] \leqq M_\vartheta$, denn nach (3.11) gilt:

$$c[b,\vartheta] = \sum_{e_{[b_i,b\rangle}} c[e,\vartheta]$$

Es ist also eine Schranke M_ϑ für jeden Kontext ϑ gefunden.

Zweiter Fall. b ist nicht Kopf einer Schleife, und es gibt austretende Kanten $e_j = [b_j, b\rangle$ von Schleifen unter den Vorgängern. Der längste zyklusfreie Pfad zu jedem b_j ist dann ebenfalls $\leqq n$. Die b_j haben andere Kontexte, da sie zu vorangehenden Schleifen gehören.

Für alle Kontexte $\vartheta'_j \in \tau(b_j)$ lassen sich nach Induktionsvoraussetzung M_{j,ϑ'_j} als Beschränkungen für alle ihre Ausführanzahlen finden.

Nach (3.22) ist $\forall \vartheta \in \tau(b)$:

$$c[e_j, \vartheta] = \sum_{\vartheta'_j \in \tau(b_j) \cap P(\vartheta)} c[e_j, \vartheta'_j]$$

Für jedes $c[e_j, \vartheta'_j]$ in der Summe ergibt sich $c[e_j, \vartheta'_j] \leqq M_{j,\vartheta'_j}$ (wg. (3.10) wie im ersten Fall). Demnach gilt:

$$c[e_j, \vartheta] \leqq \underbrace{\sum_{\vartheta'_j \in \tau(b_j) \cap P(\vartheta)} M_{j,\vartheta'_j}}_{=:M_{j,\vartheta}}.$$

Für die Kanten der Vorgänger von b, die keine Austrittskanten von Schleifen sind, findet man wie im ersten Fall $M_{i,\vartheta}$ zur Beschränkung der Durchlaufanzahl. Also ist insgesamt $c[b,\vartheta] \leqq M_\vartheta$ mit $M_\vartheta = \sum M_{j,\vartheta} + \sum M_{i,\vartheta}$.

Dritter Fall. b ist Kopf einer Schleife l. Der längste zyklusfreie Pfad zum Pre-Header $b_p := pre(l)$ hat dann Länge n. Für diesen Basisblock gibt es also $M_\vartheta \therefore c[b_p, \vartheta] \leqq M_\vartheta$ für alle $\vartheta \in \tau(b_p)$. (3.24) besagt, daß für $M_{\vartheta,0} = \tilde{k}_{\text{rep}}[b, \vartheta] \cdot M_\vartheta$ gelten muß:

$$c[b, \vartheta \circ \mathsf{F}[b]] + c[b, \vartheta \circ \mathsf{O}[b]] \leqq M_{\vartheta,0}$$

für alle Kontexte ϑ von b_p. Da

$$\tau(b) = \tau(b_p) \circ \{\mathsf{F}[b], \mathsf{O}[b]\},$$

folgt, daß

$$\forall \vartheta' \in \tau(b): \; c[b, \vartheta'] \leqq M_{\vartheta,0}.$$

Damit wurden alle Basisblöcke im Kontrollflußgraphen behandelt, denn alle Basisblöcke sind vom Anfangsknoten aus erreichbar, also gibt es zu jedem einen längsten zyklusfreien Pfad, und es gibt nach Voraussetzung neben den Schleifen keine anderen Zyklen.

$\overline{qed}$

Behauptung 3.2
Die Ausführanzahl des Anfangsblocks jeder Prozedur ist in allen seinen Kontexten beschränkt.

Beweis
Induktion über die Länge n des längsten zyklusfreien Pfades von φ_{start} zu einer Prozedur φ im Call-Graphen.

$n = 1$: Der Pfad besteht nur aus φ_{start}. (3.12) beschränkt die Ausführanzahl des Anfangsblockes von φ_{start}.

$n \to n+1$: Sei b_0 der erste Knoten eines Kontrollflußgraphen einer Prozedur, dessen längster zyklusfreier Pfad von φ_{start} aus Länge $n+1$ hat.

b_0 werde von den Knoten $b_1, \ldots, b_n$ aus nichtrekursiv aufgerufen und von den Knoten $b'_1, \ldots, b'_m$ rekursiv. Für $\Phi(b_i)$ gilt, daß die Länge des längsten zyklusfreien Pfades $\leqq n$ ist. Seien $b_{i,0}$ die Anfangsblöcke der Prozeduren $\Phi(b_i)$. Dann gilt nach Induktionsvoraussetzung, daß

$$\forall \vartheta_{i,j} \in \tau(b_{i,0}): \; \exists M_{i,j} \therefore c[b_{i,0}, \vartheta_{i,j}] \leqq M_{i,j}.$$

Damit gilt nach Behauptung 3.1 auf Seite 46, daß

$$\forall \vartheta_{i,k} \in \tau(b_i): \; \exists M'_{i,k} \therefore c[b_i, \vartheta_{i,k}] \leqq M'_{i,k}.$$

Die $M'_{i,k}$ eignen sich wg. (3.13) als Schranken für die Ausführanzahl von b_0 in den Erstaufrufskontexten. Also:

$$c[b_0, \vartheta_{i,k} \circ \mathsf{C}[b_i]] \leq M'_{i,k}.$$

Für $c[b_0, \vartheta_{i,k} \circ \mathsf{R}[b_i]]$ definiert (3.19) damit ebenso eine Schranke:

$$c[b_0, \vartheta_{i,k} \circ \mathsf{R}[b_i]] \leq \tilde{k}_{\text{rec}}[b_i, \vartheta_{i,k}] \cdot M'_{i,k}.$$

Da $\tau(b_0) = \bigcup \vartheta_{i,k} \circ \{\mathsf{C}[b_i], \mathsf{R}[b_i]\}$, sind damit Schranken für alle Ausführanzahlen von b_0 vorhanden.

Da nach Voraussetzung jede Funktion von φ_{start} aus erreichbar ist, und da die gefundenen Rekursionen die einzigen Zyklen im Call-Graphen sind, gilt die Aussage für alle Prozeduren.

$\overline{qed}$

Aus dem Beweis von (3.2) folgt, daß alle Ausführanzahlen aller Basisblöcke im Programm beschränkt sind in allen Kontexten. Weiterhin sieht man, daß für den Beweis die Zusammenhänge (3.17) und (3.21) nicht benötigt werden. Sie dienen also nur der Verbesserung der Schrankenbildung.

3.7 Verbesserungen

Die bisher vorgestellte Konstruktion des linearen Problems erlauben noch Verbesserungen. Diese gliedern sich in zwei Sorten:

(i) Verbesserung der Genauigkeit

(ii) Verkürzung der Analysedauer

Die Verbesserung der Genauigkeit geschieht z. B. durch Einbeziehung der Klassifizierung pers, die die Cache-Analyse liefert. Die Verkürzung der Analysedauer geschieht durch Entfernen redundanter Gleichungen und unnötiger Variablen.

3.7.1 Die pers-Analyse

Bis jetzt wurde die Klassifizierung pers der Cache-Analyse nicht ausgenutzt, sondern die Ausführzeit trotzdem auf zehn Takte festgesetzt. Doch ein Befehl mit der Klassifizierung pers ist bis auf die erste Ausführung immer im Cache, daher benötigt er ab der zweiten Ausführung nur einen Takt auf der idealisierten Hardware.

Um diese Analyse allgemein zu ermöglichen, müssen die Kostenfunktion und die Nebenbedingungen verändert werden. Die bisherige Ausführzeit eines Befehles $t'[i,\vartheta]$ wird ergänzt durch die *Beschleunigung* ab der zweiten Ausführung $\hat{t}'[i,\vartheta]$. Mit dieser neuen Größe dauert $i \in I$ im Kontext ϑ bei der ersten Ausführung $t'[i,\vartheta]$ und bei allen weiteren Ausführungen $t'[i,\vartheta] - \hat{t}'[i,\vartheta]$.

$\hat{t}'[i,\vartheta]$ sei für die zugrundeliegende idealisierte Maschine wie folgt definiert:

$$\hat{t}'[i,\vartheta] = \begin{cases} 9 & \text{falls } i \text{ im Kontext } \vartheta \text{ als } \mathsf{pers} \text{ klassifiziert wurde} \\ 0 & \text{sonst} \end{cases} \tag{3.25}$$

Damit dauert ein Befehl, der als pers klassifiziert wurde, beim ersten Ausführen 10 Takte, bei allen weiteren nur noch 1 Takt.

Allgemein sind beliebige andere Definitionen denkbar und lassen andere Mikroarchitekturanalysen zu. Allgemein wird aber gefordert:

$$\hat{t}'[i,\vartheta] \leq t'[i,\vartheta] \tag{3.26}$$

Da die Befehle nicht einzeln betrachtet werden, wird der Beschleunigungsbegriff auf Basisblöcke erweitert. Die Beschleunigung $\hat{t}[b,\vartheta]$ eines Basisblocks errechnet sich durch Summierung wie folgt:

$$\hat{t}[b,\vartheta] := \sum_{i \in b} \hat{t}'[i,\vartheta] \tag{3.27}$$

Im linearen Problem wird die Anzahl der Ausführungen ab der zweiten Ausführung mit neuen Variablen gezählt. Diese sollen für $b \in B$ und $\vartheta \in T$ dann $\hat{c}[b,\vartheta]$ heißen.

Um die Beschleunigung im linearen Problem zu beschreiben, sei folgende neue Zielfunktion definiert:

$$\begin{array}{rcll} \hat{Z} & : & \mathbb{N}_0^{2\cdot|B\times T|} & \to \quad \mathbb{R}_{\geq 0} \\ & & (c[b,\vartheta],\hat{c}[b,\vartheta])_{b\in B,\vartheta\in T} & \mapsto \quad \sum\limits_{b\in B,\vartheta\in\tau(b)} t[b,\vartheta]\cdot c[b,\vartheta] - \hat{t}[b,\vartheta]\cdot\hat{c}[b,\vartheta] \end{array} \tag{3.28}$$

Bleibt nur noch, die Beziehung zwischen Ausführanzahl eines Basisblocks und der Ausführanzahl ab der zweiten Ausführung zu definieren. Der Zusammenhang lautet wie folgt:

$$\begin{array}{rl} & c[b,\vartheta] = 0 \text{ und } \hat{c}[b,\vartheta] = 0 \\ \text{oder} & c[b,\vartheta] > 0 \text{ und } \hat{c}[b,\vartheta] = c[b,\vartheta] - 1 \end{array} \tag{3.29}$$

Die *oder*-Beziehung kann man allerdings nicht im linearen Problem ausdrücken, da alle Nebenbedingungen implizit *und*-verknüpft sind und keine *oder*-Verknüpfungen zulässig sind. Da sehr viele *oder*-Verknüpfungen vorkommen, nicht nur vereinzelte, scheidet aus Komplexitätsgründen ebenfalls die Möglichkeit aus, für jede Kombination von Fällen ein eigenes lineares Problem zu erzeugen, dieses für sich zu lösen und die Ergebnisse miteinander zu vergleichen, um das passende auszuwählen, denn für n *oder*-Verknüpfungen müßte man 2^n ILPs lösen. Daher muß das Problem anders gelöst werden.

Der essentielle Zusammenhang in (3.29) ist sicherlich $\hat{c}[b, \vartheta] = c[b, \vartheta] - 1$. Der Fall $c[b, \vartheta] = 0$ ist der Spezialfall, daß ein Block gar nicht ausgeführt wird. Diesen Spezialfall kann man implizit machen, indem man ausnutzt, daß alle Variablen positiv sind, und daß die Zielfunktion die $\hat{c}[b, \vartheta]$ *nur* mit negativem Koeffizienten enthält. Da die Zielfunktion maximiert werden soll, müssen daher die $\hat{c}[b, \vartheta]$ möglichst klein bleiben. (3.29) wird daher wie folgt umformuliert:

Nebenbedingung (Ausführanzahlen ab der zweiten Ausführung)

$$\hat{c}[b, \vartheta] \geqq c[b, \vartheta] - 1. \tag{3.30}$$

Alle Belegungen von $\hat{c}[b, \vartheta]$ und $c[b, \vartheta]$, die (3.29) erfüllen, erfüllen auch (3.30). Da für jedes $\hat{c}[b, \vartheta]$ genau eine Nebenbedingung (3.30) vorkommt, wird $\hat{c}[b, \vartheta]$ tatsächlich minimiert. Da alle Variablen $\geqq 0$, führen die Aussagen also zur gleichen Variablenbelegung.

3.7.2 Genauere Schleifenbetrachtung

Die bisher beschriebenen Schleifenbedingungen lassen keine Unterschätzung der Ausführzeit zu. Dennoch kann man zu engeren Schranken gelangen, wenn man mehr Wissen über den Kontrollfluß in die Nebenbedingungen einfließen läßt.

Der Benutzer kann zusätzlich zu oberen Schranken, die dazu da sind, das Problem zu beschränken, untere Schranken angeben. Diese dienen in erster Linie dazu, die Inferierung unmöglicher Pfade zu verhindern. Das gilt insbesondere für Schleifen, bei denen der Benutzer die genaue Iterationsanzahl kennt. Dann kann die Nebenbedingung in (3.24) wie folgt umformuliert werden:

Nebenbedingung (Genaue Ausführanzahl einer Schleife)

$$c[b_i, \vartheta \circ \mathsf{F}[b_h]] + c[b_i, \vartheta \circ \mathsf{O}[b_h]] = \tilde{k}_{\text{rep}}[b_h, \vartheta] \cdot c[\mathit{entry}(l), \vartheta] \tag{3.31}$$

Der Grund, warum zusätzliche Nebenbedingungen eine Verbesserung versprechen, ist, daß keine Nebenbedingungen für jeden einzelnen Aufruf einer Schleife formuliert werden können. Die Nebenbedingungen beschreiben daher nur die *Summen* der Ausführanzahlen der einzelnen Aufrufe.

Es soll nun untersucht werden, welche falschen Pfade noch inferierbar sind. Angenommen, ein Basisblock b wird innerhalb der Schleife unmittelbar vor dem Verlassen der

Schleife ausgeführt. Ist dieser in der ersten Iteration als am (always miss) kategorisiert worden und in allen weiteren als ah (always hit), dann kann es sein, daß ein ganzer Schleifendurchlauf weniger lange dauert, als *b* beim ersten Durchlauf der Schleife auszuführen, also die Schleife zu verlassen. Folgendes Beispiel veranschaulicht den Gedanken:

while *true* **do**
    if *A* **then** *b*; **break** ; **fi**
    ...
done

b wird in diesem Programm direkt vor dem Verlassen der Schleife ausgeführt.

Um die zusätzlichen Nebenbedingungen zu motivieren, die diese Situation beschreiben, sei auf Abbildung 3.5 verwiesen. Aufgrund der Tatsache, daß die Nebenbedingungen sich auf *alle* Schleifenaufrufe eines Kontextes beziehen, können noch unmögliche Pfade inferiert werden. Das linke Bild zeigt, daß die Schleife viermal aufgerufen wurde; einmal wurde sie während der ersten Iteration schon verlassen. Daraus folgt, daß in allen weiteren Iterationen diese Schleife auch nicht durchlaufen werden kann, wie fälschlicherweise inferiert wurde. Schlimmer noch; es wurden 17 Ausführungen eines inneren Basisblocks inferiert, wo nur maximal 16 möglich sind. Mit einmaligem Verlassen der Schleife bei der ersten Iteration schrumpft dies auf 12, doch die bestehenden Nebenbedingungen lassen noch die falschen Überschätzungen zu.

Allgemein läßt sich hierfür keine endliche Menge linearer Bedingungen formulieren. Es müßte gefordert werden, daß die Schleife bei jedem Aufruf ab der zweiten Iteration einmal weniger ausgeführt wird, als insgesamt. Dies läßt sich mit $b_h = head(l)$ wie folgt formulieren (falls $c[entry(l), \vartheta] > 0$ und $\sum_{e_b \in back(l)} c[e_b, \vartheta \circ \mathsf{F}[b_h]] > 0$):

$$\frac{c[b_h, \vartheta \circ \mathsf{F}[b_h]] + c[b_h, \vartheta \circ \mathsf{O}[b_h]]}{c[entry(l), \vartheta]} = \frac{c[b_h, \vartheta \circ \mathsf{O}[b_h]]}{\sum_{e_b \in back(l)} c[e_b, \vartheta \circ \mathsf{F}[b_h]]} + 1. \tag{3.32}$$

Diese Bedingung ist offensichtlich nicht linear und läßt sich also in ILP nicht umformulieren.

Aufgrund dieses Problems muß man andere Lösungen finden. Statt obigen Zusammenhang anzugeben, kann zumindest die Überschätzung der Durchläufe der Rückkanten verhindert werden. Als zweites wird bei entsprechendem Wissen verhindert werden, daß die Schleife bei der ersten Iteration verlassen werden kann. Abbildung 3.5 zeigt, wie sich die Vorhersagen im Beispiel ändern.

Die Anzahl der Durchläufe der Rückkanten ab der zweiten Iteration ist leicht zu beschränken; sie werden zweimal weniger durchlaufen als die Schleife iteriert. Analog zu (3.24) und (3.31) ergibt sich für eine Schleife l und $\forall \vartheta \in \tau(pre(l))$:

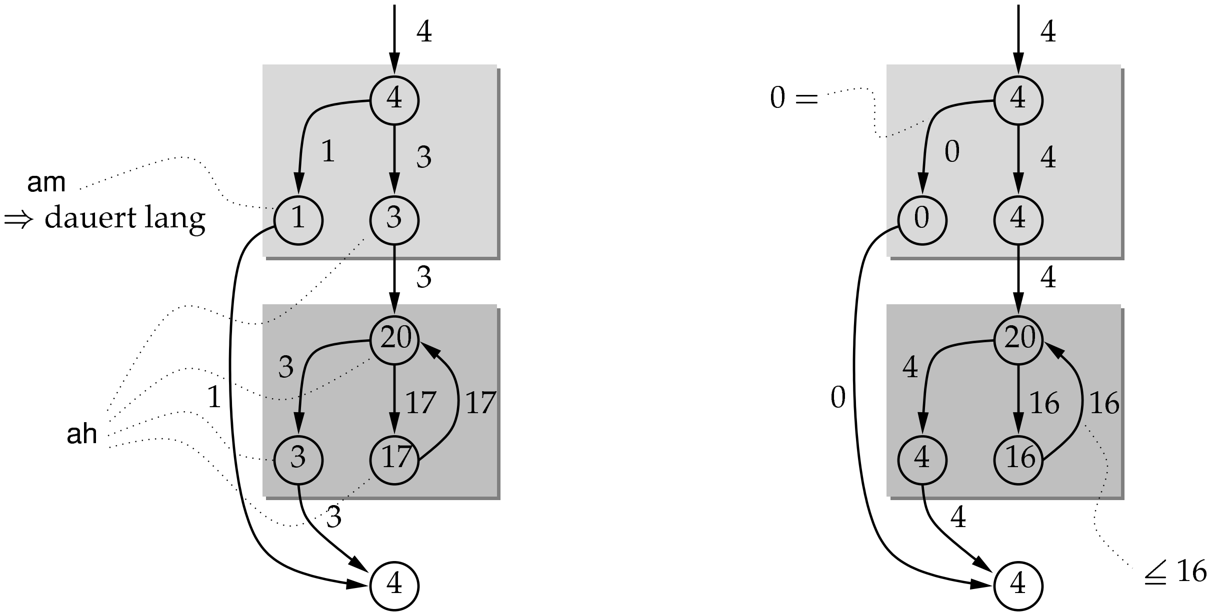


Abbildung 3.5: Gegenüberstellung von gefundenen Durchlaufanzahlen ohne (links) und mit (rechts) den Bedingungen aus (3.33) und (3.34). Im Bild wird angenommen, die Schleife werde viermal aufgerufen und fünfmal durchlaufen. Die Schleifenbedingung ist am Anfang, also wird der Kopf sechsmal durchlaufen. Die Schleifen sind einmal ausgerollt dargestellt.

