DIPLOMARBEIT

Erzeugung optimierten Codes für Stackmaschinen

ausgeführt am Institut für Computersprachen
der Technischen Universität Wien

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

durch
Martin Maierhofer
Haslingergasse 19/1
1170 Wien

Wien, 31. Januar 1997

Meinen Eltern in Dankbarkeit gewidmet.

Ein Dank gilt auch meinem Betreuer Anton Ertl für seine nützlichen Kommentare und nicht zuletzt für die Ideen, auf denen diese Arbeit aufbaut.

Zusammenfassung

Nicht zuletzt durch die Verbreitung der Programmiersprache Java bzw. der ihr zugrundeliegenden JavaVM-Architektur erleben Stackarchitekturen derzeit eine Renaissance. Für eine effiziente Unterstützung Algol-ähnlicher Hochsprachen auf Stackmaschinen bedarf es auf ihre Architektur abgestimmter Techniken. Unter anderem soll ein optimierender Compiler unnötige Speicherzugriffe, die durch das Lesen und Schreiben von lokalen Variablen entstehen, unterdrücken und so eine effiziente Ausführung des erzeugten Codes ermöglichen. Dazu werden die Variablen temporär auf dem Stack zwischengespeichert. Ein solches Verfahren ist allerdings nur sinnvoll, wenn Speicherzugriffe aufwendiger sind als Stackmanipulationen.

Im Rahmen dieser Arbeit werden zwei Ansätze zur Lösung dieses Problems vorgestellt und verglichen, wobei zwei Stackarchitekturen (u.a. JavaVM) als Zielarchitekturen Verwendung finden. Beide Methoden arbeiten „lokal", d.h. die Optimierung findet nur innerhalb von basic blocks statt. Die erste, von Phil Koopman vorgeschlagene, Methode des „stack scheduling" wird in Zusammenspiel mit einem einfachen instruction scheduling (ein „depth first postorder traversal" des Abhängigkeitsgraphen) eingesetzt. „DAG scheduling" als zweiter Ansatz verändert die Reihenfolge der Befehle (führt also ein instruction scheduling durch), um die Anzahl der Variablenzugriff so klein wie möglich zu halten, und ermittelt den optimalen Code bezüglich Variablenzugriffen.

Damit wird es möglich, die Effizienz von Koopmans Methode zu bewerten. Es zeigt sich, daß die Ergebnisse des stack scheduling dem Optimum nahe kommen. Der Anteil von Variablenzugriffen in JavaVM-Code kann gegenüber dem durch den Java Compiler `javac` erzeugten Code von typischerweise 40% auf etwa 30% reduziert werden. Dafür steigt der Anteil von Stackmanipulationsbefehlen von etwa 5% auf bis 15%.

Abstract

The success of the Java programming language and the underlying stack architecture (the JavaVM) recently have caused renewed interest in stack architectures. New and special techniques are required to provide support for the efficient execution of Algol-like high level languages on stack machines. An optimizing compiler should be able to eliminate dispensable accesses to main memory when loading and storing local variables by temporarily keeping a copy of their value on the stack. This technique, however, is only meaningful for machines that can handle stackmanipulations faster than accesses to main memory.

This thesis presents two solutions for the problem and compares the results that can be achieved for two stack architectures (one of them is the JavaVM). Both techniques work on a „local" level, meaning that each basic block is considered separately. Phil Koopman's „stack scheduling" performs well in cooperation with a simple instruction scheduling strategy (a depth first postorder traversal of the dependency graph is used). The second technique („DAG scheduling") reorders the instructions (i.e. it schedules the instructions) in order to minimize accesses to local variables and leads to optimal code.

The efficiency of Koopman's algorithm can be assessed with respect to the results achieved by DAG scheduling: stack scheduling leads to results that come quite near the optimum. Accesses to variables can be reduced from around 40% (in code produced by the Java compiler `javac`) to some 30% (after optimization) of the total instructions in JavaVM code. Instructions for stackmanipulation, however, increase from about 5% to up to 15% of the total.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Stack compilers aren't currently very efficient – but that's because no-one really tried all that hard.

Phil Koopman, *Why Stack Machines?*

Ziel dieser Arbeit ist der Vergleich von Methoden zur Optimierung von Stackcode. Neben traditionellen Methoden bedarf es hierfür speziell auf die Bedürfnisse von Stackarchitekturen abgestimmter neuer Techniken, die in der wissenschaftlichen Literatur kaum behandelt werden. Neben einer Einführung in die Besonderheiten und Vorteile von Stackarchitekturen gegenüber Registermaschinen und der Einengung der „Optimierung" auf einen speziellen Teilbereich – Zugriffe auf lokale Variablen – enthält dieses Kapitel auch einen Überblick über den weiteren Inhalt der Arbeit.

1.1 Stackmaschinen

Nach ihrer „Blüte" in den 60er Jahren und einer Periode relativer Vergessenheit erfreuen sich Stackarchitekturen seit einiger Zeit wieder größerer Aufmerksamkeit in der Fachwelt. Neben Suns „OpenBoot" bzw. dem darin spezifizierten F-Code ([Sun94]), der auch für PCI Peripheriekarten Verwendung findet, ist hier vor allem die der Programmiersprache Java zugrundeliegende Stackarchitektur der „Java Virtual Machine" (JavaVM) zu nennen, die seit etwa Mitte 1995 großes Aufsehen erregt.

Die Hauptanforderungen in diesen Anwendungsbereichen sind geringe Codegröße und plattformübergreifende Verwendbarkeit. Allerdings besitzen Stackarchitekturen darüber hinaus noch eine weitere Reihe von Merkmalen, die vor allem für die Realisierung in Hardware von Bedeutung sind. In einem Artikel im *Usenet* nennt Phil Koopman unter anderem folgende Vorteile von Stackmaschinen gegenüber „konventionellen“ Registerarchitekturen ([Koo93]):

Komplexität Seit einiger Zeit nützt praktisch jede high-performance CPU mit Registerarchitektur „pipelining“ ([Hen90]), um den Befehlsdurchsatz zu erhöhen, erkauft sich das jedoch um den Preis höherer Komplexität (so müssen etwa falsche Sprungvorhersagen durch aufwendige „Aufräumarbeiten“ kompensiert werden). Stackarchitekturen umgehen den damit verbundenen Aufwand durch die implizite Adressierung über den Stack, auf den üblicherweise via „buffering“ sehr schnell zugegriffen werden kann (vergleiche Abschnitt 2.3).

Interrupts Ein „interrupt“ kann in einer Stackarchitektur praktisch „sofort“ behandelt werden; es genügt, die aktuelle Adresse im Programm zu sichern (hierzu eignet sich ein Stack vorzüglich) und in die „interrupt service routine“ zu verzweigen. Registermaschinen hingegen müssen bei jedem interrupt ihren aktuellen Zustand (etwa den Inhalt der Register) speichern.

Codegröße [Koo89] führt gegenüber Code für CISC-Architekturen eine um den Faktor 2,5 bis 8 reduzierte Größe von Stackcode an; gegenüber RISC-Architekturen steige dieser Faktor noch einmal um 1,5 bis 2,5. Neben kompakteren Befehlscodes (Wegfall der expliziten Adressierung von Operanden) sind dafür auch die implizite Argumentübergabe über den Stack bzw. häufigere Wiederverwendung von kurzen Unterprogrammen verantwortlich.

Eine andere Untersuchung ([Ert95b]) kommt jedoch zu anderen Ergebnissen: Forth-Code ist hier meist *größer* (bis zu einem Faktor von 2) als Maschinencode für die Intel 80x86 Architektur. Die Zahlen mögen dadurch begründet sein, daß bei diesem Vergleich Forth-Programme automatisch in C-Code konvertiert wurden, welcher wiederum in Maschinencode übersetzt wurde. Der C-Code folgt somit dem Programmierstil in Forth (viele kurze Routinen, die oft aufgerufen werden) und führt offensichtlich zu kompaktem Maschinencode. Außerdem werden im Forth-Zwischencode Instruktionen mit einer Breite von 32 Bit verwendet, obwohl auch 8 Bit zur Codierung genügen würden.

Diese Eigenschaften erklären den relativ hohen Anteil von Stackarchitekturen im „embedded controller“- Bereich, da dieses Gebiet besondere Anforderungen an Größe, Preis und Leistung der Hardware stellt.

Vor der angesprochenen Renaissance hatten Stackarchitekturen allerdings immer schon eine relativ kleine aber treue Anhängerschaft, die sich vor allem durch die Verwendung der Programmiersprache Forth ([Woe92]) ergab. Forth ist wie geschaffen für die Programmierung von

Stackmaschinen, und tatsächlich entstehen immer wieder Prozessoren, die sich eine möglichst effiziente Ausführung von Forth an die Fahnen heften und teilweise auch Forth-Primitive als Instruktionen verwenden ([Hay87]).

1.2 Optimierter Code

Das Attribut „optimiert“ kann sich auf dem Gebiet der Codeerzeugung auf mehrere Qualitäten des Resultates beziehen:

- Für viele Anwendungen ist es wichtig, möglichst kleinen, kompakten Code zu erzeugen (man denke an den embedded controller-Bereich). Allerdings neigt, wie schon beschrieben, Code für Stackarchitekturen ohnehin zu geringer Größe.
- In Zeiten ständig sinkender Preise für Speicher ist allerdings für die breite Masse der Anwendungen die Ausführungsgeschwindigkeit des erzeugten Codes wichtiger als dessen Größe.

Diese Arbeit konzentriert sich auf zweiteres Gebiet, allerdings werden dabei einige Einschränkungen in Kauf genommen, um den Aufwand in Grenzen zu halten:

- Es werden kaum globale Optimierungstechniken verwendet, der Schwerpunkt liegt auf lokalen (d.h. auf einen „basic block“ beschränkten) Methoden. Das Konzept der basic blocks wird in Abschnitt 3.2 näher erläutert.
- Das Hauptaugenmerk der verwendeten Methoden liegt auf der Optimierung von Zugriffen auf lokale Variablen. Wo immer möglich soll vermieden werden, eine lokale Variable öfter aus dem Hauptspeicher nach- und auf den Stack zu laden. Stattdessen sollen Kopien von lokalen Variablen auf dem Stack zwischengespeichert und durch den daraus erwachsenden unmittelbaren Zugriff die Berechnungen beschleunigt werden.
- Als erste Zielarchitektur kommt eine sehr einfache, hypothetische Stackmaschine zum Tragen, die keine ausgefeilten Möglichkeiten zur Stackmanipulation bietet und somit die möglichen Optimierungen einschränkt.
- Eine zweite Stackarchitektur, die in dieser Arbeit herangezogen wird, ist die „Java Virtual Machine“ oder kurz JavaVM ([Sun95]). Auch diese Architektur ist auf dem Gebiet der Stackmanipulationen nicht besonders leistungsfähig und macht daher einerseits die Optimierungsmethoden etwas weniger aufwendig, läßt aber andererseits deren Effizienz etwas sinken.

Soll Code für Stackmaschinen optimiert und dadurch in der Ausführung effizient (in unserer Lesart also schneller) werden, entstehen einige Komplikationen. Analog zu optimiertem Code für Registermaschinen (bei dem die Operanden eines Befehls auch nicht jedesmal aus dem Speicher geladen, sondern in Registern zwischengespeichert werden) sollen öfter gebrauchte Variablen auf dem Stack zwischengespeichert werden. Mittels geeigneter Stackmanipulationen können diese Variablen auf dem Stack so verschoben werden, daß sie im richtigen Moment als Operanden zur Verfügung stehen. Eine Methode zur Lösung dieses Problems muß also einerseits die Variablen auswählen, für die eine Zwischenspeicherung auf dem Stack günstig scheint, andererseits auch versuchen, den Aufwand für die Stackmanipulationen minimal zu halten. Dabei zieht die Wahl der Maschinenarchitektur einige Einschränkungen nach sich (etwa bezüglich des „wahlfreien“ Zugriffs auf einzelne Elemente im Stack).

Die Problematik der Registerbelegung für konventionelle Architekturen ist mit Chaitins Methode des „graph coloring“ ([Cha81]) weitgehend entschärft, hingegen harrt das analoge Problem für Stackarchitekturen noch einer generellen Lösung. Das ist zum Teil dadurch zu erklären, daß sich der Übersetzerbau sehr auf Registerarchitekturen konzentriert hat, die ja den bei weitem größten Anteil an den Rechnerarchitekturen im high-performance Markt stellen. Allerdings ist es auch einfacher, mit 32 Registern zu „jonglieren“, auf die beliebig zugegriffen werden kann, als mit mehr oder weniger stark eingeschränktem Zugriff auf die Elemente in einem Stack fertig werden zu müssen.

1.3 Überblick über die Arbeit

Kapitel zwei dieser Arbeit enthält neben einer kurzen Einführung in das Konzept des Stack und einem historischen Ausflug in die Entwicklung der Stackmaschinen eine detailliertere Beschreibung der beiden in dieser Arbeit verwendeten Architekturen.

Einige einfache Optimierungstechniken, die nicht speziell auf Stackarchitekturen abzielen, werden in Kapitel drei behandelt; sie bilden teilweise auch die Grundlage der weiteren Optimierungen, welche Thema der folgenden zwei Kapitel sind.

Kapitel vier beschreibt neben einigen früheren Arbeiten zum Thema „Optimierung von Stackcode“ die von Phil Koopman entwickelte Methode des „stack scheduling“. Eine Übersicht über die Implementierung dieses Algorithmus rundet diesen Abschnitt ab.

In Kapitel fünf wird eine neue Methode vorgestellt, die zu Vergleichszwecken die optimale Befehlsreihenfolge („schedule“) ermittelt. Dieser Ansatz basiert auf dem bekannten „DAG scheduling“ für Registermaschinen, muß aber auf die besonderen Erfordernisse von Stackarchitekturen Rücksicht nehmen. Einige Details der Implementierung bilden den Abschluß dieses Teils.

Kapitel sechs stellt die Ergebnisse dar, die mit den beiden Optimierungsmethoden stack scheduling und DAG scheduling erzielt werden. Außerdem enthält der Teil der Arbeit eine Übersicht über die für die Implementierung verwendeten Hilfsmittel und Werkzeuge, sowie eine Beschreibung der Testeingaben zur Ermittlung der Ergebnisse.

Eine Zusammenfassung der wichtigsten Punkte sowie ein Ausblick auf mögliche Erweiterungen sind in Kapitel sieben zu finden.

Anhang A schließlich besteht aus einem statistischen Überblick über die Implementierung; ein Literaturverzeichnis beschließt die Arbeit.

Kapitel 2

Stackmaschinen

Many people who realized the importance of stacks and queues independently have given other names to these structures: stacks have been called push-down lists, reversion storages, cellars, nesting stores, piles, last-in-first-out ("LIFO") lists, and even yo-yo lists!

Donald E. Knuth, *The Art of Computer Programming*

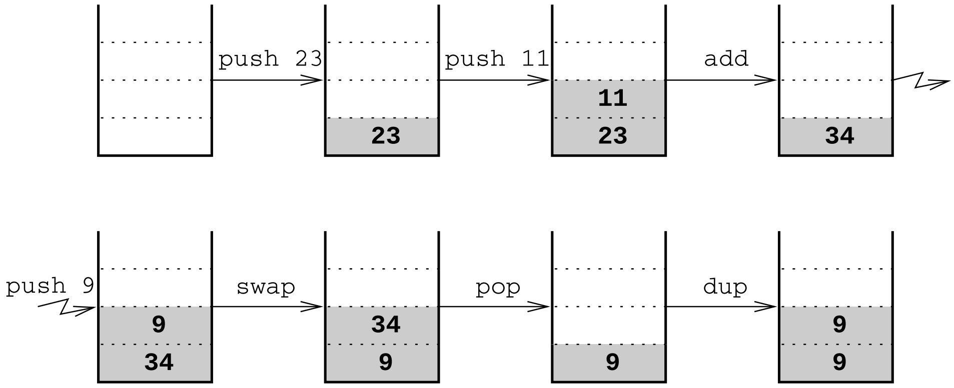
Bevor sich das nächste Kapitel einigen grundlegenden aber nicht stackspezifischen Optimierungstechniken zuwendet, erscheint es nötig, das fundamentale Konzept von Stackarchitekturen zu erläutern. Neben historischen Anmerkungen zur Entwicklung der Stackmaschinen enthält dieses Kapitel daher eine kurze Einführung in die zentrale Idee hinter diesen Maschinen, nämlich den Stack. Außerdem werden die zwei in dieser Arbeit konkret verwendeten Stackarchitekturen beschrieben.

2.1 Der Stack

Ein Stack ist zunächst einmal nicht mehr als ein Speicher für Daten samt einer Vorschrift, die den geordneten Zugriff auf die Daten regelt. Der wichtigste Teil dieser Vorschrift betrifft die Reihenfolge, in der auf die Elemente des Speichers zugegriffen werden kann: Es herrscht das Prinzip „last in, first out“, d.h. dasjenige Element, das dem Stack als letztes hinzugefügt wurde, wird auch als erstes wieder entnommen.[1] Die Anzahl der Elemente, die sich im Stack befinden, wird auch *Tiefe* des Stack genannt.

[1] Diese Eigenschaft macht den Stack etwa zur Speicherung von Rücksprungadressen bei Unterprogrammaufrufen sehr geeignet.

Beispiel 2.1 Darstellung des Stack für eine Befehlssequenz



Nicht unähnlich einem Stapel von Tellern kennt ein Stack also zwei grundlegende Zugriffsoperationen:

push Mithilfe dieser Operation kann dem Stack ein neues Element hinzugefügt werden, das „über“ dem bisherigen Inhalt zu liegen kommt, diesen also verdeckt.

pop Der dazu konträre Befehl nimmt das oberste Element vom Stack und ermöglicht somit den Zugriff auf das darunterliegende Datum.

Da diese Basisoperationen natürlich nicht allzu nützlich sind, um Berechnungen durchzuführen, unterstützen Stackmaschinen im allgemeinen eine Menge von arithmetischen und logischen Befehlen. Diese entnehmen ihre Operanden dem Stack und legen das Ergebnis wiederum auf diesem ab. Für einen n-ären Operator sind also nur die n obersten Elemente auf dem Stack interessant, die darunterliegenden werden durch die Operation in keiner Weise verändert.

Neben weiteren notwendigen Befehlen (etwa Sprüngen, Vergleichsoperatoren usw.) erweist es sich als sinnvoll, sogenannte Stackmanipulationsbefehle einzuführen. Diese ermöglichen es, die Reihenfolge der Stackelemente zu verschieben bzw. Kopien von Stackelementen herzustellen. In Beispiel 2.1 werden stellvertretend für diese Klasse von Befehlen `swap` und `dup` verwendet (`swap` vertauscht die zwei obersten Stackelemente, `dup` fügt dem Stack eine Kopie des obersten Elementes hinzu).

Stackmanipulationsbefehle haben meist keine direkte Entsprechung in einer Registermaschine, da auf die in den Registern gespeicherten Operanden ohnehin wahlfrei zugegriffen werden kann und ihre Reihenfolge daher keine Rolle spielt.

Um die Auswirkung eines Befehls auf den Stack kurz und prägnant darzustellen, hat sich in Forth folgende Konvention etabliert: Eingeschlossen in runden Klammern wird der vom Befehl benutzte Teil des Stack vor bzw. nach der Ausführung symbolisch dargestellt (getrennt durch einen doppelten Gedankenstrich). Das Element, das jeweils am weitesten rechts steht, ist dabei als das oberste im Stack anzusehen. Bereiche im Stack, die nicht tangiert werden, entfallen in der Darstellung. Die Addition etwa kann mit `( a b -- a+b )` dargestellt werden, ein `dup` mit `( a -- a a )`. Wir werden diese Konvention im weiteren für eine komprimierte Form der Befehlsbeschreibung verwenden.

2.2 Historischer Abriß

In den späten 50er Jahren schien die Zeit reif für ein neues Konzept in der Entwicklung von Computerarchitekturen. Wie auch bei anderen Erfindungen bzw. Entdeckungen entwickelten mehrere Forscher unabhängig voneinander zur ungefähr gleichen Zeit die Idee einer „Stackmaschine“. Ein ausschlaggebender Faktor mag dabei die Entwicklung und Verbreitung der Programmiersprache Algol gewesen sein – jedenfalls war laut [Bul77] der Wunsch nach effizienter Hardwareunterstützung für diese Sprache lautgeworden.

In den USA war es vor allem Robert Barton ([Bar61]), der bei Burroughs ab etwa 1959 mit der Idee einer Stackarchitektur hervortrat und das Management davon zu überzeugen versuchte. Im Frühling 1960 wurde das Projekt unter Burtons Leitung gestartet und schon 1962 war das erste Modell, die *Burroughs B5000*, samt Betriebssystem und Compiler einsatzbereit. Die ersten kommerziellen Installationen der neuen Maschine ließen nicht lange auf sich warten.

Die B5000 kann mit bis zu 192 KB Speicher umgehen, „virtual memory“, segmentierter Speicherzugriff und Multiprozessorfähigkeiten waren ebenfalls Teil des Designs. Das Betriebssystem und der Algol 60 Compiler waren komplett in Algol 60 geschrieben und fanden in 48 KB Platz – das MCP („master control program“) der B5000 war damit das erste vollkommen in einer Hochsprache geschriebene Betriebssystem.

Etwa zur selben Zeit wie Robert Burton entwarf auch der Australier Charles Hamblin unbeeinflußt von den Anstrengungen in den USA eine Maschine, die nach dem Prinzip der umgekehrt-polnischen Notation arbeitete. In Deutschland hatten sich Bauer und Amelson schon vor 1961 „a specialized stack machine“ ([Bul77]) patentieren lassen; einen weiteren Beitrag zur Entwicklung der Stackarchitekturen leistete Europa mit der Entwicklung der *English Electric KDF9*. Auch hier kam man unabhängig von den Bemühungen in den USA und Australien zu praktischen dem gleichen Ergebnis ([Dun77]).

Hewlett-Packard begab sich schließlich 1972 mit der Entwicklung der HP3000 auf das Gebiet der Stackmaschinen. Diese Architektur zeigt sich sehr von der B5000 beeinflußt, wohl auch, weil einige der Entwickler von Burroughs zu HP gekommen waren. Auch andere Prozessorhersteller (darunter Microdata, Data General und Tandem) folgten dem Trend und brachten Stackprozessoren auf den Markt.

Ab Mitte der 70er Jahre verschwanden Stackarchitekturen langsam von der Bildfläche und wurden von den erfolgreicheren Registermaschinen verdrängt. Um [Hen90] zu zitieren: „Stack-based machines fell out of favor in the late 1970s and essentially disappeared in the 1980s."

Nach einer Zeit des Nischendaseins, in der vor allem die Benutzer von Forth mit Stackarchitekturen zu tun hatten, trat 1995 mit der JavaVM nach zwanzig Jahren erstmals wieder eine Stackarchitektur in das breitere Interesse der Fachwelt. Obwohl natürlich die meisten Benutzer der Programmiersprache Java das Konzept der JavaVM nicht näher kennen (müssen), kommt es seit der Vorstellung dieser Architektur auch innerhalb des Übersetzerbaus zu einer immer stärker werdenden Beschäftigung mit dem Konzept der Stackmaschinen (zu verfolgen etwa in der *Usenet-Newsgroup* `comp.compilers`).

2.3 Konkrete Stackarchitekturen

In Stackmaschinen werden Stacks für die unterschiedlichsten Zwecke verwendet. Forth etwa benutzt unter anderem einen „return stack", der Adressen für den Rücksprung aus Unterprogrammen und Elementen der strukturierten Programmierung (wie Schleifen, Bedingungen, ...) enthält und auch als kurzfristiger Zwischenspeicher dienen kann. Außerdem gibt es in Forth einen „data stack", auf dem die arithmetischen Berechnungen ausgeführt werden, und der auch zur Parameterübergabe dient.

In den Details der unterstützten Befehle unterscheiden sich natürlich die Stackarchitekturen teils recht deutlich von einander: Während ein gewisser Grundvorrat von Befehlen universelle Verwendung findet, gibt es etwa bei den Befehlen für die Stackmanipulation doch große Unterschiede – grundsätzlich lassen sich natürlich alle Befehle auf jeder Maschine in Software emulieren.

[Koo89] teilt einen weitgefaßten Bereich von Architekturen in verschiedene Klassen ein, die eine unterschiedliche Unterstützung des Stackkonzeptes bieten; die Taxonomie beruht auf der Anzahl der Stacks, deren Größe und der Anzahl der expliziten Operanden eines Befehls. In dieser Arbeit wird hauptsächlich die engere Bedeutung von „Stackarchitektur" verwendet, die sich auf die 0-Operanden Adressierung bezieht (auch „pure stack addressing“ genannt). Beinahe alle Befehle beziehen dabei ihre Operanden vom Stack und legen auch ihre Ergebnis wieder dort ab. Der Stack ist somit implizit Quelle und Ziel des Datenflusses.

Daraus folgt, daß der Stack recht schnell zum Flaschenhals des gesamten Systems werden kann und daher eine hohe Zugriffsgeschwindigkeit wünschenswert ist. Viele Stackarchitekturen verwenden „stack buffering“, um dieses Problem zu lösen. Ein Teil des Stack (meist die obersten 4 bis 16 Elemente) wird direkt im „stack buffer“ der CPU gespeichert, die so mit voller Geschwindigkeit darauf zugreifen kann. Die restlichen Stackelemente befinden sich im Hauptspeicher. Da sich die Tiefe des Stack im allgemeinen nicht in allzu großen Sprüngen ändert, treten Lese- und Schreiboperationen zwischen „stack buffer“ und Hauptspeicher nicht mehr so oft auf. In der Fachliteratur werden mehrere Algorithmen für das „stack buffering“ gegenübergestellt und verglichen ([Has85], [Koo89], [Ert95a]).

2.3.1 HYSTAMIN

HYSTAMIN – die „*hy*pothetical *sta*ck *m*ach*in*e“ – ist, wie der Name schon sagt, eine für diese Arbeit „erfundene“ Stackarchitektur. Das Hauptziel beim Design lag auf Einfachheit und Abbildbarkeit von Befehlen auf entsprechenden Zwischencode in SUIF (siehe Abschnitt 6.1). Daneben lehnt sie sich auch etwas an Forth an (was etwa in der Verwendung von zwei Stacks deutlich wird). Im folgenden wird eine kurze Übersicht über die Maschine gegeben, soweit dies für das Verständnis der folgenden Kapitel nötig ist.

Die Maschine besitzt zwei Stacks. Der Datenstack („D-Stack“) enthält die impliziten Operanden für die meisten Instruktionen (insbesondere die arithmetischen Befehle), der Returnstack („R-Stack“) wird für Rücksprungadressen und als Zwischenspeicher des „framepointers“ verwendet.

Die Länge eines Wortes ist 32 Bit, alle arithmetischen Befehle arbeiten mit Operanden der Länge 32 Bit, der Speicherzugriff kann jedoch auch mit 8 oder 16 Bit erfolgen. Es werden nur Integer-Befehle unterstützt, Fließkommazahlen können nicht verarbeitet werden.

Der Speicher wird linear adressiert; die Maschine besitzt folgende zwei Register:

IP zeigt auf den nächsten auszuführenden Befehl im Speicher.

FP zeigt auf den aktuellen „stackframe“ im Speicher.

Die Maschine unterstützt unter anderem die Befehle aus Tabelle 2.1. Das virtuelle Array `mem` steht dabei für den Hauptspeicher.

Tabelle 2.1 Befehlssatz von HYSTAMIN (Auswahl)

Arithmetische Befehle	
`add`	`( a b -- a+b )`
`sub`	`( a b -- a-b )`
Sprungbefehle	
`jmp L`	`( -- ) [IP=L]`
`jz L`	`( a -- ) [if a=0 then IP=L]`
`call L`	`[rpush IP; IP=L]`
`ret`	`[rpop IP]`
Zugriff auf den Speicher	
`lod`	`( l -- mem[l] )`
`str`	`( d l -- ) [mem[l]=d]`
Bedingte Befehle	
`sne`	`( a b -- x ) [if a=b then x=0 else x=1]`
`sl`	`( a b -- x ) [if a<b then x=0 else x=1]`
Stackmanipulationsbefehle	
`push x`	`( -- x )`
`pop`	`( a -- )`
`dup`	`( a -- a a )`
`swap`	`( a b -- b a )`
`over`	`( a b -- a b a )`
`tuck`	`( a b -- b a b )`
`under`	`( a b -- a a b )`
Framepointer-Befehle	
`getfp`	`( -- a ) [a=fp]`
`addfp`	`( a -- ) [fp=fp+a]`

Lokale Variablen und Parameter einer Funktion werden im Speicher auf dem (virtuellen, nur in Software emulierten) „C-Stack“ gehalten. Der Zeiger auf den „stackframe“ einer Funktion ist im Register FP enthalten und wird während eines Funktionsaufrufes auf dem Returnstack zwischengespeichert. Er zeigt auf die erste lokale Variable im „C-Stack“ (Parameter werden also mit negativem Offset vom framepointer adressiert).