Nebenbedingung (Beschränkung der Rückkanten von Schleifen)

$$\sum_{e_b \in back(l)} c[e_b, \vartheta \circ \mathsf{O}[b_h]] \;\; \Delta \;\; (\tilde{k}_{\text{rep}}[b_h, \vartheta] - 2) \;\cdot\; c[entry(l), \vartheta] \tag{3.33}$$

Dabei ist $\Delta \in \{\trianglelefteq, =, \trianglerighteq\}$ abhängig davon zu wählen, wieviel über die Schleife bekannt ist, wie schon zuvor für (3.24) und (3.31) besprochen.

Wenn bekannt ist, daß eine Schleife mindestens einmal ausgeführt wird, so können Nebenbedingungen erzeugt werden, die bestimmen, daß die Austrittskanten der Schleife in der ersten Iteration nicht durchlaufen werden dürfen. Die Nebenbedingung dazu lautet für alle $e_x \in exit(l)$ und alle $\vartheta \in \tau(pre(l))$:

Nebenbedingung (Erzwingen der ersten Iteration)

$$c[e_x, \vartheta \circ \mathsf{F}[head(l)]] = 0 \tag{3.34}$$

Es ist anzumerken, daß entsprechende Bedingungen auch für rekursive Prozeduraufrufe angegeben werden können. Die Ungleichungen sind ähnlich. Im Analyse-Tool sind sie im Moment jedoch noch nicht implementiert, da die rekursiven Testprogramme speziell zum Testen des Analyse-Tools geschrieben wurden, die Schleifen hingegen oft auftraten und damit praktisch relevanter waren.

3.7.3 Benutzerdefinierte Nebenbedingungen

Die hier aufgestellten Nebenbedingungen reichen nicht immer aus, um eine enge Schranke für die Maximallaufzeit des untersuchten Programmes zu berechnen, denn in vielen Programmen gibt es oft komplizierte, nicht automatisch erkennbare Zusammenhänge. Dazu sind frei definierbare Nebenbedingungen nötig, die der Benutzer angeben kann.

Beispiele für benutzerdefinierte Nebenbedingungen werden in den folgenden Kapiteln gegeben werden.

3.7.4 Lösbarkeit II

In diesem Abschnitt soll kurz erklärt werden, warum auch das verbesserte lineare Problem lösbar ist.

Die Korrektheit der vom Benutzer aufgestellten Nebenbedingungen wird vorausgesetzt.

Die neue Zielfunktion beschreibt immer noch eine obere Schranke für die Maximallaufzeit, denn die als pers kategorisierten Befehle werden ab der zweiten Ausführung garantiert im Cache sein und deswegen so schnell wie eine als ah kategorisierte Anweisung ausgeführt. Die eingeführten Nebenbedingungen für die neuen Variablen dienen dazu, die Schranke eng zu machen – eine Unterschätzung ist nicht möglich, denn sie entsprechen genau dem Verhalten der Maschine.

Auch die neuen Nebenbedingungen für Schleifen sind korrekt, denn sie beschreiben den Kontrollfluß genauer als die schon bestehenden, lassen aber keine Unterschätzung zu.

Die neue Zielfunktion enthält negative Koeffizienten. Für diese zugehörigen Variablen muß eine untere Schranke angegeben werden, damit die Funktion beschränkt bleibt. Alle Variablen müssen aber $\geqq 0$ sein nach Voraussetzung, so daß die neue Zielfunktion nach oben beschränkt bleibt.

Eine Variablenbelegung findet man wie zuvor deswegen, weil das Programm terminiert und damit eine Variablenbelegung definiert. Diese ist zulässig, da die Nebenbedingungen das Programm korrekt beschreiben.

3.7.5 Verkleinerung der Problemgröße

Die in den vorangegangenen Abschnitten vorgestellte Konstruktion des linearen Problems enthalten keine Optimierungen in Hinblick auf die Problemgröße. Bei der Implementierung fiel auf, daß es möglich ist, das Problem stark zu verkleinern, ohne aufwendige Algorithmen zu benutzen. Das lag vor allem an der Struktur der Kontrollflußgraphen, die vorlagen. Es kamen meist wenige ein- und ausgehende Kanten bei den Basisblöcken vor, und man kann mutmaßen, daß die meisten Programme eine ähnlich einfache Struktur aufweisen. Daher sind einige Vereinfachungen möglich.

Die folgenden Vereinfachungen sind im Programm implementiert:

Gleichungen der Form $A = B$. Diese Bedingungen kann man durch Umbenennen der vorkommenden Variablen leicht entfernen, d. h. man identifiziert A mit B und entfernt die Nebenbedingung $A = B$.

Paare der Form $\sum A_i = C$ und $\sum A_i = D$. Diese Gleichungspaare implizieren $C = D$. Somit kann im gesamten linearen Problem $C = D$ gesetzt werden und eine der beiden Gleichungen entfernt werden.

Durch diese Vereinfachungen kann es nun auftreten, daß Variablen mehrfach in einer Nebenbedingung auftreten. Man kann also als weitere Vereinfachung diese Summanden zusammenfassen: $a \cdot x + b \cdot x = (a + b) \cdot x$. Diese Vereinfachung verkleinert allerdings nur die Datei, die das Problem enthält, denn der LP-Löser trägt die Koeffizienten sowieso in eine Matrix ein, so daß er implizit diese Vereinfachung vornimmt.

Es bleibt noch anzumerken, daß die obigen Vereinfachungen rein auf der Ebene des linearen Problems geschehen. D. h. sie lassen sich auf beliebige lineare Probleme anwenden. Dies legt die Schlußfolgerung nahe, daß ihre Implementierung im Programm, das das Problem erzeugt, nicht richtig gewählt ist – der LP-Löser sollte diese Vereinfachungen vor Beginn der eigentlichen Arbeit vornehmen. Allerdings ist die zweite der beiden vorgestellten Vereinfachungen aufgrund der Permutationsmöglichkeiten bei den Nebenbedingungen und den auftretenden Variablen im allgemeinen Fall komplexer als in diesem speziellen. Das implementierte Analysetool kann daher Regelmäßigkeiten bei der Erzeugung ausnutzen und die Optimierung effizienter durchführen als der LP-Löser im allgemeinen Fall.

3.8 Bedeutung des LP-relaxierten Problems

Wie bekannt ist (*siehe* [Sei96, NW88, Sch96a]), ist das Problem, ILPs zu lösen, $\mathcal{NP}$-schwer. LPs zu lösen hingegen ist wesentlich leichter, und es gibt Algorithmen, die in polynomieller Zeit arbeiten. Daher liegt die Frage nahe, was geschieht, wenn das oben erzeugte ganzzahlige lineare Problem relaxiert und erst dann gelöst wird.

Ist man nur an der WCET eines Programmes interessiert und legt keinen Wert auf die exakte Berechnung der Ausführanzahlen für Basisblöcke, so kann man auch das relaxierte Problem lösen. Der berechnete Wert der Zielfunktion ist sicherlich größer als der beim zugehörigen ILP, d. h. die berechnete Schranke für die WCET ist korrekt, da weniger Einschränkungen vorhanden sind.

Viertes Kapitel

Annotationssprache

In diesem Kapitel wird beschrieben, auf welche Weise das implementierte Analysetool „Phobos“ die Benutzerangaben erhält. Die Angabe der Benutzerannotationen erfolgt auf Quelltextebene, damit der Benutzer sich leicht zurechtfinden kann, während Phobos auf Maschinenbefehlsebene analysiert, damit alle Aspekte der Maschine betrachtet werden können. Hieraus ergeben sich Probleme, da die Zuordnung von Quelltextzeile zu Basisblöcken sehr schwierig ist. Im Bereich der Debugger ist das Problem ebenso wohlbekannt (*siehe* [CG92, ATG96, BB94, TG98]). Die Ursachen für die Zuordnungsschwierigkeiten sind in erster Linie die Optimierungen, die der Compiler beim Eingabeprogramm anwendet. Die Annotationssprache wurde daher in Hinblick auf diese Probleme speziell entworfen.

4.1 Benutzerannotation

Es gibt zwei Arten von Benutzerannotationen.

(i) Maximale Ausführanzahlen für Schleifen und Rekursionen. Dabei ordnet eine Relation Basisblöcken maximale Ausführanzahlen zu. Die Basisblöcke, die in Frage kommen, sind Köpfe von Schleifen und Aufrufsblöcke. Formal sind diese Annotationen also Tupel $\in \big(\mathit{head}(\mathcal{L}) \cup \mathcal{C}\big) \times \mathbb{N}_0$.

(ii) Beliebige lineare Beziehungen zwischen Ausführanzahlen von Basisblöcken. Der Benutzer kann diese Nebenbedingungen frei definieren; sie werden ohne Änderung zum linearen Problem hinzugefügt. Formal sind dies Ungleichungen über Linearkombinationen beliebiger Basisblockausführanzahlen, d. h. für $n \in \mathbb{N}_0, a \in \mathbb{R}$: $\sum_{i \in n} a_i\, c[b_i, \vartheta_i] \leqq a$ wobei $b_i \in B$ und $\vartheta_i \in \tau(b_i)$. Andere Vergleiche neben $\leqq$ und Ungleichungen mit Linearkombinationen auf jeder Seite lassen sich wie schon in Abschnitt 2.2.1 auf Seite 26 beschrieben leicht auf diese Grundform zurückführen.

Wie man sieht, haben die Benutzerangaben Basisblöcke als Bezugspunkte. Herauszufinden, welche Basisblöcke der Benutzer meint, wenn er Annotationen an den Quelltext schreibt, ist sehr schwierig und war daher der wichtigste Punkt bei der Entwicklung der Annotationssprache.

4.2 Aufbau der Sprache *Pholog*

Die Annotationssprache *Pholog*[1] ist sehr einfach. Es sind keine Prozedurdeklarationen möglich. Es gibt eine grundlegende Unterscheidung zwischen *Befehlen* und *Werten*. Die Auswertung erfolgt, sobald Prozeduren Werte benötigen, d. h. die Auswertung ist verzögert (*lazy*).

Die Syntax der Sprache und die Befehlsnamen sind an die Programmiersprache Lisp (*siehe* [SJ90]) angelehnt. Atomare Werte werden direkt als Werte interpretiert, komplexe Werte stehen in runden Klammern. Dabei wird der erste Wert in den runden Klammern als Funktionsname interpretiert, die restlichen als Argumente dieser Funktion.

Als Datentyp gibt es im Moment pseudoreelle Zahlen, d. h. den C-Typ `double`, Strings, Symbole und an wenigen speziellen Stellen Tupel aus Zahlen.

4.3 Werte

Es gibt folgende Arten von Werten:

(i) Konstanten. Im üblichen Sinne: z. B. `17`, `2.56`, `2e-5`, `"Hallo\n"`.

(ii) Variablen. Alles, was eine abgeschlossene Zeichenkette ohne Sonderzeichen im Sinne von Lisp ist und dennoch keine Zahl, wird als Variablenname interpretiert. Z. B. `a`, `hundert`, `a-1`, `7+4*z-1`.

(iii) Komplexe Werte. Funktionsaufrufe: `(+ 4 5)`, `(sqrt (if a 5 10))`.

[1] von griechisch $\varphi\acute{o}\beta o\varsigma$ (Name des Projekts) + $\lambda\acute{o}\gamma o\varsigma$ (Wort)

Es sind die folgenden Funktionen bekannt:

`(identity` *`Wert`*`)` Ergibt *Wert*.

`(+` *`[Wert]`* `...)` Summe beliebig vieler Zahlen. `(+)` ergibt 0.

`(-` *`Wert`* `...)` Bei nur einem Argument: der negierte Wert. Sonst die Differenz zwischen erstem Argument und der Summe aller weiteren Argumente. `(-)` ist ein Fehler.

`(1+` *`Wert`*`)` Um eins erhöhtes Argument.

`(1-` *`Wert`*`)` Um eins verringertes Argument.

`(*` *`[Wert]`* `...)` Produkt beliebig vieler Zahlen. `(*)` ergibt 1.

`(/` *`Wert`* `...)` Bei nur einem Argument: Kehrwert. Sonst der Quotient des ersten Arguments und dem Produkt aller weiteren Argumente. `(/)` ist ein Fehler.

`(%` *`Wert1 Wert2`*`)` Ganzzahliger Rest der Division. Nimmt genau zwei Argumente, rundet sie auf ganze Zahlen und gibt den Rest der Division aus.

`(rem` *`Wert1 Wert2`*`)` Rest der Division. Wie `%`, kann aber auch quasireellwertige Argumente verarbeiten.

`(floor` *`Wert`*`)` Auf ganze Zahl abgerundetes Argument.

`(ceiling` *`Wert`*`)` Auf ganze Zahl aufgerundetes Argument.

`(round` *`Wert`*`)` Auf nächste ganze Zahl gerundetes Argument.

`(sqrt` *`Wert`*`)` Quadratwurzel des Arguments.

`(exp` *`Wert1 [Wert2]`*`)` Falls nur ein Argument: $e^{\mathtt{Wert1}}$. Sonst: $\mathtt{Wert1}^{\mathtt{Wert2}}$.

`(expt` *`Wert1 [Wert2]`*`)` Synonym für `exp`.

`(log` *`Wert1 [Wert2]`*`)` Logarithmus zur Basis e, falls nur ein Argument, sonst Logarithmus des ersten Arguments zur Basis des zweiten Arguments.

`(if` *`Wert Wert1 Wert2`*`)` Falls der Betrag des ersten Arguments ungleich 0 (siehe auch nächsten Abschnitt) ist, der Wert des zweiten Arguments, sonst der Wert des dritten.

`(min` *`Wert`* `...)` Minimum der angegebenen Werte.

`(max` *`Wert`* `...)` Maximum der angegebenen Werte.

`(length` *`String`*`)` Anzahl der Zeichen des Strings.

`(values` *`Wert1`* `...)` Faßt mehrere Werte zu einem Tupel zusammen. Die vollständige Implementierung hiervon würde zusätzliche Datentypen erfordern. Daher ist dieser Befehl derzeit nur dort möglich, wo explizit darauf hingewiesen wird, daß der Aufrufer das Ergebnis dieser Funktion versteht. Es ist keine Schachtelung möglich; Funktionen und Befehle, die den speziellen Funktionswert nicht verstehen, benutzen *Wert1* und ignorieren alle weiteren.

4.4 Vergleichsoperatoren und logische Operatoren

Alle Vergleichsoperatoren gibt es in zwei Ausführungen: für genaue Überprüfung der Werte und für eine Gleichheit mit einem gewissen Spielraum ε (Gleichheit bedeutet: $a \cong b :\Longleftrightarrow |a - b| < \varepsilon$). ε ist auf $1 \cdot 10^{-8}$ voreingestellt, kann aber mit `(setq epsilon` *`Wert`*`)` umgestellt werden. ε wird auch bei vielen anderen internen Rechnungen benutzt, etwa beim `if`-Befehl oder beim Entscheiden, ob $a \cdot x$ zu 0 vereinfacht werden soll, weil $|a| < \varepsilon$, oder bei ähnlichem.

Es gibt die Vergleichsoperatoren <, >, <=, >=, = und /= mit der Semantik von Lisp, also z. B.:

`(< 5)` ergibt 1.

`(< 5 6)` ergibt 1.

`(< 5 6 10)` ergibt 1.

`(< 5 6 10 2)` ergibt 0.

`(< 0 (/ epsilon 2))` ergibt 0, denn $0 \cong \frac{\varepsilon}{2}$.

`(< 0 epsilon)` ergibt 1, denn $0 \ncong \varepsilon$.

`(>= 0 (/ epsilon 2))` ergibt 1.

Soll ein genauer Vergleich ohne Benutzung von ε durchgeführt werden, so kann ein ! an den Operatornamen angehängt werden:

`(<! 0 (/ epsilon 2))` ergibt 1.

Es gibt die üblichen logischen Operatoren: `and`, `or`, `not` und zusätzlich `bool`, der aus quasireellwertigen Zahlen boolesche Werte erzeugt, also 1 oder 0, ohne den logischen Wert zu ändern.

4.5 Befehle und ihre Argumente

Im Gegensatz zu Funktionen können Befehle nicht innerhalb von Formeln vorkommen. Befehle können verschiedenartige Argumente haben. Es gibt folgende Arten von Argumenten:

(i) Werte im Sinne des vorangehenden Abschnittes.

(ii) Nebenbedingungen. In einer speziellen Syntax dürfen hier Nebenbedingungen angegeben werden. Diese werden im nächsten Abschnitt vorgestellt.

(iii) Basisblocknamen. Sie sind syntaktisch mit Variablen identisch.

Die Befehle definieren die eigentlichen Benutzereingaben für Phobos, die in Abschnitt 4.1 auf Seite 57 beschrieben wurden. Idealerweise gibt es für jede Benutzerannotation *einen* Bezugsblock, nämlich den, für dessen Erzeugung der Compiler die Quelltextzeile, in die Annotation auftaucht, benutzt hat. Da oft keine oder mehrere Basisblöcke nach dieser Definition vorhanden waren, gibt es Befehle, die nur nach bestimmten Arten von Basisblöcken suchen und einen passenden als Bezugspunkt wählen. Unter Umständen werden auch einige Zeilen vor und hinter der Benutzerannotation mit in Betracht gezogen, um eine gewisse Fehlertoleranz zu gewährleisten.

Bei einigen Befehlen ist die Art des Bezugsblock direkt aus der Art der Benutzerannotation ersichtlich, z. B. wird für eine Iterationsbegrenzung ein Schleifenkopf benötigt. Die frei definierbaren Nebenbedingungen haben hingegen mehrere Bezugsblöcke; jeweils für eine Variable der Linearkombination. Um diese Bezugsblöcke zu definieren, kann man bestimmte Blöcke mit *Marken* benennen (mit der `label`-Befehlsgruppe).

Die Annotationssprache kann für jeden VIVU-Kontext eigene lokale Variablenbindungen verwalten, um für die Unterscheidungen, die die VIVU-Analyse zuläßt, auch verschiedene Variablenwerte zuzulassen und damit die Flexibilität zu erhöhen. Zusätzlich kennt die Sprache globale Variablen.

Wenn für einen Befehl logisch ein Basisblock als Bezugspunkt vorhanden ist, so werden Argumente für gewöhnlich in all seinen Kontexten ausgewertet. Die lokalen Variablenbindungen sind dann bekannt. Ansonsten wird *frei* ausgewertet, d. h. nur einmal und ohne Zugriff auf lokale Variablen. Mit dem `in-context-of`-Befehl kann dann ein Basisblock als Bezugspunkt angegeben werden. Im nächsten Abschnitt wird ein Beispiel für die Auswertung gegeben.

Es gibt folgende Befehle:

`(defconst` *`Bezeichner1 Wert1`* `...)` Definiert und setzt den Wert einer oder mehrerer Konstanten. Der Wert kann nicht mehr geändert werden (auch nicht gelöscht). Der Befehl ergibt einen Fehler, falls die Variable schon belegt ist. Wertet frei aus.

`(defvar` *`Bezeichner1 Wert1`* `...)` Definiert und setzt den Wert einer oder mehrerer Variablen, falls diese noch keinen Wert haben. Ansonsten wird die Variable nicht verändert. Wertet frei aus.

`(setq` *`Bezeichner1 Wert1`* `...)` Definiert und setzt (oder überschreibt) den Wert einer oder mehrerer Variablen. Wertet frei aus.

`(unsetq` *`Bezeichner1`* `...)` Löscht den Wert und die Definition einer oder mehrerer Konstanten.

`(defterm` *`Bezeichner1 Ausdruck1`* `...)` Benennt Ausdrücke. Wertet die Ausdrücke aber nicht aus, sondern ersetzt sie nachher textuell.

`(undefterm` *`Bezeichner1`* `...)` Löscht Bezeichnungen für Ausdrücke.

`(defparam` *`Bezeichner1 Wert(e)1`* `...)` Definiert mehrere Werte lokal zu einem Prozeduraufruf. Als `Wert(e)` kann entweder ein Wert angegeben werden, oder zwei Werte (zusammengefaßt mit `values`). Wird nur ein Wert angegeben, so wird dieser sowohl für den ersten als auch für den rekursiven Kontext der Prozedur benutzt, sonst werden zwei verschiedene Werte zugewiesen. Die Werte werden in allen Kontexten des Aufrufers ausgewertet, so daß an einem Prozeduraufruf mehrere Werte der gleichen Variablen stehen können. Es können gleichzeitig mehrere Bezeichner definiert werden aber es kann nicht auf schon definierte Parameter zugegriffen werden, z. B. `(defparam a 5 b (values 10 11))`. Aber `(defparam a 5 b a)` benutzt zur Definition von *b nicht* 5, sondern einen evtl. in der Bindung des *Aufrufers* bekannten Wert von *a*.

`(deflocal` *`Bezeichner1 Wert(e)1`* `...)` Definiert und setzt (oder überschreibt) mehrere Werte lokal zu einer Prozedur (entspricht dem globalen `setq`). Dies ist gleichbedeutend mit einem `defparam`-Befehl für alle Prozeduraufrufen in allen ihren Kontexten, nur daß die Bindungen der Parameter schon sichtbar sind, z. B. `(deflocal a 5 b (values 10 11) c (values a b))`.

`(deflocal (`*`Bezeichner1 Wert(e)1`* `...) ...)` Definiert Werte lokal zu Schleifen. Für jede Schachtelungsebene muß ein Argument angegeben werden. Die Semantik der Argumente ist wie bei der Variante für Prozeduren. Soll für eine Schleife kein Wert gesetzt werden, so muß trotzdem `()` angegeben werden, um die Schleifenstruktur zu finden. Wertet in allen Kontexten des Schleifenkopfes aus.

`(iterations` *`Begrenzung`* `...)` Setzt die Begrenzung für mehrere Schleifen. Benutzt die Variablenbelegungen für die erste Iteration der Schleife.

`(funcalls` *`Begrenzung`*`)` Setzt die Minimal- und/oder Maximalanzahl von Prozeduraufrufen. Benutzt die Variablenbelegungen für die erste Inkarnation der Prozedur.

`(loop-type` *`Wert`* `...)` Bestimmt die Art einer Schleife. *Siehe* Abschnitt 5.3.1 auf Seite 77 für eine Liste der Werte. Es wird frei ausgewertet.

`(label` *`Bezeichner`*`)` Benennt den Basisblock einer Quelltextzeile mit der kleinsten Adresse.

`(label>` *`Bezeichner`*`)` Benennt den Basisblock einer Quelltextzeile mit der größten Adresse.

`(iflabel` *`Bezeichner1 Bezeichner2`*`)` Benennt die Nachfolger eines Basisblocks mit zwei Nachfolgern einer Quelltextzeile. Bei mehreren werden die mit den kleinsten Adressen gewählt.

`(iflabel>` *`Bezeichner1 Bezeichner2`*`)` Benennt die Nachfolger eines Basisblocks mit zwei Nachfolgern einer Quelltextzeile. Bei mehreren werden die mit den größten Adressen gewählt.

`(calllabel` *`Bezeichner`*`)` Benennt den Basisblock eines Prozeduraufruf einer Quelltextzeile. Bei mehreren wird der mit der kleinsten Adresse gewählt.

`(calllabel>` *`Bezeichner`*`)` Benennt den Basisblock eines Prozeduraufruf einer Quelltextzeile. Bei mehreren wird der mit der größten Adresse gewählt.

`(liblabel` *`Bezeichner`*`)` Benennt den Basisblock eines Bibliotheksprozeduraufrufs einer Quelltextzeile mit der kleinsten Adresse.

`(liblabel>` *`Bezeichner`*`)` Benennt den Basisblock eines Bibliotheksprozeduraufrufs einer Quelltextzeile mit der größten Adresse.

`(looplabel` *`Bezeichner`* `...)` Benennt die Köpfe mehrerer Schleifen. Wie sonst auch bei Schleifenbefehlen muß die Schachtelungsstruktur durch die Anzahl der Argumente definiert werden, und Definitionen können durch `()` weggelassen werden.

`(looplabel` *`(Bezeichner1 Bezeichner2)`* `...)` Benennt die Pre-Header und Köpfe mehrerer Schleifen, sonst wie der vorangehende Befehl. *Bezeichner1* wird der Name des Pre-Headers, *Bezeichner2* der des Kopfes. Beide dürfen auch `()` sein.

`(outlabel` *`Bezeichner1 Bezeichner2`*`)` Benennt die zwei Nachfolger eines Basisblocks einer Quelltextzeile.

`(constraint` *`Nebenbedingung`*`)` Definiert eine Nebenbedingung. Wertet frei aus.

`(in-context-of` *`Name Befehl`*`)` Wertet den Befehl in den Kontexten des angegebenen Basisblocks aus. Bewirkt nur bei frei auswertenden Befehlen eine Änderung.

`(unwind-protect` *`Befehl`* `[`*`Befehl2`*`])` Verhindert die Erzeugung eines Fehlers bei der Auswertung des Befehls. Selbst wenn bei der Ausführung von *`Befehl1`* ein Fehler auftritt, wird trotzdem *`Befehl2`* ausgeführt, falls vorhanden. Zudem wird der Fehler als abgearbeitet betrachtet und führt nicht zu einem Fehlerabbruch beim Aufrufer von `unwind-protect`.

`(progn` *`Befehl1`* `...)` Zum Gruppieren von Befehlen. Die Befehle werden einfach hintereinander ausgewertet.

`(if` *`Wert Befehl1 [Befehl2]`*`)` Prozedurales Gegenstück zum funktionalen `if`. In dieser Variante darf der `else`-Teil weggelassen werden.

`(when` *`Wert Befehl1`* `...)` Wie `(if` *`Wert`* `(progn` *`Befehl1`* `...))`.

`(unless` *`Wert Befehl1`* `...)` Wie `(when (not` *`Wert`*`)` *`Befehl1`* `...)`.

`(ignore` *`String`*`)` Ignoriert eine Prozedur bei der Analyse.

`(ignore-silently` *`String`*`)` Ignoriert eine Prozedur bei der Analyse, ohne Warnungen auszugeben.

`(unignore` *`String`*`)` Das Gegenteil von `ignore-silently`. Mit einer Folge von `unignore` und `ignore` kann man das Warnen wieder einschalten. Phobos ignoriert von vornherein die Bezeichner `start`, `__do_global_dtors`, `__do_global_ctors`, `_etext`, `__main` und `-lg` da sie nicht vom Benutzer geschrieben wurden und zum Kontrollgerüst der Binärdatei gehören. Mit diesem Befehl kann man erzwingen, daß sie betrachten werden.

`(print` *`Wert`*|*`String`* `...)` Schreibt die Werte und Strings auf die Standardausgabe. Zwischen zwei Werte wird automatisch eine Leerstelle gedruckt, falls kein String dazwischen steht. Am Schluß wird ein Zeilenumbruch ausgegeben, wenn das letzte Argument kein String ist. Wertet frei aus.

`(sys:dump-binding` *`Bezeichner1`* `...)` Schreibt die gesamte Bindungsumgebung an einem Basisblock auf. Ein $*$ steht für alle Bindungen.

Man sieht hier, daß es zum Benennen von Basisblöcken sehr viele verschiedene Befehle gibt. Diese wurden wegen der Veränderungen, die der Compiler durch Optimierung erzeugt, eingeführt. In vielen Fällen war es so, daß mehrere Basisblöcke pro Quelltextzeile vorkamen. Um dem Rechnung zu tragen, gibt es von den Benennungsbefehlen jeweils zwei Formen. Die Standardform geht davon aus, daß der Benutzer den *ersten* Basisblock auf der Zeile meint, d. h. den mit der kleinsten Adresse. Die Form auf > benennt hingegen jeweils den Basisblock mit der größten Adresse.

Um die etwas komplizierte Auswertung mit lokalen Bindungen zu veranschaulichen, hier ein Beispiel.

Beispiel

```
void a() {
  eine_anweisung;     //$ (label nr1) (deflocal z (+ y 100))
}
void b() {
  eine_anweisung;     //$ (label nr2)
  a();                //$ (defparam y (+ x 1) (+ x 2))
  ...
  b();
}
 ...
L1: a();               //$ (defparam x 10 y 7)
L2: b();               //$ (defparam x 20)
L3: b();               //$ (defparam x (values 30 40))
```

Das Auswerten von `(in-context-of nr2 (print "x=" x))` ergibt dann folgende Ausgabe (ohne die Kommentare und bis auf Permutationen, denn die Kontexte müssen nicht sortiert sein):

```
x=20        von L2 aus nichtrekursiv
x=20        von L2 aus rekursiv
x=30        von L3 aus nichtrekursiv
x=40        von L3 aus rekursiv
```

Und `(in-context-of nr1 (print "x=" x " z=" z))` wird folgendes ausgeben:

```
x=10 z=107    von L1 aus nichtrekursiv
x=20 z=121    von L2 aus nichtrekursiv
x=20 z=121    von L2 aus rekursiv
x=30 z=131    von L3 aus nichtrekursiv
x=40 z=141    von L3 aus rekursiv
```

Der Grund, daß man den Effekt von `(+ x 2)` nicht sieht, ist, daß die Werte im rekursiven Kontext von *a* gespeichert sind. Doch es gibt keine rekursiven Aufrufe von *a*.

Die Sichtbereiche der Befehle `deflocal` und `defparam` entsprechen genau den Sichtbereichen in C. Beim Aufrufer wird die Bindung des Aufrufers benutzt, in der aufgerufenen Prozedur deren Bindung. `defparam` hat beim Aufrufer schon *Schreib*zugriff auf die Bindung des Aufgerufenen und definiert dort so die Parameter. `deflocal` hat für Lese- und Schreibzugriffe den gleichen Sichtbereich.

Schleifen verhalten sich genau wie Prozeduren. Wegen nur eines Eintrittspunktes ist der `defparam`-Befehl für die Schleifen nicht notwendig.

4.6 Begrenzungen

Schleifen können auf verschiedene Weisen begrenzt werden. Gibt man nur einen Wert an, so wird dieser Wert als obere Schranke gewählt. Man kann aber auch obere und untere Schranken für eine Schleife angeben. Es gibt folgende Begrenzungstypen:

`Wert` Definiert eine obere Schranke für die Iterationsanzahl.

`(at-most Wert)` Wie der vorangehende Befehl.

`(at-least Wert)` Definiert eine untere Schranke für die Iterationsanzahl.

`(between Wert1 Wert2)` Definiert *Wert1* als untere Schranke für die Iterationsanzahl und *Wert2* als obere.

`(exactly Wert1)` wie `(between Wert1 Wert1)` mit einmaliger Auswertung von *Wert1*.

`(plus Wert1 Wert2)` wie `(between Wert1 (+ Wert1 Wert2))` aber mit nur mit einmaliger Auswertung von *Wert1*.

`(minus Wert1 Wert2)` wie `(between (− Wert1 Wert2) Wert1)` aber mit nur einmaliger Auswertung von *Wert1*.

`(plusminus Wert1 Wert2 [Wert3])` wie `(between (− Wert1 Wert2) (+ Wert1 Wert2 bzw. Wert3))` aber mit nur einmaliger Auswertung von *Wert1*. *Wert3* ist optional.

Wichtig zu wissen ist, daß die angegebenen Begrenzungen noch je nach Schleifentyp korrigiert werden. Ist als Schleifentyp `loop-start` angegeben, also eine Schleife mit Abbruchbedingung am Anfang, so wird zu *beiden* Schranken eins hinzuaddiert. Ist der Schleifentyp `loop-end`, also eine Schleife mit Abbruchbedingung am Ende, so wird nichts an den Werten geändert. Ist der Schleifentyp undefiniert (`loop-unknown`), also eine Schleife, die *wahrscheinlich* die Schleifenbedingung am Anfang hat, so wird nur zur *oberen* Schranke eins hinzuaddiert. Beim Begrenzen von rekursiven Aufrufen tritt keine Korrektur ein.

4.7 Allgemeine Nebenbedingungen

Nebenbedingungen setzen allgemein zwei Linearkombinationen von Variablen in Beziehung. Mit $\Delta \in \{<=, =, >=\}$ erhält man folgende Formen für Nebenbedingungen:

`(Δ` *`Linearkombination1 Linearkombination2`*`)` Definiert eine Nebenbedingung, die besagt, daß *`Linearkombination1`* Δ *`Linearkombination2`*. Die Variablen der Linearkombinationen sind die Namen von Basisblöcken, die die Ausführanzahl repräsentieren.

Will man z. B. aussagen, daß der Basisblock *nr*1 maximal 50mal öfter ausgeführt werden kann als Basisblock *nr*2, so läßt sich das mit folgender Nebenbedingung ausdrücken:

```
(constraint (<= nr1 (* 50 nr2)))
```

Fünftes Kapitel
Implementierung

In diesem Kapitel wird beschrieben, wie die Implementation des im dritten Kapitel erzeugten linearen Problems aussieht. Zunächst wird erläutert, wie das Programm implementiert wurde, und danach, wie man es benutzt. Dabei wird auf die Schwierigkeiten eingegangen, die entstanden sind.

5.1 Die Einzelteile

Das Projekt wurde in der Programmiersprache C++ geschrieben. Der Gesamtumfang beträgt ca. 9 500 Zeilen Quellcode. Das Projekt gliedert sich in drei Programmteile, die unterschiedliche Aufgaben lösen:

Deimos implementiert den ersten Schritt der Analyse. Es lädt das Kompilat des zu untersuchenden Programmes und erzeugt eine maschinenunabhängige Repräsentation aller benötigten Daten. Die Ausgabe von Deimos ist eine Lisp-lesbare Datei, die alle Informationen enthält, die die nachfolgende Analyse benötigt.

Phobos implementiert die eigentliche Erzeugung des linearen Problems. Phobos liest die nötigen Eingabedateien und schreibt das daraus abgeleitete lineare Problem in eine Datei. Diese kann mit einem LP-Löser gelöst werden. Dazu wurde lp_solve verwendet.

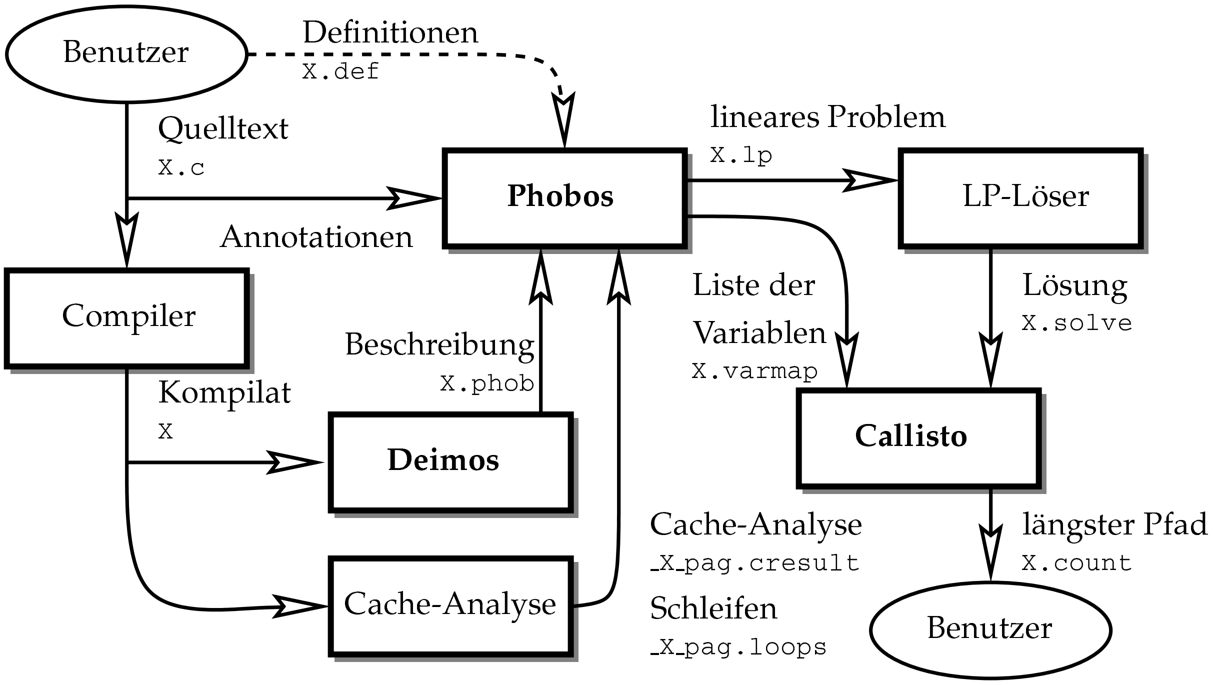


Abbildung 5.1: Ein Analyselauf eines Programmes `X` mit Deimos, Phobos und Callisto. Die Definitionsdatei ist optional.

Callisto implementiert den Vergleich dieser Vorhersage mit der tatsächlichen Maximallaufzeit.

Im folgenden werden die drei Programme und die Implementierung im einzelnen beschrieben. Abbildung 5.1 zeigt den Analyselauf schematisch.

5.1.1 Deimos

Deimos ist der kleinste bekannte Mond des Sonnensystems. Er hat einen Durchmesser von 12,6 *km und wiegt* $1{,}8 \cdot 10^{15}$ *kg.*
„In Greek mythology, Deimos is one of the sons of Ares (Mars) and Aphrodite (Venus); ‚deimos' is Greek for ‚panic'." [TNPb]

Die Programme, die untersucht werden sollten, lagen als SPARC-Executable vor. Die Ausgabe wurde mit gcc mit der Option `-g` erzeugt, so daß Debug-Informationen zur Verfügung standen. Diese wurden benötigt, um die Benutzereingaben in den Quelltexten dem Maschinencode zuordnen zu können. Zum Lesen der SPARC-Executables wurde die Bibliothek EEL (*siehe* [Lar96]) benutzt, die zu WARTS[1] gehört.

[1] Wisconsin Architectural Research Tool Set (*siehe* [WAR97])

Deimos liest nun zunächst das Programm ein und führt die Analysen, die EEL bieten, durch. EEL zerlegt das Programm zunächst in Basisblöcke, dann werden der Kontrollflußgraph und der Call-Graph aufgebaut und Schleifen gefunden. Der Call-Graph wird nach Zyklen durchsucht, um rekursive Aufrufe zu finden.

Sobald die Datenstruktur vollständig erzeugt ist, gibt Deimos den Graphen rekursiv aus.

5.1.2 Phobos

Phobos ist wie Deimos ein kleiner Marsmond. Er hat einen Durchmesser von 22,2 *km und wiegt* $1{,}08 \cdot 10^{16}$ *kg.*
„In Greek mythology, Phobos is one of the sons of Ares (Mars) and Aphrodite (Venus). ‚phobos' is Greek for ‚fear' (the root of ‚phobia')." [TNPc]

Phobos ist der größte der drei Projektteile. Er besteht aus ca. 8 200 Zeilen C++-Quellcode. Phobos liest die Programmbeschreibung, die Deimos erzeugt hat. Zusätzlich werden die Ergebnisse der Cache-Analyse von Christian Ferdinand gelesen. Diese enthalten auch die Informationen über den Kontext eines jeden Basisblocks. Weiterhin wird der Quelltexte des Eingabeprogrammes nach Benutzerannotationen durchsucht, um schließlich das lineare Problem zu erzeugen.

Nach dem Erzeugen des linearen Problems wendet Phobos die optimierenden Algorithmen an aus Abschnitt 3.7.5 auf Seite 54 an, die das entstandene Problem verkleinern.

Die Ausgabe von Phobos ist wahlweise im lp_solve- oder im CPLEX-Format. Zur Lösung der linearen Probleme wurde allerdings nur lp_solve verwendet. Von der Verwendung des kommerziell vertriebenen LP-Lösers CPLEX kann man eine Beschleunigung beim Lösen erwarten.