2.3.2 JavaVM

Die detaillierte Beschreibung dieser virtuellen Maschine ist [Sun95] zu entnehmen. Das Hauptziel beim Design der Architektur war es, einerseits Portabilität für eine möglichst große Anzahl von Computern zu erreichen, andererseits eine sichere, objektorientierte Programmiersprache effizient ausführen zu können.

Die JavaVM besitzt nur einen Stack, der Elemente der Breite 32 Bit aufnimmt. Datentypen der Größe 64 Bit (`long` und `double`) werden durch zwei aufeinanderfolgende Stackelemente repräsentiert. Da die Stackeinträge keine Typinformation besitzen, existiert in der JavaVM für jeden Operandentyp ein eigener Befehl; so gibt es den Befehl `iadd`, der zwei 32 Bit Ganzzahlen addiert, während `fadd` eine Addition von zwei 32 Bit Fließkommazahlen bewirkt.

Jede Methode (also Funktion eines Objektes oder einer Klasse) besitzt eine Menge von lokalen Variablen (ebenfalls mit einer Breite von 32 Bit), die über ihren Index (von 0 bis *n*) in der Tabelle der lokalen Variablen angesprochen werden; auch Parameter finden ihren Platz in dieser Tabelle. Die für diese Arbeit wichtigsten Befehle der JavaVM sind in Tabelle 2.2 dargestellt.

Tabelle 2.2 Befehlssatz der JavaVM (Auswahl)

Konstanten	
iconst_*n*	(-- *n*) [0<=*n*<=5]
Arithmetische Befehle	
iadd	(a b -- a+b)
isub	(a b -- a-b)
imul	(a b -- a*b)
idiv	(a b -- a/b)
Sprungbefehle	
ifle *addr*	(a --) [if a<=0 then goto addr]
invokestatic *addr*	(... --) [Methodenaufruf]
Zugriff auf den Speicher	
iload_*n*	(-- a) [a=variable mit index *n*]
istore_*n*	(a --) [variable mit index *n*=a]
iinc *n* *c*	(--) [variable mit index *n* += *c*]
Stackmanipulationsbefehle	
pop	(a --)
pop2	(a b --)
dup	(a -- a a)
dup2	(a b -- a b a b)
dup_x1	(a b -- b a b)
dup2_x1	(a b c -- b c a b c)
dup_x2	(a b c -- c a b c)
dup2_x2	(a b c d -- c d a b c d)
swap	(a b -- b a)

Kapitel 3

Allgemeine Optimierungen

DIVIDE ET IMPERA

Römisches Sprichwort

Für spezielle Optimierungen zur Vermeidung von Zugriffen auf lokale Variablen sind einige grundlegende Vorarbeiten nötig. Die Analyse der Lebensdauer von lokalen Variablen gehört ebenso dazu wie das Aufteilen des Codes in „basic blocks" und ein „peephole optimizer". Dieses Kapitel enthält neben einer Darstellung des prinzipiellen Ablaufes der Optimierung die Beschreibung dieser Teile.

3.1 Allgemeines

Alle Phasen der Optimierung laufen in einzelnen Durchgängen oder „passes" ab. Die Eingabe für einen pass besteht aus jeweils einer Funktion bzw. Methode (in JavaVM Terminologie) des Codes. Es werden also die Funktionen der Reihe nach behandelt. Eine typische Optimierung erfolgt in folgenden Schritten:

1. Die Einteilung des Codes in basic blocks sowie der Aufbau des Datenflußgraphen ergibt die Grundlage der weiteren Transformationen. Dieser Teil der Optimierung ist relativ einfach, allerdings treten bei JavaVM-Code einige Komplikationen auf, die auch Auswirkungen auf spätere Teile der Optimierung haben. Eine genauere Beschreibung dieser Schwierigkeiten findet sich im nächsten Abschnitt.

2. Jeder basic block wird mittels eines Durchlaufes des „peephole optimizers" auf einfache Befehlsmuster untersucht, die eine beschränkte Optimierung zulassen. An dieser Stelle dient der pass vor allem dazu, die Eingabe in eine relativ regelmäßige Form zu bringen.

3. Die Analyse der Lebensdauer von lokalen Variablen ermittelt für jede Variable den Punkt im Code, an dem der Wert der Variable garantiert zum letzten Mal gebraucht wird. Als Ergebnis dieser Phase wird der Befehl, an dem die Lebensdauer einer Variable endet, mit einer Anmerkung über diese Tatsache versehen. Das kann im weiteren Verlauf der Optimierung z.B. dazu verwendet werden, unnötiges Speichern von lokalen Variablen zu unterdrücken.

4. Die eigentliche Optimierung erfolgt nach der von Phil Koopman entwickelten Methode des „stack scheduling“ (näheres dazu im nächsten Kapitel) oder mittels optimalem „DAG scheduling“, das in Kapitel fünf behandelt wird.

5. Ein weiterer Durchlauf des peephole optimizers beseitigt unnötige Stackmanipulationen, die durch die vorhergehende Optimierung entstanden sein könnten, und versucht damit, das Endergebnis weiter zu vereinfachen.

6. Schließlich muß der Code für die Funktion oder Methode wieder linearisiert werden, d.h. die Befehle der basic blocks müssen wieder so in eine Sequenz zusammengefügt werden, daß die Semantik des ursprünglichen Programmes erhalten bleibt.

In den weiteren Abschnitten dieses Kapitels wenden wir uns den weiter oben angeschnittenen Phasen der Optimierung mit Ausnahme der in den folgenden Kapiteln behandelten Algorithmen zu.

3.2 Ermittlung der basic blocks

Da sich die vorliegende Arbeit auf lokale Optimierungstechniken konzentriert, ist es vor der eigentlichen Optimierung nötig, die Eingabe (also den Code für eine Funktion bzw. Methode) in eine Folge von „basic blocks“ aufzuspalten, die folgende Eigenschaft besitzen:

- Kommen die Befehle eines basic blocks im Zuge des Programmablaufes zur Ausführung, so ist diese linear, d.h. daß sie vom ersten bis zum letzten Befehl des basic blocks sequentiell fortschreitet.

Diese Eigenschaft wird durch folgende Bedingungen sichergestellt (sie geben damit gleichzeitig eine Methode an, wie die Grenzen der basic blocks sehr einfach gefunden werden können):

- Außer dem ersten ist kein Befehl eines basic blocks das Ziel für einen Sprungbefehl (es ist also nicht möglich, „mitten in einen basic block“ zu springen, sehr wohl aber, an dessen Anfang).

- Nur der letzte Befehl eines basic block darf die Befehlsausführung dahingehend verändern, daß nicht der darauffolgende Befehl als nächster ausgeführt wird (Sprünge gibt es damit nur vom Ende zum Anfang von basic blocks). Auch Befehle, die nur unter bestimmten Bedingungen den Kontrollfluß ändern (wie etwa bedingte Sprungbefehle) beenden einen basic block.

Funktions- bzw. Methodenaufrufe ändern zwar auch den Kontrollfluß des Programmes, sie dürfen aber trotzdem mitten in basic blocks vorkommen. Weil sich die Optimierung auf lokale Variablen beschränkt, die nicht über Zeiger verändert werden können, dürfen Funktionsaufrufe wie normale, den Kontrollfluß nicht verändernde Befehle behandelt werden – dadurch vergrößern sich die basic blocks und auch die Möglichkeiten zur Optimierung.

Für die Analyse der Lebenszeit von Variablen ist zusätzlich zu den basic blocks noch die Information über den Datenfluß zwischen diesen Blöcken nötig. Nachdem die Grenzen der basic blocks ermittelt wurden, muß daher noch der Datenflußgraph für die Funktion bzw. Methode aufgebaut werden. Die basic blocks bilden dabei die Knoten des Graphen und zwei Knoten werden durch eine gerichtete Kante genau dann verbunden, wenn zur Laufzeit ein Sprung vom Ende des basic blocks im ersten zum Beginn des basic blocks im zweiten Knoten auftreten kann. Zusätzlich erweist es sich für die Analyse der Lebenszeit von Variablen als nützlich, dem Datenflußgraphen einen leeren Start- bzw. Endknoten hinzuzufügen, der Ausgangs- bzw. Endpunkt aller möglichen Ausführungen des Codes für die entsprechende Funktion oder Methode ist.

Es ist eine angenehme Eigenschaft von basic blocks (zumindest für HYSTAMIN), daß der Stack zu Beginn und Ende ihrer Ausführung leer ist. Somit kann jeder basic block für die weiteren Optimierungen als abgeschlossene Einheit behandelt werden und ist damit Ausgangspunkt der entsprechenden Algorithmen.

Leider trifft diese Aussage nicht in vollem Umfang auf den Code für die JavaVM zu. Einerseits erschwert hier die in der Architektur vorgesehene Ausnahmebehandlung die Optimierung, weil beim Sprung auf eine Ausnahmebehandlungsroutine der Stack implizit gelöscht und danach mit einem Element, das die Art der Ausnahme angibt, beschrieben wird. Der erste basic block einer solchen Routine beginnt also *nicht* mit einem leeren Stack. Gleiches gilt für Code, der durch `finally` als ein in jedem Fall auszuführender gekennzeichnet ist, nur daß hierbei die Rücksprungadresse auf dem Stack zu liegen kommt.

Außerdem erzeugt der verwendete Java-Compiler für bedingte Ausdrücke (zum Beispiel „`x = y + (a > b) ? a : b;`“) basic blocks, die zu Beginn ihrer Ausführung bereits ein oder mehrere Elemente auf dem Stack erwarten (analog müssen die Vorgänger dieser Blöcke im

Datenflußgraphen die entsprechenden Elemente am Ende der Ausführung auf dem Stack haben). Diese Tatsache führt vor allem beim optimalen DAG scheduling zu Problemen, auf die in Abschnitt 5.3.4 näher eingegangen wird.

Größenverteilung der basic blocks

Die Anzahl der Befehle, die in einem basic block enthalten sind, ist aus folgenden Gründen nicht unwichtig:

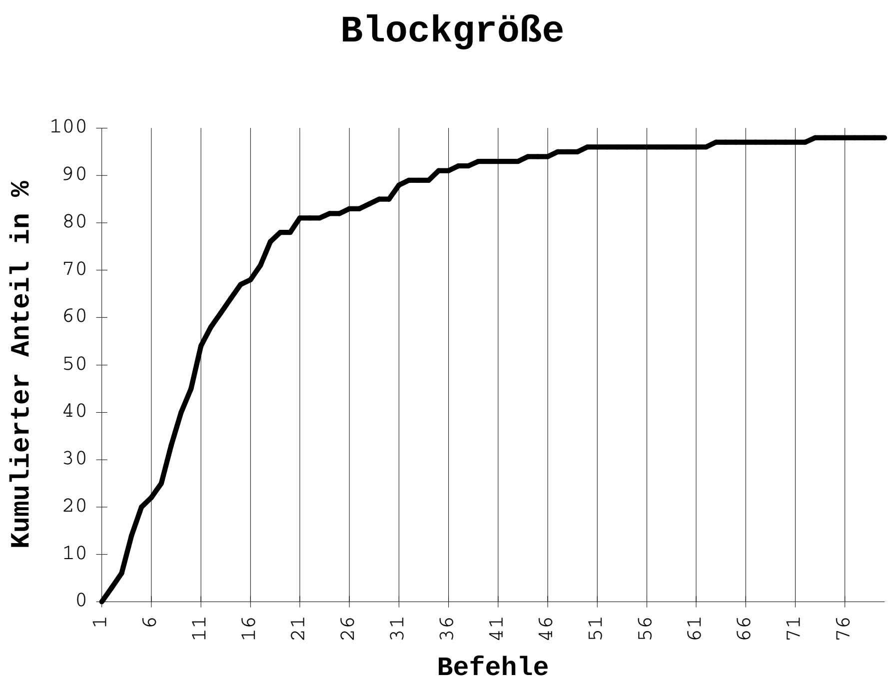
- Aus je mehr Befehlen ein basic block besteht, desto mehr Möglichkeiten zur Optimierung bestehen, weil ja ein basic block die Ebene ist, auf der die Optimierung durchgeführt wird. Viele kleine Blöcke bedeuten demnach, daß lokale Optimierungen nicht sehr effizient sein können und eher globale Verfahren zum Einsatz kommen sollten.
- Andererseits erfordert ein „großer“ basic block auch einen umfangreichen DAG, was wiederum den Aufwand für das in Kapitel 5.2 vorgestellte Verfahren enorm in die Höhe treibt und das Auffinden einer Lösung eventuell verhindert. Eine erschöpfende Suche ist ab einer gewissen Größe des DAG einfach nicht mehr mit vertretbarem Aufwand machbar. Aber auch die Laufzeit der anderen Methode („stack scheduling“, siehe Kapitel vier) steigt natürlich mit der Größe der basic blocks (nähere Informationen zum Aufwand des Algorithmus sind in Abschnitt 4.3.1 zu finden).

Für die beiden in dieser Arbeit behandelten Architekturen wird im folgenden für je eine repräsentative Eingabe die Verteilung der verschiedenen Blockgrößen dargestellt. Ein Überblick über die Eingaben ist in Abschnitt 6.1 enthalten.

Abbildung 3.1 gibt die Größenverteilung der basic blocks für den Stanford-Benchmark in HYSTAMIN-Code an. Dabei sind auf der X-Achse die Blöcke der Größe nach angeordnet und auf der Y-Achse ist der kumulierte Anteil in Prozent an der Gesamtzahl der basic blocks aufgetragen. Aus der Abbildung ist ersichtlich, daß rund zwei Drittel der basic blocks weniger als 15 Befehle enthalten. Dabei ist zu beachten, daß z.B. eine Ladeoperation für eine lokale Variable schon bis zu vier Befehle benötigt (der Offset der Variablen muß auf den Stack gebracht und mit dem Inhalt des framepointers addiert werden; danach kann die Variable von der hiermit ermittelten Adresse im Speicher geladen werden).

Auffallend ist auch, daß etwas mehr als ein Zehntel der basic blocks über 30 Befehle enthält und immerhin noch 2% der Blöcke aus mehr als 80 Befehlen bestehen.

Abbildung 3.1 Größe der basic blocks für den Stanford-Benchmark

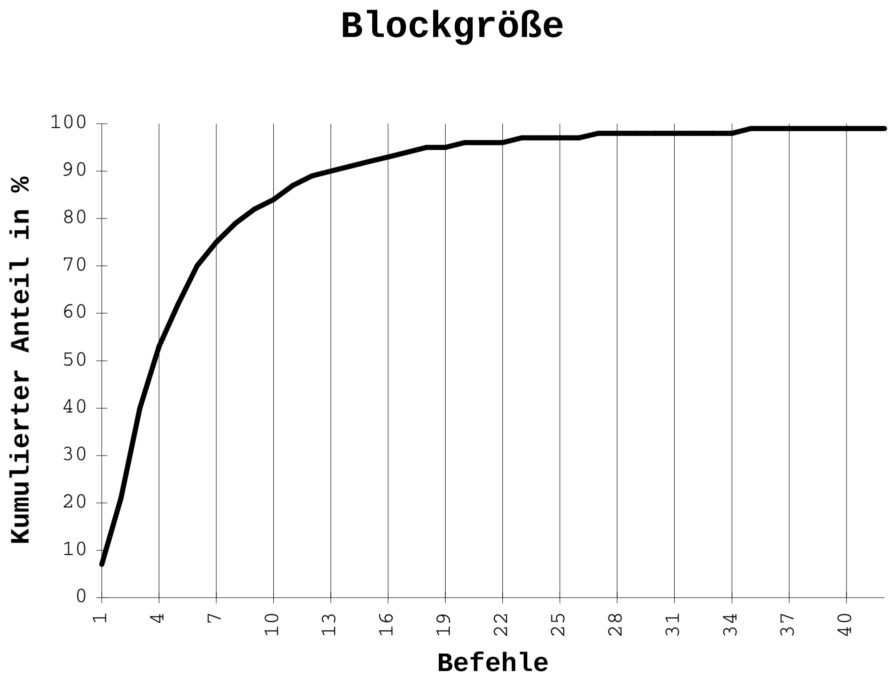


Die Verteilung der basic blocks im Code für die JavaVM (Abbildung 3.2 zeigt sie für die Klassenbibliothek des JDK) sieht doch etwas anders aus: Allein über die Hälfte der basic blocks besitzt weniger als fünf Befehle, vier Fünftel der basic blocks bestehen aus weniger als zehn Befehlen. Große Blöcke sind selten, nur etwa drei Prozent bestehen aus 25 oder mehr Befehlen, basic blocks mit über 40 Befehlen treten kaum auf.

Der Befehlssatz der JavaVM scheint also dazu zu tendieren, kompakten Code zu erzeugen (was auch durch die Designziele für diese Architektur erklärbar ist); außerdem ist der von Java favorisierte Programmierstil offensichtlich „modular" in der Hinsicht, daß eher viele kurze Funktionen verwendet werden.

Es besteht damit ein doch deutlicher Unterschied in der Verteilung der Blockgröße bei den beiden Stackarchitekturen. Der Hauptgrund dafür ist, daß der Befehlssatz von HYSTAMIN im Vergleich zu dem der JavaVM recht primitiv ist. So benötigen Zugriffe auf lokale Variablen in HYSTAMIN mehrere Befehle, während die JavaVM mit nur je einem Befehl pro Lese- und Schreiboperation das Auslangen findet.

Abbildung 3.2 Größe der basic blocks für die Klassen des JDK



3.3 Analyse der Lebenszeit von lokalen Variablen

Diese Phase der Optimierung erzeugt an sich keine Verbesserung des Codes sondern nur Informationen, die von späteren Durchläufen genutzt werden können. Konkret handelt es sich dabei um den Punkt im Code einer Funktion bzw. Methode, an dem feststeht, daß der Wert einer lokalen Variable im weiteren Verlauf der Ausführung sicher nicht mehr gebraucht werden wird. Eine Variable wird ab diesem Zeitpunkt als „dead variable“ bezeichnet, und Schreiboperationen auf sie können unterdrückt werden, da sie einen Wert schreiben, der nie wieder gelesen wird.

Der in dieser Arbeit implementierte pass verwendet zur Lösung des Problems einen iterativen Algorithmus, der im Rahmen einer generellen Lösung von Datenflußgleichungen arbeitet ([Aho86]).

Nachdem für jeden basic block die Menge der Variablen errechnet wurde, die zum letzten Mal in ebendiesem Block verwendet werden, wird für jede der dead variables der letzte Befehl im basic block ermittelt, der darauf zugreift – üblicherweise ein Ladebefehl. Der Befehl wird mit einer speziellen Anmerkung („dead“) versehen, die von den weiteren Phasen der Optimierung genutzt werden kann.

3.4 Der peephole optimizer

Eine im Übersetzerbau weit verbreitete Optimierungstechnik ist die des „peephole optimizers“. Dabei werden kurze Abschnitte des Codes auf Möglichkeiten zu einfachen Verbesserungen untersucht. Da dieser Ansatz immer nur einen kleinen Ausschnitt betrachtet, der durch den Code wandert, ist der Terminus „peephole“ (Guckloch) durchaus angebracht.

In vorliegender Arbeit wurde ein flexibler, tabellengesteuerter pass implementiert, der für diese Optimierung zuständig ist und im Prinzip nur ein ausgefeiltes Suchen und Ersetzen von Codeteilen darstellt. In einer Tabelle wird dabei eine Menge von Regeln gespeichert, die den Mechanismus des Suchens und Ersetzens beschreiben. Eine Regel besteht aus folgenden Teilen:

- Eine Folge von Befehlen und/oder Befehlsklassen, nach der in der Eingabe gesucht wird. Befehlsklassen fassen verwandte Befehle zusammen und ermöglichen somit eine kompakte Tabelle.
- Optional eine Funktion, welche als Parameter die Befehle im ursprünglichen Code erhält, weitere Bedingungen prüfen kann und dem pass mitteilt, ob die Regel tatsächlich anzuwenden ist. Außerdem kann die Funktion dem Teil, der die Ersetzung durchführt, wichtige Informationen zuführen.
- Eine Menge von Befehlen, die zur Ersetzung der ursprünglichen Befehle verwendet werden.
- Eine Anleitung, wie die Folge von Befehlen in der Eingabe optimiert wird: Befehle können gelöscht oder verändert, neue Befehle hinzugefügt werden.

Neue Regeln können sehr einfach in die Tabelle aufgenommen werden und stehen für die Optimierung danach sofort zur Verfügung. Diese geht alle Befehle der Eingabe sequentiell durch und sucht in der Tabelle nach einer Regel, deren Vergleichsmuster mit der Eingabe übereinstimmt und deren Funktion ihr Einverständnis für eine Ersetzung signalisiert. Wird eine derartige Regel gefunden, so wird die Anleitung zur Optimierung befolgt und es entsteht eine neue Folge von Befehlen, die wiederum auf Anwendbarkeit weiterer Regeln durchsucht werden kann.

Ist an einer Stelle der Eingabe keine passende Regel auffindbar, so wird die Suche beim nächsten Befehl fortgesetzt; der untersuchte Abschnitt „gleitet“ also durch die Eingabe und wird sukzessive optimiert, bis der gesamte Code durchsucht wurde.

Tabelle 3.1 Regeln für den peephole optimizer (Auswahl)

Befehlsmuster	ersetzt durch	Anwendungen Ohne	Anwendungen Koop.	Anwendungen Opt.
Stackmanipulationen				
`swap swap`	-	0	25	7
`dup_x1 pop`	`swap`	0	46	16
Variablenzugriffe				
`load_x load_x`	`load_x dup`	158	158	158
`store_x load_x`	`dup store_x`	105	105	106
`dup store_x ('dead')`	-	5	52	17
`store_x ('dead')`	`pop`	3	52	22
Kommutativität				
`swap add`	`add`	0	3	0
`swap mul`	`mul`	0	1	0
`swap if_??`	`if_??` (kommutativ)	0	4	0
Arithmetische Eigenschaften				
`const_x const_y add`	`const_z (z=x+y)`	0	0	0
`const_0 add`	-	0	0	0

Tabelle 3.1 enthält eine Auswahl der Regeln, die im peephole optimizer eingesetzt wurden. Die letzten drei Spalten geben an, wie oft die Regeln bei der Optimierung der JGL-Klassen (siehe Abschnitt 6.1, die Eingabe enthält insgesamt etwa 21.000 JavaVM-Befehle) zur Anwendung kamen. „Ohne" bedeutet dabei, daß keine der Optimierungsmethoden aus den nächsten zwei Kapitel verwendet wurden, „Koop." steht für Koopmans stack scheduling (Kapitel vier) und „Opt." für das optimale DAG scheduling aus Kapitel fünf. Die Regeln des peephole optimizers können in vier grundlegende „Familien" eingeteilt werden:

Stackmanipulationen Bestimmte Stackmanipulationen heben einander auf, andere können zusammengefaßt und dabei vereinfacht werden. Kombinationen wie etwa `swap swap` können im Zuge der Optimierungen entstehen, und es ist einfacher, sie im peephole optimizer zu eliminieren, als schon während der eigentlichen Optimierung darauf zu achten.

Variablenzugriffe Einfache Muster von Zugriffen auf lokale Variablen können schon im peephole optimizer effizienter gestaltet werden, so kann zweimaliges Lesen bzw. Schreiben und sofortiges Lesen derselben Variable vereinfacht werden. Auch Schreibzugriffe auf Variable, die im Programmverlauf nicht mehr verwendet werden, können auf sehr einfache Weise eliminiert werden. Wieder gilt, daß der peephole optimizer für diese Aufgabe gut geeignet ist und sich die anderen Algorithmen so auf die eigentliche Optimierung konzentrieren können.

Kommutativität Die Kommutativität von Operatoren und Vergleichsbefehlen kann dazu benutzt werden, einen `swap`-Befehl, der ihnen direkt vorangeht, zu eliminieren. Das stack scheduling kann also unbekümmert `swap`-Befehle einfügen, sollten sie redundant sein, sorgt der peephole optimizer für ihre Entfernung. Das DAG scheduling nutzt ohnehin die Kommutativität von Operatoren aus, um unnötige Stackmanipulationen zu vermeiden.

Arithmetische Eigenschaften Unter diesen Titel fallen Regeln, welche einfache Operationen auf Zahlen möglichst zur Kompilierzeit ausführen und nicht erst zur Laufzeit. Obwohl schon die Eingabe von SUIF bzw. `javac` in dieser Hinsicht behandelt ist, können während der Optimierung erneut solche Fälle entstehen, die vom peephole optimizer behandelt werden können. Vor allem die Berechnung von Adressen lokaler Variablen in HYSTAMIN-Code profitiert von diesen Regeln.

Kapitel 4

Frühere Methoden und stack scheduling

Great discoveries and improvements invariably involve the cooperation of many minds. I may be given credit for having blazed the trail but when I look at the subsequent developments I feel the credit is due to others rather than to myself.

Alexander Graham Bell

Dieses Kapitel beschreibt eine Methode zur Optimierung von Stackcode, die von Phil Koopman 1992 zum erstenmal publiziert wurde: das „stack scheduling“ ([Koo92]). Die Grundidee hinter seinem Vorschlag besteht darin, das Laden von lokalen Variablen auf den Stack zu eliminieren, falls sich der Wert der Variable bereits zu einem früheren Zeitpunkt dort befunden hat. Es wird versucht, diesen Wert wiederzuverwenden, statt die Variable erneut aus dem Speicher zu laden.

Bevor die Methode im Detail vorgestellt und erläutert wird, sollen noch kurz einige Arbeiten beleuchtet werden, die vor Koopmans Publikation liegen und sich mit der Optimierung von Stackcode (zum Teil aus anderem Blickwinkel) beschäftigen.

Nach der genaueren Darstellung des stack scheduling wenden wir uns der Implementierung zu. Einer Analyse des Aufwands für die Optimierung folgen Bemerkungen zur Implementierung für die beiden konkreten Architekturen. Die mit diesem Ansatz erzielbaren Ergebnisse werden in Kapitel sechs beleuchtet.

4.1 Frühere Arbeiten

Eine frühe Arbeit zum Thema Optimierung für Stackmaschinen ist [Bru75]. Allerdings konzentrieren sich die Autoren auf die Problematik, eine optimale Reihenfolge für die Berechnung eines arithmetischen Ausdruckes und seiner Teilausdrücke in einer Stackarchitektur mit beschränkter Stacktiefe zu finden. Für sehr eingeschränkte Ausdrücke (die im wesentlichen durch Bäume ohne zusätzliche Abhängigkeitskanten darstellbar sind) gibt der Artikel einen effizienten Algorithmus an, um eine optimale Reihenfolge zu finden.

Auch [Pra80] beschäftigt sich vornehmlich mit arithmetischen Ausdrücken und optimalem Code dafür in bezug auf die Stacktiefe: ermittelt wird die minimale Tiefe eines Stack, um den entsprechenden Ausdruck berechnen zu können. Die als Beispiel herangezogene Architektur ist keine „lupenreine" Stackmaschine, wie sie in Abschnitt 2.3 definiert wurde (z.B. können arithmetische Befehle einen Operanden direkt aus dem Hauptspeicher beziehen).

Wie schon der Titel von [Tan82] („Using Peephole Optimization on Intermediate Code") andeutet, legen die Autoren ihr Hauptaugenmerk auf den peephole optimizer. Der verwendete Stackcode wird von einem portablen Compiler als Zwischencode benutzt, eine Optimierungsstrategie für diesen Code ist also unabhängig sowohl vom front end des Compilers (der Quellsprache) als auch vom back end (der eigentlichen Maschinensprache). Wie der in vorliegender Arbeit verwendete peephole optimizer (siehe Abschnitt 3.4) ist auch der von Tanenbaum beschriebene tabellengesteuert und daher relativ flexibel. Der Artikel enthält u.a. eine Aufstellung der für eine große Menge an Pascal Quellcode am häufigsten verwendeten Regeln des peephole optimizers.