5.1.3 Callisto

Callisto ist der zweitgrößte Jupitermond. Er hat einen Durchmesser von 4 800 *km und wiegt* $1{,}08 \cdot 10^{23}$ *kg.*
„Callisto was a nymph, beloved of Zeus and hated by Hera. Hera changed her into a bear and Zeus then placed her in the sky as the constellation Ursa Major." [TNPa]

Nachdem die WCET-Vorhersage gemacht ist, wurde ein Programm-Tracer benutzt, um die tatsächliche Maximallaufzeit zu bestimmen. Callisto ist ein kleines Programm, das die Ausgaben von Phobos, lp_solve und auf Wunsch zum Vergleich vom Programm-Tracer liest und Ausgaben in dem Format erzeugt, das auch der Programm-Tracer ausgibt. Das Ausgabeformat des Tracers wird dabei so genau simuliert, daß man mit dem UNIX-Befehl `diff` die Ergebnisse überprüfen kann. Falls die Vorhersagen für jeden Basisblock übereinstimmen, treten keinerlei Abweichung auf, und die Dateien gleichenbis aufs Bit. Dies sollte die Bewertung der erzielten Ergebnisse erleichtern. Durch diese Übersetzung ist es weiterhin möglich, die Ausgaben der Programmpfadanalyse für weitere Programme als Eingabe zu benutzen, die sonst Eingaben des Tracers bekommen. Des weiteren kann Callisto die Ausgaben des Tracers mit denen der Programmpfadanalyse vergleichen und dann die Überschreitung der Schätzung des Analysetools von der tatsächlichen Maximallaufzeit in Prozent ausgeben. Bei Abweichungen reicht es nicht, die Ausführanzahlen der Basisblöcke direkt zu vergleichen: hat Phobos eine kleinere Ausführanzahl für einen Basisblock gefunden, so heißt das lediglich, daß der schlechteste Pfad woanders herführt, nicht aber, daß das Ergebnis falsch ist. Diese Vergleichsmöglichkeit wird in Abschnitt 6.1 auf Seite 88 noch beschrieben und unterscheidet sich vom einfacheren Vergleich der Taktzyklen, die der Tracer berechnet hat, mit dem Wert der Zielfunktion, den Phobos berechnet hat.

5.2 Probleme mit optimiertem Code

Wie schon an mehreren Stellen erwähnt, gab es Probleme mit optimiertem Code in den zu untersuchenden Kompilaten. Da auch die Benutzung von optimierenden Compilern ein wichtiger Aspekt bei diesem Projekt war, mußten Lösungen für dieses Problem gefunden werden. Optimierende Compiler können das Kompilat ziemlich deutlich beschleunigen und sind somit wichtig bei der Benutzung von Echtzeitsystemen, die die zur Verfügung stehenden Ressourcen möglichst effizient ausnutzen sollen. Ohne Optimierungen des Kompilats wären viele dieser Ressourcen verschwendet. Da man auch bei der Analyse des längsten Programmpfades daran interessiert ist, eine möglichst enge obere Schranke der Maximallaufzeit anzugeben, sollten die Optimierungen, die der Compiler bieten, nicht ausgeschaltet sein, sondern beide Programme sollten so benutzt werden, daß die Prozessorzeit effizient nutzbar ist. Die verwendeten Testprogramme wurden mühelos um einen Faktor 2 schneller, wenn Compiler-Optimierungen ausgeführt wurden. Die Analyse lieferte bei so optimierten Programmen nur um wenige Prozentpunkte schlechtere Vorhersagen als bei nicht optimierten Programmen.

Die Probleme, die durch Optimierung auftauchten, waren vielfältig. Zum einen wurde die Kontrollstruktur des Programmes teilweise verändert, wodurch die Zuordnung von Maschinenbefehlen zu Quelltextzeilen nur eingeschränkt möglich war. Um dieses Problem zu umgehen, wurden in die Beschreibungssprache, mit der der Benutzer den Quellcode annotieren kann, Befehle aufgenommen, die wenig abhängig von der Quellcodezeile sind und nicht auf Basisblockebene arbeiten, sondern deutlicher erkennbare Strukturen beschreiben.

Ein weiteres Problem trat durch Umstrukturierung vor allem an Schleifen auf. Der Compiler inferierte oft, daß eine Schleife mindestens einmal durchlaufen wird, und verschob dann entweder die Schleifenbedingung ans Ende der Schleife, damit sie einmal weniger ausgeführt werden muß, oder er rollte die Schleife einmal aus. Folgende Beispiele illustrieren das Problem:

for $i := 1$ **to** n **do** (* *wobei* $n \geqq 1$ *)
    Schleifencode
done

Da diese Schleife mindestens einmal ausgeführt wird wg. $n \geqq 1$, besteht die Möglichkeit, daß der Compiler sie wie folgt transformiert, um eine Abfrage der Schleifenabbruchbedingung einzusparen:

$i := 1;$
repeat
    Schleifencode
    $i := i + 1;$
until $i = n;$

Eine andere Variante der Optimierung ist die folgende:

$i := 1;$
Schleifencode
$i := i + 1;$ (* *oder auch* $i := 2$ *)
while $i \neq n$ **do** (5.1)
    Schleifencode
    $i := i + 1;$
done

Zusätzlich zum Problem des Auffindens der Quellcodezeilen kommt hier hinzu, daß die Definition eines Schleifendurchlaufs schwierig war. Im zweiten Kapitel wurde zu diesem Zweck die Funktion *inner* eingeführt, die einen Basisblock ausgibt, dessen Ausführanzahl mit der Durchlaufanzahl der Eintrittskante verglichen werden kann, um die Schleifendurchlaufanzahl zu definieren.

Für die Berechnung von *inner*(l) konnte eine Heuristik gefunden werden, die bei den verwendeten Testprogrammen korrekte Ergebnisse liefert, d. h. die die Maximallaufzeit garantiert nicht unterschätzt. In Einzelfällen kann es jedoch sein, daß der Benutzer spezifizieren muß, ob die Schleifenabbruchbedingung im Kompilat am Anfang oder am Ende der Schleife steht, damit Phobos die Iterationsbegrenzungen richtig berechnen kann. Greift der Benutzer so ein, so wird einfach die Ausführanzahl des Kopfes der Schleife begrenzt. Ist die Schleifenbedingung am Anfang, so ist diese Ausführanzahl gleich der Anzahl der Iterationen der Schleife, ist sie am Ende so ist die Ausführanzahl des Kopfes um eins größer als die Anzahl der Iterationen der Schleife.

Die Heuristik, die Phobos verwenden kann, geht einen etwas anderen Weg. Sie beschränkt *Kanten*durchlaufanzahlen statt Ausführanzahlen von Basisblöcken.

Die Heuristik (*siehe* Abbildung 5.2) sucht ausgehende Kanten des Kopfes der Schleife,

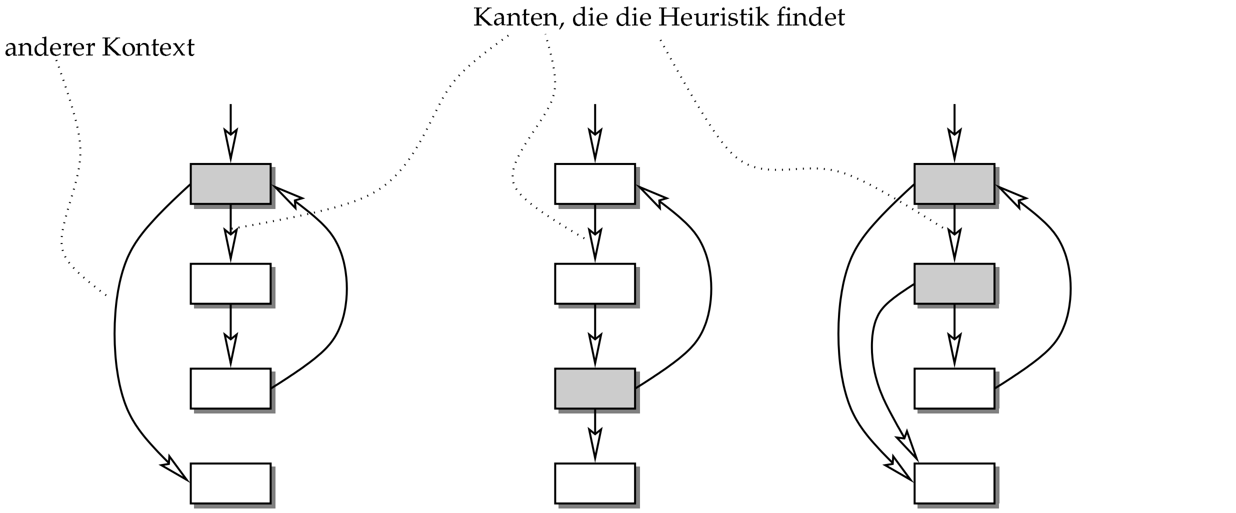


Abbildung 5.2: Heuristik, die ausgehende Kanten des Kopfes einer Schleife bestimmt, die mit der Eintrittskante der Schleife verglichen werden können, um die Anzahl der Iterationen zu beschränken. Die Schleifenabbruchbedingung ist grau unterlegt. Links ist eine Schleife mit Bedingung am Anfang, bei der die Heuristik funktioniert. In der Mitte eine Schleife mit Bedingung am Ende, bei der die Heuristik ebenfalls funktioniert. Rechts ist eine Schleife mit zweigeteilter Bedingung am Anfang, bei der die Heuristik versagt, denn es ist nicht im Maschinencode ersichtlich, ob der Block hinter der gefundenen Kante zur Abbruchbedingung oder zum Rumpf gehört.

die die Schleife nicht verlassen. Die Idee ist, daß die Kanten, die bei Schleifen mit Abbruchbedingungen am Anfang die Schleife verlassen, die einzigen Austrittskanten sind. Der Kopf der Schleife wird, wenn man die Summe der Durchlaufanzahlen der restlichen ausgehenden Kanten beschränkt, einmal öfter ausgeführt. Steht die Abbruchbedingung am Ende der Schleife, so findet diese Heuristik ebenfalls eine passende Menge von Kanten, denn die Durchlaufanzahlen aller ausgehenden Kanten des Kopfes geben in der Summe an, wie oft die Schleife ausgeführt wird.

Diese Heuristik schlägt aber z. B. fehl, wenn die Abbruchbedingung am Anfang aus mehreren Teilen besteht und der Compiler den Code so übersetzt, daß der Kopf der Schleife mehrere Austrittskanten hat:

while *A* **and** *B* **do**
  . . .
done

Das könnte übersetzt werden in

again :
 if $\neg A$ **then goto** *raus* **fi**
 if $\neg B$ **then goto** *raus* **fi**
 ...
 goto *again*;
raus :

In diesem Fall verläuft eine Kante vom ersten Basisblock der Schleife zur zweiten Bedingungsprüfung, die allerdings noch ein Teil der Abbruchbedingung ist.

Abbildung 5.2 legt nahe, vom Kopf der Schleife aus denjenigen Basisblock zu suchen, der als letzter vor einem Block mit Rücksprungkante eine Austrittskante besitzt, und diesen als Referenzblock zum Suchen der ausgehenden Kanten mit gleichem Kontext zu benutzen. Allerdings schlägt auch diese Heuristik fehl, sobald Verzweigungen (z. B. if-then-else-Strukturen) innerhalb der Schleife vor der letzten Sprungkante auftauchen. Es ist für das Analysetool nicht möglich, zu bestimmen, ob solche Verzweigungen zur Abbruchbedingung gehören oder zum Rumpf der Schleife, daher *muß* der Benutzer helfen; es gibt keine Möglichkeit der automatischen Zuordnung zu Rumpf oder Abbruchbedingung, denn die Unterscheidung existiert u. U. nur in der Hochsprache:

while $a < b$ **and** $\neg a = c$ **do** (* $a = c$ *gehört zur Abbruchbedingung* *)
 ...
done

Gegenüber

while $a < b$ **do**
 if $a = c$ **then break fi** (* $a = c$ *gehört zum Rumpf* *)
 ...
done

Keine Heuristik schützt vor Transformationen wie in (5.1). Hier findet das Analysetool nämlich die erste Iteration der Schleife nicht, da sie im Kontrollflußgraphen gar nicht mehr zur Schleife gehört. Zudem ist es extrem schwer, dieses Ausrollen allein anhand des Kompilates zu erkennen, denn der Zyklus, den man als Schleife zu identifizieren sucht, enthält die erste Iteration nicht, und es ist wahrscheinlich, daß der Compiler diese erste Iteration zusätzlich optimiert hat, da er Variablen (z. B. die Laufvariable) durch Konstanten ersetzen kann, so daß sich der erzeugte Code erheblich von dem innerhalb der Schleife unterscheiden kann.

Es ist weiterhin zu bemerken, daß die Probleme durch den Gebrauch von break-, return- oder goto-Befehlen in Schleifen vom Benutzer getragen werden sollten, denn sie zerstören z. T. die Wohlgeformtheit von Schleifen und gelten i. a. in anderen Programmiersprachen als C als schlechter Programmierstil. Da sie in C-Programmen aber recht häufig sind und bei einigen Konstruktionen, etwa bei den beiden letzten Beispielen, identische Flußgraphen erzeugen wie die normalen Schleifenkonstruktionen, wurden sie trotzdem nicht ignoriert. Doch leider konnte nicht einmal eine Heuristik gefunden werden, die die Iterationsanzahl von Schleifen ohne ihre Verwendung sicher definiert.

Die optimierten Schleifen führten auch bei dem von Thomas Ramrath implementierten Frontend (*siehe* [Ram97]), das von der Cache-Analyse benutzt wird, zu Problemen. Dieses Frontend fand nicht immer alle im Programm vorhandenen Schleifen. EEL erkannte meist mehr Schleifen, so daß in Phobos eine Möglichkeit eingebaut wurde, auch für die Schleifen korrekte Nebenbedingungen zu erzeugen, die keine Schleifenkontexte zugeordnet bekamen, d. h. in denen keine Unterscheidung zwischen erstem und weiteren Durchläufen gemacht wurde. Auf diese Weise konnte dies Problem für alle Testprogrammen gelöst werden. Ohne diese Nebenbedingungen wäre die Analyse sonst bei diesen Schleifen nicht möglich, denn das entstehende lineare Problem wäre unbeschränkt.

Ab Optimierungsgrad `-O3` wurden Prozeduren vom Compiler automatisch inline übersetzt, d. h. der Rumpf der Prozedur tauchte statt des Aufrufs der Prozedur am Aufrufspunkt auf. Das führte zu dem Problem, daß Phobos die Benutzerangaben nicht mehr richtig den Basisblöcken zuordnen konnte, denn die Debug-Informationen enthielten keine Angaben über die inline eingefügte Version der Prozedur. Deswegen wurde gcc ab `-O3` immer zusätzlich mit `-fno-inline` aufgerufen, um dieses Problem zu verhindern.

5.3 Benutzerschnittstelle und Bedienung

Es soll in diesem Abschnitt erklärt werden, wie man das Analysetool bedienen muß, um längste Programmpfade suchen zu lassen. Es werden Beispiele für die Verwendung der im vorangehenden Kapitel vorgestellten Annotationssprache gegeben.

Der grobe Ablauf einer Analyse ist in Abbildung 5.1 auf Seite 70 zu sehen.

5.3.1 Schleifen

Die wichtigste Aufgabe des Benutzers ist es, an die Schleifen und rekursiven Prozeduraufrufe des Programmes zu schreiben, wie oft diese höchstens ausgeführt werden. Damit der Benutzer nicht Gleichungen über Basisblöcken und Kanten aufstellen muß, kann er innerhalb einer Schleife einfach die obere Schranke der Ausführanzahl angeben.

```
for (int i=0; i<5; i++) {
  o+= i;                    //$ (iterations 5)
}
```

Benutzerannotationen werden in Kommentaren gesucht, die mit einem $-Zeichen beginnen.

Das Analysetool sucht nun eine Schleife in der Nähe der Quellcodezeile, in der der iterations-Befehl gefunden wurde. Der Befehl iterations ist dabei allquantifiziert über den Kontexten des Pre-Headers der Schleife. D. h. die 5 gibt nicht die Gesamtzahl der Ausführungen an, sondern besagt, daß sie in jedem Eintrittskontext der Schleife $\trianglelefteq 5$ ist. An Stelle der 5 kann ein beliebiger Lisp-Ausdruck stehen, sofern er nur die implementierten Funktionen benutzt (es stehen alle Standardfunktionen zur Verfügung). Zusätzlich zur Angabe einer oberen Schranke lassen sich auch gleichzeitig untere Schranken definieren.

Um geschachtelte Schleifen in optimierten Programmen korrekt den Benutzerannotationen zuzuordnen, entspricht die Anzahl der Parameter des iterations-Befehls der Tiefe der Schleifenschachtelung innerhalb der aktuellen Prozedur. Die Werte sind von innerer nach äußerer Schleife anzugeben. Benötigt man einen Wert nur, um die Schachtelungstiefe zu definieren, z. B. weil die Ausführanzahl der äußeren Schleifen schon an anderer Stelle definiert wurde, so kann man den entsprechenden Wert durch () ersetzen. Als Beispiel diene eine Matrixmultiplikation:

```
for(i=1; i<=5; i++) {                   //$ (iterations 5)
  for(j=1; j<=5; j++) {
    for(k=1; k<=5; k++) {
      c[i][j] += a[i][k] * b[k][j]; //$ (iterations 5 5 ())
    }
  }
}
```

Hier ist die Schleifenschachtelungstiefe 3, also muß der zweite iterations-Befehl drei Parameter haben, damit die richtigen Schleifen gefunden werden können.

Der iterations-Befehl hat deshalb so viele Parameter wie die Schleifenschachtelungstiefe, um Mißverständnisse zwischen Analysetool und Benutzer zu vermeiden. Die Compiler-Optimierungen führten bei so kompaktem Code wie im vorangehenden Beispiel oft dazu, daß die Zuordnung von Schleife zu Quelltextzeile schwierig war. Die genaue Spezifikation der Schachtelungstiefe vermeidet Zuordnungsfehler.

Wenn der Benutzer weiß, daß eine Schleifenabbruchbedingung garantiert am Ende der Schleife steht (im Kompilat), so kann er das mit dem Befehl loop-type angeben, dessen Syntax wie die von iterations ist.

loop-unknown. (Standardeinstellung). Es ist nicht bekannt, ob die Abbruchbedingung am Anfang oder am Ende steht. Phobos bleibt immer auf der sicheren Seite beim Aufstellen der Nebenbedingungen.

loop-start. Abbruchbedingung am Anfang. Der erste Basisblock der Schleife wird höchsten einmal öfter ausgeführt als der Benutzer mit der maximalen Anzahl von Iterationen angibt.

loop-end. Abbruchbedingung am Ende. Der erste Basisblock der Schleife wird höchstens so oft ausgeführt wie angegeben.

loop-heur. Benutze die Heuristik, daß die Schleife so oft ausgeführt wird, wie ausgehende Kanten des Kopfes mit gleichem Kontext wie der Kopf durchlaufen werden.

(). Keine Angabe. Der Schleifentyp für die Schleife wird nicht geändert. Dies dient wie bei iterations dazu, die loop-type-Befehle an mehreren Stellen angeben zu können, ohne jedesmal für alle geschachtelten Schleifen Werte angeben zu müssen.