Eine verwandte Arbeit auf diesem Gebiet ist [Mas80]. Die Autoren beschreiben die Optimierungen in einem Konverter, der einen Lisp-Dialekt in Code für eine abstrakte Stackmaschine umwandelt. Auch die in diesem Artikel verwendeten Methoden konzentrieren sich hauptsächlich auf den peephole optimizer und bringen nicht allzuviele neue Aspekte.

In [Mil87] findet sich die Vorstellung eines C-Compilers, der Code für die NOVIX NC4016/6016 Prozessoren erzeugt. Der Compiler verwendet die zum Teil sehr spezifischen Features dieser Prozessoren, um die Ausführung zu beschleunigen. So werden zum Beispiel „Pseudoregister" für den Zeiger auf einen stackframe (siehe auch Abschnitt 2.3.1) benutzt. Außerdem verwendet der Compiler eine Technik, die man als das Gegenteil des „inlining" in C++ bezeichnen kann: Gleichlautende Codeteile werden nur einmal erzeugt und im Programm mehrfach wie eine Funktion aufgerufen. Da Funktionsaufrufe in Stackmaschinen nur wenig Aufwand verursachen, ist das auch effizient.

Lokale Variablen hingegen werden zumindest in den angegebenen Beispielen nicht auf dem Stack gespeichert, sondern immer wieder vom Hauptspeicher nachgeladen. Die Parameter und der Rückgabewert einer Funktion werden hingegen sehr wohl auf dem Stack übergeben (zum Vergleich: HYSTAMIN übergibt die Parameter nicht auf dem Datenstack) und erst in der Funktion in den Hauptspeicherbereich geschrieben, der den stackframe der Funktion beinhaltet.

[Win88] beschreibt einen ähnlichen Compiler, der C-Code in Code für einen Prozessor (den APD MF1600) mit Forth-ähnlichem Maschinencode übersetzt. Allerdings werden die eigentlichen Algorithmen für die Übersetzung nicht angegeben, der Leser erfährt nur allgemein, daß lokale Variablen immer auf dem Stack und nicht im Hauptspeicher präsent sind (sofern sie einigen Bedingungen, etwa in bezug auf Größe oder „aliasing" genügen). Auch Parameter sowie Rückgabewerte von Funktionen werden über den Datenstack übergeben, wiederum sind jedoch keine weiteren Einzelheiten zu erfahren.

Aus den Beispielen im Artikel ist jedoch ersichtlich, daß der erzeugte Code exzessiven Gebrauch von `pick`, `rot`[1] und weiteren Stackmanipulationsbefehlen macht. Konkret wird der erzeugte Code für die C-Funktion aus Beispiel 4.1 beschrieben.

Beispiel 4.1 Beispielscode aus [Win88]

```
char *strcpy(char *a, const char *b)
{
    char *p = a;
    while ((*p++ = *b++) != 0);
    return a;
}
```

Für die innere Schleife aus dem Beispiel erzeugt der Compiler aus [Win88] Maschinencode mit 8 Stackmanipulationsbefehlen und 6 Befehlen, die konkrete Berechnungen durchführen. Ein vergleichbares Beispiel für die JavaVM kommt nach der Optimierung mit Koopmans Algorithmus (im nächsten Abschnitt beschrieben) auf 3 Stackmanipulationen und 9 andere Befehle.

4.2 Koopmans Vorschläge

Phil Koopman schlägt in seinem Artikel „A Preliminary Exploration of Optimized Stack Code Generation" ([Koo92]) zwei Methoden vor, die hauptsächlich auf die Eliminierung von Zugriffen auf lokale Variablen abzielen:

[1]`pick (...n -- ...x)` kopiert das `n`te Element im Stack an die oberste Position, `rot (a b c -- b c a)` verschiebt das drittoberste Element an den top of stack.

1. „intra-block stack scheduling“ und
2. „global scheduling“.

Während der Artikel den Algorithmus für das lokale stack scheduling recht genau und umfassend beschreibt, bleibt die Beschreibung des globalen scheduling eher vage und verweist auf händische Optimierung, ohne einen schlüssigen Ansatz dafür liefern zu können.

Für seine Untersuchungen geht Phil Koopman in seinem Artikel von folgenden Überlegungen aus:

- Code für Registermaschinen legt temporäre Variablen, die im Lauf der Berechnungen benötigt werden, üblicherweise in Registern statt im Speicher ab. Der große Vorteil dieser Vorgehensweise entsteht dadurch, daß der Prozessor auf seine Register sehr viel schneller zugreifen kann als auf den Speicher.

  Die Situation verschärft sich noch für modernere Prozessorarchitekturen, die superskalar und/oder pipelined arbeiten und daher häufiger und schneller auf Daten zugreifen müssen. Da die Geschwindigkeit der Prozessoren viel schneller steigt als die des Hauptspeichers, geht diese „Schere“ immer weiter auf, und erst recht Multiprozessorsysteme benötigen eine hohe Speicherbandbreite. Caches können diese Probleme nur zum Teil lösen und erzeugen ihrerseits weitere Komplexität. Näheres zu diesem Themenkomplex ist unter anderem in [Hen90] zu finden.

- Stackarchitekturen wird häufig zur Last gelegt, daß dieser Ansatz (nämlich temporäre Variablen in einem Bereich zu speichern, auf den schneller zugegriffen werden kann – und das ist für Stackmaschinen natürlich der Stack) zu einer großen Zahl von zusätzlichen Stackmanipulationen führt, weil auf den Stack nicht wahlfrei zugegriffen werden kann wie auf Register.

  Die Statistiken scheinen dieser Kritik auch Recht zu geben; einer Tabelle der dynamischen Häufigkeit von Forth-Wörtern in [Koo89] ist unter anderem zu entnehmen, daß ungefähr 25% der ausgeführten Wörter keine „nützlichen Berechnungen“ durchführen, sondern für Stackmanipulationen sorgen (wie dies z.B. `dup`, `over`, `swap` oder `rot` tun). Weitere 23% der ausgeführten Befehle dienen dazu, Daten zwischen Stack und Speicher auszutauschen.[2]

Die Frage ist demnach, ob für Stackarchitekturen eine Methode existiert, mit der temporäre Werte (die üblicherweise in lokalen Variablen

[2] [Hen90] enthält ähnliche Daten für Registermaschinen. Code für einen repräsentativen RISC-Prozessor besteht dabei zur Laufzeit aus etwa 29% „load/store“-Befehlen. Der Datentransfer zwischen Registern fällt unter die 20% Additionsbefehle und kann daher nicht genau angegeben werden.

abgelegt sind) genauso effizient behandelt werden können wie in Registermaschinen.

Experimentelle Ergebnisse sollen helfen, diese Frage zu beantworten. Phil Koopman beschreibt dazu des weiteren die Umgebung, in der er seine Methoden implementiert und erprobt hat. Er bedient sich dabei des GNU C Compilers und erzeugt aus dessen Zwischencode in folgenden Schritten Forth-Code:

1. Der Zwischencode des GCC vor der Registerbelegung ist der Ausgangspunkt der weiteren Schritte. Dieser Code wird eingelesen und die (unbeschränkten virtuellen) Register auf „lokale Variable“ in einem stackframe im Speicher abgebildet. Operationen auf den Registern werden in ein Laden der entsprechenden Variablen auf den Stack, der Operation und schließlich dem Speichern in die Ergebnisvariable konvertiert. Zusätzlich wird der aktuelle Zustand des Stack für alle Operationen errechnet und gespeichert sowie Informationen über dead variables, die der GCC liefert, verarbeitet.

2. Weitere Konvertierungen sind nötig, um den Code auf eine Stackarchitektur zu übertragen. Unter anderem werden die Parameterübergabe und die Rückgabe von Funktionsergebnissen in dieser Phase auf den Stack verlagert.

3. Ein peephole optimizer (siehe auch Abschnitt 3.4) soll die Struktur des erzeugten Codes für die weitere Bearbeitung vereinheitlichen und einfache Optimierungen durchführen.

4. Das stack scheduling versucht, Zugriffe auf die im ersten Schritt erzeugten lokalen Variablen zu minimieren. Dieser Schritt ist der entscheidende und wird in den nächsten Abschnitten detaillierter behandelt.

5. Ein zweiter Durchlauf des peephole optimizers kann „ungeschickte“ Stackmanipulationen, die im Verlauf des stack scheduling enstehen, reparieren (z.B. wird aus der Befehlsfolge `dup swap` das einfachere `dup`; `swap swap` kann gänzlich eliminiert werden). Außerdem wird hier das Speichern einer lokale Variable, die als „dead“ markiert wurde, unterdrückt.

6. Der letzte Schritt in Koopmans Implementierung konvertiert den Stackcode in die Befehle der Zielarchitektur (in seinem Beispiel Forth, das um einige zusätzliche Wörter – vorwiegend für die effiziente Stackmanipulation – ergänzt wurde).

4.2.1 Lokales stack scheduling

Das Ziel dieser Optimierungsstrategie ist, das Laden und Speichern von lokalen Variablen auf den und vom Stack zu minimieren. Stattdessen sollen Kopien der Werte dieser Variablen auf dem Stack für die Berechnungen herangezogen und diese dadurch beschleunigt werden. Die Bezeichnung „stack scheduling“ wurde von Phil Koopman in Anlehnung an ein von ihm so genanntes „register scheduling“[3] für Registermaschinen gewählt, da hier wie dort die effiziente Zuteilung von Variablen an Stackelemente bzw. Register im Vordergrund steht.

Neben der Beschränkung auf einen nicht wahlfreien Zugriff ist für Stackarchitekturen ein weiteres Handicap, daß der Stack an einem Punkt des Programmes immer eine konsistente Tiefe haben muß, wenn der Punkt über mehrere Pfade erreicht werden kann, weil die Argumente für den folgenden Code ja auf dem Stack liegen. Eine triviale Lösung dieses Problems ist die Beschränkung der Optimierung auf basic blocks (vergleiche Abschnitt 3.2), da der Stack am Anfang und Ende eines basic blocks leer ist (dies gilt nicht in allen Fällen für die JavaVM, Abschnitt 4.3.4 beschreibt die Probleme, die dadurch entstehen).

Das eigentliche stack scheduling auf basic block-Ebene erfolgt nun in zwei Teilen: Zuerst werden Möglichkeiten gesucht, das Laden bzw. Speichern von lokalen Variablen eliminieren zu können, und die gefundenen Möglichkeiten in einer noch näher zu beschreibenden Reihenfolge sortiert; danach wird versucht, die ermittelten Möglichkeiten der Reihe nach auszuschöpfen, d.h. die Zugriffe zu eliminieren.

Worin besteht nun so eine „Möglichkeit“, einen Variablenzugriff zu unterdrücken? Klarerweise ist sie nur dann gegeben, wenn eine Variable in einem basic block öfter als einmal verwendet wird. Für die weiteren Ausführungen nehmen wir das Programmfragment aus Beispiel 4.2 als basic block an, wobei als Zusatzbedingung gelte, daß die Variable `b` nach diesen Berechnungen nicht mehr gebraucht wird, also eine „dead variable“ darstellt.

Beispiel 4.2 C-Code eines basic block

```
...
int a, b, c;
...
b = a + c;
a = b + 5;
c = b + a; /* letzte Verwendung von b */
...
```

Die Beispiele 4.3 und 4.4 enthalten den Stackcode für diesen Abschnitt des Codes für HYSTAMIN bzw. die JavaVM. In Klammern wird dabei der

[3] In der Literatur allgemein als „register allocation“ bekannt.

Zustand des Stack angegeben, *bevor* der entsprechende Befehl ausgeführt wird, wobei der top of stack durch -- angedeutet ist; als Kommentar wird der jeweilige Befehl in Forth-Notation angegeben.

In Beispiel 4.2 hat die Variable a im stackframe den Offset 0, die Variable b den Offset 4 und c besitzt den Offset 8. Auf Zeile 1 wird der Inhalt des framepointers auf den Stack kopiert (er enthält damit zugleich die Adresse von a) und danach die Variable geladen. In Zeile 2 muß vor dem Hauptspeicherzugriff noch der Offset 8 für die Variable c addiert werden. Danach befinden sich die Werte der beiden Variablen auf dem Stack und können addiert werden, bevor die Adresse der Variable b berechnet und schließlich die Summe dorthin geschrieben wird. Der restliche Code folgt dem einfachen Grundmuster.

Beispiel 4.3 HYSTAMIN-Code für Beispiel 4.2

```
1  getfp lod                ( -- )       % a @
2  getfp push 8 add lod     ( a -- )     % c @
3  add                      ( a c -- )   % +
4  getfp push 4 add str     ( a+c -- )   % b !
5  getfp push 4 add lod     ( -- )       % b @
6  push 5                   ( b -- )     % 5
7  add                      ( b 5 -- )   % +
8  getfp str                ( b+5 -- )   % a !
9  getfp push 4 add lod     ( -- )       % b @ ('dead')
10 getfp lod                ( b -- )     % a @
11 add                      ( b a -- )   % +
12 getfp push 8 add str     ( b+a -- )   % c !
```

Ähnlich sieht der Code für die JavaVM aus (Beispiel 4.4), allerdings werden die lokalen Variablen hier nicht mit ihrem Offset im stackframe adressiert, sondern über ihren Index. Die Variable a besitzt in dem Beispiel den Index 1, b den Index 2 und c ist an Index 3 zu finden. Bis auf diese syntaktischen Details ist der Code mit dem für HYSTAMIN äquivalent.

Die Variablen a und b werden in diesem Codeteil öfters benutzt und sind daher Ziel der Optimierungsversuche. Im ersten Schritt werden die Möglichkeiten für das stack scheduling ermittelt. Jeder Befehl wird dabei mit einem „Abbild" des Laufzeitstack *vor* der Ausführung des Befehls versehen. Dieses Abbild wird im weiteren als „Stack des Befehls" oder „Stackinformation" bezeichnet.

Für jeden Befehl, der eine lokale Variable auf den Stack lädt, wird das näheste Auftreten dieser Variable auf dem Stack *vor* dem Ladebefehl ermittelt. Die so ermitteln Befehlspaare von „Auftreten" und „Laden" werden dann sortiert, wobei das Kriterium die Distanz zwischen den zwei Befehlen ist (dieser Heuristik liegt zugrunde, daß die Wahrscheinlichkeit für einen Erfolg bei näheren „Nutzungspunkten" höher ist).

Beispiel 4.4 JavaVM-Code für Beispiel 4.2

```
1  iload_1    ( -- )       % a @
2  iload_3    ( a -- )     % c @
3  iadd       ( a c -- )   % +
4  istore_2   ( a+c -- )   % b !
5  iload_2    ( -- )       % b @
6  iconst_5   ( b -- )     % 5
7  iadd       ( b 5 -- )   % +
8  istore_1   ( b+5 -- )   % a !
9  iload_2    ( -- )       % b @ ('dead')
10 iload_1    ( b -- )     % a @
11 iadd       ( b a -- )   % +
12 istore_3   ( b+a -- )   % c !
```

In unserem Beispiel finden sich drei Paare, die eine Möglichkeit zur Optimierung darstellen (für das Beispiel wurde die klarere JavaVM-Syntax gewählt), siehe Beispiel 4.5.

Beispiel 4.5 Möglichkeiten für das scheduling

```
1  iload_1    ( -- )       % a @
2  iload_3    ( a -- )     % c @
3  iadd       ( a c -- )   % +
4  istore_2   ( b -- )     % b ! (b=a+c)
5  iload_2    ( -- )       % b @
6  iconst_5   ( b -- )     % 5
7  iadd       ( b 5 -- )   % +
8  istore_1   ( a -- )     % a ! (a=b+5)
9  iload_2    ( -- )       % b @ ('dead')
10 iload_1    ( b -- )     % a @
11 iadd       ( b a -- )   % +
12 istore_3   ( c -- )     % c ! (c=b+a)
```

Die Variable `b` wird sofort nach dem Speichern bei Befehl 4 wieder geladen (dieses Laden wird allerdings meist schon durch den peephole optimizer eliminiert, siehe auch Abschnitt 3.4). Die Distanz zwischen diesen zwei Punkten ist 1. Für die zwei weiteren Paare (Befehl 7 und 9 bzw. 8 und 10) ist die Distanz 2, da jeweils ein dritter Befehl zwischen Auftreten und Nachladen der entsprechenden Variable steht.

Die Idee hinter dem Ansatz des stack scheduling ist, den Wert einer Variablen am Punkt des Auftretens am Stack zu kopieren und zwischenzuspeichern und die Kopie dann zu verwenden, anstatt die Variable erneut aus dem Speicher zu laden. Jedes der Paare, die im vorigen Schritt gefunden wurden, wird nun untersucht, ob es folgenden Bedingungen genügt:

- Der Wert der entsprechenden Variable muß am Punkt des Auftretens mit einer einfachen Stackmanipulation an die unterste Position im Stack kopierbar sein, ansonsten stellt dieses Paar keine gültige Möglichkeit für ein scheduling dar. Abhängig von der Architektur, auf die die Optimierung abzielt, sind natürlich unterschiedliche Befehle vorhanden, mit denen obige Bedingung erfüllt werden kann. Phil Koopman verwendet für das Beispiel in seinem Artikel die vier folgenden Forth-Wörter: `dup`, `tuck`, `under` (mit dem Stackeffekt `( a b -- a a b )`) und `tuck_2` (Stackeffekt: `( a b c -- a a b c )`). Die in dieser Arbeit verwendeten Architekturen besitzen teilweise andere Befehle zur Stackmanipulation, sodaß das Ergebnis der Optimierung für diese Maschinen nicht zwangsläufig mit dem Original in Koopmans Artikel übereinstimmen muß (siehe auch Abschnitt 2.3).
- Die Tiefe des Stack darf am Punkt des Ladens nur so groß sein, daß der im vorigen Schritt an die unterste Position im Stack gespeicherte Wert mit einer einfachen Stackmanipulation wieder als oberstes Stackelement fungieren kann. Hier gelten hinsichtlich der Befehle die analogen Einschränkungen wie im vorigen Punkt, Koopman nennt in diesem Zusammenhang die Forth-Wörter `swap` und `rot`. Klarerweise darf die Stacktiefe am Punkt des Ladens 0 sein, dann wird überhaupt keine Stackmanipulation an diesem Punkt benötigt, der kopierte Wert ist das einzige und damit oberste Element im Stack.

Treffen beide Bedingungen zu, so wird vor dem ersten Befehl des Paares eine Kopier-Instruktion eingefügt und der Ladebefehl durch den vorher ermittelten Stackmanipulationsbefehl ersetzt. Hinterher muß noch das Abbild des Stack für alle Befehle zwischen den zwei neu erzeugten aktualisiert und der Bereich nach weiteren Möglichkeiten für das scheduling durchsucht werden, weil sich durch die Optimierung auch die Voraussetzungen für das scheduling geändert haben.

Falls der eliminierte Ladebefehl auf eine Variable angewendet wurde, die im weiteren nicht mehr gebraucht wird (eine dead variable), so wird die Markierung „dead" an den nähesten vorigen Lese- oder Schreibbefehl der Variable weitergegeben.

Laut Phil Koopman war der Versuch, das Kopieren der Variablen im Stack in beliebige Tiefen zu ermöglichen, nicht besonders erfolgreich. Als Gründe dafür gibt er einerseits an, daß Stackmaschinen im allgemeinen beim Zugriff auf tief im Stack verborgene Elemente empfindlich langsamer werden (das stack buffering zeigt dann keine Wirkung mehr), andererseits erzeugt dieses scheduling relativ viele `drop`-Befehle, um nicht mehr gebrauchte Variablen vom Stack zu nehmen.

Betrachten wir unser Beispiel, so wird als erstes Paar das Speichern und Laden der Variable `b` (Befehl 4 und 5) untersucht, da es die geringste Distanz aller Paare besitzt. Dieses Paar erfüllt beide der vorhin genannten Bedingungen und kann daher dem scheduling unterzogen werden. Vor dem Speichern wird der Wert mittels eines `dup` an den Grund des Stack kopiert, der Ladebefehl kann völlig eliminiert werden (Beispiel 4.6).

Beispiel 4.6 Code nach dem scheduling des ersten Paares

```
1  iload_1    ( -- )       % a @
2  iload_3    ( a -- )     % c @
3  iadd       ( a c -- )   % + (b = a+c)
4  dup        ( b -- )     % (kopieren von b)
5  istore_2   ( b b -- )   % b !
6  iconst_5   ( b -- )     % 5
7  iadd       ( b 5 -- )   % +
8  istore_1   ( a -- )     % a ! (a = b+5)
9  iload_2    ( -- )       % b @ ('dead')
10 iload_1    ( b -- )     % a @
11 iadd       ( b a -- )   % +
12 istore_3   ( c -- )     % c ! (c = b+a)
```

Als nächstes Paar wählen wir nun für unser Beispiel die Befehle 7 und 9. Allerdings ist es uns hier mit den von der JavaVM zur Verfügung gestellten Befehle nicht möglich, den Wert der Variablen `b` vor dem `iadd` in das unterste Element des Stack zu kopieren (dafür wäre ein Befehl `under` wie bei HYSTAMIN nötig). Daher bleibt dieses Paar unberücksichtigt und wir gehen zur nächsten Möglichkeit über.[4]

Beispiel 4.7 Code nach dem scheduling aller drei Paare

```
1  iload_1    ( -- )       % a @
2  iload_3    ( a -- )     % c @
3  iadd       ( a c -- )   % + (b = a+c)
4  dup        ( b -- )     % (kopieren von b)
5  istore_2   ( b b -- )   % b !
6  iconst_5   ( b -- )     % 5
7  iadd       ( b 5 -- )   % + (a = b+5)
8  dup        ( a -- )     % (kopieren von a)
9  istore_1   ( a a -- )   % a ! (a = b+5)
10 iload_2    ( a -- )     % b @ ('dead')
11 swap       ( a b -- )   % (Kopie verschieben)
12 iadd       ( b a -- )   % +
13 istore_3   ( c -- )     % c ! (c = b+a)
```

Es bleibt uns nur mehr das Paar mit den Befehlen 8 und 10 als Möglichkeit für das scheduling. Hier ergeben sich beim Punkt des Auf-

[4] Eine Erweiterung der Methode, die auch das Nachladen von `b` eliminieren kann, wird im folgenden Abschnitt vorgestellt.

tretens keine weiteren Probleme und wir können analog zum ersten scheduling einfach ein `dup` vor dem `istore_1` einfügen, um den Wert von `a` an den Grund des Stack zu kopieren. Am Punkt des Nachladens ist die Stacktiefe allerdings nicht 0 wie beim ersten Paar, sondern 1, deshalb ersetzen wir `iload_1` durch `swap`, um danach `a` als oberstes Stackelement ansprechen zu können. Dieses `swap` ist in diesem Beispiel natürlich nicht nötig, da `iadd` kommutativ ist, allerdings berücksichtigt der scheduler diese Tatsache nicht; stattdessen wird die Ungeschicklichkeit im nächsten Durchlauf des peephole optimizers „repariert“ und das `swap` gelöscht. Das Ergebnis der eben beschriebenen Bemühungen ist in Beispiel 4.7 zu sehen.

Nun haben wir zwar zwei Befehle zum Nachladen von lokalen Variablen eliminiert, allerdings gibt es einige Erweiterungen des Algorithmus, die weitere Optimierungen ermöglichen.

Erweiterungen

Dieser Abschnitt enthält einige Erweiterungen der von Koopman vorgeschlagenen Methode, die in seinem Artikel nicht erwähnt werden, allerdings keiner allzu komplizierten Überlegungen bedürfen.

Betrachten wir noch einmal Beispiel 4.7, so sehen wir, daß auch das Nachladen der Variablen `b` unterdrückt werden kann, wenn nicht nur das letzte Vorkommen dieser Variable auf dem Stack berücksichtigt wird, sondern auch weitere Vorkommen, die einen größeren Abstand zum Ladebefehl aufweisen.

Allerdings ist hierbei zu beachten, daß Variablen auf dem Stack, die einen anderen Wert besitzen als den, welchen der Ladebefehl auf den Stack bringt, nicht als weitere Möglichkeiten für das scheduling verwendet werden dürfen. In der Praxis bedeutet dies, daß die Suche nach weiteren Vorkommen abgebrochen werden muß, sobald ein Speicherbefehl auf die Variable auftritt.

Es besteht natürlich die Möglichkeit, daß der Wert, welcher mit dem Schreibbefehl in der Variable gespeichert wird, früher berechnet wurde als zum Zeitpunkt des Speicherns, d.h. zwischen der Berechnung des Wertes und der Schreiboperation liegen weitere Befehle, bei deren Ausführung der Stack auch den Wert der Variable enthält. Damit sind auch sie Kandidaten für die Paarbildung, werden aber in der aktuellen Implementierung nicht berücksichtigt, was in der Praxis keine großen Auswirkungen hat (das scheduling eines Paares scheitert meist daran, daß der Stack beim Ladebefehl zu tief ist und der Wert daher nicht an die oberste Position kopiert werden kann).

Bei der JavaVM ergibt sich noch eine weitere Einschränkung in Zusammenhang mit dem Befehl `iinc`, mehr dazu im Abschnitt 4.3.4.

Beispiel 4.8 Code nach dem scheduling des 2. Paares (Erweiterung)

```
 1 iload_1    ( -- )          % a @
 2 iload_3    ( a -- )        % c @
 3 iadd       ( a c -- )      % + (b = a+c)
 4 dup        ( b -- )        % (kopieren von b)
 5 istore_2   ( b b -- )      % b ! ('dead')
 6 dup        ( b -- )        % (kopieren von b)
 7 iconst_5   ( b b -- )      % 5
 8 iadd       ( b b 5 -- )    % + (a = b+5)
 9 dup        ( b a -- )      % (kopieren von a)
10 istore_1   ( b a a -- )    % a ! (a = b+5)
11 swap       ( b a -- )      % (b verschieben)
12 swap       ( a b -- )      % (a verschieben)
13 iadd       ( b a -- )      % +
14 istore_3   ( c -- )        % c ! (c = b+a)
```

Für das vorige Beispiel kommen durch diese Erweiterung für das Nachladen von `b` (Befehl 10) nun zu `iadd` (Befehl 7) noch die Befehle 4 bis 6 dazu. Nach dem Sortieren der neuen Paare ist die nächste untersuchte Möglichkeit das Paar mit dem Auftreten bei Befehl 6. Da `b` hier als einziges Element auf dem Stack liegt, können wir dieses Paar optimieren und erhalten als Ergebnis Beispiel 4.8

Beispiel 4.9 Code nach der Optimierung durch den peephole optimizer

```
 1 iload_1    ( -- )          % a @
 2 iload_3    ( a -- )        % c @
 3 iadd       ( a c -- )      % + (b = a+c)
 4 dup        ( b -- )        % (kopieren von b)
 5 iconst_5   ( b b -- )      % 5
 6 iadd       ( b b 5 -- )    % + (a = b+5)
 7 dup        ( b a -- )      % (kopieren von a)
 8 istore_1   ( b a a -- )    % a !
 9 iadd       ( b a -- )      % + (c = b+a)
10 istore_3   ( c -- )        % c !
```

Beachtenswert hierbei ist, daß der eliminierte Ladebefehl eine Variable geladen hätte, die als „dead" markiert war. Diese Markierung wird laut Algorithmus ja an den nächsten Lade- oder Speicherbefehl für diese Variable weitergegeben, in dem Beispiel also an den Speicherbefehl `istore_2` (Befehl 5). Daraus ist ersichtlich, daß der Wert, der dabei gespeichert wird, im weiteren Programmverlauf nicht benötigt wird. Der peephole optimizer eliminiert solche Schreibbefehle (und auch das `dup`, das nur eingefügt wurde, weil der Speicherbefehl destruktiv ist). Außerdem ist es natürlich für den peephole optimizer trivial, die Befehlsfolge `swap swap` zu unterdrücken und wir kommen daher zum endgültigen Resultat, das für diesen Block optimal ist (Beispiel 4.9).

Eine zweite Erweiterung des Algorithmus beruht auf der Beobachtung, daß es nicht in jedem Fall notwendig ist, die Variable beim Punkt des Auftretens ganz an den Grund des Stack zu kopieren. Eigentlich muß der Wert nur so „tief" im Stack vergraben werden, daß er bis zum Punkt des Nachladens nicht zerstört oder verändert wird. Betrachten wir Beispiel 4.10 und den dazugehörigen JavaVM-Code in Beispiel 4.11.

Beispiel 4.10 C-Code für einen basic block

```
...
int a, b, c, d;
...
a = b + c;
d = b - c;
...
```

Beispiel 4.11 JavaVM-Code für Beispiel 4.10

```
1 iload_2    ( -- )       % b @
2 iload_3    ( b -- )     % c @
3 iadd       ( b c -- )
4 istore_1   ( a -- )     % a ! (a=b+c)
5 iload_2    ( -- )       % b @
6 iload_3    ( b -- )     % c @
7 isub       ( b c -- )
8 istore_4   ( d -- )     % d ! (d=b-c)
```

Wir erkennen drei Paare für das scheduling, nämlich die Befehle 3 und 5, 2 und 5 sowie das Paar 3 und 6. Das erste dieser Paare besitzt zwar die geringste Distanz, wird aber sofort ausgeschieden, weil `b` nicht auf den Grund des Stack kopiert werden kann. Nehmen wir nun an, das Paar mit den Befehlen 5 und 2 werde als nächstes optimiert. Daraus ergibt sich der Code aus Beispiel 4.12.

Beispiel 4.12 Code nach dem scheduling des ersten Paares

```
1 iload_2    ( -- )         % b @
2 dup        ( b -- )
3 iload_3    ( b b -- )     % c @
4 iadd       ( b b c -- )
5 istore_1   ( b a -- )     % a ! (a=b+c)
6 iload_3    ( b -- )       % c @
7 isub       ( b c -- )
8 istore_4   ( d -- )       % d ! (d=b-c)
```

Nun nehmen wir das Befehlspaar (`iadd`, `iload_3`) aus Zeile 4 und 6 als nächste und letzte Möglichkeit für das scheduling. Da der Stack beim ersten Ladebefehl bereits die Tiefe zwei hat, muß nach diesem Befehl ein `dup_x2` eingefügt werden, damit der geladene Wert auf dem

Grund des Stack landet. Zum Zeitpunkt des nochmaligen Ladens hat der Stack die Tiefe eins, also müssen wir ein `swap` statt des Ladens erzeugen, damit die Operanden für das nachfolgende `isub` in der richtigen Reihenfolge am Stack aufscheinen. Nach diesen Operationen erhalten wir den Code aus Beispiel 4.13.

Beispiel 4.13 Code nach dem scheduling des zweiten Paares

```
1 iload_2    ( -- )            % b @
2 dup        ( b -- )
3 iload_3    ( b b -- )        % c @
4 dup_x2     ( b b c -- )
5 iadd       ( c b b c -- )
6 istore_1   ( c b a -- )      % a ! (a=b+c)
7 swap       ( c b -- )
8 isub       ( b c -- )
9 istore_4   ( d -- )          % d ! (d=b-c)
```

Nun sind zwar alle Möglichkeiten zur Optimierung nach dem Original-Algorithmus ausgeschöpft, allerdings ist der erzeugte Code nicht optimal. Der Wert der Variablen `c` bei Befehl 4 muß nicht ganz auf den Grund des Stack kopiert werden, es genügt, ihn an die zweitunterste Position zu kopieren, da die Befehle bis zum Zeitpunkt des Nachladens das unterste Element des Stack nicht verändern. Als weiteren Vorteil dieser Optimierung würde das Zurechtrücken der Operanden für `isub` mittels `swap` überflüssig. Der weiter optimierte Code hätte damit das Aussehen wie in Beispiel 4.14.

Beispiel 4.14 Code nach weiterer Optimierung (nicht implementiert)

```
1 iload_2    ( -- )            % b @
2 dup        ( b -- )
3 iload_3    ( b b -- )        % c @
4 dup_x1     ( b b c -- )
5 iadd       ( b c b c -- )
6 istore_1   ( b c a -- )      % a ! (a=b+c)
7 isub       ( b c -- )
8 istore_4   ( d -- )          % d ! (d=b-c)
```

Die aktuelle Implementierung nutzt diese Erweiterung allerdings noch nicht, weil die eben dargestellte Situation durch das Vorsortieren der Paare nach der Distanz oft vermieden wird und die weitere Optimierung daher nur selten zur Anwendung kommen würde.

4.2.2 Globales stack scheduling

Zusätzlich zu dem im vorigen Abschnitt erläuterten intra-block stack scheduling geht Phil Koopman in seinem Artikel noch sehr kurz und oberflächlich auf globales scheduling ein (d.h. die Optimierung von Variablenzugriffen über die Grenzen eines basic blocks hinaus). Seinen Ausführungen nach konnte er für die von ihm als Test verwendeten Programme (die Stanford Integer Benchmarks) eine signifikante Verbesserung gegenüber dem Ergebnis der alleinigen Verwendung von lokalem scheduling feststellen.

Allerdings enthält der Artikel keinen genaueren Algorithmus, da die globale Optimierung von Hand durchgeführt wurde.[5] Einige wichtige Beobachtungen, die sich bei dem globalen Verfahren machen lassen, sind:

- Schleifenvariablen sind meist vielversprechende Kandidaten für ein scheduling über basic block Grenzen hinaus. Besonders geeignet für die Zwischenspeicherung von Variablen scheint der „return stack“ in Forth-ähnlichen Architekturen (wie z.B. HYSTAMIN) zu sein.
- Es bringt meist mehr, die Optimierung in Schleifen auf maximal zwei Variablen für das globale scheduling zu beschränken und diese dafür während der gesamten Dauer der Schleife auf dem Stack zu speichern.

Da globale Algorithmen in dieser Arbeit nicht weiter verfolgt werden und auch in der Literatur keinen näheren Details zu diesem Thema zu finden sind, wird dieser Ansatz hier auch nicht weiter ausgeführt, obwohl er in der Praxis wahrscheinlich große Bedeutung für eine gelungene Optimierung besitzt.

4.2.3 Instruction scheduling

Für die Beurteilung der Ergebnisse in Kapitel sechs ist die von den verwendeten Werkzeugen (siehe auch Abschnitt 6.1) vorgegebene Befehlsanordnung wichtig, da stack scheduling kein eigenes „instruction scheduling“ durchführt, sondern sich darauf beschränkt, die Befehle in der vorgegebenen Reihenfolge zu optimieren. Ganz im Gegensatz dazu ist das instruction scheduling der wichtigste Teil des DAG scheduling aus Kapitel fünf.

Die Optimierung von HYSTAMIN-Code verwendet SUIF-Zwischencode ([Sta96]), der von den SUIF-Werkzeugen `snoot` und `porky` erzeugt wurde, als Eingabe. Der Zwischencode ähnelt stark einem abstrakten Syntaxbaum, dessen Knoten aus SUIF-Befehlen bestehen. Die Kindknoten eines Knotens repräsentieren weitere Befehle, die die Operanden für den Elternknoten berechnen.

[5] „I just used ad-hoc techniques as neccessary.“ ([Koo92])

Als erster Schritt muß der Zwischencode in Stackcode für HYSTAMIN transformiert werden; damit ist gleichzeitig das instruction scheduling verbunden. Der Zwischencodebaum wird in einem „depth first postorder traversal“ durchlaufen: Kommt der Algorithmus dabei an einem Knoten an, wird er erst rekursiv für alle Kindknoten aufgerufen und erzeugt danach einen oder mehrere HYSTAMIN Befehle, die dem SUIF-Befehl des Knotens entsprechen. Durch diese Vorgehensweise ist sichergestellt, daß die Argumente für jeden Befehl korrekt auf dem Stack liegen.

Etwas anders stellt sich die Situation bei JavaVM-Code dar. Die Eingabe für die Optimierung ist hier ein „classfile“, das JavaVM-Code enthält und nicht mehr in Stackcode transformiert werden muß. Der Java Compiler `javac` aus dem Java Development Kit erzeugt das classfile und ist damit für das instruction scheduling verantwortlich. Er verwendet dabei die gleiche Strategie eines depth first postorder traversal des Syntaxbaumes, der die Java-Ausdrücke repräsentiert, weil das die einfachste Möglichkeit für die Generierung von Stackcode darstellt.

4.3 Implementierung

In diesem Abschnitt finden sich die wichtigsten Details zur konkreten Implementierung von Koopmans Algorithmus. Nach einigen allgemeinen Anmerkungen und einer Abschätzung des Aufwandes sind auch Besonderheiten, die durch den verschiedenen Befehlssatz der in dieser Arbeit verwendeten Architekturen entstehen, angeführt.

4.3.1 Allgemeines

Die Optimierungsmethode nach Koopman ist wie alle anderen Optimierungen ein pass, der auf Funktionen bzw. Methoden arbeitet. Die Eingabe für den pass ist eine Funktion, deren Code schon in basic blocks aufgeteilt wurde. Es werden keine weiteren Anforderungen an die Eingabe gestellt, allerdings fallen einige Optimierungsmöglichkeiten weg, wenn die nicht mehr gebrauchten Variablen („dead variables“) noch nicht ermittelt und daher die entsprechende Markierung an den jeweiligen Instruktionen nicht angebracht wurde (siehe Abschnitt 3.3).

Alternativ dazu besteht auch die Möglichkeit, die Optimierung nur auf einen bestimmten basic block einer Funktion oder Methode anzuwenden; diese Funktionalität wird vom DAG scheduling verwendet, wenn dieses keine gültige Ausführungsreihenfolge findet (vergleiche Abschnitt 5.2). In jedem Fall bildet ein basic block die „Einheit“ für die Optimierung – im Fall einer gesamten Funktion bzw. Methode als Eingabe werden einfach alle ihre basic blocks nacheinander bearbeitet.

Grundsätzlich arbeitet die Optimierung nach dem Schema aus Programm 4.1. Aus diesem Pseudocode ist ersichtlich, daß jedesmal, wenn ein Paar erfolgreich optimiert werden konnte, die gesamte Infrastruktur für das scheduling erneut aufgebaut wird. Da sich durch die Optimierung die Stackinformation ändert und damit auch die Bedingungen für die weiteren Paare zusammenhängen, ist es nötig, zumindest den durch die Befehle des optimierten Paares begrenzten Bereich erneut zu behandeln. Der Einfachheit halber wird in der vorliegenden Implementierung allerdings auf diese Möglichkeit verzichtet und gleich der gesamte basic block noch einmal untersucht.

Programm 4.1 Ablauf des stack scheduling

```
comment: Eingabe: ein basic block einer Funktion
         Ausgabe: der optimierte basic block

geändert ← true;
while geändert do
    erzeuge bild des stack;
    suche paare für das scheduling;
    sortiere paare;
    optimiere höchstens ein befehlspaar;
    if kein paar optimiert?
       then geändert ← false;
    fi;
end;
```

Sehen wir uns nun die erste Teilaufgabe des obigen Programmfragmentes näher an, nämlich die Erstellung der Stackinformation (d.h. des Abbilds des Laufzeitstack) für jeden Befehl im basic block. Jeder Instruktion wird ein Stackabbild zugeordnet, das für jedes Stackelement zur Laufzeit angibt, ob es eine bekannte Größe (d.h. den Wert einer lokalen Variablen) enthält. Da ein Stackelement mehrere Variablen gleichzeitig beherbergen kann (die dann zum entsprechenden Zeitpunkt denselben Wert besitzen), muß das dem Befehl zugeordnete Stackabbild dies auch widerspiegeln können. Zweckmäßigerweise erfolgt das dadurch, daß die Elemente des Stackabbildes Bitvektoren sind, die für jede lokale Variable angeben, ob deren Wert im entsprechenden Stackelement gespeichert ist.

Der Algorithmus beruht darauf, daß der Code im basic block symbolisch exekutiert und damit das Laufzeitverhalten der Befehle auf einem nachgebildeten Stack emuliert wird. Dabei ist auf die Werte von lokalen Variablen zu achten, die für die Suche nach den Paaren essentiell sind (Programm 4.2).

Da der Stack zu Beginn des basic block leer sein sollte (dies gilt nicht immer für JavaVM Code, siehe auch die Abschnitte 2.3.2 und 4.3.4), initialisieren wir ihn als solchen. Danach wird der basic block Be-

Programm 4.2 Erzeugen der Stackinformation für einen basic block

```
comment: Eingabe: ein basic block einer Funktion
         Ausgabe: der basic block, für jeden Befehl erweitert
                  um das Abbild des Stack vor der
                  Ausführung des Befehls

aktuelles stackabbild ← empty;
for alle befehle im basic block do
    stackabbild des befehls ← aktuelles stackabbild;
    if befehl = speichern einer lokalen variablen
      then
           aktuelles stackabbild.oberstes element ← variable;
           stackabbild des befehls.oberstes element ← variable;
    fi;
    if befehl = laden einer lokalen variablen
      then aktuelles stackabbild.push(variable);
    fi;
    if befehl = stackmanipulation
      then manipulation auf dem aktuellen stackabbild;
    fi;
    if befehl = anderer befehl
      then
           aktuelles stackabbild.pop(argumente des befehls);
           if befehl hat ein ergebnis?
             then aktuelles stackabbild.push(keine variable);
           fi;
    fi;
end;
```

fehl für Befehl durchgegangen. Um das Aussehen des Stack vor der Ausführung des nun zu behandelnden Befehls zu berechnen, wird das Stackabbild des Befehls mit dem „aktuellen“ Stackabbild initialisiert.

Um das Stackabbild für den nächsten Befehl zu berechnen, wird nun der zu behandelnde Befehl symbolisch auf dem „aktuellen“ Stackabbild exekutiert. Speichert der Befehl einen Wert in eine lokale Variable, so muß das oberste Stackelement zur Laufzeit den Wert dieser Variable enthalten. Daher wird das Stackabbild des Befehls und auch das aktuelle Stackabbild entsprechend manipuliert. Lädt der Befehl eine lokale Variable, dann wissen wir, daß das neue oberste Element des Laufzeitstack den Wert der Variable enthalten wird und fügen daher dem aktuellen Stackabbild einen entsprechenden Eintrag hinzu.

Stackmanipulationsbefehle werden einfach auf dem aktuellen Stackabbild vollzogen und emulieren damit ihr Laufzeitverhalten. Andere Befehle nehmen so viele Elemente vom Stackabbild, wie sie es auch zur Laufzeit tun würden, und geben – wenn sie ein Ergebnis liefern –

ein Element hinzu, dessen Wert bezüglich lokaler Variablen aber nicht bekannt ist.

Betrachten wir nun die Suche nach den Befehlspaaren, die eine Möglichkeit für die Optimierung darstellen. Dieser Teil des Algorithmus ist recht einfach (Programm 4.3).

Programm 4.3 Suche nach Möglichkeiten für die Optimierung

```
comment: Eingabe: ein basic block einer Funktion
         Ausgabe: alle Möglichkeiten für das scheduling

paare ← empty;
for alle befehle im basic block do
    if befehl = laden einer lokalen variablen ∧
      gültiges nachladen?
      then
          for alle vorherigen befehle do
              if stack des vorigen befehls enthält variable ∧
                gültiges auftreten?
                then
                    paare ← paare ∪ (befehl, voriger befehl);
              fi;
          end;
    fi;
end;
sortiere paare aufsteigend nach distanz;
```

Für jeden Ladebefehl im basic block wird ermittelt, ob er ein „gültiges“ Nachladen bewirkt (d.h. ob ein Wert vom untersten Element des Stack nach oben gebracht werden könnte), und dann werden die davorliegenden Instruktionen untersucht, ob ihr Abbild des Laufzeitstacks die entsprechende Variable beinhaltet. Ist dem so und stellt der gefundene Befehl auch ein „gültiges“ Auftreten dar (die Variable kann auf den Grund des Stack kopiert werden), wird das gefundene Paar der Menge der Möglichkeiten hinzugefügt. Nachdem alle Paare ermittelt wurden, werden sie nach ihrer Distanz sortiert.

Als krönendem Abschluß wenden wir uns nun dem eigentlichen Akt des „scheduling“ zu, dem allerdings nicht mehr allzuviel zu tun bleibt, wie Programm 4.4 zeigt.

Es muß nur noch der entsprechende Kopierbefehl eingefügt und der Ladebefehl durch eine Stackmanipulation ersetzt werden. War das Laden der Variable als letzter Zugriff auf diese markiert, wird die Markierung an einen vorher erfolgenden Zugriff weitergegeben, falls ein solcher existiert.

Die spezifischen Eigenheiten einer konkreten Architektur machen die Anpassung des grundlegenden Algorithmus daran nötig und beide in

Programm 4.4 Das scheduling eines Paares

```
comment: Eingabe: ein basic block einer Funktion
                  eine Menge von Paaren für das scheduling
         Ausgabe: der optimierte basic block

if paare ≠ empty
  then
       nimm erstes paar;
       füge kopierbefehl bei auftreten ein;
       ersetze ladebefehl durch stackmanipulation;
       if variable beim laden „dead“?
         then markiere vorigen zugriff auf variable;
       fi;
fi;
```

dieser Arbeit untersuchten Architekturen besitzen Eigenschaften, die in diesem Zusammenhang berücksichtigt werden müssen. Nach einer Abschätzung des Aufwandes für das stack scheduling behandeln die Abschnitte 4.3.3 und 4.3.4 daher kurz spezielle Details der Implementierung für die zwei Architekturen.

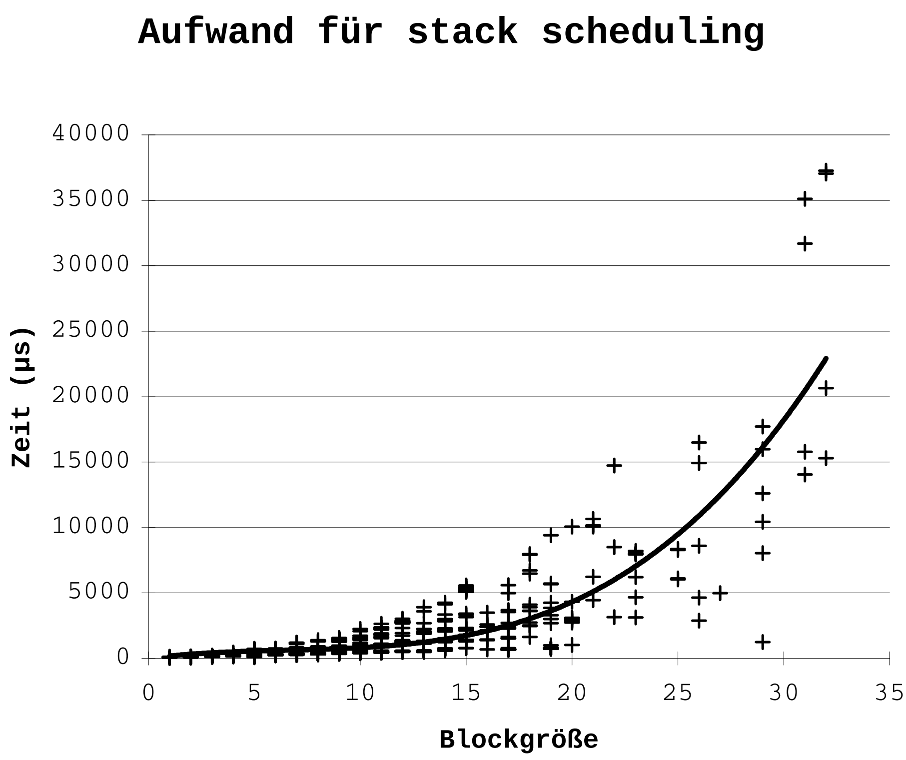
4.3.2 Aufwand

Um den Aufwand des stack scheduling für den schlechtesten Fall abzuschätzen, nehmen wir an, daß der basic block aus n Befehlen besteht, die alle die gleiche Variable laden.[6] Demnach gibt es $\frac{n(n-1)}{2}$ Paare, also in der Größenordnung $O(n^2)$. Das Erzeugen der Stackinformation verursacht linearen Aufwand $O(n)$, die Suche nach Paaren für das scheduling jedoch quadratischen ($O(n^2)$), weil für jeden der n Ladebefehle alle vorigen auf der Suche nach einem Partner für das Paar untersucht werden. Für das Sortieren mittels quicksort erhalten wir im worst case einen Aufwand von $O(m^2)$, n^2 Paare müssen sortiert werden, deshalb ist der Aufwand für das Sortieren $O(n^4)$. Das scheduling eines Paares hat konstanten Aufwand $O(c)$. Es können maximal n Paare beim scheduling erfolgreich sein, also wird die Hauptschleife (siehe Programm 4.1) n-mal durchlaufen. Da der Aufwand für einen Durchlauf der Schleife $O(n^4)$ beträgt, kommen wir damit insgesamt auf einen Aufwand von $O(n^5)$.

Abbildung 4.1 enthält eine Reihe von Meßwerten für das stack scheduling von basic blocks verschiedener Größen (die Eingabe bestand aus den Klassen der JGL, siehe auch Abschnitt 6.1). Die Daten wurden für einen Pentium-120 ermittelt. Näherungsweise steigt der Zeitauf-

[6] Die Tatsache, daß dadurch am Ende des basic block der Stack nicht leer ist, können wir der JavaVM zuschreiben.

Abbildung 4.1 Aufwand für stack scheduling



wand mit der dritten Potenz der Blockgröße, allerdings ist aufgrund der großen Streuung keine exakte Aussage möglich.

4.3.3 HYSTAMIN

Die Implementierung des stack scheduling für HYSTAMIN macht sich die Tatsache zunutze, daß auch globale Variablen auf dem Stack zwischengespeichert und wieder benutzt werden können, sofern sie zwei Bedingungen erfüllen:

- Sie sind nicht als `volatile` deklariert worden und können auch nicht durch Zugriffe auf Zeiger verändert werden (das leidige Problem des „aliasing").
- Sie können nicht durch andere Funktionen verändert werden.

Die erste Bedingung wird durch Funktionen der SUIF-Bibliothek überprüft, zweitere kann dadurch sichergestellt werden, daß für Blöcke mit einem Funktionsaufruf die Optimierung globaler Variablen desaktiviert wird.

Natürlich sind auch nur lokale Variablen Kandidaten für die Optimierung, wenn sie die erste der beiden obigen Bedingungen erfüllen (insbesondere natürlich die des aliasing).

4.3.4 JavaVM

Ein Problem bei dieser Architektur wurde schon in Abschnitt 2.3.2 erwähnt. Es trifft nicht immer zu, daß der Stack am Anfang und Ende eines basic block leer ist. Allerdings macht die Behandlung dieser Anomalie keine großen Schwierigkeiten, das „aktuelle Stackabbild“ wird nicht wie in Programm 4.2 als „leer“ initialisiert, sondern mit der Anzahl von Elementen gefüllt, die sich auch zur Laufzeit auf dem Stack befinden würden. Der Inhalt dieser Elemente ist natürlich nicht ohne weiteres zu bestimmen, ihm wird daher keine Variable zugeordnet.

Ein zweites Hindernis für eine „straight-forward“ Implementierung des schedulings ist der etwas tückische Befehl `iinc`. Da er den Inhalt einer lokalen Variable verändert, ohne den Umweg über den Stack zu nehmen, muß beim Erzeugen des Stackabbildes bei diesem Befehl die Information über eventuell auf dem Stack vorhandene Werte der Variable gelöscht werden. Außerdem muß die Suche nach einem Partner für einen Ladebefehl (siehe Programm 4.3) abgebrochen werden, sobald dabei ein `iinc` für die Variable angetroffen wird. Das Problem mit `iinc` und das analoge für Schreibbefehle kann durch eine Transformation der Eingabe in die „static single assignment form“, bei der auf jede Variable nur einmal geschrieben werden kann, gelöst werden.

Ein letztes Detail der Implementierung betrifft die Organisation des Stack für die Emulation der Befehle. Analog zum Stack der JavaVM werden auch hier Werte der Datentypen `long` und `double` durch zwei Einträge im Stack dargestellt. Dies macht die Berechnung der Stacktiefe trivial, erschwert dafür aber andere Berechnungen.

Kapitel 5

Optimales DAG scheduling

Je ne cherche pas; je trouve.

Pablo Picasso

Im vorigen Kapitel wurde Koopmans Methode des stack scheduling vorgestellt. Um abschätzen zu können, ob die Effizienz dieser Methode für die in dieser Arbeit untersuchten Architekturen weiter verbessert werden könnte, wird in diesem Kapitel eine weitere Optimierungsmethode entwickelt. Sie versucht, für die Befehle eines basic block die optimale Anordnung zu finden, in welcher der Zugriff auf lokale Variablen minimiert werden kann.

Nach einigen grundsätzlichen Bemerkungen zu DAGs und ihrer Verwendung in der Praxis des Übersetzerbaues wird im nächsten Abschnitt der Aufbau eines DAG für einen basic block beschrieben und die dabei auftretenden Probleme erläutert. Der zweite Teil des Abschnittes behandelt die Retransformation des DAG in Maschinencode, die zu einem optimalen Ergebnis führt. Auch Techniken, die den Aufwand für die Suche nach dem Optimum reduzieren, kommen zur Sprache.

Der anschließende Abschnitt enthält einen groben Überblick über die wichtigsten Teile der Implementierung sowie einige interessante Details zu speziellen Problemen und ihrer Lösung für die beiden konkreten Architekturen, die in dieser Arbeit Verwendung finden. Die mit dem Ansatz erzielten Ergebnisse werden in Kapitel sechs dargestellt.

5.1 Nachteile des stack scheduling

Das gravierendste Problem des stack scheduling ist das Fehlen des „instruction scheduling“. Die Methode verläßt sich auf die Qualität der Befehlsanordnung aus der Eingabe und kann daher nicht feststellen, ob mit einer anderen Reihenfolge der Befehle nicht ein besseres Ergebnis erzielt werden kann. Das Ergebnis der Optimierung hängt daher nicht nur vom Algorithmus an sich, sondern ebensosehr vom vorher erfolgten instruction scheduling ab. Abschnitt 4.3.1 enthält daher auch eine Beschreibung des durch andere Werkzeuge erfolgten instruction scheduling für HYSTAMIN und die JavaVM.

Ein weiterer Kritikpunkt bezieht sich auf die Darstellung der Effektivität des stack scheduling in [Koo92]. Es werden nur relative Angaben gemacht ohne daß daraus ersichtlich würde, wie weit die Ergebnisse vom Optimum entfernt sind, also wieviel Raum noch für weitere Verbesserungen besteht.

Das optimale DAG scheduling ist entwickelt worden, um diese beiden Punkte zu klären. Es verwendet als zentralen Bestandteil das instruction scheduling und ermittelt dabei optimalen Code, der somit als Qualitätsmaß für das Ergebnis des stack scheduling dienen kann.

5.2 DAG scheduling

Eine im Übersetzerbau oft vorkommende Datenstruktur ist der *directed acyclic graph*, meist kurz DAG genannt. Der nächste Abschnitt erläutert, was es mit diesem Graphen auf sich hat. Danach wenden wir uns der Frage zu, wie der DAG für einen basic block erzeugt, und – noch wichtiger – wie er dazu benutzt werden kann, einen neuen basic block zu erzeugen, welcher in bezug auf Variablenzugriffe optimiert ist.

5.2.1 DAGs

Wie schon der Name *directed acyclic graph* verrät, handelt es sich hierbei um einen gerichteten Graphen, der keine Zyklen enthalten darf. Aus der Graphentheorie (nachzulesen etwa in [Chr86]) ist bekannt, daß jeder Graph G aus einer Menge V von Knoten, einer Menge E von Kanten und einer Funktion f besteht, die jedem Element aus E ein (in unserem Fall – da ein DAG gerichtet ist – geordnetes) Paar von Elementen aus V zuordnet. Eine Kante verbindet also zwei Knoten und wir nennen den Knoten am Startpunkt einer Kante *Elternknoten* des End- oder *Kindknotens*. Drei Arten von Knoten lassen sich unterscheiden:

Wurzelknoten besitzen keine Elternknoten, stehen also an der Spitze der „Knotenhierarchie“.

Innere Knoten haben sowohl Kind- als auch Elternknoten.

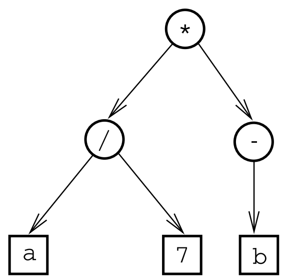
Blattknoten („Blätter") besitzen keine Kindknoten und stellen damit das Gegenstück zu Wurzelknoten dar.

Jeder Knoten stellt darüberhinaus die Wurzel eines Teilgraphen dar; dieser enthält alle Knoten, die vom Wurzelknoten aus über die gerichteten Kanten erreichbar sind.

Im Übersetzerbau wird ein DAG üblicherweise zur Darstellung eines Ausdruckes samt seiner Teilausdrücke verwendet. Jeder innere Knoten des DAG symbolisiert dabei einen Operator, der auf seine Kindknoten angewendet wird, Blattknoten stellen Variablen und Konstanten dar. Jeder Knoten bzw. der ihm zugeordnete Teilgraph repräsentiert nun einen Teil des vollständigen Ausdrucks.