5.3.2 Variablen

Es kommt vor, daß eine Schleife in unterschiedlichen Kontexten unterschiedlich oft ausgeführt wird, wie in folgendem Beispiel:

```
int fac(int i)
{
  int result=1;
  for (; i>=2; i--) {
    result*= i;            //$ (iterations (1- i))
  }
}
 ...
x= fac (5);                //$ (defparam i 5)
y= fac (10);               //$ (defparam i 10)
```

Man sieht hier, wie der Benutzer dem Analysetool mitteilen kann, wie oft die Schleife in den verschiedenen Aufrufskontexten ausgeführt wird. Der Befehl defparam sucht dabei nach einem Prozeduraufruf auf der Quelltextzeile und definiert an diesem Block eine lokale Variable, auf die alle Kontexte, die diesen Block enthalten, später zugreifen können. Im obigen Beispiel werden bei der Allquantifizierung des iterations-Befehls zwei verschiedene Werte für i in den unterschiedlichen Kontexten berechnet. Auch der defparam-Befehl ist natürlich allquantifiziert, d. h. es wird oben am Basisblock des Aufrufs in all dessen Kontexten eine Variable mit dem Wert 5 bzw. 10 definiert. Man könnte `fac` z. B. wie folgt aufrufen:

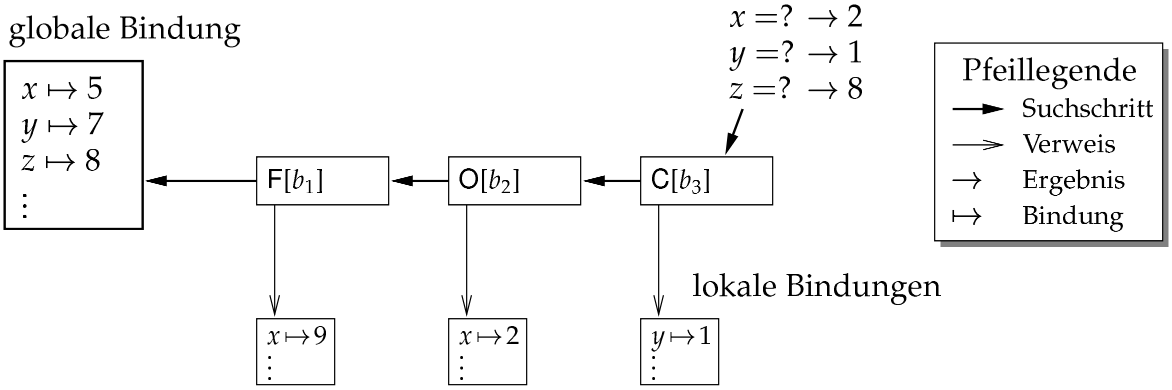


Abbildung 5.3: Darstellung des Verfahrens zum Nachschlagen von Variablenwerten. Für alle $\bar{\vartheta} \in \bar{T}$ gibt es eine lokale Bindung, in der die Variablen nachgeschlagen werden.

```
int fac2(int i, int o)
{
  return fac (i+o);       //$ (defparam i (+ i o))
}
 ...
w= fac2 (8, 10);          //$ (defparam i 8 o 10)
w= fac2 (9, 30);          //$ (defparam i 9 o 30)
```

Dabei wird der lokale Wert von `i` in `fac` am Aufruf in `fac2` erst aus den Parametern berechnet. Wie man am ersten defparam-Befehl sieht, wird das `i` im zweiten Argument in der Bindung des Aufrufers berechnet, geschrieben wird das Ergebnis jedoch in die Bindung des Aufgerufenen. Abbildung 5.3 veranschaulicht das Nachschlagen eines Variablenwertes.

Oft kommt es vor, daß am Anfang des C-Programmes Konstanten mittels `#define` definiert werden. Um diese auch in den Annotationen nicht ausschreiben zu müssen, gibt es den Befehl (defconst *variable wert*), der Werte in die globale Bindung einträgt.

5.3.3 Rekursive Prozeduraufrufe

Für rekursive Prozeduraufrufe gibt es einen dem iterations-Befehl entsprechenden funcalls-Befehl:

```
int fac (int x)
{
  if (x<=2) return x;
  return x * fac (x-1);   //$ (funcalls (exactly (max 1 (1- fac_x))))
}
```

5.3.4 Allgemeine Nebenbedingungen

Um allgemeine Nebenbedingungen aufzustellen, hat der Benutzer die Möglichkeit, Basisblöcke zu benennen (mit label und anderen ähnlichen Befehlen) und nachher in einer Nebenbedingung diese Namen als Variable für die Durchlaufanzahl zu verwenden (mit dem constraint-Befehl). Diese Methode ist aufgrund der Optimierungen des Compilers vor allem bei kurzen Codestücken sehr zerbrechlich. In den Testprogrammen mußten diese Befehle nur dort benutzt werden, wo die Zuordnung der Quellcodezeile zu einem Basisblock relativ einfach war. Eine Lösung des Problems ist wie gesagt nicht leicht; hierfür wurden ja gerade z. B. die speziellen iterations-Befehle eingeführt. Weitere Lösungsvorschläge werden in Abschnitt 5.4.1 auf Seite 81 besprochen werden.

5.3.5 Definitionsdatei

Zusätzlich zu den normalen Annotationen im Programmquelltext kann der Benutzer eine Definitionsdatei angeben, in der z. B. zusätzliche defconst-Befehle angegeben werden können.

Es können hier auch Namen von Prozeduren angegeben werden, die in der Analyse nicht berücksichtigt werden sollen (der Befehl lautet ignore). Die Ausgabe ist dann die Laufzeit ohne die Laufzeit, die in diesen Prozeduren verbracht wird. Die Angabe solcher Funktionen ist bei der Analyse von Bibliotheksprozeduren nützlich, die evtl. keine leicht zu bestimmende Laufzeit haben und daher gesondert behandelt werden sollen. Zudem kann es sein, daß der Compiler Prozeduren zur Binärdatei hinzugebunden hat, die nicht aufgerufen werden. Diese führen ohne Angabe des ignore-Befehls dazu, daß das erzeugte lineare Problem unbeschränkt ist, da keine Aufrufkanten für die Prozeduren vorhanden sind und daher keine Nebenbedingungen zur Beschränkung der Aufrufanzahl erzeugt werden. (Der Lösbarkeitsbeweis im dritten Kapitel hat auch vorausgesetzt, daß alle Prozeduren im Call-Graphen vom Hauptprogramm aus erreichbar sind).

5.3.6 Warnungen

Hat der Benutzer Angaben über Schleifenausführungen oder Rekursionstiefen nicht gemacht, so warnt das Analysetool davor. Ein Problem war im Anfangsstadium des Projektes, daß solche fehlenden Nebenbedingungen dazu führten, daß ein unlösbares lineares Problem erzeugt wurde. Dies fiel erst lp_solve auf, und man konnte kaum Rückschlüsse auf die Ursache machen (die Fehlermeldung von lp_solve beschränkt sich auf „The problem is unbounded"). Daher bemüht sich Phobos nun, schon frühzeitig zu warnen. Die Warnungen werden in der gleichen Syntax ausgegeben wie Fehler von gcc,

d. h. mit Quelltextzeile der Schleife oder Prozedur, so daß man das Tool in einem Emacs-Fenster laufen lassen kann, um komfortabel von Warnung zu Warnung zu springen und sich sofort auf der richtigen Quellcodezeile zu befinden.

Zusätzlich zu nicht eingeschränkten Schleifen und Rekursionen kann Phobos bei Bedarf das erzeugte lineare Problem noch einmal nach Variablen durchsuchen, die zwar in der Zielfunktion auftreten, für die aber keine vom Startwert abhängige Einschränkung besteht. Wird eine solche Variable gefunden, ist das Problem garantiert unbeschränkt, falls die Variable einen positiven Koeffizienten in der Zielfunktion hat.

5.4 Mögliche Verbesserungen

Es gibt Punkte, an denen das Analysetool verbessert werden kann. Im folgenden werden einige Punkte angesprochen.

5.4.1 Erweiterungen der Annotationssprache

Es wäre wünschenswert, Schleifenbegrenzungen nicht nur für den innersten Kontexte aufstellen zu können, sondern für beliebige Kontexte.

Dies ist beispielsweise sinnvoll, wenn ein Algorithmus eine Liste verwaltet, die sich während des Programmlaufs bei jedem Schritt um ein Element verlängert. Muß in jedem Schritt des Algorithmus in dieser Liste linear gesucht werden, so muß man die Suchschleife im Moment durch die Maximalanzahl n der eingefügten Elemente beschränken. Doch im Gesamtlauf des Algorithmus wäre die Anzahl der Schritte nur $\frac{n}{2}$. Dies muß im Moment mit allgemeinen Nebenbedingungen beschrieben werden, doch man könnte sich vorstellen, daß man die Schleifenbegrenzung einfach im Kontext des Startpunktes des Algorithmus auf $\frac{n}{2}$ festsetzt.

5.4.2 Zusatzbefehle

Die Bibliotheksprozeduren werden in der aktuell implementierten Analyse nicht berücksichtigt. Sie haben immer Laufzeit 0. Das führt unter Umständen zu falschen Worst-Case-Pfaden. Daher könnte ein Befehl eingebaut werden, der die Maximallaufzeit eines Bibliotheksprozeduraufrufs definiert, eventuell abhängig vom Aufrufskontext. So ein Befehl könnte etwa func-wcet heißen und als Argumente den Namen einer Prozedur und einen Term nehmen, der dann im Aufrufkontext ausgewertet wird. Die Erweiterung wäre nicht schwierig, da Phobos bereits Terme als Datentypen und Kontexte kennt.

Weiterhin wäre ein Befehl nützlich, der Fehler ähnlich wie unwind-protect abfängt und abhängig davon bedingt auswertet. Er könnte condition-case heißen und es ermöglichen, daß man spezielle Laufzeiten sowie absolute Maximallaufzeiten definieren kann, falls etwa Eingabeparameter einer Prozedur nur in manchen Kontexten bekannt sind und sonst ein Standardwert benutzt werden soll.

Es gibt natürlich viele andere ähnliche Erweiterungen. Eine vollständige Liste ist sicherlich nicht anzugeben.

5.4.3 Optimierter Code

Das Problem der optimierenden Compiler ist wohlbekannt. Vor allem im Bereich der Debugger gibt es viele Arbeiten, die sich mit dem Problem beschäftigen. Viele Ansätze, die auf einer Analyse des Kompilates aufbauen, eignen sich wenig zum Lösen des hier behandelten Problems. Ausgerollte Schleifen werden durch Rückübersetzung nicht als solche erkannt und passen deshalb nicht zum Quellcode. Läßt man den Benutzer den dekompilierten Code annotieren, so ist es für ihn schwierig, die bekannten Kontrollstrukturen seines Programmes wiederzufinden. Interessant hingegen sind die Ansätze in [TG98], die den Quelltext umschreiben, um dem Benutzer eine Vorstellung von den Compiler-Optimierungen zu geben. Die umgeschriebenen Quelltexte könnten vom Benutzer fast so leicht verstanden werden wie die Urtexte, und er könnte sehen, daß etwa Schleifen einmal weniger ausgeführt werden usw. und entsprechend korrekte Benutzerangaben in die Kommentare schreiben. Dieser Ansatz ist besser als rückübersetzte Kompilate, denn der Benutzer kann sein Programm annotieren, nachdem der Compiler strukturelle Optimierungen vorgenommen hat. Er findet sich leicht zurecht, da der modifizierte Quelltext von seinem Original abgeleitet ist und die für den Benutzer uninteressanten mikroarchitektonischen Optimierungen nicht enthält. Diese Optimierungen analysiert das Analysetool ohne Hilfe des Benutzers ohne Schwierigkeiten.

Eine Intervalanalyse zur Bestimmung der Werte von Schleifenbegrenzungen und Prozedurargumenten zu erstellen ist ebenfalls eine mögliche Verbesserung der Analyse (*siehe* [Sic97]). Sie ließe z. B. zu, daß das Analysetool in einfachen Fällen Schleifendurchlaufs- und Rekursionstiefenmaxima selbst herleiten kann. Bei vielen der Testprogramme waren oft die Anfangs- und Endwerte eines Schleifenzählers konstant. Es war beim Annotieren dann offensichtlich, daß es sehr hilfreich wäre, wenn sich der Benutzer wenigstens um solche Trivialfälle nicht mehr zu kümmern bräuchte.

Bei einigen Testprogrammen fiel besonders auf, daß die automatische Erkennung einiger Konstanten im Programm die Annotationen vereinfacht hätte. Beispielsweise wurde im Primzahltestprogramm bei verschiedenen, konstanten Zahlen geprüft, ob sie prim sind. Ein Ausschnitt aus dem Programm sieht mit Annotationen so aus:

```
int main (int, char**) {
  isprime (5);           //$ (defparam prime_i 5)
  isprime (23);          //$ (defparam prime_i 23)
  isprime (61);          //$ (defparam prime_i 61)
  isprime (79);          //$ (defparam prime_i 79)
  isprime (8849);        //$ (defparam prime_i 8849)
  isprime (811);         //$ (defparam prime_i 811)
  test (1600,19);        //$ (defparam test_x 1600      test_y 19)
  test (9000,839);       //$ (defparam test_x 9000      test_y 839)
  test (100051000,759);  //$ (defparam test_x 100051000 test_y 759)
}
```

Man sieht sehr deutlich, daß eine sehr einfache Analyse in diesem Fall alle defparam-Befehle in der `main`-Prozedur überflüssig gemacht hätte.

Annotationen, die sich schwer durch Datenflußanalysen (etwa eine Intervalanalyse) ersetzen lassen, werden immer auftauchen. Eine Dreiecksschleife ist ein solcher Fall:

```
b_1                                  //$ (label b1)
for (int i=1; i<n-1; i++) {
  for (int o=i+1; o<n; o++) {
    b_2                              //$ (label b2)
  }
}
```

Diese Schleifenart tritt z. B. in einfachen Sortieralgorithmen in dieser oder ähnlicher Form auf. Die beiden Schleifen seien l_1 und l_2. Die folgenden zusätzlichen Nebenbedingung sind $\forall\vartheta \in \tau(b_1)$ durch den Benutzer anzugeben:

$$c[b_2, \vartheta] = \frac{(n-1)\cdot n}{2}\, c[b_1, \vartheta] \tag{5.2}$$

Im letzten Beispiel ist bereits eine Benennung der Basisblöcke b_1 und b_2 vorgenommen worden. D. h. der Benutzer muß nur noch folgende Nebenbedingung hinzufügen:

```
//$ (constraint (<= b2 (* b1 (/ (* (1- var_n) var_n) 2))))
```

Sei $k = \frac{(n-1)\cdot n}{2}$. Dann stellt dies folgende Nebenbedingung auf:

$$\sum_{\vartheta\in\tau(b_2)} c[b_2, \vartheta] \leq k \cdot \sum_{\vartheta\in\tau(b_1)} c[b_1, \vartheta] \tag{5.3}$$

Im Moment gibt es noch keine Möglichkeit, diese Bedingung für einzelne Kontexte aufzustellen. Obige Angabe summiert also global alle Kontexte von b_1 und b_2 und erwartet, daß `var_n` global definiert ist. Hier sind Verbesserungen möglich.

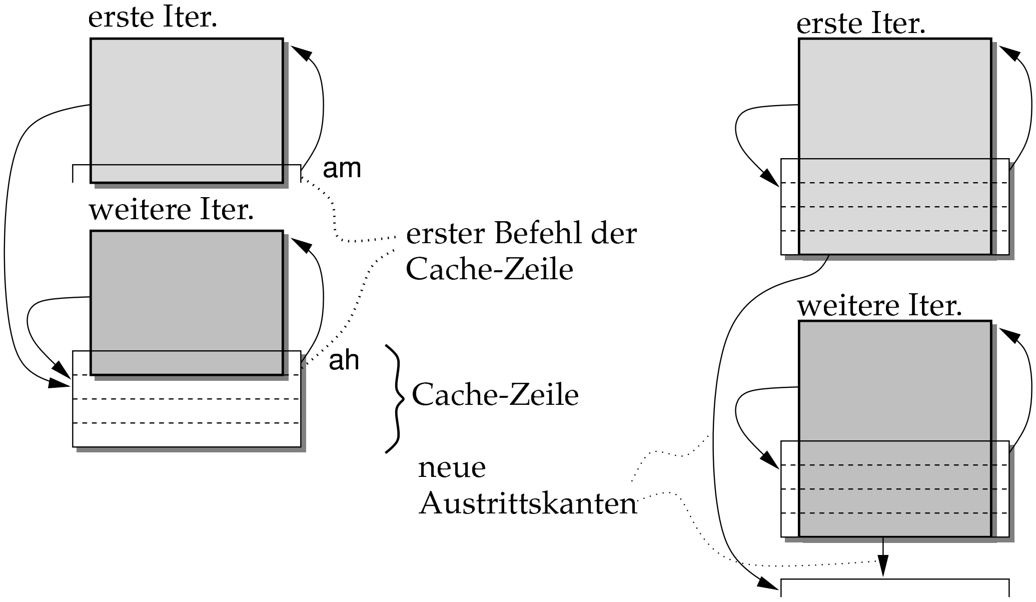


Abbildung 5.4: Eine ausgerollte Schleife mit einer Austrittskante, die mitten in eine Cache-Zeile zeigt, und wie man diese Schleife transformieren kann, um bessere Cache-Analysen in dieser Cache-Zeile zu erhalten

5.4.4 Verzögerung des Schleifenaustrittes

In Abschnitt 3.7.2 auf Seite 51 wurde eine Schleife vorgestellt, in der ein Block jedesmal vor Verlassen der Schleife ausgeführt wird. Man könnte natürlich argumentieren, daß dieser Basisblock *B* damit gar nicht mehr zur Schleife gehört, auch wenn er in der höheren Programmiersprache aufgrund von Sichtbereichen von Variablen innerhalb des Schleifenskopus' steht. Diese Betrachtungsweise ändert nichts an der Richtigkeit der Analyse und kann sogar zu einer Verbesserung der Analyse führen.

Der Cache moderner Prozessoren besteht aus Zeilen der Länge n, wobei n für gewöhnlich größer ist als die Länge eines Maschinenbefehls. Dadurch lädt der Prozessor bei einem Cache-Miss mehrere aufeinanderfolgende Befehle auf einmal in den Cache und hat damit nach Ausführung des ersten Befehls für die folgenden Befehle dieser Cache-Zeile Cache-Hits. Verläßt der Kontrollfluß wie in Abbildung 5.4 nun eine Schleife und springt mitten in eine Cache-Zeile z hinter der Schleife, so kann es sein, daß die Ausführung der Schleife diese Zeile schon in den Cache geladen hat. Es kann sein, daß die Zeile ab der zweiten Iteration im Cache steht, vorher aber nicht. In diesem Fall wird die Instruktion hinter der Schleife als nc bewertet werden müssen oder zumindest als pers, denn sie wird ja nicht mehr im Schleifenkontext analysiert, und daher ist keine Unterscheidung möglich.

Es liegt daher in diesem Falle nahe, die Instruktionen, die in der Cache-Zeile z liegen, noch mit zur Schleife hinzuzunehmen und nach erster und weiteren Iterationen zu unterscheiden. In der Abbildung ist zu sehen, wie man diese Ausdehnung vornehmen kann, und wo u. U. neue Kanten eingeführt werden müssen. Die Transformation ist ohne großen Aufwand (d. h. ohne kompliziertes Umstrukturieren des Kontrollflußgraphen) nur bei Schleifenaustritten möglich, denen eine Stück Code ohne Verzweigungen folgt.

Allgemein kann man sich sicherlich verschiedene Ansätze überlegen, die Überlegungen auch auf andere Mikroarchitekturteile neben dem Cache auszudehnen.

Sechstes Kapitel

Ergebnisse

Um das Analysetool zu testen, mußten Vergleiche zwischen den vorhergesagten Laufzeiten und den tatsächlichen Laufzeiten gezogen werden. Da eine idealisierte Hardware angenommen wurde, konnten die Ergebnisse nicht direkt mit der Laufzeit auf einer SPARC-Maschine verglichen werden.

Um vergleichen zu können, wurde der Programm-Tracer Qpt2 (Quick program Profiler and Tracer) verwendet, der zu WARTS gehört. Dieser erzeugt aus Kompilaten Programme, die das gleiche tun, aber zusätzlich mitprotokollieren, was die Maschine tut. Man erhält dadurch eine Datei, die den Programmlauf genau beschreibt. Mit Hilfe dieser Datei können die Ergebnisse mit denen der Vorhersage verglichen werden. Man muß nur sicherstellen, daß die Eingabe eine *Worst-Case-Eingabe* ist, d. h. daß das Programm bei dieser Eingabe am längsten läuft. Abbildung 6.1 zeigt schematisch den Ablauf eines Traces.