Als Beispiel möge hier der Ausdruck `(a / 7) * (-b)` dienen, dessen DAG folgendermaßen aussieht (die Wurzel des DAG wird traditionellerweise – entgegen dem Vorbild eines Baumes in der Natur – oben dargestellt):

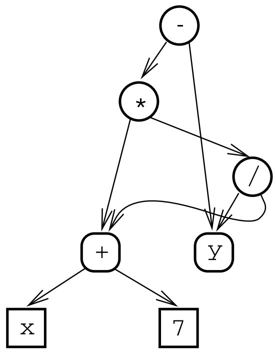
Beispiel 5.1 DAG für den Ausdruck `(a / 7) * (-b)`



Blätter werden in dieser Darstellung durch Quadrate symbolisiert, innere Knoten durch Kreise. Zu beachten ist, daß die Kindknoten eines Knotens in einer festgesetzten Reihenfolge von links nach rechts (entsprechend der Position der Operanden im Ausdruck) anzuschreiben sind. So wäre es in obigem Beispiel fatal, die Argumente des Knotens für die Division zu vertauschen, weil diese Operation bekannterweise nicht kommutativ ist. Unäre Operatoren (wie das unäre Minus in unserem Beispiel) besitzen nur einen Kindknoten.

Der entscheidende Unterschied zwischen einem DAG und einem Syntaxbaum ist, daß ein Knoten in einem DAG mehrere Elternknoten besitzen kann, während jeder Knoten in einem Syntaxbaum Kindknoten maximal eines Elternknotens ist. Mehrfache Elternknoten in einem DAG treten genau dann auf, wenn der dem Kindknoten zugeordnete Teilgraph einen Ausdruck repräsentiert, der in dem übergeordneten Ausdruck öfter als einmal vorkommt („common subexpression"). In Beispiel 5.2 ist der DAG für einen etwas komplexeren Ausruck angegeben.

Beispiel 5.2 DAG für den Ausdruck `(x + 7) * (y / (x + 7)) - y`



Sowohl der Knoten für die Variable `y` als auch der für die Addition repräsentieren hier Teilausdrücke, die im Gesamtausdruck öfters (im Beispiel je zweimal) vorkommen. In der Abbildung werden diese „shared nodes“ durch ein Quadrat mit abgerundeten Ecken symbolisiert.

Nun besteht ein basic block im allgemeinen nicht nur aus einem arithmetischen Ausdruck, sondern beinhaltet auch Zuweisungen, Funktionsaufrufe, eventuell einen abschließenden Sprungbefehl usw. Um eine Zuweisung in einem DAG ausdrücken zu können, führen wir folgende Erweiterung der grundlegenden Struktur ein: Jeder Knoten enthält zusätzlich die Information, welchen Variablen der Wert zugewiesen wurde, der durch den dem Knoten entsprechenden Teilausdruck berechnet wird. Diese Information wird benötigt, um bei der Transformation des DAG in sequentiellen Maschinencode die Variablen wieder mit dem richtigen Wert belegen zu können.

In der Abbildung wird dieser Sachverhalt dadurch veranschaulicht, daß neben dem Knoten ein strichliertes Rechteck die Namen aller Variablen enthält, denen sein „Wert“ zugewiesen wird. Weiters benötigen wir eine Datenstruktur, die für jede Variable angibt, welcher Knoten (falls ein solcher existiert) den aktuellen Wert der entsprechenden Variablen „enthält“, d.h. welcher Knoten den Wert symbolisiert, der der Variable zuletzt zugewiesen wurde.

Eine zweite Erweiterung der Struktur eines DAG sind Kanten, die zusätzliche Abhängigkeiten zwischen den Knoten darstellen und z.B. verwendet werden, um die korrekte Reihenfolge von Funktionsaufrufen sicherzustellen. Solche Kanten werden in der Abbildung als strichlierte Pfeile dargestellt und in den folgenden Abschnitten als „(zusätzliche) Abhängigkeitskanten“ bezeichnet, falls eine Unterscheidung zwischen ihnen und den Kanten, die aus dem Datenfluß der Ausdrücke in der Quellsprache entstanden sind, erwünscht ist. Die Verwendung der beiden eben beschriebenen Erweiterungen wird in den Ausführungen zur Erzeugung eines DAG im nächsten Abschnitt erläutert.

5.2.2 Erzeugung des DAG

Dieser Abschnitt beschreibt die Vorgehensweise, wie aus einem basic block (eigentlich den Befehlen, aus denen er besteht), ein DAG erzeugt werden kann. Der DAG wird später optimiert und dann wieder in eine Sequenz von Befehlen umgewandelt – das alles natürlich unter der Bedingung, daß die Semantik des optimierten Codes und die des ursprünglichen äquivalent ist.

Die grundsätzliche Vorgehensweise beim Aufbau des DAG besteht wiederum in einer symbolischen Ausführung des Codes. Der Stack für die Simulation enthält dabei die Knoten, deren Teilgraph zur Laufzeit den entsprechenden Stackwert berechnet.

Für jeden Befehl im basic block wird ein neuer Knoten erzeugt, der ebensoviele Kindknoten besitzt, wie Stackelemente vom Befehl als Argumente verwendet werden. Die Kindknoten werden vom Simulationsstack genommen, in dem danach der eben kreierte Knoten gespeichert wird, falls die Instruktion ein Ergebnis auf dem Laufzeitstack liefert. Werden common subexpressions gefunden, dann wird der bereits existierende Knoten verwendet, anstatt einen neuen zu erzeugen.

Als erstes Beispiel wird der wohlbekannte basic block aus Abschnitt 4.2.1 verwendet, hier der Einfachheit wegen in JavaVM-Syntax (Beispiel 5.3).

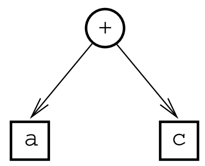
Beispiel 5.3 JavaVM Code für Beispiel 4.2

```
iload_1    ( -- )       % a @
iload_3    ( a -- )     % c @
iadd       ( a c -- )   % +
istore_2   ( a+c -- )   % b !
iload_2    ( -- )       % b @
iconst_5   ( b -- )     % 5
iadd       ( b 5 -- )   % +
istore_1   ( b+5 -- )   % a !
iload_2    ( -- )       % b @ ('dead')
iload_1    ( b -- )     % a @
iadd       ( b a -- )   % +
istore_3   ( b+a -- )   % c !
```

Die zwei ersten Befehle laden je eine lokale Variable und können sofort durch zwei Blattknoten repräsentiert werden. Die darauffolgende Addition verwendet diese als Kindknoten und der DAG für die ersten drei Befehle ist demnach trivial, wie aus Beispiel 5.4 ersichtlich ist.

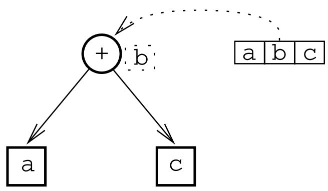
Der nächste Befehl speichert das Ergebnis der Addition in der Variablen `b`. Anstatt nun einen Knoten für „Speichern“ einzuführen, wird dem Knoten für die Addition die Information hinzugefügt, daß er den

Beispiel 5.4 DAG für die Befehle 1–3 aus Beispiel 5.3



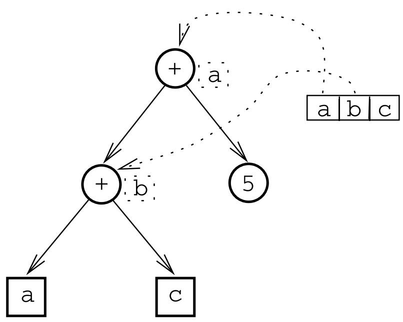
aktuellen Wert der Variablen b enthält. Außerdem wird in der Datenstruktur, die Variablen auf die entsprechenden Knoten abbildet, festgehalten, daß der Knoten für die Addition b repräsentiert. In Beispiel 5.5 wird die Datenstruktur durch eine Reihe von Variablen dargestellt, von denen ein Pfeil zu dem Knoten führt, der ihren Wert berechnet.

Beispiel 5.5 DAG für die Befehle 1–4 aus Beispiel 5.3



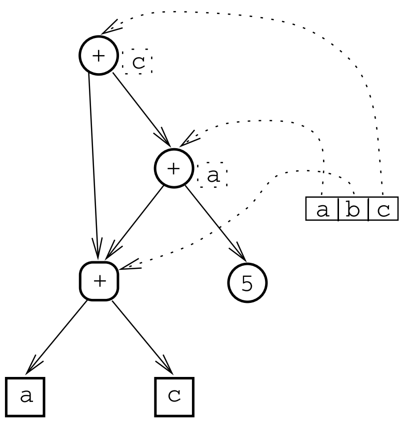
Beim nachfolgenden Ladebefehl für b wird nun – bevor ein neuer Knoten erzeugt wird – in der Datenstruktur nachgesehen, ob der Wert von b bereits in einem Knoten „bereitsteht". Da dies der Fall ist, kann dieser Knoten wiederverwendet werden. Die nachfolgende Konstante muß durch einen neuen Knoten dargestellt werden, und die zweite Addition wird zum Elternknoten dieser beiden Knoten. Der darauffolgende Speicherbefehl erzeugt wiederum keinen neuen Knoten, sondern fügt die entsprechende Information im Knoten für die Addition sowie in der globalen Datenstruktur hinzu, wie Beispiel 5.6 entnehmen zu ist.

Beispiel 5.6 DAG für die Befehle 1–8 aus Beispiel 5.3



Als nächstes wird wieder die Variable b geladen, wobei der Befehl mit der Markierung für nicht mehr verwendete Variablen versehen ist. Das bedeutet, daß wiederum der bereits vorhandene Knoten benutzt wird. Allerdings kann an dem Knoten die Information, daß sein Wert `b` zugewiesen wird, gelöscht und damit beim späteren Abbau des DAG dafür gesorgt werden, daß kein unnötiger Speicherbefehl generiert wird. Analog wird auch der Knoten für die Variable `a` wieder benutzt und ein weiterer Knoten für die dritte Addition erzeugt. Der Speicherbefehl wird wie die vorhergehenden behandelt, und der endgültige DAG für den basic block besitzt das Aussehen wie in Beispiel 5.7.

Beispiel 5.7 DAG für Beispiel 5.3



Um die zweite der genannten Erweiterungen demonstrieren zu können, wird im weiteren ein komplexeres Beispiel eines DAG untersucht. Das Funktionsfragment aus Beispiel 5.8 soll dabei zur Veranschaulichung dienen. In JavaVM-Code wird daraus Beispiel 5.9 (`a` ist die lokale Variable mit Index 0, `b` hat den Index 1 usw.).

Beispiel 5.8 C-Code für einen basic block

```
...
int a, b, c, d;
...
a = d = b + 2;
f(a); /* Deklariert als 'void f(int x)' */
c = d + 4;
g(b); /* Deklariert als 'void g(int x)' */
d = a + 4;
if (a > 0) ···;
...
```

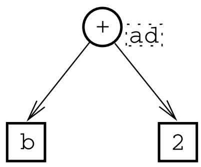
Der Aufbau des DAG für die ersten sechs Befehle birgt keine neuen Elemente, das Ergebnis ist aus Beispiel 5.10 ersichtlich.

Der Funktionsaufruf `f(a)` (Befehl 8) verwendet als Argument die Variable `a`, deren Wert im Knoten der Addition „bereitliegt". Auch die Addition `d + 4` und die Zuweisung des Resultates an `c` bringt keinerlei

Beispiel 5.9 JavaVM-Code für Beispiel 5.8

```
1  iload_1                    % b @
2  iconst_2                   % 2
3  iadd                       % b+2
4  dup
5  istore_3                   % d = b+2
6  istore_0                   % a = b+2
7  iload_0                    % a @
8  invokestatic 0x00 0x07     % f(a)
9  iload_3                    % d @
10 iconst_4                   % 4
11 iadd                       % d+4
12 istore_2                   % c = d+4
13 iload_1                    % b @
14 invokestatic 0x00 0x08     % g(b)
15 iload_0                    % a @
16 iconst_4                   % 4
17 iadd                       % a+4
18 istore_3                   % d = a+4
19 iload_0                    % a @
20 ifle 0x00 0x09             % if (a > 0) ...
```

Beispiel 5.10 DAG für die Befehle 1–6 aus Beispiel 5.9

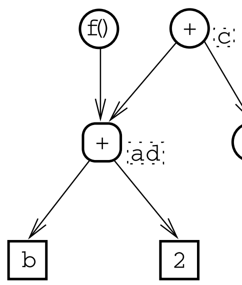


neue Erkenntnisse; wir gelangen für die ersten 12 Befehle zum DAG aus Beispiel 5.11.

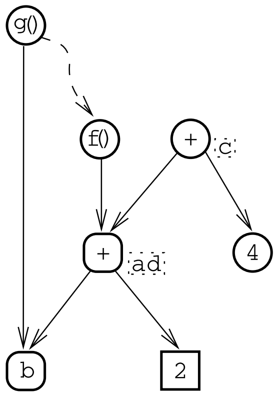
Auch der zweite Funktionsaufruf scheint auf den ersten Blick ganz harmlos. Sein Argument, die Variable `b`, wurde schon einmal geladen, der entsprechende Knoten kann also wiederverwendet werden. Allerdings darf die Methode `g()` nicht vor der Methode `f()` aufgerufen werden, da dies die Semantik des gesamten Programmes ändern könnte (man denke an die Manipulation von globalen bzw. „member"-Variablen).

Daher ist noch eine Kante vom Knoten für `g()` zum Knoten für `f()` notwendig. Diese Kante besagt, daß im Code, der später aus dem DAG erzeugt werden wird, zuerst der Befehl für `f()` und erst danach der für `g()` erzeugt werden darf. Der DAG für die ersten 14 Befehle sieht nun wie in Beispiel 5.12 aus.

Beispiel 5.11 DAG für die Befehle 1–12 aus Beispiel 5.9



Beispiel 5.12 DAG für die Befehle 1–14 aus Beispiel 5.9



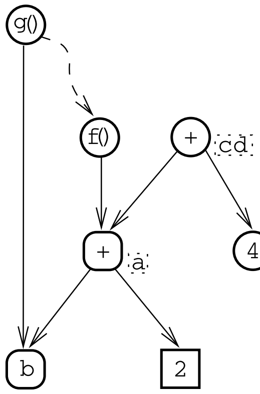
Betrachten wir nun die nächsten drei Befehle: Ein Knoten für die Variable a findet sich im DAG ebenso wie ein Knoten, der diesen mit der Konstante 4 addiert. Anstatt nun einen neuen Knoten für die Addition zu erzeugen, kann der bereits vorhandene wiederverwendet werden.[1]

Da der darauffolgende Befehl den errechneten Wert an die Variable d zuweist, wird dem Knoten diese Information hinzugefügt. Hier ist allerdings zu beachten, daß dadurch die Information beim Knoten, der bisher d repräsentierte, obsolet wird und d dort nicht mehr gespeichert werden muß. Diese Information am „alten" Knoten wird also gelöscht und außerdem die globale Datenstruktur, die lokale Variable auf entsprechende Knoten abbildet, auf den neuesten Stand gebracht (in den Abbildungen der Klarheit halber weggelassen). Das Ergebnis dieser Bemühungen ist in Beispiel 5.13 zu finden.

Es fehlt nun noch der letzte Befehl, der bedingte Sprung. Sein Argument, die Variable a, muß ebenfalls nicht neu geladen werden, ein Knoten mit ihrem Wert ist im DAG enthalten. Es wird ein neuer Knoten erzeugt, der den Knoten für a als Kindknoten besitzt. Es muß jedoch

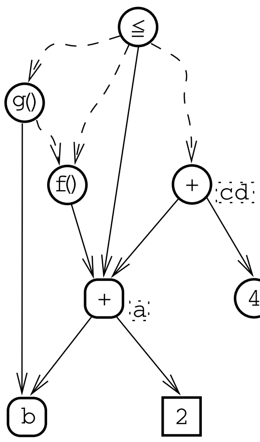
[1]Tatsächlich haben wir im basic block einen gemeinsamen Teilausdruck gefunden, da die Variablen a und d zum Zeitpunkt der Addition den gleichen Wert besitzen.

Beispiel 5.13 DAG für die Befehle 1–19 aus Beispiel 5.9



auch sichergestellt werden, daß der Sprungbefehl im basic block als letzter Befehl ausgeführt wird, da sich ansonsten die Semantik des Programmes ändern könnte. Diese Bedingung kann dadurch sichergestellt werden, daß Abhängigkeitskanten vom Knoten für den Sprungbefehl auf alle bisherigen Wurzelknoten erzeugt werden. Der endgültige DAG für den basic block aus Beispiel 5.8 wird in Beispiel 5.14 dargestellt.

Beispiel 5.14 DAG für Beispiel 5.9



Nachdem nun im großen und ganzen die Methode vorgestellt wurde, wie die Befehle eines basic block in einen DAG umgewandelt werden können (nähere Details zu den verwendeten Algorithmen finden sich in Abschnitt 5.3), wenden wir uns im nächsten Abschnitt der Frage zu, wie der DAG verändert werden kann, um eine für unsere Zwecke optimale Befehlsfolge zu finden. Im wesentlich geht es dabei darum, eine Reihenfolge anzugeben, in der die Knoten des DAG in die Befehle, die sie repräsentieren, umgewandelt werden.

5.2.3 Optimales scheduling des DAG

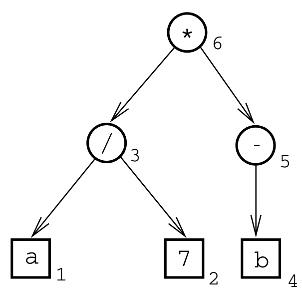
Wie kann nun der im vorigen Schritt erzeugte DAG dazu benutzt werden, den basic block zu optimieren? Hier kommt zum Tragen, daß die Reihenfolge, in der die Befehle des basic block ausgeführt werden müssen, nicht vollständig festgelegt ist. Durch geschicktes Umordnen der Befehle könnte eventuell die Nutzungsdauer der lokalen Variablen so eingeschränkt werden, daß sie öfters auf dem Stack gespeichert werden können anstatt im Speicher residieren zu müssen. Diese Tatsache stellt auch den entscheidenden Vorteil dieser Methode gegenüber Koopmans stack scheduling dar.

Was bedeutet nun „scheduling“ in Zusammenhang mit DAGS? Dazu stellen wir fest, daß die Kanten des DAG eine Reihenfolge vorgeben, in der sich dadurch verbundene Knoten (bzw. die durch sie repräsentierten Befehle) befinden müssen. So ist es etwa nicht sinnvoll, eine Addition auszuführen, ehe ihre Operanden auf dem Stack liegen.

Für jeden DAG läßt sich sofort die Menge $S(V)$ aller „gültigen“ seriellen Anordnungen seiner Knoten angeben: Führt eine Kante vom Knoten V_1 zu Knoten V_2, so muß V_2 in der Anordnung vor V_1 zu liegen kommen. Vorerst einmal sind sonst keine Bedingungen zu erfüllen. Ein Element der Menge $S(V)$ nennen wir (gültiges) „schedule“ des DAG.

Speziell für Stackarchitekturen gilt nun, daß nicht jede gültige Anordnung sinnvoll ist bzw. zu einem „guten“ Ergebnis führt.[2] Betrachten wir hierzu wieder den DAG aus Beispiel 5.1, zum leichteren Verständnis sind die Knoten jedoch numeriert (Beispiel 5.15).

Beispiel 5.15 DAG aus Beispiel 5.1 (Knoten numeriert)



In diesem Fall wäre die Reihenfolge 1-2-4-3-5-6 gültig. Allerdings läßt sich leicht erkennen, daß dieses schedule ungünstig für unsere Zwekke ist; wenn die Division aus Knoten 3 an die Reihe kommt, liegen ihre Argumente im Stack unter der Konstante aus Knoten 4 verborgen. Es

[2]In geringerem Maße ist diese Aussage auch für Registermaschinen gültig; allerdings können dort viele Komplikationen durch den wahlfreien Zugriff auf die Register vermieden werden. Erst wenn die Anzahl der Register für die Belegung mit Teilausdrücken nicht mehr ausreicht, treten Probleme auf.

muß daher hier der Stack umgeordnet werden, um einen Fehler in der Semantik des Codes zu vermeiden. Klarerweise stellt für diesen DAG die Reihenfolge 1-2-3-4-5-6 eine bessere Alternative dar, weil jeder Befehl seine Operanden schon korrekt auf dem Stack vorfindet.

Es ist für diesen speziellen DAG nun leicht einsichtig, daß die zweite genannte Reihenfolge auch das Optimum darstellt, das heißt, dieses schedule ist das „optimale" Element der Menge $S(V)$. Das Ziel für die weiteren Ausführungen ist nun, dieses beste schedule für jeden beliebigen DAG zu finden. Ist also der DAG – und damit implizit auch die Menge $S(V)$ – gegeben, so wird dasjenige Element $o \in S(V)$ gesucht, das folgende Bedingung erfüllt:

$$\forall s \in S(V) : o \trianglelefteq s$$

Die Relation $\trianglelefteq$ kann dabei beliebig gewählt werden, um die gewünschte Optimierung zu erreichen. Für unsere Zwecke wählen wir $\trianglelefteq$ als die Relation $\leq$ der Ausführungszeiten für den Maschinencode, der durch ein schedule repräsentiert wird. Dabei muß natürlich die Zeit, die für die Ausführung eines Befehls angenommen wird, entsprechend gewählt werden (mehr dazu ist in Abschnitt 5.3 zu finden).

Nachdem nun festgelegt ist, wonach es zu suchen gilt, stellt sich sofort die Frage, wie das Gesuchte möglichst schnell gefunden werden kann. Man beachte dabei, daß die Menge $S(V)$ im allgemeinen sehr groß werden kann. Im schlimmsten Fall gilt $\mid S \mid = n!$ – er tritt ein, wenn keine Abhängigkeiten zwischen den Knoten bestehen und somit jede Aneinanderreihung ein gültiges schedule ergibt.

Die erschöpfende Suche nach dem besten dieser Elemente stößt also recht bald an die Grenzen des Bewältigbaren, für größere DAGs ist das Problem in dieser simplen Form schlicht unlösbar.

Unter diesen Umständen (ein „Abgrasen" des gesamten Suchraumes ist aus praktischen Gründen nicht machbar) erweisen sich Heuristiken üblicherweise als recht hilfreich. Sie bestehen im wesentlichen aus „Daumenregeln", wie eine gute Lösung relativ einfach gefunden werden kann, und bewähren sich im allgemeinen recht gut, allerdings können sie keine Garantie liefern, wirklich das Optimum zu finden – dafür liefern sie eine Lösung mit „vernünftigem" Aufwand. [Smo91] enthält eine Gegenüberstellung wichtiger Heuristiken für das Problem des DAG scheduling, deren Hauptaugenmerk auf superskalaren Architekturen liegt (zur Ausnutzung von „instruction level parallelism").

In dieser Arbeit kann aus zwei Gründen keine Heuristik für das DAG scheduling für Stackarchitekturen angegeben werden: Erstens erwies sich die Suche nach einfachen und trotzdem effizienten Regeln als fruchtlos, und zweitens besteht eines der Ziele der Arbeit darin, ein

Abschätzung angeben zu können, wie nahe Koopmans stack scheduling dem Optimum kommt. Klarerweise ist dazu die Kenntnis des Optimums unabdingbar, und eine Heuristik liefert eben nicht in jedem Fall die beste Lösung.

In der Praxis kann auch die vorliegende Implementierung nicht garantieren, daß das Optimum gefunden wurde. Nach Ablauf einer „angemessenen“ Zeitspanne (vom Benutzer vorgegeben) wird das DAG scheduling abgebrochen, um allzulange Laufzeiten zu vermeiden. Es wurde dann nicht der gesamte Lösungsraum durchsucht, und daher ist es unmöglich zu sagen, ob das optimale schedule gefunden wurde. Konnte überhaupt keine Lösung gefunden werden, muß für den basic block die Alternative des stack scheduling verwendet werden.

Von den 3742 einzeln optimierten DAGs der JGL-Klassen (siehe Abschnitt 6.1) konnte bei 60 (das sind 1,6%) innerhalb einer Zeitspanne von fünf Sekunden gar keine Lösung gefunden werden. Bei 99 Graphen oder 2,7% konnte innerhalb dieser Zeitspanne nicht der gesamte Lösungsraum durchsucht werden, d.h. es bleibt offen, ob die gefundene Lösung wirklich das Optimum darstellt.

Um den Aufwand für die Suche nach dem optimalen schedule in Grenzen zu halten, sind einige Techniken recht hilfreich, die im folgenden nun erläutert werden. Genauere Details zur Implementierung sind in Abschnitt 5.3 zu finden.

Branch and Bound

Eine recht einfache und trotzdem wirkungsvolle Methode, den Aufwand der Suche nach dem Optimum einzugrenzen, ist das sogenannte „branch and bound“ ([Sed88]). Dieses Verfahren beruht darauf, daß es keinen Sinn macht, eine bestimmte Lösung weiter zu verfolgen, wenn absehbar ist, daß sie nicht besser als die beste bereits gefundene ist. Für unsere Zwecke ist es offensichtlich günstig, die angenommene Ausführungszeit als Maß für die Güte eines schedule anzunehmen.

Eine erste grobe Abschätzung dient als „bisheriges Optimum“, bevor eine wirkliche Lösung gefunden wurde. Die Abschätzung darf natürlich nicht kleiner sein als das reale Optimum, da ansonsten die optimale Lösung als „zu schlecht“ eliminiert würde. Eine Formel zur Berechnung der Abschätzung ist auf Seite 69 angegeben.

Sobald eine erste Lösung feststeht, kann ihre angenommene Ausführungszeit als Schranke angesehen wird, die vielversprechende schedules von nicht weiter zu verfolgenden trennt. Im allgemeinen besteht ein schedule aus der Menge der bereits angeordneten Knoten sowie dem Teil des DAG, der noch gereiht werden muß. Für die erste Menge kann die Ausführungszeit leicht errechnet werden, zusammen mit einer einfachen Abschätzung für den Rest-DAG (die wiederum nicht zu hoch liegen darf, damit möglicherweise gute Lösungen nicht zu früh ausgeschieden werden) erhalten wir damit eine Näherung der Qualität der Lösung. Diese Näherung ist somit eine untere Schranke der Laufzeit des halbfertigen schedule.

Ist die abgeschätzte Laufzeit größer als die des bisher besten gefundenen schedule, wird die „schlechte" Lösung (und damit alle weiteren, die mit der gleichen Anordnung der bisher gereihten Knoten beginnen) nicht mehr weiterverfolgt. Weil die Abschätzung für ein unvollständiges schedule eine untere Schranke für die Laufzeit darstellt, werden dabei nur schedules eliminiert, welche die bisher beste Lösung nicht übertreffen können.

Beschränkung auf sinnvolle Lösungen

Unter „sinnvoll" soll hier verstanden werden, daß die Operanden für einen Befehl möglichst schon richtig auf dem Stack liegen – daß sie überhaupt dort zu finden sind, ist eine Folge der Gültigkeit eines schedule.

Ist K der aktuelle Knoten beim DAG scheduling, so liegen die Operanden für den K entsprechenden Stackbefehl schon auf dem Stack (aufgrund der Gültigkeit des schedules ist sichergestellt, daß die Kindknoten von K schon im schedule enthalten sind). Der Befehl für K erwartet seine Operanden allerdings in der korrekten Reihenfolge als oberste Elemente des Stack. Ist es nun nicht möglich, die Operanden mit Stackmanipulationen in die korrekte Position zu bringen, so wird das schedule als „nicht sinnvoll" gekennzeichnet und von der weiteren Suche nach dem Optimum ausgenommen.

Die Alternative dazu wäre, den Wert des Operanden nach seiner Berechnung in einer temporären lokalen Variable zu speichern und vor dem scheduling von K wieder auf den Stack zu laden. Da aber die Stoßrichtung der Optimierung ja gerade in der Eliminierung von Variablenzugriffen liegt, wäre dieses Vorgehen kontraproduktiv. Unter Umständen kann es zwar vorkommen, daß durch ein temporäres Speichern in einer Variable andere Zugriffe eliminiert werden könnten; dies wird jedoch in der vorliegenden Arbeit außer acht gelassen.

Eine andere Einschränkung gilt für das scheduling von „shared nodes", die schon einmal im schedule vorkommen (d.h. der Wert des Teilgraphen wurde schon berechnet). Anstatt den Wert noch einmal zu berechnen (was die Erzeugung von Code für den gesamten Teilgraphen nach sich ziehen würde), wird eine Kopie des schon ermittelten Wertes verwendet. Dazu kann entweder ein entsprechender Kopierbefehl oder – falls der Wert in einer Variablen gespeichert ist – ein Ladebefehl in das schedule eingefügt werden.

Eliminieren von Teilbäumen

Diese Optimierung beruht auf folgender Überlegung: In einem DAG treten immer wieder „einfache“ Teilgraphen auf, nämlich Bäume[3] mit gewissen Einschränkungen in bezug auf zusätzliche Abhängigkeitskanten (dazu weiter unten mehr). Solche Teilgraphen weisen klarerweise nur ein sinnvolles schedule auf (für eine Stackarchitektur natürlich ein „depth first postorder traversal“ des Graphen, weil dadurch die Operanden für jeden Befehl schon in korrekter Anordnung auf dem Stack liegen und keine Stackmanipulationen nötig sind) und können schon vor dem eigentlichen Beginn des scheduling eliminiert und auf einen einzigen Knoten des DAG reduziert werden. Dadurch reduziert sich die Anzahl der Knoten für das scheduling und damit auch die Größe des Suchraumes.

Wie sieht nun ein genaues Kriterium für die Erkennung eines eliminierbaren Teilgraphen aus? Vorerst können alle Knoten von einer weiteren Untersuchung ausgenommen werden, deren Teilgraph einen „shared node“ enthält – der Teilgraph besitzt dann nämlich sicher keine Baumstruktur. Es werden also alle Kindknoten und deren Kindknoten usw. bis hin zu den Blättern des Graphen auf Knoten durchsucht, die mehr als einen Elternknoten besitzen. Findet sich ein solcher, dann kann der Teilgraph nicht eliminiert werden. Man beachte hierbei, daß die Wurzel des Teilgraphen sehr wohl ein „shared node“ sein darf, da sie im DAG „stellvertretend“ für den wegrationalisierten Teilbaum verbleibt.

Allerdings muß ein Knoten bzw. der durch ihn definierte Teilgraph eine weitere Bedingung erfüllen, um für die Eliminierung in Frage zu kommen: Da jeder Knoten im Teilgraphen auch in Abhängigkeit zu anderen Knoten stehen kann, müssen diese zusätzlichen Abhängigkeiten so beschaffen sein, daß sie durch die Eliminierung nicht verletzt werden können.

Um die folgenden Untersuchungen zu vereinfachen, können erst einmal alle redundanten Kanten des DAG entfernt werden. Eine Kante von Knoten V_a zu Knoten V_e ist genau dann redundant, wenn es im DAG die Knoten V_1 bis V_n ($n \geq 1$) mit Kanten von V_i zu V_{i+1} ($1 \leq i \leq n-1$) und einer Kante von V_a zu V_1 sowie einer Kante von V_n zu V_e gibt. Durch diese Bedingung ist sichergestellt, daß auch nach der Entfernung der Kante von V_a zu V_e der Endknoten vor dem Anfangsknoten gereiht werden muß.

Es werden nun für alle Knoten im zu untersuchenden Teilgraphen jene Knoten untersucht, die von ihnen abhängen, also in einem gültigen schedule *vor* ihnen gereiht werden müssen. Liegen diese Knoten im Teilgraph, so gibt es kein Problem, da ein Baum für gültigen C- bzw.

[3]Ein Baum ist ein Graph ohne „shared nodes“.

Java-Code immer in einem depth first postorder traversal behandelt werden kann, ohne Abhängigkeiten zu verletzen.

Liegt der abhängige Knoten jedoch nicht im Teilgraphen, so besagt eine einfache Faustregel,[4] daß der Teilgraph ohne Gefahr eliminiert werden kann, wenn der abhängige Knoten (das könnte ein schon früher eliminierter Teilgraph sein) keine Kindknoten, keine Elternknoten und auch keine von ihm wiederum abhängigen Knoten besitzt. Durch diese Bedingung ist sichergestellt, daß dieser einzelne Knoten in jedem Fall vor dem eliminierten Teilbaum zu liegen kommen kann – damit ist auch die Gültigkeit der Lösung gewährleistet.

Ist nun sichergestellt, daß ein Teilgraph reduzibel ist, so sind folgende Schritte zu ergreifen, um ihn zu eliminieren: Alle Abhängigkeiten von und zu Knoten im Teilgraphen werden auf den Wurzelknoten des Teilgraphen „umgelenkt“ – dies sichert eine korrektes schedule. Danach können die Knoten des Teilgraphen gelöscht werden, wobei natürlich die ihnen entsprechenden Befehle erzeugt und im Wurzelknoten des Teilgraphen gespeichert werden müssen, damit sie bei der endgültigen Transformation des DAG in Maschinencode an der geeigneten Stelle eingefügt werden können.

Weil der Code des eliminierten Teilbaumes optimal ist und der Teilgraph nur über einen Knoten mit dem Rest des Graphen zusammenhängt (d.h. die Optimierung kann für beide Teile getrennt erfolgen, da sie einander nicht beeinflussen), kann der Teilbaum vorweg optimiert und der optimale Code dann in das optimale schedule für den Rest-DAG eingefügt werden. So kann die Eliminierung durchgeführt werden, ohne befürchten zu müssen, daß dadurch das optimale schedule aus dem Suchraum entfernt wird. Außerdem berücksichtigt der Algorithmus alle Abhängigkeitskanten im Graphen und stellt somit die Gültigkeit des Resultates sicher.

Aufteilung des DAG

Der DAG für einen basic block muß nicht immer zusammenhängend sein, er kann auch aus mehreren Teilgraphen bestehen, die einander nicht berühren und daher in keiner festen Reihenfolge zueinander stehen. Ein einfacher Fall ist der DAG für den basic block aus Beispiel 5.16 (vergleiche Beispiel 5.17).

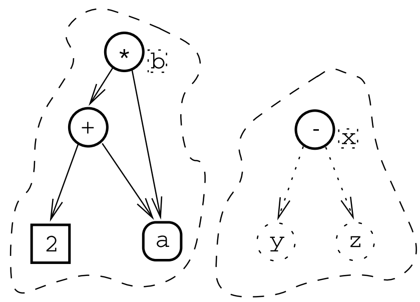
Die Kindknoten des Wurzelknotens für die zweite Zuweisung sind gepunktet dargestellt, da sie durch die im vorigen Abschnitt beschriebene Optimierung eliminiert wurden und beim scheduling nicht mehr beachtet werden müssen.

[4]Hier sind natürlich beliebig komplexe Untersuchungen möglich. Allerdings dürfte sich der Aufwand dafür in der Regel nicht lohnen.

Beispiel 5.16 C-Code für einen basic block

```
...
int a, b, x, y, z;
...
b = 2 + a;
b *= a;
x = y - z;
...
```

Beispiel 5.17 DAG für Beispiel 5.16



Es ist klar ersichtlich, daß der Graph aus Beispiel 5.17 in zwei unabhängige Teilgraphen zerfällt. Anstatt nun eine Lösung für den gesamten Graphen zu ermitteln, können die Teile getrennt behandelt und die beiden schedules in beliebiger Reihenfolge aneinandergefügt werden. Weil die beiden Graphen einander nicht beeinflussen und für jeden optimaler Code erzeugt wird, muß auch die Aneinanderreihung der beiden Codeteile optimal mit Bezug auf den gesamten Graphen sein.[5]

Das Verfahren ist sinnvoll, da die Summe des Aufwandes für das scheduling der Teilgraphen kleiner ist als der Aufwand für das des Gesamtgraphen. Zerfällt der Graph G mit n Knoten in die m Teilgraphen G_1 bis G_m, und haben diese jeweils n_1 bis n_m Knoten, so gilt (zumindest für große n_i) für die weiter oben vorgestellte maximale Größe des Suchraumes:

$$\sum_{i=1}^{m} n_i! \ll \left(\sum_{i=1}^{m} n_i\right)!$$

[5]Etwas anders stellt sich die Situation für superskalare Architekturen dar. Hier könnte durch eine „Verzahnung“ der Befehle der beiden Teilgraphen sehr wohl eine bessere Auslastung der Prozessorresourcen erzielt werden. Wie jedoch in der Einleitung erwähnt, sind superskalare Stackprozessoren selten anzutreffen (ein Beispiel dafür wäre der Inmos T9000).

Dynamic programming für Registermaschinen

[Kes96] stellt eine „dynamic programming"-Methode vor, die das Problem der Suche nach einem optimalen schedule für Registermaschinen zu lösen versucht. Die Hauptidee dabei ist, daß der Suchbaum der möglichen schedules im allgemeinen Teilbäume aufweist, die einander bis auf die Zuordnung der Register gleichen. Kennt man die Lösung für einen dieser Teilbäume, läßt sich daraus leicht diejenige für alle äquivalenten ableiten. Durch diese Technik kann der Aufwand für die Suche nach der optimalen Reihenfolge enorm reduziert werden – man beachte jedoch, daß „optimal" in dem Zusammenhang bedeutet, daß möglichst wenige Register verwendet werden, die genaue Zuordnung von Registern zu Teilausdrücken jedoch unerheblich ist, weil auf die Register wahlfrei zugegriffen werden kann.[6]

In dieser Tatsache liegt auch das Problem, das für Stackarchitekturen bei diesem Ansatz entsteht; es ist eben nicht unerheblich, an welcher Stelle im Stack die Teilausdrücke zu liegen kommen, weil der Zugriff auf den Stack implizit erfolgt. Es nutzt schließlich nicht allzuviel, möglichst wenige Elemente im Stack für eine Berechnung zu brauchen, wenn sie dafür andauernd verschoben werden müssen. Der Ansatz ist daher für das scheduling von Stackcode nicht verwendbar.

5.3 Implementierung

In diesem Abschnitt wird ein Überblick über die Implementierung der Algorithmen für das optimale DAG scheduling gegeben und noch einige Punkte erläutert, die sich dabei als wichtig erwiesen haben. Einige Bemerkungen zu speziellen Details der Implementierung für HYSTAMIN und die JavaVM runden den Abschnitt ab.

5.3.1 Allgemeines

Wie die meisten anderen, bearbeitet auch dieser Durchgang der Optimierung eine Funktion nach der anderen. Programm 5.1 stellt die grundlegende Vorgangsweise dar.

Im großen und ganzen ist der Algorithmus auf dieser Ebene eher unspektakulär, ein interessantes Detail ist jedoch durch den großen Aufwand für das scheduling bedingt. Bei großen und komplizierten Graphen kann es vorkommen, daß die Laufzeit für die Suche sehr hoch wird. Außerdem kann es für sehr komplexe Graphen zu Fällen kommen, in denen gar keine gültige Lösung gefunden werden kann (das

[6] Für Architekturen, deren Befehlssatz nicht orthogonal ist, stimmt diese Aussage nur bedingt.

Programm 5.1 Ablauf des DAG scheduling

```
comment: Eingabe: ein Funktion, in basic blocks aufgeteilt
         Ausgabe: die optimierte Funktion

for alle basic blocks do
    erzeuge den DAG für den block;
    führe einfache optimierungen durch;
    zerlege den DAG in unabhängige teile;
    for alle teile do
        setze timeout;
        suche optimale lösung;
        if keine lösung gefunden?
          then
                optimiere ganzen block mit stack scheduling;
                ignoriere weitere teile des blocks;
          else
                erzeuge code für die lösung;
        fi;
    end;
end;
```

folgt aus einer Einschränkung der Implementierung in Zusammenhang mit Knoten, die mehr als zwei Kindknoten besitzen, siehe auch Abschnitt 7.2).

Um diese Widrigkeiten abfangen zu können, ist es möglich, eine maximale Zeitspanne vorzugeben, in der das scheduling zu einem Ende kommen muß. Läuft diese Zeit ab, ohne daß eine Lösung gefunden wurde (oder wird überhaupt keine Lösung gefunden), gibt der Algorithmus auf und verweist den offensichtlich ziemlich komplexen Block an das stack scheduling. Danach müssen natürlich auch alle weiteren Teilgraphen des Blocks ignoriert werden, weil sie schon vom stack scheduling behandelt wurden – dieser pass behandelt ja einen basic block als Ganzes.

Wenden wir uns nun dem Aufbau des DAG für einen basic block zu; dieser Teil der Optimierung ist in Programm 5.2 dargestellt. Die für unsere Belange wichtigste Funktionalität befindet sich in den Abschnitten, die sich mit dem Laden und Speichern von lokalen Variablen befassen. Grundsätzlich gibt es auch hier wieder einen Stack, der das Verhalten des wirklichen Stack zur Laufzeit emuliert – allerdings sind seine Elemente nicht die numerischen Werte, sondern die Knoten des DAG, deren Teilgraphen die Werte berechnen.

Die Erkennung von Abhängigkeiten zwischen den Knoten ist großteils architekturabhängig und wird daher in den entsprechenden Abschnitten erläutert. Die Vereinfachungen des DAG anhand der im vorigen

Programm 5.2 Aufbau des DAG für einen basic block

```
comment: Eingabe: ein basic block
         Ausgabe: der DAG des basic block

stack ← empty;
for alle befehle des block do
   if befehl = laden einer lokalen variable
     then
          if ein knoten enthält den wert der variable?
            then
                 knoten ← existierender knoten;
                 if variable ist „dead“?
                   then lösche info über variable im knoten;
                 fi;
            else
                 knoten ← erzeuge neuen knoten;
                 verbinde variable → knoten in datenstruktur;
          fi;
          stack.push(knoten);
   fi;
   if befehl = speichern einer lokalen variable
     then
          if ein knoten enthält den wert der variable?
            then lösche info über variable im knoten;
          fi;
          knoten ← stack.pop();
          if variable ist nicht „dead“?
            then setze info über variable im knoten;
          fi;
          verbinde variable → knoten in datenstruktur;
   fi;
   if befehl = stackmanipulation
     then entsprechende manipulation auf stack;
   fi;
   if befehl = anderer befehl
     then
          kinder ← stack.pop(alle kindknoten);
          if kein knoten mit befehl und gleichen kindern?
            then knoten ← neuer knoten mit kindern;
            else knoten ← existierender knoten;
          fi
          if befehl hat ergebnis?
            then stack.push(knoten);
          fi;
   fi;
   erzeuge kanten für zusätzliche abhängigkeiten;
end;
```

Abschnitt beschriebenen Methoden und das Aufteilen des DAG in unabhängige Subgraphen enthält keine Besonderheiten und wird hier der Kürze der Darstellung geopfert. Stattdessen wenden wir uns der Suche nach einer optimalen Lösung zu.

Dafür sind vorerst einige Vorbereitungen erforderlich: Für das „branch and bound“ ist eine Abschätzung für die Laufzeit der Eingabe nötig (vergleiche dazu Abschnitt 5.2.3). Außerdem ist die Menge der Knoten, die im ersten Schritt zur Auswahl stehen, zu erstellen – es sind genau die Blattknoten des DAG (siehe Programm 5.3).

Programm 5.3 Initialisierung für das scheduling

comment: Eingabe: ein (Teil-)DAG
Ausgabe: eine optimales schedule der Knoten

best ← abschätzung für laufzeit;
ready ← blattknoten des DAG;
schedule(*ready*, *empty*, *best*);

Die eigentliche Suche nach dem optimalen schedule findet in der rekursiven Funktion `schedule()` in Programm 5.4 statt. Die Abbruchbedingung der Rekursion ist eine leere Menge der Knoten, die im nächsten Schritt gereiht werden können – tritt dieser Fall auf, so sind alle Knoten abgearbeitet. Auch eine Überschreitung des Zeitlimits führt zum Abbrechen der Rekursion, allerdings wird dann eine „leere“ Lösung als Zeichen dafür zurückgegeben.

Die eigentliche Funktionalität der Prozedur verteilt sich auf zwei Phasen. In der ersten wird für jeden Knoten, dessen Kind- und von ihm abhängige Knoten bereits gereiht wurden, und der für eine sinnvolle Lösung in Frage kommt (das wird von der Funktion `schedulenode()` geprüft, die im weiteren behandelt wird), ein neues schedule erzeugt. Ein schedule ist dabei die Struktur, die die genaue Reihenfolge der Knoten und andere wichtige Daten enthält:

- Einen Stack von Knoten, der den Laufzeitstack emuliert (ähnlich wie beim Aufbau des DAG).
- Für jeden Knoten in der Lösung ein Abbild des Stack nach der Ausführung der Befehle dieses Knotens (analog zum Stack im vorigen Punkt).
- Information, wie oft jeder Knoten bisher gereiht wurde (wichtig für „shared nodes“).
- Eine Abschätzung der Laufzeit für die Menge der schedules, die durch diese Anfangsreihenfolge festgelegt ist.

Programm 5.4 Rekursives DAG scheduling – `schedule()`

```
comment: Eingabe: ready: eine Menge von Knoten
                  schedule: ein vorläufiges schedule
                  best: die Laufzeit des besten schedule
         Ausgabe: bestes schedule ausgehend von schedule

if ready = empty
  then return schedule;
fi;
if timeout?
  then return empty;
fi;

schedules ← empty;
for alle knoten in ready do
    if keine kinder und abhängigkeiten mehr?
      then
           new ← schedule;
           newready ← ready;
           if schedulenode(knoten, new)?
             then
                  newready.add(elternknoten von knoten);
                  newready.add(abhängige knoten);
                  if knoten nicht „shared“? ∨
                    knoten nicht mehr gebraucht?
                    then newready.remove(knoten);
                  fi;
                  if abschätzung für new ≤ best
                    then schedules.add(new, newready);
                  fi;
           fi;
    fi;
end;

sortiere schedules nach abschätzung;
for alle elemente in schedules do
    sched ← schedule(element.ready, element.schedule, best);
    if sched ≠ empty ∧ laufzeit für sched ≤ best
      then best ← laufzeit für sched;
    fi;
end;

return bestes schedule;
```

Auch eine neue Menge von Knoten, die im nächsten Schritt gereiht werden können, muß erzeugt werden. Sie enthält alle Knoten der bisherigen Menge sowie die Elternknoten des gerade gereihten Knotens und die von ihm abhängigen Knoten. Hinzu kommt auch noch der Knoten selbst, wenn er noch nicht so oft gereiht wurde, wie er Elternknoten im DAG besitzt. Diese Vorgehensweise ist natürlich aufwendiger als die Alternative, jeden Knoten nur einmal zu reihen und durch Stackmanipulationen für die anderen Elternknoten bereitzustellen. Dafür erschließen sich dadurch bessere Optimierungen (besser gesagt, im Zusammenspiel mit `schedulenode()` findet die alternative Methode nicht immer eine optimale Reihenfolge).

Nun kommt wieder eine der im vorigen Abschnitt besprochenen Optimierungen, das branch and bound, zum Tragen: Ist abzusehen, daß die Lösungen, die durch die bisherige Reihenfolge vorgegeben sind, sicher schlechter ausfallen werden als die bisher beste, so werden sie nicht mehr weiter verfolgt. Andernfalls wird die Teillösung in die Menge der noch zu untersuchenden Lösungen aufgenommen.

Nachdem alle Knoten in „ready“ auf diese Weise behandelt worden sind, wird die Menge aller sinnvollen schedules aufsteigend nach der Abschätzung für ihre Ausführungszeit sortiert. Danach ruft sich die Funktion `schedule()` rekursiv auf. Als Rückgabewert ergibt sich entweder ein „leeres“ schedule oder die beste der Lösungen, die durch die Argumente vorgegeben war. Diese wird mit dem besten bisher gefundenen schedule verglichen; ist sie besser, wird sie als das neue Optimum vorgemerkt und der Grenzwert der Ausführungszeit angepaßt, um die Effizienz des „branch und bound“-Verfahrens zu steigern.

Nun fehlt noch die Beschreibung der Funktion `schedulenode()`, um den Überblick über die Implementierung abzurunden (Programm 5.5). Die Aufgabe dieser Funktion ist es, sinnvolle von nicht sinnvollen schedules zu unterscheiden (unter „sinnvoll“ wird hier die in Abschnitt 5.2.3 erwähnte Bedeutung verstanden, d.h. daß die Argumente für den Befehl im Knoten in die richtige Anordnung auf dem Stack gebracht werden können).

Ist der Knoten ein „shared node“ und schon einmal in der Lösung aufgetreten, so versucht der Algorithmus, ihn mittels der Funktion `scheduleshared()` an die oberste Position im emulierten Stack zu kopieren. Für Knoten mir nur einem Elternknoten wird erst geprüft, ob sich ihre Argumentknoten in der richtigen Reihenfolge auf dem Stack befinden. Wenn dies nicht der Fall ist, wird versucht, diesen Zustand mithilfe einfacher Stackmanipulationen herzustellen. Scheitert der Versuch, so ist die Lösung nicht sinnvoll und wird daher verworfen, war er hingegen erfolgreich, kann der Knoten dem schedule hinzugefügt werden – die Lösung wird in `schedule()` weiterverfolgt werden.

Programm 5.5 Scheduling eines Knotens – `schedulenode()`

comment: Eingabe: *knoten:* ein Knoten
schedule: das bisherige schedule
Ausgabe: das schedule erweitert um den Knoten

if *knoten* ist „shared“? $\wedge$ *knoten* schon in *schedule*?
then return *scheduleshared*(*knoten*, *schedule*);
fi;
if kindknoten nicht in richtiger position? $\wedge$
können nicht in richtige position gebracht werden?
then return empty;
fi;
bringe kindknoten in richtige position;
schedule.add(*knoten*);
return *schedule*;

Der Algorithmus, der die Kindknoten in die richtige Stackposition für den Knoten bringt, kann selbstverständlich die Kommutativität eines Operators ausnützen, um unnötiges Kopieren bzw. `swap`-Operationen zu verhindern.

Anstatt den Wert eines „shared node“ immer neu zu berechnen, versucht die Funktion `scheduleshared()` (Programm 5.6), ihn im Stack umzukopieren, sodaß er im weiteren Verlauf verwendet werden kann.

Wird eine Möglichkeit zum Kopieren gefunden, so wird der entsprechende Befehl im schedule an passender Stelle eingefügt (danach hat sich das Aussehen des Stack für die folgenden Befehle natürlich verändert und muß neu berechnet werden). Kann der Knoten nicht kopiert werden, versucht die Funktion noch, ihn von einer lokalen Variable nachzuladen. Allerdings werden dabei nie neue temporäre lokale Variablen extra für diesen Zweck generiert. Falls der Wert des Knotens im ursprünglichen Code nicht in einer Variablen gespeichert wurde, wird ein „leeres“ schedule zurückgegeben und damit der aufrufenden Funktion signalisiert, daß das schedule der Eingabe nicht sinnvoll ist.

Nach diesem groben Überblick über die Funktionalität des optimalen DAG-scheduling sollen noch einige Feinheiten zur Sprache kommen, bevor Überlegungen zu den beiden konkreten Architekturen den Abschnitt beschließen. Wie immer beim Übergang von der Theorie in die Praxis ergeben sich einige Komplikationen bzw. Punkte, die zu beachten sind, damit der Einsatz der Techniken sinnvoll erfolgen kann – auch wenn manche dieser Überlegungen trivial erscheinen mögen.

Programm 5.6 Scheduling eines shared node – `scheduleshared()`

comment: Eingabe: *knoten:* ein Knoten („shared“)
schedule: das bisherige schedule
Ausgabe: das schedule erweitert um den Knoten

for alle knoten in *schedule* rückwärts **do**
stack ← stack nach ausführung des *knoten*s;
if *knoten* im stack? ∧
knoten kann kopiert werden?
then
füge kopierbefehl in *schedule* ein;
return *schedule*;
fi;
end;

if *knoten* hat wert einer variablen?
then
schedule.add(laden der *variable*);
return *schedule*;
fi;
return empty;

Konstante als „shared nodes“

Es ist für die Suche nach mehrfach vorkommenden Teilausdrücken wichtig, alle äquivalenten Befehle in einem Knoten zu vereinen (dies ermöglichte etwa in Beispiel 5.9 eine weitreichende Optimierung). Auch das Laden von Konstanten auf den Stack sollte daher für gleiche Werte durch einen gemeinsamen Knoten ausgedrückt werden.

Beim scheduling des DAG ist es jedoch meist nicht sinnvoll, diesen Wert auf dem Stack wie Variablen zu behandeln. Während für Variablen auf den Speicher zugegriffen werden muß, ist der Wert der Konstante meist schon im Befehl codiert. Es reicht, den Knoten mehrfach in das schedule einzufügen, das Kopieren von Konstanten in der Funktion `scheduleshared()` kann damit umgangen werden.

Berechnung der Ausführungszeit

Für das „branch and bound“ und natürlich auch zum Vergleich der Lösungen ist ein Maß für die Qualität eines schedule vonnöten. In diesem Zusammenhang wird beim DAG scheduling die (geschätzte) Ausführungszeit der Befehle verwendet.

Da die Zeit nur zum Vergleich der Lösungen untereinander verwendet wird und eigentlich nur das Laden bzw. Speichern von lokalen

Variablen unterdrückt werden soll, nimmt der Algorithmus folgende Zeiteinheiten für die Ausführung einer Instruktion an:

- Der Zugriff auf lokale Variablen dauert 3 Zyklen.
- Alle weiteren Befehle dauern 1 Zyklus.

Wichtig im Verhältnis dieser Zahlen ist, daß die Dauer für einen Variablenzugriff nicht kürzer ist als die Dauer für zwei Stackmanipulationsbefehle. Weil das Laden einer Variablen meist durch zwei Stackmanipulationen (einmal kopieren, einmal an die oberste Position bringen) eliminiert wird, sind die in dieser Arbeit behandelten Optimierungstechniken nur bei Architekturen mit einem entsprechenden Zahlenverhältnis sinnvoll.

Abschätzung der Ausführungszeit eines DAG

Schon bevor die erste Lösung gefunden wurde, ist für das „branch and bound“ eine Grenze der Ausführungszeit nötig. Dazu dient eine Abschätzung T, die mittels folgender Formel aus den Knoten V des DAG errechnet wird:

$$T = 2 \sum_{v \in V} t(v)u(v) + 3l(v)$$

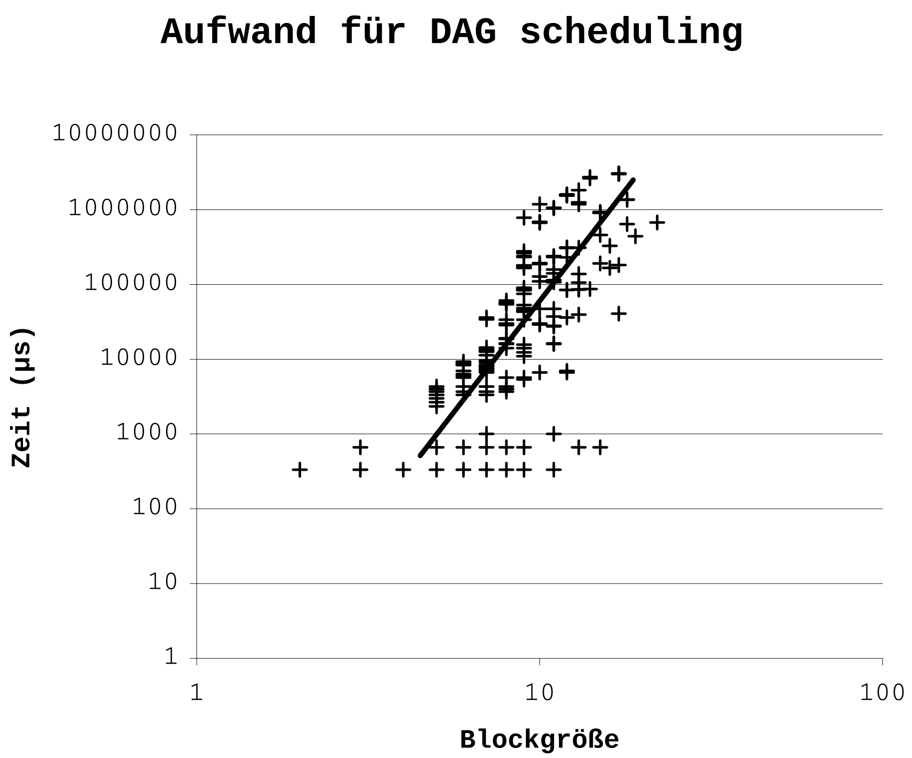
Hierbei steht $t(v)$ für die Ausführungszeit der Befehle, die dem Knoten entsprechen, $u(v)$ ist die Anzahl der Elternknoten (oder 1, wenn der Knoten keinen solchen besitzt) und $l(v)$ ist die Anzahl der lokalen Variablen, denen der Wert des Knotens zugewiesen wird (und somit die Anzahl der Speicheroperationen, die durchgeführt werden müssen).