Nachdem mit Qpt2 die Maximallaufzeit eines Testprogrammes herausgefunden worden war, wurde das Programm mit dem Analysetool untersucht. Danach lagen zwei Dateien vor, in denen die tatsächliche und die vorhergesagte Ausführanzahl für jeden Basisblock in all seinen Kontexten aufgelistet waren. Aus der Datei, die lp_solve ausgegeben hatte, wurde der Zielfunktionswert gelesen und mit der Anzahl der Taktzyklen verglichen, die der Tracer berechnet hatte. Der Quotient sollte, falls Phobos und lp_solve korrekt arbeiten, immer $\geqq 1$ sein.

Abbildung 6.2 auf Seite 89 zeigt schematisch den Ablauf der Tests.

In diesem Kapitel wird auf das gesamte Analysetool auch mit dem Namen des Haupt-

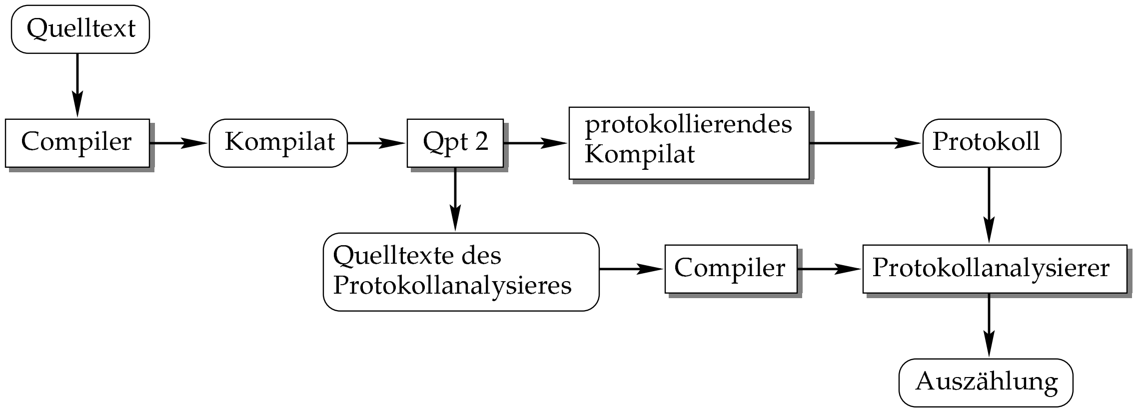


Abbildung 6.1: Erzeugung der Auszählung der Ausführanzahlen der Befehle eines Programmes mit Qpt2.

moduls „Phobos" Bezug genommen werden. Diese Namensgleichheit dürfte wenig verwirrend sein, da Phobos als Hauptmodul Träger der Analyse ist (davon abgesehen, daß lp_solve das eigentliche *Ergebnis* berechnet, hat Phobos doch das Problem formuliert).

6.1 Durchführung der Tests

Es wurden die Testprogramme in Abbildung 6.3 auf Seite 90 benutzt, um den Vergleich zwischen vorhergesagter und gemessener Laufzeit (in Zyklen auf der idealisierten Maschine) durchzuführen.

Alle Programme wurden mit gcc 2.7.2 unter Solaris 4.1.4 übersetzt. Danach wurde ein mit Qpt2 erzeugter Tracer benutzt, um die Laufzeit zu bestimmen. Die Programme wurden mit einer Worst-Case-Eingabe ausgeführt. Verglichen wurde der Wert der Zielfunktion der optimalen Lösung des von Phobos erzeugten ILPs mit der Anzahl der Taktzyklen, die der Programm-Trace ergeben hatte.

In einigen Fällen sollte die Genauigkeit von Phobos *ohne* die Cache-Analyse eine Vergleichsmöglichkeit zwischen Trace und Vorhersage überprüft werden. Zu diesem Zweck wurde in Callisto die Möglichkeit eingebaut, die Cache-Vorhersage auf den Trace zu übertragen. Mit dieser Methode konnten Fehler von Phobos ohne Einfluß der Cache-Analyse betrachtet werden.

Die Testreihen wurden mit verschiedenen Optimierungseinstellungen des Compilers durchgeführt, um den Einfluß der Compiler-Optimierung zu untersuchen. Dabei wurde die inline-Übersetzung mittels `-fno-inline` aus den vorne geschilderten Gründen unterbunden. Ab Optimierungsstufe 2 änderte sich das Kompilat nur noch in Einzelfällen marginal, so daß die Ergebnisse nur bis `-O2` vorgestellt werden.

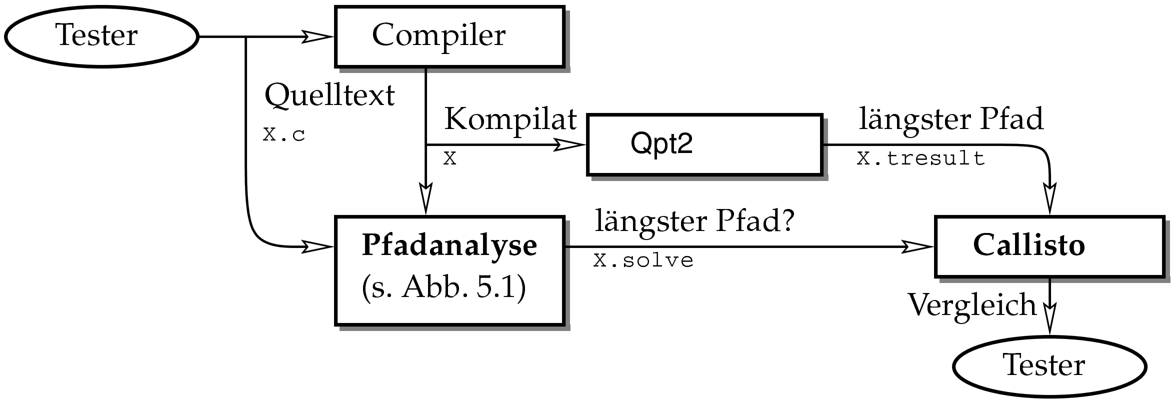


Abbildung 6.2: Darstellung eines Testlaufes eines Programmes `X`.

Neben den verschiedenen Einstellungen zur Optimierung wurde noch überprüft, inwieweit sich die pers-Analyse lohnt und wie stark die Analysezeit darunter leidet. Es wurden also alle Messungen jeweils mit ein- und ausgeschalteter pers-Analyse durchgeführt.

Die Cache-Analyse wurde bereits genau in [Fer97b] getestet. Daher wurde nur eine Cache-Einstellung verwendet. Diese simulierte einen 1 kByte großen Cache mit vierfacher Assoziativität und einer Zeilenlänge von 16 Byte. Die Einstellung wurde deswegen so klein gewählt, damit die relativ kleinen Testprogramme nicht vollständig in den Cache passen.

Für Schleifen mit Abbruchbedingung am Anfang wurde überprüft, ob der Compiler beim Optimieren die Schleifenbedingung ans Ende verschoben hatte. In dem Fall wurde mit dem loop-type-Befehl der Schleifentyp umgestellt, um eine genauere Analyse zu ermöglichen.

6.1.1 Genauigkeit der Analyse

Die kürzeren der hier besprochenen Testprogramme sind in Anhang A abgedruckt. Es wird nun untersucht, wie genau die Analysen sind. Die Ergebnisse sind in Abbildung 6.4 auf Seite 91 zu sehen.

Zunächst zu den Ergebnissen ohne Optimierung. Man kann deutlich sehen, daß die pers-Analyse eine Verbesserung darstellt. Dies gilt vor allem für die Programmen, deren Laufzeit im wesentlichen durch eine Schleife verursacht wird, in der die Cache-Analyse einige Maschinenbefehle als pers klassifiziert hat. Ein beeindruckendes Beispiel ist `prime`. Dort hatte die Cache-Analyse nur zwei Maschinenbefehle als pers klassifiziert (in 8 von 9 Kontexten). Da diese aber in der Hauptschleife liegen, führt eine Bewertung als am zu einer riesigen Überschätzung der Laufzeit. Ähnlich verhält es sich bei `sort`, dort sind die als pers klassifizierten Befehle allerdings zahlreicher.

Name	Beschreibung	Quell-zeilen	Bytes bei `-O2`
`fac`	Berechnung der Fakultät einiger Werte (rekursiv)	16	96
`prime`	Überprüfung, ob einige Zahlen prim sind	34	228
`sort`	Bubble-Sort-Algorithmus im Feld mit 500 Zahlen*)	123	236
`matmul`	5×5-Matrixmultiplikation	46	240
`circle`	Zeichnen eines Kreises	200	1240
`jfdctint`	diskrete Kosinustransformation für JPEG-Bilder	392	1476
`stats`	Summe, Durchschnitt, Varianz und Standardabweichung für zwei Felder	186	1512
`des`	Datenverschlüsselung nach dem DES-Standard*)	287	1944

*) mit Worst-Case-Eingabe

Abbildung 6.3: Die verwendeten Testprogramme.

Eine zusätzliche Überschätzung bei `prime` entstand dadurch, daß ein Pfad vorhergesagt wurde, der nicht auftreten kann: Es wurde vorausgesagt, daß der Rumpf der Schleife genau wie die Abbruchbedingung der Schleife einmal öfter ausgeführt wird, als der Benutzer bestimmt hatte, und daß dann bei dieser letzten Iteration die Schleife wegen Teilbarkeit der Zahl verlassen wird. Dieser Fall kann deswegen nicht auftreten, weil der Rumpf der Schleife eben nicht genausooft ausgeführt werden kann wie der Kopf. Da die Abbruchbedingung im Kompilat aber bei allen Optimierungsstufen am Anfang und nicht am Ende stand, konnte keine engere Schranke angegeben werden. Das Problem war hier also, daß die Ausdehnung der Abbruchbedingung überschätzt wurde.

Bei der Betrachtung der Analyse von `matmul` mag es verwundern, daß die Abweichung nicht 0,00% war, obwohl doch in dem Programm keine einzige Bedingung überprüft wird. Es stellt sich die Frage, ob die als pers analysierten Befehle eigentlich ahs und dies zur Überbewertung führt. Tatsächlich ist die Laufzeit von `matmul` so kurz, daß die erste Ausführung der pers-Befehle stark ins Gewicht fiel. Der Tracer errechnete 8 707 Maschinenzyklen, die Vorhersage erwartete 9 166. Der genaue Vergleich des vom Tracer gefundenen Worst-Case-Pfades mit dem von Phobos vorhergesagten Worst-Case-Pfad ergab keinerlei Abweichung. Die Ausführanzahlen waren für alle Basisblöcke in allen Kontexten identisch.

Dasselbe gilt für das `jfdctint`-Programm. Es gibt keinerlei bedingt ausgeführte Befehle im Programm außer den Schleifen. Daher sollte man eine 100,00%ige Vorhersage erwarten. Vergleicht man die Durchlaufanzahlen aus Trace und Vorhersage für jeden Basisblock, so ergibt sich auch hier keinerlei Abweichung.

Die Überschätzung um 18% in `circle` ist ebenfalls leicht erklärt. Dieses Programm enthält eine Bedingungsabfrage in der Hauptschleife, deren `then`-Teil viermal ausgeführt wurde, der `else`-Teil nur dreimal. Da Phobos keine Information darüber hatte, wurde jedesmal der Worst-Case gewählt; in diesem Fall wurde der `else`-Teil siebenmal

	ohne pers			mit pers		
Programm	-O0	-O1	-O2	-O0	-O1	-O2
fac	85,41%	0,39%	0,39%	0,34%	0,14%	0,14%
prime	256,26%	224,59%	224,59%	0,76%	0,92%	0,92%
sort	261,09%	47,76%	179,74%	0,00%	0,00%	0,00%
matmul	35,25%	0,00%	0,48%	5,48%	0,00%	0,48%
circle	18,69%	6,49%	3,71%	17,64%	2,51%	1,73%
jfdctint	0,04%	0,00%	0,00%	0,04%	0,00%	0,00%
stats	0,08%	0,10%	0,10%	0,08%	0,10%	0,10%
des	52,32%	20,36%	19,56%	52,23%	4,38%	3,41%

Abbildung 6.4: Die Überschätzung tabellarisch. Zur Berechnung wurde der Zielfunktionswert, also sie vorhergesagte Anzahl Taktzyklen, durch die tatsächlich durch den Tracer gefundene dividiert, eins subtrahiert und mit hundert multipliziert. Alle Angaben sind prozentuale Überschätzungen der tatsächlichen Laufzeit.

ausgeführt.

In `stats` tauchen zwar einige als pers kategorisierte Befehle auf, doch werden diese nur einmal ausgeführt, daher sind die Ergebnisse der Analysen mit und ohne pers-Betrachtung identisch.

Die Analyse von `des` zeigt eine deutliche Überschätzung der Laufzeit um über 50% mit und ohne Berücksichtigung der pers-Analyse. Eine Untersuchung der Cache-Analyse ergibt, daß einige nc-Analysen auftreten, die offenbar beim Trace zu Cache-Hits geführt haben. Um zu sehen, wieviel Schuld Phobos trifft und ob ein Großteil der Verschlechterung tatsächlich auf diese nc-Analysen zurückzuführen ist, wurde der in Abschnitt 6.1 auf Seite 88 erwähnte Vergleich des Traces ohne Verwendung der Modellierung des Cache-Verhaltens gezogen. Dabei ergab sich eine Genauigkeit von 103,75%.

6.1.1.1 Compiler-Optimierung

Vergleicht man die Optimierungsstufen miteinander, so treten ebenfalls interessante Dinge zutage. Oft wird die Analyse mit eingeschalteter Optimierung prozentual *besser*. Das ist zunächst verwunderlich, denn das entspricht nicht dem, was erwartet wurde. Erwartet worden wäre eher das Gegenteil, denn die Optimierung warf in anderen Bereichen der Verwirklichung große Probleme auf.

Die Erklärung für das unerwartete Verhalten liegt in der Cache-Analyse. Durch die Optimierung werden die Programme kleiner, wodurch die Anzahl der Cache-Konflikte, also die Anzahl der Programmzeilen, die auf die gleiche Cache-Zeile abgebildet werden, sinkt. Dadurch wiederum wurde die Cache-Analyse teilweise wesentlich genauer, wie etwa bei `des` und `circle`.

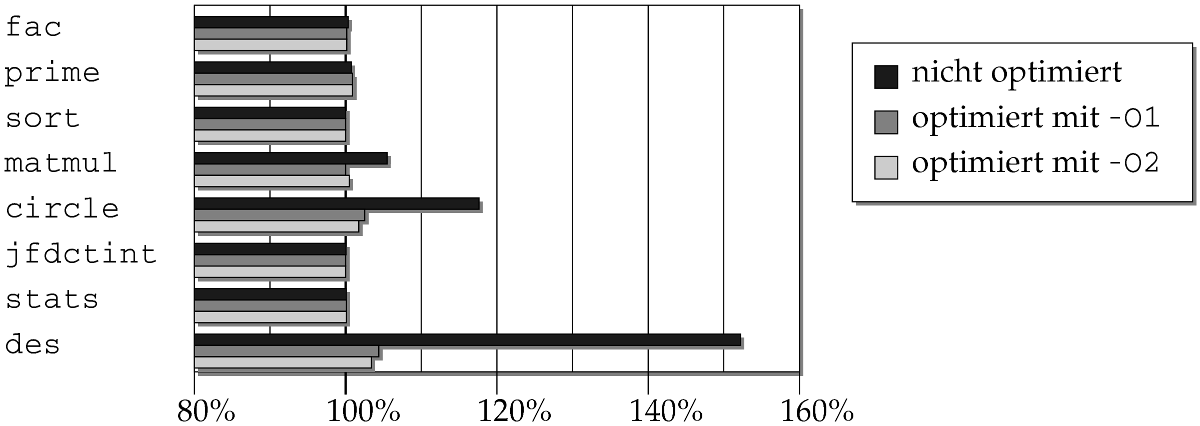


Abbildung 6.5: Grafischer Vergleich der Genauigkeit der Analysen bei verschiedenen Optimierungsstufen und bei besten Einstellungen für die Erzeugung des linearen Problems.

Insgesamt sind alle Überschätzungen sowohl bei ein- wie auch bei ausgeschalteter Optimierung als gering zu bewerten.

6.1.2 Problemgrößen und Laufzeiten

Die Laufzeit von Phobos für die Erzeugung der linearen Probleme ist bei allen Testprogrammen sehr klein. Sie liegt unter 10 Sekunden auf einem i586-Prozessor mit 133 MHz Taktrate. Auf dem gleichen Prozessor konnten alle vereinfachten linearen Probleme in unter einer Sekunde gelöst werden. Ähnliche Laufzeiten ergeben sich auf einem SUN-SuperSPARC-Rechner mit 168 MHz.

Abbildung 6.6 und Abbildung 6.7 zeigen, wie groß die linearen Probleme waren und wie lange lp_solve benötigte, um sie zu lösen. Es werden die Lösezeiten jeweils mit und ohne Betrachtung der pers-Analyse angegeben, sowohl mit wie ohne Vereinfachung des entstandenen Problems und für zwei der Optimierungsstufen. Man sieht, daß die ILPs durch die Vereinfachung stark verkleinert werden konnten, teilweise um Faktor 10.

Vergleicht man die Problemgrößen mit den Laufzeiten des LP-Lösers, so wird deutlich, daß die exponentielle Worst-Case-Laufzeit des LP-Lösers nicht erreicht wird.

Als Test wurden auch die relaxierten Probleme aller aufgestellten ILPs gelöst, und es ergaben sich nur einmal geringfügige Abweichungen von den in Abbildung 6.4 vorgestellten Genauigkeiten beim Testprogramm `des`. Auch die Lösezeit war bei keinem der Testprogramme signifikant kürzer (maximal 10%). Dies liegt daran, daß es wenige benutzerdefinierte Nebenbedingungen gibt und diese die Struktur des Problems nur wenig verändern.

Der Benutzer kann die Lösezeit aber leicht erhöhen. Wird etwa die Durchlaufanzahl einer Schleife mit Funktionen ausgerechnet, die keine ganzzahligen Ergebnisse haben

mit pers?	nein		ja		nein		ja	
vereinfacht?	ja		ja		nein		nein	
`fac`	32	*0,02s*	51	*0,03s*	333	*0,42s*	352	*0,48s*
`prime`	103	*0,08s*	213	*0,25s*	928	*2,90s*	1038	*3,91s*
`sort`	25	*0,01s*	43	*0,02s*	260	*0,28s*	278	*0,34s*
`matmul`	94	*0,05s*	110	*0,07s*	493	*0,98s*	509	*1,04s*
`circle`	25	*0,01s*	30	*0,01s*	277	*0,27s*	282	*0,28s*
`jfdctint`	37	*0,01s*	38	*0,02s*	222	*0,19s*	223	*0,20s*
`stats`	75	*0,04s*	88	*0,05s*	755	*1,88s*	768	*1,95s*
`des`	373	*0,63s*	408	*0,89s*	2100	*23,90s*	2135	*25,69s*

Abbildung 6.6: Die Problemgrößen der erzeugten linearen Probleme bei Optimierungsstufe `-O0` in Anzahl der Nebenbedingungen und wie hoch die Lösezeiten waren. Die Problemgrößen und Lösezeiten nehmen v.l.n.r. zu. Alle Zeiten wurden auf einem i586-Prozessor mit 133 MHz Taktfrequenz gemessen.

mit pers?	nein		ja		nein		ja	
vereinfacht?	ja		ja		nein		nein	
`fac`	32	*0,02s*	37	*0,02s*	275	*0,29s*	280	*0,30s*
`prime`	123	*0,12s*	193	*0,25s*	728	*2,00s*	798	*2,73s*
`sort`	67	*0,03s*	79	*0,04s*	235	*0,22s*	247	*0,25s*
`matmul`	95	*0,05s*	96	*0,05s*	313	*0,37s*	314	*0,36s*
`circle`	26	*0,01s*	30	*0,01s*	258	*0,24s*	262	*0,24s*
`jfdctint`	38	*0,01s*	38	*0,01s*	150	*0,08s*	150	*0,08s*
`stats`	79	*0,04s*	79	*0,04s*	565	*1,05s*	565	*1,05s*
`des`	378	*0,56s*	419	*0,72s*	1658	*15,15s*	1699	*17,0s*

Abbildung 6.7: Die Problemgrößen der erzeugten linearen Probleme bei Optimierungsstufe `-O2` in Anzahl der Nebenbedingungen und wie hoch die Lösezeiten waren. Die Problemgrößen und Lösezeiten nehmen v.l.n.r. zu. Alle Zeiten wurden auf einem i586-Prozessor mit 133 MHz Taktfrequenz gemessen.