5.3.2 Aufwand

Weiter oben wurde schon festgestellt, daß die Größe des Suchraumes für einen Graphen mit n Knoten maximal $n!$ ist. Da in der Implementierung des DAG scheduling jedoch Techniken verwendet werden, die den Suchraum verkleinern, erwarten wir realistischerweise einen exponentiellen Aufwand des DAG scheduling.

Abbildung 5.1 stellt wie Abbildung 4.1 eine Reihe von Meßwerten für das DAG scheduling der JGL-Klassen (siehe Abschnitt 6.1) dar. Auf der X-Achse ist logarithmisch die Größe der basic blocks aufgetragen, die Y-Achse gibt – ebenfalls logarithmisch – die vom DAG scheduling benötigte Zeit an. Fälle, in denen das DAG scheduling nach fünf Sekunden abgebrochen werden mußte, weil die Suche noch nicht beendet war, sind in dem Diagramm nicht berücksichtigt. Wie erwartet steigt der Aufwand annähernd exponentiell mit der Größe der basic blocks, wie schon in Abbildung 4.1 ist allerdings die Streuung sehr hoch.

Abbildung 5.1 Aufwand für DAG scheduling



5.3.3 HYSTAMIN

Bei dieser Architektur gibt es einige Konstellationen, welche Abhängigkeiten zwischen Befehlen erzeugen, die durch Abhängigkeitskanten im DAG symbolisiert werden müssen:

- Lese- und Schreibzugriffe direkt auf den Hauptspeicher müssen serialisiert werden. Lesebefehle dürfen nicht über Schreibbefehle hinweg verschoben werden, ansonsten bestünde die Gefahr, daß der zu lesende Wert überschrieben wird. Auch Schreibbefehle dürfen ihre Reihenfolge untereinander nicht vertauschen, Lesebefehle hingegen sehr wohl.
- Manipulationen des R-Stack dürfen ihre Reihenfolge nicht untereinander ändern. Da diese Befehle eher selten auftreten, ist dies keine große Einschränkung.
- Funktionsaufrufe müssen in der ursprünglichen Reihenfolge erfolgen, außerdem dürfen Speicherzugriffe und Manipulationen des R-Stack nicht über Funktionsaufrufe hinweg verschoben werden.

Lokale Variablen, auf die über einen Zeiger zugegriffen werden kann, oder die als `volatile` deklariert wurden, kommen natürlich nicht in Betracht, wenn es darum geht, Knoten im DAG wiederzuverwenden. Für Zugriffe auf diese Variablen muß immer ein neuer Knoten angelegt werden, weil sich ihr Wert „unvorhersagbar“ ändern kann.

5.3.4 JavaVM

Wegen der Architektur der JavaVM kommt es zu einigen Problemen mit dem einfachen Algorithmus, wie er oben beschrieben wurde. Für folgende Befehle erweist es sich als nötig, Abhängigkeitskanten einzuführen:

- Der Befehl `iinc` verändert den Inhalt einer lokalen Variable ohne daß diese zuerst auf den Stack geladen werden müßte. Deshalb ist es nötig, weitere Ladebefehle für die Variable nicht durch bereits im DAG vorhandene Knoten zu ersetzen; außerdem müssen solche Ladebefehle nach dem `iinc` ausgeführt werden. Auch Schreibbefehle auf die Variable müssen ihre ursprüngliche Position relativ zum `iinc` behalten.
- Schreibbefehle auf Arrayelemente bilden Barrieren für Lesebefehle für das Array. Ansonsten könnte ein noch zu lesendes Arrayelement mit einem neuen Wert überschrieben werden. Außerdem müssen die Schreibbefehle in ihrer ursprünglichen Ordnung erhalten bleiben.
- Ähnliches gilt für Zugriffe auf member-Variablen von Klassen.
- Methodenaufrufe müssen untereinander in der ursprünglichen Reihenfolge bleiben. Außerdem dürfen Zugriffe auf member-Variablen und Arrays nicht über solche Aufrufe hinweg verschoben werden.
- Monitorbefehle (sie werden dazu benutzt, um kritische Abschnitte – in Java als `synchronized` gekennzeichnet – zu markieren) bilden Barrieren für alle zwischen ihnen liegenden Befehle.

Eine diffizile Situation entsteht beim Anlegen eines neuen Objektes: Hierzu muß erst mit `new` ein Objekt erzeugt und dann der passende Konstruktor aufgerufen werden (der Aufruf unterscheidet sich im Code nicht von einem „normalen“ Methodenaufruf). Da sich die Referenz auf das Objekt schon nach dem Erzeugen mittels `new` auf dem Stack befindet, speichert der Algorithmus für das DAG-scheduling diesen Wert – wenn er im Verlauf der weiteren Ausführung noch gebraucht wird – sofort in einer lokalen Variablen.

Dieses Verhalten wird allerdings vom Java-Interpreter nicht gutgeheißen und eine Runtime-Exception („object not initialized") ist die Folge. Es ist also nötig, einen Konstruktoraufruf erkennen zu können und das Speichern auf eine lokale Variable entsprechend zu unterdrücken.

Ein weiterer heikler Punkt ist die schon in Abschnitt 3.2 erwähnte Tatsache, daß ein basic block in JavaVM-Code nicht unbedingt mit einem leeren Stack beginnen bzw. enden muß. Um dennoch eine gültige Lösung finden zu können, sind folgende Vorkehrungen zu treffen:

- Ist der Stack am Ende des basic block nicht leer, so muß ein spezieller Knoten erzeugt werden, der so viele Kindknoten hat, wie sich Elemente am Stack befinden, aber keine realen Befehle symbolisiert. Von ihm werden Abhängigkeitskanten zu allen anderen „elternlosen" Knoten eingeführt, was sicherstellt, daß sich die richtigen Elemente am Schluß in der korrekten Reihenfolge auf dem Stack befinden.

- Ist der Stack am Beginn des basic block nicht leer, so müssen so viele spezielle Knoten erzeugt werden, wie sich Elemente auf dem Stack zur Laufzeit befinden würden. Diese Knoten müssen, bevor die rekursive Funktion `schedule()` aufgerufen wird, schon in der richtigen Reihenfolge im schedule zu liegen kommen. So kann garantiert werden, daß auch zur Laufzeit auf die richtigen Elemente zugegriffen wird.

Schließlich bereitet auch hier die Verwendung von je zwei Stackeinträgen für Werte vom Typ `long` bzw. `double` einige Probleme; es stimmen etwa die Anzahl der Elemente, die vom Stack genommen werden, nicht mit der Anzahl der Kindknoten überein.

Kapitel 6

Ergebnisse

Errors using inadequate data are much less than those using no data at all.

Charles Babbage

In diesem Kapitel wenden wir uns den Ergebnissen zu, die mit stack scheduling und optimalem DAG scheduling erreicht werden können. Von besonderem Interesse ist hier der Vergleich der beiden Ergebnisse, um feststellen zu können, wie nahe das Resultat von Koopmans Algorithmus dem Optimum kommt.

Nach einer Auflistung der für die Implementierung verwendeten Werkzeuge sowie der Eingaben, mit denen die Optimierungen überprüft wurden, werden die Resultate für die zwei untersuchten Architekturen in separaten Abschnitten vorgestellt und erläutert. Anschließend findet sich eine Darstellung der verwendeten Stackmanipulationsbefehle bzw. ihrer Auftrittshäufigkeit vor und nach der Optimierung.

Am Schluß dieses Kapitels wenden wir uns der Auswirkung der Optimierungen auf die Stacktiefe zu. Dies ist nicht ganz unerheblich, da die Stacktiefe zum Beispiel sowohl die Größe als auch den Algorithmus für das stack buffering (siehe Abschnitt 2.3) in realen Stack-Prozessoren beeinflußt.

6.1 Verwendete Werkzeuge und Testfälle

Dieser Abschnitt enthält einen Überblick über die für die Implementierung verwendeten Werkzeuge sowie die Eingaben, mit denen die Optimierung getestet wurde. Außerdem sind – soweit bekannt – Bezugsquellen für die entsprechenden Programme im Internet angegeben.

Beide Optimierer wurden in C++ geschrieben und mit dem GNU C++ Compiler g++[1] in der Version 2.7.2 übersetzt. Als Betriebssystem kam Linux 2.0.27[2] auf einem PC zum Einsatz. Für die HYSTAMIN-Optimierung wurde weiters das SUIF Compiler Toolkit[3] in der Version 1.0.2 verwendet; der SUIF-Zwischencode wird in Maschinencode für HYSTAMIN übertragen und dieser dann optimiert.

Die Auswertung der Optimierungen für die JavaVM wurde mit den Werkzeugen des Java Development Kit (JDK) 1.0.2 für Linux[4] sowie dem „just-in-time" Java-Compiler kaffe 0.60[5] durchgeführt.

Die Optimierung für HYSTAMIN wurde mit folgenden C-Sourcen getestet, wobei einige Adaptionen am Quellcode nötig waren (unter anderem, weil HYSTAMIN keine Fließkommaberechnungen unterstützt):

- compress 4.0 ist ein Standard-UNIX Programm zur Datenkomprimierung und enthält einige mathematisch aufwendige Routinen.
- Der Quellcode für Linux 0.01,[6] aus einer sehr frühen Phase der Entwicklung von Linux, ist ein Beispiel für systemnahe „low level" Programme.
- Der Stanford Benchmark 4.1[7] enthält einige „Standard"-Algorithmen als Testfall. Unter anderem finden sich hier Bubble- und Quicksort, das Königinnenproblem, die Türme von Hanoi und eine Fast Fourier Transformation.

Folgende Java-Programme kamen als Testfall für die Optimierung zum Einsatz:

- IceKey[8] ist eine Klasse, die ICE-Verschlüsselung durchführt, und damit ein Beispiel für rechenintensive Aufgaben.
- Die Java Generic Library (JGL)[9] in der Version 1.1 ist eine Bibliothek mit Algorithmen und Datenstrukturen nicht unähnlich der Standard Template Library (STL) für C++.
- Die Java-Klassen aus dem JDK[10] enthalten die Standardbibliothek für Java und umfassen etwa 140000 Bytecode-Befehle. Unter anderem ist auch der Java Compiler `javac` darin enthalten.

[1] ftp://prep.ai.mit.edu/pub/gnu/
[2] http://www.linux.org/
[3] http://suif.stanford.edu/
[4] http://www.blackdown.org/java-linux.html
[5] http://www.tjwassoc.demon.co.uk/kaffe/kaffe.htm
[6] ftp://sunsite.unc.edu/pub/Linux/distributions/slackware/source/kernel-source/Historic/linux-0.01.tgz
[7] ftp://ftp.complang.tuwien.ac.at/pub/forth/Stanford-benchmarks.tar.gz
[8] http://munkora.cs.mu.oz.au/~mkwan/ice/
[9] http://www.objectspace.com/jgl/
[10] http://www.javasoft.com/

- Eine Gruppe von Wissenschaftern an der University of Washington sammelten für ein Forschungsprojekt[11] eine Menge von Java-Benchmarks. Darin finden sich unter anderem Algorithmen wie die „Türme von Hanoi“, Berechnungen der Mandelbrot-Menge, eine Implementierung der DES-Verschlüsselung, ein Java-Compiler und eine Menge von „microbenchmarks“.

6.2 Ergebnisse

Dieser Abschnitt enthält ein Darstellung der mit den verwendeten Methoden erreichbaren Ergebnisse. Zur Veranschaulichung wird dazu der Code in folgende Befehlsklassen eingeteilt:

Zugriff auf lokale Variablen Dazu zählen das Laden und Speichern von lokalen Variablen.

Stackmanipulation In diese Kategorie fallen alle Befehle, die keine Berechnungen durchführen, sondern die Stackelemente bzw. deren Reihenfolge manipulieren.

Andere Befehle Schließlich muß es natürlich noch Befehle geben, die „nützliche“ Berechnungen durchführen, Sprünge ausführen usw.

Da die Testfälle zu sehr ähnlichen Ergebnissen führen, wird im folgenden nur je ein repräsentatives Beispiel für HYSTAMIN und die JavaVM angeführt.

6.2.1 HYSTAMIN

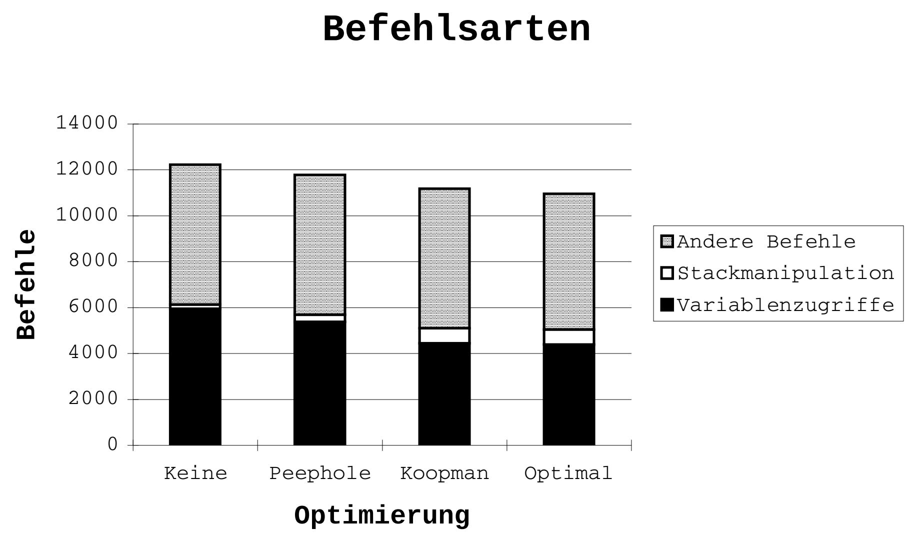
Abbildung 6.1 enthält eine Darstellung der Befehlsklassen für den ursprünglichen und optimierten Code des Stanford Benchmark. Daraus wird ersichtlich, daß die Ergebnisse für das optimale DAG scheduling nur unwesentlich besser sind als die des stack scheduling. Da erstere Optimierung äußerst aufwendig ist (etwa 30 Minuten für das Beispiel – zum Vergleich: die Optimierung via stack scheduling dauert nur etwa zehn Sekunden), entsteht ein krasses Mißverhältnis zwischen Aufwand und Ergebnis bei den beiden Optimierungsmethoden.

Ein interessantes Detail bietet der Vergleich von Ergebnissen für das DAG scheduling mit bzw. ohne Berücksichtigung der Kommutativität von Operatoren. In zweiterem Fall bleibt die Anzahl der Zugriffe auf lokale Variablen beinahe gleich wie bei ersterem, allerdings wird ein

[11] `http://www.cs.washington.edu/research/interpreters/`

leichter Anstieg der Stackmanipulationsbefehle gegenüber einer Optimierung unter Berücksichtigung von Kommutativität sichtbar. Offensichtlich können einige `swap`-Operationen vor kommutativen Operatoren eliminiert werden.

Abbildung 6.1 Optimierung des Stanford Benchmark



Die Anzahl der Variablenzugriffe im Code des Stanford Benchmark sinkt von 2125 im nicht optimierten Code auf 1590 im Code, der durch stack scheduling optimiert wurde, und weiter auf 1567 nach dem DAG scheduling. Das bedeutet eine Reduktion der Zugriffe auf etwa 75% bzw. 74% des ursprünglichen Wertes. Eine Optimierung mittels peephole optimizer ohne stack- oder DAG scheduling drückt die Zahl der Variablenzugriffe auf 1921 oder ca. 90% des unoptimierten Wertes.

In der Abbildung ist die Tatsache berücksichtigt, daß ein Variablenzugriff drei oder vier Maschinenbefehle benötigt und der dadurch erzeugte Anteil an den gesamten Befehlen daher höher ist, als aus den absoluten Zahlen der Zugriffe zu schließen wäre.

Tabelle 6.1 Ergebnisse der Optimierung des Stanford Benchmark

Kl.	unoptimiert		Peephole		Koopman		Optimal	
Vz.	5950	54,3%	5378	49,0%	4452	40,6%	4387	40,0%
Sm.	192	1,8%	316	2,9%	654	6,0%	662	6,1%
A.	6083	55,5%	6080	55,4%	6079	55,4%	5918	53,9%
Σ	12225	111,6%	11774	107,3%	11185	102,0%	10967	100,0%

Die Anzahl der Stackbefehle verschiebt sich von 192 oder 1,5% im unoptimierten Code auf 654 bzw. 6,2% nach der Optimierung mit stack scheduling und 662 (das sind 6,1%) nach dem DAG scheduling. Das bedeutet eine signifikante Änderung der Befehlszusammensetzung. Daß die absolute Befehlsanzahl sinkt, ist der Tatsache zu verdanken, daß ein Variablenzugriff zumindest drei Befehle benötigt und bei der Optimierung durch maximal zwei Stackmanipulationsbefehle ersetzt wird. Tabelle 6.1 faßt die Zahlen noch einmal zusammen und stellt die Anteile der Befehlsklassen relativ zur Anzahl der Befehle nach der Optimierung mit DAG scheduling dar. Die erste Spalte gibt die Befehlsklasse für die jeweilige Zeile an: „Vz.“ steht für Variablenzugriffe, „Sm.“ für Stackmanipulationen und „A.“ für andere Befehle.

Abbildung 6.2 Optimierung des Stanford Benchmark

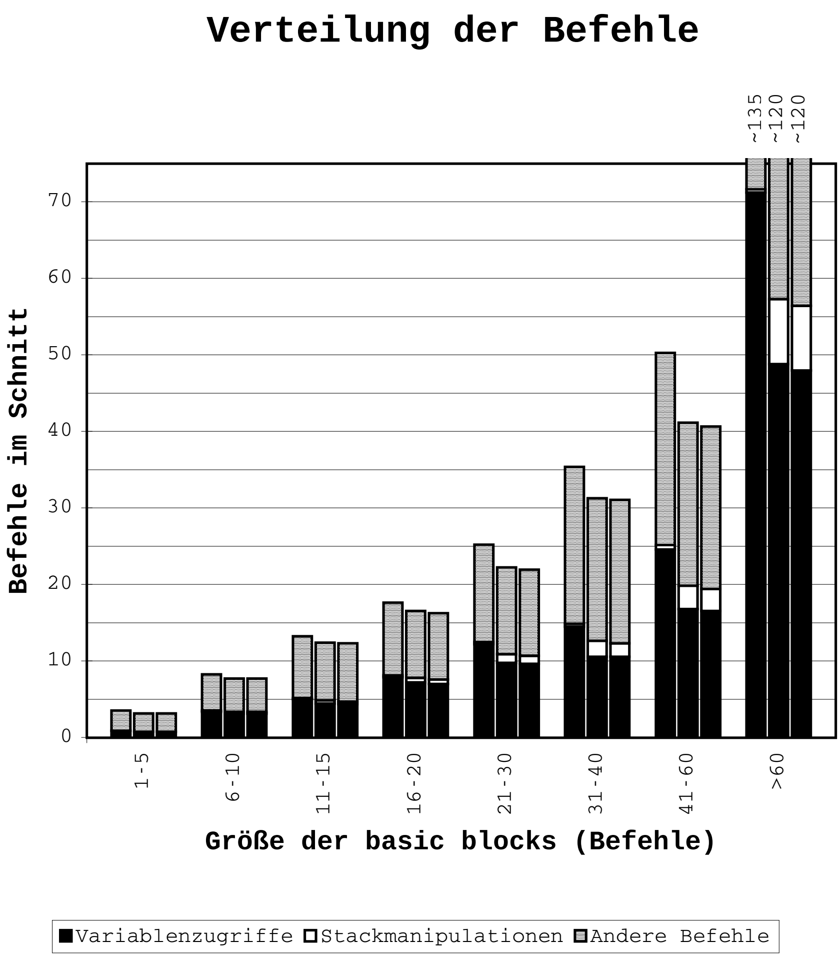


Abbildung 6.2 enthält die eben erläuterten Daten aus einem anderen Blickwinkel gesehen. Hier wird das Ergebnis der Optimierung in Relation zur Größe des optimierten basic blocks dargestellt. Auf der X-Achse sind dabei jeweils drei Säulen für einen Größenbereich von basic blocks aufgetragen. Von links nach rechts sind die Daten ohne Optimierung, nach der Optimierung mit stack scheduling und nach der Optimierung mit DAG scheduling dargestellt. Die Größe der Säulen gibt

die durchschnittliche Anzahl von Befehlen der verschiedenen Klassen für basic blocks der entsprechenden Länge an.

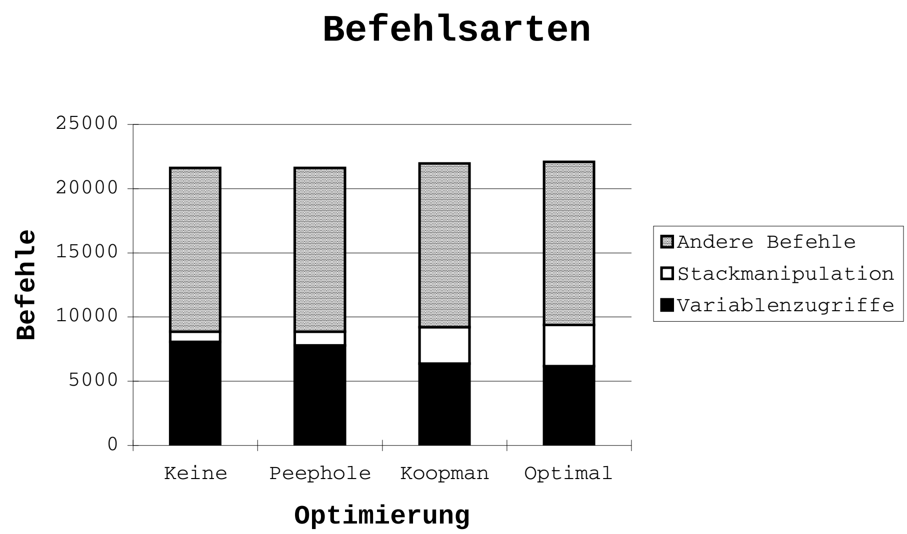
Wie aus der Abbildung ersichtlich, ist das Ergebnis der Optimierung von der Größe der basic blocks relativ unabhängig; auch gibt es keine signifikanten Unterschiede zwischen den beiden Optimierungsmethoden.

Die Laufzeit des erzeugten Codes konnte mangels einer Umgebung, in der die aufgerufenen Bibliotheksfunktionen zur Verfügung stehen, leider nicht ermittelt werden. Allerdings darf für die Annahme eines Hardware-unterstützten Stackbuffers von einer Verringerung der Ausführungszeit in dieser Architektur ausgegangen werden, die sich aufgrund der Ergebnisse jedoch bei beiden Optimierungen auf gleichem Niveau befindet.

6.2.2 JavaVM

Im Grunde ähnlich, doch nicht exakt gleich stellt sich die Situation für JavaVM-Code dar. Wie Abbildung 6.3 zu entnehmen ist, verkleinert sich die Zahl der Zugriffe auf lokale Variablen im Code für die Klassen der JGL bei der Optimierung mit Koopmans stack scheduling von 8063 oder 37% auf 6383 bzw. 29% der gesamten Befehle; DAG scheduling verringert den Wert auf 6175 oder 28%. Demnach verbleiben nach dem DAG scheduling 77% der ursprünglichen Zugriffsbefehle, während das stack scheduling deren 79% übrigließ. Der Peephole optimizer alleine bringt es auf 7792 Variablenzugriffe bzw. 96% des ursprünglichen Wertes.

Abbildung 6.3 Optimierung der JGL-Klassen



Der Anteil der Stackmanipulationsbefehle steigt von ca. 4% im ursprünglichen Code auf 13% im Code nach der Optimierung mittels stack scheduling und knapp 15% beim DAG scheduling. In absoluten Zahlen bedeutet das einen Anstieg von 795 auf 2818 bzw. 3219 Befehle. Im Gegensatz zu HYSTAMIN steigt die Gesamtzahl der Befehle nach der Optimierung leicht an, weil hier ein Variablenzugriff nur einen Befehl benötigt; die Eliminierung ersetzt ihn dann im Durchschnitt durch etwa 1,3 Stackmanipulationsbefehle. Tabelle 6.2 enthält das Ergebnis in der gleichen Anordnung wie Tabelle 6.1 im vorigen Abschnitt.

Tabelle 6.2 Ergebnisse der Optimierung der JGL-Klassen

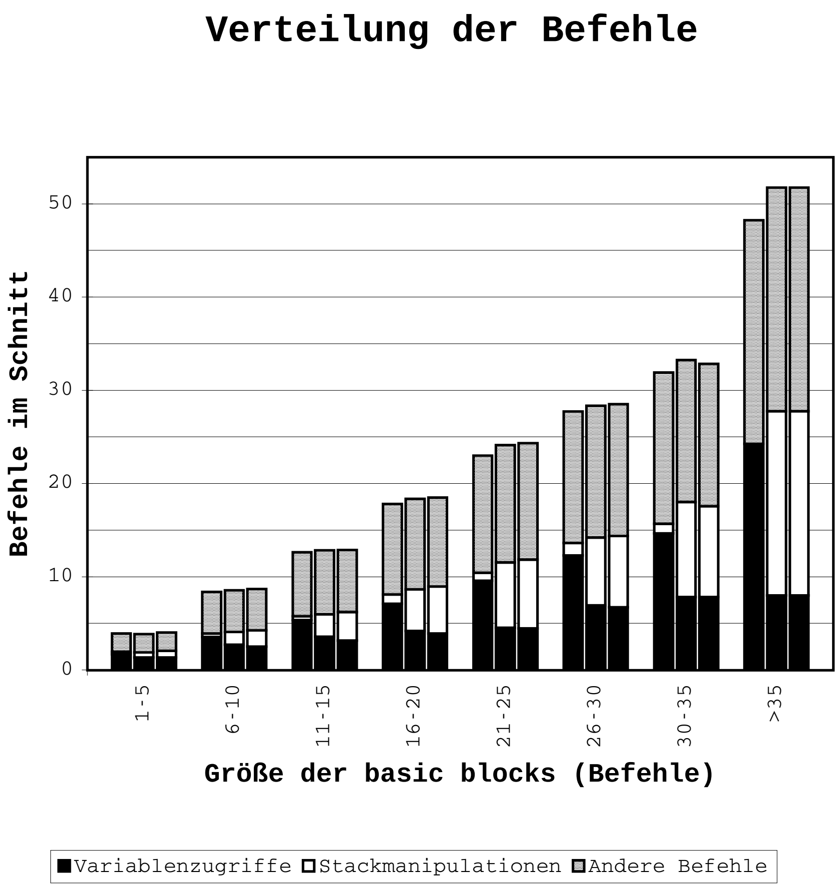
Kl.	unoptimiert		Peephole		Koopman		Optimal	
Vz.	8063	36,5%	7792	35,3%	6383	28,9%	6175	28,0%
Sm.	793	3,6%	1072	4,8%	2818	12,8%	3219	14,6%
A.	12761	57,8%	12761	57,8%	12761	57,8%	12703	57,4%
Σ	21617	97,9%	21625	97,9%	21962	99,5%	22097	100,0%

Wie schon im vorigen Abschnitt soll auch hier die Optimierung in Relation zur Größe der basic blocks bewertet werden. Abbildung 6.4 ist zu entnehmen, daß die Effizienz der Optimierung mit der Größe der basic blocks ansteigt. In Kapitel drei wurde allerdings dargestellt, daß große basic blocks in JavaVM-Code selten sind; die Hälfte der basic blocks besitzt maximal fünf Befehle – insgesamt ist daher ein eher geringer Effekt der Optimierung festzustellen. Die Effizienz der beiden Methoden gleicht sich, es ist jedoch zu bedenken, daß mit der Größe und Komplexität eines basic blocks auch die Wahrscheinlichkeit steigt, daß das DAG scheduling zu keinem Ergebnis führt und daher stack scheduling einspringen muß.

Der Umstand, daß die Gesamtzahl der Befehle durch die Optimierung steigt, bringt auch ein enttäuschendes Ergebnis für die Ausführungszeit des optimierten Codes mit sich. Die Laufzeit ist sowohl in einem Interpreter als auch einem „just-in-time" Compiler gleich wie die des ursprünglichen, nicht optimierten Codes, tendenziell sogar etwas schlechter. Erklärbar ist dieses Verhalten dadurch, daß die softwaremäßige Emulation der JavaVM sowohl für den Stack als auch für den Bereich der lokalen Variablen den Hauptspeicher verwendet, während Stack-CPUs üblicherweise einige Elemente des Stack in einem schnelleren Stackbuffer zwischenspeichern können, was bessere Ergebnisse auf Stack-Hardware erwarten läßt.

Das Laden einer der ersten vier Variablen benötigt in einem Interpreter für die JavaVM nur eine Leseoperation auf dem Bereich der lokalen Variablen sowie eine Speicheroperation auf dem Bereich des Stack (bei Variablen mit einem größeren Index muß zusätzlich dieser Index aus dem Code gelesen werden). Ein einfaches `dup` benötigt ebenso je eine Lese- und Schreiboperationen, komplexere Befehle wie `dup_x2` müssen

Abbildung 6.4 Optimierung der JGL-Klassen



einen Teil des gesamten Stack umkrempeln und in diesem Bereich dementsprechend viele Lese- und Schreiboperationen durchführen.

Ein realer Geschwindigkeitsvorteil für optimierten Code dürfte sich daher erst für JavaVM-CPUs ergeben, die einen schnelleren Zugriff auf den Stack bieten und durch einen stack buffer die Zahl von Zugriffen auf den Hauptspeicher (als potentiellen Flaschenhals) verringern.

6.3 Verwendete Stackmanipulationen

Besonders für die Hardware-Implementierung von Stackprozessoren sind Daten über die Verwendung von Stackmanipulationsbefehlen von Interesse. Besonders häufig genutzte Befehle sind wichtige Kandidaten für eine Optimierung der Ausführungsgeschwindigkeit. In diesem Abschnitt wird deshalb die Auswirkung der Optimierungstechniken auf die Verwendung von Stackmanipulationsbefehlen untersucht.

Abbildung 6.5 Stackmanipulationsbefehle im Stanford Benchmark

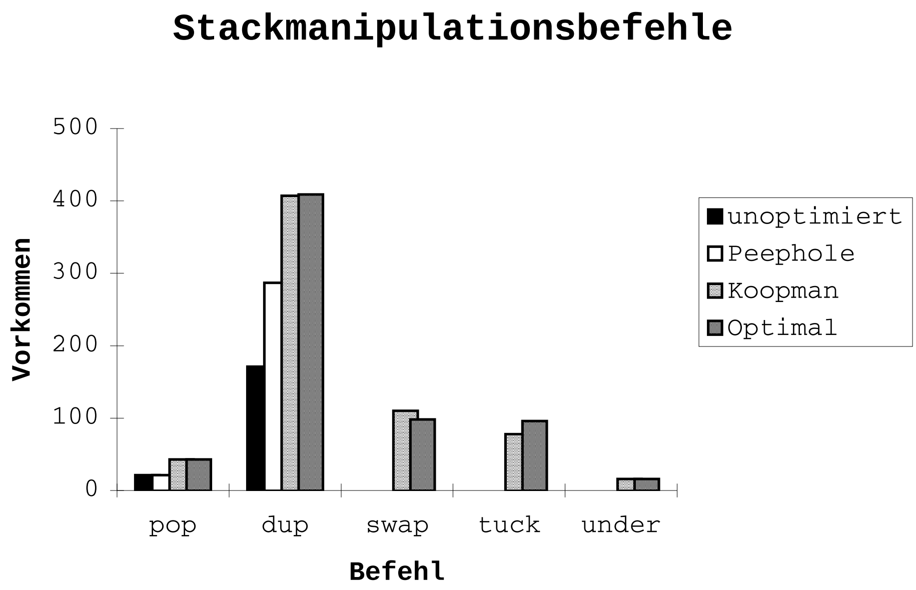


Abbildung 6.5 stellt die Anzahl der Stackmanipulationsbefehle im Code des Stanford Benchmark dar. Die Häufigkeit im unoptimierten Code ist dabei der nach den verschiedenen Optimierungen gegenübergestellt. Generell ist zu vermerken, daß die Optimierung alle Arten von Stackmanipulationen häufiger werden läßt, wobei die Befehle `swap`, `tuck` und `under` in unoptimiertem Code gar nicht erst vorkommen. Der Befehl `dup` ist der bei weitem häufigste Stackmanipulationsbefehl, mit einigem Respektabstand folgen `swap` und `tuck`. Tabelle 6.3 faßt die Daten zusammen.

Tabelle 6.3 Stackmanipulationsbefehle im Stanford Benchmark

Befehl	unoptimiert	Peephole	Koopman	Optimal
`pop`	21	21	43	43
`dup`	171	295	407	409
`swap`	0	0	110	98
`tuck`	0	0	78	96
`under`	0	0	16	16
Summe	192	316	654	662

Ein ähnliches Aussehen zeigen die Daten für den JavaVM-Code der JGL-Klassen (Abbildung 6.6 und Tabelle 6.4). Wieder ist `dup` für den Löwenanteil der Stackmanipulationsbefehle verantwortlich, abgeschlagen landen `swap` und `dup_x1` (entspricht einem `tuck`) auf den Plätzen. Befehle, die zwei Stackelemente auf einmal manipulieren (`dup2` u.ä.), treten nicht auf, obwohl die Optimierung durchaus Gebrauch davon machen würde – offensichtlich kommen solche Situationen relativ selten vor.

Abbildung 6.6 Stackmanipulationsbefehle in den Klassen der JGL

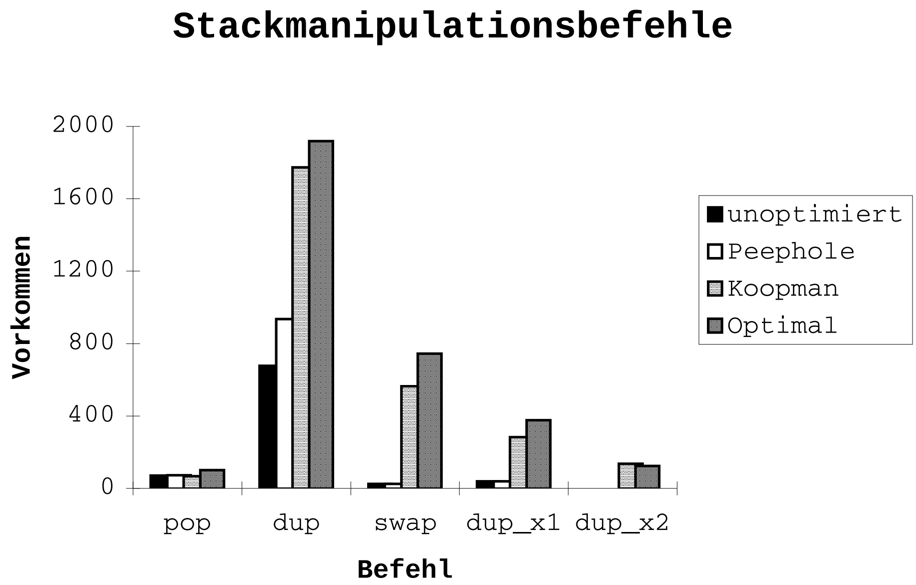


Tabelle 6.4 Stackmanipulationsbefehle in den Klassen der JGL

Befehl	unoptimiert	Peephole	Koopman	Optimal
`pop`	71	73	67	89
`dup`	678	935	1772	1897
`swap`	25	25	562	739
`dup_x1`	39	39	281	372
`dup_x2`	0	0	136	122
Summe	793	1072	2818	3219

6.4 Auswirkungen auf die Stacktiefe

In diesem Abschnitt wird eine Auswirkung der besprochenen Optimierungen angesprochen, die für reale Stackmaschinen durchaus von Bedeutung ist. Durch die Optimierung verändert sich die durchschnittliche Tiefe des Stack während der Ausführung des Codes. Die Ergebnisse in diesem Abschnitt stammen aus Veröffentlichungen von Christopher Bailey und wurden nicht für die Optimierung mit der vorliegenden Implementierung ermittelt.

Aus der Beschreibung der Optimierung in Kapitel vier und fünf läßt sich sofort ein Schluß ziehen, der für die Praxis von Relevanz ist:

Die durchschnittliche Anzahl der Elemente auf dem Stack erhöht sich nach den Optimierungen. Diese Aussage kann trivialerweise aus der Beobachtung abgeleitet werden, daß der Sinn der Optimierung ja gerade darin liegt, lokale Variablen auf dem Stack und nicht im Speicher zu halten.

Es stellt sich die Frage nach den Auswirkungen dieser Tatsache auf Stackmaschinen. Christopher Bailey hat dazu in einer Reihe von Artikeln ([Bai94], [Bai95a], [Bai95b]) folgende Beobachtungen gemacht:

- Der Großteil der Hauptspeicherzugriffe der von ihm untersuchten Stackarchitektur (UTSA, „University of Teesside Stack Architecture") geht auf das Konto von data stack spilling[12] (46%) und instruction fetch (24%). Der Zugriff auf lokale Variable ist nur für etwa 24% der Speicherzugriffe verantwortlich und der direkte Zugriff auf den Speicher trägt nur 5% zum gesamten Aufkommen bei. Diese Daten wurden für repräsentativen C-Code ohne Einsatz eines Stack-Cache ermittelt.

- Stack scheduling läßt die Häufigkeit von größeren Stacktiefen teilweise signifikant ansteigen. In [Bai94] zeigt sich für ausgewählte Beispiele, daß die Wahrscheinlichkeit für Stacktiefen zwischen 2 und 6 (und vor allem 3) stark sinkt und dafür Tiefen zwischen 7 und 12 häufiger werden.

- Die Verwendung von Optimierungstechniken wie stack scheduling läßt die Effizienz von Stackbuffern sinken. [Bai95b] gibt als Beispiel bei Verwendung einer „demand fed" Bufferstrategie an, daß die Buffergröße um etwa 30% erhöht werden muß, um die gleiche „buffer spill rate" wie vor dem scheduling zu erreichen.

- Natürlich beeinflußt die Menge der vorhandenen Befehle in der Zielarchitektur die Möglichkeiten des stack scheduling. [Bai95a] enthält Angaben über die Effizienz des Algorithmus für Architekturen, die sich nur in der Anzahl der Stackelemente unterscheiden, auf die mit Stackmanipulationsbefehlen direkt zugegriffen werden kann.

  Kann nur das oberste Stackelement manipuliert werden, sinkt die Anzahl der Variablenzugriffe im Code auf etwas über 90% des unoptimierten Wertes. Beim Zugriff auf zwei Stackelemente (was im großen und ganzen auf die in dieser Arbeit verwendeten Architekturen zutrifft) reduziert sich dieser Wert auf etwa 80%. Kann auf drei bzw. vier Elemente zugegriffen werden, erhalten wir einen Wert von 70% bzw. 65%. Die dynamische Anzahl der Zugriffe (also während der Ausführung des Codes) sinkt hingegen nicht so

[12] Data stack spilling wird durch Über- bzw. Unterlauf des stack buffers verursacht. Dabei werden Stackelemente zwischen dem Hauptspeicher und dem stack buffer verschoben.

stark und liegt für die Architektur mit Zugriffsmöglichkeit auf vier Elemente immerhin noch bei etwa 75% des ursprünglichen Wertes.

Als Zusammenfassung aus den Untersuchungen von Christopher Bailey kann festgehalten werden, daß die Optimierung von Zugriffen auf lokale Variablen Konsequenzen nach sich zieht, die den erzielbaren Geschwindigkeitsgewinn zum Teil wieder zunichte machen können. Um es mit [Bai94] zu sagen:

> „Scheduling is not the clear cut issue that we may have assumed when we began this study, but a complex one with many implications for machine design and subtle trade-offs for performance.“

Kapitel 7

Zusammenfassung und Ausblick

My interest is in the future because I am going to spend the rest of my life there.

Charles F. Kettering

Abschließend ist eine Zusammenfassung der wichtigsten Punkte der Arbeit sowie ein Ausblick auf weitere Möglichkeiten zur Optimierung angebracht. Nachteile der bestehenden Implementierung und Vorschläge für deren Verbesserung sind hier ebenso zu finden wie eine Betrachtung über den Nutzen der Optimierung.

7.1 Zusammenfassung

Ausgehend von der Tatsache, daß lokale Variablen in der Ausführung einer Funktion oder Methode eine zentrale Rolle spielen und eine Optimierung auf diesem Gebiet am vielversprechendsten erscheint, wurden in der vorliegenden Arbeit zwei Methoden untersucht und verglichen, die die Zugriffshäufigkeit auf lokale Variablen in Stackarchitekturen verringern sollen. Für Registermaschinen ist das analoge Problem der Registerallokation weitgehend gelöst, das Konzept einer Stackarchitektur zieht jedoch einige Konsequenzen nach sich, die eine einfache Lösung bislang unmöglich machen.

Zwei unterschiedliche Stackarchitekturen wurden in dieser Arbeit zur empirischen Bewertung der entwickelten Optimierungsstrategien verwendet. Die beiden Architekturen unterscheiden sich sowohl in ihrem Befehlssatz als auch in anderen Punkten (Anzahl der Hardware-unterstützten Stacks, Größenverteilung der basic blocks) und ermöglichen so einen Überblick über die Effizienz der Methoden für ein breiteres Spektrum der Menge existierender Stackmaschinen.

Die grundlegende Idee hinter den vorgestellten Optimierungstechniken besteht darin, den Wert einer lokalen Variable auf dem Stack zwischenzuspeichern, um nicht jedesmal auf den (langsameren) Hauptspeicher zugreifen zu müssen, wenn eine Variable benötigt wird. Da auf den Stack in einer implizit festgelegten Reihenfolge zugegriffen wird, während Register in der Regel beliebig gelesen und geschrieben werden können, sind effiziente Manipulationen des Stack nötig, um überhaupt eine Verbesserung der Laufzeit erzielen zu können.

Beide in dieser Arbeit behandelten Optimierungsmethoden benötigen einige Vor- und Nachbearbeitungen des zu optimierenden Codes, so muß etwa eine Funktion bzw. Methode in basic blocks aufgespalten und der Datenflußgraph erzeugt werden, weil die Optimierungen auf lokaler Ebene, d.h. auf einzelnen basic blocks arbeiten. Auch die Analyse der Lebenszeit von Variablen und ein peephole-optimizer sind in dieser Hinsicht nützliche Elemente der Optimierung.

Die erste Optimierungstechnik wurde von Phil Koopman entwickelt und nennt sich „stack scheduling“. Ein basic block wird dabei auf Ladebefehle für lokale Variablen untersucht. Für jede dieser Instruktionen sucht der Algorithmus einen davorliegenden zweiten Befehl, vor dessen Ausführung der Wert der später geladenen Variablen auf dem Stack liegt. Die solcherart ermittelten Befehlspaare werden aufsteigend nach der Distanz zwischen den Instruktionen sortiert und dann in dieser Reihenfolge optimiert. Dabei wird versucht, den Wert der Variable vor dem ersten Befehl des Paares an den Grund des Stack zu kopieren. Ist dies möglich, dann kann der Ladebefehl durch einen Stackmanipulationsbefehl ersetzt werden, der die Kopie der Variablen vom Grund des Stack an die oberste Position verschiebt.

Ladebefehle werden bei dieser Methode durch im Mittel 1,3 Stackmanipulationsbefehle ersetzt, Speicherbefehle auf lokale Variablen, die im weiteren Programmverlauf nicht mehr verwendet werden, können eliminiert werden. Die vorliegende Implementierung enthält noch einige kleinere Erweiterungen des ursprünglichen Vorschlages.

Das Resultat des stack scheduling ist durchaus brauchbar: Das Ergebnis kommt dem Optimum recht nahe. Allerdings verhindern praktische Gründe eine Verkleinerung der Laufzeit von optimiertem Code auf einer in Software simulierten JavaVM.

Die zweite Methode der Optimierung, das „optimale DAG scheduling“, beschreitet einen anderen Weg und sucht nach einer Anordnung der Befehle eines basic block, die eine Eliminierung von Zugriffen auf lokale Variable möglichst fördert. Grundlage für diese Technik ist der DAG (ein gerichteter, azyklischer Graph), der die Beziehungen der Befehle eines basic block untereinander symbolisiert und somit sicherstellt, daß durch das Umordnen die Semantik des gesamten Programmes nicht verändert wird.

Um die beste Reihenfolge der Befehle aus dem DAG abzuleiten, wird üblicherweise eine Heuristik verwendet (die nicht garantieren kann, wirklich die beste Lösung zu finden, aber meistens sehr gute Ergebnisse erzielt). In dieser Arbeit kommt jedoch eine erschöpfende Suche nach der optimalen Lösung zur Anwendung. Einige Methoden (wie etwa „branch and bound") werden verwendet, um den Aufwand für die Suche zu verringern, allerdings ist diese noch immer sehr viel komplexer als Koopmans Ansatz.

Knoten, die im DAG mehrfach referenziert werden – der von ihnen repräsentierte Wert wird zur Laufzeit öfters verwendet –, können mit einer dem stack scheduling ähnlichen Methode behandelt werden, um ein Zwischenspeichern ihres Wertes in Variablen zu vermeiden. Auch hier werden Speicherbefehle auf „dead variables" gänzlich unterdrückt.

Im Vergleich zu Koopmans stack scheduling schneidet das optimale DAG scheduling zwar etwas besser ab, allerdings entsteht dabei ein krasses Mißverhältnis zwischen Aufwand und Ergebnis. Die Verbesserung gegenüber dem Ergebnis von Koopmans Methode bleibt meist marginal und auch ohne Auswirkungen auf die Laufzeit des JavaVM-Codes, die Laufzeit des DAG scheduling ist jedoch um Größenordnungen höher als die von Koopmans Algorithmus.

7.2 Ausblick

Die derzeitige Implementierung besitzt noch einige Nachteile bzw. verbesserungswürdige Punkte, deren Korrektur eine bessere Qualität der Ergebnisse nach ziehen würde:

- Die Optimierung für HYSTAMIN benützt in der jetzigen Implementierung nur den Datenstack für das Zwischenspeichern der Variablen; für kurze Abschnitte ist allerdings auch der Returnstack für eine solche Aufgabe gut geeignet und die Optimierung könnte durch die Nutzung dieses Potentials einiges an Effizienz gewinnen.

- Code für die JavaVM-Architektur verwendet neben lokalen Variablen auch member-Variablen. Der Zugriff darauf könnte wie bei lokalen Variablen durch Kopien der Werte auf dem Stack optimiert werden. Allerdings ist dabei zu beachten, daß der Wert auf dem Stack wieder in die Variable zurückgeschrieben werden muß, bevor andere Methoden aufgerufen werden, und daß sich der Wert während der Aufrufe ändern kann. Dieses Verhalten entspricht dem von globalen Variablen in C-Code (die Optimierung für HYSTAMIN berücksichtigt globale Variablen beim stack scheduling, allerdings nur in basic blocks ohne Funktionsaufrufe).

- Das DAG scheduling unterstützt Operatoren mit mehr als zwei Operanden nur sehr rudimentär. Die Operanden müssen sich schon in der richtigen Reihenfolge auf dem Stack befinden, ansonsten wird das schedule als ungültig markiert.

- Eine gute Heuristik für das DAG scheduling könnte den Aufwand für diese Methode drastisch reduzieren und ihren Einsatz auch für andere als Vergleichszwecke ermöglichen. In der jetzigen Form ist der Algorithmus für „reale“ Anwendungen jedenfalls nicht geeignet, da die Laufzeit einfach zu groß ist. Da aber Koopmans stack scheduling dem optimalen Ergebnis des DAG scheduling meist recht nahe kommt und noch dazu schnell ist (auch hier ist die vorliegende Implementierung nicht vorrangig auf Effizienz ausgelegt), stellt sich die Frage, ob es nötig und sinnvoll ist, eine solche Heuristik zu erarbeiten.

- Genauere Annahmen über die Ausführungszeit der Befehle bzw. Untersuchungen für mehrere Alternativen könnten nähere Aufschlüsse über ein Architektur geben, für die mit den beschriebenen Optimierungen noch bessere Ergebnisse erzielt werden können.

Letztlich scheinen die Möglichkeiten der lokalen Optimierung durch das stack scheduling gut ausgeschöpft zu werden, eine weitere Steigerung der Qualität ist wohl nur mit globalen Methoden zu erreichen. Für die Schwierigkeiten, mit denen dabei zu rechnen ist, spricht auch die Tatsache, daß Phil Koopman in seiner Beschreibung in [Koo92] nur sehr vage auf globales scheduling eingeht, dafür aber von einer signifikanten Verbesserung der Ergebnisse berichtet.

Natürlich stellt sich nicht zuletzt die Frage nach der Sinnhaftigkeit der Optimierung für JavaVM-Code. Die Ergebnisse haben gezeigt, daß es in der Laufzeit eines Interpreters keinen Unterschied zwischen optimiertem und ursprünglichen Code gibt. Messungen auf echter Stackhardware sind nötig, um den realen Wert der Optimierung feststellen zu können. Für eine Java-CPU werden mit großer Wahrscheinlichkeit andere Ergebnisse bei der Laufzeit auftreten, wenn der Speicherzugriff langsamer ist als Stackmanipulationen im (hoffentlich vorhandenen) stack buffer.

Anhang A

Statistik zur Implementierung

It's not enough to be busy. The question is: What are we busy about?

Henry David Thoreau

Eine Statistik zu den verschiedenen Teilen der Implementierung soll in diesem Anhang die Komplexität der Methoden veranschaulichen. Der Code des Optimierers für HYSTAMIN besteht aus 9811 Zeilen C++, die sich auf folgende Bereiche aufteilen:

Tabelle A.1 Code der Optimierung für HYSTAMIN

Bereich	Zeilen	Anteil
Assemblerausgabe	289	3%
Live-Variable Analyse	519	5%
Datenflußanalyse	664	7%
Peephole optimizer	796	8%
Allgemeine Datenstrukturen	1021	10%
Koopmans stack scheduling	1047	11%
Befehlsselektion	1696	17%
Allgemeines	1759	18%
Optimales DAG scheduling	2020	21%
Summe	9811	100%

In der Analyse der Lebenszeit von Variablen ist auch ein generischer pass zum Lösen von Datenflußgleichungen enthalten. Der Bereich der Datenflußanalyse beinhaltet das Auffinden der basic blocks sowie die Generierung und das Entfernen des Datenflußgraphen. Zu den Datenstrukturen gehören unter anderem template-Klassen für Listen, Bitvektoren und Stacks. Die Befehlsselektion ist für die Übertragung von

SUIF-Zwischencode in HYSTAMIN-Code verantwortlich, und unter „Allgemeines“ fallen die Klassen für Befehle, Funktionen sowie die Unterstützung für das Debugging und die Symboltabellenverwaltung.

Ein recht ähnliches Bild bietet der Optimierer für JavaVM-Code, dessen 9614 Zeilen C++ sich folgendermaßen aufteilen:

Tabelle A.2 Code der Optimierung für die JavaVM

Bereich	Zeilen	Anteil
Live-Variable Analyse	456	5%
Koopmans stack scheduling	624	6%
Peephole optimizer	653	7%
Ein- und Ausgabe	701	7%
Datenflußanalyse	824	9%
Allgemeine Datenstrukturen	1314	14%
Optimales DAG scheduling	1974	20%
Allgemeines	3068	32%
Summe	9614	100%

Im Bereich „Ein- und Ausgabe“ befinden sich Klassen, die einen portablen Zugriff auf Dateien im „big endian“-Format erlauben; auch das Einlesen und Schreiben des „classfiles“ wird hier behandelt. „Allgemeines“ ist unter anderem deshalb so groß, weil sich darin neben einer umfangreichen Klasse für Bytecode-Befehle eine Tabelle mit allen JavaVM-Instruktionen samt wichtigen Informationen über sie findet.

Literaturverzeichnis

[Aho86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, Reading, 1986.

[Bai94] Christopher Bailey, R. Sotudeh, M. Ould-Khaoua. The effects of Local Variable Optimisation on a C-based Stack Processor Environment. *Proceedings of the 1994 Euroforth Conference*, Winchester, November 1994.

[Bai95a] Christopher Bailey, R. Sotudeh. The Effects of Intra-Block Scheduling in a Stack Processor Environment. *Proceedings of the 1995 Rochester Forth Conference*, Rochester, June 1995.

[Bai95b] Christopher Bailey, R. Sotudeh. Trade-offs for Memory Bandwith Reduction In Stack-Processor Design. *Proceedings of the 10^{th} International Conference on Mathematical and Computer Modelling*, Boston, July 1995.

[Bar61] Robert S. Barton. A new approach to the functional design of a computer. *Proceedings of the Western Joint Computer Conference*, 1961.

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

[Bru75] J. L. Bruno, T. Lassagne. The Generation of Optimal Code for Stack Machines. *Journal of the ACM*, Vol. 22, No. 3, July 1975.

[Bul77] David M. Bulman. Stack Computers. *IEEE Computer*, May 1977.

[Cha81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, Peter W. Markstein. Register Allocation Via Coloring. *Computer Languages*, Vol. 6, No. 1, 1981.

[Chr86] Nicos Christofides. *Graph Theory: An Algorithmic Approach*. Academic Press, New York, 1986.

[Dun77] Fraser G. Duncan. Stack Machine Development: Australia, Great Britain and Europe. *IEEE Computer*, May 1977.

[Ert95a] M. Anton Ertl. Stack Caching for Interpreters. *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, La Jolla, June 1995.

[Ert95b] M. Anton Ertl, Martin Maierhofer. Translating Forth to Efficient C. *Proceedings of the 1995 Euroforth Conference*, Schloß Dagstuhl, October 1995.

[Fan95] Xiaoming Fan, Holger Heitsch, Tomasz Malitka, Bernd Rosenthal, Klaus Schleisiek (ed). Three Instruction Set Structures for a Stack Processor. *Proceedings of the 1995 Euroforth Conference*, Schloß Dagstuhl, October 1995.

[Has85] Makoto Hasekawa, Yoshiharu Shigei. High-speed top-of-stack scheme for interpreters: A management algorithm and its analysis. *International Symposium on Computer Architecture*, 1981.

[Hay87] John R. Hayes, Martin E. Fraeman, Robert L. Williams, Thomas Zaremba. An Architecture for the Direct Execution of the Forth Programming Language. *Proceedings of the 2^{nd} International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, October 1987.

[Hen90] John L. Hennessy, David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, San Mateo, 1990.

[Kes96] Christoph W. Keßler. *Scheduling Expression DAGs for Minimal Register Need*,[1] Universität Trier, 1996

[Koo89] Philip Koopman. *Stack Computers: The New Wave*. Ellis Horwood, Chichester, 1989.

[Koo92] Philip Koopman. A Preliminary Exploration of Optimized Stack Code Generation. *Proceedings of the 1992 Rochester Forth Conference*, Rochester, June 1992.

[Koo93] Philip Koopman. Usenet Nuggets: Why Stack Machines? *Computer Architecture News*, Vol. 21, No. 1, March 1993.

[Mas80] Larry M. Masinter, L. Peter Deutsch. Local Optimization in a Compiler for Stack-based Lisp Machines. *Conference Record of the 1980 LISP Conference*, Stanford, 1980. Reprint New York, 1985.

[1] `ftp://ftp.informatik.uni-trier.de/pub/reports/96-12.ps`

[Mil87] Daniel L. Miller. Stack Machines and Compiler Design. *Byte*, April 1987.

[Pra80] Bhaskaram Prabhala, Ravi Sethi. Efficient Computation of Expressions with Common Subexpressions. *Journal of the ACM*, Vol. 27, No. 1, January 1980.

[Sed88] Robert Sedgewick. *Algorithms*, 2^{nd} ed. Addison-Wesley, Reading, 1988.

[Smo91] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind, David Hunnicutt. Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling. *Proceedings of the 24^{th} Annual International Symposium on Microarchitecture*, Albuquerque, November 1991.

[Sta96] Stanford Compiler Group. *The SUIF Library*,[2] Stanford University, 1996.

[Sun94] Sun Microsystems Computer Company. *Writing FCode Programs*. SunSoft Press, 1994.

[Sun95] Sun Microsystems Computer Company. *The Java Virtual Machine Specification*,[3] 1995.

[Tan82] Andrew S. Tanenbaum, Hans van Staveren, Johan W. Stevenson. Using Peephole Optimization on Intermediate Code. *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 1, January 1982.

[Win88] A. Winfield, S. Kelly. A C-to-Forth Translator. *EuroFORML'88 Conference Proceedings*, Southampton, September 1988.

[Woe92] Jack Woehr. *Forth: The New Model*. Prentice-Hall, Englewood Cliffs, 1992.

[2] `http://suif.stanford.edu/suif/docs/suif_toc.html`
[3] `http://www.javasoft.com/doc/language_vm_specification.html`