	Zeit	Speicher
`fac`	0,42s	415 kB
`prime`	1,13s	951 kB
`sort`	0,78s	405 kB
`matmul`	0,82s	667 kB
`circle`	1,10s	563 kB
`jfdctint`	4,57s	714 kB
`stats`	2,02s	901 kB
`des`	4,70s	2 486 kB

Abbildung 6.8: Laufzeit und Speicherbedarf von Phobos für die Programme bei `-O0` und allen Optimierungen.

müssen (wie etwa im Beispiel des Primzahlprogrammes, das $O(\sqrt{x})$ Laufzeit hat), so ist darauf zu achten, daß der Wert zum Schluß gerundet wird. Sonst tauchen nichtganzzahlige Schranken im Problem auf, die die Lösedauer erhöhen können. Bei `prime` verdoppelt sich dadurch die Lösezeit.

Weiterhin hat der Benutzer mit dem constraint-Befehl die Möglichkeit, die gute Löseeigenschaft zu zerstören, denn es können ja beliebige Abhängigkeiten definiert werden. Selbst harmlos aussehende Nebenbedingungen können extreme Analysezeitverschlechterungen ergeben. In den Testprogrammen ist dieser Fall nicht eingetreten.

6.2 Bewertung und Zusammenfassung

Die obigen Ergebnisse zeigen, daß die implementierte Pfadanalyse genaue Ergebnisse liefert. Die Cache-Analyse konnte auf natürliche Weise integriert werden und läßt Erweiterungen für andere Mikroarchitektursimulationen zu. Wie man sehen kann, sind die Laufzeiten bei den Testprogrammen gut; nur wenige Sekunden Analysedauer, um eine garantiert korrekte und dazu genaue Maximallaufzeit eines Programmes zu berechnen.

Es hat sich gezeigt, daß die automatisch erzeugten Nebenbedingungen ein ILP ergeben, das nicht viel mehr Zeit zum Lösen benötigt als das relaxierte Problem.

Es gibt einige Punkte, die als Schwachstelle des Systems gelten können. Zum einen wächst die Analysedauer in der Schachtelungstiefe von rekursiven Prozeduren und Schleifen exponentiell, denn die Anzahl der Kontexte wird mit jeder neuen Schachtelung verdoppelt. Dies liegt an der VIVU-Analyse. In der Praxis gibt es hoffentlich wenige Probleme, die größere Schleifenschachtelungstiefen als 5 haben, so daß dies Problem nicht allzusehr ins Gewicht fallen dürfte. Eine mögliche Lösung für sehr große Schachtelungstiefen wäre die Begrenzung der Länge der VIVU-Kontexte.

Zum anderen ist es in komplexen Programmen z. T. schwierig, die Nebenbedingungen so zu formulieren, daß die Schranken eng ausfallen. Der Benutzer muß seine Programme sehr gut kennen, damit die Pfadanalyse gute Ergebnisse liefert. Es ist zu erwarten, daß auch die Integration einer Analyse der Variablenwerte nichts an diesem Umstand ändert, denn oft gibt es nichttriviale Abhängigkeiten im Programm.

Die implementierte Annotationssprache versucht, dem Benutzer die Eingabe der Nebenbedingungen so komfortabel wie möglich zu machen, so daß alle Testprogramme bei genauer Kenntnis in wenigen Stunden so beschrieben werden konnten, daß obige gute Ergebnisse herauskamen.

Die implementierten Programme, die annotierten Testprogramme und die daraus erzeugten ILPs sind frei erhältlich unter `http://www.cs.uni-sb.de/~theiling/diplom/`.

Siebtes Kapitel

Vergleich und Ausblick

In dieser Arbeit wurde gezeigt, daß die Kombination von abstrakter Interpretation und linearem Programmieren sich sehr gut zur Worst-Case-Pfadsuche eignet. Die entstehenden linearen Probleme beschreiben dann ausschließlich den Kontrollfluß, und es gibt keinerlei Nebenbedingungen für mikroarchitektonische Eigenschaften. Aus den in [LMW96] vorgestellten Testergebnissen geht hervor, daß die Modellierung von mehrfach assoziativen Caches dazu führt, daß die linearen Probleme sehr groß werden. Diese Berechnung ist in dem hier vorgestellten Ansatz ausgelagert. Die Cache-Analyse mittels abstrakter Interpretation aus [Fer97b] hingegen hat sich für beliebige Cache-Geometrien als nutzbar erwiesen. Die in [LMW96] aufgetretenen Probleme der Integration der Cache-Analyse ins ILP wurden erfolgreich umgangen. Insgesamt ist die Analyse ebensogut wie die in [LMW95a, LMW96] vorgestellte und hat günstigere Laufzeiten bei hoher Cache-Assoziativität. Obwohl die dort verwendeten Cache-Konfliktgraphen eine genauere Modellierung des Cache ermöglichen, sind die Ergebnisse trotzdem nicht besser und die Laufzeiten der Analyse höher.

Wie in [LMW95a, LMW96] sind die hier vorgestellten Nebenbedingungen von Netzwerkflußproblemen abgeleitet und haben daher eine gute Lösezeit. Wie auch dort können durch beliebige Benutzerannotation die Laufzeiten stark zunehmen.

Ottoson und Sjödin stellen in [OS97] einen Ansatz vor, der die WCET mittels nichtlinearer Probleme approximiert. Der Ansatz kann ebenso wie der aus [LMW96] Caches behandeln und eignet sich ebenso zum Modellieren anderer Mikroarchitekturteile. Das Hauptproblem des Ansatzes ist die Komplexität der Analyse. Die benutzte Prolog-Constraint-Erweiterung versucht durch extensive Suche das Problem möglichst gut zu approximieren. Aus [OS97] geht hervor, daß sich der Ansatz noch nicht für die Praxis eignet.

Lim et al. beschreiben in [LBJ+95], wie eine detaillierte Analyse das Pipeline-Verhalten, den Daten- und Befehls-Cache simulieren kann. Es werden leider nur drei Testprogramme vorgestellt und keine Vergleichswerte für die Analysezeit genannt. Daher sind die Ergebnisse nicht mit denen dieser Arbeit vergleichbar. Zudem wird das Cache-Verhalten für einzelne potentielle Worst-Case-Pfade simuliert und nicht wie bei der abstrakten Interpretation über allen Pfaden abstrahiert. Das hier beschriebene Verfahren basiert im Gegensatz dazu die Worst-Case-Pfadsuche auf den Ergebnissen der Cache-Analyse. Die Modellierung von Pipeline- und Daten-Cache-Verhalten fehlt dieser Arbeit und der in [LMW96] beschriebenen. Für den hier vorgestellten Ansatz ist eine Erweiterung auf Daten-Caches möglich, indem zwei oder mehr Cache-Vorhersagedateien gelesen werden. Alternativ wäre vorstellbar, daß die Pipeline-Analyse und die Cache-Analysen ihre Ergebnisse in maximalen Taktanzahlen ausgeben und daß Phobos diese liest. Eine Pipeline-Analyse wird am Lehrstuhl Wilhelm gerade von Jörn Schneider vorbereitet (*siehe* [Sch98]).

Auch Puschner und Koza stellen in [PK89] eine Methode zur Pfadsuche vor, die allerdings der modernen Mikroarchitektur mit Cache und Pipelines keine Berücksichtigung schenkt. Daher sind die Ergebnisse nicht vergleichbar. Interessant ist aber die Entscheidung, speziell für die Analyse Sprachkonstrukte einzuführen, um bei möglichen Falschangaben des Benutzers diese zur Laufzeit zu erkennen und Ausnahmebehandlungen zu ermöglichen. Diese können dann dazu benutzt werden, direkt und lokal auf die Überschreitung zu reagieren, statt dem Echtzeitsystem die Feststellung der Überschreitung der Gesamtlaufzeit zu überlassen. Auch hierzu sind also Erweiterungen des Compilers vonnöten.

Ein Ziel dieser Arbeit war die Verwendung optimierender Compiler. Es wurde gezeigt, daß es möglich ist, diese bis zu einem bestimmten Grad einzusetzen und dabei trotzdem keinerlei Beschränkungen der Vorhersagegenauigkeit hinnehmen zu müssen. Viele der entstehenden Probleme bei optimierten Kompilaten konnten u. a. durch die Erweiterung des Sprachumfangs der Annotationssprache gelöst werden. Andere Probleme bleiben bestehen und bedürfen idealerweise einer Schnittstelle zum Compiler. Vor allem im Bereich der Debugger hat man ähnliche Probleme im Abbilden des Kompilates auf die Sicht des Benutzers, der nur den Quelltext kennt (*siehe* [vDKT93, BHS92, CG92, ATG96, BB94]). Dabei wird für gewöhnlich das Kompilat rückübersetzt nach `C`, um dem Benutzer die Optimierungen vor Augen zu führen (*siehe* [JR94]). Solche rückübersetzten Kompilate eignen sich auch für Phobos, denn sie könnten dem Benutzer als Annotationsgrundlage dienen. Leider weichen die Rückübersetzungen meist sehr stark vom Quelltext ab, so daß es dem Benutzer schwerfällt, die Strukturen wiederzuerkennen. Ein interessanter anderer Ansatz ist der aus [TG98], bei dem der Compiler selbst den Quelltext bis zur Ausgabe des Kompilates durchreicht und die vorgenommenen strukturellen Änderungen dort sichtbar macht. Dadurch ist eine gute Deckungsgleichheit mit dem Quelltext gegeben, so daß sich der Benutzer in diesem zusätzlich ausgegebenen veränderten Quelltext gut zurechtfindet. Ein weiterer Vorteil ist hier die Auskopplung der Optimierungen auf Maschinenbefehlsebene, die den Benutzer auch beim Annotieren mit Phobos nicht interessieren müssen, denn um die Modellierung kümmert sich die Mikroarchitekturanalyse. Daher wäre dieser Ansatz gut in Phobos integrierbar.

Es hat sich weiter gezeigt, daß die Annotationen an vielen Stellen trivial sind, und daß eine Intervalanalyse (*siehe* [Sic97]) Abhilfe schaffen könnte. Auch diese wäre gut in Phobos integrierbar.

Anhang A
Testprogramme

In diesem Anhang werden Auszüge der Testprogramme aufgelistet. Es ist nicht immer das ganze Programm angegeben, sondern oft nur relevante Teile.

Die in den Testprogrammen vorkommende Variable `opt` wird bei optimierten Programmen auf `loop-end` vordefiniert, bei nichtoptimierten auf `loop-start`, da dies ein typischer Unterschied war.

Manches hätte auch kürzer formuliert werden können; einige Befehle dienen dazu zu testen, ob Phobos die Annotationen richtig interpretiert.

A.1 Fakultätsberechnung

Um Phobos für rekursive Prozeduren zu testen, wurde eine einfache rekursive Fakultätsfunktion implementiert.

```
int fac (int n) {
  if (n<=2) return n;
  else
    return n*fac(n-1);   //$ (funcalls (exactly (max 1 (1- n))))
}

main () {
  fac (10);              //$ (defparam n 10)
```

```
  fac (34);              //$ (defparam n 34)
  fac (55);              //$ (defparam n 55)
  fac (2034);            //$ (defparam n 2034)
}
```

A.2 Primzahltest

Ein einfacher Algorithmus, der überprüft, ob eine Zahl prim ist. Die Laufzeit ist $O(\sqrt{n})$, falls n die zu testende Zahl ist. Zusätzlich wurden die lokalen Bindungen von Phobos getestet, daher die Zusatzprozedur `test`.

```
bool isprime (int i) {
  if (i<2) return false;
  if (i==2) return true;
  if ((i & 1)==0) return false;
  for (int o=3; o*o<=i; o+=2) {
    if (i % o == 0)
      return false;
       //$ (iterations (floor (/ (1- (floor (sqrt prime_i))) 2))) (loop-type loop-start)
  }
  return true;
}

void test (int x, int y) { //$ (defterm thesum (+ test_x test_y))
  isprime (x + y);           //$ (deflocal prime_i thesum)
}

int main (int, char**) {
  isprime (5);               //$ (deflocal prime_i 5)
  isprime (23);              //$ (deflocal prime_i 23)
  isprime (61);              //$ (deflocal prime_i 61)
  isprime (79);              //$ (deflocal prime_i 79)
  isprime (8849);            //$ (deflocal prime_i 8849)
  isprime (811);             //$ (deflocal prime_i 811)
  test  (1600,19);           //$ (deflocal test_x 1600 test_y 19)
  test  (9000,839);          //$ (deflocal test_x 9000 test_y 839)
  test  (100051000,759);     //$ (deflocal test_x 100051000 test_y 759)
}
```

A.3 Bubble-Sort-Implementierung

Dieses Testprogramm wurde der cinderella-Testsuite entnommen. Es implementiert einen einfachen Sortieralgorithmus.

```
#define NUMELEMS 500
#define MAXDIM   (NUMELEMS+1)
#define FALSE 0
#define TRUE 1

//$ (defconst NUMELEMS 500) (defconst NUMELEMS1 (1- NUMELEMS))
//$ (defconst MAXDIM   (1+ NUMELEMS))

int Array[MAXDIM];

void Exchange(int *a, int *b) {
   int tmp;
   tmp=*a; *a=*b; *b=tmp;
}

void sort() {
  int Sorted;
  int Temp, LastIndex, Index, i;

  for (i=1; i<=NUMELEMS-1; i++) { //$ (label aussen)
    Sorted= TRUE;
    for (Index=1; Index<=NUMELEMS-1; Index ++) {
                                  //$ (iterations NUMELEMS1 NUMELEMS1)
      if (Index > NUMELEMS-i)
        break;
      if (Array [Index] > Array [Index + 1]) {    //$ (label innen)
        Exchange(&Array[Index], &Array[Index+1]);
        Sorted= FALSE;
      }
    }
    if (Sorted) break;
  }
}
//$ (constraint (<= innen (* aussen (/ (* NUMELEMS1 NUMELEMS) 2))))

void timingInit() {
  int i;
  /* worst-case init: */
  for (i=1; i<=NUMELEMS; i++) //$ (iterations NUMELEMS)
    Array[i]= -i;
```

```
}

int main() {
  timingInit();
  sort();
}
```

A.4 Matrixmultiplikation

Eine Standard-5×5-Matrixmultiplikation, die bei Matrixhöhe und -breite n $O(n^3)$ Zeit benötigt.

```
void matmul2(a,b,c)
  int a[6][6], b[6][6], c[6][6];  /* die Felder beginnen bei 1 */
{
  int i,j,k;              //$ (defterm e5 (exactly 5))
  for(i=1;i<= 5 ;i++) {
    for(j=1;j<= 5 ;j++) {
      c[i][j] = 0;        //$ (iterations e5 e5) (loop-type opt opt)
    }
  }
  for(i=1;i<= 5 ;i++) {
    for(j=1;j<= 5 ;j++) {
      for(k=1;k<= 5 ;k++) {
        c[i][j] += a[i][k] * b[k][j];
                          //$ (iterations e5 e5 e5) (loop-type opt opt opt)
      }
    }
  }
}
```

A.5 Zeichnen eines Kreises

Ebenfalls aus der cinderella-Testsuite entnommen ist das Testprogramm zum Zeichnen von Kreisen. Ursprünglich stammt es aus Rajesh Kumar Guptas Doktorarbeit. Es wurden Teile weggelassen, die die berechneten Punkte an die Zeichenroutinen übertragen, da diese nicht verfügbar waren.

Das Programm ist relativ lang, deshalb wurde auf den Abdruck verzichtet. Es waren nur Schleifenbegrenzungen nötig, keinerlei zusätzliche Nebenbedingungen, um die Analyse durchzuführen.

A.6 Diskrete Kosinustransformation für JPEG-Bilder

Dieses Testprogramm dient der Überarbeitung von JPEG-Bildern, um die Kompressionsrate zu erhöhen. Es beruht auf dem Prinzip, daß das menschliche Auge auf leichte Farbunterschiede weniger reagiert als auf Helligkeitsunterschiede. Daher werden schwache Farbunterschiede im Bild entfernt.

Das Testprogramm besteht aus drei Schleifen mit fest vorgegebenen Start- und Endwerten. Keine zusätzlichen Nebenbedingungen außer den Schleifenbegrenzungen wurden angegeben.

A.7 Statistikberechnungen

Das Programm berechnet Summe, Varianz, Durchschnitt und Standardabweichung für zwei Felder aus 1000 Zahlen. Das Programm erforderte neben Schleifenbegrenzungen keinerlei zusätzliche Nebenbedingungen.

A.8 Verschlüsselung nach dem DES-Standard

Ein Programm, das Daten verschlüsselt. Es enthält 12 Schleifenbegrenzungen und zwei zusätzliche Nebenbedingungen, die komplexere Zusammenhänge definieren. Die meisten Schleifen haben feste Anfangs- und Endwerte.

Abbildungsverzeichnis

Symbolverzeichnis

Symbole, zu denen keine Seitenzahl angegeben ist, werden als bekannt vorausgesetzt. Ggf. ist eine kurze, u. U. informale Definition angegeben. Symbole, die in einem bestimmten Kontext in der Arbeit leicht andere Definitionen haben können, sind mit *lokal* markiert. Dabei handelt es sich meist um Symbole, die im lokalen Kontext aus eingeschränkteren Menge gewählt sind, als hier allgemein angegeben.

$\emptyset = \{\}$	leere Menge	
$\therefore$	so, daß	
$[v_1, v_2\rangle$	Kante eines Graphen	19
$\exists$	es existiert	
$\forall$	für alle	
$M^n = \underbrace{M \times M \times \cdots \times M}_{n\text{-mal}}$	Menge von Tupeln	
$f^{-1}(y) = \{x \mid x \in X,\ f(x) = y\}$	Umkehrabbildung	
$\lceil x \rceil = \mathsf{min}\{y \in \mathbb{Z} \mid y \geqq x\}$	aufgerundeter Wert	
$\lfloor x \rfloor = \mathsf{max}\{y \in \mathbb{Z} \mid y \leqq x\}$	abgerundeter Wert	
$\lvert x \rvert = \begin{cases} x & \text{falls } x \geqq 0 \\ -x & \text{sonst} \end{cases}$	Betragswert	
$(a_i)_{i=1,\ldots,n} = (a_1, a_2, \ldots, a_n)$	Kurzschreibweise für Tupel	
$k \circ k' = (k_1, \ldots, k_n, k'_1, \ldots, k'_m)$	Konkatenation von Tupeln	
$f(A) = \{f(a) \mid a \in A\}$	Abstraktion von f auf Mengen	

$A \diamond b = \{a \diamond b \mid a \in A\}$	Abstraktion von $\diamond$ auf Mengen	
$b \vdash b' \Leftrightarrow b,\ b' \in V,\ [b, b'\rangle \in E$	Pfad der Länge 1 von b nach b'	
$b \vdash^* b' \Leftrightarrow \exists b_i \therefore [b, b_1\rangle, [b_i, b_{i+1}\rangle, [b_n, b'\rangle \in E$	Pfad von b nach b'	
$M^+ = \bigcup_{i=1}^{\infty} M^i$	positive Kleenesche Hülle	
$M^* = \bigcup_{i=0}^{\infty} M^i$	Kleenesche Hülle	
$M \subseteq N \iff \forall m \in M:\ m \in N$	Teilmenge	
$M \subset N \iff M \subseteq N$ und $M \neq N$	echte Teilmenge	
$b \in B$	ein Basisblock	*lokal*
b_{start}	Einsprungpunkt ins Programm	19
B	Menge der Basisblöcke	18
$back : \mathcal{L} \rightarrow \mathfrak{P}(\mathcal{E})$	Rückkanten einer Schleife	23
$c[b, \vartheta] \in \mathbb{N}_0,\ b \in B,\ \vartheta \in T$	Ausführanzahl eines Basisblocks	35
$c[e, \vartheta] \in \mathbb{N}_0,\ e \in \mathfrak{E},\ \vartheta \in T$	Durchlaufanzahl einer Kante	35
$\hat{c}[b, \vartheta] \in \mathbb{N}_0,\ b \in B,\ \vartheta \in T$	Ausführanzahl ab zweiter Ausführung	50
$\mathcal{C} \subseteq B$	Menge der Prozeduraufrufe	18
$\mathsf{C}[b],\ b \in \mathcal{C}$	erster Aufruf von b aus	25
$e \in \mathfrak{E}$	eine Kante	*lokal*
ε	Der leere Kontext	25
$E \subseteq V \times V$	Menge der Kanten	19
$\mathcal{E}$	Menge aller Kanten	20
$\hat{E}$	Kanten im Call-Graphen	19
$\mathfrak{E} = \mathcal{E} \cup \{[b, f_{call}(b)\rangle \mid b \in \mathcal{C}\}$	Kantenmenge von $\mathfrak{G}$	20
$entry : \mathcal{L} \rightarrow \mathcal{E}$	Eintrittskante einer Schleife	23
$exit : \mathcal{L} \rightarrow \mathfrak{P}(\mathcal{E})$	Austrittskanten einer Schleife	23
$f_{call} : B \rightarrow B \cup \{\mathsf{nocall}\}$	Prozeduraufruf eines Basisblocks	18
$\mathcal{F}$	Menge der Prozeduren	18
$\mathsf{F}[b],\ b \in head(\mathcal{L})$	erster Durchlauf einer Schleife	25
$G = (V, E, v_0)$	Kontrollflußgraph	19
$\mathcal{G}$	Menge der Kontrollflußgraphen	20
$\hat{G} = (\mathcal{F}, \hat{E}, \varphi_{start})$	Call-Graph	19
$\mathfrak{G} = (B, \mathfrak{E})$	Gesamtkontrollflußgraph	20
$head : \mathcal{L} \rightarrow B$	Kopf einer Schleife	22
$i \in I$	ein Maschinenbefehl	*lokal*
I	Folge der Maschinenbefehle	17

$in(b) \subseteq \mathcal{E},\ b \in B$	eingehende Kanten	20
$inner : \mathcal{L} \to B$	innerer Basisblock	23
$k_{\mathrm{rec}}[b, \vartheta],\ b \in B,\ \vartheta \in T$	Rekursionstiefe	41
$\tilde{k}_{\mathrm{rec}}[b, \vartheta],\ b \in B,\ \vartheta \in T$	geschätzte Rekursionstiefe	41
$k_{\mathrm{rep}}[b, \vartheta],\ b \in B,\ \vartheta \in T$	Ausführanzahl einer Schleife	44
$\tilde{k}_{\mathrm{rep}}[b, \vartheta],\ b \in B,\ \vartheta \in T$	geschätzte Ausführanzahl	44
$l \in \mathcal{L}$	eine Schleife	*lokal*
$\mathcal{L}$	Menge der Schleifen	22
$\mathbb{N} = \{1, 2, 3, \ldots\}$	natürliche Zahlen	
$\mathbb{N}_0 = \mathbb{N} \cup \{0\}$	natürliche Zahlen mit 0	
$\mathsf{O}[b],\ b \in head(\mathcal{L})$	weitere Durchläufe einer Schleife	25
$out(b) \subseteq \mathcal{E},\ b \in B$	ausgehende Kanten	20
$\varphi \in \mathcal{F}$	eine Prozedur	*lokal*
φ_{start}	Hauptprogramm	18
$P(\vartheta),\ \vartheta \in T$	Menge der Kontexte mit Präfix ϑ	40
$P(\Theta),\ \Theta \subseteq T$	Menge der Kontexte mit Präfixen Θ	40
$P_f(b, \vartheta),\ b \in B,\ \vartheta \in \tau(b)$	Kontexte mit Präfix eines Prozeduraufrufs	41
$\mathcal{P}$	Programm	17
$\mathfrak{P}(M) = \{N \mid N \subseteq M\}$	Potenzmenge	
$\Phi : B \to \mathcal{F}$	Prozedur eines Basisblocks	18
$pre : \mathcal{L} \to B$	Pre-Header einer Schleife	22
$\mathbb{R}$	reelle Zahlen	
$\mathbb{R}_{\geqq 0} = \{x \mid x \in \mathbb{R} \text{ und } x \geqq 0\}$	positive reelle Zahlen	
$\mathsf{R}[b],\ b \in \mathcal{C}$	rekursiver Aufruf von b aus	25
$t[b, \vartheta] \in \mathbb{R}_{\geqq 0},\ b \in B,\ \vartheta \in T$	Ausführzeit eines Basisblocks	34
$\hat{t}[b, \vartheta],\ b \in B,\ \vartheta \in T$	Beschleunigung ab zweiter Ausführung	50
$\hat{t}'[i, \vartheta],\ i \in I,\ \vartheta \in T$	Beschleunigung ab zweiter Ausführung	50
$t'[i, \vartheta] \in \mathbb{R}_{\geqq 0},\ i \in I,\ \vartheta \in T$	Ausführzeit eines Befehls	33
$\tau : B \to \mathfrak{P}(T)$	Kontexte eines Basisblocks	25
$\tau([b, b'\rangle) = \tau(b)$	Kontexte einer Kante	25
$\vartheta \in T$	ein Kontext	*lokal*
$T = \bar{T}^*$	Menge der Kontexte	24
$\bar{T}$	Menge der Schritte eines Kontextes	24
$v \in B$	ein Knoten im Kontrollflußgraphen	*lokal*
$V \subseteq B$	Menge der Knoten	19

$\hat{Z}$	Zielfunktion im verbesserten Problem	50
Z	Zielfunktion	35
$\mathbb{Z} = \mathbb{N}_0 \cup -\mathbb{N}$	ganze Zahlen	

Glossar

Assoziativität eines Cache. Die Anzahl gleichzeitig im Cache speicherbarer Speicherzeilen, die in die gleiche Cache-Zeile fallen.

Cache. Schneller Zwischenspeicher zwischen Prozessor und Hauptspeicher, der den Zugriff auf häufig verwendete Daten beschleunigen soll.

Cache-Hit. Ist ein Datum beim Zugriff des Prozessors auf den Hauptspeicher im Cache zwischengespeichert, so bezeichnet man den Speicherzugriff als *Cache-Hit*.

Cache-Miss. Ist ein Datum beim Zugriff des Prozessors auf den Hauptspeicher nicht im Cache zwischengespeichert, so bezeichnet man den Speicherzugriff als *Cache-Miss*.

Cache-Zeile. Ein Cache gliedert sich in Blöcke eingeteilt, die Cache-Zeilen heißen. Die Größe der Blöcke wird als Zeilenlänge des Cache bezeichnet und entspricht der Größe der Blöcke, die zwischen Hauptspeicher und Cache ausgetauscht werden.

EEL. (*executable editing library*). Eine Bibliothek zum Bearbeiten von Programmen, die im Binärformat vorliegen (*siehe* [WAR97]).

`else`-Teil. Bei einem bedingten Stück Code der Teil, der ausgeführt wird, wenn die Bedingung nicht erfüllt ist. Benannt nach dem Pascal-Schlüsselwort `ELSE` in `IF b THEN x ELSE y`.

gcc GNU C Compiler. Der benutzte Compiler für C/C++.

Hashing. Das eindeutige Abbilden von Objekten beliebiger Datentypen auf eine Zahl zwischen 0 und $t-1$, wobei t als Hashtafelgröße bezeichnet wird. Für gewöhnlich verwendet, um beliebige Objekte in Wörterbücher einzutragen.

ILP (*Integer linear programming* oder *integer linear program*). Ganzzahliges lineares Programmieren bzw. ganzzahliges lineares Problem. Siehe Abschnitt 2.2 auf Seite 26.

inline. Man sagt, ein Compiler übersetze einen Prozeduraufruf *inline*, falls der Aufruf durch den Rumpf der Prozedur ersetzt wird. Bei vielen Prozeduren wird dadurch die Ausführung beschleunigt, da ein Sprung fehlt, und da mehr Optimierungen möglich sind, wenn Parameter für den speziellen Aufruf bekannt sind.

LP (*linear programming* oder *linear program*). Lineares Programmieren bzw. lineares Problem. Siehe Abschnitt 2.2 auf Seite 26.

Qpt2 Quick program Profiler and Tracer. Im WARTS enthalten. Siehe dort.

quasireell. Da in Rechenanlagen aufgrund ihrer Unendlichkeit keine reellwertigen Zahlen gespeichert werden können, müssen sie u. U. mit einem gewissen Fehler dargestellt werden. Die Menge der geschlossen abspeicherbaren (d. h. nicht durch mathematische Formeln beschriebenen) Zahlen heißt Menge der *quasireellen* Zahlen.

Speicherzeile. Teilt man den Hauptspeicher in Einheiten der Zeilenlänge des Cache ein, so heißt eine dieser Einheiten eine Speicherzeile.

`then`-Teil. Bei einem bedingten Stück Code der Teil, der ausgeführt wird, wenn die Bedingung erfüllt ist. Benannt nach dem Pascal-Schlüsselwort `THEN` in `IF b THEN x ELSE y`.

WARTS Wisconsin Architectural Research Tool Set. Die Bibliothek, in der der Tracer Qpt2 und die Bibliothek EEL enthalten sind. Siehe [WAR97].

Zeilenlänge eines Cache. Die Anzahl Bytes, die ein Cache in einem Stück aus dem Hauptspeicher lädt und als unzerlegbare Einheit speichert. Typische Werte sind 16 Byte (z. B. AMD-K6-Prozessor) oder 32 Byte (z. B. DEC Alpha 21164).

Literaturverzeichnis

[AFMW96] Martin Alt, Christian Ferdinand, Florian Martin und Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *Proceedings of SAS'96, Static Analysis Symposium*, LNCS 1145, Seite 52–66. Springer, September 1996.

[AM95] Martin Alt und Florian Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *Proceedings of SAS'95, Static Analysis Symposium*, LNCS 983, Seite 33–50. Springer, September 1995.

[AMW95] Martin Alt, Florian Martin und Reinhard Wilhelm. Generating Dataflow Analyzers with PAG. Technical Report A10-95, Universität des Saarlandes, 1995.

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

[ATG96] Ali-Reza Adl-Tabatabai und Thomas Gross. Source-Level Debugging of Scalar Optimized Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Seite 33–43, Mai 1996.

[BB94] Peter T. Breuer und Jonathan P. Bowen. Decompilation: The Enumeration of Types and Grammars. In *ACM Transactions on Programming Languages and Systems*, Band 16(5), Seite 1613–1647, September 1994.

[Ber] Michel Berkelaar. FTP directory of lp_solve. `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

[BHS92] Gary Brooks, Gilbert J. Hansen und Steve Simmons. A New Approach to Debugging Optimized Code. In Christopher W. Fraser, Hrsg., *Proceedings* *of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, Seite 1–11, San Francisco, CA, June 1992. ACM Press.

[BL92] Thomas Ball und James Larus. Optimally Profiling and Tracing Programs. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, Seite 59–70, Januar 1992.

[CC77a] Patrick Cousot und Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, Seite 238–252, Januar 1977.

[CC77b] Patrick Cousot und Radhia Cousot. Static Determination of Dynamic Properties of Generalized Type Unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, Band 12(3), Seite 77–94, Raleigh, NC, März 1977.

[CG92] Christina Cifuentes und K. John Gough. A New Approach to Debugging Optimized Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[Chv83] Vašek Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.

[Cin] Cinderella 3.0 Home Page. `http://www.ee.princeton.edu/~yauli/cinderella-3.0/`.

[CLR90] Thomas H. Cormen, Charles E. Leiserson und Roland L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CWM93] Samit Chaudhuri, Robert A. Walker und John Mitchel. The Structure of Assignment, Precedence, and Resource Constraints in the ILP Approach to the Scheduling Problem. In Edna Straub, Hrsg., *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Seite 25–31. IEEE Computer Society Press, 1993.

[Fer97a] Christian Ferdinand. A Fast and Efficient Cache Persistence Analysis. Technical Report 10/97, Universität des Saarlandes, Sonderforschungsbereich 124, August 1997.

[Fer97b] Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. Dissertation, Universität des Saarlandes, September 1997.

[FMW97] Christian Ferdinand, Florian Martin und Reinhard Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, Seite 37–46, Juni 1997.

[FMW98] Christian Ferdinand, Florian Martin und Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming*, 1998. Ausgewählt für Sonderband SAS'96.

[FMWA97] Christian Ferdinand, Florian Martin, Reinhard Wilhelm und Martin Alt. Cache Behavior Prediction by Abstract Interpretation. Technical Report 05/97, Universität des Saarlandes, Sonderforschungsbereich 124, Mai 1997.

[FW98] Christian Ferdinand und Reinhard Wilhelm. Fast and Efficient Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 1998. submitted for publication.

[HBL+95] Yerang Hur, Young Hyun Bae, Sung-Soo Lim, Sung-Kwan Kim, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Minsuk Lee, Heanshik Shin und Chong Sang Kim. Worst Case Timing Analysis of RISC Processors: R3000/R3010 Case Study. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, Seite 308–319, Dezember 1995.

[HU94] John E. Hopcroft und Jeffrey D. Ullman. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Addison-Wesley, dritte Auflage, 1994.

[JPE95] Multimedia-Datenformate, Kapitel 2: JPEG-Bilddatenkompression. `http://www.tp4.ruhr-uni-bochum.de/MultiMedia/02_JPEG/index.html`.

[JR94] Daniel Jackson und Eugene J. Rollins. A New Model of Program Dependencies for Reverse Engeneering. In *Proceedings of the 1994 ACM SIGSOFT Conference*, Dezember 1994.

[Käs97] Daniel Kästner. Instruktionsanordnung und Registerallokation auf der Basis ganzzahliger linearer Programmierung für den digitalen Signalprozessor ADSP-2106x. Diplomarbeit, Universität des Saarlandes, September 1997. `http://www.cs.uni-sb.de/RW/users/kaestner/isra.ps.gz`.

[KL98] Daniel Kästner und Marc Langenbach. Integer Linear Programming vs. Graph-Based Methods in Code Generation. Technical Report A/01/98, Universität des Saarlandes, Februar 1998. `http://www.cs.uni-sb.de/RW/users/kaestner/codegen.ps.gz`.

[Lar90] James Larus. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software Practice and Experience*, 20(12):1241–1258, Dezember 1990.

[Lar96] James Larus. *EEL Guts: Using the EEL Executable Editing Library*. Computer Sciences Department, University of Wisconsin-Madison, 1996. `http://www.cs.wisc.edu/~larus/eel.html`.

[LBJ+95] Sung-Soo Lim, Young Hyun Bae, Gye Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon und Chong Sang Kim. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, Juli 1995.

[LM95] Yau-Tsun Steven Li und Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, Seite 456–461, Juni 1995.

[LMW95a] Yau-Tsun Steven Li, Sharad Malik und Andrew Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, Seite 298–307, Dezember 1995.

[LMW95b] Yau-Tsun Steven Li, Sharad Malik und Andrew Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, Seite 380–387, November 1995.

[LMW96] Yau-Tsun Steven Li, Sharad Malik und Andrew Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, Dezember 1996.

[LS95] James R. Larus und Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Seite 291–300, Juni 1995.

[LT79] Thomas Lengauer und Robert Endre Tarjan. A Fast Algorithm for Finding Dominators in Flowgraphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, Juli 1979.

[Mar95] Florian Martin. Entwurf und Implementierung eines Generators für Datenflußanalysatoren. Diplomarbeit, Universität des Saarlandes, 1995.

[MAWF98] Florian Martin, Martin Alt, Reinhard Wilhelm und Christian Ferdinand. Analysis of Loops. In Kai Koskimies, Hrsg., *Proceedings of the 7th International Conference on Compiler Construction*, Band 1383 of *Lecture Notes in Computer Science*. Springer, März /April 1998.

[Mue94] Frank Mueller. Static Cache Simulation and its Applications. PhD Thesis, Florida State University, Juli 1994.

[NW88] George L. Nemhauser und Laurence A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons Ltd., New York, 1988.

[OS97] Greger Ottoson und Mikael Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *Proceedings of the ACM SIGPLAN*

Workshop on Language, Compiler and Tool Support for Real-Time Systems, Seite 47–55, Juni 1997.

[PK89] Peter Puschner und Christian Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1:159–176, 1989.

[PS91] Chang Yun Park und Alan C. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, 24(5):48–57, Mai 1991.

[Ram97] Thomas Ramrath. Entwurf und Implementierung eines Frontends für Analysen zur Vorhersage des Cache- und Pipeline-Verhaltens. Diplomarbeit, Universität des Saarlandes, November 1997.

[RH97] Monika Rauch Henzinger. Continuous Profiling: Where Have All the Cycles Gone? Vortrag im Rahmen des Informatik-Kolloquiums, Universität des Saarlandes, August 1997.

[San95] Georg Sander. *VCG: Visualisation of Compiler Graphs, User Documentation V.1.30*. Universität des Saarlandes, Februar 1995.

[Sch96a] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons Ltd., 1996.

[Sch96b] Hartmut Schwab. *Documentation for lp_solve*, 1996. Im lp_solve-Paket als `HARTMUT_DOCUMENTATION` enthalten.

[Sch98] Jörn Schneider. Statische Pipeline-Analyse für Echtzeitsysteme. Diplomarbeit, Universität des Saarlandes, Oktober 1998. noch nicht erschienen.

[Sei96] Raimund Seidel. *Skript zur Vorlesung Optimierung*. Universität des Saarlandes, Sommersemester 1996.

[Sic97] Martin Sicks. Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches. Diplomarbeit, Universität des Saarlandes, 1997.

[SJ90] Guy L. Steele Jr. *Common Lisp, The Language*. Digital Press, zweite Auflage, 1990.

[SP81] Micha Sharir und Amir Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In Steven S. Muchnick und Neil D. Jones, Hrsg., *Program Flow Analysis: Theory and Applications*, Kapitel 7, Seite 189–233. Prentice-Hall, 1981.

[SPA90] *SPARC Architecture Manual*, 1990.

[SS93] Günter Scheja und Uwe Storch. *Lehrbuch der Algebra*. B. G. Teubner Stuttgart, zweite Auflage, 1993. Teil 1.

[Sta96] John A. Stankovic. *Real-Time and Embedded Systems*. ACM 50th Anniversary Report on Real-Time Computing Research, 1996. `http://www-ccs.cs.umass.edu/sdcr/rt.ps`.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, dritte Auflage, 1997.

[TG98] Caroline Tice und Susan L. Graham. OPTVIEW: A New Approach for Examining Optimized Code. In *Proceedings of the 1998 Workshop on Program Analysis for Software Tools and Engineering, Montreal*, Juni 1998.

[TMLA98] Stephan Thesing, Florian Martin, Oliver Lauer und Martin Alt. *PAG Manual V1.0 12.03.1998*. Fachbereich 14, Universität des Saarlandes, 1998.

[TNPa] The Nine Planets: Callisto. `http://www.seds.org/billa/tnp/callisto.html`.

[TNPb] The Nine Planets: Deimos. `http://www.seds.org/billa/tnp/deimos.html`.

[TNPc] The Nine Planets: Phobos. `http://www.seds.org/billa/tnp/phobos.html`.

[vDKT93] Arie van Deursen, Paul Klint und Frank Tip. Origin Tracking. *Journal of Symbolic Computation*, 15(5):523–546, 1993.

[WAR97] WARTS: Wisconsin Architectural Research Tool Set. `http://www.cs.wisc.edu/~larus/warts.html`.

[WM95] Reinhard Wilhelm und Dieter Maurer. *Übersetzerbau — Theorie, Konstruktion, Generierung*. Springer-Verlag, zweite Auflage, 1995.

[Zei95] Patrik Zeimetz. Transformationen zur Verbesserung der Datenlokalität in Bezug auf Cachespeicher. Diplomarbeit, Universität des Saarlandes, 1995.

Stichwortverzeichnis